Microsoft DirectX 8.1 (C++)

Programmers Guide

[This is preliminary documentation and is subject to change.]

This guide contains a description of the graphics pipeline implemented by Microsoft® Direct3D®. It is a guide for developers who are implementing three-dimensional (3-D) graphics functionality into their applications. The guide contains architecture descriptions, functional block diagrams, and descriptions of the building blocks in the pipeline, as well as code snippets and sample applications. The information is divided into the following sections:

- Getting Started with Direct3D

  This section contains both an overview of the pipeline and tutorials that can help you get a simple graphics application running in a few minutes.

- Using Direct3D

  This section explains how to use the fixed function pipeline. Included here are the basic functional steps in the graphics pipeline: converting geometry, adding lighting, and rendering output.

- Programmable Pipeline

  This section covers the new programmable extensions to the pipeline. Included here are details about using vertex shaders for manipulating object geometry, pixel shaders for controlling pixel shading, and effects and effects files for building applications that can run on a variety of hardware platforms.

- Advanced Topics

  This section contains examples of different types of special effects you can implement. Topics such as environment and bump mapping, antialiasing, vertex blending, and tweening show how to apply leading-edge special effects to your application.

- Samples

  This section contains sample applications.

- Direct3D Appendix

  This section contains details on additional topics, such as X Files and graphics state.

For more information about specific API methods, see the Reference pages.

Microsoft DirectX 8.1 (C++)

Getting Started with Direct3D

[This is preliminary documentation and is subject to change.]

This section provides a brief introduction to the three-dimensional (3-D) graphics functionality in the Microsoft® Direct3D® application programmer interface (API). Here you will find an overview of the *graphics pipeline* and tutorials to help you get basic Direct3D functionality up and running quickly.

- Direct3D Architecture
- DirectX Graphics C/C++ Tutorials

Microsoft DirectX 8.1 (C++)

Direct3D Architecture

[This is preliminary documentation and is subject to change.]

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

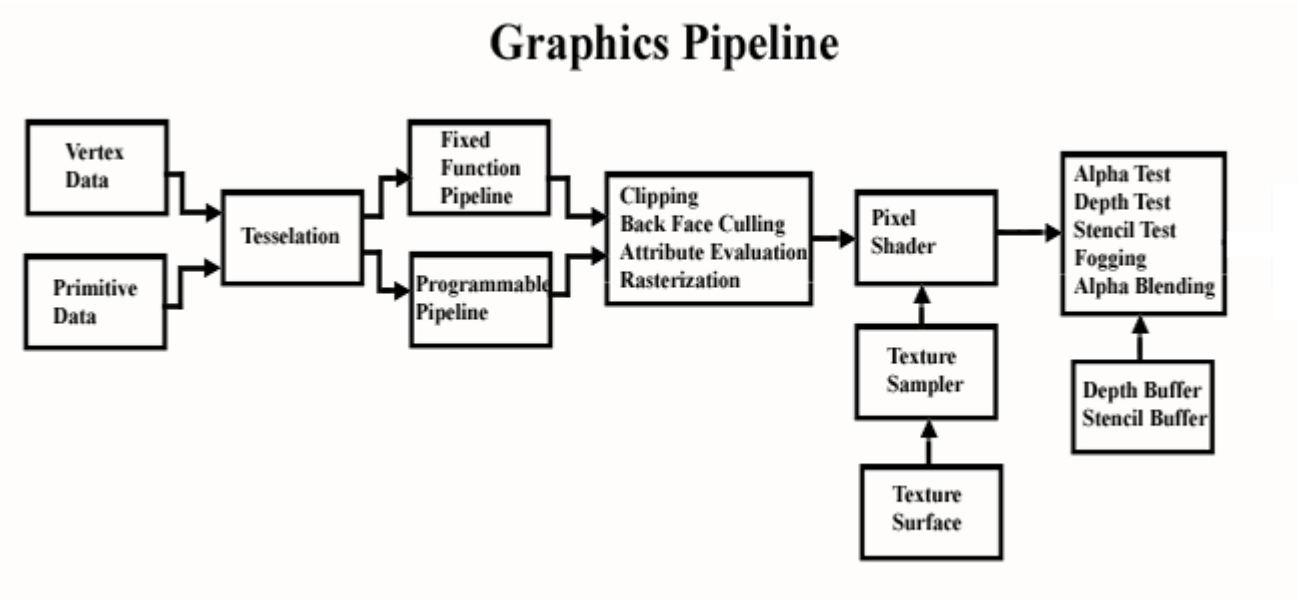
- Architectural Overview for Direct3D
- Hardware Abstraction Layer
- System Integration
- Programmable Vertex Shader Architecture
- Programmable Pixel Shader Architecture

Microsoft DirectX 8.1 (C++)

Architectural Overview for Direct3D

[This is preliminary documentation and is subject to change.]

This is an illustration of the graphics pipeline. The functionality in each of the blocks is introduced below and contains links to more information.



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

Microsoft DirectX 8.1 (C++)

Hardware Abstraction Layer

[This is preliminary documentation and is subject to change.]

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

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

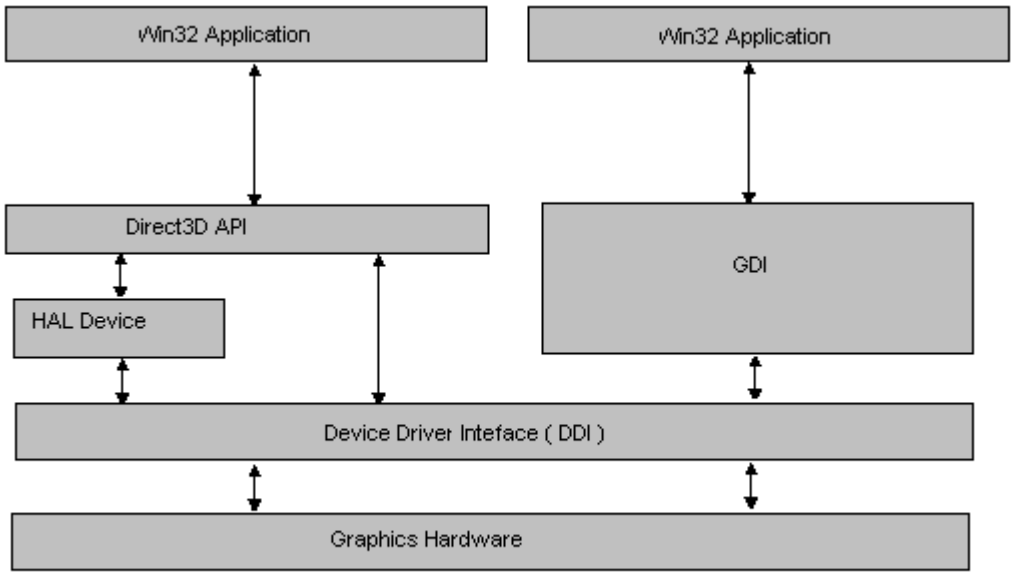
In DirectX 8.0, the HAL can have three different vertex processing modes: software vertex processing, hardware vertex processing, and mixed vertex processing on the same device. The pure device mode is a variant of the HAL device. The pure device type supports hardware vertex processing only, and allows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

Microsoft DirectX 8.1 (C++)

System Integration

[This is preliminary documentation and is subject to change.]

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



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

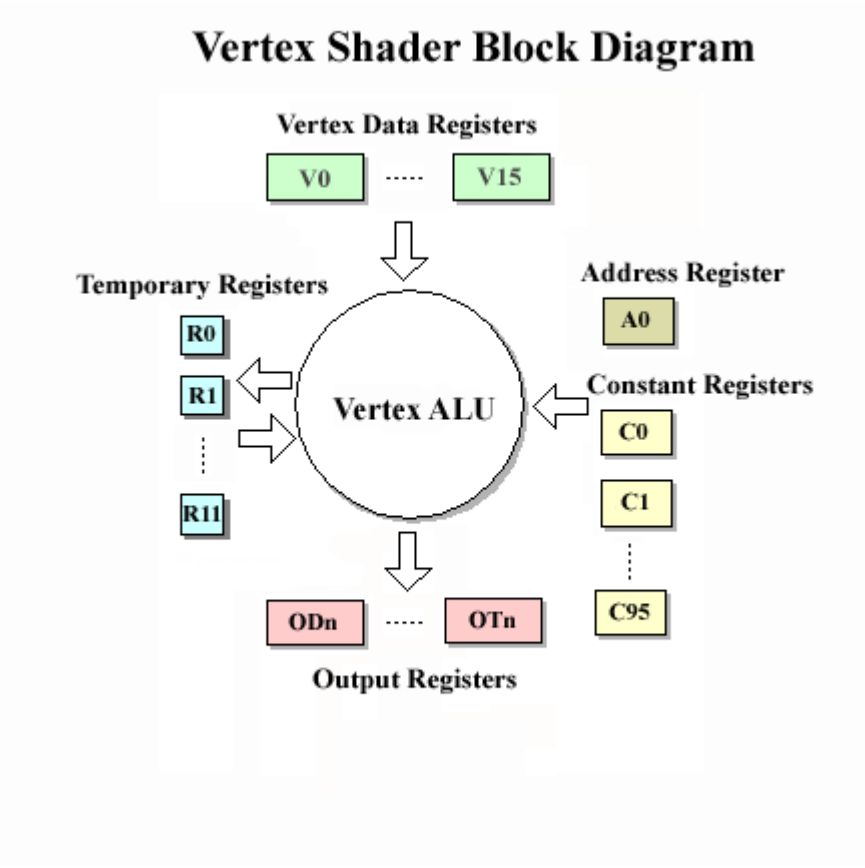
For more information on devices supported by Direct3D, see Device Types.

Microsoft DirectX 8.1 (**shader versions 1.0 and 1.1**)

Programmable Vertex Shader Architecture

[This is preliminary documentation and is subject to change.]

The following diagram illustrates the vertex shader architecture. The vertex shader architecture streams vertex data into the shader from the graphics pipeline, performs operations on the data using an arithmetic logic unit (ALU), and outputs the transformed vertex data to the graphics pipeline for further processing.



Registers are used for inputting data, outputting data and to hold temporary result. Each register holds four fixed point numbers. The vertex shader architecture defines four types of registers, each operating on a different type of data.

- V0 - V15 - vertex registers. These registers are used to stream vertex data into the shader ALU. Vertex data is defined by an application and can contain any data.
- R0 - R11 - temporary registers. Temporary registers are used to hold intermediate results.
- A0 - address register. Version 1.1 supports one address register, version 1.0 does not have one. An address register, can be used to provide an integer offset during the lookup of any register.
- Const0 - Const95 - constant value registers. These are designed to input constant values to the shader.
- ODn, OFog, OPos, OPts, OTn - output registers. These registers output data from the vertex shader to the graphics pipeline for further processing. The registers contain transformed position data, texture coordinates, diffuse color and specular color.

For more information about vertex shader registers see Registers.

For more information about the vertex shader instruction set, see the Vertex Shader Reference.

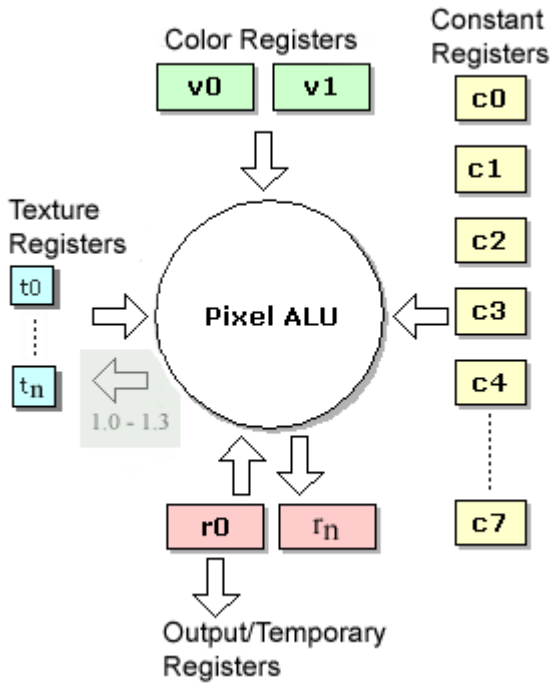
Microsoft DirectX 8.1 (C++)

Programmable Pixel Shader Architecture

[This is preliminary documentation and is subject to change.]

The following diagram illustrates the pixel shader architecture. The pixel shader architecture uses pixel data from a series of input registers, performs operations on the data using an arithmetic logic

unit (ALU), and outputs the transformed pixel data to the graphics pipeline for further processing.



ALU data input and output is done with four types of registers:

- **Color registers** v0 - v1. These registers stream vertex shader color data to a pixel shader. The vertex shader data can come from the fixed function pipeline or a programmable vertex shader.
- **Constant registers** c0 - cn. These registers are used to provide constants to the shader.
- **Texture registers** t0 - t_n. For versions 1.0 to 1.3, texture registers are used to provide texture data and/or texture coordinate data depending the settings of the texture stages, as well as on the instruction being executed. For these versions, the texture registers can also be used as temporary registers to store intermediate results. This is represented by the read arrow from the ALU in the diagram. For version 1.4, the texture registers can be called texture coordinate registers because they contain texture coordinate data only. Texture coordinate registers in version 1.4 cannot be used for intermediate results because they cannot read from the ALU.
- **Output/Temporary registers** r0 - r_n. These registers provide temporary storage for intermediate calculations. The r0 register also provides the output of the pixel shader.

The pixel shader ALU provides arithmetic instructions for transforming pixel data. It also has a series of texture instructions that can be used to sample textures or perturb texture coordinates.

For more information about pixel shader registers, see Registers.

For more information about the pixel shader instruction set, see the Pixel Shader Reference.

Microsoft DirectX 8.1 (C++)

DirectX Graphics C/C++ Tutorials

[This is preliminary documentation and is subject to change.]

The tutorials in this section show how to use Microsoft® Direct3D® and Direct3DX in a C/C++ application for common tasks. The tasks are divided into required steps. In some cases, steps are organized into substeps for clarity.

The following tutorials are provided.

- Tutorial 1: Creating a Device
- Tutorial 2: Rendering Vertices
- Tutorial 3: Using Matrices
- Tutorial 4: Creating and Using Lights
- Tutorial 5: Using Texture Maps
- Tutorial 6: Using Meshes

Note The sample code in these tutorials is from source projects whose location is provided in each tutorial.

The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.

Some comments in the included sample code might differ from the source files in the Microsoft Platform Software Development Kit (SDK). Changes are made for brevity only and are limited to comments to avoid changing the behavior of the sample code.

See Also

DirectX Graphics C/C++ Samples

Microsoft DirectX 8.1 (C++)

Tutorial 1: Creating a Device

[This is preliminary documentation and is subject to change.]

To use Microsoft® Direct3D®, you first create an application window, then you create and initialize Direct3D objects. You use the COM interfaces that these objects implement to manipulate them and to create other objects required to render a scene. The CreateDevice sample project on which this tutorial is based illustrates these tasks by creating a Direct3D device and rendering a blue screen.

This tutorial uses the following steps to initialize Direct3D, render a scene, and eventually shut down.

- Step 1: Creating a Window
- Step 2: Initializing Direct3D
- Step 3: Handling System Messages
- Step 4: Rendering and Displaying a Scene
- Step 5: Shutting Down

Note The path of the CreateDevice sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut01_CreateDevice.

Microsoft DirectX 8.1 (C++)

Step 1: Creating a Window

[This is preliminary documentation and is subject to change.]

The first thing any Microsoft® Windows® application must do when it is executed is create an application window to display to the user. To do this, the CreateDevice sample project begins execution at its **WinMain** function. The following sample code performs window initialization.

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    // Register the window class.
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                      GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                      "D3D Tutorial", NULL };
    RegisterClassEx( &wc );

    // Create the application's window.
    HWND hWnd = CreateWindow( "D3D Tutorial", "D3D Tutorial 01: CreateDevice",
                              WS_OVERLAPPEDWINDOW, 100, 100, 300, 300,
                              GetDesktopWindow(), NULL, wc.hInstance, NULL );
```

The preceding code sample is standard Windows programming. The sample starts by defining and registering a window class called "D3D Tutorial." After the class is registered, the sample code creates a basic top-level window that uses the registered class, with a client area of 300 pixels wide by 300 pixels tall. This window has no menu or child windows. The sample uses the WS_OVERLAPPEDWINDOW window style to create a window that includes Minimize, Maximize, and Close buttons common to windowed applications. (If the sample were to run in full-screen mode, the preferred window style is WS_EX_TOPMOST, which specifies that the created window should be placed above all non-topmost windows and should stay above them, even when the window is deactivated). Once the window is created, the code sample calls standard Microsoft Win32® functions to display and update the window.

With the application window ready, you can begin setting up the essential Microsoft Direct3D® objects, as described in Step 2: Initializing Direct3D.

Microsoft DirectX 8.1 (C++)

Step 2: Initializing Direct3D

[This is preliminary documentation and is subject to change.]

The CreateDevice sample project performs Microsoft® Direct3D® initialization in the **InitD3D** application-defined function called from **WinMain** after the window is created. After you create an application window, you are ready to initialize the Direct3D object that you will use to render the scene. This process includes creating a Direct3D object, setting the presentation parameters, and finally creating the Direct3D device.

After creating a Direct3D object, you can use the **IDirect3D8::CreateDevice** method to create a Direct3D device. You can also use the Direct3D object to enumerate devices, types, modes, and so on. The code fragment below creates a Direct3D object with the **Direct3DCreate8** function.

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

The only parameter passed to **Direct3DCreate8** should always be D3D_SDK_VERSION. This informs Direct3D that the correct header files are being used. This value is incremented whenever a header or other change would require applications to be rebuilt. If the version does not match, **Direct3DCreate8** will fail.

The next step is to retrieve the current display mode by using the **IDirect3D8::GetAdapterDisplayMode** method as shown in the code fragment below.

```
D3DDISPLAYMODE d3ddm;
if( FAILED( g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
    return E_FAIL;
```

The **Format** member of the **D3DDISPLAYMODE** structure will be used when creating the Direct3D device. To run in windowed mode, the **Format** member is used to create a back buffer that matches the adapter's current mode.

By filling in the fields of the **D3DPRESENT_PARAMETERS** you can specify how you want your 3-D application to behave. The CreateDevice sample project sets its **Windowed** member to TRUE, its **SwapEffect** member to D3DSWAPEFFECT_DISCARD, and its **BackBufferFormat** member to **d3ddm.Format**.

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;
```

The final step is to use the **IDirect3D8::CreateDevice** method to create the Direct3D device, as illustrated in the following code example.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &g_pd3dDevice ) ) )
```

The preceding code sample creates the device with the default adapter by using the D3DADAPTER_DEFAULT flag. In most cases, the system will have only a single adapter, unless it has multiple graphics hardware cards installed. Indicate that you prefer a hardware device over a software device by specifying D3DDEVTYPE_HAL for the *DeviceType* parameter. This code sample uses D3DCREATE_SOFTWARE_VERTEXPROCESSING to tell the system to use software vertex processing. Note that if you tell the system to use hardware vertex processing by specifying D3DCREATE_HARDWARE_VERTEXPROCESSING, you will see a significant performance gain on video cards that support hardware vertex processing.

Now that the Direct3D has been initialized, the next step is to ensure that you have a mechanism to process system messages, as described in Step 3: Handling System Messages.

Microsoft DirectX 8.1 (C++)

Step 3: Handling System Messages

[This is preliminary documentation and is subject to change.]

After you have created the application window and initialized Microsoft® Direct3D®, you are ready to render the scene. In most cases, Microsoft Windows® applications monitor system messages in their message loop, and they render frames whenever no messages are in the queue. However, the CreateDevice sample project waits until a WM_PAINT message is in the queue, telling the application that it needs to redraw all or part of its window.

```
// The message loop.
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Each time the loop runs, **DispatchMessage** calls **MsgProc**, which handles messages in the queue. When WM_PAINT is queued, the application calls **Render**, the application-defined function that will redraw the window. Then the Microsoft Win32® function **ValidateRect** is called to validate the entire client area.

The sample code for the message-handling function is shown below.

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            PostQuitMessage( 0 );
            return 0;

        case WM_PAINT:
            Render();
            ValidateRect( hWnd, NULL );
            return 0;
    }

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

Now that the application handles system messages, the next step is to render the display, as described in Step 4: Rendering and Displaying a Scene.

Microsoft DirectX 8.1 (C++)

Step 4: Rendering and Displaying a Scene

[This is preliminary documentation and is subject to change.]

To render and display the scene, the sample code in this step clears the back buffer to a blue color, transfers the contents of the back buffer to the front buffer, and presents the front buffer to the screen.

To clear a scene, call the **IDirect3DDevice8::Clear** method.

```
// Clear the back buffer to a blue color
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
```

The first two parameters accepted by **Clear** inform Microsoft® Direct3D® of the size and address of the array of rectangles to be cleared. The array of rectangles describes the areas on the render target surface to be cleared.

In most cases, you use a single rectangle that covers the entire rendering target. You do this by setting the first parameter to 0 and the second parameter to NULL. The third parameter determines the method's behavior. You can specify a flag to clear a render-target surface, an associated depth buffer, the stencil buffer, or any combination of the three. This tutorial does not use a depth buffer, so D3DCLEAR_TARGET is the only flag used. The last three parameters are set to reflect clearing values for the render target, depth buffer, and stencil buffer. The CreateDevice sample project sets the clear color for the render target surface to blue (D3DCOLOR_XRGB(0,0,255)). The final two parameters are ignored by the **Clear** method because the corresponding flags are not present.

After clearing the viewport, the CreateDevice sample project informs Direct3D that rendering will begin, then it signals that rendering is complete, as shown in the following code fragment.

```
// Begin the scene.
g_pd3dDevice->BeginScene();

// Rendering of scene objects happens here.

// End the scene.
g_pd3dDevice->EndScene();
```

The **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene** methods signal to the system when rendering is beginning or is complete. You can call rendering methods only between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene, you display it by using the **IDirect3DDevice8::Present** method.

```
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

The first two parameters accepted by **Present** are a source rectangle and destination rectangle. The sample code in this step presents the entire back buffer to the front buffer by setting these two parameters to NULL. The third parameter sets the destination window for this presentation. Because this parameter is set to NULL, the **hWndDeviceWindow** member of **D3DPRESENT_PARAMETERS** is used. The fourth parameter is the *DirtyRegion* parameter and in most cases should be set to NULL.

The final step for this tutorial is shutting down the application, as described in Step 5: Shutting Down

Microsoft DirectX 8.1 (C++)

Step 5: Shutting Down

[This is preliminary documentation and is subject to change.]

At some point during execution, your application must shut down. Shutting down a Microsoft® DirectX® application not only means that you destroy the application window, but you also deallocate any DirectX objects your application uses, and you invalidate the pointers to them. The CreateDevice sample project calls **Cleanup**, an application-defined function to handle this when it receives a WM_DESTROY message.

```
VOID Cleanup()
{
    if( g_pd3dDevice != NULL)
        g_pd3dDevice->Release();
    if( g_pD3D != NULL)
        g_pD3D->Release();
}
```

The preceding function deallocates the DirectX objects it uses by calling the **IUnknown::Release** methods for each object. Because this tutorial follows COM rules, the reference count for most objects should become zero and should be automatically removed from memory.

In addition to shutdown, there are times during normal execution—such as when the user changes the desktop resolution or color depth—when you might need to destroy and re-create the Microsoft Direct3D® objects in use. Therefore it is a good idea to keep your application's cleanup code in one place, which can be called when the need arises.

This tutorial has shown you how to create a device. Tutorial 2: Rendering Vertices shows you how to use vertices to draw geometric shapes.

Microsoft DirectX 8.1 (C++)

Tutorial 2: Rendering Vertices

[This is preliminary documentation and is subject to change.]

Applications written in Microsoft® Direct3D® use vertices to draw geometric shapes. Each three-dimensional (3-D) scene includes one or more of these geometric shapes. The Vertices sample project creates the simplest shape, a triangle, and renders it to the display.

This tutorial shows how to use vertices to create a triangle with the following steps:

- Step 1: Defining a Custom Vertex Type
- Step 2: Setting Up the Vertex Buffer
- Step 3: Rendering the Display

Note The path of the Vertices sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut02_Vertices.

The sample code in the Vertices project is nearly identical to the sample code in the CreateDevice project. The Rendering Vertices tutorial focuses only on the code unique to vertices and does not cover initializing Direct3D, handling Microsoft® Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

Microsoft DirectX 8.1 (C++)

Step 1: Defining a Custom Vertex Type

[This is preliminary documentation and is subject to change.]

The Vertices sample project renders a 2-D triangle by using three vertices. This introduces the concept of the vertex buffer, which is a Microsoft® Direct3D® object that is used to store and render vertices. Vertices can be defined in many ways by specifying a custom vertex structure and corresponding custom flexible vector format (FVF). The format of the vertices in the Vertices sample project is shown in the following code fragment.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex.
    DWORD color;        // The vertex color.
};
```

The structure above specifies the format of the custom vertex type. The next step is to define the FVF that describes the contents of the vertices in the vertex buffer. The following code fragment defines a FVF that corresponds with the custom vertex type created above.

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

Flexible Vertex Format Flags describe what type of custom vertex is being used. The sample code above uses the D3DFVF_XYZRHW and D3DFVF_DIFFUSE flags, which tell the vertex buffer that the custom vertex type has a transformed point followed by a color component.

Now that the custom vector format and FVF are specified, the next step is to fill the vertex buffer with vertices, as described in Step 2: Setting Up the Vertex Buffer.

Note The vertices in the Vertices sample project are transformed. In other words, they are already in 2-D window coordinates. This means that the point (0,0) is at the top-left corner and the positive x-axis is right and the positive y-axis is down. These vertices are also lit, meaning that they are not using Direct3D lighting but are supplying their own color.

Microsoft DirectX 8.1 (C++)

Step 2: Setting Up the Vertex Buffer

[This is preliminary documentation and is subject to change.]

Now that the custom vertex format is defined, it is time to initialize the vertices. The Vertices sample project does this by calling the application-defined function **InitVB** after creating the required Microsoft® Direct3D® objects. The following code fragment initializes the values for three custom vertices.

```
CUSTOMVERTEX g_Vertices[] =
{
    { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
    {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};
```

The preceding code fragment fills three vertices with the points of a triangle and specifies which color each vertex will emit. The first point is at (150, 50) and emits the color red (0xffff0000). The second point is at (250, 250) and emits the color green (0xff00ff00). The third point is at (50, 250) and emits the color blue-green (0xff00ffff). Each of these points has a depth value of 0.5 and an RHW of 1.0. For more information on this vector format see **Transformed and Lit Vertices**

The next step is to call **IDirect3DDevice8::CreateVertexBuffer** to create a vertex buffer as shown in the following code fragment.

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                              0 /* Usage */, D3DFVF_CUSTOMVERTEX,
                                              D3DPOOL_DEFAULT, &g_pVB ) ) )
    return E_FAIL;
```

The first two parameters of **CreateVertexBuffer** tell Direct3D the desired size and usage for the new vertex buffer. The next two parameters specify the vector format and memory location for the new buffer. The vector format here is D3DFVF_CUSTOMVERTEX, which is the FVF that the sample code specified earlier. The D3DPOOL_DEFAULT flag tells Direct3D to create the vertex buffer in the memory allocation that is most appropriate for this buffer. The final parameter is the address of the vertex buffer to create.

After creating a vertex buffer, it is filled with data from the custom vertices as shown in the following code fragment.

```
VOID* pVertices;
if( FAILED( g_pVB->Lock( 0, sizeof(g_Vertices), (BYTE**)&pVertices, 0 ) ) )
    return E_FAIL;
memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();
```

The vertex buffer is first locked by calling **IDirect3DVertexBuffer8::Lock**. The first parameter is the offset into the vertex data to lock, in bytes. The second parameter is the size of the vertex data to lock, in bytes. The third parameter is the address of a BYTE pointer, filled with a pointer to vertex data. The fourth parameter tells the vertex buffer how to lock the data.

The vertices are then copied into the vertex buffer using **memcpy**. After the vertices are in the vertex buffer, a call is made to **IDirect3DVertexBuffer8::Unlock** to unlock the vertex buffer. This mechanism of locking and unlocking is required because the vertex buffer may be in device memory.

Now that the vertex buffer is filled with the vertices, it is time to render the display, as described in Step 3: Rendering the Display.

Microsoft DirectX 8.1 (C++)

Step 3: Rendering the Display

[This is preliminary documentation and is subject to change.]

Now that the vertex buffer is filled with vertices, it is time to render the display. Rendering the display starts by clearing the back buffer to a blue color and then calling **BeginScene**.

```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0L )
g_pd3dDevice->BeginScene();
```

Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source; in this case, use stream 0. The source of the stream is specified by calling **IDirect3DDevice8::SetStreamSource**.

```
g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to let Direct3D know what vertex shader to use by calling **IDirect3DDevice8::SetVertexShader**. Full, custom vertex shaders are an advanced topic, but in most cases the vertex shader is only the FVF code. This lets Direct3D know what types of vertices it is dealing with. The following code fragment sets the vertex shader.

```
g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
```

The only parameter for **SetVertexShader** is a handle to the vertex shader to use. The value for this parameter can be a handle returned by **IDirect3DDevice8::CreateVertexShader**, or an FVF code. Here, the FVF code defined by D3DFVF_CUSTOMVERTEX is used.

For more information on vertex shaders, see Vertex Shaders.

The next step is to use **IDirect3DDevice8::DrawPrimitive** to render the vertices in the vertex buffer as shown in the following code fragment.

```
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

The first parameter accepted by **DrawPrimitive** is a flag that tells Direct3D what type of primitives to draw. This sample uses the flag D3DPT_TRIANGLELIST to specify a list of triangles. The second parameter is the index of the first vertex to load. The third parameter tells the number of primitives to draw. Because this sample draws only one triangle, this value is set to 1.

For more information on different kinds of primitives, see 3-D Primitives.

The last steps are to end the scene and then present the back buffer to the front buffer. This is shown in the following code fragment.

```
g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

After the back buffer is presented to the front buffer, the client window shows a triangle with three different colored points.

This tutorial has shown you how to use vertices to render geometric shapes. Tutorial 3: Using Matrices introduces the concept of matrices and how to use them.

Microsoft DirectX 8.1 (C++)

Tutorial 3: Using Matrices

[This is preliminary documentation and is subject to change.]

This tutorial introduces the concept of matrices and shows how to use them. The Vertices sample project rendered 2-D vertices to draw a triangle. However, in this tutorial you will be working with transformations of vertices in 3-D. Matrices are also used to set up cameras and viewports.

Before the Matrices sample project renders geometry, it calls the **SetupMatrices** application-defined function to create and set the matrix transformations that are used to render the 3-D triangle. Typically, three types of transformation are set for a 3-D scene. Steps for creating each one of these typical transformations are listed below.

- Step 1: Defining the World Transformation Matrix
- Step 2: Defining the View Transformation Matrix
- Step 3: Defining the Projection Transformation Matrix

Note The path of the Matrices sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut03_Matrices.

The order in which these transformation matrices are created does not affect the layout of the objects in a scene. However, Direct3D applies the matrices to the scene in the following order: (1) World , (2) View, (3) Projection.

The sample code in the Matrices project is nearly identical to the sample code in the Vertices project. The Using Matrices tutorial focuses only on the code unique to matrices and does not cover initializing Direct3D, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

Microsoft DirectX 8.1 (C++)

Step 1: Defining the World Transformation Matrix

[This is preliminary documentation and is subject to change.]

The world transformation matrix defines how to translate, scale, and rotate the geometry in the 3-D model space.

The following code fragment rotates the triangle on the y-axis and then sets the current world transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matWorld;
D3DXMatrixRotationY( &matWorld, timeGetTime()/150.0f );
g_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
```

The first step is to rotate the triangle around the Y-axis by calling the **D3DXMatrixRotationY** method. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second parameter is the angle of rotation in radians.

The next step is to call **IDirect3DDevice8::SetTransform** to set the world transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the **D3DTS_WORLD** macro to specify that the world transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on world transformations, see **World Transformation**.

After defining the world transformation for the scene, you can prepare the view transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the view transformation matrix is described in Step 2: Defining the View Transformation Matrix.

Microsoft DirectX 8.1 (C++)

Step 2: Defining the View Transformation Matrix

[This is preliminary documentation and is subject to change.]

The view transformation matrix defines the position and rotation of the view. The view matrix is the camera for the scene.

The following code fragment creates the view transformation matrix and then sets the current view transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matView;
D3DXMatrixLookAtLH( &matView, &D3DXVECTOR3( 0.0f, 3.0f,-5.0f ),
                              &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ),
                              &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

The first step is to define the view matrix by calling **D3DXMatrixLookAtLH**. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second, third, and fourth parameters define the eye point, look-at point, and "up" direction. Here the eye is set back along the z-axis by five units and up three units, the look-at point is set at the origin, and "up" is defined as the y-direction.

The next step is to call **IDirect3DDevice8::SetTransform** to set the view transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the D3DTS_VIEW flag to specify that the view transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on view transformations, see **View Transformation**.

After defining the world transformation for the scene, you can prepare the projection transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the projection transformation matrix is described in Step 3: Defining the Projection Transformation Matrix

Microsoft DirectX 8.1 (C++)

Step 3: Defining the Projection Transformation Matrix

[This is preliminary documentation and is subject to change.]

The projection transformation matrix defines how geometry is transformed from 3-D view space to 2-D viewport space.

The following code fragment creates the projection transformation matrix and then sets the current projection transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

The first step is to call **D3DXMatrixPerspectiveFovLH** to set up the projection matrix. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second parameter defines the field of view, which tells how objects in the distance get smaller. A typical field of view is 1/4 pi, which is what the sample uses. The third parameter defines the aspect ratio. The sample uses the typical aspect ratio of 1. The fourth and fifth parameters define the near and far clipping plane. This determines the distance at which geometry should no longer be rendered. The Matrices sample project has its near clipping plane set at 1 and its far clipping plane set at 100.

The next step is to call **IDirect3DDevice8::SetTransform** to apply the transformation to the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the D3DTS_PROJECTION flag to specify that the projection transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on projection transformations, see Projection Transformation.

This tutorial has shown you how to use matrices. Tutorial 4: Creating and Using Lights shows how to add lights to your scene for more realism.

Microsoft DirectX 8.1 (C++)

Tutorial 4: Creating and Using Lights

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® lights add more realism to 3-D objects. When used, each geometric object in the scene will be lit based on the location and type of lights that are used. The sample code in this tutorial introduces the topics of lights and materials.

This tutorial has the following steps to create a material and a light.

- Step 1: Initializing Scene Geometry
- Step 2: Setting up Material and Light

Note The path of the Lights sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut04_Lights.

The sample code in the Lights project is nearly identical to the sample code in the Matrices project. The Creating and Using Lights tutorial focuses only on the code unique to creating and using lights

and does not cover setting up Direct3D, handling Microsoft Windows messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Microsoft DirectX 8.1 (C++)

Step 1: Initializing Scene Geometry

[This is preliminary documentation and is subject to change.]

One of the requirements of using lights is that each surface has a normal. To do this, the Lights sample project uses a different custom vertex type. The new custom vertex format has a 3-D position and a surface normal. The surface normal is used internally by Microsoft® Direct3D® for lighting calculations.

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The 3-D position for the vertex.
    D3DXVECTOR3 normal;   // The surface normal for the vertex.
};

// Custom FVF.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL)
```

Now that the correct vector format is defined, the Lights sample project calls **InitGeometry**, an application-defined function that creates a cylinder. The first step is to create a vertex buffer that stores the points of the cylinder as shown in the following sample code.

```
// Create the vertex buffer.
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 50*2*sizeof(CUSTOMVERTEX),
                                              0 /* Usage */, D3DFVF_CUSTOMVERTEX,
                                              D3DPOOL_DEFAULT, &g_pVB ) ) )
    return E_FAIL;
```

The next step is to fill the vertex buffer with the points of the cylinder. Note that in the following sample code, each point is defined by a position and a normal.

```
for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);
    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0f, cosf(theta) );
    pVertices[2*i+0].normal   = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0f, cosf(theta) );
    pVertices[2*i+1].normal   = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
}
```

After the preceding sample code fills the vertex buffer with the vertices for a cylinder, the vertex buffer is ready for rendering. But first, the material and light for this scene must be set up before rendering the cylinder. This is described in Step 2: Setting up Material and Light.

Microsoft DirectX 8.1 (C++)

Step 2: Setting up Material and Light

[This is preliminary documentation and is subject to change.]

To use lighting in Microsoft® Direct3D®, you must create one or more lights. To determine which color a geometric object reflects, a material is created that is used to render geometric objects. Before rendering the scene, the Lights sample project calls **SetupLights**, an application-defined function that sets up one material and one directional light.

Creating a Material

A material defines the color that is reflected off the surface of a geometric object when a light hits it. The following code fragment uses the **D3DMATERIAL8** structure to create a material that is yellow.

```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
mtrl.Diffuse.b = mtrl.Ambient.b = 0.0f;
mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
g_pd3dDevice->SetMaterial( &mtrl );
```

The diffuse color and ambient color for the material are set to yellow. The call to the **IDirect3DDevice8::SetMaterial** method applies the material to the Microsoft® Direct3D® device used to render the scene. The only parameter that **SetMaterial** accepts is the address of the material to set. After this call is made, every primitive will be rendered with this material until another call is made to **SetMaterial** that specifies a different material.

Now that material has been applied to the scene, the next step is to create a light.

Creating a Light

There are three types of lights available in Microsoft® Direct3D®: point lights, directional lights, and spotlights. The sample code creates a directional light, which is a light that goes in one direction, and it oscillates the direction of the light.

The following code fragment uses the **D3DLIGHT8** structure to create a directional light.

```
D3DXVECTOR3 vecDir;
D3DLIGHT8 light;
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;
```

The following code fragment sets the diffuse color for this light to white.

```
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
```

The following code fragment rotates the direction of the light around in a circle.

```
vecDir = D3DXVECTOR3(cosf(timeGetTime()/360.0f),
                     0.0f,
                     sinf(timeGetTime()/360.0f) );
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );
```

The call to **D3DXVec3Normalize** normalizes the direction vector used to determine the direction of the light.

A range can be specified to tell Direct3D how far the light will have an effect. This member does not affect directional lights. The following code fragment assigns a range of 1000 units to this light.

```
light.Range = 1000.0f;
```

The following code fragment assigns the light to the Direct3D device by calling **IDirect3DDevice8::SetLight**.

```
g_pd3dDevice->SetLight( 0, &light );
```

The first parameter that **SetLight** accepts is the index that this light will be assigned to. Note that if a light already exists at that location, it will be overwritten by the new light. The second parameter is a pointer to the light structure that defines the light. The Lights sample project places this light at index 0.

The following code fragment enables the light by calling **IDirect3DDevice8::LightEnable**.

```
g_pd3dDevice->LightEnable( 0, TRUE);
```

The first parameter that **LightEnable** accepts is the index of the light to enable. The second parameter is a Boolean value that tells whether to turn the light on (TRUE) or off (FALSE). In the sample code above, the light at index 0 is turned on.

The following code fragment tells Direct3D to render lights by calling **IDirect3DDevice8::SetRenderState**.

```
g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
```

The first two parameters that **SetRenderState** accepts is which device state variable to modify and what value to set it to. This code sample sets the D3DRS_LIGHTING device variable to TRUE, which has the effect of enabling the rendering of lights.

The final step in this code sample is to turn on ambient lighting by again calling **SetRenderState**.

```
g_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00202020 );
```

The preceding code fragment sets the D3DRS_AMBIENT device variable to a light gray color (0x00202020). Ambient lighting will light up all objects by the given color.

For more information on lighting and materials, see Lights and Materials.

This tutorial has shown you how to use lights and materials. Tutorial 5: Using Texture Maps shows you how to add texture to surfaces.

Tutorial 5: Using Texture Maps

[This is preliminary documentation and is subject to change.]

While lights and materials add a great deal of realism to a scene, nothing adds more realism than adding textures to surfaces. Textures can be thought of as wallpaper that is shrink-wrapped onto a surface. You could place a wood texture on a cube to make it look like the cube is actually made of wood. The Texture sample project adds a banana peel texture to the cylinder created in Tutorial 4: Creating and Using Lights. This tutorial covers how to load textures, set up vertices, and display objects with texture.

This tutorial implements textures using the following steps:

- Step 1: Defining a Custom Vertex Format
- Step 2: Initializing Screen Geometry
- Step 3: Rendering the Scene

Note The path of the Texture sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut05_Textures.

The sample code in the Texture project is nearly identical to the sample code in the Lights project, except that the Texture sample project does not create a material or a light. The Using Texture Maps tutorial focuses only on the code unique to textures and does not cover initializing Microsoft® Direct3D®, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Microsoft DirectX 8.1 (C++)

Step 1: Defining a Custom Vertex Format

[This is preliminary documentation and is subject to change.]

Before using textures, a custom vertex format that includes texture coordinates must be used. Texture coordinates tell Microsoft® Direct3D® where to place a texture for each vector in a primitive. Texture coordinates range from 0.0 to 1.0, where (0.0, 0.0) represents the top-left side of the texture and (1.0, 1.0) represents the bottom-right side of the texture.

The following sample code shows how the Texture sample project sets up its custom vertex format to include texture coordinates.

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The position.
    D3DCOLOR    color;    // The color.
    FLOAT       tu, tv;   // The texture coordinates.
};
```

```
// The custom FVF, which describes the custom vertex structure.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)
```

For more information on texture coordinates, see **Texture Coordinates**.

Now that a custom vertex type has been defined, the next step is to load a texture and create a cylinder, as described in Step 2: Initializing Screen Geometry.

Microsoft DirectX 8.1 (C++)

Step 2: Initializing Screen Geometry

[This is preliminary documentation and is subject to change.]

Before rendering, the Texture sample project calls **InitGeometry**, an application-defined function that creates a texture and initializes the geometry for a cylinder.

Textures are created from file-based images. The following code fragment uses **D3DXCreateTextureFromFile** to create a texture from Banana.bmp that will be used to cover the surface of the cylinder.

```
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "Banana.bmp",
                                       &g_pTexture ) ) )
    return E_FAIL;
```

The first parameter that **D3DXCreateTextureFromFile** accepts is a pointer to the Microsoft® Direct3D® device that will be used to render the texture. The second parameter is a pointer to an ANSI string that specifies the filename from which to create the texture. This sample specifies Banana.bmp to load the image from that file. The third parameter is the address of a pointer to a texture object.

When the banana texture is loaded and ready to use, the next step is to create the cylinder. The following code sample fills the vertex buffer with a cylinder. Note that each point has the texture coordinates (tu, tv).

```
for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);

    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0, cosf(theta) );
    pVertices[2*i+0].color    = 0xffffffff;
    pVertices[2*i+0].tu       = ((FLOAT)i)/(50-1);
    pVertices[2*i+0].tv       = 1.0f;

    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0, cosf(theta) );
    pVertices[2*i+1].color    = 0xff808080;
    pVertices[2*i+1].tu       = ((FLOAT)i)/(50-1);
    pVertices[2*i+1].tv       = 0.0f;
}
```

Each vertex includes position, color, and texture coordinates. The code sample above sets the texture coordinates for each point so that the texture will wrap smoothly around the cylinder.

Now that the texture is loaded and the vertex buffer is ready for rendering, it is time to render the

display, as described in Step 3: Rendering the Scene.

Microsoft DirectX 8.1 (C++)

Step 3: Rendering the Scene

[This is preliminary documentation and is subject to change.]

After scene geometry has been initialized, it is time to render the scene. In order to render an object with texture, the texture must be set as one of the current textures. The next step is to set the texture stage states values. Texture stage states enable you to define the behavior of how a texture or textures are to be rendered. For example, you could blend multiple textures together.

The Texture sample project starts by setting the texture to use. The following code fragment sets the texture that the Microsoft® Direct3D® device will use to render with **IDirect3DDevice8::SetTexture**.

```
g_pd3dDevice->SetTexture( 0, g_pTexture );
```

The first parameter that **SetTexture** accepts is a stage identifier to set the texture to. A device can have up to eight set textures, so the maximum value here is 7. The Texture sample project has only one texture and places it at stage 0. The second parameter is a pointer to a texture object. The Texture sample project uses the texture object that it created in its **InitGeometry** application-defined function.

The following code sample sets the texture stage state values by calling the **IDirect3DDevice8::SetTextureStageState** method.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,   D3DTOP_DISABLE );
```

The first parameter that **SetTextureState** accepts is the stage index for the state variable to be set. This code sample is changing the values for the texture at stage 0, so it puts a 0 here. The next parameter is the texture state to set. For a list of all valid texture states and their meaning, see **D3DTEXTURESTAGESTATETYPE**. The next parameter is the value to set the texture state to. The value that you place here is based on the texture stage state value that you are modifying.

After setting up the desired values for each texture stage state, the cylinder can be rendered and texture will be added to the surface.

Another way to use texture coordinates is to have them automatically generated. This is done by using a texture coordinate index (TCI). The TCI uses a texture matrix to transform the (x,y,z) TCI coordinates into (tu, tv) texture coordinates. In the Texture sample project, the position of the vertex in camera space is used to generate texture coordinates.

The first step is to create the matrix that will be used for the transformation, as demonstrated in the following code fragment.

```
D3DXMATRIX mat;
mat._11 = 0.25f; mat._12 = 0.00f; mat._13 = 0.00f; mat._14 = 0.00f;
mat._21 = 0.00f; mat._22 =-0.25f; mat._23 = 0.00f; mat._24 = 0.00f;
mat._31 = 0.00f; mat._32 = 0.00f; mat._33 = 1.00f; mat._34 = 0.00f;
```

```
mat._41 = 0.50f; mat._42 = 0.50f; mat._43 = 0.00f; mat._44 = 1.00f;
```

After the matrix is created, it must be set by calling **IDirect3DDevice8::SetTransform**, as shown in the following code fragment.

```
g_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
```

The D3DTS_TEXTURE0 flag tells Direct3D to apply the transformation to the texture located at texture stage 0. The next step that this sample takes is to set more texture stage state values to get the desired effect. That is done in the following code fragment.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUN
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPA
```

The texture coordinates are set up, and now the scene is ready to be rendered. Notice that the coordinates are automatically created for the cylinder. This particular setup gives the effect of the texture being laid over the rendering screen after the geometric shapes have been rendered.

For more information on textures, see Textures.

This tutorial has shown you how to use textures for surfaces. Tutorial 6: Using Meshes shows you how to use complex geometric shapes with meshes.

Microsoft DirectX 8.1 (C++)

Tutorial 6: Using Meshes

[This is preliminary documentation and is subject to change.]

Complicated geometry is usually modeled using 3-D modeling software and saved to a file. An example of this is the .x file format. Microsoft® Direct3D® uses meshes to load the objects from these files. Meshes are somewhat complicated, but Microsoft® Direct3DX contains functions that make using meshes easier. The Meshes sample project introduces the topic of meshes and shows how to load, render, and unload a mesh.

This tutorial shows how to load, render, and unload a mesh using the following steps.

- Step 1: Loading a Mesh Object
- Step 2: Rendering a Mesh Object
- Step 3: Unloading a Mesh Object

Notes The path of the Meshes sample project is:

(SDK root)\Samples\Multimedia\Direct3D\Tutorials\Tut06_Meshes.

The sample code in the Meshes project is nearly identical to the sample code in the Lights project, except that the code in the Meshes project does not create a material or a light. The Using Meshes tutorial focuses only on the code unique to meshes and does not cover setting up Direct3D, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on

selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

This tutorial uses textures to cover the surface of the mesh. For more information on loading and using textures, see Tutorial 5: Using Texture Maps.

Microsoft DirectX 8.1 (C++)

Step 1: Loading a Mesh Object

[This is preliminary documentation and is subject to change.]

A Microsoft® Direct3D® application must first load a mesh before using it. The Meshes sample project loads the tiger mesh by calling **InitGeometry**, an application-defined function, after loading the required Direct3D objects.

A mesh needs a material buffer that will store all the materials and textures that will be used. The function starts by declaring a material buffer as shown in the following code fragment.

```
LPD3DXBUFFER pD3DXMtrlBuffer;
```

The following code fragment uses the **D3DXLoadMeshFromX** method to load the mesh.

```
// Load the mesh from the specified file.
if( FAILED( D3DXLoadMeshFromX( "tiger.x", D3DXMESH_SYSTEMMEM,
                               g_pd3dDevice, NULL,
                               &pD3DXMtrlBuffer, &g_dwNumMaterials,
                               &g_pMesh ) ) )
    return E_FAIL;
```

The first parameter that **D3DXLoadMeshFromX** accepts is a pointer to a string that tells the name of the Microsoft DirectX® file to load. This sample loads the tiger mesh from Tiger.x.

The second parameter tells Direct3D how to create the mesh. The sample uses the D3DXMESH_SYSTEMMEM flag, which is equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM. Both of these flags tell Direct3D to put the index buffer and vertex buffer for the mesh in system memory.

The third parameter is a pointer to a Direct3D device that will be used to render the mesh.

The fourth parameter is a pointer to an **ID3DXBuffer** object. This object will be filled with information about neighbors for each face. This information is not required for this sample, so this parameter is set to NULL.

The fifth parameter also takes a pointer to a **ID3DXBuffer** object. After this method is finished, this object will be filled with D3DXMATERIAL structures for the mesh.

The sixth parameter is a pointer to the number of D3DXMATERIAL structures placed into the *ppMaterials* array after the method returns.

The seventh parameter is the address of a pointer to a mesh object, representing the loaded mesh.

After loading the mesh object and material information, you need to extract the material properties and texture names from the material buffer.

The Meshes sample project does this by first getting the pointer to the material buffer. The following code fragment uses the **ID3DXBuffer::GetBufferPointer** method to get this pointer.

```
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

The following code fragment creates new mesh and texture objects based on the total number of materials for the mesh.

```
g_pMeshMaterials = new D3DMATERIAL8[g_dwNumMaterials];
g_pMeshTextures  = new LPDIRECT3DTEXTURE8[g_dwNumMaterials];
```

For each material in the mesh the following steps occur.

The first step is to copy the material, as shown in the following code fragment.

```
g_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
```

The second step is to set the ambient color for the material, as shown in the following code fragment.

```
g_pMeshMaterials[i].Ambient = g_pMeshMaterials[i].Diffuse;
```

The final step is to create the texture for the material, as shown in the following code fragment.

```
// Create the texture.
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice,
                                       d3dxMaterials[i].pTextureFilename,
                                       &g_pMeshTextures[i] ) ) )
    g_pMeshTextures[i] = NULL;
}
```

After loading each material, you are finished with the material buffer and need to release it by calling **IUnknown::Release**.

```
pD3DXMtrlBuffer->Release();
```

The mesh, along with the corresponding materials and textures are loaded. The mesh is ready to be rendered to the display, as described in Step 2: Rendering a Mesh Object.

Microsoft DirectX 8.1 (C++)

Step 2: Rendering a Mesh Object

[This is preliminary documentation and is subject to change.]

In step 1 the mesh was loaded and is now ready to be rendered. It is divided into a subset for each material that was loaded for the mesh. To render each subset, the mesh is rendered in a loop. The first step in the loop is to set the material for the subset, as shown in the following code fragment.

```
g_pd3dDevice->SetMaterial( &g_pMeshMaterials[i] );
```

The second step in the loop is to set the texture for the subset, as shown in the following code

fragment.

```
g_pd3dDevice->SetTexture( 0, g_pMeshTextures[i] );
```

After setting the material and texture, the subset is drawn with the **ID3DXBaseMesh::DrawSubset** method, as shown in the following code fragment.

```
g_pMesh->DrawSubset( i );
```

The **DrawSubset** method takes a **DWORD** that specifies which subset of the mesh to draw. This sample uses a value that is incremented each time the loop runs.

After using a mesh, it is important to properly remove the mesh from memory, as described in Step 3: Unloading a Mesh Object.

Microsoft DirectX 8.1 (C++)

Step 3: Unloading a Mesh Object

[This is preliminary documentation and is subject to change.]

After any Microsoft® DirectX® program finishes, it needs to deallocate any DirectX objects that it used and invalidate the pointers to them. The mesh objects used in this sample also need to be deallocated. When it receives a WM_DESTROY message, the Meshes sample project calls **Cleanup**, an application-defined function, to handle this.

The following code fragment deletes the material list.

```
if( g_pMeshMaterials )
    delete[] g_pMeshMaterials;
```

The following code fragment deallocates each individual texture that was loaded and then deletes the texture list.

```
if( g_pMeshTextures )
{
    for( DWORD i = 0; i < g_dwNumMaterials; i++ )
    {
        if( g_pMeshTextures[i] )
            g_pMeshTextures[i]->Release();
    }
delete[] g_pMeshTextures;
```

The following code fragment deallocates the mesh object.

```
Delete the mesh object
if( g_pMesh )
    g_pMesh->Release();
```

This tutorial has shown you how to load and render meshes. This is the last tutorial in this section. To see how a typical Direct3D application is written, see DirectX Graphics C/C++ Samples.

Microsoft DirectX 8.1 (C++)

Using Direct3D

[This is preliminary documentation and is subject to change.]

This section explains the operation of the Microsoft® Direct3D® fixed function pipeline. The pipeline consists of several building blocks. These blocks are detailed in the following sections.

- Vertex Data
- Transforms
- Viewports and Clipping
- Lights and Materials
- Textures
- Rendering
- About Devices

Microsoft DirectX 8.1 (C++)

Vertex Data

[This is preliminary documentation and is subject to change.]

Vertex data is presented to the pipeline in the form of object primitives. The following topics discuss three-dimensional (3-D) coordinates, how coordinates make up vertex data for an object, and how the vertex data declaration determines how the pipeline functions.

- 3-D Coordinate Systems and Geometry
- Fixed Function Vertex and Pixel Processing
- Programmable Vertex and Pixel Processing
- Programmable Stream Model
- Device-Supported Primitive Types

Microsoft DirectX 8.1 (C++)

3-D Coordinate Systems and Geometry

[This is preliminary documentation and is subject to change.]

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

- 3-D Coordinate Systems
- 3-D Primitives

- Rectangles
- Vectors, Vertices, and Quaternions
- Face and Vertex Normal Vectors
- Triangle Interpolants
- Triangle Rasterization Rules

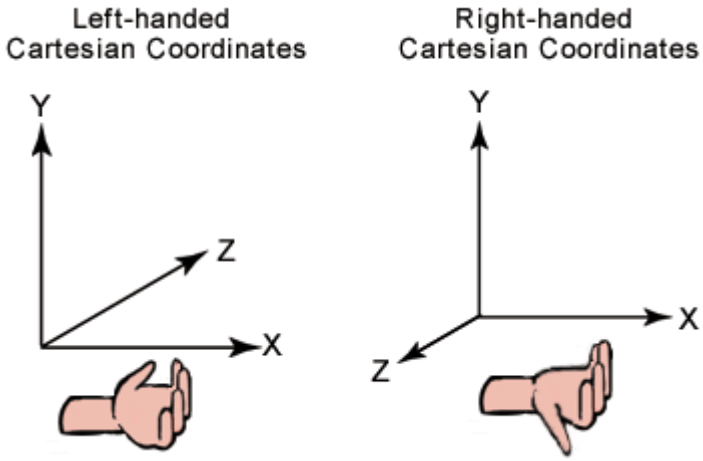
These topics provide you with a high-level understanding of the basic concepts employed by a Direct3D application. For more information about these topics, see Further Information.

Microsoft DirectX 8.1 (C++)

3-D Coordinate Systems

[This is preliminary documentation and is 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 that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.



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

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

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

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

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

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

Microsoft DirectX 8.1 (C++)

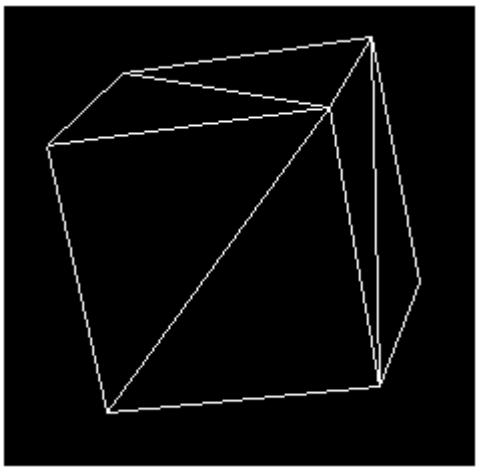
3-D Primitives

[This is preliminary documentation and is subject to change.]

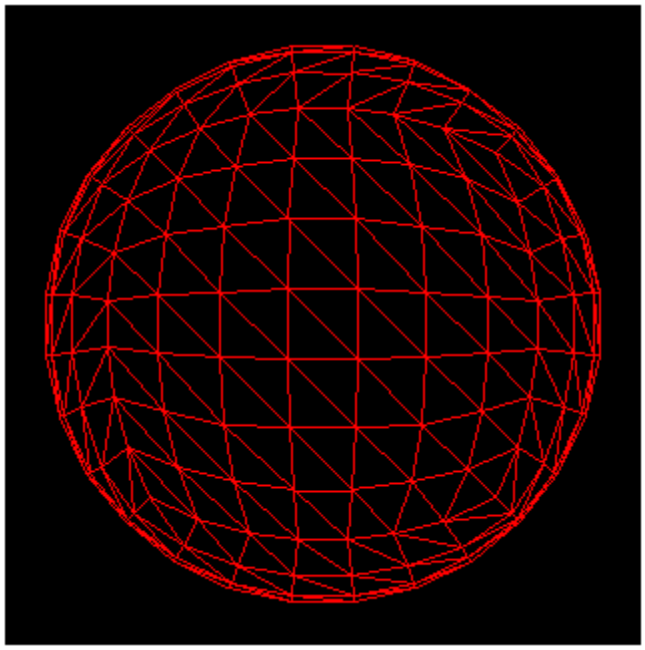
A 3-D primitive is a collection of vertices that form a single 3-D entity. The simplest primitive is a collection of points in a 3-D coordinate system, which is called a point list.

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

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



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



Microsoft DirectX 8.1 (C++)

Rectangles

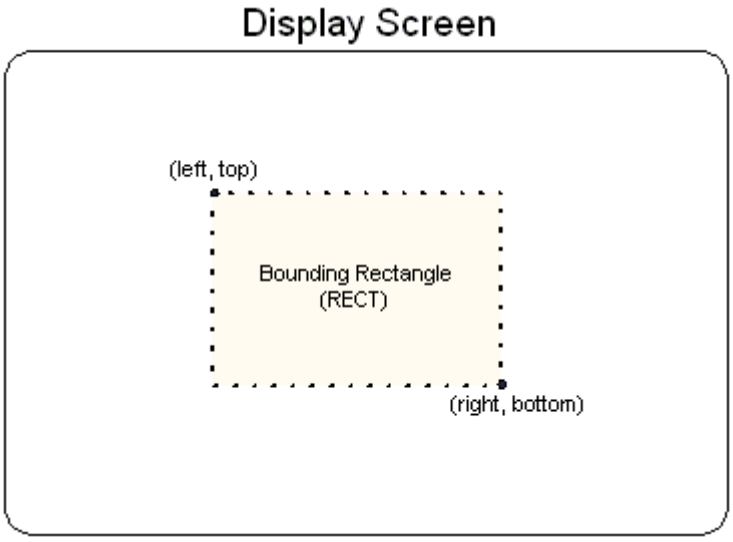
[This is preliminary documentation and is subject to change.]

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

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

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

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



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

Microsoft DirectX 8.1 (C++)

Vectors, Vertices, and Quaternions

[This is preliminary documentation and is subject to change.]

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

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

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

$$Q = q_1 \circ q_2$$

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

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

When you use composition and interpolation together, they provide you with a simple way to

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

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

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

- **D3DXQuaternionBaryCentric**
- **D3DXQuaternionConjugate**
- **D3DXQuaternionDot**
- **D3DXQuaternionExp**
- **D3DXQuaternionIdentity**
- **D3DXQuaternionInverse**
- **D3DXQuaternionIsIdentity**
- **D3DXQuaternionLength**
- **D3DXQuaternionLengthSq**
- **D3DXQuaternionLn**
- **D3DXQuaternionMultiply**
- **D3DXQuaternionNormalize**
- **D3DXQuaternionRotationAxis**
- **D3DXQuaternionRotationMatrix**
- **D3DXQuaternionRotationYawPitchRoll**
- **D3DXQuaternionSlerp**
- **D3DXQuaternionSquad**
- **D3DXQuaternionToAxisAngle**

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

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

- **D3DXVec3TransformCoord**
- **D3DXVec3TransformNormal**
- **D3DXVec3Unproject**

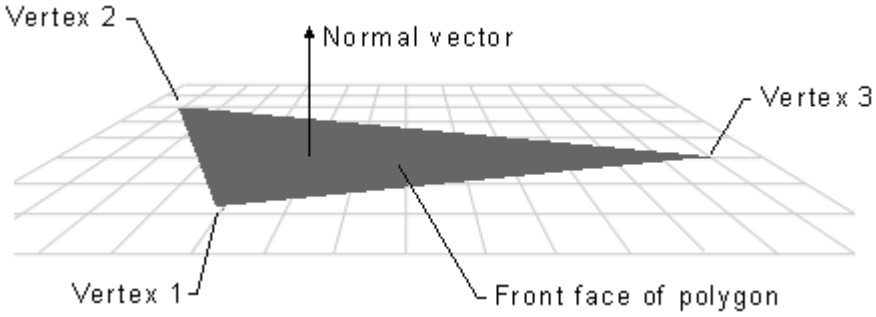
Many additional functions that simplify tasks using two- and four-component-vectors are included among the Math Functions supplied by the Direct3DX utility library.

Microsoft DirectX 8.1 (C++)

Face and Vertex Normal Vectors

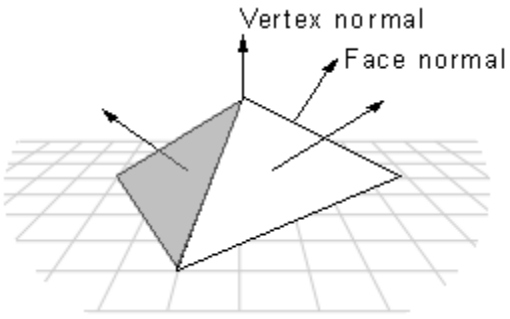
[This is preliminary documentation and is subject to change.]

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 Microsoft® Direct3D®, only the front of a face is visible. A front face is one in which vertices are defined in clockwise order.



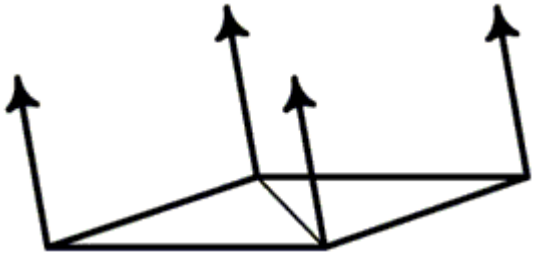
Any face that is not a front face is a back face. Direct3D does not always render back faces; therefore, back faces are said to be culled. You can change the culling mode to render back faces if you want. See Culling State for more information.

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



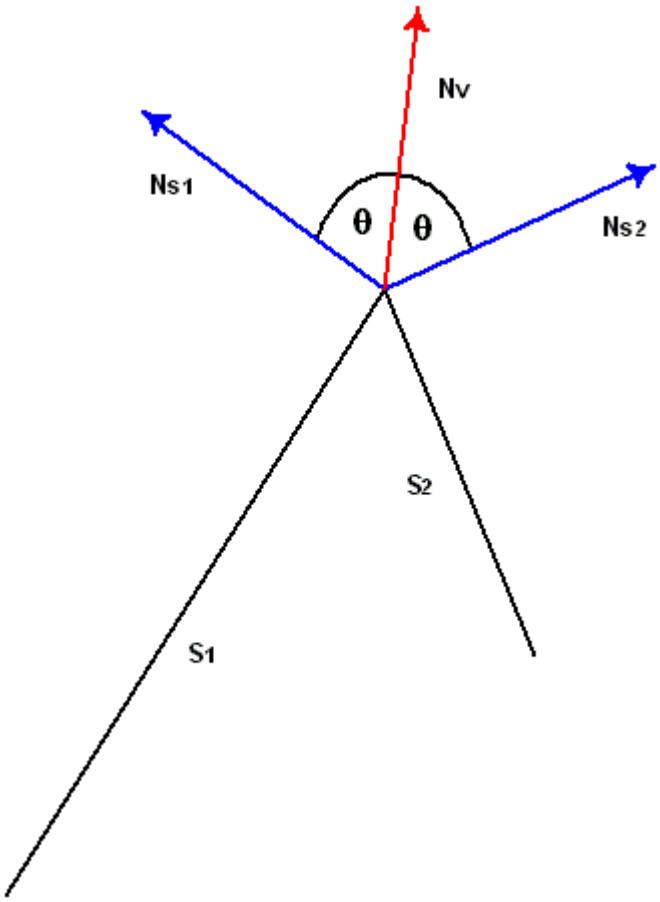
When applying Gouraud shading to a polygon, Direct3D uses the vertex normals to calculate the angle between the light source and the surface. It calculates the color and intensity values for the vertices and interpolates them for every point across all the primitive's surfaces. Direct3D calculates the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

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



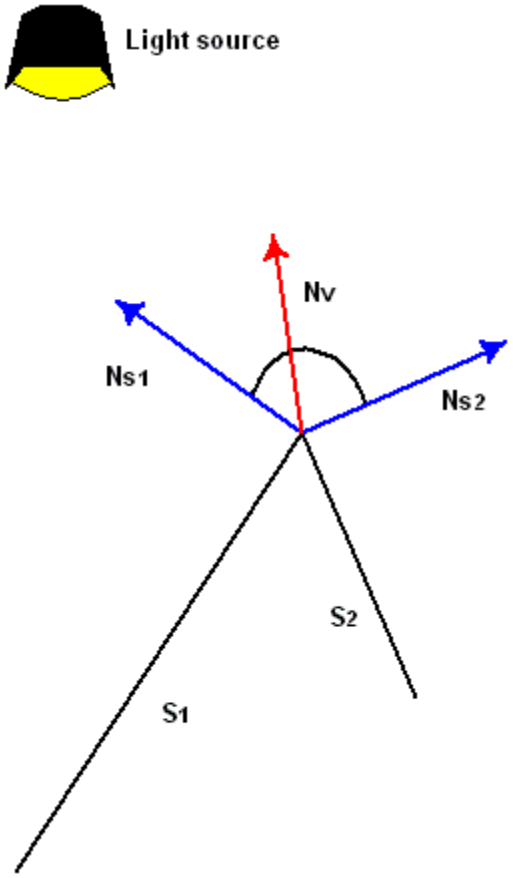
It is more likely, however, that your object is made up of triangle strips and the triangles are not coplanar. One simple way to achieve smooth shading across all the triangles in the strip is to first calculate the surface normal vector for each polygonal face with which the vertex is associated. The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

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

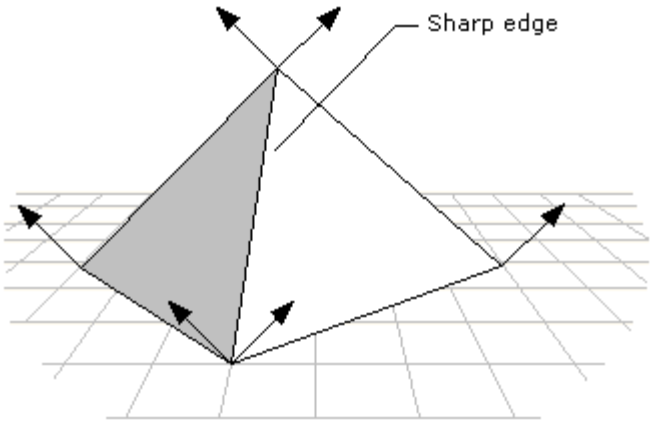


If the vertex normal leans toward one of the faces with which it is associated, it causes the light intensity to increase or decrease for points on that surface, depending on the angle it makes with the light source. An example is shown in the following figure. Again, these surfaces are seen edge-on.

The vertex normal leans toward S1, causing it to have a smaller angle with the light source than if the vertex normal had equal angles with the surface normals.



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



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

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

Microsoft DirectX 8.1 (C++)

Triangle Interpolants

[This is preliminary documentation and is subject to change.]

During rendering, the pipeline interpolates vertex data across each triangle. Five possible types of data can be interpolated:

- Diffuse color
- Specular color
- Diffuse alpha (triangle opacity)
- Specular alpha
- Fog factor (taken from specular alpha for fixed function vertex pipeline and from fog register for programmable vertex pipeline)

The interpolation of the vertex data is dependent on the current shading mode, as shown in the following table.

Shading mode	Description
Flat	Only the fog factor is interpolated in flat shade mode. For all other interpolants, 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.

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.

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

Microsoft DirectX 8.1 (C++)

Triangle Rasterization Rules

[This is preliminary documentation and is subject to change.]

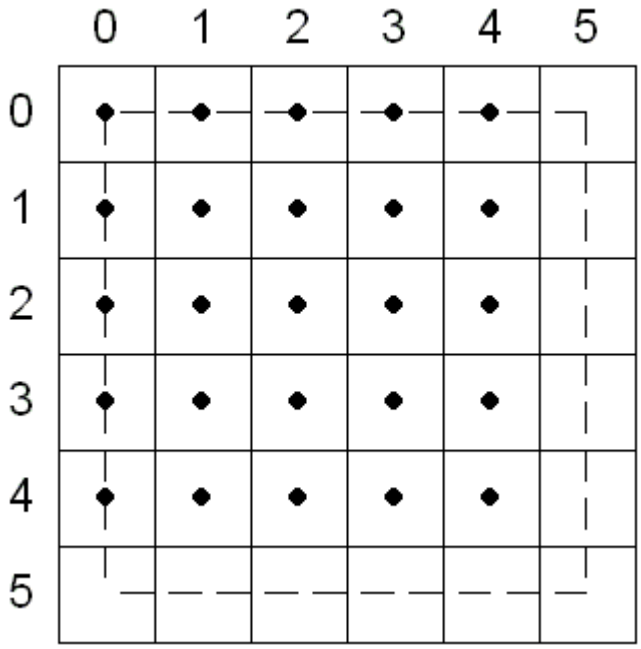
Often, the points specified for vertices do not precisely match the pixels on the screen. When this happens, Microsoft® Direct3D® applies triangle rasterization rules to decide which pixels apply to a given triangle.

Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI and OpenGL. In Direct3D, the center of the pixel is the decisive point. If

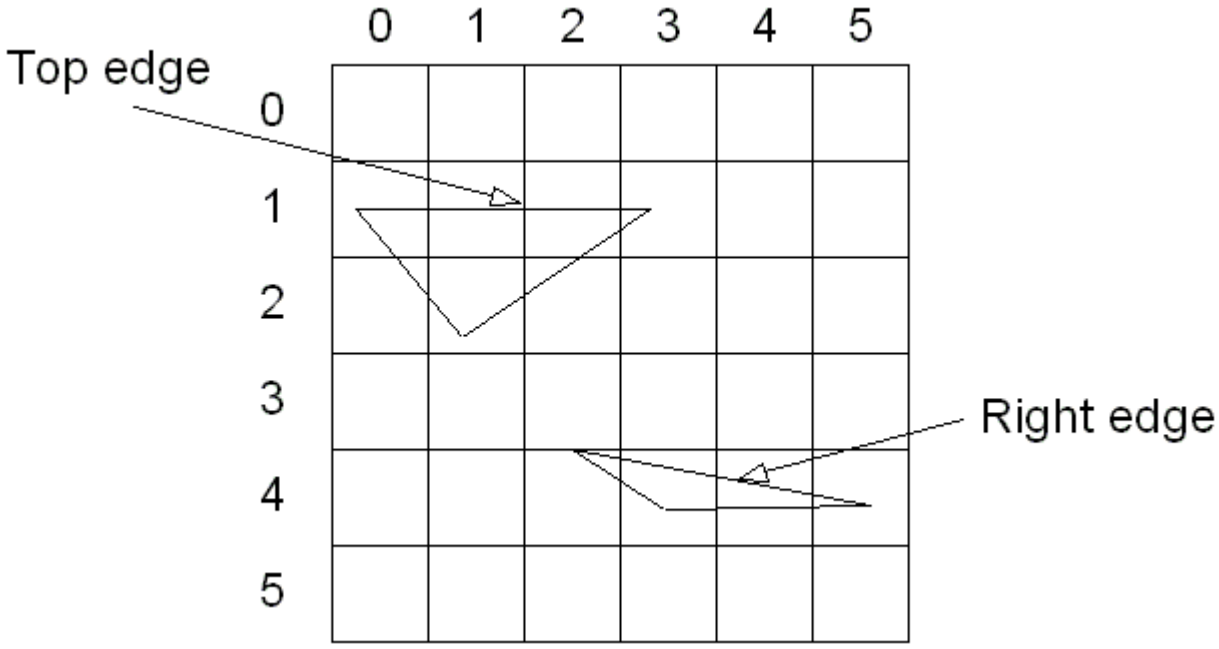
the center is inside a triangle, the pixel is part of the triangle. Pixel centers are at integer coordinates.

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

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

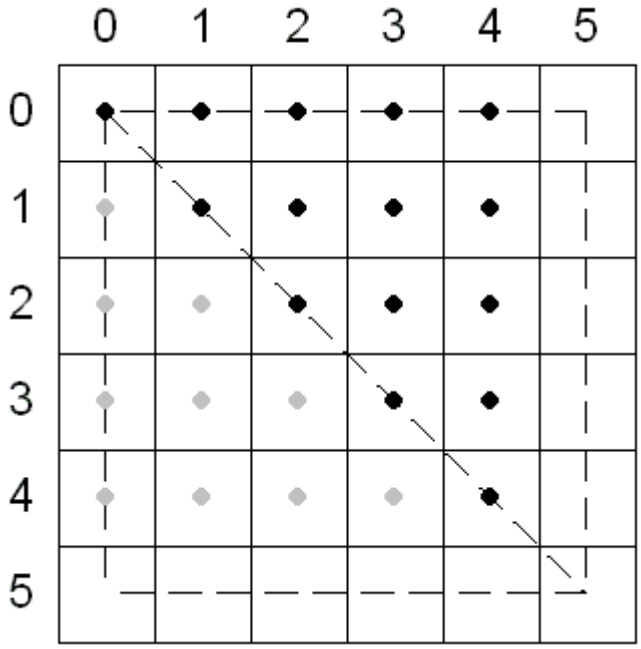


In the top-left filling convention, *top* refers to the vertical location of horizontal spans, and *left* refers to the horizontal location of pixels within a span. An edge cannot be a top edge unless it is horizontal—in general, most triangles have only left and right edges.

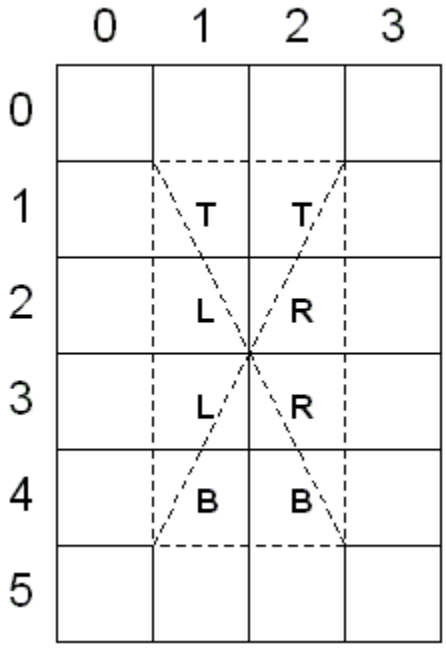


The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at (0, 0), (5, 0), and

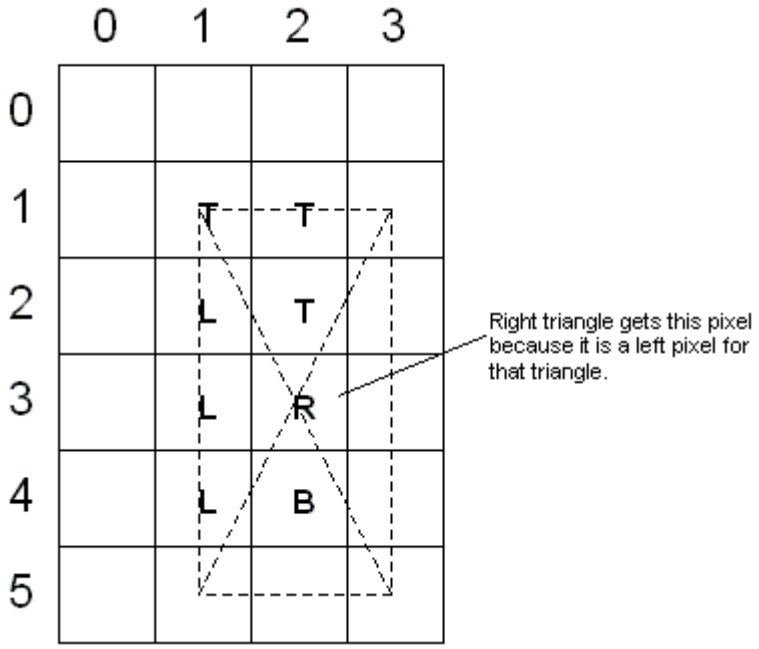
(5, 5), and the other at (0, 5), (0, 0), and (5, 5). The first triangle in this case gets 15 pixels (shown in black), whereas the second gets only 10 pixels (shown in gray) because the shared edge is the left edge of the first triangle.



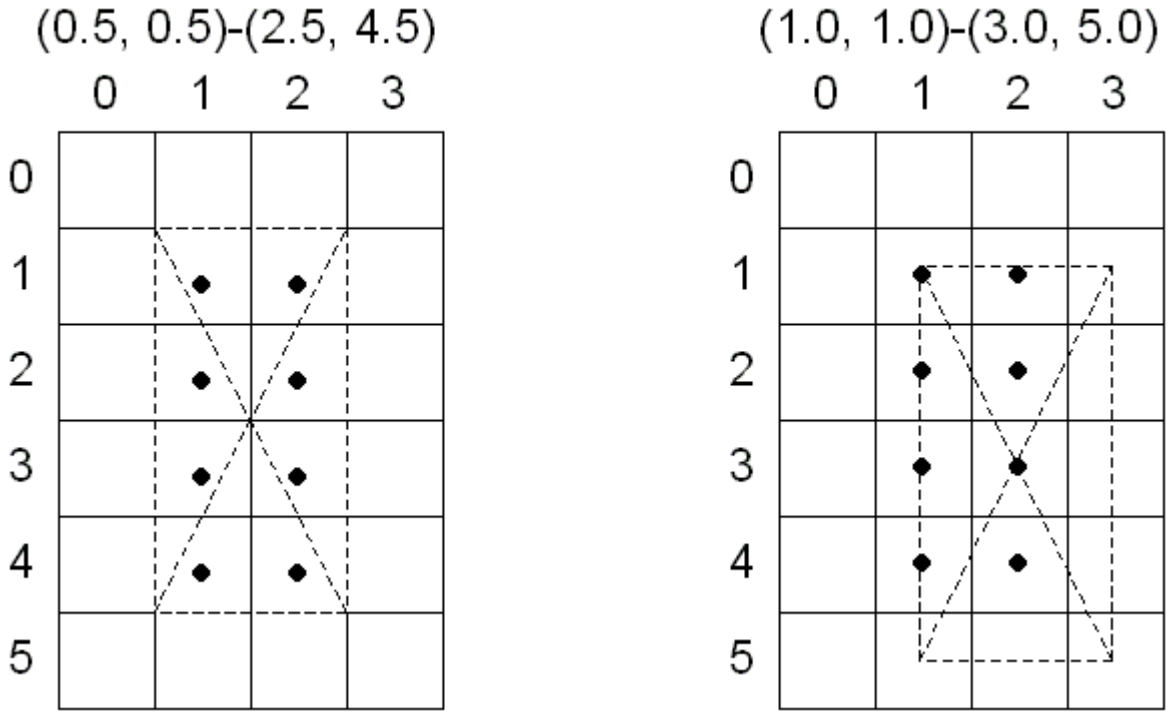
If you define a rectangle with its upper-left corner at (0.5, 0.5) and its lower-right corner at (2.5, 4.5), the center point of this rectangle is at (1.5, 2.5). When the Direct3D rasterizer tessellates this rectangle, the center of each pixel is unambiguously inside each of the four triangles, and the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



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



For both rectangles, the same pixels are affected.



Microsoft DirectX 8.1 (C++)

Fixed Function Vertex and Pixel Processing

[This is preliminary documentation and is subject to change.]

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

The geometry pipeline takes vertices as input. The transformation engine applies three transformations—the world, view, and projection transformations—to the vertices, clips the result, and passes everything to the rasterizer. The following image illustrates the sequence of steps.

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

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

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

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

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

More information on the vertex formats is contained in Vertex Formats.

More information on legacy vertex formats is contained in Legacy FVF Format.

Microsoft DirectX 8.1 (C++)

Vertex Formats

[This is preliminary documentation and is subject to change.]

A flexible vertex format (FVF) code describes the contents of vertices stored interleaved in a single data stream. It generally specifies data to be processed by the fixed function vertex processing pipeline.

Microsoft® Direct3D® applications can define model vertices in several different ways. Support for flexible vertex definitions, also known as *flexible vertex formats* or *flexible vertex format codes*,

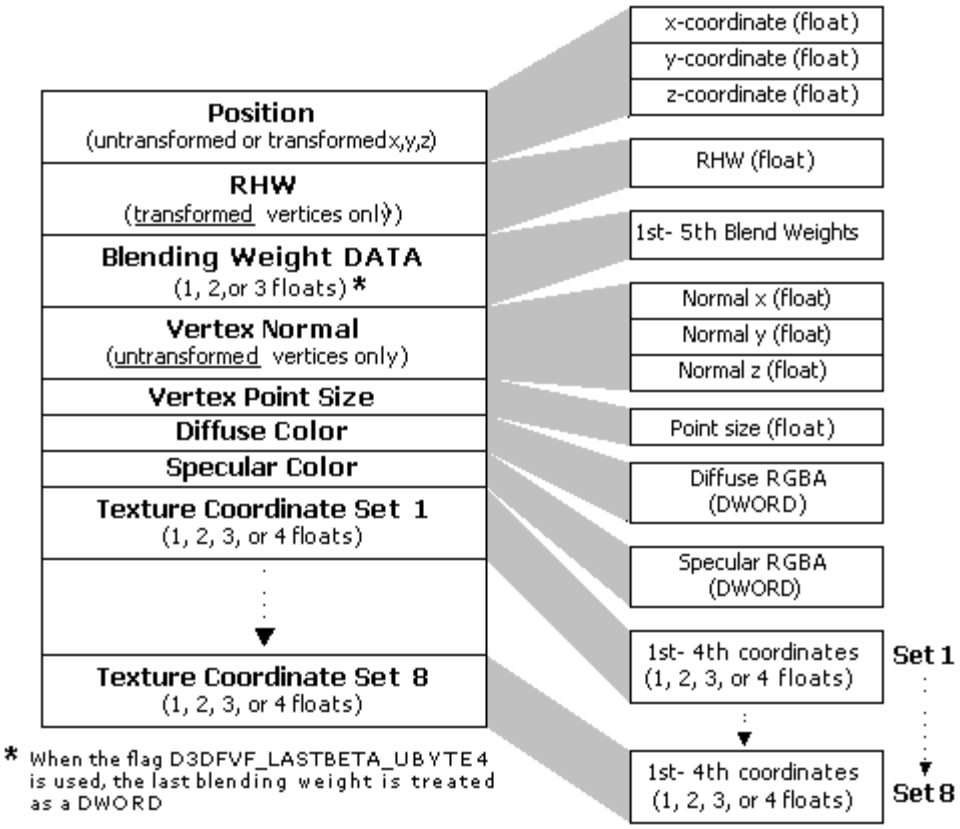
makes it possible for your application to use only the vertex components it needs, eliminating those components that aren't used. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render models. You describe how your vertices are formatted by using a combination of Flexible Vertex Format Flags.

The FVF specification includes formats for point size, specified by D3DFVF_PSIZE. This size is expressed in camera space units for non-TL vertices, and in device-space units for TL vertices.

The rendering methods of the **IDirect3DDevice8** interface provides C++ applications with methods that accept a combination of these flags, and uses them to determine how to render primitives. Basically, these flags tell the system which vertex components—position, vertex blending weights, normal, colors, the number and format of texture coordinates—your application uses and, indirectly, which parts of the rendering pipeline you want Direct3D to apply to them. In addition, the presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory and which you've omitted.

To determine device limitations, you can query a device for the D3DFVFCAPS_DONOTSTRIPELEMENTS and D3DFVFCAPS_TEXCOORDCOUNTMASK flexible vertex format flags. For more information, see the **FVFCaps** member of the **D3DCAPS8** structure.

One significant requirement that the system places on how you format your vertices is on the order in which the data appears. The following illustration depicts the required order for all possible vertex components in memory, and their associated data types.



Note Texture coordinates can be declared in different formats, allowing textures to be addressed using as few as one coordinate or as many as four texture coordinates (for 2-D projected texture coordinates). For more information, see Texture Coordinate Formats. Use the **D3DFVF_TEXCOORDSIZE*n*** set of macros to create bit patterns that identify the texture coordinate formats that your vertex format uses.

No application will use every component—the reciprocal homogeneous W (RHW) and vertex normal fields are mutually exclusive. Nor will most applications try to use all eight sets of texture coordinates, but Direct3D has this capacity. There are several restrictions on which flags you can use with other flags. For example, you cannot use the D3DFVF_XYZ and D3DFVF_XYZRHW flags together, as this would indicate that your application is describing a vertex's position with both untransformed and transformed vertices.

To use indexed vertex blending, the D3DFVF_LASTBETA_UBYTE4 flag should appear at the end of the FVF. The presence of this flag indicates that the fifth blending weight will be treated as a **DWORD** instead of float. For more information, see Indexed Vertex Blending.

The following code samples shows the difference between an FVF code that uses the D3DFVF_LASTBETA_UBYTE4 flag and one that doesn't. The FVF defined below does not use the D3DFVF_LASTBETA_UBYTE4 flag. The flag D3DFVF_XYZ3 is present when four blending indices are used because you always use (1 - the sum of the first three) for the fourth.

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB3|D3DFVF_NORMAL|D3DFVF_TEX1)

struct BLENDVERTEX
{
    D3DXVECTOR3 v;       // Referenced as v0 in the vertex shader
    FLOAT       blend1;  // Referenced as v1.x in the vertex shader
    FLOAT       blend2;  // Referenced as v1.y in the vertex shader
    FLOAT       blend3;  // Referenced as v1.z in the vertex shader
                         // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    D3DXVECTOR3 n;       // Referenced as v3 in the vertex shader
    FLOAT       tu, tv;  // Referenced as v7 in the vertex shader
};
```

The FVF defined below uses the D3DFVF_LAST_UBYTE4 flag.

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4 |D3DFVF_NORMAL|

struct BLENDVERTEX
{
    D3DXVECTOR3 v;       // Referenced as v0 in the vertex shader
    FLOAT       blend1;  // Referenced as v1.x in the vertex shader
    FLOAT       blend2;  // Referenced as v1.y in the vertex shader
    FLOAT       blend3;  // Referenced as v1.z in the vertex shader
                         // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    DWORD       indices; // Referenced as v2.xyzw in the vertex shader
    D3DXVECTOR3 n;       // Referenced as v3 in the vertex shader
    FLOAT       tu, tv;  // Referenced as v7 in the vertex shader
};
```

Use the following topics for additional information about the various formats your application can use to declare vertices.

- Untransformed and Unlit Vertices
- Untransformed and Lit Vertices
- Transformed and Lit Vertices
- Transformed and Lit Vertex Functionality
- Flexible Vertex Formats and Vertex Shaders

Microsoft DirectX 8.1 (C++)

Untransformed and Unlit Vertices

[This is preliminary documentation and is subject to change.]

The presence of the D3DFVF_XYZ, or any D3DFVF_XYZB*n* flag, and D3DFVF_NORMAL flags in the vertex description that you pass to rendering methods identifies the untransformed and unlit vertex type. By using untransformed and unlit vertices, your application effectively requests that Microsoft® Direct3D® perform all transformation and lighting operations using its internal algorithms. You can disable the Direct3D lighting engine for the primitives being rendered by setting the D3DRS_LIGHTING render state to FALSE.

Many applications use this vertex type, as it frees them from implementing their own transformation and lighting engines. However, because the system is making calculations for you, it requires that you provide a certain amount of information with each vertex.

- You are required to specify vertices in untransformed model coordinates. The system then applies world, view, and projection transformations to the model coordinates to position them in your scene and determine their final locations on the screen.
- You can include a vertex normal for more realistic lighting effects. Vertices lit by the system that don't include a vertex normal will use a dot product of 0 in all lighting calculations. These vertices are assumed to receive no incident light. The system uses the vertex normal, along with the current material, in its lighting calculations. For details, see Face and Vertex Normal Vectors, Lights and Materials.

Other than these requirements, you have the flexibility to use or disregard the other vertex components. For example, you can include a diffuse or specular color with your untransformed vertices. This was not possible before Microsoft DirectX® 6.0. Including individual colors for each vertex makes it possible to achieve shading effects that are more subtle and flexible than lighting calculations that use only the material color. Keep in mind that you must enable per-vertex color through the D3DRS_COLORVERTEX render state. Untransformed, unlit vertices can also include up to eight sets of texture coordinates.

When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid vertex format structure that includes a position, a vertex normal, a diffuse color, and two sets of texture coordinates.

```
//
// The vertex format description for this vertex would be:
// (D3DFVF_XYZ | D3DFVF_NORMAL |  D3DFVF_DIFFUSE | D3DFVF_TEX2)
//
typedef struct _UNLITVERTEX
{
    float x, y, z;     // position
    float nx, ny, nz;  // normal
    DWORD Diffuse;     // diffuse color
    float tu1, tv1;    // texture coordinates
    float tu2, tv2;
  tv2;
} UNLITVERTEX, *LPUNLITVERTEX;
```

The vertex description for the preceding structure is a combination of the D3DFVF_XYZ, D3DFVF_NORMAL, D3DFVF_DIFFUSE, and D3DFVF_TEX2 flexible vertex format flags.

For more information, see Vertex Formats.

Microsoft DirectX 8.1 (C++)

Untransformed and Lit Vertices

[This is preliminary documentation and is 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 Microsoft® Direct3D® rendering methods, you are identifying your vertices as untransformed but already lit. For information about other dependencies and exclusions, see the description for the Flexible Vertex Format Flags.

By using untransformed and lit vertices, your application requests that Direct3D not perform any lighting calculations on your vertices, but it should still transform them using the previously set world, view, and projection matrices. Because the system isn't doing lighting calculations, it doesn't need a vertex normal. The system uses the diffuse and specular components at each vertex for shading. These colors might be arbitrary, or they might be computed using your own lighting formulas. If you don't include a diffuse or specular component, the system uses the default colors. The default value for the diffuse color is 0xFFFFFFFF, and the default value for the specular color is 0x0.

Like the other vertex types, except for including a position and some amount of color information, you are free to include or disregard the texture coordinate sets in the unlit vertex format.

When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid untransformed and lit vertex, with diffuse and specular vertex colors, and three sets of texture coordinates.

```
//
// The vertex format description for this vertex would be:
// (D3DFVF_XYZ | D3DFVF_DIFFUSE |  D3DFVF_SPECULAR | D3DFVF_TEX3)
//
typedef struct _LITVERTEX {
    float x, y, z;   // position
    DWORD Diffuse;   // diffuse color
    DWORD Specular;  // specular color
    float tu1, tv1;  // texture coordinates
    float tu2, tv2;
    float tu3, tv3;
} LITVERTEX, *LPLITVERTEX;
```

The vertex description for the preceding structure is a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags.

Microsoft DirectX 8.1 (C++)

Transformed and Lit Vertices

[This is preliminary documentation and is subject to change.]

If you include the D3DFVF_XYZRHW flag in your vertex format description, you are telling the system that your application is using transformed and lit vertices. This means that Microsoft® Direct3D® doesn't transform your vertices with the world, view, or projection matrices, nor does it

perform any lighting calculations. It assumes that your application has taken care of these steps. This fact makes transformed and lit vertices common when porting existing 3-D applications to Direct3D. In short, Direct3D does not modify transformed and lit vertices at all. It passes them directly to the driver to be rasterized.

The vertex format flags associated with untransformed vertices and lighting (D3DFVF_XYZ and D3DFVF_NORMAL) are not allowed if D3DFVF_XYZRHW is present. For more about flag dependencies and exclusions, see the description for each of the Flexible Vertex Format Flags.

The system requires that the vertex position you specify be already transformed. The x and y values must be in screen coordinates, and z must be the depth value of the pixel to be used in the z-buffer. Z values can range from 0.0 to 1.0, where 0.0 is the closest possible position to the user, and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include a reciprocal of homogeneous W (RHW) value. RHW is the reciprocal of the W coordinate from the homogeneous point (x,y,z,w) at which the vertex exists in *projection space*. This value often works out to be the distance from the eyepoint to the vertex, taken along the z-axis.

Other than the position and RHW requirements, this vertex format is similar to an untransformed and lit vertex. To recap:

- The system doesn't do any lighting calculations with this format, so it doesn't need a vertex normal.
- You can specify a diffuse or specular color. If you don't, the system uses 0x0 for specular color and 0xFFFFFFFF for diffuse color.
- You can use up to eight sets of texture coordinates, or none at all.

When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid transformed and lit vertex, with diffuse and specular vertex colors, and one set of texture coordinates.

```
//
// The vertex format description for this vertex would be
// (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |  D3DFVF_SPECULAR | D3DFVF_TEX1)
//
typedef struct _TRANSLITVERTEX {
    float x, y;      // screen position
    float z;         // Z-buffer depth
    float rhw;       // reciprocal homogeneous W
    DWORD Diffuse;   // diffuse color
    DWORD Specular;  // specular color
    float tu1, tv1;  // texture coordinates
} TRANSLITVERTEX, *LPTRANSLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZRHW, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flexible vertex format flags.

For more information, see Vertex Formats.

Microsoft DirectX 8.1 (C++)

Transformed and Lit Vertex Functionality

[This is preliminary documentation and is subject to change.]

Vertex data specified with a flexible vertex format (FVF) code has the property of being either transformed or nontransformed. The terminology used for FVF-specified transformed vertices is TL (transformed and lit) vertices. Microsoft® Direct3D® for Microsoft DirectX® 8.0 continues to support TL vertex data, as in previous releases. However, TL vertices are subject to some conditions that are not applicable to nontransformed vertices.

TL vertex data is possible only for vertices specified by an FVF code, and it is not valid as input for a programmable vertex shader. Programmable vertex shaders have the flexibility to take various forms of transformed vertex data, but this is different from FVF-TL vertex data because the interpretation is part of the shader, not a property of the vertex data. Because it is defined by an FVF code, TL vertex data is also inherently single stream.

TL vertex data is not guaranteed to be clipped by Direct3D, thus it is required that TL vertex primitives be clipped prior to sending them to Direct3D for rendering. The x and y values are required to be clipped to the viewport, or, if available, the device guardband. The z values for TL vertices are required to be between 0. and 1, inclusive. It is also required that the RHW values be within the range [0 < 1/**D3DCAPS8.MaxVertexW**]. The RHW requirement is to guarantee that the w values are within the supported range of the hardware rasterization device. Note that the w, and RHW, range of vertices resulting from a perspective projection transformation can be modified by applying a scale to all values in the projection matrix.

Note If D3DPMISCCAPS_CLIPTLVERTS is set, then the device clips post-transformed vertex data to the z and (x, y) viewport limits. Clipping to user-defined clip planes is not supported for post-transformed vertex data. If the D3DPMISCCAPS_CLIPTLVERTS capability is not set, then the application is required to clip these primitives to the z limit and at least to the guardband extent in x and y. On the other hand, clipping for pre-transformed vertices is fully supported in both drawing and vertex processing calls to a destination vertex buffer followed by another drawing call. This includes user-defined clip planes as well as the z and (x, y) viewport limits.

TL vertex data is always passed directly to the driver for rendering. When using vertex buffers with TL vertex data, there can be significant performance advantages to having the driver allocate these vertex buffers in AGP or video memory. Note that the allocation of a TL vertex buffer may be driver-allocated even when non-TL vertex data —either FVF or non-FVF—is not allowed to be driver-allocated, as would be the case when running on a hardware device that does not support transformation and lighting.

For more information, see Transformed and Lit Vertices.

Microsoft DirectX 8.1 (C++)

Flexible Vertex Formats and Vertex Shaders

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® for Microsoft DirectX® 8.0 has a simplified programming model for using the fixed function vertex processing pipeline with a single input stream, providing functionality very similar to that of previous DirectX releases. In this case, the vertex shader consists of a flexible vertex format (FVF) code that is passed in place of a vertex shader handle when setting the current vertex shader. The handle space for vertex shaders is managed by the run-time library so that handles that are valid FVF codes are reserved for this usage.

Setting an FVF code as the current vertex shader causes the vertex processing to load from stream zero only, and to interpret the vertex elements as defined in the FVF code.

The following code example illustrates how to use an FVF code as a vertex shader in your C++ application.

First, define a structure for your custom vertex type and initialize the vertices. In this case, three vertices are initialized for rendering a triangle.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)

CUSTOMVERTEX g_Vertices[] =
{
    { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
    {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};
```

Then, render the primitive using stream zero.

```
d3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
d3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

For more information, see Vertex Shaders.

Microsoft DirectX 8.1 (C++)

Legacy FVF Format

[This is preliminary documentation and is subject to change.]

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

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

This following topics cover legacy FVF format types.

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

Microsoft DirectX 8.1 (C++)

Vertex Legacy Type

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

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

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

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

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

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

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

pBigSquareVB->Unlock();
```

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

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

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

Microsoft DirectX 8.1 (C++)

LVertex Legacy Type

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

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

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

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

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

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

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

pBigSquareVB->Unlock();
```

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

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

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

Microsoft DirectX 8.1 (C++)

TLVertex Legacy Type

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

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

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

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

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

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

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f; v[3].rhw = 1.0f;
v[3].diffuse  = 0xffffff00;
```

```
v[3].specular = 0xffff0000;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

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

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

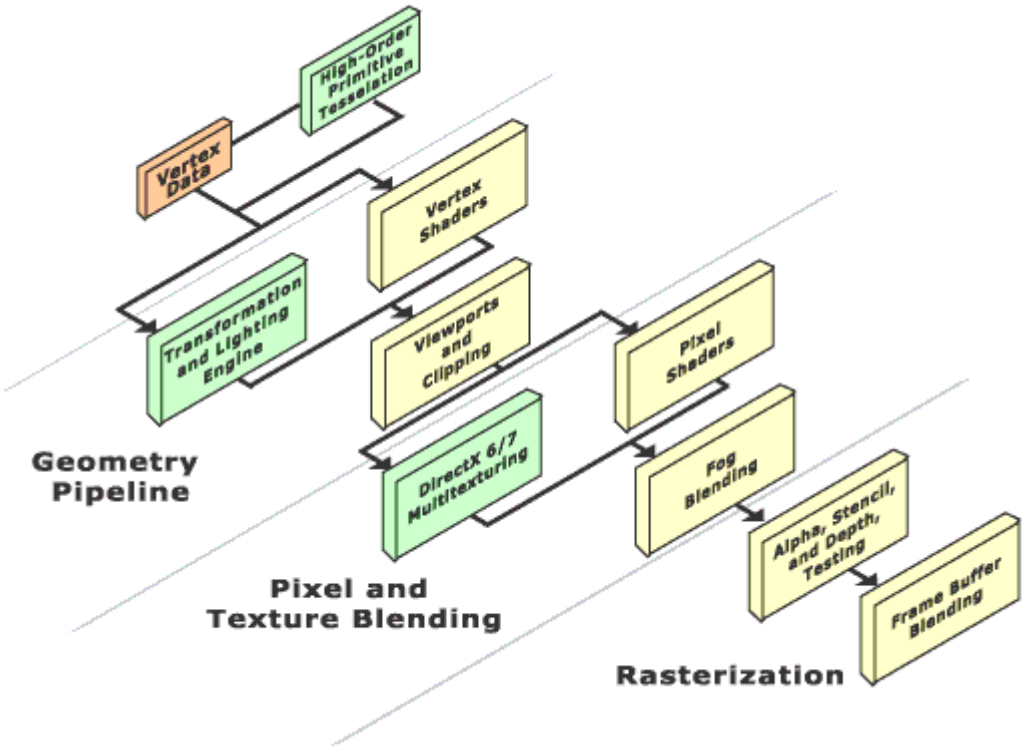
Passing an FVF to the **IDirect3DDevice8::SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

Microsoft DirectX 8.1 (C++)

Programmable Vertex and Pixel Processing

[This is preliminary documentation and is subject to change.]

Microsoft® DirectX® 8.0 features a new kind of graphics pipeline, featuring a high degree of programmability. The following diagram illustrates the major components and data flow in the programmable vertex and pixel pipeline. The transformation and lighting engine and DirectX 6.0 and 7.0 multitexturing modules have been replaced in the programmable pipeline by vertex and pixel shaders.



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

texture addressing operations.

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

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

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

Microsoft DirectX 8.1 (C++)

Introduction to Procedural Shaders

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

- *Matrix Palette Skinning*. Character animation with 8-10 bones per mesh.
- *Anisotropic Lighting*. Lighting that currently can be done only at the cost of textures for look-up tables.
- *Membrane Shaders*. Shaders for balloons, skin, and so on (1/cos(eye DOT normal).

- *Kubelka-Munk Shaders*. Shaders that take into account light that penetrates the surface.
- *Procedural Geometry*. Compositing meshes with procedural ones (spheres) to simulate muscles moving under the skin.
- *Displacement Mapping*. Modifying a mesh with a wave pattern or hump that can be tiled/repeated.

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

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

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

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

Fifth, a procedural model follows in the footsteps of a photo-real rendering heritage. There has been a tradition of using programmable shaders in high-end photo-real rendering for many years. In general, this area is unconstrained by performance, so programmable shaders represent the ultimate no-compromise goal for rendering technologies.

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

Microsoft DirectX 8.1 (C++)

Integration of Vertex Shaders into the Geometry Pipeline

[This is preliminary documentation and is subject to change.]

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

Any tessellation of high-order primitives should be done before execution of the vertex shader. Implementations that perform surface tessellation after the shader processing must do so in a way that is not apparent to the application and shader code. Because no semantic information is normally provided before the shader, a special token is used to identify which input stream component

represents the base position relative to which all other components are interpolated. No noninterpolable data channels are supported.

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

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

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

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

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

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

Microsoft DirectX 8.1 (C++)

Programmable Stream Model

[This is preliminary documentation and is subject to change.]

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

- ColorVertex Shader
- SingleTexture Shader
- MultiTexture Shader

Microsoft DirectX 8.1 (C++)

ColorVertex Shader

[This is preliminary documentation and is subject to change.]

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

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

```
struct XYZBuffer
{
    FLOAT x, y, z;
};

struct ColBuffer
{
    D3DCOLOR color;
};
```

The next step is to create a vertex shader declaration as shown in the code below.

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

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

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

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

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

```
g_d3dDevice->SetVertexShader( vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, sizeof(XYZBuffer));
g_d3dDevice->SetStreamSource( 1, colbuf, sizeof(ColBuffer));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count
```

Microsoft DirectX 8.1 (C++)

SingleTexture Shader

[This is preliminary documentation and is subject to change.]

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

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

```
struct XYZBuffer
{
```

```
    D3DVALUE x, y, z;
};

struct TEX0Buffer
{
    D3DVALUE tu, tv;
};
```

The next step is to create a vertex shader declaration as shown in the code below.

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

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

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

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

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

```
g_d3dDevice->SetVertexShader( vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, sizeof(XYZBuffer));
g_d3dDevice->SetStreamSource( 1, tex0buf, sizeof(TEX0Buffer));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count
```

Microsoft DirectX 8.1 (C++)

MultiTexture Shader

[This is preliminary documentation and is subject to change.]

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

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

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

struct Tex0Buffer
{
    D3DVALUE tu, tv;
```

```
};

struct Tex1Buffer
{
    D3DVALUE tu2, tv2;
};
```

The next step is to create a vertex shader declaration as shown in the code below.

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

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

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

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

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

```
g_d3dDevice->SetVertexShader( vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, sizeof(XYZBuffer) );
g_d3dDevice->SetStreamSource( 1, tex0buf, sizeof(Tex0Buffer) );
g_d3dDevice->SetStreamSource( 2, tex1buf, sizeof(Tex1Buffer) );
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count
```

Microsoft DirectX 8.1 (C++)

Device-Supported Primitive Types

[This is preliminary documentation and is subject to change.]

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

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

You can render primitive types from a C++ application with any of the rendering methods of the **IDirect3DDevice8** interface.

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

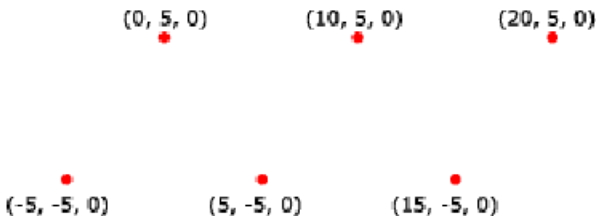
Microsoft DirectX 8.1 (C++)

Point Lists

[This is preliminary documentation and is 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.

The following illustration depicts a rendered point list.



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

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

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

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

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

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

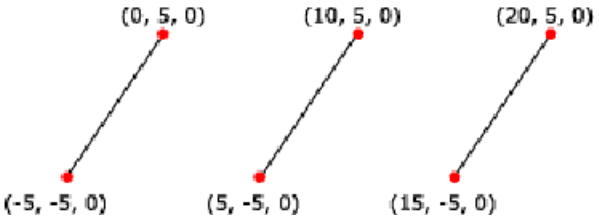
Microsoft DirectX 8.1 (C++)

Line Lists

[This is preliminary documentation and is 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. Applications create a line list by filling an array of vertices, and the number of vertices in a line list must be an even number greater than or equal to two.

The following illustration shows a rendered line list.



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

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

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

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

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

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

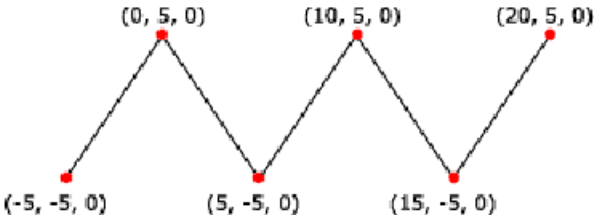
Microsoft DirectX 8.1 (C++)

Line Strips

[This is preliminary documentation and is 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.

The following illustration depicts a rendered line strip.



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

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

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

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

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_LINESTRIP, 0, 5 );
```

Microsoft DirectX 8.1 (C++)

Triangle Lists

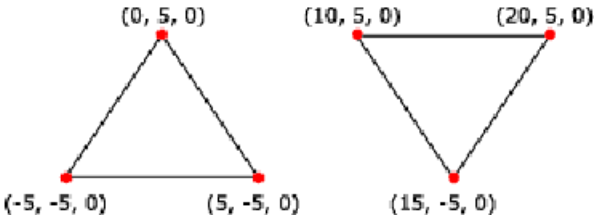
[This is preliminary documentation and is subject to change.]

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

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

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

The following illustration depicts a rendered triangle list.



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

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

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

};
```

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

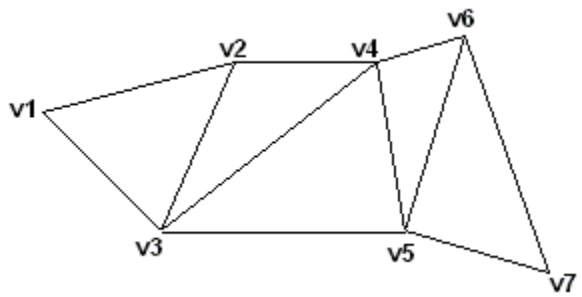
```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );
```

Microsoft DirectX 8.1 (C++)

Triangle Strips

[This is preliminary documentation and is 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 need only seven vertices to define the following triangle strip.

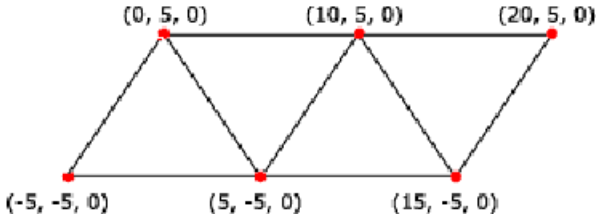


The system uses vertices v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the

vertices of the second and fourth triangles are out of order; this is required to make sure that all the triangles are drawn in a clockwise orientation.

Most objects in 3-D scenes are composed of triangle strips. This is because triangle strips can be used to specify complex objects in a way that makes efficient use of memory and processing time.

The following illustration depicts a rendered triangle strip.



The following code shows how to create vertices for this triangle strip.

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

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

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this triangle strip.

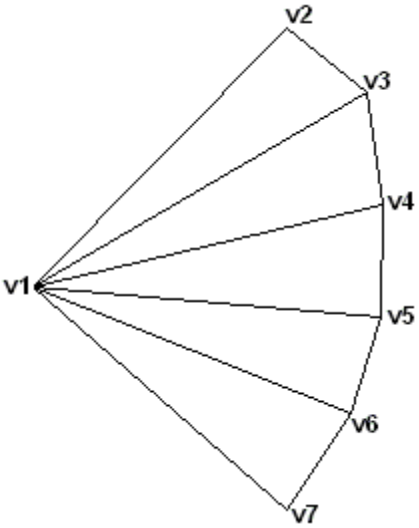
```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 4);
```

Microsoft DirectX 8.1 (C++)

Triangle Fans

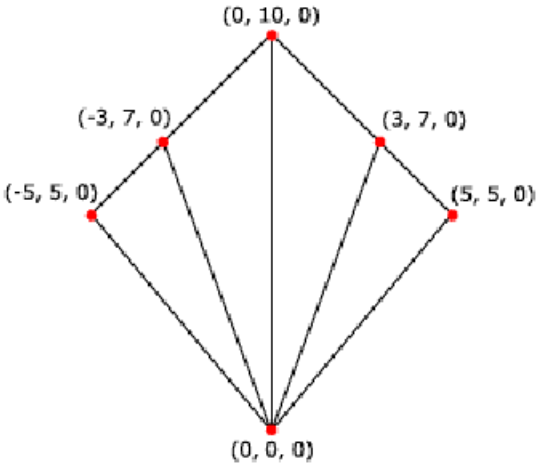
[This is preliminary documentation and is subject to change.]

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



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

This illustration depicts a rendered triangle fan.



The following code shows how to create vertices for this triangle fan.

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

CUSTOMVERTEX Vertices[] =
{
    { 0.0,  0.0,  0.0},
    {-5.0,  5.0,  0.0},
    {-3.0,   7.0,  0.0},
    { 0.0,  10.0,  0.0},
    { 3.0,   7.0,  0.0},
    { 5.0,   5.0,  0.0},
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this triangle fan.

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 4 );
```

Microsoft DirectX 8.1 (C++)

Transforms

[This is preliminary documentation and is subject to change.]

Transformations are used to convert object geoemtry from one coordinate space to another. The most common transformations are done using matrices. A matrix is an essential tool for holding the transformation values and applying them to the data. The topics below introduce matrices, and explain how to use them to generate the world, view and projection transformations.

- Matrices
- World Transformation

  Convert from world coordinates to view coordinates

- View Transformation

  Convert from view coordinates to projection coordinates

- Projection Transformation

  Convert from projection coordinates to screen coordinates

Microsoft DirectX 8.1 (C++)

Matrices

[This is preliminary documentation and is subject to change.]

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

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

Microsoft DirectX 8.1 (C++)

3-D Transformations

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

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

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

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

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

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

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

Microsoft DirectX 8.1 (C++)

Translation and Scaling

[This is preliminary documentation and is subject to change.]

Translation

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

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

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

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

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

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

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

Scaling

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

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

Microsoft DirectX 8.1 (C++)

Rotation

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

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

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

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

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

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

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

    return pOut;
}
```

Microsoft DirectX 8.1 (C++)

Matrix Concatenation

[This is preliminary documentation and is subject to change.]

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

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

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

Use the **D3DXMatrixMultiply** function to perform matrix multiplication.

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

$$W = R_y \cdot T_w$$

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

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

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

Microsoft DirectX 8.1 (C++)

World Transformation

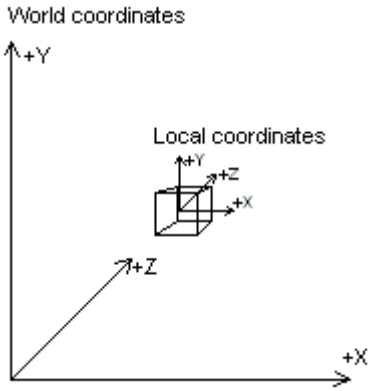
[This is preliminary documentation and is subject to change.]

The discussion of the world transformation introduces basic concepts and provides details on how to

set up a world transformation matrix in a Microsoft® Direct3D® application.

What Is a World Transformation?

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



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

Setting Up a World Matrix

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

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

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

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

    // Using the left-to-right order of matrix concatenation,
```

```
    // apply the translation to the object's world position
    // before applying the rotations.
    D3DXMatrixTranslation(pMatWorld, m_xPos, m_yPos, m_zPos);
    D3DXMatrixIdentity(&MatRot);

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

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

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

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

Microsoft DirectX 8.1 (C++)

View Transformation

[This is preliminary documentation and is subject to change.]

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

What Is a View Transformation?

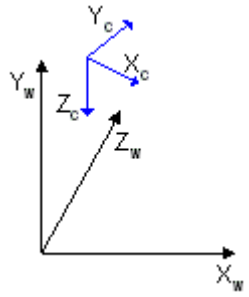
The view transformation locates the viewer in world space, transforming vertices into camera space. In camera space, the camera, or viewer, is at the origin, looking in the positive z-direction. Recall that Microsoft® Direct3D® uses a left-handed coordinate system, so z is positive into a scene. The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

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

applies.

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

In this formula, *V* is the view matrix being created, *T* is a translation matrix that repositions objects in the world, and R_x through R_z are rotation matrices that rotate objects along the x-, y-, and z-axis. The translation and rotation matrices are based on the camera's logical position and orientation in world space. So, if the camera's logical position in the world is <10,20,100>, the aim of the translation matrix is to move objects -10 units along the x-axis, -20 units along the y-axis, and -100 units along the z-axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how the much the axes of camera space are rotated out of alignment with world space. For example, if the camera mentioned earlier is pointing straight down, its z-axis is 90 degrees (pi/2 radians) out of alignment with the z-axis of world space, as shown in the following illustration.



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

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

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

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

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

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

    D3DXVECTOR3 XAxis, YAxis, ZAxis;

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

    // Normalize the z basis vector.
    D3DXVec3Normalize(&ZAxis, &ZAxis);

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

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

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

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

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

    return pOut;
```

```
}
```

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

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

Microsoft DirectX 8.1 (C++)

Projection Transformation

[This is preliminary documentation and is 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.

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

Microsoft DirectX 8.1 (C++)

What Is the Projection Transformation?

[This is preliminary documentation and is 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:

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

The viewing matrix puts the camera at the origin of the scene. Because the projection matrix needs to

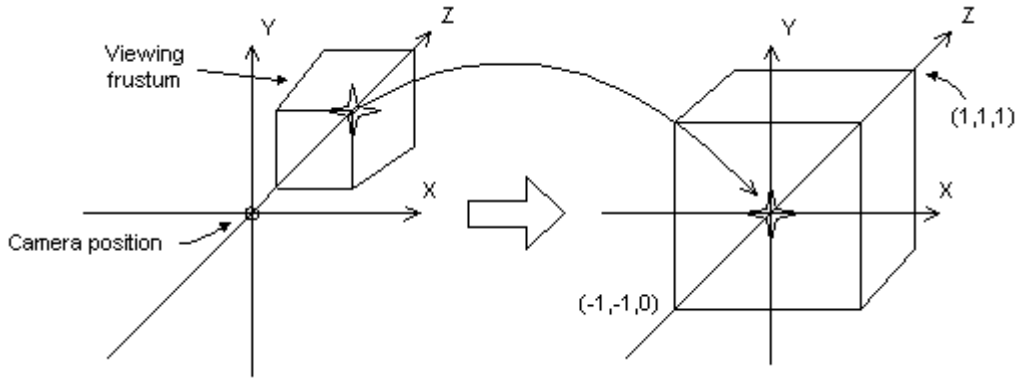
have the camera at (0, 0, -*D*), it translates the vector by -*D* in the z-direction, by using the following matrix.

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

Multiplying these two matrices gives the following composite matrix.

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

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



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

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

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

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

$$w = \cot\left(fov_w/2\right)$$
$$h = \cot\left(fov_h/2\right)$$
$$Q = \frac{Z_f}{Z_f - Z_n}$$

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

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

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

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

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

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

Microsoft DirectX 8.1 (C++)

Setting Up a Projection Matrix

[This is preliminary documentation and is subject to change.]

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

```
D3DXMATRIX
ProjectionMatrix(const float near_plane, // Distance to near clipping
                                         // plane
                 const float far_plane,  // Distance to far clipping
                                         // plane
                 const float fov_horiz,  // Horizontal field of view
                                         // angle, in radians
                 const float fov_vert)   // Vertical field of view
                                         // angle, in radians
```

```
{
    float    h, w, Q;

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

    D3DXMATRIX ret;
    ZeroMemory(&ret, sizeof(ret));

    ret(0, 0) = w;
    ret(1, 1) = h;
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
    ret(2, 3) = 1;
    return ret;
}   // End of ProjectionMatrix
```

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

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

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

Microsoft DirectX 8.1 (C++)

A W-Friendly Projection Matrix

[This is preliminary documentation and is subject to change.]

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

The following illustration shows a noncompliant 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 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.

Microsoft DirectX 8.1 (C++)

Viewports and Clipping

[This is preliminary documentation and is subject to change.]

Conceptually, a viewport is a two-dimensional (2-D) rectangle into which a three-dimensional (3-D) scene is projected. In Microsoft® Direct3D®, the rectangle exists as coordinates within a Direct3D surface that the system uses as a rendering target. The projection transformation converts vertices into the coordinate system used for the viewport.

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

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

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

- Viewport Rectangle
- The Viewing Frustum
- Clipping Volumes
- Viewport Scaling
- Using Viewports

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

Microsoft DirectX 8.1 (C++)

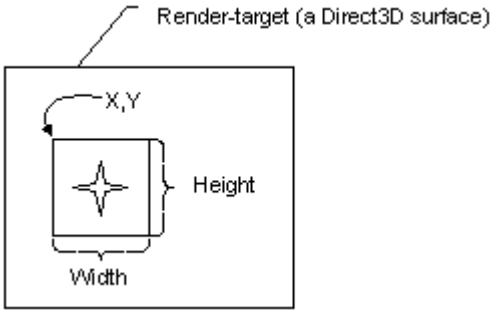
Viewport Rectangle

[This is preliminary documentation and is subject to change.]

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

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

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



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

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

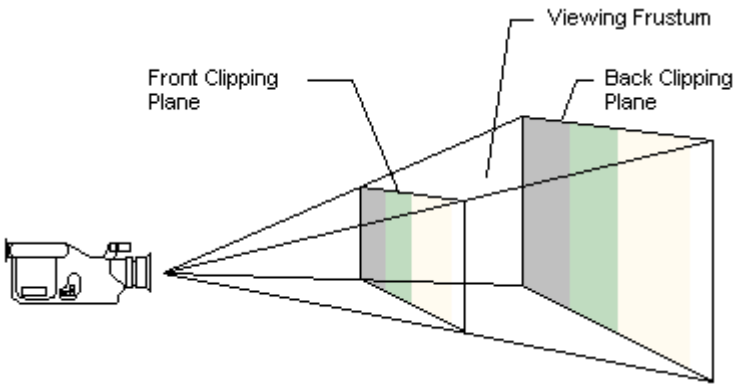
Note The **D3DVIEWPORT8** structure members **MinZ** and **MaxZ** indicate the depth-ranges into which the scene will be rendered and are not used for clipping. Most applications will set these members to 0.0 and 1.0 to enable the system to render to the entire range of depth values in the depth buffer.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

Microsoft DirectX 8.1 (C++)

The Viewing Frustum

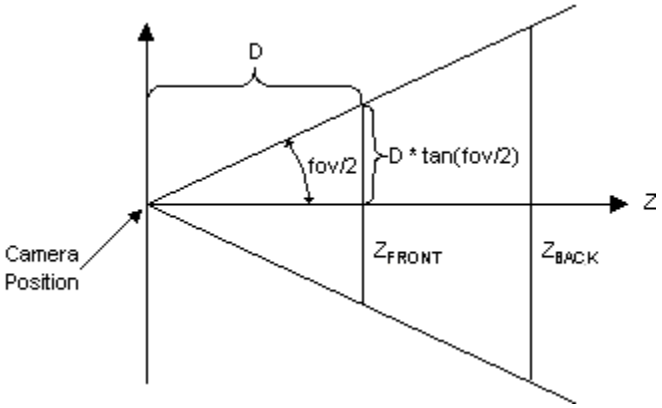
[This is preliminary documentation and is subject to change.]

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 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?

Microsoft DirectX 8.1 (C++)

Clipping Volumes

[This is preliminary documentation and is subject to change.]

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

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

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

With the exception of vertex buffers, applications enable or disable clipping by way of the D3DRS_CLIPPING render state. Clipping information for vertex buffers is generated during processing, for more information see Fixed Function Vertex Processing and Programmable Vertex Processing.

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

When the device receives pre-transformed and lit vertices (TL-Vertices) that need to be clipped, in order to perform the clipping operation, the vertices are back transformed to the clipping space using the vertex's rhw and the viewport information. Clipping is then performed. Not all devices are capable of performing this back-transform in order to clip TL-vertices.

The D3DPMISCCAPS_CLIPTLVERTS device capability indicates whether or not the device is capable of clipping TL-Vertices. If this capability is not set, then the application is responsible for clipping the TL-vertices that it intends to send down to the device to be rendered. The device is always capable of clipping TL-Vertices in the software vertex processing mode (either when the device itself is created in the software vertex processing mode or switched to the software vertex processing mode when created as a mixed mode vertex processing device using the D3DRS_SOFTWAREVERTEXPROCESSING renderstate).

Microsoft DirectX 8.1 (C++)

Viewport Scaling

[This is preliminary documentation and is subject to change.]

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

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

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

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

Note The viewport scaling matrix incorporates the **MinZ** and **MaxZ** members of the **D3DVIEWPORT8** structure to scale vertices to fit the depth range [**MinZ**, **MaxZ**]. This represents different semantics from previous releases of Microsoft DirectX, in which these members were used for clipping.

For more information, see Viewport Rectangle and Clipping Volumes. Applications typically set **MinZ** and **MaxZ** to 0.0 and 1.0 to cause the system to render to the entire depth range. However, you can use other values to achieve certain affects. You might set both values to 0.0 to force all objects into the foreground, or set both to 1.0 to render all objects into the background.

Microsoft DirectX 8.1 (C++)

Using Viewports

[This is preliminary documentation and is subject to change.]

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

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

Microsoft DirectX 8.1 (C++)

Setting the Viewport Clipping Volume

[This is preliminary documentation and is subject to change.]

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

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

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

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

```
HRESULT hr;

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

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

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

Microsoft DirectX 8.1 (C++)

Clearing a Viewport

[This is preliminary documentation and is subject to change.]

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

The **IDirect3DDevice8** interface offers C++ developers the **IDirect3DDevice8::Clear** method to clear the viewport. The method accepts one or more rectangles that define the areas on the surfaces being cleared. In cases where the scene being rendered includes motion throughout the entire viewport rectangle—in a first-person perspective game, for example—you might want to clear the entire viewport each frame. In this situation, you set the *Count* parameter to 1, and the *pRects* parameter to the address of a single rectangle that covers the entire viewport area. If it is more convenient, you can set the *pRects* parameter to NULL and the *Count* parameter to 0 to indicate that the entire viewport rectangle should be cleared.

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

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

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

Microsoft DirectX 8.1 (C++)

Manually Transforming Vertices

[This is preliminary documentation and is subject to change.]

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

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

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

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

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

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

Microsoft DirectX 8.1 (C++)

Lights and Materials

[This is preliminary documentation and is subject to change.]

Lights are used to illuminate objects in a scene. When lighting is enabled, Microsoft® Direct3D® calculates the color of each object vertex 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.
- The color and intensity of light produced by light sources in the scene or the scene's ambient light level.

When you use Direct3D lighting and materials, you allow Direct3D to handle the details of illumination for you. Advanced users can perform lighting on their own, if desired.

How you work with lighting and materials makes a big difference in the appearance of the rendered scene. Materials define how light reflects off a surface. Direct light and ambient light levels define the light that is reflected. You must use materials to render a scene if lighting is enabled. Lights are not required to render a scene, but details in a scene rendered without light are not visible. At best, rendering an unlit scene results in a silhouette of the objects in the scene. This is not enough detail for most purposes.

Additional information is contained in the following topics:

- Direct3D Light Model vs. Nature
- Color Values for Lights and Materials
- Direct Light vs. Ambient Light
- Light Properties
- Using Lights
- Mathematics of Lighting
- Materials

Microsoft DirectX 8.1 (C++)

Direct3D Light Model vs. Nature

[This is preliminary documentation and is subject to change.]

In nature, when light is emitted from a source, it is reflect off of hundreds, if not thousands or millions, of objects before reaching the user's eye. Each time it is reflected, some light is absorbed by a surface, some is scattered in random directions, and the rest goes on to another surface or to the user's eye. This process continues until the light is reduced to nothing or a user perceives the light.

Obviously, the calculations required to perfectly simulate the natural behavior of light are too time-consuming to use for real-time 3-D graphics. Therefore, with speed in mind, the Microsoft® Direct3D® light model approximates the way light works in the natural world. Direct3D describes light in terms of red, green, and blue components that combine to create a final color.

In Direct3D, when light reflects off a surface, the light color interacts mathematically with the surface itself to create the color eventually displayed on the screen. For specific information about the algorithms Direct3D uses, see Mathematics of Lighting.

The Direct3D light model generalizes light into two types: ambient light and direct light. Each has different attributes, and each interacts with the material of a surface in different ways. Ambient light is light that has been scattered so much that its direction and source are indeterminate: it maintains a low-level of intensity everywhere. The indirect lighting used by photographers is a good example of ambient light. Ambient light in Direct3D, as in nature, has no real direction or source, only a color and intensity. In fact, the ambient light level is completely independent of any objects in a scene that generate light. Ambient light does not contribute to specular reflection.

Direct light is the light generated by a source within a scene; it always has color and intensity, and it travels in a specified direction. Direct light interacts with the material of a surface to create specular highlights, and its direction is used as a factor in shading algorithms, including Gouraud shading. When direct light is reflected, it does not contribute to the ambient light level in a scene. The sources in a scene that generate direct light have different characteristics that affect how they illuminate a scene. For more information, see Lights and Materials.

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.

Microsoft DirectX 8.1 (C++)

Color Values for Lights and Materials

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® describes color in terms of four components—red, green, blue, and alpha—that combine to make a final color. The **D3DCOLORVALUE** C++ structure is defined to contain values for each component. Each member is a floating-point value that typically ranges from 0.0 to 1.0, inclusive. Although both lights and materials use the same structure to describe color, the values in the structure are used a little differently by each.

Color values for light sources represent the amount of a particular light component it emits. Because lights don't use an alpha component, only the red, green, and blue components of the color are relevant. You can visualize the three components as the red, green, and blue lenses on a projection television. Each lens might be off (a 0.0 value in the appropriate member), it might be as bright as possible (a 1.0 value), or it might be some level in between. The colors coming through the lenses combine to make the light's final color. A combination like R: 1.0, G: 1.0, B: 1.0 creates a white light, where R: 0.0, G: 0.0, B: 0.0 doesn't emit light at all. You can make a light that emits only one component, resulting in a pure red, green, or blue light, or, the light could use combinations to emit colors like yellow or purple. You can even set negative color component values to create a "dark light" that actually removes light from a scene. Or, you might set the components to some value larger than 1.0 to create an extremely bright light.

With materials, on the other hand, color values represent how much of a light component is reflected by a surface that is rendered with that material. A material whose color components are R: 1.0, G: 1.0, B: 1.0, A: 1.0 reflects all the light that comes its way. Likewise, a material with R: 0.0, G: 1.0, B: 0.0, A: 1.0 reflects all the green light that is directed at it. Materials have multiple reflectance values to create various types of effects; for more information, see Material Properties.

Color values for ambient light are different from those used for direct light sources and materials. For more information, see Direct Light vs. Ambient Light.

Microsoft DirectX 8.1 (C++)

Direct Light vs. Ambient Light

[This is preliminary documentation and is 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, and it is a factor for shading algorithms, such as Gouraud shading. Different types of lights emit direct light in different ways, creating special attenuation effects. You create a set of light parameters for direct light by calling the **IDirect3DDevice8::SetLight** method.

Ambient light is effectively everywhere in a scene. You can think of it as a general level of light that fills an entire scene, regardless of the objects and their locations in that scene. Ambient light has no position or direction, only color and intensity. Each light adds to the overall ambient light in a scene. Set the ambient light level with a call to the **IDirect3DDevice8::SetRenderState** method, specifying D3DRS_AMBIENT as the *State* parameter, and the desired RGBA color as the *Value* parameter.

Ambient light color takes the form of an RGBA value, where each component is an integer value from 0 to 255. This is unlike most color values in Microsoft® Direct3D®. For more information, see Color Values for Lights and Materials.

You can use the **D3DCOLOR_RGBA** macro to generate RGBA values. The red, green, and blue components combine to make the final color of the ambient light. The alpha component controls the transparency of the color. When using hardware acceleration or RGB emulation, the alpha component is ignored.

Microsoft DirectX 8.1 (C++)

Light Properties

[This is preliminary documentation and is subject to change.]

Light properties describe a light source's type and color. Depending on the type of light being used, a light can have properties for attenuation and range, or for spotlight effects. But, not all types of lights use all properties. Microsoft® Direct3D® uses the **D3DLIGHT8** structure to carry information about light properties for all types of light sources. This section contains information for all light properties. Information is divided into the following groups.

- Light Type
- Light Color
- Color Vertices
- Light Position, Range, and Attenuation
- Light Direction

Microsoft DirectX 8.1 (C++)

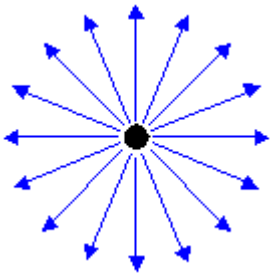
Light Type

[This is preliminary documentation and is subject to change.]

The light type property defines which type of light source you're using. The light type is set by using a value from the **D3DLIGHTTYPE** C++ enumeration in the **Type** member of the light's **D3DLIGHT8** structure. There are three types of lights in Microsoft® Direct3D®—point lights, spotlights, and directional lights. Each type illuminates objects in a scene differently, with varying levels of computational overhead.

Point Light

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



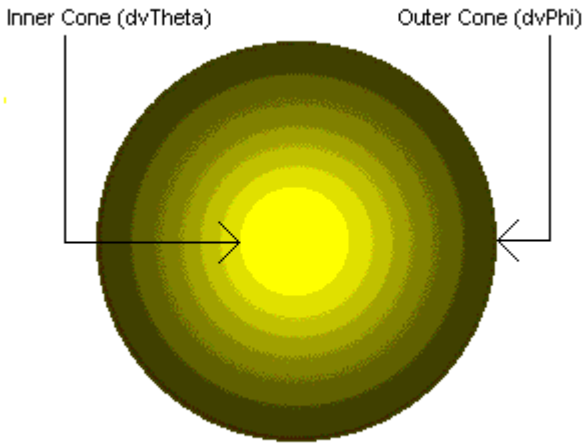
A light bulb is a good example of a point light. Point lights are affected by attenuation and range, and illuminate a mesh on a vertex-by-vertex basis. During lighting, Microsoft® Direct3D® uses the point light's position in world space and the coordinates of the vertex being lit to derive a vector for the direction of the light, and the distance that the light has traveled. Both are used, along with the vertex normal, to calculate the contribution of the light to the illumination of the surface.

Directional Light

Directional lights have only color and direction, not position. They emit parallel light. This means that all light generated by directional lights travels through a scene in the same direction. Imagine a directional light as a light source at near infinite distance, such as the sun. Directional lights are not affected by attenuation or range, so the direction and color you specify are the only factors considered when Microsoft® Direct3D® calculates vertex colors. Because of the small number of illumination factors, these are the least computationally intensive lights to use.

SpotLight

Spotlights have color, position, and direction in which they emit light. Light emitted from a spotlight is made up of a bright inner cone and a larger outer cone, with the light intensity diminishing between the two, as shown in the following illustration.

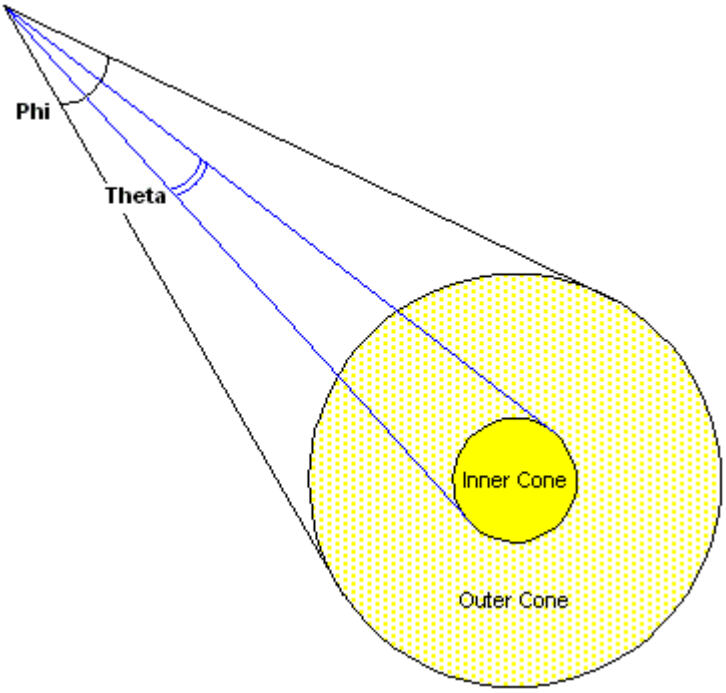


Spotlights are affected by falloff, attenuation, and range. These factors, as well as the distance light travels to each vertex, are figured in when computing lighting effects for objects in a scene. Computing these effects for each vertex makes spotlights the most computationally expensive of all lights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

The **D3DLIGHT8** C++ structure contains three members that are used only by spotlights. These members—**Falloff**, **Theta**, and **Phi**—control how large or small a spotlight object's inner and outer cones are, and how light decreases between them.

The **Theta** value is the radian angle of the spotlight's inner cone, and the **Phi** value is the angle for the outer cone of light. The **Falloff** value controls how light intensity decreases between the outer edge of the inner cone and the inner edge of the outer cone. Most applications set **Falloff** to 1.0 to create falloff that occurs evenly between the two cones, but you can set other values as needed.

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.



- Spotlight Model

  This page contains more information about how a spotlight works.

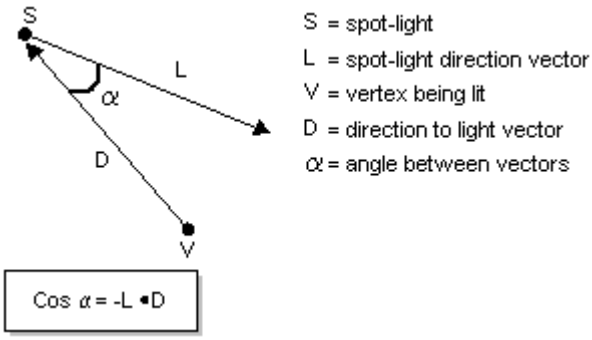
Microsoft DirectX 8.1 (C++)

Spotlight Model

[This is preliminary documentation and is subject to change.]

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

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

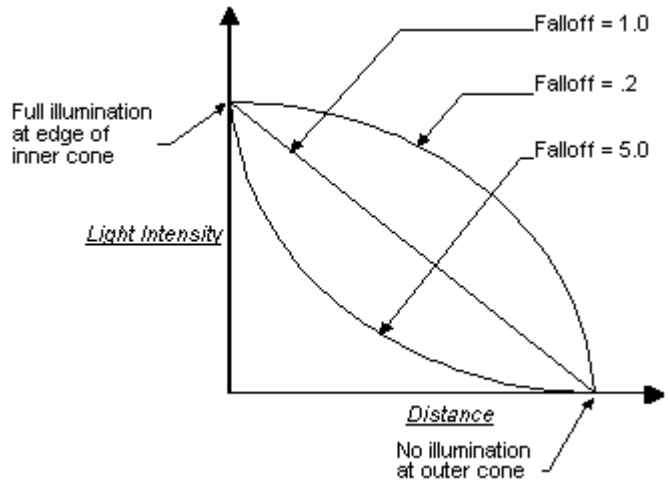


The system compares this value to the cosine of the spotlight's inner and outer cone angles. In the light's **D3DLIGHT8** structure, the **Theta** and **Phi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant from the center of illumination, rather than across the total cone angle, Direct3D divides these cone angles in half 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, then 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, a is the angle between vectors *L* and *D*, f is half of the outer cone angle, ? is half of the inner cone angle, and *p* is the spotlight's falloff property **Falloff** in the **D3DLIGHT8** structure. This formula generates a value between 0.0 and 1.0 that scales the light's intensity at the vertex to account for falloff. Attenuation as a factor of the vertex's distance from the light is also applied. The *p* value corresponds to the **Falloff** member of the **D3DLIGHT8** structure and controls the shape of the falloff curve. The following illustration shows how different **Falloff** values can affect the falloff curve.



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

Microsoft DirectX 8.1 (C++)

Light Color

[This is preliminary documentation and is subject to change.]

Lights in Microsoft® Direct3D® emit three colors that are used independently in the system's lighting computations: a diffuse color, an ambient color, and a specular color. Each is incorporated by the Direct3D lighting module, interacting with a counterpart from the current material, to produce a final color used in rendering. The diffuse color interacts with the diffuse reflectance property of the current material, the specular color with the material's specular reflectance property, and so on. For specifics about how Direct3D applies these colors, see Mathematics of Lighting.

In a C++ application, the **D3DLIGHT8** structure includes three members for these colors—**Diffuse**, **Ambient**, and **Specular**—each one is a **D3DCOLORVALUE** structure that defines the color being emitted.

The type of color that applies most heavily to the system's computations is the diffuse color. The most common diffuse color is white (R:1.0 G:1.0 B:1.0), but you can create colors as needed to achieve desired effects. For example, you could use red light for a fireplace, or you could use green light for a traffic signal set to "Go."

Generally, you set the light color components to values between 0.0 and 1.0, inclusive, but this isn't a requirement. For example, you might set all the components to 2.0, creating a light that is "brighter than white." This type of setting can be especially useful when you use attenuation settings other than constant.

Note that although Direct3D uses RGBA values for lights, the alpha color component is not used. For more information, see Color Values for Lights and Materials.

Microsoft DirectX 8.1 (C++)

Color Vertices

[This is preliminary documentation and is subject to change.]

Usually material colors are used for lighting. However, you can specify that material colors—emissive, ambient, diffuse, and specular—are to be overridden by diffuse or specular vertex colors. This is done by calling **IDirect3DDevice8::SetRenderState** and setting the device state variables listed in the following table.

Device state variable	Meaning	Type	Defaul
D3DRS_AMBIENTMATERIALSOURCE	Defines where to get ambient material color.	D3DMATERIALCOLORSOURCE	D3DM
D3DRS_DIFFUSEMATERIALSOURCE	Defines where to get	D3DMATERIALCOLORSOURCE	D3DM

	diffuse material color.		
D3DRS_SPECULARMATERIALSOURCE	Defines where to get specular material color.	D3DMATERIALCOLORSOURCE	D3DM
D3DRS_EMISSIVEMATERIALSOURCE	Defines where to get emissive material color.	D3DMATERIALCOLORSOURCE	D3DM
D3DRS_COLORVERTEX	Disables or enables use of vertex colors.	BOOL	TRUE

The alpha/transparency value always comes only from the diffuse color's alpha channel.

The fog value always comes only from the specular color's alpha channel.

D3DMATERIALCOLORSOURCE can have the following values.

- D3DMCS_MATERIAL - Material color is used as source.
- D3DMCS_COLOR1 - Diffuse vertex color is used as source.
- D3DMCS_COLOR2 - Specular vertex color is used as source.

Microsoft DirectX 8.1 (C++)

Light Position, Range, and Attenuation

[This is preliminary documentation and is subject to change.]

The position, range, and attenuation properties define a light's location in world space and how the light it emits behaves over distance. As with all light properties you use in C++, these are contained in a light's **D3DLIGHT8** structure.

Position

Light position is described using a **D3DVECTOR** structure in the **Position** member of the **D3DLIGHT8** structure. The x-, y-, and z-coordinates are assumed to be in world space. Directional lights are the only type of light that don't use the position property.

Range

A light's range property determines the distance, in world space, at which meshes in a scene no longer receive light emitted by that object. The **Range** member contains a floating-point value that

represents the light's maximum range, in world space. Directional lights don't use the range property.

Attenuation

Attenuation controls how a light's intensity decreases toward the maximum distance specified by the range property. Three **D3DLIGHT8** structure members represent light attenuation: **Attenuation0**, **Attenuation1**, and **Attenuation2**. These members contain floating-point values ranging from 0.0 through infinity, controlling a light's attenuation. Some applications set the **Attenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that changes as 1 / *D*, where *D* is the distance from the light source to the vertex. The maximum light intensity is at the source, decreasing to 1 / (*Light Range*) at the light's range. Typically, an application sets **Attenuation0** to 0.0, **Attenuation1** to a constant value, and **Attenuation2** to 0.0.

You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside the normal range to create even stranger attenuation effects. Negative attenuation values, however, are not allowed.

Microsoft DirectX 8.1 (C++)

Light Direction

[This is preliminary documentation and is subject to change.]

A light's direction property determines the direction that the light emitted by the object travels, in world space. Direction is used only by directional lights and spotlights, and is described with a vector.

C++ applications set the light direction in the **Direction** member of the light's **D3DLIGHT8** structure. **Direction** member is of type **D3DVECTOR**. Direction vectors are described as distances from a logical origin, regardless of the light's position in a scene. Therefore, a spotlight that points straight into a scene—along the positive z-axis—has a direction vector of <0,0,1> no matter where its position is defined to be. Similarly, you can simulate sunlight shining directly on a scene by using a directional light whose direction is <0,-1,0>. Obviously, you don't have to create lights that shine along the coordinate axes; you can mix and match values to create lights that shine at more interesting angles.

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.

Microsoft DirectX 8.1 (C++)

Using Lights

[This is preliminary documentation and is subject to change.]

Setting and Retrieving Light Properties

You set lighting properties in a C++ application by preparing a **D3DLIGHT8** structure and then calling the **IDirect3DDevice8::SetLight** method. The **SetLight** method accepts the index at which the device should place the set of light properties to its internal list of light properties, and the address of a prepared **D3DLIGHT8** structure that defines those properties. You can call **SetLight**

with new information as needed to update the light's illumination properties.

The system allocates memory to accommodate a set of lighting properties each time you call the **SetLight** method with an index that has never been assigned properties. Applications can set a number of lights, with only a subset of the assigned lights enabled at a time. Check the **MaxActiveLights** member of the **D3DCAPS8** structure when you retrieve device capabilities to determine the maximum number of active lights supported by that device. If you no longer need a light, you can disable it or overwrite it with a new set of light properties.

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

```
/*
 * For the purposes of this example, the d3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
D3DLIGHT8 d3dLight;
HRESULT   hr;

// Initialize the structure.
ZeroMemory(&d3dLight, sizeof(D3DLIGHT8));

// Set up a white point light.
d3dLight.Type = D3DLIGHT_POINT;
d3dLight.Diffuse.r  = 1.0f;
d3dLight.Diffuse.g  = 1.0f;
d3dLight.Diffuse.b  = 1.0f;
d3dLight.Ambient.r  = 1.0f;
d3dLight.Ambient.g  = 1.0f;
d3dLight.Ambient.b  = 1.0f;
d3dLight.Specular.r = 1.0f;
d3dLight.Specular.g = 1.0f;
d3dLight.Specular.b = 1.0f;

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

// Don't attenuate.
d3dLight.Attenuation0 = 1.0f;
d3dLight.Range        = 1000.0f;

// Set the property information for the first light.
hr = d3dDevice->SetLight(0, &d3dLight);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

You can update a set of light properties with another call to **SetLight** at any time. Just specify the index of the set of light properties to update and the address of the **D3DLIGHT8** structure that contains the new properties.

Note Assigning a set of light properties to the device does not enable the light source whose properties are being added. Enable a light source by calling the **IDirect3DDevice8::LightEnable**

method for the device.

You can retrieve all the properties for an existing light source from C++ by calling the **IDirect3DDevice8::GetLight** method for the device. When calling the **GetLight** method, pass in the first parameter the zero-based index of the light source for which the properties will be retrieved, and supply the address of a **D3DLIGHT8** structure in the second parameter. The device fills the **D3DLIGHT8** structure to describe the lighting properties it uses for the light source at that index.

The following code example illustrates this process.

```
/*
 * For the purposes of this example, the pd3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;
D3DLIGHT8 light;

// Get the property information for the first light.
hr = pd3dDevice->GetLight(0, &light);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

If you supply an index outside the range of the light sources assigned in the device, the **GetLight** method fails, returning D3DERR_INVALIDCALL.

Enabling and Disabling Lights

Once you assign a set of light properties for a light source in a scene, the light source can be activated by calling the **IDirect3DDevice8::LightEnable** method for the device. New light sources are disabled by default. The **LightEnable** method accepts two parameters. Set the first parameter to the zero-based index of the light source to be affected by the method, and set the second parameter to TRUE to enable the light or FALSE to disable it.

The following code example illustrates the use of this method by enabling the first light source in the device's list of light source properties.

```
/*
 * For the purposes of this example, the d3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;

hr = pd3dDevice->LightEnable(0, TRUE);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

Check the **MaxActiveLights** member of the **D3DCAPS8** structure when you retrieve device capabilities to determine the maximum number of active lights supported by that device.

If you enable or disable a light that has no properties that are set with **IDirect3DDevice8::SetLight**, the **LightEnable** method creates a light source with the properties listed in following table and enables or disables it.

Member	Default
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

Microsoft DirectX 8.1

Mathematics of Lighting

[This is preliminary documentation and is subject to change.]

The Microsoft® Direct3D® Light Model covers ambient, diffuse, specular, and emissive lighting. This is enough flexibility to solve a wide range of lighting situations. You refer to the total amount of light in a scene as the *global illumination* and compute it using the following equation.

```
Global Illumination = Ambient Light + Diffuse Light + Specular Light + Emissive L
```

Ambient lighting is constant lighting. It is constant in all directions and it colors all pixels of an object the same. It is fast to calculate but leaves objects looking flat and unrealistic. To see how ambient lighting is calculated by Direct3D, see Ambient Lighting.

Diffuse lighting depends on both the light direction and the object surface normal. It varies across the surface of an object as a result of the changing light direction and the changing surface numeral vector. It takes longer to calculate diffuse lighting because it changes for each object vertex, however the benefit of using it is that it shades objects and gives them three-dimensional (3-D) depth. To see how diffuse lighting is calculated in Direct3D, see Diffuse Lighting.

Specular lighting identifies the bright specular highlights that occur when light hits an object surface and reflects back toward the camera. It is more intense than diffuse light and falls off more rapidly across the object surface. It takes longer to calculate specular lighting than diffuse lighting, however the benefit of using it is that it adds significant detail to a surface. To see how specular lighting is calculated in Direct3D, see Specular Lighting.

Emissive lighting is light that is emitted by an object, for example, a glow. To see how emissive lighting is calculated in Direct3D, see Emissive Lighting.

Realistic lighting can be accomplished by applying each of these types of lighting to a 3-D scene. To achieve a more realistic lighting effect, you add more lights; however, the scene takes longer to render. To achieve all the effects a designer wants, some games use more CPU power than is commonly available. In this case, it is typical to reduce the number of lighting calculations to a

minimum by using lighting maps and environment maps to add lighting to a scene while using texture maps.

All lighting computations are made in model space by transforming the light source's position and direction, along with the camera position, to model space using the inverse of the world matrix. As a result, if the world or view matrices introduce non-uniform scaling, the resultant lighting might be inaccurate. To see how lighting transformations are calculated, see Camera Space Transformations.

Diffuse and specular light values can be affected by a given light's attenuation and spotlight characteristics. Terms for both of these are included in the diffuse and specular equations. For more information, see Attenuation and Spotlight Terms.

Microsoft DirectX 8.1

Ambient Lighting

[This is preliminary documentation and is subject to change.]

Ambient lighting provides constant lighting for a scene. It lights all object vertices the same because it is not dependent on any other lighting factors such as vertex normals, light direction, light position, range, or attenuation. It is the fastest type of lighting but it produces the least realistic results. Microsoft® Direct3D® contains a single global ambient light property that you can use without creating any light. Alternatively, you can set any light object to provide ambient lighting. The ambient lighting for a scene is described by the following equation.

$$\text{Ambient Lighting} = M_c * [G_a + \text{sum}(L_{ai})]$$

The parameters are defined in the following table.

Parameter	Default value	Type	Description
M_c	(0,0,0,0)	D3DCOLORVALUE	Material ambient color.
G_a	(0,0,0,0)	D3DCOLORVALUE	Global ambient color.
L_{ai}	(0,0,0,0)	D3DVECTOR	Light ambient color, of the ith light.
sum	N/A	N/A	Summation of the ambient light from the light objects.

The value for M_c is one of three values: one of the two possible vertex colors in a vertex declaration, or the material ambient color. The value is:

- vertex color1, if AMBIENTMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if AMBIENTMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in vertex declaration.
- material ambient color

Note If either AMBIENTMATERIALSOURCE option is used, and the vertex color is not provided, then the material ambient color is used.

To use the material ambient color, use SetMaterial as shown in the example code below.

G_a is the global ambient color. It is set using SetRenderState(D3DRENDERSTATE_AMBIENT). There is one global ambient color in a Direct3D scene. This parameter is not associated with a Direct3D light object.

L_{ai} is the ambient color of the ith light in the scene. Each Direct3D light has a set of properties, one of which is the ambient color. The term, sum(L_{ai}) is a sum of all the ambient colors in the scene.

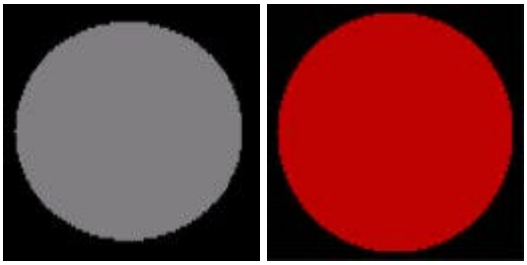
Example

In this example, the object is colored using the scene ambient light and a material ambient color. The code is shown below.

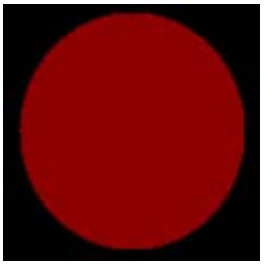
```
#define GRAY_COLOR                      0x00bfbfbf

// create material
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Ambient.r = 0.75f;
mtrl.Ambient.g = 0.0f;
mtrl.Ambient.b = 0.0f;
mtrl.Ambient.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, GRAY_COLOR);
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is bright red.



The resulting scene is shown below. The only object in the scene is a sphere. Ambient light lights all object vertices with the same color. It is not dependent on the vertex normal or the light direction. As a result, the sphere looks like a 2-D circle because there is no difference in shading around the surface of the object.



To give objects a more realistic look, apply diffuse or specular lighting in addition to ambient lighting.

Microsoft DirectX 8.1

Diffuse Lighting

[This is preliminary documentation and is subject to change.]

After adjusting the light intensity for any attenuation effects, Microsoft® Direct3D® computes how much of the remaining light reflects from a vertex, given the angle of the vertex normal and the direction of the incident light. Direct3D skips to this step for directional lights because they do not attenuate over distance. The system considers two reflection types, diffuse and specular, and uses a different formula to determine how much light is reflected for each. After calculating the amounts of light reflected, Direct3D applies these new values to the diffuse and specular reflectance properties of the current material. The resulting color values are the diffuse and specular components that the rasterizer uses to produce Gouraud shading and specular highlighting.

Diffuse lighting is described by the following equation.

$$\text{Diffuse Lighting} = \text{sum}[V_d * L_d * (N \cdot L_{dir}) * \text{Atten} * \text{Spot}]$$

The parameters are defined in the following table.

Parameter	Default value	Type	Description
sum	N/A	N/A	Summation of each light's diffuse component.
V_d	(0,0,0,0)	D3DCOLORVALUE	Vertex diffuse color.
L_d	(0,0,0,0)	D3DCOLORVALUE	Light diffuse color.
N	N/A	D3DVECTOR	Vertex normal.
L_{dir}	(0,0,0,0)	D3DCOLORVALUE	Direction vector from object vertex to the light.
Atten	(0,0,0,0)	D3DCOLORVALUE	Light attenuation.
Spot	(0,0,0,0)	D3DVECTOR	Characteristics of the spotlight cone.

The value for V_d is one of three values: one of the two possible vertex colors in a vertex declaration, or the material diffuse color. The value is:

- vertex color1, if DIFFUSEMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if DIFFUSEMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material diffuse color

Note: If either DIFFUSEMATERIALSOURCE option is used, and the vertex color is not provided, the material diffuse color is used.

To calculate the attenuation (Atten) or the spotlight characteristics (Spot), see Attenuation and Spotlight Terms

Diffuse components are clamped to be from 0 to 255, after all lights are processed and interpolated separately. The resulting diffuse lighting value is a combination of the ambient, diffuse and emissive light values.

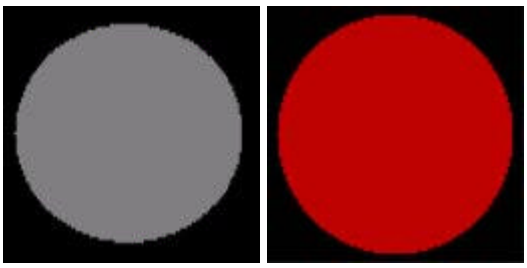
Example

In this example, the object is colored using the light diffuse color and a material diffuse color. The code is shown below.

```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );

D3DLIGHT8 light;
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;

D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );

// set directional light diffuse color
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Diffuse.a = 1.0f;
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );

// if a material is used, SetRenderState must be used
// vertex color = light diffuse color * material diffuse color
mtrl.Diffuse.r = 0.75f;
mtrl.Diffuse.g = 0.0f;
mtrl.Diffuse.b = 0.0f;
mtrl.Diffuse.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is bright red.



The resulting scene is shown below. The only object in the scene is a sphere. The diffuse lighting calculation takes the material and light diffuse color and modifies it by the angle between the light direction and the vertex normal using the dot product. As a result, the backside of the sphere gets darker as the surface of the sphere curves away from the light.

Combining the diffuse lighting with the ambient lighting from the previous example shades the entire surface of the object. The ambient light shades the entire surface and the diffuse light helps reveal the object's three-dimensional (3-D) shape.

Diffuse lighting is more intensive to calculate than ambient lighting. Because it depends on the vertex normals and light direction, you can see the objects geometry in 3-D space, which produces a more realistic lighting than ambient lighting. You can use specular highlights to achieve a more realistic look.

Microsoft DirectX 8.1

Specular Lighting

[This is preliminary documentation and is 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. The system uses a simplified version of the Phong specular-reflection model, which employs a halfway vector to approximate the intensity of specular reflection.

The default lighting state does not calculate specular highlights. To enable specular lighting, be sure to set the **D3DRS_SPECULARENABLE** to TRUE.

Specular Lighting is described by the following equation.

$$\text{Specular Lighting} = V_s * \text{sum}[L_s * (N \cdot H)^P * \text{Atten} * \text{Spot}]$$

The parameters are defined in the following table.

Parameter	Default value	Type	Description
V_s	(0,0,0,0)	D3DCOLORVALUE	Vertex specular color.
sum	N/A	N/A	Summation of each light's specular component.
N	N/A	D3DVECTOR	Vertex normal.
H	(0,0,0,0)	D3DCOLORVALUE	Half way vector.

P	(0,0,0,0)	D3DCOLORVALUE	Specular reflection power. Range is -infinity to +infinity
L_s	(0,0,0,0)	D3DCOLORVALUE	Light specular color.
Atten	(0,0,0,0)	D3DCOLORVALUE	Light attenuation.
Spot	(0,0,0,0)	D3DVECTOR	Characteristics of the spotlight cone.

The value for V_c is one of three values: one of the two possible vertex colors in a vertex declaration, or the material specular color. The value is:

- vertex color1, if SPECULARMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if SPECULARMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material specular color

Note: If either SPECULARMATERIALSOURCE option is used, and the vertex color is not provided, then the material specular color is used.

To calculate the attenuation (Atten) or the spotlight characteristics (Spot), see Attenuation and Spotlight Terms

The halfway vector (H) exists midway between the vector from an object vertex to the light source and the vector from an object vertex and the camera position. Microsoft® Direct3D® provides two ways to compute the halfway vector. When D3DRS_LOCALVIEWER is set to TRUE, the system calculates the halfway vector using the position of the camera and the position of the vertex, along with the light's direction vector. The following formula illustrates this.

$H = norm(norm(C_p - V_p) + L_{dir})$ where the parameters are defined in the following table:

Parameter	Default value	Type	Description
C_p	(0,0,0,0)	D3DVECTOR	Camera position.
V_p	(0,0,0,0)	D3DVECTOR	Vertex position.
L_{dir}	(0,0,0,0)	D3DVECTOR	Direction vector from vertex position to the light position.

When **D3DRS_LOCALVIEWER** is set to TRUE, Direct3D determines the halfway vector by the following formula.

$H = norm(norm(- V_p) + L_{dir})$

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

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

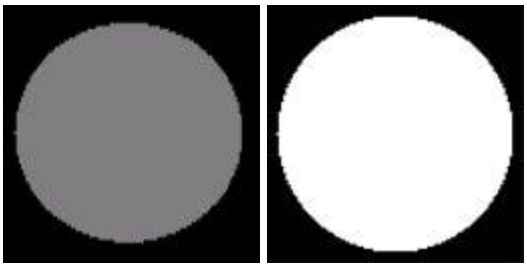
Example

In this example, the object is colored using the scene specular light color and a material specular color. The code is shown below.

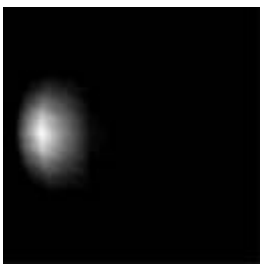
```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );

D3DLIGHT8 light;
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;

D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );

light.Specular.r = 1.0f;
light.Specular.g = 1.0f;
light.Specular.b = 1.0f;
light.Specular.a = 1.0f;

light.Range = 1000;
light.Falloff = 0;
light.Attenuation0 = 1;
light.Attenuation1 = 0;
light.Attenuation2 = 0;
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_SPECULARENABLE, TRUE );

mtrl.Specular.r = 1.0f;
mtrl.Specular.g = 1.0f;
mtrl.Specular.b = 1.0f;
mtrl.Specular.a = 1.0f;
mtrl.Power = 20;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_SPECULARMATERIALSOURCE, D3DMCS_MATERIAL);
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is white.



The resulting specular highlight is shown below.



Combining the specular highlight with the ambient and diffuse lighting produces the following image. With all three types of lighting applied, this more clearly resembles a realistic object.

Specular lighting is more intensive to calculate than diffuse lighting. It is typically used to provide visual clues about the surface material. The specular highlight varies in size and color with the material of the surface.

Microsoft DirectX 8.1

Emissive Lighting

[This is preliminary documentation and is subject to change.]

Emissive lighting is described by a single term.

```
Emissive Lighting = M_e
```

The parameter is defined in the following table.

Parameter	Default value	Type	Description
M_e	(0,0,0,0)	D3DCOLORVALUE	Material emissive color.

The value for M_e is one of three values: one of the two possible vertex colors in a vertex declaration, or the material emissive color. The value is:

- vertex color1, if EMISSIVEMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if EMISSIVEMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material emissive color

Note If either EMISSIVEMATERIALSOURCE option is used, and the vertex color is not provided, the material emissive color is used.

Example

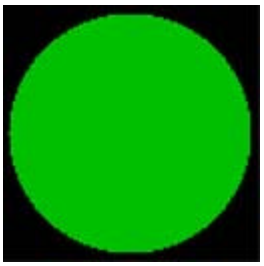
In this example, the object is colored using the scene ambient light and a material ambient color. The code is shown below.

```
// create material
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Emissive.r = 0.0f;
```

```
mtrl.Emissive.g = 0.75f;
mtrl.Emissive.b = 0.0f;
mtrl.Emissive.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_EMISSIVEMATERIALSOURCE, D3DMCS_MATERIAL);
```

According to the equation, the resulting color for the object vertices is the material color.

The image below shows the material color, which is green. Emissive light lights all object vertices with the same color. It is not dependent on the vertex normal or the light direction. As a result, the sphere looks like a 2-D circle because there is no difference in shading around the surface of the object.



This image shows how the emissive light blends with the other three types of lights, from the previous examples. On the right side of the sphere, there is a blend of the green emissive and the red ambient light. On the left side of the sphere, the green emissive light blends with red ambient and diffuse light producing a red gradient. The specular highlight is white in the center and creates a yellow ring as the specular light value falls off sharply leaving the ambient, diffuse and emissive light values which blend together to make yellow.

Microsoft DirectX 8.1 (C++)

Camera Space Transformations

[This is preliminary documentation and is subject to change.]

Vertices in the camera space are computed by transforming the object vertices with the world view matrix.

V = V * wvMatrix

Normals in camera space are computed by transforming the object normals with the inverse transpose of the world view matrix, then normalizing the result. This uses only a 3×3 portion of the 4×4 matrix.

$N = norm([N * wvMatrix^{-1}]^{T})$

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

N = norm(N)

Light position in camera space is computed by transforming the light source position with the view matrix.

Lp = Lp * vMatrix

The direction to the light in camera space for a directional light, is computed by multiplying the light source direction by the view matrix, normalizing, and negating the result.

L_{dir} = -norm(L_{dir} * vMatrix)

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

L_{dir} = norm(L_{dir}), where the parameters are defined in the following table.

Parameter	Default value	Type	Description
L_{dir}	(0,0,0,0)	D3DCOLORVALUE	Direction vector from object vertex to the light
V	(0,0,0,0)	D3DVECTOR	Vertex position
wvMatrix	Identity	D3DMATRIX	Composite matrix containing the world and view transforms
N	(0,0,0,0)	D3DVECTOR	Vertex normal
Lp	(0,0,0,0)	D3DVECTOR	Light position
vMatrix	Identity	D3DMATRIX	Matrix containing the view transform

Microsoft DirectX 8.1 (C++)

Attenuation and Spotlight Terms

[This is preliminary documentation and is subject to change.]

The diffuse and specular lighting components of the global illumination equation contain terms that describe light attenuation and the spotlight cone. These terms are described below.

Attenuation Term

The attenuation of a light depends on the type of light and the distance between the light and the vertex position. To calculate attenuation, use one of the following three equations.

- Atten = 1, if the light is a directional light.
- Atten = 0, if the distance between the light and the vertex exceeds the light's range.
- Atten = 1/($att0_i$ + $att1_i$*d + $att2_i$*di^2).

Parameter	Default value	Type	Description
att0i	(0,0,0,0)	FLOAT	Linear attenuation factor
att1i	(0,0,0,0)	FLOAT	Squared attenuation factor
att2i	(0,0,0,0)	FLOAT	Exponential attenuation factor
di	(0,0,0,0)	FLOAT	Distance from vertex position to light position

The att0, att1, att2 values are specified by the **Attenuation0**, **Attenuation1**, and **Attenuation2** members of the **D3DLIGHT8** structure.

```
The distance between the light and the vertex position is always positive.
di = ||L_dir||
where:
```

Parameter	Default value	Type	Description
L_{dir}	0.0	D3DVECTOR	Direction vector from vertex position to the light position

If *di* is greater than the light's range, that is, the **Range** member of a **D3DLIGHT8** structure, Direct3D makes no further attenuation calculations and applies no effects from the light to the vertex. The *dvAttenuation0*, *dvAttenuation1*, and *dvAttenuation2* values are the light's attenuation constants as specified by the members of a light object's **D3DLIGHT8** structure. The corresponding structure members are **Attenuation0**, **Attenuation1**, and **Attenuation2**.

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

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

Spotlight Term

$$spot_i = \begin{cases} 1 & \text{for non- spotlights or if } rho_i > \cos(\frac{theta_i}{2}) \\ 0 & \text{if } rho_i \le \cos(\frac{phi_i}{2}) \\ \left[\frac{rho_i - \cos(\frac{phi_i}{2})}{\cos(\frac{theta_i}{2}) - \cos(\frac{phi_i}{2})}\right]^{falloff} & \text{otherwise} \end{cases}$$

Parameter	Default value	Type	Description
rho	0.0	N/A	Angle

phi	0.0	FLOAT	Penumbra angle of spotlight in radians. Range: [thetaI, p)
theta	0.0	FLOAT	Umbra angle of spotlight in radians. Range: [0, p)
falloff	0.0	FLOAT	Falloff factor. Range: (-infinity, +infinity)

where:

$rho = norm(L_{dcs}) \cdot norm(L_{dir})$

Parameter	Default value	Type	Description
L_{dcs}	0.0	D3DVECTOR	Direction vector from origin to the light position in camera space
L_{dir}	0.0	D3DVECTOR	Direction vector from vertex position to the light position

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

Microsoft DirectX 8.1 (C++)

Materials

[This is preliminary documentation and is 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 Microsoft® Direct3D® the following things about the polygons it is rendering.

- How they reflect ambient and diffuse light
- What their specular highlights look like
- Whether the polygons appear to emit light

Direct3D applications written in C++ use the **D3DMATERIAL8** structure to describe material properties. For more information, see Material Properties.

Setting Material Properties

Microsoft® Direct3D® rendering devices can render with one set of material properties at a time.

In a C++ application, you set the material properties that the system uses by preparing a **D3DMATERIAL8** structure, and then calling the **IDirect3DDevice8::SetMaterial** method.

To prepare the **D3DMATERIAL8** structure for use, set the property information in the structure to create the desired effect during rendering. The following code example sets up the **D3DMATERIAL8** structure for a purple material with sharp white specular highlights.

```
D3DMATERIAL8 mat;

// Set the RGBA for diffuse reflection.
mat.Diffuse.r = 0.5f;
mat.Diffuse.g = 0.0f;
mat.Diffuse.b = 0.5f;
mat.Diffuse.a = 1.0f;

// Set the RGBA for ambient reflection.
mat.Ambient.r = 0.5f;
mat.Ambient.g = 0.0f;
mat.Ambient.b = 0.5f;
mat.Ambient.a = 1.0f;

// Set the color and sharpness of specular highlights.
mat.Specular.r = 1.0f;
mat.Specular.g = 1.0f;
mat.Specular.b = 1.0f;
mat.Specular.a = 1.0f;
mat.Power = 50.0f;

// Set the RGBA for emissive color.
mat.Emissive.r = 0.0f;
mat.Emissive.g = 0.0f;
mat.Emissive.b = 0.0f;
mat.Emissive.a = 0.0f;
```

After preparing the **D3DMATERIAL8** structure, you apply the properties by calling the **SetMaterial** method of the rendering device. This method accepts the address of a prepared **D3DMATERIAL8** structure as its only parameter. You can call **SetMaterial** with new information as needed to update the material properties for the device. The following code example shows how this might look in code.

```
// This code example uses the material properties defined for
// the mat variable earlier in this topic. The pd3dDev is assumed
// to be a valid pointer to an IDirect3DDevice8 interface.
HRESULT hr;
hr = pd3dDev->SetMaterial(&mat);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

When you create a Direct3D device, the current material is automatically set to the default shown in the following table.

Member	Value
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Emissive	(R:0, G:0, B:0, A:0)
Power	(0.0)

Retrieving Material Properties

You retrieve the material properties that the rendering device is currently using by calling the **IDirect3DDevice8::GetMaterial** method for the device. Unlike the **IDirect3DDevice8::SetMaterial** method, **GetMaterial** doesn't require preparation. The

GetMaterial method accepts the address of a **D3DMATERIAL8** structure, and fills the provided structure with information describing the current material properties before returning.

```
// For this example, the pd3dDev variable is assumed to
// be a valid pointer to an IDirect3DDevice8 interface.
HRESULT hr;
D3DMATERIAL8 mat;

hr = pd3dDev->GetMaterial(&mat);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

For more information about material properties see:

- Material Properties

Default Material Properties Note If your application does not specify material properties for rendering, the system uses a default material. The default material reflects all diffuse light—white, for example—with no ambient or specular reflection, and no emissive color.

Microsoft DirectX 8.1 (C++)

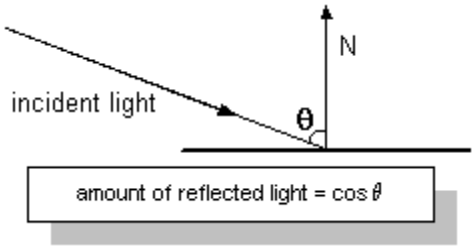
Material Properties

[This is preliminary documentation and is subject to change.]

Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlight characteristics. Microsoft® Direct3D® uses the **D3DMATERIAL8** structure to carry all material property information. Material properties affect the colors that Direct3D uses to rasterize polygons that use the material. With the exception of the specular property, each property is described as an RGBA color that represents how much of the red, green, and blue parts of a given type of light it reflects, and an alpha blending factor—the alpha component of the RGBA color. The material's specular property is described in two parts: color and power. For more information, see Color Values for Lights and Materials.

Diffuse and Ambient Reflection

The **Diffuse** and **Ambient** members of the **D3DMATERIAL8** structure describe how a material reflects the ambient and diffuse light in a scene. Because most scenes contain much more diffuse light than ambient light, diffuse reflection plays the largest part in determining color. Additionally, because diffuse light is directional, the angle of incidence for diffuse light affects the overall intensity of the reflection. Diffuse reflection is greatest when the light strikes a vertex parallel to the vertex normal. As the angle increases, the effect of diffuse reflection diminishes. The amount of light reflected is the cosine of the angle between the incoming light and the vertex normal, as shown here.



Ambient reflection, like ambient light, is nondirectional. Ambient reflection has a lesser impact on the apparent color of a rendered object, but it does affect the overall color and is most noticeable when little or no diffuse light reflects off the material. A material's ambient reflection is affected by the ambient light set for a scene by calling the **IDirect3DDevice8::SetRenderState** method with the D3DRS_AMBIENT flag.

Diffuse and ambient reflection work together to determine the perceived color of an object, and are usually identical values. For example, to render a blue crystalline object, you create a material that reflects only the blue component of diffuse and ambient light. When placed in a room with a white light, the crystal appears to be blue. However, in a room that has only red light, the same crystal would appear to be black, because its material doesn't reflect red light.

Emission

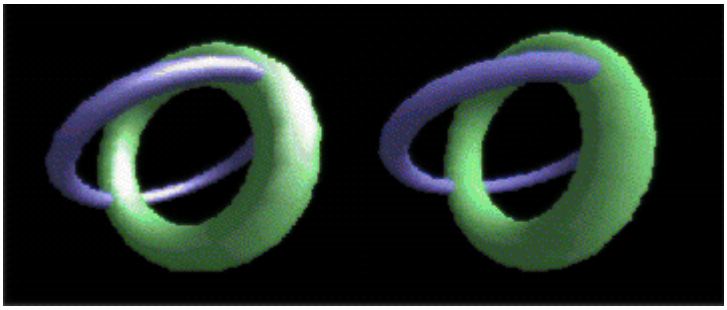
Materials can be used to make a rendered object appear to be self-luminous. The **Emissive** member of the **D3DMATERIAL8** structure is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take on part of the emitted color.

You can use a material's emissive property to add the illusion that an object is emitting light, without incurring the computational overhead of adding a light to the scene. In the case of the blue crystal, the emissive property is useful if you want to make the crystal appear to light up, but not cast light on other objects in the scene. Remember, materials with emissive properties don't emit light that can be reflected by other objects in a scene. To achieve this reflected light, you need to place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL8** structure contains two members that describe the specular highlight color as well as the material's overall shininess. You establish the color of the specular highlights by setting the **Specular** member to the desired RGBA color—the most common colors are white or light gray. The values you set in the **Power** member control how sharp the specular effects are.

Specular highlights can create dramatic effects. Drawing again on the blue crystal analogy: a larger **Power** value creates sharper specular highlights, making the crystal appear to be quite shiny. Smaller values increase the area of the effect, creating a dull reflection that make the crystal look frosty. To make an object truly matte, set the **Power** member to zero and the color in **Specular** to black. Experiment with different levels of reflection to produce a realistic appearance for your needs. The following illustration shows two identical models. The one on the left uses a specular reflection power of 10; the model on the right has no specular reflection.



Textures

[This is preliminary documentation and is subject to change.]

Textures are a powerful tool in creating realism in computer-generated 3-D images. Microsoft® Direct3D® supports an extensive texturing feature set, providing developers with easy access to advanced texturing techniques. This section discusses the purposes and uses of textures in Direct3D. The information is presented in the following topics.

- Basic Texturing Concepts
- Texture Coordinates
- Texture Resources
- Texture Filtering
- Texture Wrapping
- Compressed Texture Resources
- Volume Texture Resources
- Automatic Texture Management
- Hardware Considerations for Texturing
- Texture Blending
- Surfaces

Microsoft DirectX 8.1 (C++)

Basic Texturing Concepts

[This is preliminary documentation and is 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—such as scuffs, cracks, fingerprints, and smudges—that give 3-D objects realistic visual complexity. In recent years, textures have gained popularity among developers as a tool for enhancing the realism of computer-generated 3-D images.

In its everyday use, the word *texture* refers to an object's smoothness or roughness. In computer graphics, however, a texture is a bitmap of pixel colors that give an object the appearance of texture.

Because Microsoft® Direct3D® textures are bitmaps, any bitmap can be applied to a Direct3D primitive. For instance, applications can create and manipulate objects that appear to have a wood grain pattern in them. Grass, dirt, and rocks can be applied to a set of 3-D primitives that form a hill. The result is a realistic-looking hillside. You can also use texturing to create effects such as signs along a roadside, rock strata in a cliff, or the appearance of marble on a floor.

In addition, Direct3D supports more advanced texturing techniques such as texture blending—with or without transparency—and light mapping. For more information, see Texture Blending and Light Mapping with Textures.

If your application creates a hardware abstraction layer (HAL) device or a software device, it can use 8-, 16-, 24-, or 32-bit textures.

Additional information is contained in the following topics.

- Texture Addressing Modes
- Texture Dirty Regions

- Texture Palettes

Microsoft DirectX 8.1 (C++)

Texture Addressing Modes

[This is preliminary documentation and is subject to change.]

Your Microsoft® Direct3D® application can assign texture coordinates to any vertex of any primitive. For details, see Texture Coordinates. Typically, the u- and v-texture coordinates that you assign to a vertex are in the range of 0.0 to 1.0 inclusive. However, by assigning texture coordinates outside that range, you can create certain special texturing effects.

You control what Direct3D does with texture coordinates that are outside the [0.0, 1.0] range by setting the texture addressing mode. For instance, you can have your application set the texture addressing mode so that a texture is tiled across a primitive. The following topics contain additional details.

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.

Setting the Addressing Mode

You can set texture addressing modes for individual texture stages by calling the **IDirect3DDevice8::SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW values to update the u-, v-, or w-addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set. This can be any member of the **D3DTEXTUREADDRESS** enumerated type. To retrieve the current texture address mode for a texture stage, call **IDirect3DDevice8::GetTextureStageState**, using the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW members of the **D3DTEXTURESTAGESTATETYPE** enumeration to identify the address mode about which you want information.

Device Limitations

Although the system generally allows texture coordinates outside the range of 0.0 and 1.0, inclusive, hardware limitations often affect how far outside that range texture coordinates can be. A rendering device communicates this limit in the **MaxTextureRepeat** member of the **D3DCAPS8** structure when you retrieve device capabilities. The value in this member describes the full range of texture coordinates allowed by the device. For instance, if this value is 128, then the input texture coordinates must be kept in the range -128.0 to +128.0. Passing vertices with texture coordinates outside this range is invalid. The same restriction applies to the texture coordinates generated as a result of automatic texture coordinate generation and texture coordinate transformations.

The interpretation of **MaxTextureRepeat** is also affected by the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE capability bit. When this bit is set, the value in the **MaxTextureRepeat** member of **D3DCAPS8** is used precisely as described. However, when D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is not set, texture repeating limitations depend on the size of the texture indexed by the texture coordinates. In this case, **MaxTextureRepeat** must be scaled by the current texture size at the largest level of detail to compute the valid texture coordinate range. For example, given a texture dimension of 32 and **MaxTextureRepeat** value of 512, the actual valid texture coordinate range is 512/32 = 16, so the texture coordinates for this device must be within the range of -16.0 to +16.0.

Additional information on the texture addressing modes is contained in the following topics.

- Wrap Texture Address Mode
- Mirror Texture Address Mode
- Clamp Texture Address Mode
- Border Color Texture Address Mode

Microsoft DirectX 8.1 (C++)

Wrap Texture Address Mode

[This is preliminary documentation and is subject to change.]

The wrap texture address mode, identified by the D3DTADDRESS_WRAP member of the **D3DTEXTUREADDRESS** enumerated type, makes Microsoft® Direct3D® repeat the texture on every integer junction. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_WRAP results in the texture being applied three times in both the u-and v-directions.

This is illustrated in the following figure.

The effects of this texture address mode are similar to, but distinct from, those of the mirror mode. For more information, see Mirror Texture Address Mode.

Microsoft DirectX 8.1 (C++)

Mirror Texture Address Mode

[This is preliminary documentation and is subject to change.]

The mirror texture address mode, identified by the D3DTADDRESS_MIRROR member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to mirror the texture at every integer boundary. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_MIRROR results in the texture being applied three times in both the u- and v-directions. Every other row and column that it is applied to is a mirror image of the preceding row or column.

This is illustrated in the following figure.

The effects of this texture address mode are similar to, but distinct from, those of the wrap mode. For more information, see Wrap Texture Address Mode.

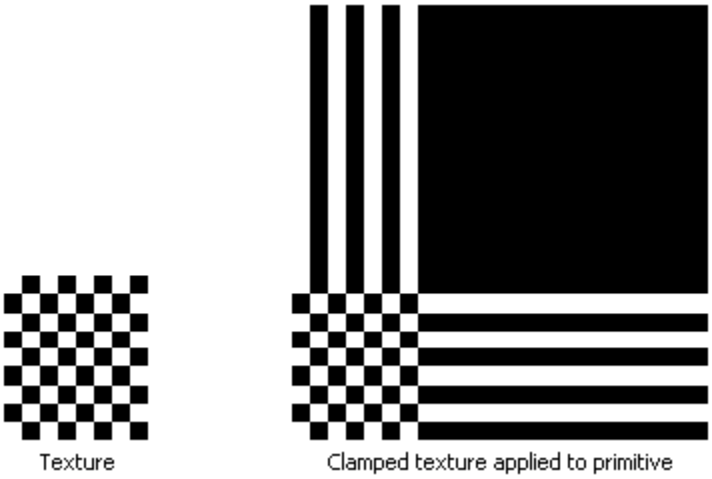
Microsoft DirectX 8.1 (C++)

Clamp Texture Address Mode

[This is preliminary documentation and is subject to change.]

The clamp texture address mode, identified by the D3DTADDRESS_CLAMP member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to clamp your texture coordinates to the [0.0, 1.0] range. That is, it applies the texture once, then smears the color of edge pixels. For example, suppose that your application creates a square primitive and assigns texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0) to the primitive's vertices. Setting the texture addressing mode to D3DTADDRESS_CLAMP results in the texture being applied once. The pixel colors at the top of the columns and the end of the rows are extended to the top and right of the primitive respectively.

This is illustrated in the following figure.



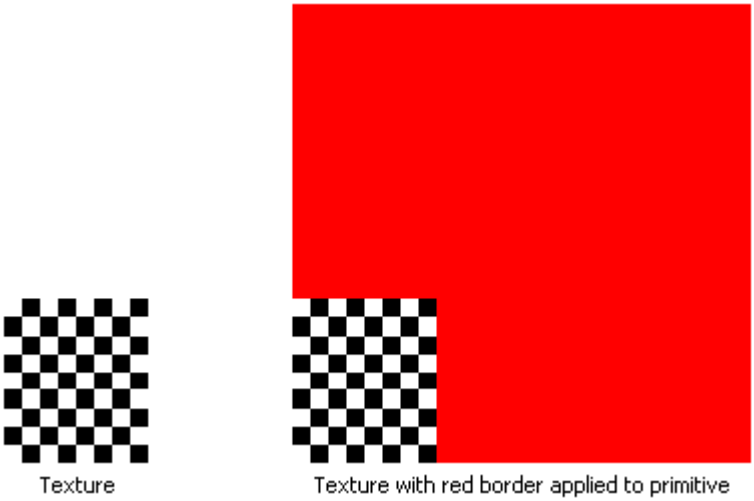
Microsoft DirectX 8.1 (C++)

Border Color Texture Address Mode

[This is preliminary documentation and is subject to change.]

The border color texture address mode, identified by the D3DTADDRESS_BORDER member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to use an arbitrary color, known as the border color, for any texture coordinates outside the range of 0.0 through 1.0, inclusive.

This is shown in the following figure in which the application specifies that the texture be applied to the primitive using a red border.



Applications set the border color by calling **IDirect3DDevice8::SetTextureStageState**. Set the first parameter for the call to the desired texture stage identifier, the second parameter to the D3DTSS_BORDERCOLOR stage state value, and the third parameter to the new RGBA border color.

Microsoft DirectX 8.1 (C++)

Texture Dirty Regions

[This is preliminary documentation and is subject to change.]

Applications can optimize what subset of a texture is copied by specifying dirty regions on textures. Only those regions marked as dirty are copied by a call to **IDirect3DDevice8::UpdateTexture**. However, the dirty regions may be expanded to optimize alignment. When a texture is created, the entire texture is considered dirty. Only the following four operations affect the dirty state of a texture.

- Adding a dirty region to a texture
- Locking some buffer in the texture. This operation adds the locked region as a dirty region. The application can turn off this automatic dirty region update if it has better knowledge of the actual dirty regions.
- Using the texture as a destination in **IDirect3DDevice8::CopyRects** marks the entire texture as dirty.
- Using the texture as a source in **UpdateTexture** clears all the dirty regions on the source texture.

Dirty regions are set on the top level of a mipmapped texture, and **UpdateTexture** can expand the dirty region down the mip chain in order to minimize the number of bytes copied for each sublevel. Note that the sublevel dirty region coordinates are rounded outward, that is, their fractional parts are rounded toward the nearest edge of the texture.

Because each type of texture has different types of dirty regions, there are methods on each texture type. 2-D textures use dirty rectangle, and volume textures use boxes.

- **IDirect3DCubeTexture8::AddDirtyRect**
- **IDirect3DTexture8::AddDirtyRect**
- **IDirect3DVolumeTexture8::AddDirtyBox**

Passing NULL for the *pDirtyRect* or *pDirtyBox* parameters for the above methods expands the dirty region to cover the entire texture.

Each lock method can take D3DLOCK_NO_DIRTY_UPDATE, which prevents any changes to the dirty state of the texture. For more information, see Locking Resources.

Applications should use D3DLOCK_NO_DIRTY_UPDATE when further information about the true set of regions changed during a lock operation is available. You should note that a lock or copy to only a sublevel—that is, without locking or copying to the top level—of texture does not update the dirty regions for that texture. Applications assume the same responsibility for updating dirty regions when they lock lower levels without locking the topmost level.

Microsoft DirectX 8.1 (C++)

Texture Palettes

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports paletted textures through a set of 256 entry palettes associated with the **IDirect3DDevice8** object. A palette is made current by calling the **IDirect3DDevice8::SetCurrentTexturePalette** method. The current palette is used for translating all paletted textures for all active texture stages. **IDirect3DDevice8::SetPaletteEntries** updates all of a palette's 256 entries. Each entry is a **PALETTEENTRY** structure of the format D3DFMT_A8R8G8B8. All entries default to 0xFFFFFFFF.

The **IDirect3DDevice8** palettes contain an alpha channel. This alpha channel can be used when the D3DPTEXTURECAPS_ALPHAPALETTE device capability flag is set, indicating that the device supports alpha from the palette. The palette alpha channel is used when the texture format does not have an alpha channel. If the device does not support alpha from the palette and the texture format does not have an alpha channel, then a value of 0xFF is used for alpha.

There is a maximum of 16,384 palettes. Because memory resources associated with the set of palettes are proportional to the maximum palette number that an application references, use contiguous palette numbers starting at zero.

Microsoft DirectX 8.1 (C++)

Texture Coordinates

[This is preliminary documentation and is subject to change.]

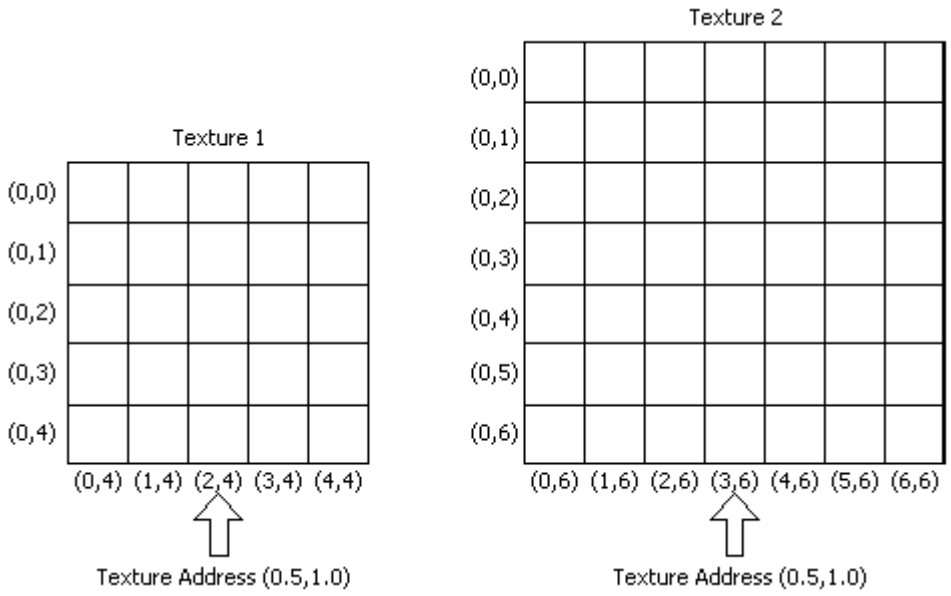
Most textures, like bitmaps, are a two-dimensional array of color values. Cubic-environment map textures are an exception. For details, see **Cubic Environment Mapping**. The individual color values are called a texture element, or texel. Each texel has a unique address in the texture. The address can be thought of as a column and row number, which are labeled u and v respectively.

Texture coordinates are in texture space. That is, they are relative to the location (0,0) in the texture. When a texture is applied to a primitive in 3-D space, its texel addresses must be mapped into object coordinates. They must then be translated into screen coordinates, or pixel locations.

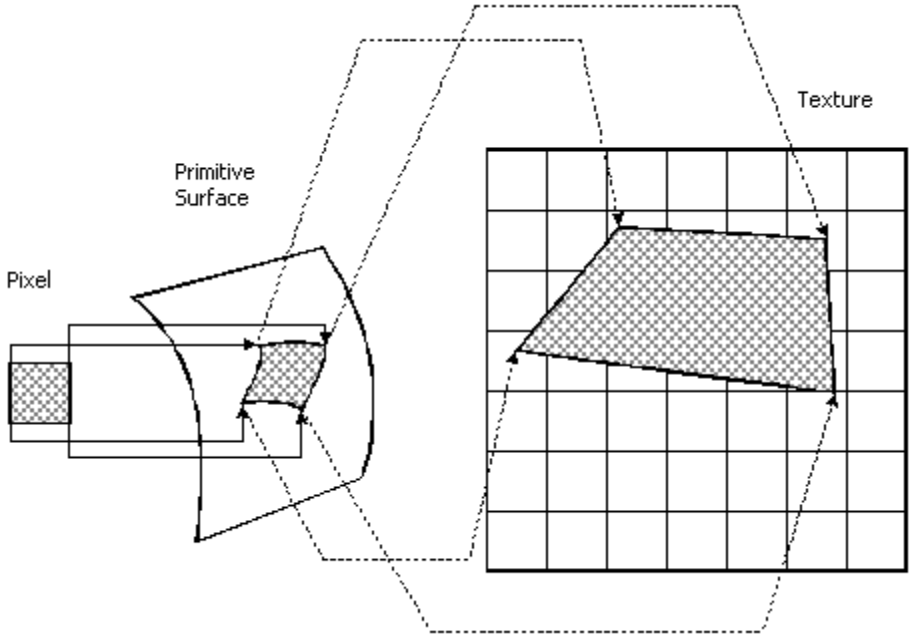
Microsoft® Direct3D® maps texels in texture space directly to pixels in screen space, skipping the intermediate step for greater efficiency. This mapping process is actually an inverse mapping. That is, for each pixel in screen space, the corresponding texel position in texture space is calculated. The texture color at or around that point is sampled. The sampling process is called texture filtering. For more information, see Texture Filtering.

Each texel in a texture can be specified by its texel coordinate. However, in order to map texels onto primitives, Direct3D requires a uniform address range for all texels in all textures. Therefore, it uses a generic addressing scheme in which all texel addresses are in the range of 0.0 to 1.0 inclusive. Direct3D applications specify texture coordinates in terms of u,v values, much like 2-D Cartesian coordinates are specified in terms of x,y coordinates. Technically, the system can actually process texture coordinates outside the range of 0.0 and 1.0, and does so by using the parameters you set for texture addressing. For more information, see Texture Addressing Modes.

A result of this is that identical texture addresses can map to different texel coordinates in different textures. In the following illustration, the texture address being used is (0.5,1.0). However, because the textures are different sizes, the texture address maps to different texels. Texture 1, on the left, is 5x5. The texture address (0.5,1.0) maps to texel (2,4). Texture 2, on the right, is 7x7. The texture address (0.5,1.0) maps to texel (3,6).



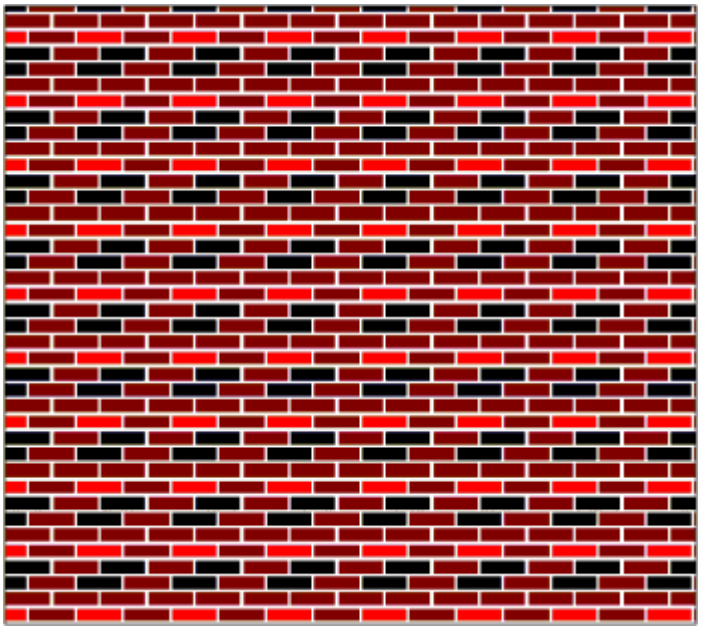
A simplified version of the texel mapping process is shown in the following diagram. Admittedly, this example is extremely simple. For more detailed information, see Directly Mapping Texels to Pixels.



For this example, a pixel, shown at the left of the illustration, is idealized into a square of color. The addresses of the four corners of the pixel are mapped onto the 3-D primitive in object space. The shape of the pixel is often distorted because of the shape of the primitive in 3-D space and because of the viewing angle. The corners of the surface area on the primitive that correspond the corners of the pixel are then mapped into texture space. The mapping process distorts the pixel's shape again, which is common. The final color value of the pixel is computed from the texels in the region to which the pixel maps. You determine the method that Direct3D uses to arrive at the pixel color when you set the texture filtering method. For more information, see Texture Filtering.

Your application can assign texture coordinates directly to vertices. This capability gives you control over which portion of a texture is mapped onto a primitive. For instance, suppose you create a rectangular primitive that is exactly the same size as the texture in the following illustration. In this example, you want your application to map the whole texture onto the whole wall. The texture

coordinates your application assigns to the vertices of the primitive are (0.0,0.0), (1.0,0.0), (1.0,1.0), and (0.0,1.0).



If you decide to decrease the height of the wall by one-half, you can distort the texture to fit onto the smaller wall, or you can assign texture coordinates that cause Direct3D to use the bottom half of the texture.

If you decide to distort or scale the texture to fit the smaller wall, the texture filtering method that you use will influence the quality of the image. For more information, see Texture Filtering.

If, instead, you decide to assign texture coordinates to make Direct3D use the bottom half of the texture for the smaller wall, the texture coordinates your application assigns to the vertices of the primitive in this example are (0.0,0.0), (1.0,0.0), (1.0,0.5), and (0.0,0.5). Direct3D applies the bottom half of the texture to the wall.

It is possible for texture coordinates of a vertex to be greater than 1.0. When you assign texture coordinates to a vertex that are not in the range of 0.0 to 1.0 inclusive, you should also set the texture addressing mode. For further information, see Texture Addressing Modes.

Additional information is contained in the following topics.

- Directly Mapping Texels to Pixels
- Texture Coordinate Formats
- Texture Coordinate Processing

Microsoft DirectX 8.1 (C++)

Directly Mapping Texels to Pixels

[This is preliminary documentation and is subject to change.]

Applications often need to apply textures to geometry in a scene so that texels map directly to on-screen pixels. For example, take an application that needs to display text within a texture on an object within a scene. In order to clearly display textual information in a texture, the application

needs some way to ensure that the textured geometry receives texels undisrupted by texture filtering. Failing this, the resulting image is often blurred, or in the case of point-sampled texture filtering, can cause rough edges.

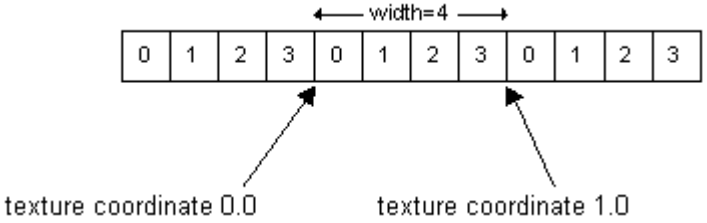
Because the Microsoft® Direct3D® pixel and texture sampling rules are carefully defined to unify pixel and texture sampling while supporting image and texture filtering, getting the texels in a texture to map directly to on-screen pixels can be a significant and often frustrating challenge. Overcoming this challenge requires a clear understanding of how Direct3D maps the floating-point texture coordinates for a vertex to the integer pixel coordinates used by the rasterizer.

Direct3D performs the following computations to map floating point texture coordinates to texel addresses.

$$T_x = (u \times M_x) - 0.5$$
$$T_y = (v \times M_y) - 0.5$$

In these formulas, T_x and T_y are the horizontal and vertical output texel coordinates, and *u* and *v* are the horizontal and vertical texture coordinates supplied for the vertex. The M_x and M_y elements represent the number of horizontal or vertical texels at the current mipmap level. The remainder of this discussion targets horizontal mapping of texels to pixels; keep in mind that mapping in the vertical is identical to the horizontal case.

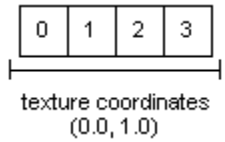
Placing the texture coordinate limits 0.0 and 1.0 into these formulas maps a texture coordinate of 0.0 halfway between the first and last texels of a repeated texture map. The coordinate 1.0 is mapped halfway between the last texel of the current iteration of the texture map and the next iteration. For a repeated texture that's four texels wide, at mipmap level 0, the following illustration shows where the system maps the coordinates 0.0 and 1.0.



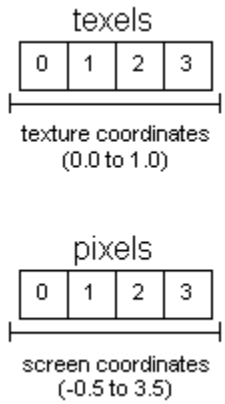
Given an understanding of this mapping, you can apply a simple bias to your screen-space geometry coordinates to force the system to map each texel to a corresponding pixel. For example, to draw a four-sided polygon that maps each texel from the preceding texture to one, and only one, pixel on the screen, you must force geometry coordinates to overlap the pixels, effectively placing the center of each texel at the center of each pixel. The result is the 1-to-1 mapping often sought-after by applications.

To map the texture with a 4-texel width to the pixel coordinates 0 through 3, draw a four-sided polygon, from two triangles, that has screen-space coordinates of –0.5 to 3.5 and texture coordinates of 0.0 to 1.0. For example, take the pixel at screen coordinate 0.0. Because 0.0 is one-half pixel away from the first vertex, at -0.5, and the total width is 4.0, the iterated texture coordinate is 0.125. Scaling this by the texture size, which is 4, results in the coordinate 0.5. Subtracting this 0.5 bias produces a texture address of 0.0, which fully maps to the first texel in the map.

To summarize, texture coordinates overlap the texture map evenly on both sides. The following illustration shows the mapping of a texture that is four texels wide.



The system normalizes pixel coordinates in the same way it normalizes texel coordinates. Therefore, if the vertices overlap the pixels into which to render, and if the vertices use texture coordinates of 0.0 and 1.0, the pixels and texels line up. If both are of similar size and properly aligned, they match exactly, texel to pixel, as shown in the following figure.



Microsoft DirectX 8.1 (C++)

Texture Coordinate Formats

[This is preliminary documentation and is subject to change.]

Texture coordinates in Microsoft® Direct3D® can include one, two, three, or four floating point elements to address textures with varying levels of dimension. A 1-D texture—a texture surface with dimensions of 1-by-*n* texels—is addressed by one texture coordinate. The most common case, 2-D textures, are addressed with two texture coordinates commonly called *u* and *v*. Direct3D supports two types of 3-D textures, *cubic-environment maps* and *volume textures*. Cubic environment maps aren't truly 3-D, but they are addressed with a 3-element vector. For details, see Cubic Environment Mapping.

As described in Vertex Formats, applications encode texture coordinates in the vertex format. The vertex format can include multiple sets of texture coordinates. Use the D3DFVF_TEX0 through D3DFVF_TEX8 Flexible Vertex Format Flags to describe a vertex format that includes no texture coordinates, or as many as eight sets.

Each texture coordinate set can have between one and four elements. The D3DFVF_TEXTUREFORMAT1 through D3DFVF_TEXTUREFORMAT4 flags describe the number of elements in a texture coordinate in a set, but these flags aren't used by themselves. Rather, the **D3DFVF_TEXCOORDSIZE*n*** set of macros use these flags to create bit patterns that describe the number of elements used by a particular set of texture coordinates in the vertex format. These macros accept a single parameter that identifies the index of the coordinate set whose number of elements is being defined. The following example illustrates how these macros are used.

```
// This vertex format contains two sets of texture coordinates.
// The first set (index 0) has 2 elements, and the second set
// has 1 element. The description for this vertex format would be:
//     dwFVF = D3DFVF_XYZ  | D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_TEX2 |
```

```
//             D3DFVF_TEXCOORDSIZE2(0) | D3DFVF_TEXCOORDSIZE1(1);
//
typedef struct CVF
{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DCOLOR  diffuse;
    float     u, v;   // 1st set, 2-D
    float     t;      // 2nd set, 1-D
} CustomVertexFormat;
```

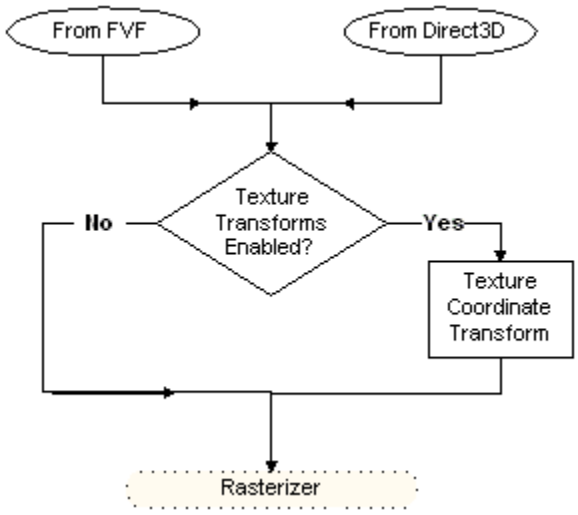
Note With the exception of cubic-environment maps and volume textures, rasterizers cannot address textures by using any more than two elements. Applications can supply up to three elements for a texture coordinate, but only if the texture is a cube map, a volume texture, or the D3DTTFF_PROJECTED texture transform flag is used. The D3DTTFF_PROJECTED flag causes the rasterizer to divide the first two elements by the third (or n^{th}) element. For more information, see Texture Coordinate Transformations.

Microsoft DirectX 8.1 (C++)

Texture Coordinate Processing

[This is preliminary documentation and is subject to change.]

The following figure shows the path taken by the texture coordinates from their source, through processing, and to the rasterizer.



There are two sources from which the system can draw texture coordinates. For a given texture stage, you can use texture coordinates included in the vertex format (D3DFVF_TEX1 through D3DFVF_TEX8), or you can use texture coordinates automatically generated by Microsoft® Direct3D®. For details about the latter case, see Automatically Generated Texture Coordinates. If the **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state for the current texture stage is set to D3DTTFF_DISABLE (the default setting), input coordinates are not transformed. If **D3DTSS_TEXTURETRANSFORMFLAGS** is set to any other value, the transformation matrix for that stage is applied to the input coordinates.

The **D3DTEXTURETRANSFORMFLAGS** enumerated type defines valid values for the **D3DTSS_TEXTURETRANSFORMFLAGS** texture-stage state. With the exception of the D3DTTFF_DISABLE flag, which bypasses texture coordinate transformation, the values defined in

this enumeration configure the number of output coordinates that the system passes to the rasterizer. The D3DTTFF_COUNT1 through D3DTTFF_COUNT4 flags instruct the system to pass one, two, three, or four elements from the output coordinates to the rasterizer. Currently 1-D texture coordinates aren't supported.

The D3DTTFF_PROJECTED flag is special: it tells the system that the texture coordinates are for a projected texture. Combine the D3DTTFF_PROJECTED flag with another member of **D3DTEXTURETRANSFORMFLAGS** to instruct the rasterizer to divide all the elements by the last element before rasterization takes place. For example, when explicitly using three-element texture coordinates, or when transformation results in a three-element texture coordinate, you can combine the D3DTTFF_COUNT3 and D3DTTFF_PROJECTED flags to cause the rasterizer to divide the first two elements by the last, producing 2-D texture coordinates required to address a 2-D texture.

Note With the exception of cubic-environment maps and volume textures, rasterizers cannot address textures by using texture coordinates with more than two elements. If you specify more elements than can be used to address the current texture for that stage, the extraneous elements are ignored. This also applies when using 2-D texture coordinates for a 1-D texture.

Additional information is contained in the following topics.

- Automatically Generated Texture Coordinates
- Texture Coordinate Transformations
- Special Effects

Microsoft DirectX 8.1 (C++)

Automatically Generated Texture Coordinates

[This is preliminary documentation and is subject to change.]

The system can use the transformed camera-space position or the normal from a vertex as texture coordinates, or it can compute the three element vectors used to address a cubic environment map. Like texture coordinates that you explicitly specify in a vertex, you can use automatically generated texture coordinates as input for texture coordinate transformations.

Automatically generated texture coordinates can significantly reduce the bandwidth required for geometry data by eliminating the need for explicit texture coordinates in the vertex format. In many cases, the texture coordinates that the system generates can be used with transformations to produce special effects. Of course, this is a special-purpose feature, and you will use explicit texture coordinates for many occasions.

Configuring Automatically Generated Texture Coordinates

In C++, the D3DTSS_TEXCOORDINDEX texture-stage state (from the **D3DTEXTURESTAGESTATETYPE** enumerated type) controls how the system generates texture coordinates.

Normally, this state instructs the system to use a particular set of texture coordinates encoded in the vertex format. When you include the D3DTSS_TCI_CAMERASPACENORMAL, D3DTSS_TCI_CAMERASPACEPOSITION, or D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flags in the value that you assign to this state, the system behavior is quite different. If any of these flags are present, the texture stage ignores

the texture coordinates within the vertex format in favor of coordinates that the system generates. The meanings for each flag are shown in the following list.

D3DTSS_TCI_CAMERASPACENORMAL
: Use the vertex normal, transformed to camera space, as input texture coordinates.

D3DTSS_TCI_CAMERASPACEPOSITION
: Use the vertex position, transformed to camera space, as input texture coordinates.

D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR
: Use the reflection vector, transformed to camera space, as input texture coordinates. The reflection vector is computed from the input vertex position and normal vector.

The preceding flags are mutually exclusive. If you include one flag, you can still specify an index value, which the system uses to determine the texture wrapping mode.

The following code example shows how these flags are used in C++.

```
/*
 * For this example, the d3dDevice variable is a valid
 * pointer to an IDirect3DDevice8 interface.
 *
 * Use the vertex position (camera-space) as the input
 * texture coordinates for this texture stage, and the
 * wrap mode set in the D3DRENDERSTATE_WRAP1 render state.
 */
d3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                                    D3DTSS_TCI_CAMERASPACEPOSITION | 1 );
```

Note Automatically generated texture coordinates are most useful as input values for a texture coordinate transformation, or to eliminate the need for your application to compute three-element vectors for cubic-environment maps.

See Also

Texture Coordinate Transformations, Cubic Environment Mapping

Microsoft DirectX 8.1 (C++)

Texture Coordinate Transformations

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® devices can transform the texture coordinates for vertices by applying a 4×4 matrix. The system applies transformations to texture coordinates in the same manner as geometry, and any transformations that can be communicated in a 4×4 matrix—scales, rotations, translations, projections, shears, or any combination of these—can be applied.

Note Direct3D does not modify transformed and lit vertices. As a result, an application using transformed and lit vertices cannot use Direct3D to transform the texture coordinates of the vertices.

Devices that support hardware-accelerated transformation and lighting operations (TnLHAL Device) also accelerate the transformation of texture coordinates. When hardware acceleration of transformations isn't available, platform-specific optimizations in the Direct3D geometry pipeline apply to texture coordinate transformations.

Texture coordinate transformations are useful for producing special effects while avoiding the need to directly modify the texture coordinates of your geometry. You could use simple translation or rotation matrices to animate textures on an object, or you can transform texture coordinates that are automatically generated by Direct3D to simplify and perhaps accelerate advanced effects such as projected textures and dynamic light-mapping. Additionally, you might use texture coordinate transforms to reuse a single set of texture coordinates for multiple purposes, in multiple texture stages.

Setting and Retrieving Texture Coordinate Transformations

Like the matrices that your application uses for geometry, you set and retrieve texture coordinate transformations by calling the **IDirect3DDevice8::SetTransform** and **IDirect3DDevice8::GetTransform** methods. These methods accept the D3DTS_TEXTURE0 through D3DTS_TEXTURE7 members of the **D3DTRANSFORMSTATETYPE** enumerated type to identify the transformation matrices for texture stages 0 through 7, respectively.

The following code sets a matrix to apply to the texture coordinates for texture stage 0.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.
//
D3DMATRIX matTrans = D3DXMatrixIdentity( NULL );

// Set-up the matrix for the desired transformation.
d3dDevice->SetTransform( D3DTS_TEXTURE0, &matTrans );
```

Enabling Texture Coordinate Transformations

The **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state controls the application of texture coordinate transformations. Values for this texture stage state are defined by the **D3DTEXTURETRANSFORMFLAGS** enumerated type.

Texture coordinate transformations are disabled when D3DTSS_TEXTURETRANSFORMFLAGS is set to D3DTTFF_DISABLE (the default value). Assuming that texture coordinate transformations were enabled for stage 0, the following code disables them.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

D3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_DISABLE
```

The other values defined in **D3DTEXTURETRANSFORMFLAGS** are used to enable texture coordinate transformations, and to control how many resulting texture coordinate elements are passed to the rasterizer. For example, take the following code.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

D3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2
```

The D3DTTFF_COUNT2 value instructs the system to apply the transformation matrix set for texture stage 0, and then pass the first two elements of the modified texture coordinates to the rasterizer.

The D3DTTFF_PROJECTED texture transformation flag indicates coordinates for a projected texture. When this flag is specified, the rasterizer divides the elements passed-in by the last element.

Take the following code, for example.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

d3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                    D3DTTFF_COUNT3 | D3DTTFF_PROJECTED );
```

This example informs the system to pass three texture coordinate elements to the rasterizer. The rasterizer divides the first two elements by the third, producing the 2-D texture coordinates needed to address the texture.

Microsoft DirectX 8.1 (C++)

Special Effects

[This is preliminary documentation and is subject to change.]

The following list contains examples of special effects accomplished with texture coordinate processing.

Animating textures (by translation or rotation) on a model

- Define two-dimensional (2-D) texture coordinates in your vertex format.

```
// Use a single texture, with 2-D texture coordinates. This
// bit-pattern should be expanded to include position, normal,
// and color information as needed.
DWORD dwFVFTex = D3FVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
```

- Configure the rasterizer to use 2-D texture coordinates.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2);
```

- Define and set an appropriate texture coordinate transformation matrix.

```
// M is a D3DMATRIX being set to translate texture
// coordinates in the U and V directions.
//      1   0  0  0
//      0   1  0  0
//      du dv  1  0 (du and dv change each frame)
//      0   0  0  1

D3DMATRIX M = D3DXMatrixIdentity(); // declared in d3dutil.h
M._31 = du;
M._32 = dv;
```

Creating texture coordinates as a linear function of a model's camera-space position

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position, in camera space, as input to a texture transformation.

```
// The input vertices have no texture coordinates, saving
// bandwidth. Three texture coordinates are generated by
// using vertex position in camera space (x, y, z).
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION
```

- Instruct the rasterizer to expect 2-D texture coordinates.

```
// Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2);
```

- Define and set a matrix that applies a linear function.

```
// Generate texture coordinates as linear functions
// so that:
//      u = Ux*x + Uy*y + Uz*z + Uw
//      v = Vx*x + Vy*y + Vz*z + Vw
// The matrix M for this case is:
//      Ux  Vx  0  0
//      Uy  Vy  0  0
//      Uz  Vz  0  0
//      Uw  Vw  0  0

SetTransform(D3DTS_TEXTURE0, &M);
```

Performing environment mapping with a cubic environment map

- Use the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag to instruct the system to automatically generate texture coordinates as reflection vectors for cubic mapping.

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEREFLECTI
```

- Instruct the rasterizer to expect texture coordinates with three elements.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3);
```

Performing projective texturing

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position as input to a texture transformation matrix.

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION
```

- Create and apply the texture projection matrix. This is beyond the scope of this documentation, and is the topic of several industry articles.
- Instruct the rasterizer to expect three-element projected texture coordinates.

```
// Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTF_PROJECTED | D3D
```

Microsoft DirectX 8.1 (C++)

Texture Resources

[This is preliminary documentation and is subject to change.]

Texture resources are implemented in the **IDirect3DTexture8** interface. To obtain a pointer to a texture interface, call the **IDirect3DDevice8::CreateTexture** method or any of the following D3DX functions.

- **D3DXCreateTexture**

- **D3DXCreateTextureFromFile**
- **D3DXCreateTextureFromFileEx**
- **D3DXCreateTextureFromFileInMemory**
- **D3DXCreateTextureFromFileInMemoryEx**
- **D3DXCreateTextureFromResource**
- **D3DXCreateTextureFromResourceEx**

The following code example uses **D3DXCreateTextureFromFile** to load a texture from Tiger.bmp.

```
// The following code example assumes that D3dDevice
// is a valid pointer to an IDirect3DDevice8 interface.

LPDIRECT3DTEXTURE8 pTexture;

D3DXCreateTextureFromFile( d3dDevice, "tiger.bmp", &pTexture);
```

The first parameter that **D3DXCreateTextureFromFile** accepts is a pointer to an **IDirect3DDevice8** interface. The second parameter tells Microsoft® Direct3D® the name of the file from which to load the texture. The third parameter takes the address of a pointer to a **IDirect3DTexture8** interface, representing the created texture object.

Rendering with Texture Resources

Microsoft® Direct3D® supports multiple texture blending through the concept of texture stages. Each texture stage contains a texture and operations that can be performed on the texture. The textures in the texture stages form the set of current textures. For more information, see Texture Blending. The state of each texture is encapsulated in its texture stage.

In a C++ application, the state of each texture must be set with the **IDirect3DDevice8::SetTextureStageState** method. Pass the stage number (0-7) as the value of the first parameter. Set the value of the second parameter to a member of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The final parameter is the state value for the particular texture state.

Using texture interface pointers, your application can render a blend of up to eight textures. Set the current textures by invoking the **IDirect3DDevice8::SetTexture** method. Direct3D blends all current textures onto the primitives that it renders.

Note The **SetTexture** method increments the reference count of the texture surface being assigned. When the texture is no longer needed, you should set the texture at the appropriate stage to NULL. If you fail to do this, the surface will not be released, resulting in a memory leak.

Your application can set the texture wrapping state for the current textures by calling the **IDirect3DDevice8::SetRenderState** method. Pass a value from D3DRS_WRAP0 through D3DRS_WRAP7 as the value of the first parameter, and use a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags to enable wrapping in the u, v, or w directions.

Your application can also set the texture perspective and texture filtering states. See Texture Filtering.

Microsoft DirectX 8.1 (C++)

Texture Filtering

[This is preliminary documentation and is subject to change.]

When Microsoft® Direct3D® renders a primitive, it maps the 3-D primitive onto a 2-D screen. If the primitive has a texture, Direct3D must use that texture to produce a color for each pixel in the primitive's 2-D rendered image. For every pixel in the primitive's on-screen image, it must obtain a color value from the texture. This process is called texture filtering.

When a texture filter operation is performed, the texture being used is typically also being magnified or minified. In other words, it is being mapped onto a primitive image that is larger or smaller than itself. Magnification of a texture can result in many pixels being mapped to one texel. The result can be a chunky appearance. Minification of a texture often means that a single pixel is mapped to many texels. The resulting image can be blurry or aliased. To resolve these problems, some blending of the texel colors must be performed to arrive at a color for the pixel.

Direct3D simplifies the complex process of texture filtering. It provides you with three types of texture filtering—linear filtering, anisotropic filtering, and mipmap filtering. If you select no texture filtering, Direct3D uses a technique called nearest-point sampling.

Each type of texture filtering has advantages and disadvantages. For instance, linear texture filtering can produce jagged edges or a chunky appearance in the final image. However, it is a computationally low-overhead method of texture filtering. Filtering with mipmaps usually produces the best results, especially when combined with anisotropic filtering. However, it requires the most memory of the techniques that Direct3D supports.

Applications that use texture interface pointers should set the current texture filtering method by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Pass a member of the **D3DTEXTUREFILTERTYPE** enumerated type as the value in the third parameter to set the magnification, minification, or mipmapping filter.

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

Note Although the texture-filtering render states present in the **D3DRENDERSTATETYPE** enumerated type are superseded by texture stage states, the **IDirect3DDevice8::SetRenderState**, as opposed to the **IDirect3DDevice2** version, does not fail if you attempt to use them. Rather, the system maps the effects of these render states to the first stage in the multitexture cascade, stage 0. Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

Microsoft DirectX 8.1 (C++)

Nearest-Point Sampling

[This is preliminary documentation and is subject to change.]

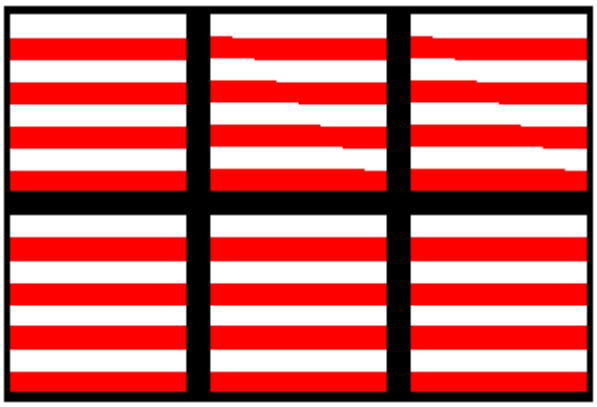
Applications are not required to use texture filtering. Microsoft® Direct3D® can be set so that it computes the texel address, which often does not evaluate to integers, and simply copies the color of the texel with the closest integer address. This process is called *nearest-point sampling*. This can be a fast and efficient way to process textures if the size of the texture is similar to the size of the primitive's image on the screen. If not, the texture must be magnified or minified. The result can be a chunky, aliased, or blurred image.

Your C++ application can select nearest-point sampling by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Pass a member of the **D3DTEXTUREFILTERTYPE** enumerated type as the value in the third parameter to set the magnification, minification, or mipmapping filter.

You should use nearest-point sampling carefully, as it can sometimes cause graphic artifacts when the texture is sampled at the boundary between two texels. This boundary is the position along the texture (u or v) at which the sampled texel transitions from one texel to the next. When point sampling is used, the system chooses one sample texel or the other, and the result can change abruptly from one texel to the next texel as the boundary is crossed. This effect can appear as undesired graphic artifacts in the displayed texture. When linear filtering is used, the resulting texel is computed from both adjacent texels and smoothly blends between them as the texture index moves through the boundary.

This effect can be seen when mapping a very small texture onto a very large polygon: an operation often called *magnification*. For example, when using a texture that looks like a checkerboard, nearest-point sampling results in a larger checkerboard that shows distinct edges. By contrast, linear texture filtering results in an image where the checkerboard colors vary smoothly across the polygon.

In most cases, applications receive the best results by avoiding nearest-point sample wherever possible. The majority of hardware today is optimized for linear filtering, so your application should not suffer degraded performance. If the effect you desire absolutely requires the use of the nearest-point sampling—such as when using textures to display readable text characters—then your application should be extremely careful to avoid sampling at the texel boundaries, which might result in undesired effects. The following image shows what these artifacts can look like.



Notice that the two squares in the top-right of the group appear different than their neighbors, with diagonal offsets running through them. To avoid graphic artifacts like these, you must be familiar with Direct3D texture sampling rules for nearest-point filtering. Direct3D maps a floating-point texture coordinate ranging from [0.0, 1.0] (0.0 to 1.0, inclusive) to an integer texel space value

ranging from [–0.5, *n* – 0.5], where *n* is the number of texels in a given dimension on the texture. The resulting texture index is rounded to the nearest integer. This mapping can introduce sampling inaccuracies at texel boundaries.

For a simple example, imagine an application that renders polygons with the D3DTADDRESS_WRAP texture addressing mode. Using the mapping employed by Direct3D, the *u* texture index maps as follows for a texture with a width of 4 texels.

Notice that the texture coordinates—0.0 and 1.0—for this illustration are exactly at the boundary between texels. Using the method by which Direct3D maps values, the texture coordinates range from [–0.5, 4 – 0.5], where 4 is the width of the texture. For this case, the sampled texel is the 0^{th} texel for a texture index of 1.0. However, if the texture coordinate was only slightly less than 1.0, the sampled texel would be the n^{th} texel instead of the 0^{th} texel.

The implication of this is that magnifying a small texture using texture coordinates of exactly 0.0 and 1.0 with nearest-point filtering on a screen-space aligned triangle results in pixels for which the texture map is sampled at the boundary between texels. Any inaccuracies in the computation of texture coordinates, however small, results in artifacts along the areas in the rendered image which correspond to the texel edges of the texture map.

Performing this mapping of floating point texture coordinates to integer texels with perfect accuracy is difficult, computationally expensive, and generally not necessary. Most hardware implementations use an iterative approach for computing texture coordinates at each pixel location within a triangle. Iterative approaches tend to hide these inaccuracies because the errors are accumulated evenly during iteration.

The Direct3D reference rasterizer uses a direct-evaluation approach for computing texture indexes at each pixel location. Direct evaluation differs from the iterative approach in that any inaccuracy in the operation exhibits a more random error distribution. The result of this is that the sampling errors that occur at the boundaries can be more noticeable because the reference rasterizer does not perform this operation with perfect accuracy.

The best approach is to use nearest-point filtering only when necessary. When you must use it, it is recommended that you offset texture coordinates slightly from the boundary positions to avoid artifacts.

Microsoft DirectX 8.1 (C++)

Linear Texture Filtering

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® uses a form of linear texture filtering called bilinear filtering. Like Nearest-Point Sampling, bilinear texture filtering first computes a texel address, which is usually not an integer address. Bilinear filtering then finds the texel whose integer address is closest to the computed address. In addition, the Direct3D rendering module computes a weighted average of the

texels that are immediately above, below, to the left of, and to the right of the nearest sample point.

Select bilinear texture filtering by invoking the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_LINEAR.

Microsoft DirectX 8.1 (C++)

Anisotropic Texture Filtering

[This is preliminary documentation and is 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 to texels, its shape is distorted. Microsoft® Direct3D® measures the anisotropy of a pixel as the elongation—that is, length divided by width—of a screen pixel that is inverse-mapped into texture space.

You can use anisotropic texture filtering in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_ANISOTROPIC.

Your application must also set the degree of anisotropy to a value greater than one. Do this by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are setting the degree of isotropy. Pass D3DTSS_MAXANISOTROPY as the value of the second parameter. The final parameter should be the degree of isotropy.

You can disable isotropic filtering by setting the degree of isotropy to one; any value larger than one enables it. Check the MaxAnisotropy flag in the **D3DCAPS8** structure to determine the possible range of values for the degree of anisotropy.

Microsoft DirectX 8.1 (C++)

Texture Filtering with Mipmaps

[This is preliminary documentation and is 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 user. Lower-resolution images are used as the object appears farther away. Mipmapping improves the quality of rendered textures at the expense of using more memory.

Microsoft® Direct3D® represents mipmaps as a chain of attached surfaces. The highest resolution texture is at the head of the chain and has the next level of the mipmap as an attachment. In turn, that level has an attachment that is the next level in the mipmap, and so on, down to the lowest resolution level of the mipmap.

The following illustrations shows an example of these levels. The bitmap textures represent a sign on a container in a 3-D, first-person game. When created as a mipmap, the highest-resolution texture is first in the set. Each succeeding texture in the mipmap set is smaller in height and width by a power of 2. In this case, the maximum-resolution mipmap is 256 pixels by 256 pixels. The next, texture is 128x128. The last texture in the chain is 64x64.

This sign has a maximum distance from which it is visible. If the user begins far away from the sign, the game displays the smallest texture in the mipmap chain, which in this case the 64x64 texture.

As the user moves the point of view closer to the sign, progressively higher-resolution textures in the mipmap chain are used. The resolution in the following illustration is 128x128.

The highest-resolution texture is used when the user's point of view is at the minimum allowable distance from the sign.

This is a computationally lower-overhead way of simulating perspective for textures. Rather than render a single texture at many resolutions, it is faster to use multiple textures at varying resolutions.

Direct3D can assess which texture in a mipmap set is the closest resolution to the desired output, and it can map pixels into its texel space. If the resolution of the final image is between the resolutions of the textures in the mipmap set, Direct3D can examine texels in both mipmaps and blend their color

values together.

To use mipmaps, your application must build a set of mipmaps. Applications apply mipmaps by selecting the mipmap set as the first texture in the set of current textures. For more information, see Texture Blending.

Next, your application must set the filtering method that Direct3D uses to sample texels. The fastest method of mipmap filtering is to have Direct3D select the nearest texel. Use the D3DTFP_POINT enumerated value to select this. Direct3D can produce better filtering results if your application uses the D3DTFP_LINEAR enumerated value. This selects the nearest mipmap, and then computes a weighted average of the texels surrounding the location in the texture to which the current pixel maps.

Mipmap textures are used in 3-D scenes to decrease the time required to render a scene. They also improve the scene's realism. However, they often require large amounts of memory.

Creating a Set of Mipmaps

The following example shows how your application can call the **IDirect3DDevice8::CreateTexture** method to build a chain of five mipmap levels: 256×256, 128×128, 64×64, 32×32, and 16×16.

```
// This code example assumes that the variable d3dDevice is a
// valid pointer to a IDirect3DDevice8 interface.

IDirect3DTexture8 * pMipMap;

d3dDevice->CreateTexture(256, 256, 5, 0, D3DFMT_R8G8B8, D3DPOOL_MANAGED, &pMipMap
```

The first two parameters that are accepted by **CreateTexture** are the size and width of the top-level texture. The third parameter specifies the number of levels in the texture. If you set this to zero, Microsoft® Direct3D® creates a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size of 1x1. The fourth parameter specifies the usage for this resource; in this case, 0 is specified to indicate no specific usage for the resource. The fifth parameter specifies the surface format for the texture. Use a value from the **D3DFORMAT** enumerated type for this parameter. The sixth parameter specifies a member of the **D3DPOOL** enumerated type indicating the memory class into which to place the created resource. Unless you are using dynamic textures, D3DPOOL_MANAGED is recommended. The final parameter takes the address of a pointer to a **IDirect3DTexture8** interface.

Note Each surface in a mipmap chain has dimensions that are one-half that of the previous surface in the chain. If the top-level mipmap has dimensions of 256×128, the dimensions of the second-level mipmap are 128×64, the third-level is 64×32, and so on, down to 1×1. You cannot request a number of mipmap levels in **Levels** that would cause either the width or height of any mipmap in the chain to be smaller than 1. In the simple case of a 4×2 top-level mipmap surface, the maximum value allowed for **Levels** is three. The top-level dimensions are 4×2, the second-level dimensions are 2×1, and the dimensions for the third level are 1×1. A value larger than 3 in **Levels** results in a fractional value in the height of the second-level mipmap, and is therefore disallowed.

Selecting and Displaying a Mipmap

Call the **IDirect3DDevice8::SetTexture** method to set the mipmap texture set as the first texture in the list of current textures. For more information, see Texture Blending.

After your application selects the mipmap texture set, it must assign values from the **D3DTEXTUREFILTERTYPE** enumerated type to the D3DTSS_MIPFILTER texture stage state.

Microsoft® Direct3D® then automatically performs mipmap texture filtering. Enabling mipmap texture filtering is demonstrated in the following code example.

```
d3dDevice->SetTexture(0, pMipMap);
d3dDevice->SetTextureStageState(0, D3DTSS_MIPFILTER, D3DTEXF_POINT);
```

Your application can also manually traverse a chain of mipmap surfaces by using the **IDirect3DTexture8::GetSurfaceLevel** method and specifying the mipmap level to retrieve. The following example traverses a mipmap chain from highest to lowest resolutions.

```
IDirect3DSurface8 * pSurfaceLevel;

for (int iLevel = 0; iLevel < pMipMap->GetLevelCount(); iLevel++)
{
    pMipMap->GetSurfaceLevel(iLevel, &pSurfaceLevel);

    //Process this level.

    pSurfaceLevel->Release();
}
```

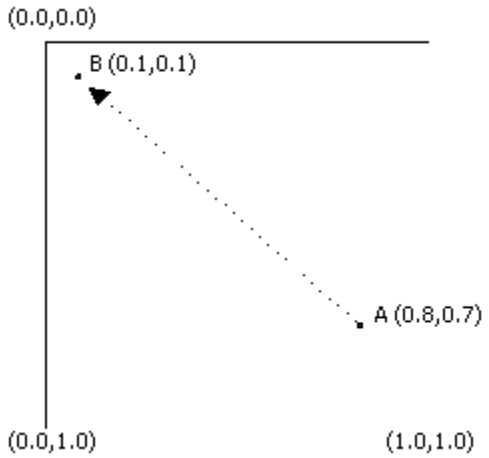
Applications need to manually traverse a mipmap chain to load bitmap data into each surface in the chain. This is typically the only reason to traverse the chain. An application can retrieve the number of levels in a mipmap by calling **IDirect3DBaseTexture8::GetLevelCount**.

Microsoft DirectX 8.1 (C++)

Texture Wrapping

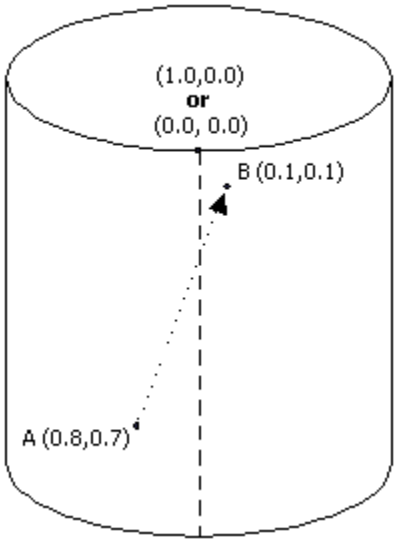
[This is preliminary documentation and is subject to change.]

In short, texture wrapping changes the basic way that Microsoft® Direct3D® rasterizes textured polygons using the texture coordinates specified for each vertex. While rasterizing a polygon, the system interpolates between the texture coordinates at each of the polygon's vertices to determine the texels that should be used for every pixel of the polygon. Normally, the system treats the texture as a 2-D plane, interpolating new texels by taking the shortest route from point A within a texture to point B. If point A represents the u, v position (0.8, 0.1), and point B is at (0.1,0.1), the line of interpolation looks like the following illustration.



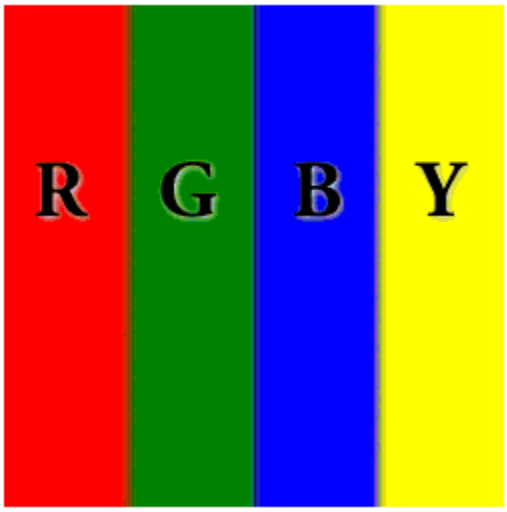
Note that the shortest distance between A and B in this illustration runs roughly through the middle

of the texture. Enabling u- or v-texture coordinate wrapping changes how Direct3D perceives the shortest route between texture coordinates in the u- and v-directions. By definition, texture wrapping causes the rasterizer to take the shortest route between texture coordinate sets, assuming that 0.0 and 1.0 are coincident. The last bit is the tricky part: you can imagine that enabling texture wrapping in one direction causes the system to treat a texture as though it were wrapped around a cylinder. For example, consider the following illustration.



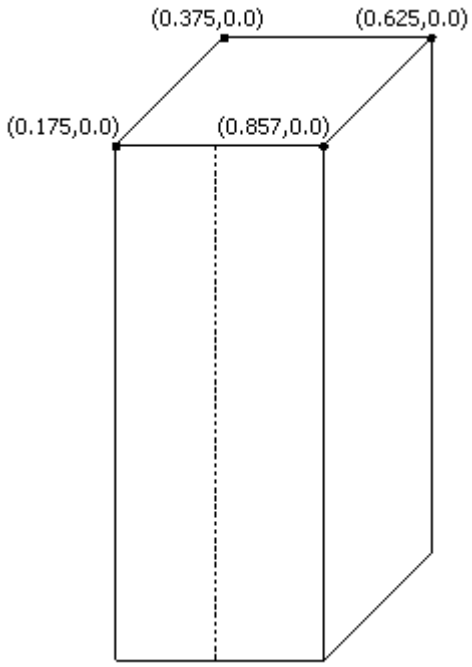
This diagram shows how wrapping in the u- direction affects how the system interpolates texture coordinates. Using the same points as in the example for normal, or nonwrapped, textures, you can see that the shortest route between points A and B is no longer across the middle of the texture; it's now across the border where 0.0 and 1.0 exist together. Wrapping in the v-direction is similar, except that it wraps the texture around a cylinder that is lying on its side. Wrapping in both the u- and v-directions is a more complex. In this situation, you can envision the texture as a torus, or doughnut.

The most common practical application for texture wrapping is to perform environment mapping. Usually, an object textured with an environment map appears very reflective, showing a mirrored image of the object's surroundings in the scene. For the sake of this discussion, picture a room with four walls, each one painted with a letter R, G, B, Y and the corresponding colors: red, green, blue, and yellow. The environment map for such a simple room might look like the following illustration.

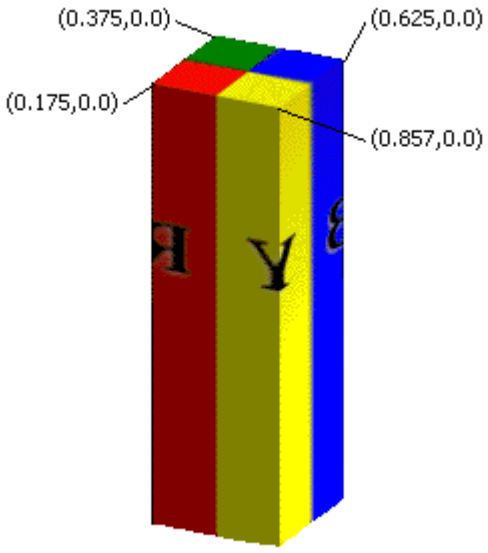


Imagine that the room's ceiling is held up by a perfectly reflective, four-sided, pillar. Mapping the environment map texture to the pillar is simple; making the pillar look as though it's reflecting the letters and colors is not as easy. The following diagram shows a wire frame of the pillar with the

applicable texture coordinates listed near the top vertices. The seam where wrapping will cross the edges of the texture is shown with a dotted line.



With wrapping enabled in the u- direction, the textured pillar shows the colors and symbols from the environment map appropriately and, at the seam in the front of the texture, the rasterizer properly chooses the shortest route between the texture coordinates, assuming that u-coordinates 0.0 and 1.0 share the same location. The textured pillar looks like the following illustration.



If texture wrapping isn't enabled, the rasterizer does not interpolate in the direction needed to generate a believable, reflected image. Rather, the area at the front of the pillar contains a horizontally compressed version of the texels between u- coordinates 0.175 and 0.875, as they pass through the center of the texture. The wrap effect is ruined.

Using Texture Wrapping

To enable texture wrapping, call the **IDirect3DDevice8::SetRenderState** method as shown in the code example below.

```
d3dDevice->SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_0);
```

The first parameter accepted by **SetRenderState** is a render state to set. Specify one of the D3DRS_WRAP0 through D3DRS_WRAP7 enumerated values which specify which texture level to set the wrapping for. Specify the D3DWRAPCOORD_0 through D3DWRAPCOORD_3 flags in the second parameter to enable texture wrapping in the corresponding direction, or combine them to enable wrapping in multiple directions. If you omit a flag, texture wrapping in the corresponding direction is disabled. To disable texture wrapping for a set of texture coordinates, set the value for the corresponding render state to 0.

Note Do not confuse texture wrapping with the similarly named texture addressing modes. For more information, see Texture Addressing Modes.

Microsoft DirectX 8.1 (C++)

Compressed Texture Resources

[This is preliminary documentation and is subject to change.]

Texture maps are digitized images drawn on three-dimensional shapes to add visual detail. They are mapped onto these shapes during rasterization, and the process can consume large amounts of both the system bus and memory. To reduce the amount of memory consumed by textures, Microsoft® Direct3D® supports the compression of texture surfaces. Some Direct3D devices support compressed texture surfaces natively. On such devices, once you have created a compressed surface and loaded the data into it, the surface can be used in Direct3D like any other texture surface. Direct3D handles decompression when the texture is mapped to a 3-D object.

Storage Efficiency and Texture Compression

All texture compression formats are powers of 2. While this does not mean that a texture is necessarily square, it does mean that both X and Y are powers of 2. For example, if a texture is originally 512×128 bytes, then the next mipmapping would be 256×64 and so on, with each level decreasing by a power of 2. At lower levels, where the texture is filtered to 16×2 and 8×1, there will be wasted bits because the compression block is always a 4×4 block of texels. Unused portions of the block are padded. Although there are wasted bits at the lowest levels, the overall gain is still significant. The worst case is, in theory, a 2K×1 texture (2^0 power). Here, only a single row of pixels is encoded per block, while the rest of the block is unused.

Mixing Formats Within a Single Texture

It is important to note that any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.

Once the 128-bit blocks are used, the alpha channel must be specified in either explicit (format DXT2 and DXT3) or interpolated mode (format DXT4 and DXT5) for the entire texture. As with color, once interpolated mode (format DXT4 and DXT5) is selected, then either eight interpolated

alphas or six interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

Microsoft® Direct3D® provides services to compress surfaces that are used for texturing 3-D models. This section provides information on creating and manipulating the data in a compressed texture surface.

Information is contained in the following topics.

- Opaque and One-bit Alpha Textures
- Textures with Alpha Channels
- Compressed Texture Formats
- Using Compressed Textures

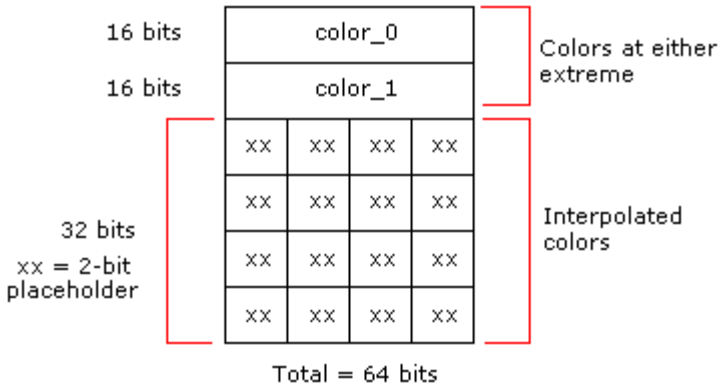
Microsoft DirectX 8.1 (C++)

Opaque and One-Bit Alpha Textures

[This is preliminary documentation and is subject to change.]

Texture format DXT1 is for textures that are opaque or have a single transparent color.

For each opaque or one-bit alpha block, two 16-bit values (RGB 5:6:5 format) and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits for 16 texels, or four bits per texel. In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation. This layout is shown in the following diagram.



The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors are used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to RGB 5:6:5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels.

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to RGBA 5:5:5:1, where the final bit is used for encoding the alpha mask.

The following code example illustrates the algorithm for deciding whether three- or four-color

encoding is selected.

```
if (color_0 > color_1)
{
    // Four-color block: derive the other two colors.
    // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
    // These 2-bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
    color_2 = (2 * color_0 + color_1 + 1) / 3;
    color_3 = (color_0 + 2 * color_1 + 1) / 3;
}
else
{
    // Three-color block: derive the other color.
    // 00 = color_0,  01 = color_1,  10 = color_2,
    // 11 = transparent.
    // These 2-bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
     color_2 = (color_0 + color_1) / 2;
     color_3 = transparent;

}
```

It is recommended that you set the RGBA components of the transparency pixel to zero before blending.

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel [1][2] refers to the texture map pixel at (x,y) = (2,1).

This table contains the memory layout for the 8-byte (64-bit) block.

Word address	**16-bit word**
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1—the colors at the two extremes—are laid out as follows:

Bits	**Color**
4:0 (LSB)	Blue color component
10:5	Green color component
15:11	Red color component

Bitmap Word_0 is laid out as follows:

Bits	**Texel**
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]

11:10	Texel[1][1]
13:12	Texel[1][2]
15:14 (MSB)	Texel[1][3]

Bitmap Word_1 is laid out as follows:

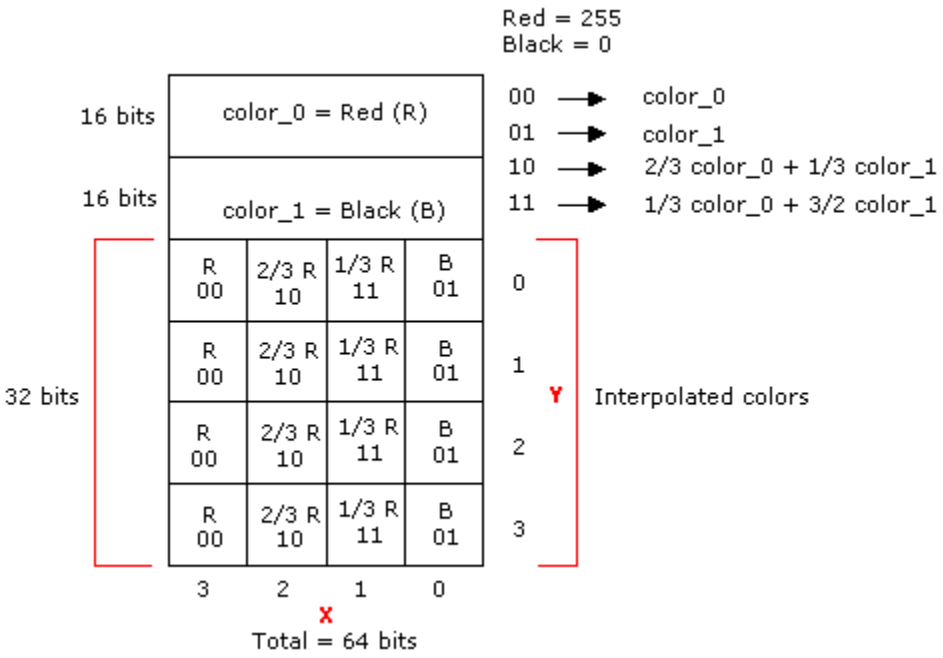
Bits	**Texel**
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

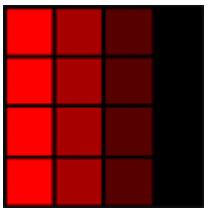
As an example of opaque encoding, assume that the colors red and black are at the extremes. Red is color_0, and black is color_1. There are four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used.

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

The bitmap then looks like the following graphic.



This looks like the following series of colors.



Example of One-Bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format can be used is leaves on a tree, shown against a blue sky. Some texels can be marked as transparent while three shades of green are still available for the leaves. Two colors fix the extremes, and the third is an interpolated color.

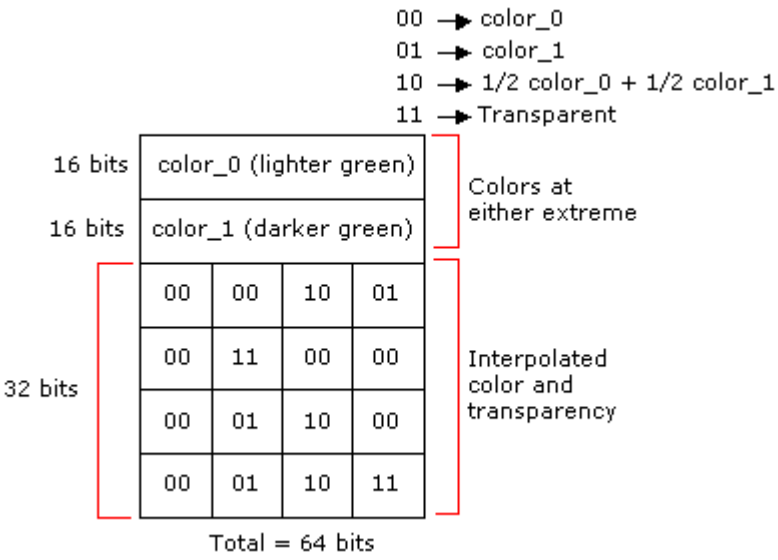
Here is an example of such a picture.

Note that where the image is shown as white, the texel would be encoded as transparent. Also note that the RGBA components of the transparent texels should be set to zero before blending.

The bitmap encoding for the colors and the transparency is determined using the following calculations.

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11   ?   Transparent
```

The bitmap then looks like the following graphic.



Microsoft DirectX 8.1 (C++)

Textures with Alpha Channels

[This is preliminary documentation and is subject to change.]

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation that is analogous to what is used for color encoding.

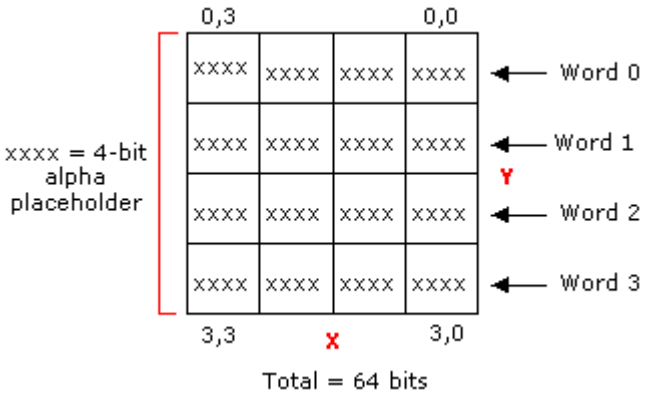
The transparency block and the color block are arranged as shown in the following table.

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These four bits can be achieved through a variety of means such as dithering or by simply using the four most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

The following diagram shows a 64-bit transparency block.



Note The compression method of Microsoft® Direct3D® uses the four most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This table contains the layout for Word 0.

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]

15:12 (MSB)	[0][3]

This table contains the layout for Word 1.

Bits	**Alpha**
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This table contains the layout for Word 2.

Bits	**Alpha**
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This table contains the layout for Word 3.

Bits	**Alpha**
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

The difference between DXT2 and DXT3 is that in the DXT2 format it is assumed that the color data has been premultiplied by alpha. In the DXT3 format it is assumed the color is not premultiplied by alpha. These two formats are needed because in most cases by the time a texture is used, simply examining the data is not sufficient to determine if the color values have been multiplied by alpha. Because this information is needed at run time, the two FOURCC codes are used to differentiate these cases. However, the data and interpolation method used for these two formats is identical.

The color compare used in DXT1 to determine if the texel is transparent is not used in this format. It is assumed that without the color compare the color data is always treated as if in 4-color mode. In other words, the *if* statement at the top of the DXT1 code should be the following:

```
if ((color_0 > color_1) OR !DXT1) {
```

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following code example illustrates this algorithm.

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
    // 8-alpha block:  derive the other six alphas.
    // Bit code 000 = alpha_0, 001 = alpha_1, others are interpolated.
    alpha_2 = (6 * alpha_0 + 1 * alpha_1 + 3) / 7;    // bit code 010
    alpha_3 = (5 * alpha_0 + 2 * alpha_1 + 3) / 7;    // bit code 011
    alpha_4 = (4 * alpha_0 + 3 * alpha_1 + 3) / 7;    // bit code 100
    alpha_5 = (3 * alpha_0 + 4 * alpha_1 + 3) / 7;    // bit code 101
    alpha_6 = (2 * alpha_0 + 5 * alpha_1 + 3) / 7;    // bit code 110
    alpha_7 = (1 * alpha_0 + 6 * alpha_1 + 3) / 7;    // bit code 111
}
else {
    // 6-alpha block.
    // Bit code 000 = alpha_0, 001 = alpha_1, others are interpolated.
    alpha_2 = (4 * alpha_0 + 1 * alpha_1 + 2) / 5;    // Bit code 010
    alpha_3 = (3 * alpha_0 + 2 * alpha_1 + 2) / 5;    // Bit code 011
    alpha_4 = (2 * alpha_0 + 3 * alpha_1 + 2) / 5;    // Bit code 100
    alpha_5 = (1 * alpha_0 + 4 * alpha_1 + 2) / 5;    // Bit code 101
    alpha_6 = 0;                                      // Bit code 110
    alpha_7 = 255;                                    // Bit code 111
}
```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

The difference between DXT4 and DXT5 is that in the DXT4 format it is assumed that the color data has been premultiplied by alpha. In the DXT5 format, it is assumed the color is not premultiplied by alpha. These two formats are needed because, in most cases, by the time a texture is used simply examining the data is not sufficient to determine if the color values have been multiplied by alpha. Because this information is needed at run time, the two FOURCC codes are used to differentiate these cases. However, the data and interpolation method used for these two formats is identical.

The color compare used in DXT1 to determine if the texel is transparent is not used with these formats. It is assumed that without the color compare the color data is always treated as if in 4-color mode. In other words the *if* statement at the top of the DXT1 code should be:

```
if ((color_0 > color_1) OR !DXT1) {
```

Microsoft DirectX 8.1 (C++)

Compressed Texture Formats

[This is preliminary documentation and is subject to change.]

This section contains information on the internal organization of compressed texture formats. You do

not need these details to use compressed textures, because you can use Microsoft® Direct3DX functions for conversion to and from compressed formats. However, this information is useful if you want to operate on compressed surface data directly.

Direct3D uses a compression format that divides texture maps into 4x4 texel blocks. If the texture contains no transparency—is opaque—or if the transparency is specified by a one-bit alpha, an 8-byte block represents the texture map block. If the texture map does contain transparent texels, using an alpha channel, a 16-byte block represents it.

- Opaque and One-bit Alpha Textures
- Textures with Alpha Channels

Note Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.

When 128-bit blocks are used, the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. As with color, once interpolated mode is selected, either eight interpolated alphas or six interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

The pitch for DXTn formats is different from what was returned in DirectX 7. It now refers the pitch of a row of blocks. For example, if you have a width of 16, then you will have a pitch of four blocks (4*8 for DXT1, 4*16 for DXT2-5.)

Microsoft DirectX 8.1 (C++)

Using Compressed Textures

[This is preliminary documentation and is subject to change.]

Determining Support for Compressed Textures

Before your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling the **IDirect3D8::CheckDeviceFormat** method. This method determines whether a surface format can be used as a texture on a device representing the adapter. To test the adapter, specify any pixel format that uses the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs). If **CheckDeviceFormat** returns D3D_OK, the device can create texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with Microsoft® Direct3D® by calling the **IDirect3DDevice8::SetTexture** method. The following code example shows how to determine if the adapter supports a compressed texture format.

```
BOOL IsCompressedTextureFormatOk( D3DFORMAT TextureFormat,
                                  D3DFORMAT AdapterFormat ) {
    HRESULT hr = pD3D->CheckDeviceFormat( D3DADAPTER_DEFAULT,
                                          D3DDEVTYPE_HAL,
                                          AdapterFormat,
                                          0,
                                          D3DRTYPE_TEXTURE,
                                          TextureFormat);
```

```
    return SUCCEEDED( hr );
}
```

If the device does not support texturing from compressed texture surfaces, you can still store texture data in a compressed format surface, but you must convert any compressed textures to a supported format before they can be used for texturing.

Creating Compressed Textures

After creating a device that supports a compressed texture format on the adapter, you can create a compressed texture resource. Call **IDirect3DDevice8::CreateTexture** and specify a compressed texture format for the *Format* parameter.

Before loading an image into a texture object, retrieve a pointer to the texture surface by calling the **IDirect3DTexture8::GetSurfaceLevel** method.

Now you can use any Direct3DX function that begins with D3DXLoadSurface to load an image into the surface that was retrieved by using **GetSurfaceLevel**. These functions handle conversion to and from compressed texture formats.

You can create and convert compressed texture (DDS) files using the DXTex Tool supplied with the SDK. You can also create your own DDS files.

The advantage of this behavior is that an application can copy the contents of a compressed surface to a file without calculating how much storage is required for a surface of a particular width and height in the specific format.

The following table shows the five types of compressed textures. For more information on how the data is stored, see Compressed Texture Formats. You only need this information if you are writing your own compression routines.

FOURCC	Description	Alpha-premultiplied?
DXT1	Opaque / one-bit alpha	N/A
DXT2	Explicit alpha	Yes
DXT3	Explicit alpha	No
DXT4	Interpolated alpha	Yes
DXT5	Interpolated alpha	No

Note When you transfer data from a non-premultiplied format to a premultiplied format, Direct3D scales the colors based on the alpha values. Transferring data from a premultiplied format to a non-premultiplied format is not supported. If you try to transfer data from a premultiplied-alpha source to a non-premultiplied-alpha destination, the method returns D3DERR_INVALIDCALL. If you transfer data from a premultiplied-alpha source to a destination that has no alpha, the source color components, which have been scaled by alpha, are copied as is.

Decompressing Compressed Texture Surfaces

As with compressing a texture surface, decompressing a compressed texture is performed through Microsoft® Direct3D® copying services.

To copy a compressed texture surface to a uncompressed texture surface, use the function

D3DXLoadSurfaceFromSurface. This functions handles compression to and from compressed and uncompressed surfaces.

Microsoft DirectX 8.1 (C++)

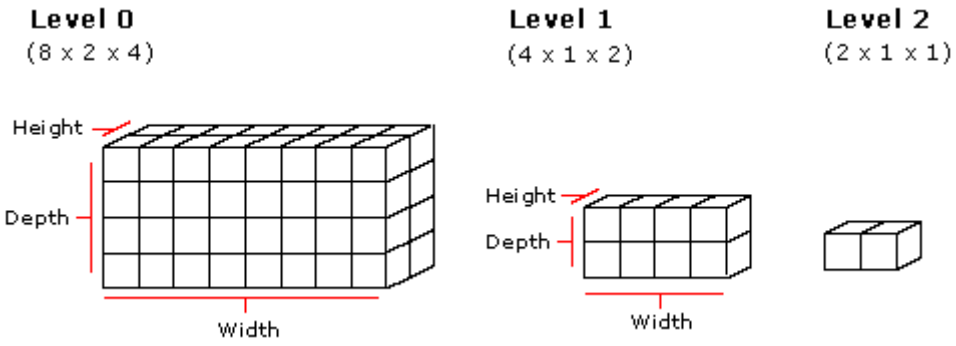
Volume Texture Resources

[This is preliminary documentation and is subject to change.]

Volume textures are three-dimensional collections of pixels (texels) that can be used to paint a two-dimensional primitive such as a triangle or a line. Three-element texture coordinates are required for each vertex of a primitive that is to be textured with a volume. As the primitive is drawn, each contained pixel is filled with the color value from some pixel within the volume, in accordance with rules analogous to the two-dimensional texture case. Volumes are not rendered directly because there are no three-dimensional primitives that can be painted with them.

You can use volume textures for special effects such as patchy fog, explosions, and so on.

Volumes are organized into slices and can be thought of as *width* x *height* 2-D surfaces stacked to make a *width* x *height* x *depth* volume. Each slice is a single row. Volumes can have subsequent levels, in which the dimensions of each level are truncated to half the dimensions of the previous level. The illustration below gives an idea of what a volume texture with multiple levels looks like.



Creating a Volume Texture

The code examples below show the steps required to use a volume texture.

First, specify a custom vertex type that has three texture coordinates for each vertex, as shown in this code example.

```
struct VOLUMEVERTEX
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu, tv, tw;
};

#define D3DFVF_VOLUMEVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|
                             D3DFVF_TEX1|D3DFVF_TEXCOORDSIZE3(0))
```

Next, fill the vertices with data.

```
VOLUMEVERTEX g_vVertices[4] =
```

```
{
    { 1.0f, 1.0f, 0.0f, 0xffffffff, 1.0f, 1.0f, 0.0f },
    {-1.0f, 1.0f, 0.0f, 0xffffffff, 0.0f, 1.0f, 0.0f },
    { 1.0f,-1.0f, 0.0f, 0xffffffff, 1.0f, 0.0f, 0.0f },
    {-1.0f,-1.0f, 0.0f, 0xffffffff, 0.0f, 0.0f, 0.0f }
};
```

Now, create a vertex buffer and fill it with data from the vertices.

The next step is to use the **IDirect3DDevice8::CreateVolumeTexture** method to create a volume texture, as shown in this code example.

```
LPDIRECT3DVOLUMETEXTURE8 volTexture;

d3dDevice->CreateVolumeTexture( 8, 4, 4, 1, 0, D3DFMT_R8G8B8,
                                D3DPOOL_MANAGED, &volTexture );
```

Before rendering the primitive, set the current texture to the volume texture created above. The code example below shows the entire rendering process for a strip of triangles.

```
if( SUCCEEDED( d3dDevice->BeginScene() ) )
{
    // Draw the quad, with the volume texture.
    d3dDevice->SetTexture( 0, pVolumeTexture );
    d3dDevice->SetVertexShader( D3DFVF_VOLUMEVERTEX );
    d3dDevice->SetStreamSource( 0, pVB, sizeof(VOLUMEVERTEX) );
    d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2);

   // End the scene.
   d3dDevice->EndScene();
}
```

Microsoft DirectX 8.1 (C++)

Automatic Texture Management

[This is preliminary documentation and is subject to change.]

Texture management is the process of determining which textures are needed for rendering at a given time and ensuring that those textures are loaded into video memory. As with any algorithm, texture management schemes vary in complexity, but any approach to texture management involves the following key tasks.

- Tracking the amount of available texture memory.
- Calculating which textures are needed for rendering, and which are not.
- Determining which existing texture resources can be reloaded with another texture image, and which surfaces should be destroyed and replaced with new texture resources.

Microsoft® Direct3D® implements system-supported texture management to ensure that textures are loaded for optimal performance. Texture resources that Direct3D manages are referred to as *managed textures*.

The texture manager tracks textures with a time-stamp that identifies when the texture was last used. It then uses a least-recently-used algorithm to determine which textures should be removed. Texture priorities are used as tie breakers when two textures are targeted for removal from memory. If two textures have the same priority value, the least-recently-used texture is removed. However, if two

textures have the same time-stamp, the texture that has a lower priority is removed first.

You request automatic texture management for texture surfaces when you create them. To retrieve a managed texture in a C++ application, create a texture resource by calling **IDirect3DDevice8::CreateTexture** and specifying the D3DPOOL_MANAGED for the *Pool* parameter. You are not allowed to specify where you want the texture created. You cannot use the D3DPOOL_DEFAULT or D3DPOOL_SYSTEMMEM flags when creating a managed texture. After creating the managed texture, you can call the **IDirect3DDevice8::SetTexture** method to set it to a stage in the rendering device's texture cascade.

You can assign a priority to managed textures by calling the **IDirect3DResource8::SetPriority** method for the texture surface.

Direct3D automatically downloads textures into video memory as needed. The system might cache managed textures in local or nonlocal video memory, depending on the availability of nonlocal video memory or other factors. The cache location of your managed textures is not communicated to your application, nor is this information required to use automatic texture management. If your application uses more textures than can fit in video memory, Direct3D removes older textures from video memory to make room for the new textures. If you use a removed texture again, the system uses the original system-memory texture surface to reload the texture in the video-memory cache. Although reloading the texture is necessary, it also decreases the application's performance.

You can dynamically modify the original system-memory copy of the texture by updating or locking the texture resource. When the system detects a dirty surface—after an update is completed, or when the surface is unlocked—the texture manager automatically updates the video-memory copy of the texture. The performance hit incurred is similar to reloading a removed texture.

When entering a new level in a game, your application might need to flush all managed textures from video memory. You can explicitly request to remove all managed textures by calling the **IDirect3DDevice8::ResourceManagerDiscardBytes** method and specifying a value of 0 for the *Bytes* parameter. When you call this method, Direct3D destroys any cached local and nonlocal video-memory textures, but leaves the original system-memory copies untouched.

For more information on resource management, see Managing Resources.

Microsoft DirectX 8.1 (C++)

Hardware Considerations for Texturing

[This is preliminary documentation and is subject to change.]

Current hardware does not necessarily implement all the functionality that the Microsoft® Direct3D® interface enables. Your application must test user hardware and adjust its rendering strategies accordingly.

Many 3-D accelerator cards do not support diffuse iterated values as arguments to blending units. However, your application can introduce iterated color data when it performs texture blending.

Some 3-D hardware may not have a blending stage associated with the first texture. On these adapters, your application must perform blending in the second and third texture stages in the set of current textures.

Due to limitations in much of today's hardware, few display adapters can perform trilinear mipmap interpolation through the multiple texture blending interface offered by **IDirect3DDevice8**. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTEXF_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTEXF_POINT mipmap filter mode, which is widely supported.

Microsoft DirectX 8.1 (C++)

Texture Blending

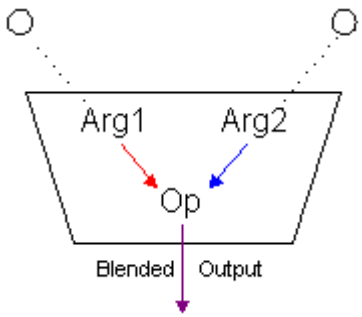
[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® can blend as many as eight textures onto primitives in a single pass. The use of multiple texture blending can profoundly increase the frame rate of a Direct3D application. An application employs multiple texture blending to apply textures, shadows, specular lighting, diffuse lighting, and other special effects in a single pass.

To use texture blending, your application should first check if the user's hardware supports it. This information is found in the **TextureCaps** member of the **D3DCAPS8** structure. For details on how to query the user's hardware for texture blending capabilities, see **IDirect3DDevice8::GetDeviceCaps**.

Texture Stages and the Texture Blending Cascade

Microsoft® Direct3D® supports single-pass multiple texture blending through the use of *texture stages*. A texture stage takes two arguments and performs a blending operation on them, passing on the result for further processing or for rasterization. You can visualize a texture stage as shown in the following illustration.

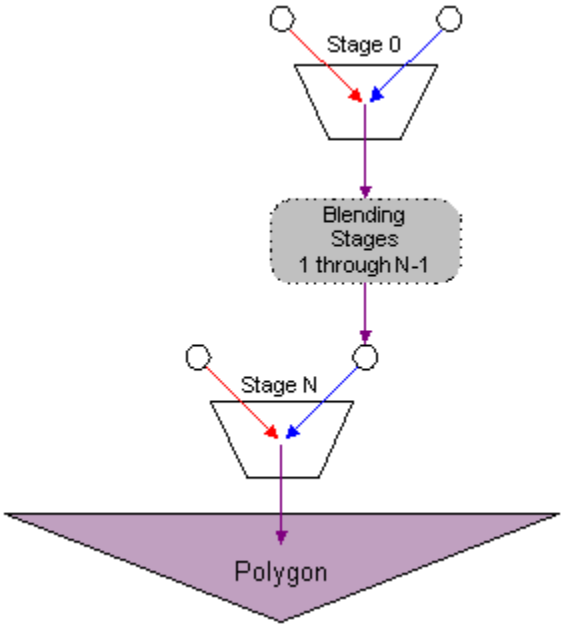


As the preceding illustration shows, texture stages blend two arguments by using a specified operator. Common operations include simple modulation or addition of the color or alpha components of the arguments, but more than two dozen operations are supported. The arguments for a stage can be an associated texture, the iterated color or alpha (iterated during Gouraud shading), arbitrary color and alpha, or the result from the previous texture stage. For more information, see Texture Blending Operations and Arguments.

Note Direct3D distinguishes color blending from alpha blending. Applications set blending operations and arguments for color and alpha individually, and the results of those settings are independent of one another.

The combination of arguments and operations used by multiple blending stages define a simple flow-based blending language. The results from one stage flow down to another stage, from that stage to the next, and so on. The concept of results flowing from stage to stage to eventually be rasterized on

a polygon is often called the *texture blending cascade*. The following illustration shows how individual texture stages make up the texture blending cascade.



Each stage in a device has a zero-based index. Direct3D allows up to eight blending stages, although you should always check device capabilities to determine how many stages the current hardware supports. The first blending stage is at index 0, the second is at 1, and so on, up to index 7. The system blends stages in increasing index order.

Use only the number of stages you need; the unused blending stages are disabled by default. So, if your application only uses the first two stages, it need only set operations and arguments for stage 0 and 1. The system blends the two stages, and ignores the disabled stages.

Optimization Note If your application varies the number of stages it uses for different situations—such as four stages for some objects, and only two for others—you don't need to explicitly disable all previously used stages. One option is to disable the color operation for the first unused stage, then all stages with a higher index will not be applied. Another option is to disable texture mapping altogether by setting the color operation for the first texture stage (stage 0). A third option is When a texture stage has D3DTSS_COLORARG1 equal to D3DTA_TEXTURE and the texture pointer for the stage is NULL, this stage and all stages after it are not processed.

Additional information is contained in the following topics.

- Texture Blending Operations and Arguments
- Assigning the Current Textures
- Creating Blending Stages
- Alpha Texture Blending
- Multipass Texture Blending
- Light Mapping with Textures

Microsoft DirectX 8.1 (C++)

Texture Blending Operations and Arguments

[This is preliminary documentation and is subject to change.]

Applications associate a blending stage with each texture in the set of current textures. Microsoft® 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 texture stage is used by calling **IDirect3DDevice8::SetTextureStageState**. You can set separate operations for the color and alpha channels, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state; specify alpha operations by using D3DTSS_ALPHAOP. Both stage states use values from the **D3DTEXTUREOP** enumerated type.

Texture blending arguments use the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHARG1, and D3DTSS_ALPHARG2 members of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The corresponding argument values are identified using Texture Argument Flags.

Note You can disable a texture stage—and any subsequent texture blending stages in the cascade—by setting the color operation for that stage to D3DTOP_DISABLE. Disabling the color operation effectively disables the alpha operation as well. Alpha operations cannot be disabled when color operations are enabled. Setting the alpha operation to D3DTOP_DISABLE when color blending is enabled causes undefined behavior.

To determine the supported texture blending operations of a device, query the **TextureCaps** member of the **D3DCAPS8** structure.

Microsoft DirectX 8.1 (C++)

Assigning the Current Textures

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® maintains a list of up to eight current textures. It blends these textures onto all the primitive it renders. Only textures created as texture interface pointers can be used in the set of current textures.

Applications call the **IDirect3DDevice8::SetTexture** method to assign textures into the set of current textures. The first parameter must be from the a number in the range of 0-7, inclusive. Pass the texture interface pointer as the second parameter.

The following C++ code example demonstrates how a texture can be assigned to the set of current textures.

```
// This code example assumes that the variable lpd3dDev is a
// valid pointer to an IDirect3DDevice8 interface and pTexture
// is a valid pointer to an IDirect3DBaseTexture8 interface.

// Set the third texture.
d3dDevice->SetTexture(2, pTexture);
```

Note Software devices do not support assigning a texture to more than one texture stage at a time.

Microsoft DirectX 8.1 (C++)

Creating Blending Stages

[This is preliminary documentation and is subject to change.]

A blending stage is a set of texture operations and their arguments that define how textures are blended. When making a blending stage, C++ applications invoke the **IDirect3DDevice8::SetTextureStageState** method. The first call specifies the operation that is performed. Two additional invocations define the arguments to which the operation is applied. The following code example illustrates the creation of a blending stage.

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice8 interface.

// Set the operation for the first texture.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_ADD);

// Set argument 1 to the texture color.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);

// Set argument 2 to the iterated diffuse color.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
```

Texel data in textures contain color and alpha values. Applications can define separate operations for both color and alpha values in a single blending stage. Each operation, color and alpha, has its own arguments. For details, see **D3DTEXTURESTAGESTATETYPE**.

Although not part of the Microsoft® Direct3D® API, you can insert the following macros into your application to abbreviate the code required for creating texture blending stages.

```
#define SetTextureColorStage( dev, i, arg1, op, arg2 )     \
   dev->SetTextureStageState( i, D3DTSS_COLOROP, op);      \
   dev->SetTextureStageState( i, D3DTSS_COLORARG1, arg1 ); \
   dev->SetTextureStageState( i, D3DTSS_COLORARG2, arg2 );

#define SetTextureAlphaStage( dev, i, arg1, op, arg2 )     \
   dev->SetTextureStageState( i, D3DTSS_ALPHAOP, op);      \
   dev->SetTextureStageState( i, D3DTSS_ALPHARG1, arg1 );  \
   dev->SetTextureStageState( i  D3DTSS_ALPHARG2, arg2 );
```

Microsoft DirectX 8.0 (C++)

Alpha Texture Blending

[This is preliminary documentation and is subject to change.]

When Microsoft® Direct3D® renders a primitive, it generates a color for the primitive based on the primitive's material, or the colors of its vertices, and lighting information. For details, see Lights and Materials. If an application enables texture blending, Direct3D must then combine the color value of the processed polygon pixel with the pixel already stored in the frame buffer. Direct3D uses the following formula to determine the final color for each pixel in the primitive's image.

FinalColor = *TexelColor* × *SourceBlendFactor* + *PixelColor* × *DestBlendFactor*

In this formula, *FinalColor* is the final pixel color that is output to the target rendering surface. *TexelColor* represents the incoming color value, after texture filtering, that corresponds to the current pixel. For details on how Direct3D maps texels to pixels, see Texture Filtering. *SourceBlendFactor*

is a calculated value that Direct3D uses to determine the percentage of the incoming color value to apply to the final color. *PixelColor* is the color of the pixel currently stored in the primitive's image. *DestBlendFactor* represents the percentage of the current pixel's color that will be used in the final rendered pixel. The values of *SourceBlendFactor* and *DestBlendFactor* range from 0.0 to 1.0 inclusive.

As you can see from the preceding formula, the final rendered pixel is not rendered as transparent if the *SourceBlendFactor* is D3DBLEND_ONE and the *DestBlendFactor* is D3DBLEND_ZERO. It is completely transparent if the *SourceBlendFactor* is D3DBLEND_ZERO and the *DestBlendFactor* is D3DBLEND_ONE. If an application sets these factors to any other values, the resulting final rendered pixel is blended with some degree of transparency.

After texture filtering, every pixel color value has red, green, and blue color values. By default, Direct3D uses D3DBLEND_SRCALPHA as the *SourceBlendFactor* and D3DBLEND_INVSRCALPHA as the *DestBlendFactor*. Therefore, applications can control the transparency of processed pixels by setting the alpha values in textures.

A C++ application controls the blending factors with the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Invoke the **IDirect3DDevice8::SetRenderState** method and pass either render state value as the value of the first parameter. The second parameter must be a member of the **D3DBLEND** enumerated type.

Microsoft DirectX 8.0 (C++)

Multipass Texture Blending

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® applications can achieve numerous special effects by applying various textures to a primitive over the course of multiple rendering passes. The common term for this is multipass texture blending. A typical use for multipass texture blending is to emulate the effects of complex lighting and shading models by applying multiple colors from several different textures. One such application is called light mapping. For more information, see Light Mapping with Textures.

Note Some devices are capable of applying multiple textures to primitives in a single pass. For details, see Texture Blending.

If the user's hardware does not support multiple texture blending, your application can use multipass texture blending to achieve the same visual effects. However, the application cannot sustain the frame rates that are possible when using multiple texture blending.

To perform multipass texture blending in a C/C++ application.

1. Set a texture in texture stage 0 by calling the **IDirect3DDevice8::SetTexture** method.
2. Select the desired color and alpha blending arguments and operations with the **IDirect3DDevice8::SetTextureStageState** method. The default settings are well-suited for multipass texture blending.
3. Render the appropriate objects in the scene.
4. Set the next texture in texture stage 0.
5. Set the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states to adjust the source and destination blending factors as needed. The system blends the new textures with the existing pixels in the render-target surface according to these parameters.

6. Repeat Steps 3, 4, and 5 with as many textures as needed.

Microsoft DirectX 8.1 (C++)

Light Mapping with Textures

[This is preliminary documentation and is 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. Microsoft® Direct3D® supports multipass and multiple texture blending. These capabilities enable your application to render scenes with a more realistic appearance than scenes rendered with shading techniques alone. By applying one or more light maps, your application can map areas of light and shadow onto its primitives.

A light map is a texture or group of textures that contains information about lighting in a 3-D scene. You can store the lighting information in the alpha values of the light map, in the color values, or in both.

If you implement light mapping using multipass texture blending, your application should render the light map onto its primitives on the first pass. It should use a second pass to render the base texture. The exception to this is specular light mapping. In that case, render the base texture first; then add the light map.

Multiple texture blending enables your application to render the light map and the base texture in one pass. If the user's hardware provides for multiple texture blending, your application should take advantage of it when performing light mapping. This significantly improves your application's performance.

Using light maps, a Direct3D application can achieve a variety of lighting effects when it renders primitives. It can map not only monochrome and colored lights in a scene, but it can also add details such as specular highlights and diffuse lighting.

Information on using Direct3D texture blending to perform light mapping is presented in the following topics.

- Monochrome Light Maps
- Color Light Maps
- Specular Light Maps
- Diffuse Light Maps

Microsoft DirectX 8.1 (C++)

Monochrome Light Maps

[This is preliminary documentation and is 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 for more information. These adapters also generally do not support multiple texture blending. If your application is running on an adapter such as this, it

can use multipass texture blending to perform monochrome light mapping.

To perform monochrome light mapping, an application stores the lighting information in the alpha data of its light map textures. The application uses the texture filtering capabilities of Microsoft® Direct3D® to perform a mapping from each pixel in the primitive's image to a corresponding texel in the light map. It sets the source blending factor to the alpha value of the corresponding texel.

The following C++ code example illustrates how an application can use a texture as a monochrome light map.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface and that lptexLightMap is a valid
// pointer to a texture that contains monochrome light map data.

// Set the light map texture as the current texture.
d3dDevice->SetTexture(0, lptexLightMap);

// Set the color operation.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

// Set argument 1 to the color operation.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1,
       D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE);
```

Because display adapters that do not support destination alpha blending usually do not support multiple texture blending, this example sets the light map as the first texture, which is available on all 3-D accelerator cards. The sample code sets the color operation for the texture's blending stage to blend the texture data with the primitive's existing color. It then selects the first texture and the primitive's existing color as the input data.

Microsoft DirectX 8.1 (C++)

Color Light Maps

[This is preliminary documentation and is 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 C++ code example demonstrates light mapping with RGB color data.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface and that lptexLightMap is a valid
// pointer to a texture that contains RGB light map data.

// Set the light map texture as the first texture.
d3dDevice->SetTexture(0, lptexLightMap);

d3dDevice->SetTextureStageState( 0,D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

This example sets the light map as the first texture. It then sets the state of the first blending stage to modulate the incoming texture data. It uses the first texture and the current color of the primitive as the arguments to the modulate operation.

Microsoft DirectX 8.1 (C++)

Specular Light Maps

[This is preliminary documentation and is subject to change.]

When illuminated by a light source, shiny objects—those that use highly reflective materials—receive specular highlights. In some cases, the specular highlights produced by the lighting module is not accurate. To produce a more appealing highlight, many Microsoft® Direct3D® applications apply specular light maps to primitives.

To perform specular light mapping, first modulate the specular light map with the primitive's existing texture. Then add the monochrome or RGB light map.

The following code example illustrates this process in C++.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexSpecLightMap is a valid pointer to a texture that contains RGB
// specular light map data.
// lptexLightMap is a valid pointer to a texture that contains RGB
// light map data.

// Set the base texture.
d3dDevice->SetTexture(0, lptexBaseTexture );

// Set the base texture operation and arguments.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the specular light map.
d3dDevice->SetTexture(1, lptexSpecLightMap);

// Set the specular light map operation and args.
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// Set the RGB light map.
d3dDevice->SetTexture(2, lptexLightMap);

// Set the RGB light map operation and arguments.
d3dDevice->SetTextureStageState(2,D3DTSS_COLOROP, D3DTOP_MODULATE);
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Microsoft DirectX 8.1 (C++)

Diffuse Light Maps

[This is preliminary documentation and is subject to change.]

When illuminated by a light source, matte surfaces display diffuse light reflection. The brightness of diffuse light depends on the distance from the light source and the angle between the surface normal

and the light source direction vector. The diffuse lighting effects simulated by lighting calculations produce only general effects.

Your application can simulate more complex diffuse lighting with texture light maps. Do this by adding the diffuse light map to the base texture, as shown in the following C++ code example.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexDiffuseLightMap is a valid pointer to a texture that contains
// RGB diffuse light map data.

// Set the base texture.
d3dDevice->SetTexture(0,lptexBaseTexture );

// Set the base texture operation and args.
d3dDevice->SetTextureStageState(0,D3DTSS_COLOROP,
       D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the diffuse light map.
d3dDevice->SetTexture(1,lptexDiffuseLightMap );

// Set the blend stage.
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Microsoft DirectX 8.1 (C++)

Surfaces

[This is preliminary documentation and is subject to change.]

A surface represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Surface objects are contained within the **IDirect3DSurface8** interface.

An **IDirect3DSurface8** interface is obtained by calling one of the following methods.

- **IDirect3DCubeTexture8::GetCubeMapSurface**
- **IDirect3DDevice8::CreateDepthStencilSurface**
- **IDirect3DDevice8::CreateImageSurface**
- **IDirect3DDevice8::CreateRenderTarget**
- **IDirect3DDevice8::GetBackBuffer**
- **IDirect3DDevice8::GetDepthStencilSurface**
- **IDirect3DDevice8::GetFrontBuffer**
- **IDirect3DDevice8::GetRenderTarget**
- **IDirect3DSwapChain8::GetBackBuffer**
- **IDirect3DTexture8::GetSurfaceLevel**

The **IDirect3DSurface8** interface enables you to indirectly access memory through the **IDirect3DDevice8::CopyRects** method. This method allows you to copy a rectangular region of pixels from one **IDirect3DSurface8** interface to another **IDirect3DSurface8** interface. The surface interface also has methods to directly access display memory. For example, you can use the

IDirect3DSurface8::LockRect method to lock a rectangular region of display memory. It is important to call **IDirect3DSurface8::UnlockRect** after you are done working with the locked rectangular region on the surface.

Surface formats dictate how data for each pixel in surface memory is interpreted. Microsoft® Direct3D® uses the **D3DFORMAT** member of the **D3DSURFACE_DESC** structure to describe the surface format. You can retrieve the format of an existing surface by calling the **IDirect3DSurface8::GetDesc** method.

Additional information can be found in the following topics.

- Width vs. Pitch
- Flipping Surfaces
- Page Flipping and Back Buffering
- Copying To Surfaces
- Copying Surfaces
- Accessing Surface Memory Directly
- Private Surface Data
- Gamma Controls

Microsoft DirectX 8.1 (C++)

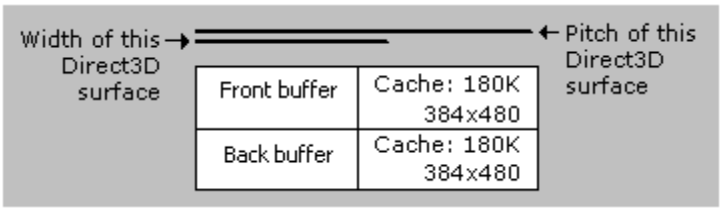
Width vs. Pitch

[This is preliminary documentation and is subject to change.]

Although the terms *width* and *pitch* are often used informally, they have very important, and distinctly different, meanings. As a result, you should understand the meanings for each, and how to interpret the values that Microsoft® Direct3D® uses to describe them.

Direct3D uses the **D3DSURFACE_DESC** structure to carry information describing a surface. Among other things, this structure is defined to contain information about a surface's dimensions, as well as how those dimensions are represented in memory. The structure uses the **Height** and **Width** members to describe the logical dimensions of the surface. Both members are measured in pixels. Therefore, the **Height** and **Width** values for a 640×480 surface are the same whether it is an 8-bit surface or a 24-bit RGB surface.

When you lock a surface using the **IDirect3DSurface8::LockRect** method, the method fills in a D3DLOCKED_RECT structure that contains the pitch of the surface and a pointer to the locked bits. The value in the **Pitch** member describes the surface's memory pitch, also called *stride*. Pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes rather than pixels, a 640×480×8 surface has a very different pitch value than a surface with the same dimensions but a different pixel format. Additionally, the pitch value sometimes reflects bytes that Direct3D has reserved as a cache, so it is not safe to assume that pitch is simply the width multiplied by the number of bytes per pixel. Rather, visualize the difference between width and pitch as shown in the following illustration.



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8.

When accessing surfaces directly, take care to stay within the memory allocated for the dimensions of the surface and stay out of any memory reserved for cache. Additionally, when you lock only a portion of a surface, you must stay within the rectangle you specify when locking the surface. Failing to follow these guidelines will have unpredictable results. When rendering directly into surface memory, always use the pitch returned by the **LockRect** method. Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of the problem.

For more information, see Accessing Surface Memory Directly.

Microsoft DirectX 8.1 (C++)

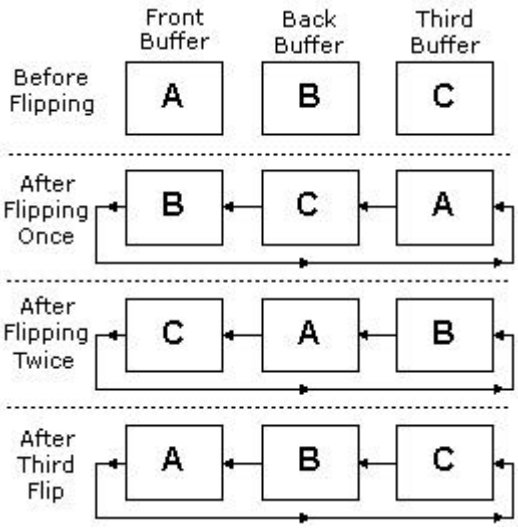
Flipping Surfaces

[This is preliminary documentation and is subject to change.]

A Microsoft® Direct3D® application typically displays an animated sequence by generating the frames of the animation in back buffers and presenting them in sequence. Back buffers are organized into swap chains. A swap chain is a series of buffers that "flip" to the screen one after another. This can be used to render one scene in memory and then flip the scene to the screen when rendering is complete. This avoids the phenomenon known as tearing and allows for smoother animation.

Each device created in Direct3D has at least one swap chain. When you initialize the first Direct3D device, you set the *BackBufferCount* member of the **D3DPRESENT_PARAMETERS** structure, which tells Direct3D the number of back buffers that will be in the swap chain. The call to **IDirect3D8::CreateDevice** then creates the Direct3D device and corresponding swap chain.

When you use the **IDirect3DDevice8::Present** method to request a surface flip operation, the pointers to surface memory for the front buffer and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. When a flipping chain contains a front buffer and more than one back buffer, the pointers are switched in a circular pattern, as shown in the following illustration.



You can create addition swap chains for a device by calling **IDirect3DDevice8::CreateAdditionalSwapChain**. An application can create one swap chain per view and associate each swap chain with a particular window. The application renders images in the back buffers of each swap chain, and then presents them individually. The two parameters that **CreateAdditionalSwapChain** takes are a pointer to a **D3DPRESENT_PARAMETERS** structure and the address of a pointer to an **IDirect3DSwapChain8** interface. You can then use **IDirect3DSwapChain8::Present** to display the contents of the next back buffer to the front buffer. Note that a device can only have one full-screen swap chain.

You can gain access to a specific back buffer by calling the **IDirect3DDevice8::GetBackBuffer** or **IDirect3DSwapChain8::GetBackBuffer** methods, which return a pointer to a **IDirect3DSurface8** interface that represents the returned back buffer surface. Note that calling this method increases the internal reference count on the **IDirect3DDevice8** interface so be sure to call **IUnknown::Release** when you are done using this surface or you will have a memory leak.

Remember, Direct3D flips surfaces by swapping surface memory pointers within the swap chain, not by swapping the surfaces themselves. This means that you will always render to the back buffer that will be displayed next.

It is important to note the distinction between a "flipping operation", as performed by a display adapter driver, and a "Present" operation applied to a swap chain created with D3DSWAPEFFECT_FLIP.

The term "flip" conventionally denotes an operation that alters the range of video memory addresses that a display adapter uses to generate its output signal, thus causing the contents of a previously hidden back buffer to be displayed. In DirectX 8, the term is often used more generally to describe the presentation of a back buffer in any swap chain created with the D3DSWAPEFFECT_FLIP swap effect.

While such "Present" operations are almost invariably implemented by flip operations when the swap chain is a full-screen one, they are necessarily implemented by copy operations when the swap chain is windowed. Furthermore, a display adapter driver may use flipping to implement Present operations against full-screen swap chains based on the D3DSWAPEFFECT_DISCARD, D3DSWAPEFFECT_COPY and D3DSWAPEFFECT_COPY_VSYNC swap effects.

The discussion above applies to the commonly used case of a full-screen swap chain created with D3DSWAPEFFECT_FLIP.

For a more general discussion of the different swap effects for both windowed and full-screen swap chains, see **D3DSWAPEFFECT**.

Microsoft DirectX 8.1 (C++)

Page Flipping and Back Buffering

[This is preliminary documentation and is subject to change.]

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

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

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

Microsoft DirectX 8.1 (C++)

Copying To Surfaces

[This is preliminary documentation and is subject to change.]

When using the **IDirect3DDevice8::CopyRects** method, you pass an array of rectangles on the source surface or NULL to specify the entire surface. You also pass an array of points on the destination surface to which the top-left position of each rectangle on the source image is copied. This method does not support clipping. The operation will fail unless all the source rectangles and their corresponding destination rectangles are completely contained within the source and destination surfaces respectively. This method does not support alpha blending, color keys, or format conversion. Note that the destination and source surfaces must be distinct.

CopyRects Example

The following example copies two rectangles from the source surface to a destination surface. The first rectangle is copied from (0, 0, 50, 50) on the source surface to the same location on the destination surface, and the second rectangle is copied from (50, 50, 100, 100) on the source surface to (150 ,150, 200, 200) on the destination surface.

```
//The following assumptions are made:
//-d3dDevice is a valid Direct3DDevice8 object.
//-pSource and pDest are valid IDirect3DSurface8 pointers.

RECT  rcSource[] = {  0,  0,  50,  50,
```

```
                      50, 50, 100, 100 };
POINT ptDest[]   = {  0,  0, 150, 150 };

d3dDevice->CopyRect( pSource, rcSource, 2, pDest, ptDest);
```

The following methods are also available in C++/C for copying images to a Microsoft® Direct3D® surface.

- **D3DXLoadSurfaceFromFile**
- **D3DXLoadSurfaceFromFileInMemory**
- **D3DXLoadSurfaceFromMemory**
- **D3DXLoadSurfaceFromResource**
- **D3DXLoadSurfaceFromSurface**
- **IDirect3DDevice8::CopyRects**

Microsoft DirectX 8.1 (C++)

Copying Surfaces

[This is preliminary documentation and is subject to change.]

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

Note DirectX 8.0 provides Microsoft® Direct3DX functions that enable you to load artwork from files, apply color conversion, and resize artwork. For more information on the available functions see Texturing Functions.

Microsoft DirectX 8.1 (C++)

Accessing Surface Memory Directly

[This is preliminary documentation and is subject to change.]

You can directly access the surface memory by using the **IDirect3DSurface8::LockRect** method. When you call this method, the *pRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface to access directly. To request that the entire surface be locked, set *pRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface. Note that a multisample back buffer cannot be locked.

The **LockRect** method fills a **D3DLOCKED_RECT** structure with all the information to properly access the surface memory. The structure includes information about the pitch and has a pointer to the locked bits. When you finish accessing the surface memory, call the **IDirect3DSurface8::UnlockRect** method to unlock it.

While you have a surface locked, you can directly manipulate the contents. The following list

describes some tips for avoiding common problems with directly rendering surface memory.

- Never assume a constant display pitch. Always examine the pitch information returned by the **LockRect** method. This pitch can vary for a number of reasons, including the location of the surface memory, the display card type, or even the version of the Microsoft® Direct3D® driver. For more information, see Width vs. Pitch.
- Make certain you copy to unlocked surfaces. Direct3D copy methods will fail if called on a locked surface.
- Limit your application's activity while a surface is locked.
- Always copy data aligned to display memory. Microsoft® Windows® 95 and Windows 98 use a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to Direct3D. Copying data unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.
- A surface may not be locked if it belongs to a resource assigned to the D3DPOOL_DEFAULT memory pool. Back buffer surfaces, which may be accessed using the **IDirect3DDevice8::GetBackBuffer** and **IDirect3DSwapChain8::GetBackBuffer** methods, may be locked only if the swap chain was created with the **Flags** member of the **D3DPRESENT_PARAMETERS** structure set to include D3DPRESENTFLAG_LOCKABLE_BACKBUFFER.

Microsoft DirectX 8.1 (C++)

Private Surface Data

[This is preliminary documentation and is subject to change.]

You can store any kind of application-specific data with a surface. For example, a surface representing a map in a game might contain information about terrain.

A surface can have more than one private data buffer. Each buffer is identified by a GUID that you supply when attaching the data to the surface.

To store private surface data, use the **IDirect3DSurface8::SetPrivateData** method, passing a pointer to the source buffer, the size of the data, and an application-defined GUID for the data. Optionally, the source data can exist in the form of a COM object; in this case, you pass a pointer to the object's **IUnknown** interface pointer and you set the D3DSPD_IUNKNOWNPOINTER flag.

SetPrivateData allocates an internal buffer for the data and copies it. You can then safely free the source buffer or object. The internal buffer or interface reference is released when **IDirect3DSurface8::FreePrivateData** is called. This happens automatically when the surface is freed.

To retrieve private data for a surface, you must allocate a buffer of the correct size and then call the **IDirect3DSurface8::GetPrivateData** method, passing the GUID that was assigned to the data by **SetPrivateData**. You are responsible for freeing any dynamic memory you use for this buffer. If the data is a COM object, this method retrieves the **IUnknown** pointer.

If you don't know how big a buffer to allocate, first call **GetPrivateData** with zero in *SizeOfData*. If the method fails with D3DERR_MOREDATA, it returns the necessary number of bytes for the buffer in *SizeOfData*.

Microsoft DirectX 8.1 (C++)

Gamma Controls

[This is preliminary documentation and is subject to change.]

Gamma controls allow you to change how the system displays the contents of the surface, without affecting the contents of the surface itself. Think of these controls as very simple filters that Microsoft® Direct3D® applies to data as it leaves a surface and before it is rendered on the screen.

Gamma controls are simply a property of a swap chain. Gamma controls make it possible to dynamically change how a surface's red, green, and blue levels map to the actual levels that the system displays. By setting gamma levels, you can cause the user's screen to flash colors—red when the user's character is shot, green when the character picks up a new item, and so on—without copying new images to the frame buffer to achieve the effect. Or, you might adjust color levels to apply a color bias to the images in the back buffer.

There is always at least one swap chain (the implicit swap chain) for each device because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device. Because the gamma ramp is a property of the swap chain, the gamma ramp can be applied when the swap chain is windowed. The gamma ramp takes effect immediately. There is no waiting for a VSYNC operation.

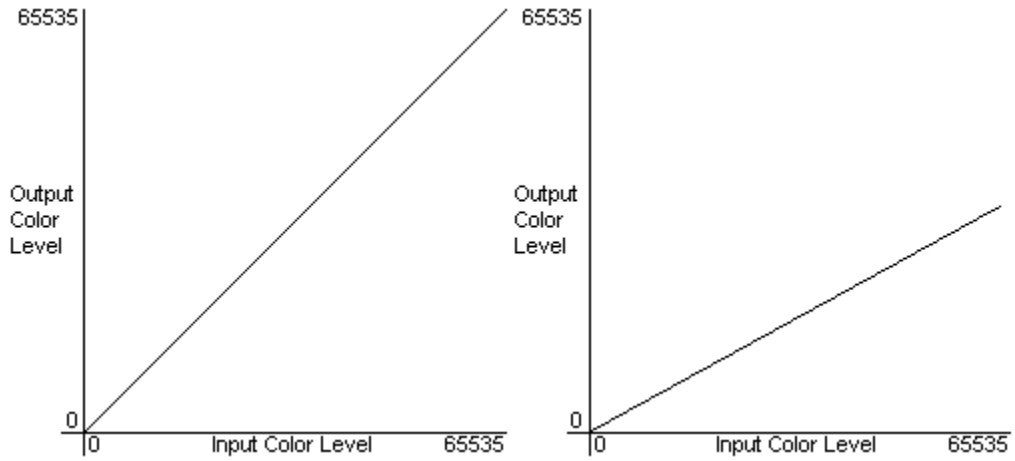
The **IDirect3DDevice8::SetGammaRamp** and **IDirect3DDevice8::GetGammaRamp** methods allow you to manipulate ramp levels that affect the red, green, and blue color components of pixels from the surface before they are sent to the digital-to-analog converter (DAC) for display.

Gamma Ramp Levels

In Microsoft® Direct3D®, the term *gamma ramp* describes a set of values that map the level of a particular color component—red, green, blue—for all pixels in the frame buffer to new levels that are received by the digital-to-analog converter (DAC) for display. The remapping is performed by way of three look-up tables, one for each color component.

Here's how it works: Direct3D takes a pixel from the frame buffer and evaluates its individual red, green, and blue color components. Each component is represented by a value from 0 to 65535. Direct3D takes the original value and uses it to index a 256-element array (the ramp), where each element contains a value that replaces the original one. Direct3D performs this look-up and replace process for each color component of each pixel in the frame buffer, thereby changing the final colors for all the on-screen pixels.

It's handy to visualize the ramp values by graphing them. The left graph of the two following graphs shows a ramp that doesn't modify colors at all. The right graph shows a ramp that imposes a negative bias to the color component to which it is applied.



The array elements for the graph on the left contain values identical to their index—0 in the element at index 0, and 65535 at index 255. This type of ramp is the default, as it doesn't change the input values before they're displayed. The right graph provides more variation; its ramp contains values that range from 0 in the first element to 32768 in the last element, with values ranging uniformly in between. The effect is that the color component that uses this ramp appears muted on the display. You are not limited to using linear graphs; if your application can assign arbitrary mapping if needed. You can even set the entries to all zeroes to remove a color component completely from the display.

Setting and Retrieving Gamma Ramp Levels

Gamma ramp levels are effectively look-up tables that Microsoft® Direct3D® uses to map the frame buffer color components to new levels that will be displayed. You can set and retrieve ramp levels for the primary surface by calling the **IDirect3DDevice8::SetGammaRamp** and **IDirect3DDevice8::GetGammaRamp** methods. **SetGammaRamp** accepts two parameters and **GetGammaRamp** accepts one parameter. For **SetGammaRamp**, the first parameter is either D3DSGR_CALIBRATE or D3DSGR_NO_CALIBRATION. The second parameter, *pRamp*, is a pointer to a **D3DGAMMARAMP** structure. The **D3DGAMMARAMP** structure contains three 256-element arrays of **WORD**s, one array each to contain the red, green, and blue gamma ramps. **GetGammaRamp** has one parameter that takes a pointer to a **D3DGAMMARAMP** type that will be filled with the current gamma ramp.

You can include the DDSGR_CALIBRATE value for the first parameter of **SetGammaRamp** to invoke the calibrator when setting new gamma levels. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently. Setting a calibrated gamma ramp provides a consistent and absolute gamma value for the user, regardless of the display adapter and monitor.

Not all systems support gamma calibration. To determine if gamma calibration is supported, call **IDirect3DDevice8::GetDeviceCaps**, and examine the **Caps2** member of the associated **D3DCAPS8** structure after the method returns. If the D3DCAPS2_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.

When setting new ramp levels, keep in mind that that the levels you set in the arrays are only used when your application is in full-screen, exclusive mode. Whenever your application changes to normal mode, the ramp levels are set aside, taking effect again when the application reinstates full-screen mode.

If the device does not support gamma ramps in the swap chain's current presentation mode (full-screen or windowed), no error value is returned. Applications can check the D3DCAPS2_FULLSCREENGAMMA and D3DCAPS2_CANCALIBRATEGAMMA capability

bits in the Caps2 member of the D3DCAPS8 type to determine the capabilities of the device and whether a calibrator is installed.

Microsoft DirectX 8.1 (C++)

Rendering

[This is preliminary documentation and is subject to change.]

Applications use the DrawPrimitive family of methods to render a 3-D scene. The following topics discuss rendering with DrawPrimitive.

- Shading
- Clearing Surfaces
- Beginning and Ending a Scene
- Presenting a Scene
- Rendering Primitives
- Fog, Alpha Blending, and Depth Buffers

Microsoft DirectX 8.1 (C++)

Shading

[This is preliminary documentation and is subject to change.]

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

- Shading Modes
- Comparing Shading Modes
- Setting the Shading Mode

Microsoft DirectX 8.1 (C++)

Shading Modes

[This is preliminary documentation and is 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 face. Microsoft® Direct3D® supports two shading modes:

Flat Shading

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

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

Gouraud Shading

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

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

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

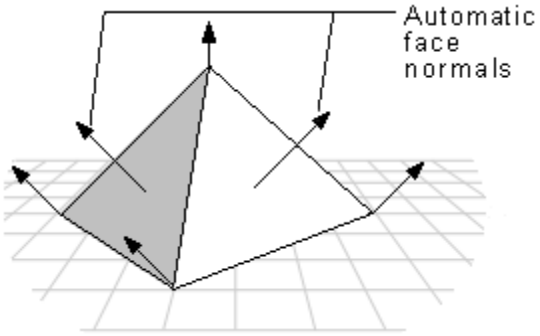
For more information, see Face and Vertex Normal Vectors.

Microsoft DirectX 8.1 (C++)

Comparing Shading Modes

[This is preliminary documentation and is subject to change.]

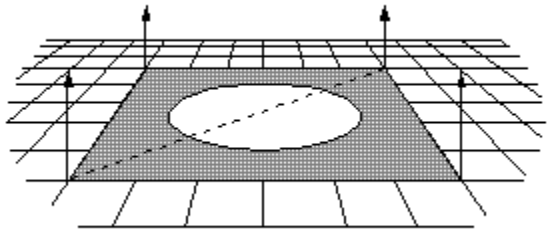
In flat shading mode, the pyramid below is displayed with a sharp edge between adjoining faces. In Gouraud shading mode, however, shading values are interpolated across the edge, and the final appearance is of a curved surface.



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

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

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



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

Microsoft DirectX 8.1 (C++)

Setting the Shading Mode

[This is preliminary documentation and is subject to change.]

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

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

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

Microsoft DirectX 8.1 (C++)

Clearing Surfaces

[This is preliminary documentation and is subject to change.]

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

Use the **IDirect3DDevice8::Clear** method to clear the viewport.

For more information about using this method, see Clearing a Viewport.

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

Microsoft DirectX 8.1 (C++)

Beginning and Ending a Scene

[This is preliminary documentation and is subject to change.]

Applications written in C++ notify Microsoft® Direct3D® that scene rendering is about to begin by calling the **IDirect3DDevice8::BeginScene** method. **BeginScene** causes the system to check its internal data structures and the availability and validity of rendering surfaces. It also sets an internal flag to signal that a scene is in progress. After you begin a scene, you can call the various rendering methods to render the primitives or individual vertices that make up the objects in the scene. Attempts to call rendering methods when a scene is not in progress fail. For more information, see Rendering Primitives.

After you complete the scene, call the **IDirect3DDevice8::EndScene** method. The **EndScene** method flushes cached data, verifies the integrity of rendering surfaces, and clears an internal flag that signals when a scene is in progress.

All rendering methods must be bracketed by calls to the **BeginScene** and **EndScene** methods. If surfaces are lost or internal errors occur, the scene methods return error values. If **BeginScene** fails, the scene does not begin, and subsequent calls to **EndScene** will fail.

To summarize these cases:

- If **BeginScene** returns any error, you must not call **EndScene** because the scene has not successfully begun.
- If **BeginScene** succeeds, but you get an error while rendering the scene, you must call **EndScene**.
- If **EndScene** fails, for any reason, you need not call it again.

For example, some simple scene code might look like the following example.

```
HRESULT hr;

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

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

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

Do not attempt to call GDI functions on Direct3D surfaces, such as the render target or textures, while a scene is being rendered—is between **BeginScene** and **EndScene** calls. Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, be sure that all GDI calls are made outside the scene functions.

Microsoft DirectX 8.1 (C++)

Presenting a Scene

[This is preliminary documentation and is subject to change.]

This section introduces the presentation application programming interfaces (APIs) and discusses the issues involved in presenting a scene to the display.

Information is divided into the following topics.

- Introduction to Presenting a Scene
- Multiple Views in Windowed Mode
- Multiple-Monitor Operations
- Manipulating the Depth Buffer
- Accessing the Color Front Buffer

Microsoft DirectX 8.1 (C++)

Introduction to Presenting a Scene

[This is preliminary documentation and is subject to change.]

The presentation application programming interfaces (APIs) are a set of methods that control the state of the device that affects what the user sees on the monitor. These methods include setting display modes and once-per-frame methods that are used to present images to the user.

- **IDirect3DDevice8::Present**

- **IDirect3DDevice8::Reset**
- **IDirect3DDevice8::GetGammaRamp**
- **IDirect3DDevice8::SetGammaRamp**
- **IDirect3DDevice8::GetRasterStatus**

Familiarity with the following terms is necessary to understand the presentation APIs.

- *Front Buffer*. A rectangle of memory that is translated by the graphics adapter and displayed on the monitor or other output device.
- *Back Buffer*. A surface whose contents can be promoted to the front buffer.
- *Swap Chain*. A collection of back buffers that can be serially presented to the front buffer. Typically, a full-screen swap chain presents subsequent images with the flipping DDI, and a windowed swap chain presents images with the blitting DDI.

Because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device, there is always at least one swap chain per device. The **IDirect3DDevice8** interface has a set of methods that manipulate the implicit swap chain and are a copy of the swap chain's own interface. Applications can create additional swap chains; however, this is not necessary for the typical single window or full-screen application.

The front buffer is not directly exposed in the Direct3D API for DirectX 8.0. As a result, applications cannot lock or render to the front buffer. For details, see Accessing the Color Front Buffer.

Note DirectX 7.*x* provided a number of presentation APIs that were called together. A good example of this is the **IDirectDraw7::SetCooperativeLevel**, **IDirectDraw7::SetDisplayMode**, and **IDirectDraw7::CreateSurface** sequence. Additionally, the **IDirectDrawSurface7::Flip** and **IDirectDrawSurface7::Blt** methods signaled the transport of rendered frames to the monitor. DirectX 8.0 collapses these groups of APIs into two main methods, **Reset** and **Present**. **Reset** subsumes **SetCooperativeLevel**, **SetDisplayMode**, **CreateSurface**, and some of the parameters to **Flip**. **Present** subsumes **Flip** and the presentation uses of **Blt**.

A call to **IDirect3D8::CreateDevice** represents an implicit reset of the device. The DirectX 8.0 API has no notion of a primary surface; you cannot create an object that represents the primary surface. It is considered to be an internal property of the device.

Gamma ramps are associated with a swap chain and are manipulated with the **IDirect3DDevice8::GetGammaRamp** and **IDirect3DDevice8::SetGammaRamp** methods.

Microsoft DirectX 8.1 (C++)

Multiple Views in Windowed Mode

[This is preliminary documentation and is subject to change.]

In addition to the swap chain that is owned and manipulated through the Direct3DDevice object, an application can use the **IDirect3DDevice8::CreateAdditionalSwapChain** method to create additional swap chains to present multiple views from the same device.

Typically, the application creates one swap chain per view, and it associates each swap chain with a particular view. The application renders images in the back buffers of each swap chain, and then uses the **IDirect3DDevice8::Present** method to present them individually. Note that only one swap chain at a time can be full-screen on each adapter.

Microsoft DirectX 8.1 (C++)

Multiple-Monitor Operations

[This is preliminary documentation and is subject to change.]

When a device is successfully reset (**IDirect3DDevice8::Reset**) or created (**IDirect3D8::CreateDevice**) in full-screen operations, the Microsoft® Direct3D® object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D object can neither assume full-screen operations nor allocate video memory. In addition, when a Direct3D object assumes exclusive mode, all devices other than the one that went full-screen are placed in lost state. For details, see Lost Devices.

Along with exclusive mode, the Direct3D object is informed of the focus window that the device will use. Exclusive mode is released when the final full-screen device owned by that Direct3D object is either reset to windowed mode or destroyed.

Devices can be divided into two categories when a Direct3D object owns exclusive mode. The first category of devices have the following characteristics.

- They are created by the same Direct3D object that created the device that is full-screen.
- They have the same focus window as the device that is full-screen.
- They represent a different adapter from any full-screen device.

Devices in this category have no restrictions concerning their ability to be reset or created, and they are not placed in lost state. Devices in this category can even be put into full-screen mode.

Devices that do not fall in the first category—devices created by another Direct3D object, created with a different focus window, and created for an adapter with a device that is already full-screen—cannot be reset and remain in lost state until the exclusive mode is lost. As a result, a multiple-monitor application can place several devices in full-screen mode, but only if all these devices are for different adapters, were created by the same Direct3D object, and share the same focus window.

Microsoft DirectX 8.1 (C++)

Manipulating the Depth Buffer

[This is preliminary documentation and is subject to change.]

Depth buffers are associated with the device. Applications are required to move the depth buffers when they set render targets. The **IDirect3DDevice8::GetDepthStencilSurface** and **IDirect3DDevice8::SetRenderTarget** methods are used to manipulate depth buffers.

Microsoft DirectX 8.1 (C++)

Accessing the Color Front Buffer

[This is preliminary documentation and is subject to change.]

Accessing the front buffer is allowed through the **IDirect3DDevice8::GetFrontBuffer** method. This method is the only way to get a screen shot of an anti-aliased scene.

Microsoft DirectX 8.1 (C++)

Rendering Primitives

[This is preliminary documentation and is subject to change.]

The following topics introduce the **DrawPrimitive** rendering methods and provide information about using them in your application.

- Vertex Data Streams
- Setting the Stream Source
- Rendering from Vertex and Index Buffers
- Rendering from User Memory Pointers

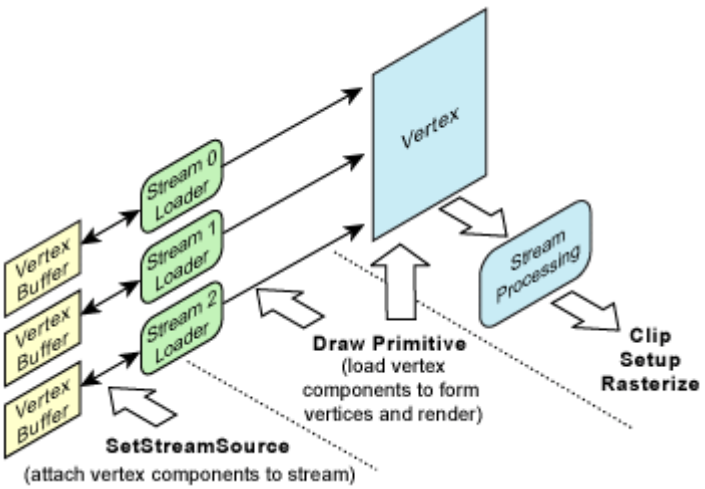
Microsoft DirectX 8.1 (C++)

Vertex Data Streams

[This is preliminary documentation and is subject to change.]

The rendering interfaces for Microsoft® Direct3D® consist of methods that render primitives from vertex data stored in one or more data buffers. Vertex data consists of vertex elements combined to form vertex components. Vertex elements, the smallest unit of a vertex, represent entities such as position, normal, or color.

Vertex components are one or more vertex elements stored contiguously (interleaved per vertex) in a single memory buffer. A complete vertex consists of one or more components, where each component is in a separate memory buffer. To render a primitive, multiple vertex components are read and assembled so that complete vertices are available for vertex processing. The illustration below shows the process of rendering primitives using vertex components.



Rendering primitives consists of two steps. First, set up one or more vertex component streams; second, invoke a **DrawPrimitive** method to render from those streams. Identification of vertex elements within these component streams is specified by the vertex shader. For details, see Vertex

Shaders.

The **DrawPrimitive** methods specify an offset in the vertex data streams so that an arbitrary contiguous subset of the primitives within one set of vertex data can be rendered with each draw invocation. This enables you to change the device rendering state between groups of primitives that are rendered from the same vertex buffers.

Both indexed and nonindexed drawing methods are supported. For more information, see Rendering from Vertex and Index Buffers.

Microsoft DirectX 8.1 (C++)

Setting the Stream Source

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8::SetStreamSource** method binds a vertex buffer to a device data stream, creating an association between the vertex data and one of several data stream ports that feed the primitive processing functions. The actual references to the stream data do not occur until a drawing method, such as **IDirect3DDevice8::DrawPrimitive**, is called.

A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on. The *Stride* parameter specifies the size of the component, in bytes.

Microsoft DirectX 8.1 (C++)

Rendering from Vertex and Index Buffers

[This is preliminary documentation and is subject to change.]

Both indexed and nonindexed drawing methods are supported by Microsoft® Direct3D®. The indexed methods use a single set of indices for all vertex components. Vertex data is presented to the Direct3D application programming interface (API) in vertex buffers, and index data for the indexed drawing methods is presented in index buffers. For more information, see the following reference topics.

- **IDirect3DDevice8::DrawPrimitive**
- **IDirect3DDevice8::DrawIndexedPrimitive**

These methods draw primitives from the current set of data input streams. For details on setting the current index array to an index buffer, see the **IDirect3DDevice8::SetIndices** method.

Microsoft DirectX 8.1 (C++)

Rendering from User Memory Pointers

[This is preliminary documentation and is subject to change.]

A secondary set of rendering interfaces supports passing vertex and index data directly from user

memory pointers. These interfaces support a single stream of vertex data only. For more information, see the following reference topics.

- **IDirect3DDevice8::DrawPrimitiveUP**
- **IDirect3DDevice8::DrawIndexedPrimitiveUP**

These methods render with data specified by user memory pointers, instead of vertex and index buffers.

Microsoft DirectX 8.1 (C++)

Fog, Alpha Blending, and Depth Buffers

[This is preliminary documentation and is subject to change.]

After the texture maps and/or the pixel shaders have calculated per pixel colors, some effects can be applied to an entire frame to produce the final pixel color value. These effects are illustrated in the topics below.

- Fog
- Alpha Blending
- Depth Buffers

Microsoft DirectX 8.1 (C++)

Fog

[This is preliminary documentation and is subject to change.]

Adding fog to a 3-D scene can enhance realism, provide ambiance or set a mood, and obscure artifacts sometimes caused when distant geometry comes into view. Microsoft® Direct3D® supports two fog models—pixel fog and vertex fog—each with its own features and programming interface.

Essentially, fog is implemented by blending the color of objects in a scene with a chosen fog color based on the depth of an object in a scene or its distance from the viewpoint. As objects grow more distant, their original color blends more and more with the chosen fog color, creating the illusion that the object is being increasingly obscured by tiny particles floating in the scene. The following illustration shows a scene rendered without fog, and a similar scene rendered with fog enabled.

In this illustration, the scene on the left has a clear horizon, beyond which no scenery is visible, even though it would be visible in the real world. The scene on the right obscures the horizon by using a fog color identical to the background color, making polygons appear to fade into the distance. By combining discrete fog effects with creative scene design you can add mood and soften the color of objects in a scene.

Direct3D provides two ways to add fog to a scene, pixel fog and vertex fog, named for how the fog effects are applied. For details, see Pixel Fog and Vertex Fog. In short, pixel fog—also called table fog—is implemented in the device driver, and vertex fog is implemented in the Direct3D lighting engine.

Note Regardless of which type of fog—pixel or vertex—you use, your application must provide a compliant projection matrix to ensure that fog effects are properly applied. This restriction applies even to applications that do not use the Direct3D transformation and lighting engine. For additional details about how you can provide an appropriate matrix, see A W-Friendly Projection Matrix.

The following topics introduce fog and present information about using various fog features in Microsoft® Direct3D® applications.

- Fog Formulas
- Fog Parameters
- Fog Blending
- Fog Color
- Vertex Fog
- Pixel Fog

Fog blending is controlled by render states; it is not part of the programmable pixel pipeline.

Microsoft DirectX 8.1 (C++)

Fog Formulas

[This is preliminary documentation and is subject to change.]

C++ applications can control how fog affects the color of objects in a scene by changing how Microsoft® Direct3D® computes fog effects over distance. The **D3DFOGMODE** enumerated type contains members that identify the three fog formulas. All formulas calculate a fog factor as a function of distance, given parameters that your application sets.

D3DFOG_LINEAR

$$f = \frac{end - d}{end - start}$$

where

- *start* is the distance at which fog effects begin.
- *end* is the distance at which fog effects no longer increase.
- *d* represents depth, or the distance from the viewpoint. For range based fog, the value for *d* is the distance between the camera position and a vertex. For non-range based fog, the value for *d* is the absolute value of the Z coordinate in camera space.

Linear and exponential formulas are supported for both pixel fog and vertex fog.

D3DFOG_EXP

$$f = \frac{1}{e^{d \times density}}$$

where

- *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.
- *d* represents depth, or the distance from the viewpoint, as stated earlier.

D3DFOG_EXP2

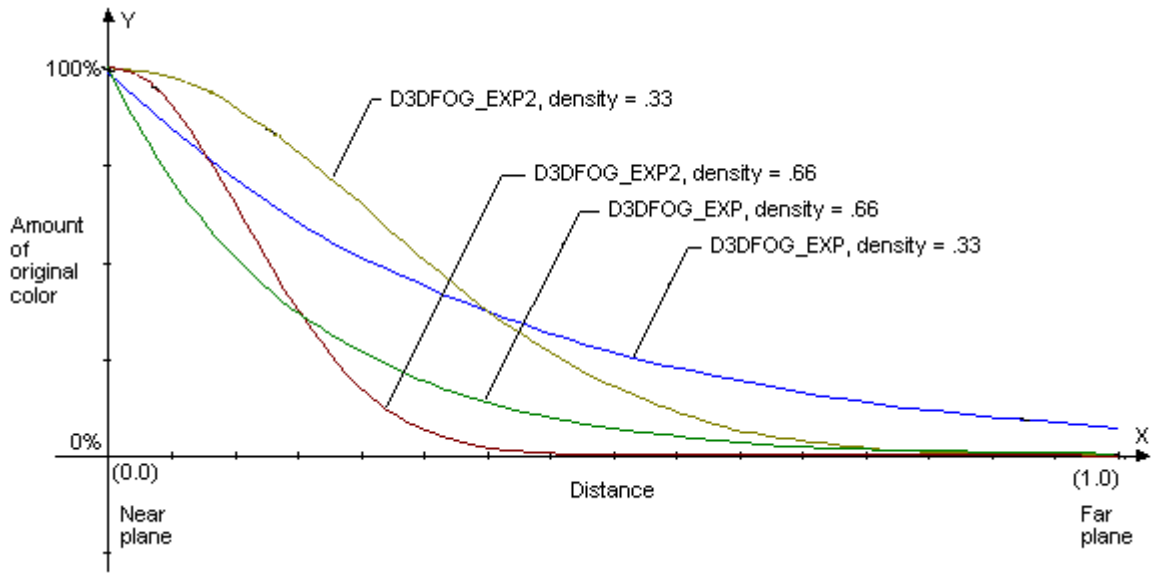
$$f = \frac{1}{e^{(d \times density)^2}}$$

where

- *e* is the base of natural logarithms as stated above.
- *density* is an arbitrary fog density that can range from 0.0 to 1.0 as stated above.
- *d* represents depth, or the distance from the viewpoint, as stated above.

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.



D3DFOG_LINEAR is 1.0 at *start* and 0.0 at *end*. It is not measured relative to the near or far planes.

When Direct3D calculates fog effects, it uses the fog factor from one of the preceding equations in a blending formula, shown here.

$$C = f \cdot C_i + (1 - f) \cdot C_f$$

This formula effectively scales the color of the current polygon C_i by the fog factor *f*, and adds the product to the fog color C_f, scaled by the bitwise inverse of the fog factor. The resulting color value is a blend of the fog color and the original color, as a factor of distance. The formula applies to all devices supported in Microsoft DirectX® 7.0 and later. For the legacy ramp device, the fog factor scales the diffuse and specular color components, clamped to the range of 0.0 and 1.0, inclusive. The fog factor typically starts at 1.0 for the near plane and decreases to 0.0 at the far plane.

Microsoft DirectX 8.1 (C++)

Fog Parameters

[This is preliminary documentation and is subject to change.]

Fog parameters are controlled through device render states. Both pixel and vertex fog types support all the fog formulas introduced in Fog Formulas. The **D3DFOGMODE** enumerated type defines constants that you can use to identify the fog formula you want Microsoft® Direct3D® to use. The D3DRS_FOGTABLEMODE render state controls the fog mode that Direct3D uses for pixel fog, and the D3DRS_FOGVERTEXMODE render state controls the mode for vertex fog.

When using the linear fog formula, you set the starting and ending distances through the D3DRS_FOGSTART and D3DRS_FOGEND render states. How the system interprets these values depends on the type of fog your application uses—pixel or vertex fog—and, when using pixel fog, if z-based or w-based depth is being used. The following table summarizes fog types and their start and end units.

Fog type	Fog start/end units
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

The D3DRS_FOGDENSITY render state controls the fog density applied when an exponential fog formula is enabled. Fog density is essentially a weighting factor, ranging from 0.0 to 1.0 (inclusive), that scales the distance value in the exponent.

The color that the system uses for fog blending is controlled through the D3DRS_FOGCOLOR device render state. For more information, see Fog Color and Fog Blending.

Microsoft DirectX 8.1 (C++)

Fog Blending

[This is preliminary documentation and is 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 Fog Formulas. The D3DRS_FOGENABLE render state controls fog blending. Set this render state to TRUE to enable fog blending as shown in the following example code. The default is FALSE.

```
//
// For this example, g_pDevice is a valid pointer
// to an IDirect3DDevice8 interface.
HRESULT hr;
hr = g_pDevice->SetRenderState(
                    D3DRS_FOGENABLE,
                    TRUE);
if FAILED(hr)
    return hr;
```

You must enable fog blending for both pixel fog and vertex fog. For information about using these types of fog, see Pixel Fog and Vertex Fog.

Microsoft DirectX 8.1 (C++)

Fog Color

[This is preliminary documentation and is subject to change.]

Fog color for both pixel and vertex fog is set through the D3DRS_FOGCOLOR render state. The render state values can be any RGB color, specified as an RGBA color; the alpha component is ignored.

The following C++ example sets the fog color to white.

```
/* For this example, the d3dDevice variable is
 * a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;

hr = d3dDevice->SetRenderState(
                    D3DRS_FOGCOLOR,
                    0x00FFFFFF); // Highest 8 bits are not used.

if(FAILED(hr))
    return hr;
```

Microsoft DirectX 8.1 (C++)

Vertex Fog

[This is preliminary documentation and is subject to change.]

When the system performs vertex fogging, it applies fog calculations at each vertex in a polygon, and then interpolates the results across the face of the polygon during rasterization. Vertex fog effects are computed by the Microsoft® Direct3D® lighting and transformation engine. For more information, see Fog Parameters.

If your application does not use Direct3D for transformation and lighting, it must perform fog calculations on its own. In this case, your application can place the fog factor that it computes in the alpha component of the specular color for each vertex. You are free to use whatever formulas you want—range-based, volumetric, or otherwise. Direct3D uses the supplied fog factor to interpolate across the face of each polygon. Applications that do not use Direct3D transformation and lighting need not set vertex fog parameters, but must still enable fog and set the fog color through the associated render states. For more information, see Fog Parameters.

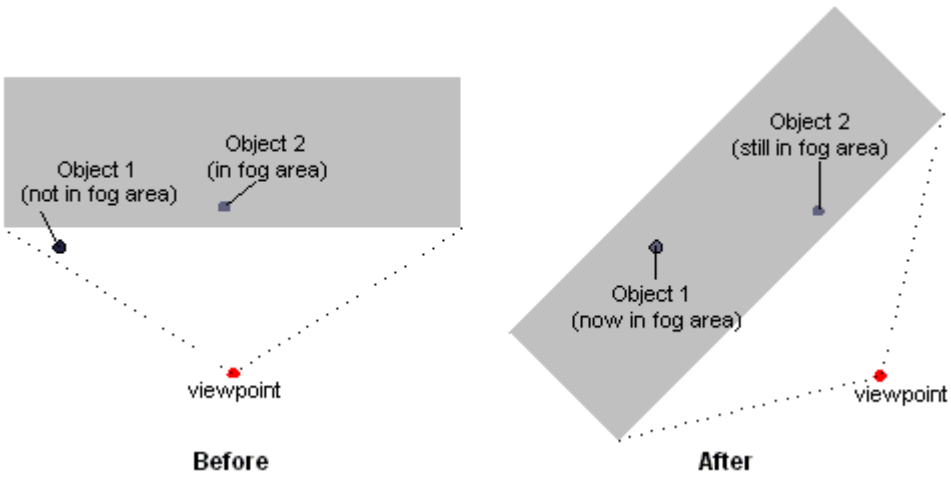
Note When using a vertex shader, you must use vertex fog. The vertex shader must write the per-vertex fog intensity to oFog. This is then used after the pixel shader to linearly interpolate with the fog color. This intensity is not available in a pixel shader.

Applications that perform their own transformation and lighting must also perform their own vertex fog calculations. As a result, such an application need only enable fog blending and set the fog color through the associated render states, as described in Fog Blending and Fog Color.

Range-Based Fog

Note Direct3D uses range-based fog calculations only when using vertex fog with the Direct3D transformation and lighting engine. This is because pixel fog is implemented in the device driver, and no hardware currently exists to support per-pixel range-based fog. If your application performs its own transformation and lighting, it must perform its own fog calculations, range-based or otherwise.

Sometimes, using fog can introduce graphic artifacts that cause objects to be blended with the fog color in nonintuitive ways. For example, imagine a scene in which there are two visible objects: one distant enough to be affected by fog, and the other near enough to be unaffected. If the viewing area rotates in place, the apparent fog effects can change, even if the objects are stationary. The following illustration shows a top-down view of such a situation.



Range-based fog is another, more accurate, way to determine the fog effects. In range-based fog, Direct3D uses the actual distance from the viewpoint to a vertex for its fog calculations. Direct3D increases the effect of fog as the distance between the two points increases, rather than the depth of the vertex within in the scene, thereby avoiding rotational artifacts.

If the current device supports range-based fog, it will set the D3DPRASTERCAPS_FOGRANGE capability flag in the **RasterCaps** member of the **D3DCAPS8** structure when you call the **IDirect3DDevice8::GetDeviceCaps** method. To enable range-based fog, set the D3DRS_RANGEFOGENABLE render state to TRUE.

Range-based fog is computed by Direct3D during transformation and lighting. Applications that don't use the Direct3D transformation and lighting engine must also perform their own vertex fog calculations. In this case, provide the range-based fog factor in the alpha component of the specular component for each vertex.

Using Vertex Fog

Use the following steps to enable vertex fog in your application.

1. Enable fog blending by setting D3DRS_FOGENABLE to TRUE.
2. Set the fog color in the D3DRS_FOGCOLOR render state.
3. Choose the desired fog formula by setting the D3DRS_FOGVERTEXMODE render state to a member of the **D3DFOGMODE** enumerated type.
4. Set the fog parameters as desired for the selected fog formula in the render states.

The following example, written in C++, shows what these steps might look like in code.

```
// For brevity, error values in this example are not checked
```

```
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_pDevice is a valid
// pointer to an IDirect3DDevice8 interface.
void SetupVertexFog(DWORD Color, DWORD Mode, BOOL UseRange, FLOAT Density)
{
    float Start = 0.5f,    // Linear fog distances
          End   = 0.8f;

    // Enable fog blending.
    g_pDevice->SetRenderState(D3DRS_FOGENABLE, TRUE);

    // Set the fog color.
    g_pDevice->SetRenderState(D3DRS_FOGCOLOR, Color);

    // Set fog parameters.
    if(D3DFOG_LINEAR == Mode)
    {
        g_pDevice->SetRenderState(D3DRS_FOGVERTEXMODE, Mode);
        g_pDevice->SetRenderState(D3DRS_FOGSTART, *(DWORD *)(&Start));
        g_pDevice->SetRenderState(D3DRS_FOGEND,   *(DWORD *)(&End));
    }
    else
    {
        g_pDevice->SetRenderState(D3DRS_FOGVERTEXMODE, Mode);
        g_pDevice->SetRenderState(D3DRS_FOGDENSITY, *(DWORD *)(&Density));
    }

    // Enable range-based fog if desired (only supported for
    // vertex fog). For this example, it is assumed that UseRange
    // is set to a nonzero value only if the driver exposes the
    // D3DPRASTERCAPS_FOGRANGE capability.
    // Note: This is slightly more performance intensive
    //       than non-range-based fog.
    if(UseRange)
        g_pDevice->SetRenderState(
                        D3DRS_RANGEFOGENABLE,
                        TRUE);
}
```

Some fog parameters are required as floating-point values, even though the**IDirect3DDevice8::SetRenderState** method only accepts **DWORD** values in the second parameter. This example successfully provides the floating-point values to these methods without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, and then dereferencing them.

Microsoft DirectX 8.1 (C++)

Pixel Fog

[This is preliminary documentation and is subject to change.]

Pixel fog gets its name from the fact that it is calculated on a per-pixel basis in the device driver, unlike vertex fog, in which Microsoft® Direct3D® computes fog effects when it performs transformation and lighting. Pixel fog is sometimes called table fog because some drivers use a precalculated look-up table to determine the fog factor, using the depth of each pixel, to apply in blending computations.

Pixel fog can be applied using any fog formula identified by members of the **D3DFOGMODE** enumerated type. Pixel-fog formula implementations are driver-specific, and if a driver doesn't support a complex fog formula, it should degrade to a less complex formula.

Note Pixel fog is not supported when using a vertex shader.

Eye-Relative vs. Z-Based Depth

To alleviate fog-related graphic artifacts caused by uneven distribution of z-values in a depth buffer, most hardware devices use eye-relative depth instead of z-based depth values for pixel fog. Eye-relative depth is essentially the w element from a homogeneous coordinate set. Microsoft® Direct3D® takes the reciprocal of the RHW element from a device space coordinate set to reproduce true w. If a device supports eye-relative fog, it sets the D3DPRASTERCAPS_WFOG flag in the **RasterCaps** member of the **D3DCAPS8** structure when you call the **IDirect3DDevice8::GetDeviceCaps** method.

Note With the exception of the reference rasterizer, software devices always use z to calculate pixel fog effects.

When eye-relative fog is supported, the system automatically uses eye-relative depth rather than z-based depth if the provided projection matrix produces z-values in world space that are equivalent to w-values in device space. You set the projection matrix by calling the **IDirect3DDevice8::SetTransform** method, using the D3DTS_PROJECTION value and passing a **D3DMATRIX** structure that represents the desired matrix. If the projection matrix isn't compliant with this requirement, fog effects are not applied properly. For details about producing a compliant matrix, see A W-Friendly Projection Matrix. The perspective projection matrix provided in What Is the Projection Transformation? produces a compliant projection matrix.

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

Direct3D checks the fourth column of the projection matrix. If the coefficients are [0,0,0,1] (for an affine projection) the system will use z-based depth values for fog. In this case, you must also specify the start and end distances for linear fog effects in device space, which ranges from 0.0 at the nearest point to the user, and 1.0 at the farthest point.

Using Pixel Fog

Use the following steps to enable pixel fog in your application.

1. Enable fog blending by setting the D3DRS_FOGENABLE render state to TRUE.
2. Set the desired fog color in the D3DRS_FOGCOLOR render state.
3. Choose the fog formula to use by setting the D3DRS_FOGTABLEMODE render state to the corresponding member of the **D3DFOGMODE** enumerated type.
4. Set the fog parameters as desired for the selected fog mode in the associated render states. This includes the start and end distances for linear fog, and fog density for exponential fog mode.

The following example shows what these steps might look like in code.

```
// For brevity, error values in this example are not checked
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_pDevice is a valid
```

```
// pointer to an IDirect3DDevice8 interface.
void SetupPixelFog(DWORD Color, DWORD Mode)
{
    float Start = 0.5f,    // For linear mode
          End   = 0.8f,
          Density = 0.66;  // For exponential modes

    // Enable fog blending.
    g_pDevice->SetRenderState(D3DRS_FOGENABLE, TRUE);

    // Set the fog color.
    g_pDevice->SetRenderState(D3DRS_FOGCOLOR, Color);

    // Set fog parameters.
    if(D3DFOG_LINEAR == Mode)
    {
        g_pDevice->SetRenderState(D3DRS_FOGTABLEMODE, Mode);
        g_pDevice->SetRenderState(D3DRS_FOGSTART, *(DWORD *)(&Start));
        g_pDevice->SetRenderState(D3DRS_FOGEND,   *(DWORD *)(&End));
    }
    else
    {
        g_pDevice->SetRenderState(D3DRS_FOGTABLEMODE, Mode);
        g_pDevice->SetRenderState(D3DRS_FOGDENSITY, *(DWORD *)(&Density));
    }

}
```

Note Some fog parameters are required as floating-point values, even though the **IDirect3DDevice8::SetRenderState** method only accepts **DWORD** values in the second parameter. The preceding example provides the floating-point values to **SetRenderState** without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, and then dereferencing them.

Microsoft DirectX 8.1 (C++)

Alpha Blending

[This is preliminary documentation and is subject to change.]

Alpha blending is used to display an image that has transparent or semi-transparent pixels. In addition to a red, green, and blue color channel, each pixel in an alpha bitmap has a transparency component known as its *alpha channel*. The alpha channel typically contains as many bits as a color channel. For example, an 8-bit alpha channel can represent 256 levels of transparency, from 0 (the entire pixel is transparent) to 255 (the entire pixel is opaque). The list below shows some special effects you can create using alpha blending.

- Billboarding
- Clouds, Smoke, and Vapor Trails
- Fire, Flares, and Explosions

Microsoft DirectX 8.1 (C++)

Billboarding

[This is preliminary documentation and is subject to change.]

When creating 3-D scenes, an application can sometimes gain performance advantages by rendering 2-D objects in a way that makes them appear to be 3-D objects. This is the basic idea behind the technique of billboarding.

A billboard in the normal sense of the word is a sign along a roadway. Microsoft® Direct3D® applications can create and render this type of billboard by defining a rectangular solid and applying a texture to it. Billboarding in the more specialized sense of 3-D graphics is an extension of this. The goal is to make 2-D objects appear to be 3-D. The technique is to apply a texture that contains the object's image to a rectangular primitive. The primitive is rotated so that it always faces the user. It doesn't matter if the object's image is not rectangular. Portions of the billboard can be made transparent, so the parts of the billboard image that you don't want seen are not visible.

Many games employ billboarding for animated sprites. For instance, when the user is moving through a 3-D maze, he or she may see weapons or rewards that can be picked up. These are typically 2-D images textured on a rectangular primitive. Billboarding is often used in games to render images of trees, bushes, and clouds.

When an image is applied to a billboard, the rectangular primitive must first be rotated so that the resulting image faces the user. Your application must then translate it into position. The application can then apply a texture to the primitive.

Billboarding works best for symmetrical objects, especially objects that are symmetrical along the vertical axis. It also requires that the altitude of the viewpoint doesn't increase too much. If the user is allowed to view the billboarded from above, it becomes readily apparent that the object is 2-D rather than 3-D.

Microsoft DirectX 8.1 (C++)

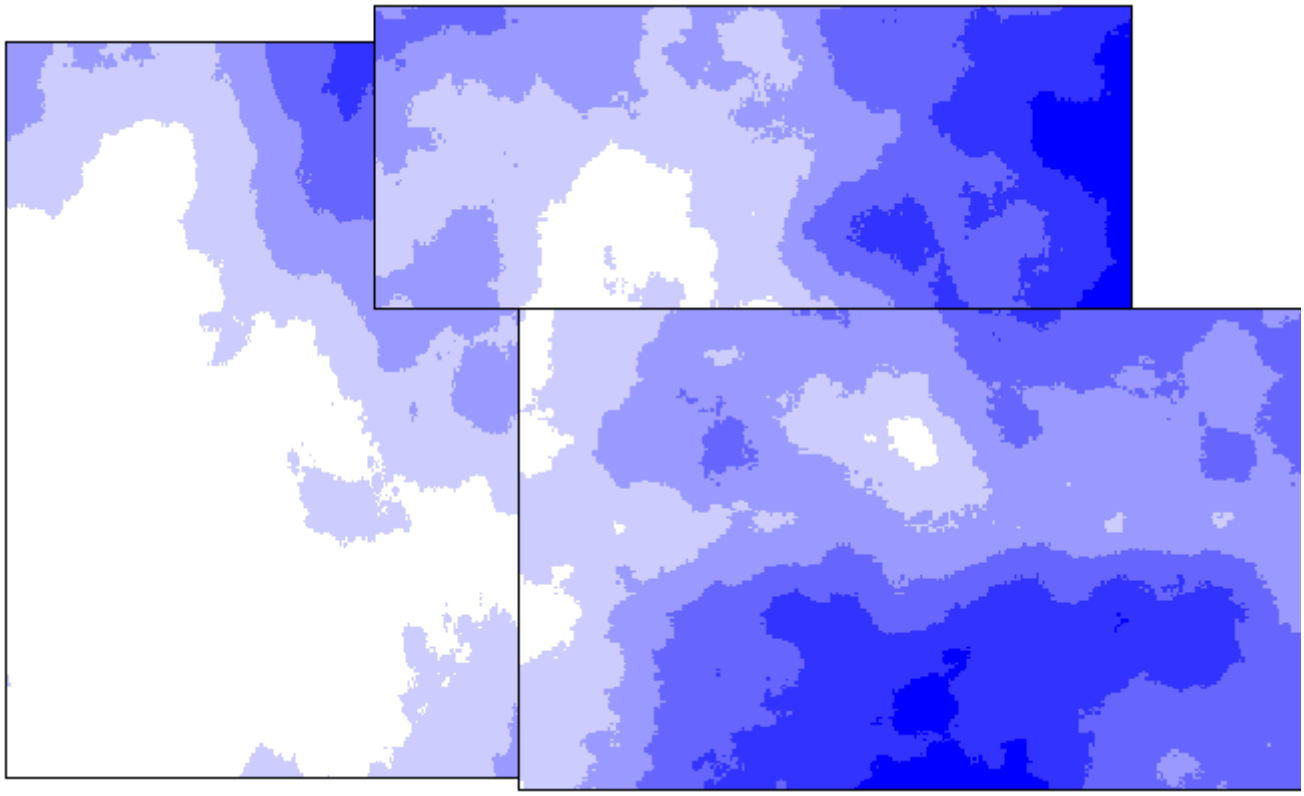
Clouds, Smoke, and Vapor Trails

[This is preliminary documentation and is 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 on two axes instead of one, your application can enable the user to look at a billboard from any angle. Typically, an application rotates the billboard on the horizontal and vertical axes.

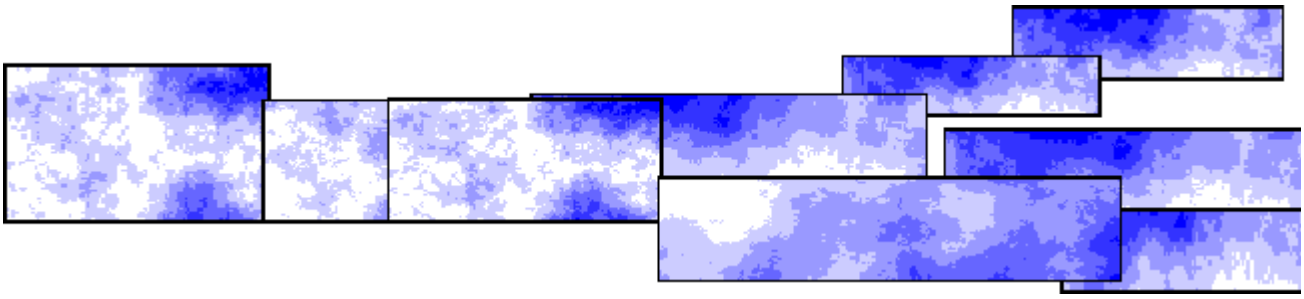
To make a simple cloud, your application can rotate a rectangular primitive on one or two axes so that the primitive faces the user. A cloud-like texture can then be applied to the primitive with transparency. For details on applying transparent textures to primitives, see Texture Blending. You can animate the cloud by applying a series of textures over time.

An application can create more complex clouds by forming them from a group of primitives. Each part of the cloud is a rectangular primitive. The primitives can be moved independently over time to give the appearance of a dynamic mist. This concept is illustrated in the following figure.



The appearance of smoke is displayed in a manner similar to clouds. It typically requires multiple billboards, like complex clouds. Smoke usually billows and rises over time, so the billboards that make up the smoke plume need to move accordingly. You may need to add more billboards as the plume rises and disperses.

A vapor trail is a smoke plume that doesn't rise. However, like a smoke plume, it disperses over time. The following figure illustrates the technique of using billboards to simulate a vapor trail.



Microsoft DirectX 8.1 (C++)

Fire, Flares, and Explosions

[This is preliminary documentation and is subject to change.]

You can use Microsoft® Direct3D® to simulate natural phenomena involving energy releases. For instance, an application can generate the appearance of fire by applying flame-like textures to a set of billboards. This is especially effective if the application uses a sequence of fire textures to animate the flames on each billboard in the fire. Varying the speed of the animation playback from billboard to billboard increases the appearance of real flames. The semblance of intermingled 3-D flames can be achieved by layering the billboards and the textures on the billboards.

You can simulate flares and flashes by applying successively brighter light maps to all primitives in a scene. Although this is a computationally high-overhead technique, it allows your application to simulate a localized flare or flash. That is, the portion of the scene where the flare or flash originates can brighten first.

Another technique is to position a billboard in front of the scene so that the entire render target area is covered. The application applies successively whiter textures to the billboard and decreases the transparency over time. The entire scene fades to white as time passes. This is a low overhead method of creating a flare. However, using this technique, it can be difficult to generate the appearance of a bright flash from a single point light source.

Explosions can be displayed in a 3-D scene procedure similar to those used for fire, flashes, and flares. For instance, your application might use a billboard to display a shock wave and rising plume of smoke when the explosion occurs. At the same time, your application can use a set of billboards to simulate flames. In addition, it can position a single billboard in front of the scene to add a flare of light to the entire scene.

Energy beams can be simulated using billboards. Your application can also display them using primitives that are defined as line lists or line strips. For details, see Line Lists and Line Strips.

Your application can create force fields using billboards or primitives defined as triangle lists. To create a force field from triangle lists, define a set of disjoint triangles in a triangle list equally spaced over the region covered by the force field. The gaps between the triangles allow the user to see the scene behind the triangles, as you might expect when looking at a force field. Apply a texture to the triangle list that gives the triangles the appearance of glowing with energy. For further information, see Triangle Lists.

Microsoft DirectX 8.1 (C++)

Depth Buffers

[This is preliminary documentation and is subject to change.]

A depth buffer, often called a z-buffer or a w-buffer, is a property of the device that stores depth information to be used by Microsoft® Direct3D®. When Direct3D renders a 3-D scene to a target surface, it can use the memory in an associated depth-buffer surface as a workspace to determine how the pixels of rasterized polygons occlude one another. Direct3D uses an off-screen Direct3D surface as the target to which final color values are written. The depth-buffer surface that is associated with the render-target surface is used to store depth information that tells Direct3D how deep each visible pixel is in the scene.

When a 3-D scene is rasterized with depth buffering enabled, each point on the rendering surface is tested. The values in the depth buffer can be a point's z-coordinate or its homogeneous w-coordinate—from the point's (x,y,z,w) location in projection space. A depth buffer that uses z values is often called a z-buffer, and one that uses w values is called a w-buffer. Each type of depth buffer has advantages and disadvantages, which are discussed later.

At the beginning of the test, the depth value in the depth buffer is set to the largest possible value for the scene. The color value on the rendering surface is set to either the background color value or the color value of the background texture at that point. Each polygon in the scene is tested to see if it intersects with the current coordinate (x,y) on the rendering surface. If it does, the depth value—which will be the z coordinate in a z-buffer, and the w coordinate in a w-buffer—at the current point

is tested to see if it is smaller than the depth value stored in the depth buffer. If the depth of the polygon value is smaller, it is stored in the depth buffer and the color value from the polygon is written to the current point on the rendering surface. If the depth value of the polygon at that point is larger, the next polygon in the list is tested. This process is shown in the following illustration.

Note Although most applications don't use this feature, you can change the comparison that Direct3D uses to determine which values are placed in the depth buffer and subsequently the render-target surface. To do so, change the value for the D3DRS_ZFUNC render state.

Nearly all 3-D accelerators on the market support z-buffering, making z-buffers the most common type of depth buffer today. However ubiquitous, z-buffers have their drawbacks. Due to the mathematics involved, the generated z values in a z-buffer tend not to be distributed evenly across the z-buffer range (typically 0.0 to 1.0, inclusive). Specifically, the ratio between the far and near clipping planes strongly affects how unevenly z values are distributed. Using a far-plane distance to near-plane distance ratio of 100, 90% of the depth buffer range is spent on the first 10% of the scene depth range. Typical applications for entertainment or visual simulations with exterior scenes often require far-plane/near-plane ratios of anywhere between 1000 to 10000. At a ratio of 1000, 98% of the range is spent on the 1st 2% of the depth range, and the distribution becomes worse with higher ratios. This can cause hidden surface artifacts in distant objects, especially when using 16-bit depth buffers, the most commonly supported bit-depth.

A w-based depth buffer, on the other hand, is often more evenly distributed between the near and far clip planes than a z-buffer. The key benefit is that the ratio of distances for the far and near clipping planes is no longer an issue. This allows applications to support large maximum ranges, while still getting relatively accurate depth buffering close to the eye point. A w-based depth buffer isn't perfect, and can sometimes exhibit hidden surface artifacts for near objects. Another drawback to the w-buffered approach is related to hardware support: w-buffering isn't supported as widely in hardware as z-buffering.

Z-buffering requires overhead during rendering. Various techniques can be used to optimized rendering when using z-buffers. For details, see Z-Buffer Performance.

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

This section presents information on using depth buffers for hidden line and hidden surface removal.

Additional information is contained in the following topics.

- Querying for Depth Buffer Support
- Creating a Depth Buffer
- Enabling Depth Buffering
- Retrieving a Depth Buffer
- Clearing Depth Buffers
- Changing Depth Buffer Write Access
- Changing Depth Buffer Comparison Functions
- Using Z-Bias

Microsoft DirectX 8.1 (C++)

Querying for Depth Buffer Support

[This is preliminary documentation and is subject to change.]

As with any feature, the driver that your application uses might not support all types of depth buffering. Always check the driver's capabilities. Although most drivers support z-based depth buffering, not all will support w-based depth buffering. Drivers do not fail if you attempt to enable an unsupported scheme. They fall back on another depth buffering method instead, or sometimes disable depth buffering altogether, which can result in scenes rendered with extreme depth-sorting artifacts.

You can check for general support for depth buffers by querying Microsoft® Direct3D® for the display device that your application will use before you create a Direct3D device. If the Direct3D object reports that it supports depth buffering, any hardware devices you create from this Direct3D object will support z-buffering.

To query for depth buffering support, you can use the **IDirect3D8::CheckDeviceFormat** method, as shown in the following code example.

```
// The following example assumes that pCaps is a valid pointer to an
// initialized D3DCAPS8 structure.
    if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                           pCaps->DeviceType, Format,
                                           D3DUSAGE_DEPTHSTENCIL,
                                           D3DRTYPE_SURFACE,
                                           D3DFMT_D16 ) ) )
        return E_FAIL;
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 16-bit depth buffers are rejected.

In addition, you can use **IDirect3D8::CheckDepthStencilMatch** to determine whether a supported depth-stencil format is compatible with the render target format in the display mode, particularly if a depth format must be equal to the render target format.

Using **CheckDepthStencilMatch** to determine depth-stencil compatibility with a render target is illustrated in the following code example.

```
// Reject devices that cannot create a render target of RTFormat while
// the back buffer is of RTFormat and the depth-stencil buffer is
// at least 8 bits of stencil.
if( FAILED( m_pD3D->CheckDepthStencilMatch( pCaps->AdapterOrdinal,
                                            pCaps->DeviceType,
                                            AdapterFormat,
                                            RTFormat,
                                            D3DFMT_D24S8 ) ) )
        return E_FAIL;
```

Once you know that the driver supports depth buffers, you can verify w-buffer support. Although depth buffers are supported for all software rasterizers, w-buffers are supported only by the reference rasterizer, which is not suited for use by real-world applications. Regardless of the type of device your application uses, verify support for w-buffers before you attempt to enable w-based depth buffering.

1. After creating your device, call the **IDirect3DDevice8::GetDeviceCaps** method, passing an initialized **D3DCAPS8** structure.
2. After the call, the **LineCaps** member contains information about the driver's support for rendering primitives.

3. If the **RasterCaps** member of this structure contains the D3DPRASTERCAPS_WBUFFER flag, then the driver supports w-based depth buffering for that primitive type.

Microsoft DirectX 8.1 (C++)

Creating a Depth Buffer

[This is preliminary documentation and is subject to change.]

A depth buffer is a property of the device. To create a depth buffer that is managed by Microsoft® Direct3D®, set the appropriate members of the **D3DPRESENT_PARAMETERS** structure as shown in the following code example.

```
D3DPRESENT_PARAMETERS d3dpp;

ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed               = TRUE;
d3dpp.SwapEffect             = D3DSWAPEFFECT_COPY_VSYNC;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
```

By setting the **EnableAutoDepthStencil** member to TRUE, you instruct Direct3D to manage depth buffers for the application. Note that **AutoDepthStencilFormat** must be set to a valid depth buffer format. The D3DFMT_D16 flag specifies a 16-bit depth buffer, if one is available.

The following call to the **IDirect3D8::CreateDevice** method creates a device that then creates a depth buffer.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &d3dDevice ) ) )
    return E_FAIL;
```

The depth buffer is automatically set as the render target of the device. When the device is reset, the depth buffer is automatically destroyed and recreated in the new size.

To create a new depth buffer surface, use the **IDirect3DDevice8::CreateDepthStencilSurface** method.

To set a new depth-buffer surface for the device, use the **IDirect3DDevice8::SetRenderTarget** method.

To use the depth buffer in your application, you need to enable the depth buffer. For details, see Enabling Depth Buffering.

Microsoft DirectX 8.1 (C++)

Enabling Depth Buffering

[This is preliminary documentation and is subject to change.]

After you create a depth buffer, as described in Creating a Depth Buffer, you enable depth buffering by calling the **IDirect3DDevice8::SetRenderState** method. Set the D3DRS_ZENABLE render state

to enable depth-buffering. Use the D3DZB_TRUE member of the **D3DZBUFFERTYPE** enumerated type (or TRUE) to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE (or FALSE) to disable depth buffering.

Note To use w-buffering, your application must set a compliant projection matrix even if it doesn't use the Microsoft® Direct3D® transformation pipeline. For information about providing an appropriate projection matrix, see A W-Friendly Projection Matrix. The projection matrix discussed in What Is the Projection Transformation? is compliant.

Microsoft DirectX 8.1 (C++)

Retrieving a Depth Buffer

[This is preliminary documentation and is subject to change.]

The following code example shows how to use the **IDirect3DDevice8::GetDepthStencilSurface** method to retrieve a pointer to the depth-buffer surface owned by the device.

```
LPDIRECT3DSURFACE8 pZBuffer;

m_d3dDevice->GetDepthStencilSurface( &pZBuffer );
```

Microsoft DirectX 8.1 (C++)

Clearing Depth Buffers

[This is preliminary documentation and is subject to change.]

Many C++ applications clear the depth buffer before rendering each new frame. You can explicitly clear the depth buffer through Microsoft® Direct3D® by calling **IDirect3DDevice8::Clear** and specifying D3DCLEAR_ZBUFFER for the *Flags* parameter. The **Clear** method allows you to specify an arbitrary depth value in the *Z* parameter.

Microsoft DirectX 8.1 (C++)

Changing Depth Buffer Write Access

[This is preliminary documentation and is subject to change.]

By default, the Microsoft® Direct3D® system is allowed to write to the depth buffer. Most applications leave writing to the depth buffer enabled, but you can achieve some special effects by not allowing the Direct3D system to write to the depth buffer.

You can disable depth buffer writes in C++ by calling the **IDirect3DDevice8::SetRenderState** method with the *State* parameter set to D3DRS_ZWRITEENABLE and the *Value* parameter set to 0.

Microsoft DirectX 8.1 (C++)

Changing Depth Buffer Comparison Functions

[This is preliminary documentation and is subject to change.]

By default, when depth-testing is performed on a rendering surface, the Microsoft® Direct3D® system updates the render-target surface if the corresponding depth value (z or w) for each point is less than the value in the depth buffer. In a C++ application, you change how the system performs comparisons on depth values by calling the **IDirect3DDevice8::SetRenderState** method with the *State* parameter set to D3DRS_ZFUNC. The *Value* parameter should be set to a value in the **D3DCMPFUNC** enumerated type.

Microsoft DirectX 8.1 (C++)

Using Z-Bias

[This is preliminary documentation and is 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 and then the shadow, the shadow might not be visible, or depth artifacts might be visible. You can reverse the order in which you render the coplanar objects in hopes of reversing the effect, but depth artifacts are still likely.

A C++ application can help ensure that coplanar polygons are rendered properly by adding a bias to the z-values that the system uses when rendering the sets of coplanar polygons. To add a z-bias to a set of polygons, call the **IDirect3DDevice8::SetRenderState** method just before rendering them, setting the *State* parameter to D3DRS_ZBIAS, and the *Value* parameter to a value between 0-16 inclusive. A higher z-bias value increases the likelihood that the polygons you render will be visible when displayed with other coplanar polygons.

Microsoft DirectX 8.1 (C++)

About Devices

[This is preliminary documentation and is subject to change.]

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

Direct3D currently supports two main types of Direct3D devices: a hardware abstraction layer (HAL) device with hardware-accelerated rasterization and shading with both hardware and software vertex processing; and a reference device.

You can think of these devices as two separate drivers. Software and reference devices are represented by software drivers, and the HAL device is represented by a hardware driver. The most

common way to take advantage of these devices is to use the HAL device for shipping applications, and the reference device for feature testing. These are provided by third parties to emulate particular devices—for example, developmental hardware that has not yet been released.

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

Hardware-accelerated devices give much better performance than software devices. The HAL device type is available on all Direct3D supported graphic adapters. In most cases, applications target computers that have hardware acceleration and rely on software emulation to accommodate lower-end computers.

With the exception of the reference device, software devices do not always support the same features as a hardware device. Applications should always query for device capabilities to determine which features are supported.

Because the behavior of the software and reference devices provided with Microsoft® DirectX® 8.0 is identical to that of the HAL device, application code authored to work with the HAL device will work with the software or reference devices without modifications. Note that while the provided software or reference device behavior is identical to that of the HAL device, the device capabilities do vary, and a particular software device may implement a much smaller set of capabilities.

Behaviors

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

Additional information is contained in the following topics.

- Device Types
- Lost Devices
- Using Devices

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

For more information, see
- Device States

Microsoft DirectX 8.1 (C++)

Device Types

[This is preliminary documentation and is subject to change.]

HAL Device

The primary device type is the hardware abstraction layer (HAL) device, which supports hardware accelerated rasterization and both hardware and software vertex processing. If the computer on which your application is running is equipped with a display adapter that supports Microsoft® Direct3D®, your application should use it for 3-D operations. Direct3D HAL devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.

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

To create a HAL device from C++, call the **IDirect3D8::CreateDevice** method, and pass the D3DDEVTYPE_HAL constant as the device type.

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

Reference Device

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

To create a reference device from C++, call the **IDirect3D8::CreateDevice** method, and pass the D3DDEVTYPE_REF constant as the device type.

Microsoft DirectX 8.1 (C++)

Lost Devices

[This is preliminary documentation and is subject to change.]

A Microsoft® Direct3D® device can be in either an operational state or a lost state. The operational state is the normal state of the device; the device executes and presents all rendering as expected. The device makes a transition to the lost state when an event, such as the loss of keyboard focus in a full-screen application, causes rendering to become impossible. The lost state is characterized by the silent failure of all rendering operations, which means that the rendering methods can return success codes even though the rendering operations fail. In this situation, the error code D3DERR_DEVICELOST is returned by **IDirect3DDevice8::Present**.

By design, the full set of scenarios that can cause a device to become lost is not specified. Some typical examples include loss of focus, such as when the user presses ALT+TAB or when a system dialog is initialized. Devices can also be lost due to a power management event, or when another application assumes full-screen operation. In addition, any failure from **IDirect3DDevice8::Reset** puts the device into a lost state.

Note The only methods that are guaranteed to work after a device is lost are **IDirect3DDevice8::TestCooperativeLevel**, **IDirect3DDevice8::Reset**, and **IUnknown::Release**. It is invalid to call any other application programming interfaces (APIs).

Responding to a Lost Device

If a device is lost, the application queries the device to see if it can be restored to the operational state. If not, the application waits until the device can be restored.

If the device can be restored, the application prepares the device by destroying all video-memory resources and any swap chains. Then, the application calls the **IDirect3DDevice8::Reset** method. **Reset** is the only method that has an effect when a device is lost, and is the only method by which an application can change the device from a lost to an operational state. **Reset** will fail unless the application releases all resources that are allocated in D3DPOOL_DEFAULT, including those created by the **IDirect3DDevice8::CreateRenderTarget** and **IDirect3DDevice8::CreateDepthStencilSurface** methods.

For the most part, the high-frequency calls of Direct3D do not return any information about whether the device has been lost. The application can continue to call rendering methods, such as **IDirect3DDevice8::DrawPrimitive**, without receiving notification of a lost device. Internally, these operations are discarded until the device is reset to the operational state.

The application can determine what to do on encountering a lost device by querying the return value of the **IDirect3DDevice8::TestCooperativeLevel** method.

- If the method returns D3D_OK, the device is operational.
- If the device is lost but cannot be restored at the current time, the return value is D3DERR_DEVICELOST. This is the case when the user presses ALT+TAB, causing a full-screen device to lose focus. Applications should respond by pausing until the device can be reset. A D3DERR_DEVICENOTRESET return code from **TestCooperativeLevel** indicates this situation.
- If the device is lost and can be restored, the return code from **TestCooperativeLevel** is D3DERR_DEVICENOTRESET. Note that the return code from the **IDirect3DDevice8::Present** method is still D3DERR_DEVICELOST.

In all cases, destroying video-memory resources is a prerequisite to calling **Reset**, even if the device has not been lost.

Locking Operations

Internally, Direct3D does enough work to ensure that a lock operation will succeed after a device is lost. However, it is not guaranteed that the video-memory resource's data will be accurate during the lock operation. It is guaranteed that no error code will be returned. This allows applications to be written without concern for device loss during a lock operation.

Resources

Resources can consume video memory. Because a lost device is disconnected from the video memory owned by the adapter, it is not possible to guarantee allocation of video memory when the device is lost. As a result, all resource creation methods are implemented to succeed by returning D3D_OK, but do in fact allocate only dummy system memory. Because any video-memory resource must be destroyed before the device is resized, there is no issue of over-allocating video memory. These dummy surfaces allow lock and copy operations to appear to function normally until the application calls **IDirect3DDevice8::Present** and discovers that the device has been lost.

All video memory must be released before a device can be reset from a lost state to an operational state. This means that the application should release any swap chains created with **IDirect3DDevice8::CreateAdditionalSwapChain** and any resources placed in the

D3DPOOL_DEFAULT memory class. The application need not release resources in the D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM memory classes. Other state data is automatically destroyed by the transition to an operational state.

You are encouraged to develop applications with a single code path to respond to device loss. This code path is likely to be similar, if not identical, to the code path taken to initialize the device at startup.

Retrieved Data

Direct3D allows applications to validate texture and render states against single pass rendering by the hardware using **IDirect3DDevice8::ValidateDevice**. This method, which is typically invoked during initialization of the application, will return D3DERR_DEVICELOST if the device has been lost.

Direct3D also allows applications to copy generated or previously written images from video-memory resources to nonvolatile system-memory resources. Because the source images of such transfers might be lost at any time, Direct3D allows such copy operations to fail when the device is lost.

The copy operation **IDirect3DDevice8::CopyRects** can return D3DERR_DEVICELOST when the source object is in volatile memory (D3DPOOL_DEFAULT) and the destination object is in nonvolatile memory (D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED). Another copy operation, **IDirect3DDevice8::GetFrontBuffer**, can fail due to retrieving data from the primary surface. Note that these cases are the only instance of D3DERR_DEVICELOST outside of the **IDirect3DDevice8::Present**, **IDirect3DDevice8::TestCooperativeLevel**, and **IDirect3DDevice8::Reset** methods.

Programmable Shaders

In Microsoft® DirectX® 8.1, **programmable vertex shaders** and **programmable pixel shaders** don't need to be recreated after **Reset**. They will be remembered. In previous versions of DirectX, a lost device required shaders to be recreated.

Microsoft DirectX 8.1 (C++)

Using Devices

[This is preliminary documentation and is subject to change.]

This section provides information about using Microsoft® Direct3D® devices in an application. Information is divided into the following topics.

- Determining Hardware Support
- Selecting a Device
- Creating a Device
- Setting Transformations
- Processing Vertex Data
- Working with Mouse Cursors
- Direct3D Resources

Microsoft DirectX 8.1 (C++)

Determining Hardware Support

[This is preliminary documentation and is subject to change.]

Direct3D provides the following functions to determine hardware support.

IDirect3D8::CheckDeviceFormat
Used to verify whether a surface format can be used as a texture, whether a surface format can be used as a texture and a render target, or whether a surface format can be used as a depth-stencil buffer. In addition, this method is used to verify depth buffer format support and depth-stencil buffer format support.

IDirect3D8::CheckDeviceType
Used to verify a device's ability to perform hardware acceleration, a device's ability to build a swap chain for presentation, or a device's ability to render to the current display format.

IDirect3D8::CheckDepthStencilMatch
Used to verify whether a depth-stencil buffer format is compatible with a render-target format. Note that before calling this method, the application should call **CheckDeviceFormat** on both the depth-stencil and render target formats.

Microsoft DirectX 8.1 (C++)

Selecting a Device

[This is preliminary documentation and is subject to change.]

Applications can query hardware to detect the supported Microsoft® Direct3D® device types. This section contains information on the primary tasks involved in enumerating display adapters and selecting Direct3D devices.

An application must perform a series of tasks to select an appropriate Direct3D device. Note that the following steps are intended for a full-screen application. In most cases, a windowed application can skip most of these steps.

1. Initially, the application must enumerate the display adapters on the system. An adapter is a physical piece of hardware. Note that the graphics card might contain more than a single adapter, as is the case with a dual-head display. Applications that are not concerned with multimonitor support can disregard this step, and pass D3DADAPTER_DEFAULT to the **IDirect3D8::EnumAdapterModes** method in step 2.
2. For each adapter, the application enumerates the supported display modes by calling **EnumAdapterModes**.
3. If required, the application checks for the presence of hardware acceleration in each enumerated display mode by calling **IDirect3D8::CheckDeviceType**, as shown in the following code example. Note that this is only one of the possible uses for **CheckDeviceType**; for details, see Determining Hardware Support.

```
D3DPRESENT_PARAMETERS Params;
// Initialize values for D3DPRESENT_PARAMETERS members.

Params.BackBufferFormat = D3DFMT_X1R5G5B5;

if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter,
                                  Device.m_DevType,
```

```
                                        Params.BackBufferFormat, Params.BackBuffer
                                        FALSE)))
        return E_FAIL;
```

4. The application checks for the desired level of functionality for the device on this adapter by calling the **IDirect3D8::GetDeviceCaps** method. This method filters out those devices that do not support the required functionality. The device capability returned by **GetDeviceCaps** is guaranteed to be constant for a device across all display modes verified by **CheckDeviceType**.
5. Devices can always render to surfaces of the format of an enumerated display mode that is supported by the device. If the application is required to render to a surface of a different format, it can call **IDirect3D8::CheckDeviceFormat**. If the device can render to the format, then it is guaranteed that all capabilities returned by **GetDeviceCaps** are applicable.
6. Lastly, the application can determine if multisampling techniques, such as full-scene antialiasing, are supported for a render format by using the **IDirect3D8::CheckDeviceMultiSampleType** method.

After completing the above steps, the application should have a list of display modes in which it can operate. The final step is to verify that enough device-accessible memory is available to accommodate the required number of buffers and antialiasing. This test is necessary because the memory consumption for the mode and multisample combination is impossible to predict without verifying it. Moreover, some display adapter architectures might not have a constant amount of device-accessible memory. This means that an application should be able to report out-of-video-memory failures when going into full-screen mode. Typically, an application should remove full-screen mode from the list of modes it offers to a user, or it should attempt to consume less memory by reducing the number of back buffers or using a less expensive multisampling technique.

A windowed application performs a similar set of tasks.

1. It determines the desktop rectangle covered by the client area of the window.
2. It enumerates adapters, looking for the adapter whose monitor covers the client area. If the client area is owned by more than one adapter, then the application can choose to drive each adapter independently, or to drive a single adapter and have Direct3D transfer pixels from one device to another at presentation. The application can also disregard the above two steps and use the D3DADAPTER_DEFAULT adapter. Note that this might result in slower operation when the window is placed on a secondary monitor.
3. The application should call **CheckDeviceType** to determine if the device can support rendering to a back buffer of the specified format while in desktop mode. **IDirect3D8::GetAdapterDisplayMode** can be used to determine the desktop display format, as shown in the following code example.

```
D3DPRESENT_PARAMETERS Params;
// Initialize values for D3DPRESENT_PARAMETERS members.

// Use the current display mode.
D3DDISPLAYMODE mode;

if(FAILED(m_pD3D->GetAdapterDisplayMode(Device.m_uAdapter , &mode)))
        return E_FAIL;

Params.BackBufferFormat = mode.Format;

if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter, Device.m_DevType,
Params.BackBufferFormat, Params.BackBufferFormat, FALSE)))
       return E_FAIL;
```

Creating a Device

[This is preliminary documentation and is subject to change.]

Note All rendering devices created by a given Microsoft® Direct3D® object share the same physical resources. Although your application can create multiple rendering devices from a single Direct3D object, because they share the same hardware, extreme performance penalties will be incurred.

To create a Direct3D device in a C++ application, your application must first create a Direct3D object, as explained in Direct3D Object.

First, initialize values for the **D3DPRESENT_PARAMETERS** structure that is used to create the Direct3D device. The following code example specifies a windowed application where the back buffer is flipped to the front buffer on VSYNC only.

```
LPDIRECT3DDEVICE8 d3dDevice = NULL;

D3DPRESENT_PARAMETERS d3dpp;

ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC;
```

Next, create the Direct3D device. The following **IDirect3D8::CreateDevice** call specifies the default adapter, a hardware abstraction layer (HAL) device, and software vertex processing.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                      &d3dpp, &d3dDevice ) ) )
    return E_FAIL;
```

Note that a call to create, release, or reset the device should happen only on the same thread as the window procedure of the focus window.

After creating the device, set its state.

Microsoft DirectX 8.1 (C++)

Setting Transformations

[This is preliminary documentation and is subject to change.]

In C++, transformations are applied by using the **Direct3DDevice8.SetTransform** method. For example, you can use code like the following to set the view transformation.

```
HRESULT    hr;
D3DMATRIX  view;

// Fill in the view matrix.

hr = pDev->SetTransform(D3DTS_VIEW, &view);

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

}

There are several possible settings for the first parameter in a call to **SetTransform**. The most common are D3DTS_WORLD, D3DTS_VIEW, and D3DTS_PROJECTION. These transformation states are defined by the **D3DTRANSFORMSTATETYPE** enumerated type. This enumeration also contains the states D3DTS_TEXTURE1 through D3DTS_TEXTURE7, which you can use to apply transformations to texture coordinates.

Microsoft DirectX 8.1 (C++)

Processing Vertex Data

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8** interface supports vertex processing in both software and hardware. In general, the device capabilities for software and hardware vertex processing are not identical. Hardware capabilities are variable, depending on the display adapter and driver, while software capabilities are fixed.

The following flags control vertex processing behavior for the hardware abstraction layer (HAL) and reference devices.

- D3DCREATE_SOFTWARE_VERTEXPROCESSING
- D3DCREATE_HARDWARE_VERTEXPROCESSING
- D3DCREATE_MIXED_VERTEXPROCESSING

Specify one of the vertex processing behavior flags when calling **IDirect3D8::CreateDevice**. The mixed-mode flag enables the device to perform both software and hardware vertex processing. Only one vertex processing flag may be set for a device at any one time. Note that the D3DCREATE_HARDWARE_VERTEXPROCESSING flag is required to be set when creating a pure device (D3DCREATE_PUREDEVICE).

To avoid dual vertex processing capabilities on a single device, only the hardware vertex processing capabilities can be queried at run time. Software vertex processing capabilities are fixed and cannot be queried at run time.

You can consult the **VertexProcessingCaps** member of the **D3DCAPS8** structure to determine the hardware vertex processing capabilities of the device. For software vertex processing the following capabilities are supported.

- D3DVTXPCAPS_DIRECTIONALLIGHTS
- D3DVTXPCAPS_LOCALVIEWER
- D3DVTXPCAPS_MATERIALSOURCE7
- D3DVTXPCAPS_POSITIONALLIGHTS
- D3DVTXPCAPS_TEXGEN
- D3DVTXPCAPS_TWEENING

In addition, the following table lists the values that are set for members of the **D3DCAPS8** structure for a device in software vertex processing mode.

Member	Software Vertex Processing Capabilities
MaxActiveLights	Unlimited

MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	0xFFFFFFFF

Software vertex processing provides a guaranteed set of vertex processing capabilities, including an unbounded number of lights and full support for programmable vertex shaders. You can toggle between software and hardware vertex processing at any time when using the HAL Device, the only device type that supports both hardware and software vertex processing. The only requirement is that vertex buffers used for software vertex processing must be allocated in system memory.

Note The performance of hardware vertex processing is comparable to that of software vertex processing. For this reason, it's a good idea to provide, within a single device type, both hardware- and software-emulation functionality for vertex processing. This is not the case for rasterization, for which host processors are much slower than specialized graphics hardware. Thus both hardware- and software-emulated rasterization is not provided within a single device type. Software vertex processing is the only instance of functionality duplicated between the run time and the hardware (driver) within a single device. Thus all other device capabilities represent potentially variable functionality provided by the driver.

Microsoft DirectX 8.1 (C++)

Working with Mouse Cursors

[This is preliminary documentation and is subject to change.]

The mouse cursor methods enable the application to specify a color cursor by providing a surface that contains an image. The system will ensure that this cursor will be updated at half the display rate or more if the application's frame rate is slow. However, the cursor will never be updated more frequently than the display refresh rate.

The mouse cursor position is tied to the system cursor, appropriately scaled for the current display mode spatial resolution, but it can be moved explicitly by the application. This is analogous to the behavior of the Microsoft® Win32® API–supported system mouse cursor. For more information on how to use a mouse cursor in your Microsoft® Direct3D® application, see the following reference topics.

- **IDirect3DDevice8::ShowCursor**
- **IDirect3DDevice8::SetCursorPosition**
- **IDirect3DDevice8::SetCursorProperties**

Direct3D ensures that the mouse is supported either by hardware implementations or by the Direct3D run time that performs hardware-accelerated blitting operations when calling **IDirect3DDevice8::Present**.

Microsoft DirectX 8.1 (C++)

Direct3D Resources

[This is preliminary documentation and is subject to change.]

Resources are the textures and buffers that are used to render a scene. Applications need to create, load, copy, and use resources. This section gives a brief introduction to resources and the steps and methods used by applications when working with resources. For more information on specific resource types, see Textures, Vertex Buffers, and Index Buffers.

All resources, including the geometry resources **IDirect3DIndexBuffer8** and **IDirect3DVertexBuffer8**, inherit from the **IDirect3DResource8** interface. The texture resources, **IDirect3DCubeTexture8**, **IDirect3DTexture8**, and **IDirect3DVolumeTexture8**, also inherit from the **IDirect3DBaseTexture8** interface.

Additional information is divided into the following topics.

- Resource Properties
- Resource Relationships
- Managing Resources
- Application-Managed Resources and Allocation Strategies
- Manipulating Resources
- Locking Resources

Microsoft DirectX 8.1 (C++)

Resource Properties

[This is preliminary documentation and is subject to change.]

All resources share the following properties.

- *Usage*. The way a resource is used—for example, as a texture or a render target.
- *Format*. The format of the data—for example, the pixel format of a 2-D surface.
- *Pool*. The type of memory where the resource is allocated.
- *Type*. The type of resource—for example, a vertex buffer or render target.

Resource uses are enforced. An application that will use a resource in a certain operation must specify that operation at resource creation time. The following usages are defined for resources.

- D3DUSAGE_DEPTHSTENCIL
- D3DUSAGE_DONOTCLIP
- D3DUSAGE_DYNAMIC
- D3DUSAGE_RTPATCHES
- D3DUSAGE_NPATCHES
- D3DUSAGE_POINTS
- D3DUSAGE_RENDERTARGET
- D3DUSAGE_SOFTWAREPROCESSING
- D3DUSAGE_WRITEONLY

The D3DUSAGE_RTPATCHES, D3DUSAGE_NPATCHES, and D3DUSAGE_POINTS flags indicate to the driver that the data in these buffers is likely to be used for triangular or grid patches, N patches, or point sprites, respectively. These flags are provided in case the hardware cannot perform these operations without host processing. Therefore, the driver will want to allocate these surfaces in system memory so that the CPU can access them. If the driver can perform these operations entirely in hardware, then it can allocate these surfaces in video or /AGP memory to avoid a host copy and improve performance at least twofold. Note that the information provided by these flags is not absolutely required. A driver can detect that such operations are being performed on the

data, and it will move the buffer back to system memory for subsequent frames.

For details on the usage flags and how they relate to specific resources, see the reference pages on the individual resource creation methods.

For information on the surface format of resources, see the **D3DFORMAT** enumerated type.

The class of memory that holds a resource's buffers is called a pool. Pool values are defined by the **D3DPOOL** enumerated type. A pool cannot be mixed for different objects contained in a single resource—that is, mip levels in a mipmap—and once a pool is chosen for a resource, the pool cannot be changed.

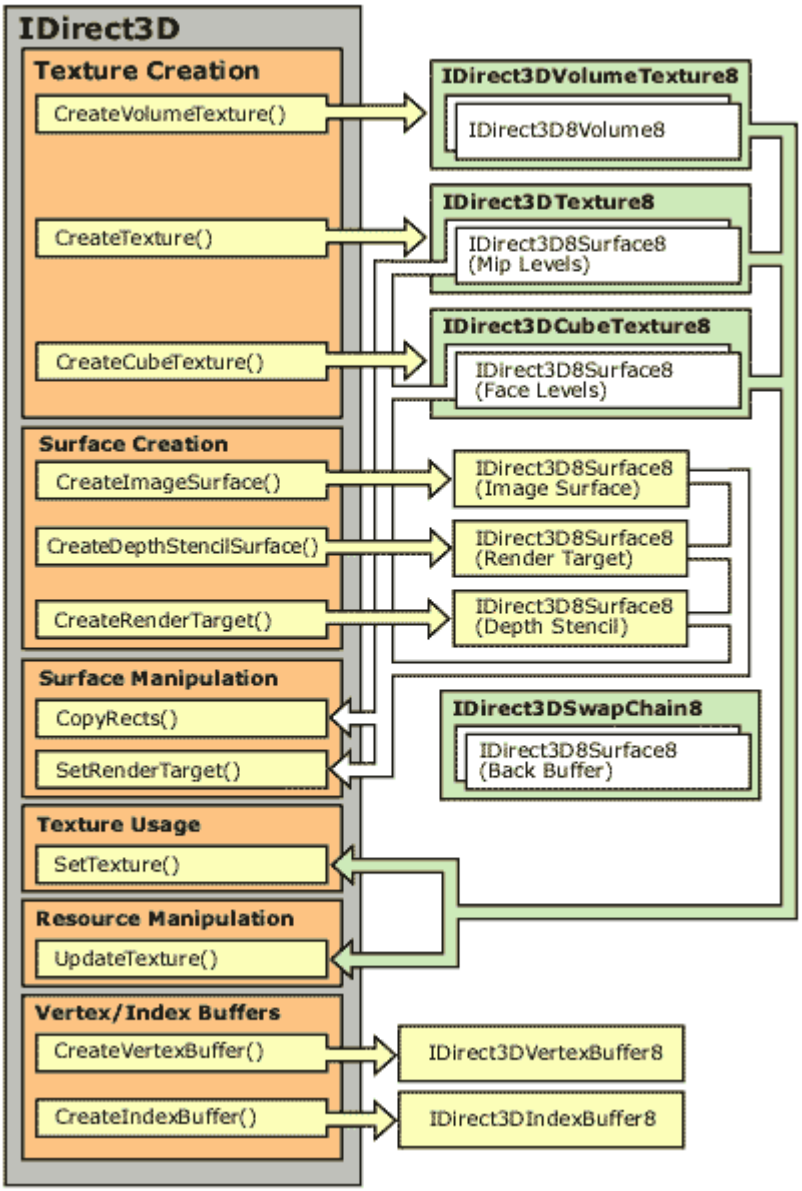
The resources types are set implicitly at run time when the application calls a resource creation method such as **IDirect3DDevice8::CreateCubeTexture**. Resource types are defined by the **D3DRESOURCETYPE** enumerated type. Applications can query these types at run time; however, it is expected that most scenarios will not require run-time type checking.

Microsoft DirectX 8.1 (C++)

Resource Relationships

[This is preliminary documentation and is subject to change.]

The following diagram illustrates the operations that are syntactically possible on specific resources and their contents. For more information on how to use resources, see Manipulating Resources.



Arrows running from left to right indicate that the target types are created by the given methods, while arrows running from right to left indicate that those resource types can be passed as arguments to the given methods.

Microsoft DirectX 8.1 (C++)

Managing Resources

[This is preliminary documentation and is subject to change.]

Resource management is the process where resources are promoted from system-memory storage to device-accessible storage and discarded from device-accessible storage. The Microsoft® Direct3D® run time has its own management algorithm based on a least-recently-used priority technique. Direct3D switches to a most-recently-used priority technique when it detects that more resources than can coexist in device-accessible memory are used in a single frame—between **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene** calls.

Use the D3DPOOL_MANAGED flag at creation time to specify a managed resource. Managed

resources persist through transitions between the lost and operational states of the device. The device can be restored with a call to **IDirect3DDevice8::Reset**, and such resources continue to function normally without being reloaded with artwork. However, if the device must be destroyed and recreated, all resources created using D3DPOOL_MANAGED must be recreated.

Use the D3DPOOL_DEFAULT flag at creation time to specify that a resource be placed in the default pool. Resources in the default pool do not persist through transitions between the lost and operational states of the device. These resources must be released before calling **Reset** and must then be recreated.

For more information on the lost state of a device, see Lost Devices.

Note that resource management is not supported for all types and usages. For example, objects created with the D3DUSAGE_RENDERTARGET flag are not supported. In addition, resource management is not recommended for objects whose contents are changing with high frequency. For example, a managed vertex buffer that changes every frame can significantly degrade performance for some hardware. However, this is not a problem for texture resources.

Microsoft DirectX 8.1 (C++)

Application-Managed Resources and Allocation Strategies

[This is preliminary documentation and is subject to change.]

Managed vertex-buffer or index-buffer resources cannot be declared dynamic by specifying D3DUSAGE_DYNAMIC at creation time. This would require an additional copy for every modification to the vertex buffer contents. Dynamic vertex buffers are intended for rendering dynamic geometry as well as data pulled in from BSP trees or other visibility data structures. This can be accomplished by pre-allocating buffers of the desired format. These resources are then parceled out to support application needs by a resource manager within the application. Because there are only a few different vertex strides that an application will use simultaneously, and a different vertex buffer is required only for each unique stride, the total number of dynamic vertex buffers is small. When managing dynamic resources in this way, it is important to ensure that high-frequency demands on the resources do not significantly decrease the application's performance.

When simultaneously using both D3D-managed and application-managed resources, all application managed resources should be allocated in D3DPOOL_DEFAULT memory prior to creating any managed resources. Allocating resources in D3DPOOL_DEFAULT memory after allocating resources in D3DPOOL_MANAGED memory will give the Microsoft® Direct3D® memory manager an inaccurate account of available memory.

Microsoft DirectX 8.1 (C++)

Manipulating Resources

[This is preliminary documentation and is subject to change.]

Your application manipulates resources in order to render a scene. First, the application creates texture resources by using the following methods.

- **IDirect3DDevice8::CreateCubeTexture**

- **IDirect3DDevice8::CreateTexture**
- **IDirect3DDevice8::CreateVolumeTexture**

The texture objects returned by the texture creation methods are containers for surfaces or volumes; these containers are generically known as buffers. The buffers owned by the resource inherit the usages, format, and pool of the resource but have their own type. For more information, see Resource Properties.

The application gains access to the contained surfaces, for the purpose of loading artwork, by calling the following methods. For details, see Locking Resources.

- **IDirect3DCubeTexture8::LockRect**
- **IDirect3DTexture8::LockRect**
- **IDirect3DVolumeTexture8::LockBox**

The lock methods take arguments denoting the contained surface—for example, the mipmap sub-level or cube face of the texture—and return pointers to the pixels. The typical application never uses a surface object directly.

In addition, the application creates geometry-oriented resources by using the following methods.

- **IDirect3DDevice8::CreateIndexBuffer**
- **IDirect3DDevice8::CreateVertexBuffer**

Your application locks and fills the buffer resources by calling the following methods.

- **IDirect3DIndexBuffer8::Lock**
- **IDirect3DVertexBuffer8::Lock**

If your application is allowing the Microsoft® Direct3D® run time to manage these resources, then the resource creation process ends here. Otherwise, the application manages the promotion of system memory resources to device-accessible resources, where the hardware accelerator can use them, by calling the **IDirect3DDevice8::UpdateTexture** method.

To present images rendered from resources, the application also needs color and depth-stencil buffers. For typical applications, the color buffer is owned by the device's swap chain, which is a collection of back buffer surfaces, and is implicitly created with the device. Depth-stencil surfaces can be implicitly created, or explicitly created by using the **IDirect3DDevice8::CreateDepthStencilSurface** method. The application associates a device and its depth and color buffer with a call to **IDirect3DDevice8::SetRenderTarget**.

For details on presenting the final image, see Presenting a Scene.

Microsoft DirectX 8.1 (C++)

Locking Resources

[This is preliminary documentation and is subject to change.]

Locking a resource means granting CPU access to its storage. The following locking methods are defined for resources.

- D3DLOCK_DISCARD
- D3DLOCK_READONLY
- D3DLOCK_NOOVERWRITE
- D3DLOCK_NOSYSLOCK
- D3DLOCK_NO_DIRTY_UPDATE

For details on locking flags and how they relate to specific resources, see the reference pages on the individual resource locking methods. Application developers should note that the D3DLOCK_DISCARD, D3DLOCK_READONLY, and D3DLOCK_NOOVERWRITE flags are only hints. The run time does not check that applications are following the functionality specified by these flags. An application that specifies D3DLOCK_READONLY but then writes to the resource should expect undefined results. An application that specifies D3DUSAGE_WRITEONLY but then reads from the resource should expect a significant performance penalty. In general, working against locking flags, including the locking usage flags, is not guaranteed to work in later releases and may incur a significant performance penalty.

A lock operation is followed by an unlock operation. For example, after locking a texture, the application subsequently relinquishes direct access to locked textures by unlocking them. In addition to granting processor access, any other operations involving that resource are serialized for the duration of a lock. Only a single lock for a resource is allowed, even for non-overlapping regions, and no accelerator operations on a surface can be ongoing while a lock operation is outstanding on that surface.

Each resource interface has methods for locking the contained buffers. Each texture resource can also lock a sub-portion of that resource. Two-dimensional resources (surfaces) allow the locking of sub-rectangles, and volume resources allow the locking of sub-volumes or boxes. Each lock method returns a structure that contains a pointer to the storage backing the resource and values representing the distances between rows or planes of data, depending on the resource type. For details, see the method lists for the resource interfaces. The returned pointer always points to the top-left byte in the locked sub-regions.

When working with index and vertex buffers, you can make multiple lock calls; however, you must ensure that the number of lock calls matches the number of unlock calls.

The DXT set of compressed texture formats, which store pixels in encoded 4×4 blocks, can only be locked on 4×4 boundaries.

Microsoft DirectX 8.1 (C++)

Programmable Pipeline

[This is preliminary documentation and is subject to change.]

Using shaders and effects, developers can now program the pipeline. These topics are covered in the following sections.

- Vertex Shaders
- Pixel Shaders
- Effects

Microsoft DirectX 8.1 (shader versions 1.0, 1.1)

Vertex Shaders

[This is preliminary documentation and is subject to change.]

Previous to Microsoft® DirectX® 8.*n*, Microsoft® Direct3D® operated a fixed function pipeline for converting three-dimensional (3-D) geometry to rendered screen pixels. The user sets attributes of the pipeline that control how Direct3D transforms, lights, and renders pixels. The fixed function vertex format is declared at compile time and determines the input vertex format. Once defined, the user has little control over pipeline changes during runtime.

Programmable shaders add a new dimension to the graphics pipeline by allowing the transform, lighting, and rendering functionality to be modified at runtime. A shader is declared at runtime but, once done, the user is free to change which shader is active as well as to control the shader data dynamically using streaming data. This gives the user a new level of dynamic flexibility over the way that pixels are rendered.

A vertex shader file contains vertex shader instructions. Vertex shaders can control vertex color and how textures are applied to vertices. Lighting can be added through the use of vertex shader instructions. The shader instruction file contains ASCII text so it is readable and in some ways looks similar to assembly language. A vertex shader is invoked after any DrawPrimitive or DrawIndexedPrimitive call. Shaders can be dynamically switched using **SetVertexShader** to specify a new shader file, or by changing the instructions in the ASCII text shader file using the streaming data inputs. The Vertex Shader Assembler Reference has a complete listing of shader instructions.

For additional information, see the following sections.

- Create a Vertex Shader

  This section contains a code sample that uses a vertex shader to apply a constant color to object vertices. This example contains a detailed explanation of the methods used.

- Shader2 - Apply vertex colors

  Additional examples shows more code samples that add textures and blend vertex colors and textures.

- Shader3 - Apply a texture map

  Additional examples shows more code samples that add textures and blend vertex colors and textures.

- Shader4 - Apply a texture map with lighting

  Additional examples shows more code samples that add textures and blend vertex colors and textures.

- Debugging

Microsoft DirectX 8.1 (version 1.0, 1.1)

Create a Vertex Shader

[This is preliminary documentation and is subject to change.]

This example creates a vertex shader that applies a constant color to an object. The example will show the contents of the shader file as well as the code required in the application to set up the Microsoft® Direct3D® pipeline for the shader data.

To create a vertex shader

- Step 1: Declare the vertex data
- Step 2: Design the shader functionality
- Step 3: Check for vertex shader support
- Step 4: Declare the shader registers
- Step 5: Create the shader
- Step 6: Render the output pixels

If you already know how to build and run Direct3D samples, you can cut and paste code from this example into your existing application.

Step 1: Declare the vertex data

This example uses a quadrilateral that is made up of two triangles. The vertex data will contain (x,y,z) position and a diffuse color. The **D3DFVF_CUSTOMVERTEX** macro is defined to match the vertex data. The vertex data is declared in a global array of vertices (g_Vertices). The four vertices are centered about the origin, and each vertex is given a different diffuse color.

```
// Declare vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD diffuseColor;
};
// Declare custom FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)

// Declare the vertex position and diffuse color data.
CUSTOMVERTEX g_Vertices[]=
{
    //  x        y      z     diffuse color
    { -1.0f, -1.0f, 0.0f, 0xffff0000 },   // red   - bottom right
    { +1.0f, -1.0f, 0.0f, 0xff00ff00 },   // green - bottom left
    { +1.0f, +1.0f, 0.0f, 0xff0000ff },   // blue  - top left
    { -1.0f, +1.0f, 0.0f, 0xffffff00 },   // red and green = yellow - top right
};
```

Step 2: Design the shader functionality

This shader applies a constant color to each vertex. The shader file VertexShader.vsh follows:

```
vs.1.0              // version instruction
m4x4 oPos, v0, c0   // transform vertices by view/projection matrix
mov oD0, c4         // load constant color
```

This file contains three instructions.

The first instruction in a shader file must be the shader version declaration. This instruction (vs) declares the vertex shader version, which is 1.0 in this case.

The second instruction (m4x4) transforms the object vertices using the view/projection matrix. The matrix is loaded into four constant registers c0, c1, c2, c3 (as shown below).

The third instruction (mov) copies the constant color in register c4 to the output diffuse color register oD0. This results in coloring the output vertices.

Step 3: Check for vertex shader support

The device capability can be queried for vertex shader support before using a vertex shader.

```
D3DCAPS8 caps;
m_pd3dDevice->GetDeviceCaps(&caps);       // initialize m_pd3dDevice before using
if( D3DSHADER_VERSION_MAJOR( caps.VertexShaderVersion ) < 1 )
        return E_FAIL;
```

The caps structure returns the functional capabilities of the hardware after **GetDeviceCaps** is called. Use the **D3DSHADER_VERSION_MAJOR** macro to test the supported version number. If the version number is less than 1.0, this call will fail. The result of this method should be used to control whether or not vertex shaders are invoked by an application.

Step 4: Declare the shader registers

The shader is created by declaring the shader registers and compiling the shader file. Once created, Direct3D returns a shader handle, which is an integer number that is used to identify the shader.

```
// Create the shader declaration.
DWORD dwDecl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION,  D3DVSDT_FLOAT3),
    D3DVSD_REG( D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR ),
    D3DVSD_END()
};
```

Step 5: Create the shader

The shader is assembled and created next.

```
// Create the vertex shader.
TCHAR        strPath[512];                              // location of the shader
LPD3DXBUFFER pCode;                                     // assembled shader code
DXUtil_FindMediaFile( strPath, _T("VertexShader.vsh") );
D3DXAssembleShaderFromFile( strPath, 0, NULL, &pCode, NULL );  // assemble shader
m_pd3dDevice->CreateVertexShader( dwDecl, (DWORD*)pCode->GetBufferPointer(), &m_h
pCode->Release();
```

Once the shader file is located, D3DXAssembleShaderFromFile reads the shader instructions and validates the shader instructions. CreateVertexShader then takes the shader declaration and the assembled instructions and creates the shader. It returns the shader handle which is used to render the output.

Step 6: Render the output pixels

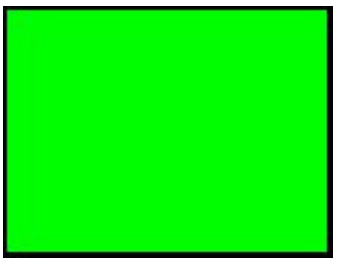
Here is an example of the code that could be used in the render loop to render the object, using the vertex shader. The render loop updates the vertex shader constants as a result of changes in the 3-D scene and draws the output vertices with a call to DrawPrimitive.

```
// Turn lighting off. This is included for clarity but is not required.
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

// Update vertex shader constants from view projection matrix data.
D3DXMATRIX mat, matView, matProj;
D3DXMatrixMultiply( &mat, &matView, &matProj );
D3DXMatrixTranspose( &mat, &mat );
m_pd3dDevice->SetVertexShaderConstant( 0, &mat, 4 );

// Declare and define the constant vertex color.
float color[4] = {0,1,0,0};
m_pd3dDevice->SetVertexShaderConstant( 4, &color, 1 );


// Render the output.
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( m_hVertexShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

Lighting is turned off just to make it clear that the vertex color is from the shader only. This statement is optional in this example.

The view and projection matrixes contain camera position and orientation data. Getting updated data and updating the shader constant registers is included in the render loop because the scene may change between rendered frames.

As usual, DrawPrimitive renders the output data using the vertex data provided from **SetStreamSource**. SetVertexShader is called to tell Direct3D to use the vertex shader. The result of the vertex shader is shown in the following image. It shows the constant color on the plane object.



Microsoft DirectX 8.1 (shader versions 1.0, 1.1)

Shader2 - Apply vertex colors

[This is preliminary documentation and is subject to change.]

This example applies the vertex color from the vertex data to the object. The vertex data contains position data as well as diffuse color data. This is reflected in the vertex declaration and the fixed vertex function macro. These are shown below.

```
struct CUSTOMVERTEX_POS_COLOR
{
    float       x, y, z;
    DWORD       diffuseColor;
```

```
};
#define D3DFVF_CUSTOMVERTEX_POS_COLOR (D3DFVF_XYZ|D3DFVF_DIFFUSE)

// Create vertex data with position and texture coordinates.
CUSTOMVERTEX_POS_COLOR g_Vertices[]=
{
    //  x      y     z     diffuse
    { -1.0f, 0.25f, 0.0f, 0xffff0000,  },       //  - bottom right - red
    {  0.0f, 0.25f, 0.0f, 0xff00ff00,  },       //  - bottom left - green
    {  0.0f, 1.25f, 0.0f, 0xff0000ff,  },       //  - top left - blue
    { -1.0f, 1.25f, 0.0f, 0xffffffff,  },       //  - top right - white
};
```

The vertex shader declaration needs to reflect the position and color data also.

```
DWORD dwDecl2[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(0, D3DVSDT_FLOAT3),      // Register 0 will contain the position d
    D3DVSD_REG(1, D3DVSDT_D3DCOLOR ),   // Register 1 will contain the color data
    D3DVSD_END()
};
```

One way for the shader to get the transformation matrix is from a constant register. This is done by calling SetVertexShaderConstant.

```
    D3DXMATRIX mat;
        D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );
    D3DXMatrixTranspose( &mat, &mat );
    hr = m_pd3dDevice->SetVertexShaderConstant( 1, &mat, 4 );
        if(FAILED(hr)) return hr;
```

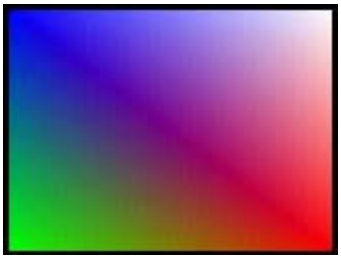
This declaration declares one stream of data, which contains the position and the color data. The color data is assigned to vertex register 7.

Lastly, here is the shader file.

```
vs.1.0              ; version instruction
m4x4 oPos, v0, c0   ; transform vertices by view/projection matrix
mov oD0, v1         ; load color from register 7 to diffuse color
```

It contains three instructions. The first is always the version instruction. The second instruction transforms the vertices. The third instruction moves the color in the vertex register to the output diffuse color register. The result is output vertices using the vertex color data.

The resulting output looks like the following:



```
// Here is an example of the class used to produce this vertex shader.
// This example is for illustration only. It has not been optimized for performan
```

```
// CVShader.h

// Use vertex color to color the object.

CUSTOMVERTEX_POS_COLOR g_VerticesVS[]=
{
    //  x      y     z     diffuse
    { -1.0f, 0.25f, 0.0f, 0xffff0000,  },       //  - bottom right - red
    {  0.0f, 0.25f, 0.0f, 0xff00ff00,  },       //  - bottom left - green
    {  0.0f, 1.25f, 0.0f, 0xff0000ff,  },       //  - top left - blue
    { -1.0f, 1.25f, 0.0f, 0xffffffff,  },       //  - top right - red
};


class CVShader
{
public:
    CVShader();

    HRESULT ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior, D3DFORMAT Format );
    HRESULT DeleteDeviceObjects();
    HRESULT Render();
    HRESULT RestoreDeviceObjects(LPDIRECT3DDEVICE8 l_pd3dDevice);
    HRESULT InitMatrices();
    HRESULT UpdateVertexShaderConstants();

private:

    LPDIRECT3DVERTEXBUFFER8     m_pQuadVB;
    LPDIRECT3DDEVICE8           m_pd3dDevice;
    DWORD                                   m_hVertexShader;
    D3DXMATRIX                          m_matView;
    D3DXMATRIX                          m_matProj;
};

CVShader::CVShader()
{
    m_pQuadVB                   = NULL;
    m_pd3dDevice                = NULL;
    m_hVertexShader             = 0;
}


HRESULT CVShader::ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior,
                                         D3DFORMAT Format )
{
    if( D3DSHADER_VERSION_MAJOR( pCaps->VertexShaderVersion ) < 1 )
        return E_FAIL;

    return S_OK;
}


HRESULT CVShader::DeleteDeviceObjects()
{
    SAFE_RELEASE(m_pQuadVB);

    HRESULT hr;
    if(m_hVertexShader > 0)
    {
        hr = m_pd3dDevice->DeleteVertexShader(m_hVertexShader);
        if(FAILED(hr))
        {
            ::MessageBox(NULL,"","DeleteVertexShader failed",MB_OK);
```

```
            return E_FAIL;
        }

        m_hVertexShader = 0;
    }

    // local to this class
    m_pd3dDevice = NULL;

    return S_OK;
}

HRESULT CVShader::InitMatrices()
{
    HRESULT hr;

    D3DXVECTOR3 from( 0.0f, 0.0f, 3.0f );
    D3DXVECTOR3 at( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 up( 0.0f, 1.0f, 0.0f );

    D3DXMATRIX matWorld;
    D3DXMatrixIdentity( &matWorld );
    hr = m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );

    D3DXMatrixIdentity( &m_matView );
    D3DXMatrixLookAtLH( &m_matView, &from, &at, &up );
    m_pd3dDevice->SetTransform( D3DTS_VIEW, &m_matView );

    D3DXMatrixIdentity( &m_matProj );
    D3DXMatrixPerspectiveFovLH( &m_matProj, D3DX_PI/4, 1.0f, 0.5f, 1000.0f );
    m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &m_matProj );

    return S_OK;
}


HRESULT CVShader::Render()
{
    if(m_pQuadVB)
    {
        HRESULT hr;
        hr = m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

        UpdateVertexShaderConstants();

        hr = m_pd3dDevice->SetStreamSource( 0, m_pQuadVB,
                    sizeof(CUSTOMVERTEX_POS_COLOR) );
        hr = m_pd3dDevice->SetVertexShader( m_hVertexShader );
        hr = m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
        }

    return S_OK;
}


HRESULT CVShader::RestoreDeviceObjects(LPDIRECT3DDEVICE8 l_pd3dDevice)
{
    HRESULT hr;

    if(l_pd3dDevice == NULL)
    {
        ::MessageBox(NULL,"","Invalid D3D8 Device ptr",MB_OK);
        return E_FAIL;
    }
```

```
    else
        m_pd3dDevice = l_pd3dDevice;

    InitMatrices();

    // Create quad Vertex Buffer.
    hr = m_pd3dDevice->CreateVertexBuffer( 4*sizeof(CUSTOMVERTEX_POS_COLOR),
                                           D3DUSAGE_WRITEONLY,
                                           D3DFVF_CUSTOMVERTEX_POS_COLOR,
                                           D3DPOOL_DEFAULT,
                                           &m_pQuadVB );
    if( FAILED(hr) ) return hr;

    // Fill the quad VB.
    CUSTOMVERTEX_POS_COLOR1* pVertices = NULL;
    hr = m_pQuadVB->Lock( 0, 4*sizeof(CUSTOMVERTEX_POS_COLOR), (BYTE**)&pVertices
    if(FAILED(hr)) return hr;

    for( DWORD i=0; i<4; i++ )
        pVertices[i] = g_VerticesVS2[i];

    m_pQuadVB->Unlock();


    // Create the vertex shader.
    TCHAR        strVertexShaderPath[512];
    LPD3DXBUFFER pCode;

    DWORD dwDecl2[] =
    {
        D3DVSD_STREAM(0),
        D3DVSD_REG(0, D3DVSDT_FLOAT3),
            D3DVSD_REG(1, D3DVSDT_D3DCOLOR ),
        D3DVSD_END()
    };

    // Find the vertex shader file.
    DXUtil_FindMediaFile( strVertexShaderPath, _T("VShader.vsh") );

    // Assemble the vertex shader from the file.
    if( FAILED( hr = D3DXAssembleShaderFromFile( strVertexShaderPath,
                                                 0, NULL, &pCode, NULL ) ) )
        return hr;

    // Create the vertex shader.
    if(SUCCEEDED(hr = m_pd3dDevice->CreateVertexShader( dwDecl2,
                                           (DWORD*)pCode->GetBufferPointer(), &m_hVe
        pCode->Release();

    return hr;
}

HRESULT CVShader::UpdateVertexShaderConstants()
{
    HRESULT hr;
    D3DXMATRIX mat;
    D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );
    D3DXMatrixTranspose( &mat, &mat );
    hr = m_pd3dDevice->SetVertexShaderConstant( 1, &mat, 4 );
    return hr;
}
```

Microsoft DirectX 8.1 (version 1.0, 1.1)

Shader3 - Apply a texture map

[This is preliminary documentation and is subject to change.]

This example applies a texture map to the object.

The vertex data contains object position data as well as texture position or uv data. This causes changes to the vertex declaration structure and the fixed vertex function macro. The vertex data is also shown below.

```
struct CUSTOMVERTEX_POS_TEX1
{
    float       x, y, z;                // object position data
    float       tu1, tv1;               // texture position data
};
#define D3DFVF_CUSTOMVERTEX_POS_TEX1 (D3DFVF_XYZ|D3DFVF_TEX1)

CUSTOMVERTEX_POS_TEX1 g_Vertices[]=
{
    //  x      y     z      u1    v1
    { -0.75f, -0.5f, 0.0f, 0.0f, 0.0f },        //  - bottom right
    {  0.25f, -0.5f, 0.0f, 1.0f, 0.0f },        //  - bottom left
    {  0.25f,  0.5f, 0.0f, 1.0f, -1.0f },       //  - top left
    { -0.75f,  0.5f, 0.0f, 0.0f, -1.0f },       //  - top right
};

D3DUtil_CreateTexture( m_pd3dDevice, _T("earth.bmp"), &m_pTexture0, D3DFMT_R5G6B5
```

The texture image must be loaded. In this case, the file "earth.bmp" contains a 2-D texture map of the earth and will be used to color the object.

The vertex shader declaration needs to reflect the object position and texture position data.

```
DWORD dwDecl2[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION,  D3DVSDT_FLOAT3),
        D3DVSD_REG(8, D3DVSDT_FLOAT2 ),
    D3DVSD_END()
};
```

This declaration declares one stream of data that contains the object position and the texture position data. The texture position data is assigned to vertex register 8.

The rendering code tells Microsoft® Direct3D® where to find the data stream and the shader, and sets up texture stages because a texture map is being applied.

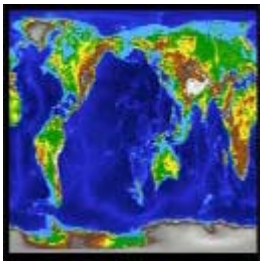
```
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX_POS_TEX1) );
m_pd3dDevice->SetVertexShader( m_hVertexShader );

m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
m_pd3dDevice->SetTexture( 0, NULL );
```

Because there is a single texture, the texture stage states need to be set for texture state 0. These methods tell Direct3D that the texel values will be used to provide diffuse color for the object vertices. In other words, a 2-D texture map will be applied like a decal.

Here is the shader file.

```
vs.1.0                          ; version instruction
m4x4 oPos, v0, c0       ; transform vertices by view/projection matrix
mov oT0, v8                     ; move texture color to output texture register
```

The shader file contains three instructions. The first is always the version instruction. The second instruction transforms the vertices. The third instruction moves the texture colors from register v8 to the output diffuse color register. That results in a texture mapped object, which is shown below.



Microsoft DirectX 8.1 (version 1.0, 1.1)

Shader4 - Apply a texture map with lighting

[This is preliminary documentation and is subject to change.]

This example uses a vertex shader to apply a texture map and add lighting to the scene. The object used is a sphere. The sample code applies a texture map of the earth to the sphere and applies diffuse lighting to simulate night and day.

The code sample adds to the Shader3 example by adding lighting to a texture mapped object. Refer to Shader3 for information about loading the texture map and setting up the texture stage states.

There is a detailed explanation of the sample code framework at Sample Framework. You can cut and paste the sample code into the sample framework to quickly get a working sample.

Create Vertex Shader

The vertex data has been modified from the Shader3 sample to include vertex normals. For lighting to appear, the object must have vertex normals. The data structure for the vertex data and the flexible vertex format (FVF) macro used to declare the data type are shown below.

```
struct CUSTOMVERTEX_POS_NORM_COLOR1_TEX1
{
    float       x, y, z;        // position
    float       nx, ny, nz;     // normal
    DWORD       color1;         // diffuse color
    float       tu1, tv1;       // texture coordinates
};
#define D3DFVF_CUSTOMVERTEX_POS_NORM_COLOR1_TEX1 (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF
```

A shader declaration defines the input vertex registers and the data associated with them. This matches the FVF macro used to create the vertex buffer data later.

```
DWORD dwDecl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(0,  D3DVSDT_FLOAT3),      // position
    D3DVSD_REG(4,  D3DVSDT_FLOAT3),      // normal
    D3DVSD_REG(7,  D3DVSDT_D3DCOLOR),    // diffuse color
    D3DVSD_REG(8,  D3DVSDT_FLOAT2),      // texture coordinate
    D3DVSD_END()
};
```

This declares one stream of data, which contains the vertex position, normal, diffuse color, and texture coordinates. The integer in each line is the register number that will contain the data. So, the texture coordinate data will be in register v8, for instance.

Next, create the shader file. You can create a shader from an ASCII text string or load it from a shader file that contains the same instructions. This example uses a shader file.

```
// Shader file
// v7  vertex diffuse color used for the light color
// v8  texture
// c4  view projection matrix
// c12 light direction
vs.1.0                      // version instruction
m4x4 oPos,    v0,    c4     // transform vertices using view projection transfor
dp3  r0     , v4   , c12    // perform lighting N dot L calculation in world spa
mul  oD0    , r0.x , v7     // calculate final pixel color from light intensity
                            // interpolated diffuse vertex color
mov  oT0.xy , v8            // copy texture coordinates to output
```

You always enter the version instruction first. The last instruction moves the texture data to the output register oT0. After you write the shader instructions, you can use them to create the shader.

```
// Now that the file exists, use it to create a shader.
TCHAR        strVertexShaderPath[512];
LPD3DXBUFFER pCode;
DXUtil_FindMediaFile( strVertexShaderPath, _T("VShader3.vsh") );
D3DXAssembleShaderFromFile( strVertexShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreateVertexShader( dwDecl, (DWORD*)pCode->GetBufferPointer(), &m_h
pCode->Release();
```

After the file is located, Direct3D creates the vertex shader and returns a shader handle and the assembled shader code. This sample uses a shader file, which is one method for creating a shader. The other method is to create an ASCII text string with the shader instructions in it. For an example, see Programmable Shaders for DirectX 8.0.

Vertex Shader Constants

You can define vertex shader constants outside of the shader file as shown in the following example. Here, you use constants to provide the shader with a view/projection matrix, a diffuse light color, RGBA, and the light direction vector.

```
float constants[4] = {0, 0.5f, 1.0f, 2.0f};
m_pd3dDevice->SetVertexShaderConstant( 0, &constants, 1 );

D3DXMATRIX mat;
D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );
```

```
D3DXMatrixTranspose( &mat, &mat );
m_pd3dDevice->SetVertexShaderConstant( 4, &mat, 4 );

float color[4] = {1,1,1,1};
m_pd3dDevice->SetVertexShaderConstant( 8, &color, 1 );

float lightDir[4] = {-1,0,1,0}; // fatter slice
m_pd3dDevice->SetVertexShaderConstant( 12, &lightDir, 1 );
```

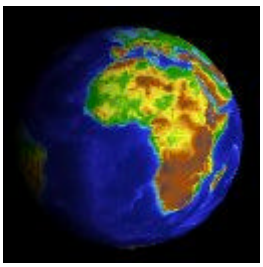
You can also define constants inside a shader using the def instruction

Render

After you write the shader instructions, connect the vertex data to the correct vertex registers and initialize the constants, you should render the output. Rendering code tells Direct3D where to find the vertex buffer data stream and provides Direct3D with the shader handle. Because you use a texture, you must set texture stages to tell Direct3D how to use the texture data.

```
// Identify the vertex buffer data source.
m_pd3dDevice->SetStreamSource(0, m_pVB, sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX1)
// Identify the shader.
m_pd3dDevice->SetVertexShader( m_hVertexShader );

// Define the texture stage(s) and set the texture(s) used
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTexture( 0, m_pTexture0 );

// Draw the object.
DWORD dwNumSphereVerts = 2 * m_dwNumSphereRings*(m_dwNumSphereSegments + 1);
m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, dwNumSphereVerts - 2);

// Set the texture stage to NULL after the render commands. Leaving this
// out will cause a memory leak.
m_pd3dDevice->SetTexture( 0, NULL );
```

The output image follows:



With the texture map applied, the sphere looks like the planet Earth. The lighting creates a bright to dark gradient on the face of the globe.

Additional Code

There is additional code required to support this example. Shown below are a few of the other methods for creating the sphere object, loading the texture, and checking for the correct version of pixel shader support.

```
// Confirm that the hardware supports version 1 shader instructions.
if( D3DSHADER_VERSION_MAJOR( pCaps->VertexShaderVersion ) < 1 )
```

```
    return E_FAIL;

// Load texture map for the sphere object.
LPDIRECT3DTEXTURE8 m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, _T("earth.bmp"), &m_pTexture0, D3DFMT_R5G6B5

// Create the sphere object.
DWORD dwNumSphereVerts = 2*m_dwNumSphereRings*(m_dwNumSphereSegments+1);
// once for the top, once for the bottom vertices

// Get the World-View(WV) matrix set.
D3DXMATRIX matWorld, matView, matWorldView;
m_pd3dDevice->GetTransform( D3DTS_WORLD, &matWorld );
m_pd3dDevice->GetTransform( D3DTS_VIEW,  &matView );
D3DXMatrixMultiply( &matWorldView, &matWorld, &matView );
m_pd3dDevice->CreateVertexBuffer(
    dwNumSphereVerts*sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX1),
    D3DUSAGE_WRITEONLY,
    D3DFVF_CUSTOMVERTEX_POS_NORM_COLOR1_TEX1,
    D3DPOOL_DEFAULT,
    &m_pVB)     );


CUSTOMVERTEX_POS_NORM_COLOR1_TEX1* pVertices;
HRESULT hr;
hr = m_pVB->Lock(0, dwNumSphereVerts*sizeof(pVertices),
                                    (BYTE**)&pVertices, 0);
if(SUCCEEDED(hr))
{
    FLOAT fDeltaRingAngle = ( D3DX_PI / m_dwNumSphereRings );
    FLOAT fDeltaSegAngle  = ( 2.0f * D3DX_PI / m_dwNumSphereSegments );

    // Generate the group of rings for the sphere.
    for( DWORD ring = 0; ring < m_dwNumSphereRings; ring++ )
    {
        FLOAT r0 = sinf( (ring+0) * fDeltaRingAngle );
        FLOAT r1 = sinf( (ring+1) * fDeltaRingAngle );
        FLOAT y0 = cosf( (ring+0) * fDeltaRingAngle );
        FLOAT y1 = cosf( (ring+1) * fDeltaRingAngle );

        // Generate the group of segments for the current ring.
        for( DWORD seg = 0; seg < (m_dwNumSphereSegments+1); seg++ )
        {
            FLOAT x0 =  r0 * sinf( seg * fDeltaSegAngle );
            FLOAT z0 =  r0 * cosf( seg * fDeltaSegAngle );
            FLOAT x1 =  r1 * sinf( seg * fDeltaSegAngle );
            FLOAT z1 =  r1 * cosf( seg * fDeltaSegAngle );

            // Add two vertices to the strip, which makes up the sphere
            // (using the transformed normal to generate texture coords).
            pVertices->x = x0;
            pVertices->y = y0;
            pVertices->z = z0;
            pVertices->nx = x0;
            pVertices->ny = y0;
            pVertices->nz = z0;
            pVertices->color = HIGH_WHITE_COLOR;
            pVertices->tu = -((FLOAT)seg)/m_dwNumSphereSegments;
            pVertices->tv = (ring+0)/(FLOAT)m_dwNumSphereRings;
            pVertices++;

            pVertices->x = x1;
            pVertices->y = y1;
            pVertices->z = z1;
```

```
            pVertices->nx = x1;
            pVertices->ny = y1;
            pVertices->nz = z1;
            pVertices->color = HIGH_WHITE_COLOR;
            pVertices->tu = -((FLOAT)seg)/m_dwNumSphereSegments;
            pVertices->tv = (ring+1)/(FLOAT)m_dwNumSphereRings;
            pVertices++;
        }

        hr = m_pVB->Unlock();
    }
}
```

Microsoft DirectX 8.1 (C++)

Pixel Shaders

[This is preliminary documentation and is subject to change.]

Before Microsoft® DirectX® 8.0, Microsoft® Direct3D® used a fixed function pipeline for converting three-dimensional (3-D) geometry to rendered screen pixels. The user sets attributes of the pipeline that control how Direct3D transforms, lights, and renders pixels. The fixed function vertex format is declared at compile time and determines the input vertex format. Once defined, the user has little control over pipeline changes during run time.

Programmable shaders add a new dimension to the graphics pipeline by allowing the vertex transform, lighting, and individual pixel coloring functionality to be modified at run time. Pixel shaders are short programs that execute for each pixel when triangles are rasterized. This gives the user a new level of dynamic flexibility over the way that pixels are rendered.

A pixel shader contains pixel shader instructions made up of ASCII text. Arithmetic instructions can be used to apply diffuse and/or specular color. Texture addressing instructions provide a variety of operations for reading and applying texture data. Functionality is available for masking and swapping color components. The shader ASCII text looks similar to assembly language and is assembled using Direct3DX assembler functions from either a text string or a file. The assembler output is a series of opcodes that an application may provide to Direct3D by means of IDirect3DDevice8::CreatePixelShader. The Pixel Shader Reference has a complete listing of shader instructions.

To understand more about pixel shaders, see the following sections.

- Create a Pixel Shader contains a code sample that uses a pixel shader to apply Gouraud interpolated diffuse colors to an object. This example contains a detailed explanation of the methods used.
- Texture Considerations details the texture stage states that are ignored during pixel shaders.
- Confirming Pixel Shader Support gives a more detailed explanation of the structures for enumerating pixel shader support.
- Pixel Shader Examples shows more code samples that add textures and blend vertex colors and textures.
- Converting Texture Operations gives examples of converting texture operations to pixel shader instructions.
- Debugging provides debugging information.

Microsoft DirectX 8.1 (C++)

Create a Pixel Shader

[This is preliminary documentation and is subject to change.]

This example uses a pixel shader to apply a Gouraud interpolated diffuse color to a geometric plane. The example will show the contents of the shader file as well as the code required in the application to set up the Microsoft® Direct3D® pipeline for the shader data.

To create a pixel shader

1. Check for pixel shader support.
2. Declare the vertex data.
3. Design the pixel shader.
4. Create the pixel shader.
5. Render the output pixels.

If you already know how to build and run Direct3D samples, you should be able to cut and paste code from this example into your existing application.

Step 1: Check for pixel shader support

To check for pixel shader support, use the following code. This example checks for pixel shader versions 1.1.

```
D3DCAPS8 caps;
m_pd3dDevice->GetDeviceCaps(&caps);        // init m_pd3dDevice before using
if( D3DDPS_VERSION(1,1) != caps.PixelShaderVersion )
        return E_FAIL;
```

The caps structure returns the functional capabilities of the pipeline. Use the macro D3DDPS_VERSION to test for the supported shader version number. If the version number is less than 1.1, this call will fail. If the hardware does not support the shader version that is tested, the application will have to fall back to another rendering approach (perhaps a lower shader version is available).

Step 2: Declare the vertex data

This example uses a plane, which is made up of two triangles. The data structure for each vertex will contain position and diffuse color data. The D3DFVF_CUSTOMVERTEX macro defines a data structure to match the vertex data. The actual vertex data is declared in a global array of vertices called g_Vertices[]. The four vertices are centered about the origin, and each vertex is given a different diffuse color.

```
// Declare vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD diffuseColor;
};
// Declare custom FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
```

```
// Declare the vertex position and diffuse color data.
CUSTOMVERTEX g_Vertices[]=
{
//      x      y      z   diffuse color
    { -1.0f, -1.0f, 0.0f, 0xffff0000 },  // red   - bottom left
    { +1.0f, -1.0f, 0.0f, 0xff00ff00 },  // green - bottom right
    { +1.0f, +1.0f, 0.0f, 0xff0000ff },  // blue  - top right
    { -1.0f, +1.0f, 0.0f, 0xffffffff },  // white - top left
};
```

Step 3: Design the pixel shader

This shader moves the Gouraud interpolated diffuse color data to the output pixels. The shader file PixelShader.txt follows:

```
ps.1.0        // version instruction
mov r0,v0     // Move the diffuse vertex color to the output register.
```

The first instruction in a pixel shader file declares the pixel shader version, which is 1.0.

The second instruction moves the contents of the color register (v0) into the output register (r0). The color register contains the vertex diffuse color because the vertex data is declared to contain the interpolated diffuse color in step 1. The output register determines the pixel color used by the render target (because there is no additional processing, such as fog, in this case).

Step 4: Create the pixel shader

The pixel shader is created from the pixel shader instructions. In this example, the instructions are contained in a separate file. The instructions could also be used in a text string.

```
TCHAR        strPath[512];           // used to locate the shader file
LPD3DXBUFFER pCode;                  // points to the assembled shader code
DXUtil_FindMediaFile( strPath, _T("PixelShader.txt") );
```

This function is a helper function used by the Sample Framework. The sample framework is the foundation on which many of the samples are built.

```
D3DXAssembleShaderFromFile( strPath, 0, NULL, &pCode, NULL );    // assemble shad
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(), &m_hPixelShad
```

Once the shader is created, the handle m_hPixelShader is used to refer to it.

Step 5: Render the output pixels

Rendering the output pixels is very similar to using the fixed function pipeline sequence of calls except that the pixel shader handle is now used to set the shader.

```
// Turn lighting off for this example. It will not contribute to the final pixel
// The pixel color will be determined solely by interpolating the vertex colors.
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_hPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

The source of the vertex data is set with SetStreamSource. In this example, SetVertexShader uses the

same Flexible Vertex Format (FVF) macro used during vertex data declaration to tell Direct3D to do fixed function vertex processing. Vertex shaders and pixel shaders may be used together or separately. The fixed function pipeline can also be used instead of either pixel or vertex shaders. SetPixelShader tells Direct3D which pixel shader to use, DrawPrimitive tells Direct3D how to draw the plane.

The gouraud shaded pixels are shown in the following image.

Pixel Shader Examples contains examples that show how to apply texture maps and blend between textures and vertex colors.

Microsoft DirectX 8.1 (C++)

Texture Considerations

[This is preliminary documentation and is subject to change.]

The pixel shader completely replaces the pixel-blending functionality specified by the Microsoft® DirectX® 6.0 and 7.0 multi-texturing application programming interfaces (APIs), specifically those operations defined by the D3DTSS_COLOROP, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAOP, D3DTSS_ALPHAARG1, and D3DTSS_ALPHAARG2 texture stage states, and associated arguments and modifiers. When a procedural pixel shader is set, these states are ignored.

Pixel Shaders and Texture Stage States

When pixel shaders are in operation, the following texture stage states are still observed.

- D3DTSS_ADDRESSU
- D3DTSS_ADDRESSV
- D3DTSS_ADDRESSW
- D3DTSS_BUMPENVMAT00
- D3DTSS_BUMPENVMAT01
- D3DTSS_BUMPENVMAT10
- D3DTSS_BUMPENVMAT11
- D3DTSS_BORDERCOLOR
- D3DTSS_MAGFILTER
- D3DTSS_MINFILTER
- D3DTSS_MIPFILTER
- D3DTSS_MIPMAPLODBIAS
- D3DTSS_MAXMIPLEVEL
- D3DTSS_MAXANISOTROPY
- D3DTSS_BUMPENVLSCALE

- D3DTSS_BUMPENVLOFFSET
- D3DTSS_TEXCOORDINDEX
- D3DTSS_TEXTURETRANSFORMFLAGS

Because these texture stage states are not part of the pixel shader, they are not available at shader compile time so the driver can make no assumptions about them. For example, the driver cannot differentiate between bilinear and trilinear filtering at that time. The application is free to change these states without requiring the regeneration of the currently bound shader.

Pixel Shaders and Texture Sampling

Texture sampling and filtering operations are controlled by the standard texture stage states for minification, magnification, mip filtering, and the wrap addressing modes. For more information, see Texture Stage States. This information is not available to the driver at shader compile time, so shaders must be able to continue operation when this state changes. The application is responsible for setting textures of the correct type (image map, cube map, volume map, etc.) needed by the pixel shader. Setting a texture of the incorrect type will produce unexpected results.

Post-Shader Pixel Processing

Other pixel operations—such as fog blending, stencil operations, and render target blending—occur after execution of the shader. Render target blending syntax is updated to support new features as described in this topic.

Pixel Shader Inputs

Diffuse and specular colors are saturated (clamped) to the range 0 through 1 before use by the shader because this is the range of valid inputs to the shader.

Color values input to the pixel shader are assumed to be perspective correct, but this is not guaranteed in all hardware. Colors generated from texture coordinates by the address shader are always iterated in a perspective correct manner. However, they are also clamped to the range 0 to 1 during iteration.

Pixel Shader Outputs

The result emitted by the pixel shader is the contents of register r0. Whatever it contains when the shader completes processing is sent to the fog stage and render target blender.

Microsoft DirectX 8.1 (C++)

Confirming Pixel Shader Support

[This is preliminary documentation and is subject to change.]

You can query members of **D3DCAPS8** to determine the level of support for operations involving pixel shaders. The following table lists the device capabilities related to programmable pixel processing in Microsoft® DirectX® 8.1.

Device capability	Description

MaxPixelShaderValue	Range of values that may be stored in registers is [-MaxPixelShaderValue, MaxPixelShaderValue].
MaxSimultaneousTextures	Number of texture stages that can be used in the programmable pixel shader.
PixelShaderVersion	Level of support for pixel shaders.

The **PixelShaderVersion** capability indicates the level of pixel shader supported. Only pixel shaders with version numbers equal to or less than this value will be successfully created. The major version number is encoded in the second byte of **PixelShaderVersion**. The low byte contains a minor version number. The pixel shader version is indicated by the first token in each shader.

Each implementation sets the **PixelShaderVersion** member to indicate the maximum pixel shader version that it can fully support. This implies that implementations should never fail the creation of a valid shader of the version less than or equal to the version reported by **PixelShaderVersion**.

Setting Pixel Shader Texture Inputs

The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. The default association is a one-to-one mapping between texture stage number and texture coordinate declaration order. This means that the first set of texture coordinates defined in the vertex format are, by default, associated with texture stage 0.

Texture coordinates can be associated with any stage, using either of the following two techniques. When using a fixed function vertex shader, the texture stage state flag TSS_TEXCOORDINDEX can be used in IDirect3DDevice8::SetTextureStageState to associate coordinates with a stage. Otherwise, the texture coordinates are output by the vertex shader oTn registers when using a programmable vertex shader.

Setting the Pixel Shader Constant Registers

You can use the following methods to set and retrieve the values in the pixel shader constant registers.

- **IDirect3DDevice8::GetPixelShaderConstant**
- **IDirect3DDevice8::SetPixelShaderConstant**

In addition, you can use the **def** instruction to set the constant registers of a pixel shader, inside a pixel shader. This instruction must come before all other instructions except the version instruction.

Compiling and Creating a Pixel Shader

The Direct3DX utility library provides a set of functions to compile pixel shaders. The following functions are provided.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFile**

The **IDirect3DDevice8::CreatePixelShader** create a pixel shader in DirectX 8.1 from a compiled shader declaration. The compiled shader declaration is obtained from **D3DXAssembleShader**.

A given shader might fail creation because of the restraints of the DirectX 8.1 hardware model.

Microsoft DirectX 8.1 (C++)

Pixel Shader Examples

[This is preliminary documentation and is subject to change.]

The topic Create a Pixel Shader provides an example of how to use a pixel shader to apply a single diffuse color. The following are examples of other pixel shader functions. Each example builds on the previous example by adding a piece of additional pixel shader functionality.

- Apply a Texture Map
- Blend a Diffuse Vertex Color with a Texture
- Blend Two Textures Using a Constant Color

Apply a Texture Map

This example applies a texture map to a plane. The differences between this example and the previous example are as follows:

- The vertex data structure and the Flexible Vertex Format (FVF) macro include texture coordinates. The vertex data includes u,v data. The vertex data no longer needs diffuse color because the pixel colors will come from the texture map.
- The texture is linked to texture stage 0 with SetTexture. Because the previous example did not use a texture, there was no SetTexture method used.
- The shader uses the t0 texture register instead of the v0 diffuse color register.

The sample code follows:

```
// Define vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    FLOAT u1, v1;
};

// Define corresponding FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_TEX|D3DFVF_TEXCOORDSIZE2(0))

// Create vertex data with position and texture coordinates.
static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     u1    v1
    { -1.0f, -1.0f, 0.0f,  0, 1, },
    {  1.0f, -1.0f, 0.0f,  1, 1, },
    {  1.0f,  1.0f, 0.0f,  1, 0, },
    { -1.0f,  1.0f, 0.0f,  0, 0, },
    // v1 is flipped to meet the top down convention in Windows
    // the upper left texture coordinate is (0,0)
    // the lower right texture coordinate is (1,1).
};

// Create a texture. This file is in the DirectX 8.1 media from the SDK down
TCHAR  strPath[512];
```

```
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
LPDIRECT3DTEXTURE8      m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m_pTexture0, D3DFMT_R5G6B5 );

// Create the pixel shader.
DXUtil_FindMediaFile( strPShaderPath, _T("PixelShader2.txt") );
```

This function is a helper function used by the Sample Framework. The sample framework is the foundation on which many of the samples are built.

```
D3DXAssembleShaderFromFile( strPShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),&m_hPixel

// Load the texture and render the output pixels.
m_pd3dDevice->SetTexture( 0, m_pTexture0 );             // load TSS0 from th
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

Contents of the file "PixelShader2.txt"
// Applies a texture map to object vertices.
ps.1.0                 // Version instruction must be first in the file.
tex t0                 // Declare texture register t0, which will be loaded
mov r0, t0             // Move the contents of the texture register (t0) to
```

The resulting image is shown in the following example.

Blend a Diffuse Vertex Color with a Texture

This example blends or modulates the colors in a texture map with the vertex colors. The differences between this example and the previous example are as follows:

- The vertex data structure, the FVF macro, and the vertex data include diffuse color.
- The shader file uses the multiply instruction (mul) to blend or modulate the texture colors with the vertex diffuse color.

The texture create and load code is the same. It is included here for completeness.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color1;
    FLOAT tu1, tv1;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1|D3DFVF_TE

static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     diffuse     u1    v1
```

```
    { -1.0f, -1.0f, 0.0f, 0xffff0000, 0, 1, }, // red
    {  1.0f, -1.0f, 0.0f, 0xff00ff00, 1, 1, }, // green
    {  1.0f,  1.0f, 0.0f, 0xff0000ff, 1, 0, }, // blue
    { -1.0f,  1.0f, 0.0f, 0xffffffff, 0, 0, }, // white
    // v1 is flipped to meet the top down convention in Windows
    // the upper left texture coordinate is (0,0)
    // the lower right texture coordinate is (1,1).
};

// Create a texture. This file is in the DirectX 8.1 media from the SDK down
TCHAR  strPath[512];
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
LPDIRECT3DTEXTURE8      m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m_pTexture0, D3DFMT_R5G6B5 );

// Create the pixel shader.
DXUtil_FindMediaFile( strPShaderPath, _T("PixelShader3.txt") );
D3DXAssembleShaderFromFile( strPShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),&m_hPixel

// Load the texture and render the output pixels.
m_pd3dDevice->SetTexture( 0, m_pTexture0 );             // load TSS0 from th
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

Contents of the file "PixelShader3.txt"
ps.1.0          // version instruction
tex t0          // declare texture register t0 which will be loaded from Tex
mul r0, v0, t0  // v0*t0, then move to r0
```

The inputs to the shader are shown in the following example. The first image shows the vertex colors. The second image shows the texture map.

The resulting image is shown in the following example. It shows the output, which is a blend of the vertex color and the texture image.

Blend Two Textures Using a Constant Color

This example blends two texture maps, using the vertex color, to determine how much of each texture map color to use. The differences between this example and the previous example are

as follows:

- The vertex data structure, the FVF macro, and the vertex data include a second set of texture coordinates because there is a second texture. SetTexture is also called twice, using two texture stage states.
- The shader file declares two texture registers and uses the linear interpolate (lrp) instruction to blend the two textures. The values of the diffuse colors determine the ratio of texture0 to texture1 in the output color.

Here is the sample code.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
        DWORD color;
    FLOAT tu1, tv1;
    FLOAT tu2, tv2;             // a second set of texture coordinates
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3D_FVF_DIFFUSE|D3DFVF_TEX2|D3DFVF_T

static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     color       u1    v1    u2    v2
    { -1.0f, -1.0f, 0.0f, 0xff0000ff, 1.0f, 1.0f, 1.0f, 1.0f },
    { +1.0f, -1.0f, 0.0f, 0xffff0000, 0.0f, 1.0f, 0.0f, 1.0f },
    { +1.0f, +1.0f, 0.0f, 0xffffff00, 0.0f, 0.0f, 0.0f, 0.0f },
    { -1.0f, +1.0f, 0.0f, 0xffffffff, 1.0f, 0.0f, 1.0f, 0.0f },
};

// Create a texture. This file is in the DirectX 8.1 media from the SDK down
TCHAR  strPath[512];
LPDIRECT3DTEXTURE8      m_pTexture0, m_pTexture1;
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m_pTexture0, D3DFMT_R5G6B5 );
DXUtil_FindMediaFile( strPath, _T("snow2.jpg"));
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m_pTexture1, D3DFMT_R5G6B5 );

// Load the textures stages.
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->SetTexture( 1, m_pTexture1 );        // Use a second texture s

m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_hPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

Contents of the file "PixelShader5.txt"
ps.1.0              // pixel shader version
tex t0              // texture register t0 is loaded from texture stage 0
tex t1              // texture register t1 is loaded from texture stage 1
mov r1, t1          // move texture register1 into output register r1
lrp r0, v0, t0, r1  // linearly interpolate between t0 and r1 by a proportio
                    // specified in v0
```

The resulting output is as follows:

Microsoft DirectX 8.1 (C++)

Converting Texture Operations

[This is preliminary documentation and is subject to change.]

Pixel shaders extend and generalize the multi-texture capabilities of Microsoft® DirectX® 6.0 and 7.0 in the following ways.

- A set of general read/write registers is added to enable more flexible expression. The serial cascade using D3DTA_CURRENT requires the specification of a separate result register argument for each stage.
- The D3DTOP_MODULATE2X and D3DTOP_MODULATE4X texture operations are broken into separate modifiers applicable to any instruction orthogonally. This eliminates the need for separate D3DTOP_MODULATE and D3DTOP_MODULATE2X operations.
- The bias and unbias texture operation modifiers are orthogonal. This eliminates the need for separate add and add bias operations.
- An optional third argument is added to modulate add, so the procedural pixel shader can do arg1×arg2 + arg0. This eliminates the D3DTOP_MODULATEALPHA_ADDCOLOR and D3DTOP_MODULATECOLOR_ADDALPHA texture operations.
- An optional third argument is added to the blend operation, so the procedural pixel shader can use arg0 as the blend proportion between arg1 and arg2. This eliminates the D3DTOP_BLENDDIFFUSEALPHA, D3DTOP_BLENDTEXTUREALPHA, D3DTOP_BLENDFACTORALPHA, D3DTOP_BLENDTEXTUREALPHAPM, and D3DTOP_BLENDCURRENTALPHA texture operations.
- Texture address modifying operations, such as D3DTOP_BUMPENVMAP, are broken out from the color and alpha operations and defined as a third operation type, specifically for operating on texture addresses.

To support this increased flexibility efficiently, the application programming interface (API) syntax is changed from **DWORD** pairs to an ASCII assemble code syntax. This exposes the functionality offered by procedural pixel shaders.

Note When you use pixel shaders, specular add is not specifically controlled by a render state, and it is up to the pixel shader to implement if needed. However, fog blending is still applied by the fixed function pipeline.

Microsoft DirectX 8.1 (C++)

Debugging

[This is preliminary documentation and is subject to change.]

MFC Pixel Shader Sample Application

You can use the MFCPixelShader Sample application to learn pixel shader instructions interactively. Programmed into this application are diffuse vertex colors and two texture images. The application has five working pixel shaders that you can select by pushing the buttons in the **Shaders** box. It includes a view window on the left to show the rendered result, an instruction window on the right to allow users to enter instructions for validation, and a third window to view debug output.

As an example, run the application and type the following instructions in the instruction window.

```
ps.1.0
tex t0
mov r0, t0
```

This results in the Microsoft® DirectX® 5 logo image in the rendered view window.

This shader applies a texture map. Notice that the compilation result text window says **Success**, which indicates that all the instructions are valid.

Next, remove the second instruction, which is the texture declaration. Once this is deleted, the compilation result says:

(Statement 2) (Validation Error) Read of uninitialized components(*) in t0: *R/X/0 *G/Y/1 *B/Z/2 *A/W/3

This error identifies the statement that fails, *Statement 2*, and why it fails *Uninitialized component in t0*. You can fix this problem by adding Statement 2 again. When you do this, the shader works again.

This is a simple example but it illustrates the usefulness of the tool. By trying different instructions, registers, and instruction sequences, you can better understand pixel shaders and vertex shaders. The sample application also has an **Open** button, which supports loading of a shader file so that you can load any shader files you have already created.

Shader Debuggers

Some graphics chip companies provide a shader debugging tool on their Web sites. Find these tools by searching the Web or by reading the article listed below. You can attach a debugger to a program while it is running and use the debugger to step through a shader. By setting breakpoints, you can step through the shader code one line at a time and watch register state changes. For more information about vertex shaders and debugging tips, see Using Vertex Shaders: Part 1.

Texture Blending Debugging

Another sample application that is part of the software development kit (SDK) installation is MFCTex. This Microsoft Foundation Classes (MFC) application is a good way to learn how to perform multi-texture blending operations in the fixed function pipeline.

Diagnostic Support

Another option for help with debugging DirectX problems is to use the DirectX Diagnostic Viewer (DXDiag.exe) to create a dump of your machine. This is done by running DxDiag.exe after your machine has crashed and sending the dump to Microsoft, using either the **Report** button on the **More Help** tab or by sending it to directx@microsoft.com. The dump can be used to track down and reproduce the problem.

Additional debug information can be found at http://msdn.microsoft.com/directx

Microsoft DirectX 8.1 (C++)

Effects

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® provides a rich feature set for creating complex and visually realistic three-dimensional (3-D) scenes. Effect files help you write an application that fully uses the rendering capabilities for the hardware on which it runs. Effects are a collection of different rendering techniques that can fit onto a variety of hardware devices.

For example, if you want to create a realistic rippled pond of water that reflects light as shown in the following image, you begin with the first technique that renders the water, adds specular highlights, adds caustic textures and apply light to the water in a single pass. If your hardware cannot render this technique in a single pass, a second technique might render the water, add specular highlights or caustic textures, but not apply light to the water.

Before you use a technique you can validate it using Direct3D to see if it is supported by the current hardware configuration.

Effects are defined in an effect file. An effect file consists of one or more techniques. Each technique consists of one or more passes. These files are text based and can be changed without recompiling the source application. This enables you to program games that make optimum use of video card functionality. Effect files also make it easy to upgrade an existing game to run on newer video cards as additional features are developed.

The following topics discuss effects techniques and how you can use them in your application.

- Create an Effect
- Multiple Techniques
- Effect File Format

Microsoft DirectX 8.1 (C++)

Create an Effect

[This is preliminary documentation and is subject to change.]

This example uses an effect file to apply a texture map to an object. The example shows the contents of the effect file as well as the code required in the application to load and run the file.

To create an effect

Step 1: Create the effect file
Step 2: Load the effect file
Step 3: Render the effect

Step 1: Create the effect file

```
/*
 * Step 1: Create the effect file.
 * This effect file maps a 3-D texture map onto the object.
 * This code needs to be in a file named effects.fx.
 */

Texture DiffuseTexture;

Technique T0
{
    Pass P0
    {
        Texture[0]   = NULL;
        PixelShader  = NULL;
        VertexShader = XYZ | Normal | Diffuse | Tex1;

        Lighting     = False;
        CullMode     = None;

        Texture[0]   = <DiffuseTexture>;
        ColorOp[0]   = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1]   = Disable;
    }
}
```

Step 2: Load the effect file

```
{
    HRESULT hr;
    D3DXTECHNIQUE_DESC technique;

    if(FAILED(hr = D3DXCreateEffectFromFile(m_pd3dDevice, "effect.fx", &m_pE
        return hr;

    if(FAILED(hr = FindNextValidTechnique(NULL, &technique)))
        return hr;

    m_pEffect->SetTechnique(technique.Index);
    m_pEffect->SetTexture("DiffuseTexture", m_pTexture0);
}
```

Once the effect file is created, **ID3DXEffect::FindNextValidTechnique** returns a technique

that has been validated on the hardware.

Step 3: Render the effect

```
{
    HRESULT hr;
    UINT uPasses;

    if(FAILED(hr = m_pd3dDevice->SetStreamSource(0, m_pVB,
                            sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX1))))
        return hr;

    m_pEffect->Begin(&uPasses, 0);

    for(UINT uPass = 0; uPass < uPasses; uPass++)
    {
        m_pEffect->Pass(uPass);
        m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, dwNumSphereVerts
    }

    m_pEffect->End();

}
```

Microsoft DirectX 8.1 (C++)

Multiple Techniques

[This is preliminary documentation and is subject to change.]

An effect file defines the techniques used. The basic layout of an effect file starts with one or more declarations and then defines each technique for that effect. This sample shows a basic effect file that contains two textures and two techniques. This effect file allows a device that doesn't support single pass rendering for two textures to use multiple passes to render the textures.

```
// Declare two textures.
texture tex0;        //First texture
texture tex1;        //Second texture

// Technique 't0' will render the scene in one pass.  The color
// for each pixel is calculated to be tex0 + tex1.  Since it uses
// two textures at once, it will only work on cards which support
// multitexture

technique t0
{
    pass p0
    {
        Texture[0] = <tex0>;
        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;

        Texture[1] = <tex1>;
        ColorOp[1] = Add;
        ColorArg1[0] = Texture;
        ColorArg2[0] = Current;

        ColorOp[2] = Disable;
```

```
    }
}

// Technique 't1' renders the scene in two passes.  The first pass sets
// each pixel to the color of tex0.  The second pass adds in the color
// of tex1.  The end result should end up looking identical to the first
// technique.  However, this technique can be used on cards that do not
// support multitexture.

technique t1
{
    pass p0
    {
        AlphaBlendEnable = False;

        Texture[0] = <tex0>;

        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Disable;
    }

    pass p1
    {
        AlphaBlendEnable = True;
        SrcBlend = One;
        DestBlend = One;

        Texture[0] = <tex1>;

        ColorOp[0] = SelectArg1;
        ColorArg[1] = Texture;
        ColorOp[1] = Disable;
    }
}
```

Microsoft DirectX 8.1 (C++)

Effect File Format

[This is preliminary documentation and is subject to change.]

This section contains reference information for the effect file format. An effect file consists of a set of parameter declarations, followed by descriptions of various techniques.

- Parameter Types
- Shader Declaration Syntax
- Render States
- Constant Value Syntax
- Examples

Technique descriptions have the following syntax.

- Variable declarations go before the body of the technique.
- The body of the technique contains one or more pass descriptions.
- A pass contains one or more state assignments.
- States can be assigned to either a constant value or to a parameter.

```
{type} {id};
{type} {id} = {const};

TECHNIQUE {id}
{
    PASS {id}
    {
        {state}      = {const};
        {state}[{n}] = {const};
        {state}      = <{id}>;
        {state}[{n}] = <{id}>;
    }
}
```

Microsoft DirectX 8.1 (C++)

Parameter Types

[This is preliminary documentation and is subject to change.]

Parameter declarations have the following syntax. If a *const* value is provided as an initial value for the parameter, its type must match *type*.

```
{type} {id};
{type} {id} = {const};
```

The following table lists all of the valid parameter types that are used for effect file parameter declarations along with a sample.

Type	Sample
DWORD	`DWORD minVertices;`
FLOAT	`FLOAT fRotationAdvances;`
VECTOR	`VECTOR vecPoint;`
MATRIX	`MATRIX matIndentity;`
TEXTURE	`TEXTURE tex1;`
VERTEXSHADER	`VERTEXSHADER v1;`
PIXELSHADER	`PIXELSHADER p1;`

Microsoft DirectX 8.1 (C++)

Shader Declaration Syntax

[This is preliminary documentation and is subject to change.]

This topic discusses the proper syntax for shader declarations.

```
DECL
{
    STREAM n;
    SKIP n;
    FVF a|b|c;
```

```
    type v#;
    type v#[n];
}
```

The following table describes the type of values used for the above syntax.

Syntax	Description
a|b|c	**DWORD** expression which can consist of **DWORD**s and flexible vector format flags. To use a flexible vertex format (FVF), just strip off the D3DFVF_ prefix.
Type	One of the following types: UBYTE, SHORT, FLOAT, D3DCOLOR.
v#	A valid vertex shader input register. See the table below for the list of FVF flags and their corresponding vertex shader input register.
n	A number between 1 and 4

The following table lists the FVF codes and the corresponding vertex shader input registers.

FVF	Name	Register Number
D3DFVF_XYZ	D3DVSDE_POSITION	0
D3DFVF_XYZRHW	D3DVSDE_BLENDWEIGHT	1
D3DFVF_XYZB1 through	D3DVSDE_BLENDINDICES	2
D3DFVF_XYZB5		
D3DFVF_NORMAL	D3DVSDE_NORMAL	3
D3DFVF_PSIZE	D3DVSDE_PSIZE	4
D3DFVF_DIFFUSE	D3DVSDE_DIFFUSE	5
D3DFVF_SPECULAR	D3DVSDE_SPECULAR	6
D3DFVF_TEX0	D3DVSDE_TEXCOORD0	7
D3DFVF_TEX1	D3DVSDE_TEXCOORD1	8
D3DFVF_TEX2	D3DVSDE_TEXCOORD2	9
D3DFVF_TEX3	D3DVSDE_TEXCOORD3	10
D3DFVF_TEX4	D3DVSDE_TEXCOORD4	11
D3DFVF_TEX5	D3DVSDE_TEXCOORD5	12
D3DFVF_TEX6	D3DVSDE_TEXCOORD6	13
D3DFVF_TEX7	D3DVSDE_TEXCOORD7	14
	D3DVSDE_POSITION2	15
	D3DVSDE_NORMAL2	16

Microsoft DirectX 8.1 (C++)

Render States

[This is preliminary documentation and is subject to change.]

Device render states control the behavior of the Microsoft® Direct3D® device's rasterization module. They do this by altering the attributes of the rendering state, what type of shading is

used, fog attributes, and other rasterizer operations.

Applications written in C++ control the characteristics of the rendering state by invoking the **IDirect3DDevice8::SetRenderState** method. The **D3DRENDERSTATETYPE** enumerated type specifies all possible rendering states. Your application passes a value from the **D3DRENDERSTATETYPE** enumeration as the first parameter to the **SetRenderState** method.

Fixed function vertex processing is controlled by the **SetRenderState** method and the following device render states. Most of these controls do not have any effect when using programmed vertex shaders.

- D3DRS_SPECULARENABLE
- D3DRS_FOGSTART
- D3DRS_FOGEND
- D3DRS_FOGDENSITY
- D3DRS_RANGEFOGENABLE
- D3DRS_LIGHTING
- D3DRS_AMBIENT
- D3DRS_FOGVERTEXMODE
- D3DRS_COLORVERTEX
- D3DRS_LOCALVIEWER
- D3DRS_NORMALIZENORMALS
- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE
- D3DRS_VERTEXBLEND

In addition, the fixed-function vertex processing pipeline uses the following methods to set transforms, materials, and lights.

- **IDirect3DDevice8::SetTransform**
- **IDirect3DDevice8::SetMaterial**
- **IDirect3DDevice8::SetLight**
- **IDirect3DDevice8::LightEnable**

Note D3DRS_SPECULARENABLE controls the addition of specular color in the pixel pipeline. D3DRS_FOGSTART, D3DRS_FOGEND, and D3DRS_FOGDENSITY control the start, end, and density of pixel fog density computation.

Additional information is contained in the following topics.

- Alpha Blending State
- Alpha Testing State
- Ambient Lighting State
- Antialiasing State
- Culling State
- Depth Buffering State
- Fog State
- Lighting State
- Outline and Fill State
- Per-Vertex Color State
- Primitive Clipping State

- Shading State
- Stencil Buffer State
- Texture Wrapping State

Microsoft DirectX 8.0

Constant Value Syntax

[This is preliminary documentation and is subject to change.]

The following table list shows the proper syntax for constant values and a sample.

Type	Syntax	Example
DWORD	# 0x#	54573153 0xff12a1fa
FLOAT	#f #.f .#f #.#f	5f 5.f .4f 4.5f
VECTOR	{float} {float, float} {float, float, float} {float, float, float, float}	{ 4.5f } { 4.5f, 1.0f } { 4.5f, 1.0f, 2.0f } { 4.5f, 1.0f, 2.0f, 3.4f }
MATRIX	{ float, float, float, float, float, float, float, float, float, float, float, float, float, float, float, float }	{ 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f }

VERTEXSHADER	DECL { ... }	decl
		{
		stream 0;
		float v0[3];
		float v3[3];
		ubyte v5[4];
		float v7[2];
		}

	DECL { ... }, ASM { ... }	decl
		{
		stream 0;
		float v0[3];
		float v3[3];
		ubyte v5[4];
		float v7[2];
		}
		asm
		{
		ps.1.0
		tex t0
		mov r0, t0
		}

PIXELSHADER	ASM { ... }	asm { ps.1.0 tex t0

```
        mov r0, t0
    }
```

Microsoft DirectX 8.1 (C++)

Examples

[This is preliminary documentation and is subject to change.]

Two example binary template definitions and an example of a binary data object follow. Note that data is stored in little-endian format, which is not shown in these examples.

The closed template *RGB* is identified by the UUID {55b6d780-37ec-11d0-ab39-0020af71e433} and has three members *r*, *g*, and *b* each of type *float*.

```
TOKEN_TEMPLATE, TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_OBRACE,
TOKEN_GUID, 55b6d780, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_FLOAT, TOKEN_NAME, 1, 'r', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'g', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'b', TOKEN_SEMICOLON,
TOKEN_CBRACE
```

The closed template *Matrix4x4* is identified by the UUID {55b6d781-37ec-11d0-ab39-0020af71e433} and has one member—a two-dimensional array named *matrix* of type *float*.

```
TOKEN_TEMPLATE, TOKEN_NAME, 9, 'M', 'a', 't', 'r', 'i', 'x', '4', 'x', '4',
TOKEN_GUID, 55b6d781, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_ARRAY, TOKEN_FLOAT, TOKEN_NAME, 6, 'm', 'a', 't', 'r', 'i', 'x',
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_CBRACE
```

The binary data object that follows shows an instance of the *RGB* template defined earlier. The example object is named *blue*, and its three members *r*, *g*, and *b* have the values 0.0, 0.0 and 1.0, respectively. Note that data is stored in little-endian format, which is not shown in this example.

```
TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_NAME, 4, 'b', 'l', 'u', 'e', TOKEN_OBRAC
TOKEN_FLOAT_LIST, 3, 0.0, 0.0, 1.0, TOKEN_CBRACE
```

Microsoft DirectX 8.1 (C++)

Advanced Topics

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® provides a powerful set of tools that you can use to increase the realistic appearance of a 3-D scene. This section presents information on common special effects that can be produced with Direct3D, but the range of possible effects is not limited to those presented here. The discussion in this section is organized into the following topics.

- Antialiasing
- Bump Mapping

- Environment Mapping
- Geometry Blending
- Indexed Vertex Blending
- Matrix Stacks
- Stencil Buffer Techniques
- Vertex Tweening
- Object Geometry

Microsoft DirectX 8.1 (C++)

Antialiasing

[This is preliminary documentation and is subject to change.]

Antialiasing is a technique you can use to reduce the appearance of stair-step pixels when drawing any line that is not exactly horizontal or vertical. In three-dimensional scenes, this artifact is most noticeable on the boundaries between polygons of different colors. Antialiasing effectively blends the pixels at these boundaries to produce a more natural look to the scene.

Microsoft® Direct3D® supports two antialiasing techniques: edge antialiasing and full-surface antialiasing. Which technique is best for your application depends on your requirements for performance and visual fidelity.

- Edge Antialiasing
- Full-Scene Antialiasing

The section below shows how to use special effects to simulate or use antialiasing.

- Motion Blur

Microsoft DirectX 8.1 (C++)

Edge Antialiasing

[This is preliminary documentation and is subject to change.]

In edge antialiasing, you render a scene, then re-render the convex silhouettes of the objects to antialias with lines. The system redraws those edges, blurring them to reduce artifacts.

First, find out if the Microsoft® Direct3D® device supports antialiasing by calling the **IDirect3DDevice8::GetDeviceCaps** method. If a device supports edge antialiasing, **GetDeviceCaps** sets the D3DPRASTERCAPS_ANTIALIASEDGES capability flag in the **D3DCAPS8** structure to TRUE.

If the device supports antialiasing, set the D3DRS_EDGEANTIALIAS render state to TRUE, as shown in the code example below.

```
d3dDevice->SetRenderState( D3DRS_EDGEANTIALIAS, TRUE );
```

Now, redraw only the edges in the scene, using **IDirect3DDevice8::DrawPrimitive** and either the D3DPT_LINESTRIP or D3DPT_LINELIST primitive type. The behavior of edge antialiasing is undefined for primitives other than lines, so make sure to disable the feature by setting D3DRS_EDGEANTIALIAS to FALSE when antialiasing is complete.

Redrawing every edge in your scene can work without introducing major artifacts, but it can be computationally expensive. In addition, it can be difficult to determine which edges should be antialiased. The most important edges to redraw are those between areas of very different colors, for example, silhouette edges, or boundaries between very different materials. Antialiasing the edge between two polygons of roughly the same color has no effect, yet is still computationally expensive. For these reasons, if current hardware supports full-scene antialiasing, it is often preferred. For more information, see Full-Scene Antialiasing.

Microsoft DirectX 8.1 (C++)

Full-Scene Antialiasing

[This is preliminary documentation and is 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. Full-scene antialiasing, when supported, only affects triangles and groups of triangles; lines cannot be antialiased by using Microsoft® Direct3D® services. Full-scene antialiasing is done in Direct3D by using multisampling on each pixel. When multisampling is enabled all subsamples of a pixel are updated in one pass, but when used for other effects that involve multiple rendering passes, the application can specify that only some subsamples are to be affected by a given rendering pass. This latter approach enables simulation of motion blur, depth of field focus effects, reflection blur, and so on.

In both cases, the various samples recorded for each pixel are blended together and output to the screen. This enables the improved image quality of antialiasing or other effects.

Before creating a device with the **IDirect3D8::CreateDevice** method, you need to determine if full-scene antialiasing is supported. Do this by calling the **IDirect3D8::CheckDeviceMultiSampleType** method as shown in the code example below.

```
/*
 * The code below assumes that pD3D is a valid pointer
 * to a IDirect3D8 interface.
 */

if( SUCCEEDED(pD3D->CheckDeviceMultiSampleType( D3DADAPTER_DEFAULT,
                                                D3DDEVTYPE_HAL , D3DFMT_R8G8
                                                FALSE, D3DMULTISAMPLE_2_SAMP
    // Full-scene antialiasing is supported. Enable it here.
```

The first parameter that **CheckDeviceMultiSampleType** accepts is an ordinal number that denotes the display adapter to query. This sample uses D3DADAPTER_DEFAULT to specify the primary display adapter. The second parameter is a value from the **D3DDEVTYPE** enumerated type, specifying the device type. The third parameter specifies the format of the surface. The fourth parameter tells Direct3D whether to inquire about full-windowed multisampling (TRUE) or full-scene antialiasing (FALSE). This sample uses FALSE to tell Direct3D that it is inquiring about full-scene antialiasing. The last parameter specifies the multisampling technique that you want to test. Use a value from the **D3DMULTISAMPLE_TYPE** enumerated type. This sample tests to see if two levels of multisampling are supported.

If the device supports the level of multisampling that you want to use, the next step is to set the presentation parameters by filling in the appropriate members of the

D3DPRESENT_PARAMETERS structure to create a multisample rendering surface. After that, you can create the device. The sample code below shows how to set up a device with a multisampling render surface.

```
/*
 * The example below assumes that pD3D is a valid pointer
 * to a IDirect3D8 interface, d3dDevice is a pointer to a
 * IDirect3DDevice8 interface, and hWnd is a valid handle
 * to a window.
 */

D3DPRESENT_PARAMETER d3dPP
ZeroMemory( &d3dPP, sizeof( d3dPP ) );
d3dPP.Windowed        = FALSE
d3dPP.SwapEffect      = D3DSWAPEFFECT_DISCARD;
d3dPP.MultiSampleType = D3DMULTISAMPLE_2_SAMPLES;
pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                   D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                   &d3dpp, &d3dDevice)
```

Note To use multisampling, the *SwapEffect* member of D3DPRESENT_PARAMETER must be set to D3DSWAPEFFECT_DISCARD.

The last step is to enable multisampling antialiasing by calling the **IDirect3DDevice8::SetRenderState** method and setting the D3DRS_MULTISAMPLEANTIALIAS to TRUE. After setting this value to TRUE, any rendering that you do will have multisampling applied to it. You might want to enable and disable multisampling, depending on what you are rendering.

Microsoft DirectX 8.1 (C++)

Motion Blur

[This is preliminary documentation and is subject to change.]

You can enhance the perceived speed of an object in a 3-D scene by blurring the object and leaving a blurred trail of object images behind the object. Microsoft® Direct3D® applications accomplish this by rendering the object multiple times per frame.

Recall that Direct3D applications typically render scenes into an off-screen buffer. The contents of the buffer are displayed on the screen when the application calls the **IDirect3DDevice8::Present** method. Your Direct3D application can render the object multiple times into a scene before it displays the frame on the screen.

Programmatically, your application makes multiple calls to a **DrawPrimitive** method, repeatedly passing the same 3-D object. Before each call, the position of the object is updated slightly, producing a series of blurred object images on the target rendering surface. If the object has one or more textures, your application can enhance the motion blur effect by rendering the first image of the object with all its textures nearly transparent. Each time the object renders, the transparency of the object's texture decreases. When your application renders the object in its final position, it should render the object's textures without transparency. The exception is if you're adding motion blur to another effect that requires texture transparency. In any case, the initial image of the object in the frame should be the most transparent. The final image should be the least transparent.

After your application renders the series of object images onto the target rendering surface and renders the rest of the scene, it should call the **Present** method to display the frame on the screen.

If your application is simulating the effect of the user moving through a scene at high speed, it can add motion blur to the entire scene. In this case, your application renders the entire scene multiple times per frame. Each time the scene renders, your application must move the viewpoint slightly. If the scene is highly complex, the user may see a visible performance degradation as acceleration is increased because of the increasing number of scene renderings per frame.

Microsoft DirectX 8.1 (C++)

Bump Mapping

[This is preliminary documentation and is subject to change.]

Bump mapping is a special form of specular or diffuse environment mapping that simulates the reflections of finely tessellated objects without requiring extremely high polygon counts. The following image, based on the BumpEarth Sample, demonstrates bump mapping effects.

The globe on the left is a sphere textured with an image of the Earth's surface, with a specular environment map applied. The globe on the right is exactly the same, but also has a bump map applied. The polygon count for the second globe is unchanged from that of the first globe.

Bump mapping in Microsoft® Direct3D® can be accurately described as per-pixel texture coordinate perturbation of specular or diffuse environment maps, because you provide information about the contour of the bump map in terms of delta-values, which the system applies to the *u* and *v* texture coordinates of an environment map in the next texture stage. The delta values are encoded in the pixel format of the bump map surface. For more information, see Bump Map Pixel Formats.

The BumpEarth sample uses a height-map to store contour data. When the sample starts, it processes the pixels in the height map to compute the appropriate *u* and *v* delta values, based on the relative height of each pixel to the four neighboring pixels.

Direct3D does not natively support height-maps; they are merely a convenient format in which to store and visualize contour data. The preceding image is based on the height map used by the BumpEarth sample to produce the bump map it supplies to Direct3D. and is included here for descriptive purposes. Your application can store contour information in any format, or even generate a procedural bump map, as the BumpWaves Sample does.

This is the height map image that the BumpEarth sample uses to compute the delta values encoded in the pixel format of the bump map texture.

Bump mapping relies completely on multiple texture blending services, and requires that the device support at least two blending stages—one for the bump map, and another for an

environment map. A minimum of three texture blending stages are required to apply a base texture map, the most common case. The following figure shows the relationships between the base texture, the bump map, and the environment map in the texture blending cascade.

You must prepare the texture stages appropriately to enable bump mapping. This task is the topic of Configuring Bump Mapping Parameters.

This section provides information about performing bump mapping with Microsoft® Direct3D®. The following topics introduce bump mapping, identify and define its key concepts, and provide details about how you can use bump-mapping in your applications.

- Bump Map Pixel Formats
- Bump Mapping Formulas
- Using Bump Mapping

Microsoft DirectX 8.1 (C++)

Bump Map Pixel Formats

[This is preliminary documentation and is subject to change.]

A bump map is a IDirect3DTexture8 object that uses a specialized pixel format. Rather than storing red, green, and blue color components, each pixel in a bump map stores the delta values for *u* and *v* (D_u and D_v) and sometimes a luminance component, *L*. These values are applied by the system as described in the Bump Mapping Formulas topic.

You can specify a bump map pixel format by setting the **D3DFORMAT** enumerated type to one of the following bump map pixel formats.

Format	Description
D3DFMT_V8U8	16-bit bump-map format.
D3DFMT_L6V5U5	16-bit bump-map format with luminance.
D3DFMT_X8L8V8U8	32-bit bump-map format with luminance where 8 bits are reserved for each element.
D3DFMT_Q8W8V8U8	32-bit bump-map format.
D3DFMT_V16U16	32-bit bump-map format.
D3DFMT_W11V11U10	32-bit bump-map format.

The D_u and D_v components of a pixel are signed values that range from –1.0 to +1.0. The luminance component, when used, is an unsigned integer value that ranges from 0 to 255.

Note Before picking a bump map pixel format, you should check if the particular format is supported. For more information, see Detecting Support for Bump Mapping.

Microsoft DirectX 8.1 (C++)

Bump Mapping Formulas

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® applies the following formulas to the D_u and D_v components in each bump map pixel.

$$D_u' = D_u M_{0,0} + D_v M_{1,0}$$
$$D_v' = D_u M_{0,1} + D_v M_{1,1}$$

In these formulas, the D_u and D_v variables are taken directly from a bump map pixel and transformed by a 2×2 matrix to produce the output delta values D_u' and D_v'. The system uses the output values to modify the texture coordinates that address the environment map in the next texture stage. The coefficients of the transformation matrix are set though the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT01, and D3DTSS_BUMPENVMAT11 texture stage states.

In addition to the *u* and *v* delta values, the system can compute a luminance value that it uses to modulate the color of the environment map in the next blending stage.

$$L' = LS + O$$

In this formula, L' is the output luminance being computed. The L variable is the luminance value taken from a bump map pixel, which is multiplied by the scaling factor, S, and offset by the value in the variable O. The D3DTSS_BUMPENVLSCALE and D3DTSS_BUMPENVLOFFSET texture stage states control the values for the S and O variables in the formula. This formula is only applied when the texture blending operation for the stage that contains the bump map is set to D3DTOP_BUMPENVMAPLUMINANCE. When using D3DTOP_BUMPENVMAP, the system uses a value of 1.0 for L'.

After computing the output delta values D_u' and D_v', the system adds them to the texture coordinates in the next texture stage, and modulates the chosen color by the luminance to produce the color applied to the polygon.

Microsoft DirectX 8.1 (C++)

Using Bump Mapping

[This is preliminary documentation and is subject to change.]

Detecting Support for Bump Mapping

A Microsoft® Direct3D® device can perform bump mapping if it supports either the D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE texture blending operation. Additionally, applications should check the device capabilities to make sure the device supports an appropriate number of blending stages, usually at least three, and exposes at least one bump-mapping pixel format.

The following code example checks device capabilities to detect support for bump mapping in the current device, using the given criteria.

```
BOOL SupportsBumpMapping()
{
    D3DCAPS8 d3dCaps;

    d3dDevice->GetDeviceCaps( &d3dCaps );

    // Does this device support the two bump mapping blend operations?
    if ( 0 == d3dCaps.TextureOpCaps & ( D3DTEXOPCAPS_BUMPENVMAP |
                                        D3DTEXOPCAPS_BUMPENVMAPLUMINANCE ))
        return FALSE;

    // Does this device support up to three blending stages?
    if( d3dCaps.MaxTextureBlendStages < 3 )
        return FALSE;

    return TRUE;
}
```

Creating a Bump Map Texture

You create a bump map texture like any other texture. Your application verifies support for bump mapping and retrieves a valid pixel format, as discussed in Detecting Support for Bump Mapping.

After the surface is created, you can load each pixel with the appropriate delta values, and luminance, if the surface format includes luminance. The values for each pixel component are described in Bump Map Pixel Formats.

Configuring Bump Mapping Parameters

When your application has created a bump map and set the contents of each pixel, you can prepare for rendering by configuring bump mapping parameters. Bump mapping parameters include setting the required textures and their blending operations, as well as the transformation and luminance controls for the bump map itself.

1. Set the base texture map if used, bump map, and environment map textures into texture blending stages.
2. Set the color and alpha blending operations for each texture stage.
3. Set the bump map transformation matrix.
4. Set the luminance scale and offset values as needed.

The following code example sets three textures—the base texture map, the bump map, and a specular environment map—to the appropriate texture blending stages.

```
    // Set the three textures.
    d3dDevice->SetTexture( 0, d3dBaseTexture );
    d3dDevice->SetTexture( 1, d3dBumpMap );
    d3dDevice->SetTexture( 2, d3dEnvMap );
```

After setting the textures to their blending stages, the following code example prepares the blending operations and arguments for each stage.

```
    // Set the color operations and arguments to prepare for
    // bump mapping.

    // Stage 0: the base texture.
    d3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
    d3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
```

```
d3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
d3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
d3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );

// Stage 1: the bump map - Use luminance for this example.
d3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
d3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_BUMPENVMAPLUM
d3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// Stage 2: a specular environment map.
d3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP, D3DTOP_ADD );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Once the blending operations and arguments are set, the following code example sets the 2×2 bump mapping matrix to the identity matrix by setting the D3DTSS_BUMPENVMAT00 and D3DTSS_BUMPENVMAT11 texture stage states to 1.0. Setting the matrix to the identity causes the system to use the delta values in the bump map unmodified, but this is not a requirement.

```
// Set the bump mapping matrix.
//
// NOTE
//    These calls rely on the following inline shortcut function:
//       inline DWORD F2DW( FLOAT f ) { return *((DWORD*)&f); }
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00, F2DW(1.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01, F2DW(0.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10, F2DW(0.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT11, F2DW(1.0f) );
```

If you set the bump mapping operation to include luminance (D3DTOP_BUMPENVMAPLUMINANCE), you must set the luminance controls. The luminance controls configure how the system computes luminance before modulating the color from the texture in the next stage. For details, see Bump Mapping Formulas.

```
// Set luminance controls. This is only needed when using
// a bump map that contains luminance, and when the
// D3DTOP_BUMPENVMAPLUMINANCE texture blending operation is
// being used.
//
// NOTE
//    These calls rely on the following inline shortcut function:
//       inline DWORD F2DW( FLOAT f ) { return *((DWORD*)&f); }
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE, F2DW(0.5f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );
```

After your application configures bump mapping parameters, it can render as normal, and the rendered polygons receive bump mapping effects.

Note The preceding example shows parameters set for specular environment mapping. When performing diffuse light mapping, applications set the texture blending operation for the last stage to D3DTOP_MODULATE. For more information, see Diffuse Light Maps.

Microsoft DirectX 8.1 (C++)

Environment Mapping

[This is preliminary documentation and is subject to change.]

Environment mapping is a technique that simulates highly reflective surfaces without using ray-tracing. In practice, environment mapping applies a special texture map that contains an image of the scene surrounding an object to the object itself. The result approximates the appearance of a reflective surface, close enough to fool the eye, without incurring any of the expensive computations involved in ray-tracing.

The following image is taken from the SphereMap Sample, which, as its name implies, uses a type of environment mapping called spherical environment mapping. For details, see Spherical Environment Mapping.

The teapot in this image appears to reflect its surroundings; this is actually a texture being applied to the object. Because environment mapping uses a texture, combined with specially computed texture coordinates, it can be performed in real-time.

This section provides information about performing two common types of environment mapping with Microsoft® Direct3D®. There are many types of environment mapping in use throughout the graphics industry, but the following topics target the two most common forms: cubic environment mapping and spherical environment mapping.

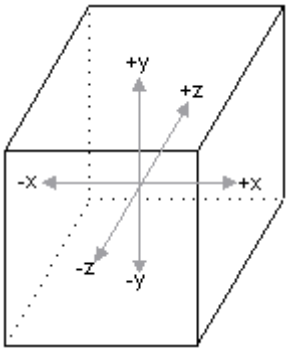
- Cubic Environment Mapping
- Spherical Environment Mapping

Microsoft DirectX 8.1 (C++)

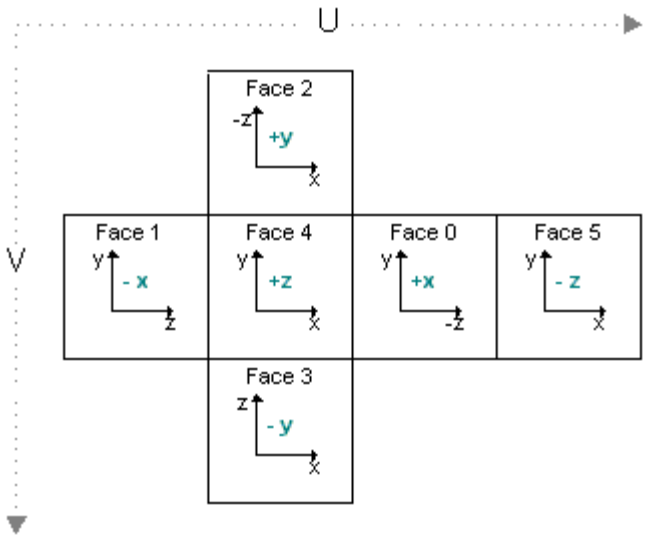
Cubic Environment Mapping

[This is preliminary documentation and is subject to change.]

Cubic environment maps—sometimes casually referred to as cube maps—are textures that contain image data representing the scene surrounding an object, as if the object were in the center of a cube. Each face of the cubic environment map covers a 90-degree field of view in the horizontal and vertical, and there are six faces per cube map. The orientation of the faces is given in the following illustration.



Each face of the cube is oriented perpendicular to the x/y, y/z, or x/z plane, in world space. The following figure shows how each plane corresponds to a face.

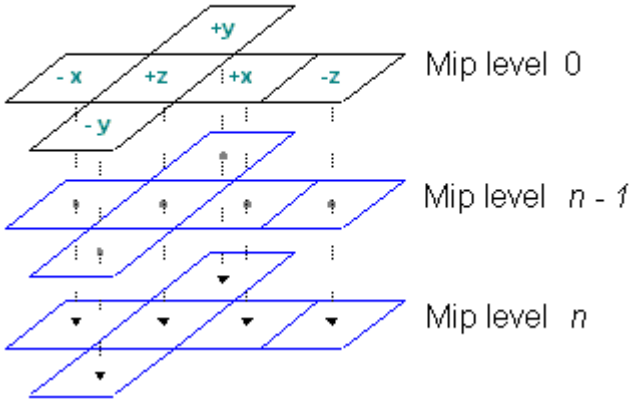


Cubic-environment maps are implemented as a series of texture objects. Applications can use static images for cubic-environment mapping, or they can render into the faces of the cube map to perform dynamic environment mapping. This technique requires that the cube-map surfaces be valid render-target surfaces, created with the D3DUSAGE_RENDERTARGET flag set.

The faces of a cube map don't need to contain extremely detailed renderings of the surrounding scene. In most cases, environment maps are applied to curved surfaces. Given the amount of curvature used by most applications, the resulting reflective distortion makes extreme detail in the environment map wasteful in terms of memory and rendering overhead.

Mipmapped Cubic Environment Maps

Cube maps can be mipmapped. To create a mipmapped cube map, set the *Levels* parameter of the IDirect3DDevice8::CreateCubeTexture method to the number of levels that you want. You can envision the topography of these surfaces as shown in the following graphic.



Applications that create mipmapped cubic-environment maps can access each face by calling the **IDirect3DCubeTexture8::GetCubeMapSurface** method. Start by setting the appropriate value from the D3DCUBEMAP_FACES enumerated type, as discussed in Accessing Cubic Environment Map Faces. Next, select the level to retrieve by setting the *Level* parameter to the mipmap level that you want. Remember that 0 corresponds with the top-level image.

Texture Coordinates for Cubic Environment Maps

Texture coordinates that index a cubic-environment map aren't simple u, v style coordinates, as used when standard textures are applied. In fact, cubic environment maps don't use texture coordinates at all. In place of a set of texture coordinates, cubic environment maps require a 3-D vector. You must take care to specify a proper vertex format. In addition to telling the system how many sets of texture coordinates your application uses, you must provide information about how many elements are in each set. Microsoft® Direct3D® offers the **D3DFVF_TEXCOORDSIZEn** set of macros for this purpose. These macros accept a single parameter, identifying the index of the texture coordinate set for which the size is being described. In the case of a 3-D vector, you include the bit pattern created by the D3DFVF_TEXCOORDSIZE3 macro. The following code example shows how this macro is used.

```
/*
 * Create a flexible vertex format descriptor for a vertex that
 * contains a position, normal, and one set of 3-D texture
 * coordinates.
 */
DWORD dwFVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE
```

In some cases, such as diffuse light mapping, you use the camera-space vertex normal for the vector. In other cases, like specular environment mapping, you use a reflection vector. Because transformed vertex normals are widely understood, the information here concentrates on computing a reflection vector.

Computing a reflection vector on your own requires understanding of the position of each vertex, and of a vector from the viewpoint to that vertex. Direct3D can automatically compute the reflection vectors for your geometry. Using this feature saves memory because you don't need to include texture coordinates for the environment map. It also reduces bandwidth and, in the case of a TnLHAL Device, it can be significantly faster than the computations that your application can make on its own. To use this feature, in the texture stage that contains the cubic-environment map, set the D3DTSS_TEXCOORDINDEX texture stage state to a combination of the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag and the index of a texture coordinate set. In some situations, like diffuse light mapping, you might use

the D3DTSS_TCI_CAMERASPACENORMAL flag, which causes the system to use the transformed, camera-space, vertex normal as the addressing vector for the texture. The index is only used by the system to determine the wrapping mode for the texture.

The following code example shows how this value is used.

```
/*
 * The m_d3dDevice variable is a valid pointer
 * to an IDirect3DDevice8 interface.
 *
 * Automatically generate texture coordinates for stage 2.
 * This assumes that stage 2 is assigned a cube map.
 * Use the wrap mode from the texture coordinate set at index 1.
 */
m_d3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX,
                                   D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR |
```

When you enable automatic texture coordinate generation, the system uses one of two formulas to compute the reflection vector for each vertex. When the D3DRS_LOCALVIEWER render state is set to TRUE, the formula used is the following:

$$R = 2(E \bullet N)N - E$$

In the preceding formula, *R* is the reflection vector being computed, *E* is the normalized position-to-eye vector, and *N* is the camera-space vertex normal.

When the D3DRS_LOCALVIEWER render state is set to FALSE, the system uses the following formula.

$$R = 2N_zN - I$$

The *R* and *N* elements in this formula are identical to the previous formula. The N_z element is the world-space *z* of the vertex normal, and *I* is the vector (0,0,1) of an infinitely distant viewpoint. The system uses the reflection vector from either formula to select and address the appropriate face of the cube map.

Note In most cases, applications should enable automatic normalization of vertex normals. To do this, set D3DRS_NORMALIZENORMALS to TRUE. If you do not enable this render state, the appearance of the environment map will be drastically different than you might expect.

Additional information is contained in the following topics.

- Creating Cubic-Environment Map Surfaces

Microsoft DirectX 8.1 (C++)

Creating Cubic-Environment Map Surfaces

[This is preliminary documentation and is subject to change.]

You create a cubic environment map texture by calling the **IDirect3DDevice8::CreateCubeTexture** method. Cubic-environment map textures must be square, with dimensions that are a power of two.

The following code example shows how your C++ application might create a simple cubic-environment map.

```
/*
 * For this example, m_d3dDevice is a valid
 * pointer to an IDirect3DDevice8 interface.
 */

LPDIRECT3DCUBETEXTURE8 m_pCubeMap;

m_d3dDevice->CreateCubeTexture( 256, 1, D3DUSAGE_RENDERTARGET, D3DFMT_R8G8B8
                                D3DPOOL_DEFAULT, &m_pCubeMap );
```

Accessing Cubic Environment Map Faces

You can navigate between faces of a cubic environment map by using the **IDirect3DCubeTexture8::GetCubeMapSurface** method.

The following code example uses **GetCubeMapSurface** to retrieve the cube-map surface used for the positive-y face (face 2).

```
/*
 * For this example, m_pCubeMap is a valid
 * pointer to a IDirect3DCubeTexture8 interface.
 */
LPDIRECT3DSURFACE8 pFace2;

m_pCubeMap->GetCubeMapSurface( D3DCUBEMAP_FACE_POSITIVE_Y, 0, &pFace2);
```

The first parameter that **GetCubeMapSurface** accepts is a **D3DCUBEMAP_FACES** enumerated value that describes the attached surface that the method should retrieve. The second parameter tells Microsoft® Direct3D® which level of a mipmapped cube texture to retrieve. The third parameter accepted is the address of the **IDirect3DSurface8** interface, representing the returned cube texture surface. Because this cube-map is not mipmapped, 0 is used here.

Note After calling this method, the internal reference count on the **IDirect3DSurface8** interface is increased. When you are done using this surface, be sure to call the **IUnknown::Release** method on this **IDirect3DSurface8** interface or you will have a memory leak.

Rendering to Cubic Environment Maps

You can copy images to the individual faces of the cube map just like you would any other texture or surface object. The most important thing to do before rendering to a face is set the transformation matrices so that the camera is positioned properly and points in the proper direction for that face: forward (+z), backward (-z), left (-x), right (+x), up (+y), or down (-y).

The following C++ code example prepares and sets a view matrix according to the face being rendered.

```
/*
 * For this example, pCubeMap is a valid pointer to a
 * IDirect3DCubeTexture8 interface and d3dDevice is a
 * valid pointer to a IDirect3DDevice8 interface.
 */
void RenderFaces()
```

```
{
    // Save transformation matrices of the device.
    D3DXMATRIX matProjSave, matViewSave;
    d3dDevice->GetTransform( D3DTS_VIEW,       &matViewSave );
    d3dDevice->GetTransform( D3DTS_PROJECTION, &matProjSave );

    // Store the current back buffer and z-buffer.
    LPDIRECT3DSURFACE8 pBackBuffer, pZBuffer;
    d3dDevice->GetRenderTarget( &pBackBuffer );
    d3dDevice->GetDepthStencilSurface( &pZBuffer );
```

Remember, each face of a cubic-environment map represents a 90-degree field of view. Unless your application requires a different field of view angle—for special effects, for example—take care to set the projection matrix accordingly.

This code example creates and sets a projection matrix for the most common case.

```
    // Use 90-degree field of view in the projection.
    D3DMATRIX matProj;
    D3DXMatrixPerspectiveFovLH( matProj, D3DX_PI/2, 1.0f, 0.5f, 1000.0f );
    d3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );

    // Loop through the six faces of the cube map.
    for( DWORD i=0; i<6; i++ )
    {
        // Standard view that will be overridden below.
        D3DVECTOR vEnvEyePt = D3DVECTOR( 0.0f, 0.0f, 0.0f );
        D3DVECTOR vLookatPt, vUpVec;

        switch( i )
        {
            case D3DCUBEMAP_FACE_POSITIVE_X:
                vLookatPt = D3DVECTOR( 1.0f, 0.0f, 0.0f );
                vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
            case D3DCUBEMAP_FACE_NEGATIVE_X:
                vLookatPt = D3DVECTOR(-1.0f, 0.0f, 0.0f );
                vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
            case D3DCUBEMAP_FACE_POSITIVE_Y:
                vLookatPt = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                vUpVec    = D3DVECTOR( 0.0f, 0.0f,-1.0f );
                break;
            case D3DCUBEMAP_FACE_NEGATIVE_Y:
                vLookatPt = D3DVECTOR( 0.0f,-1.0f, 0.0f );
                vUpVec    = D3DVECTOR( 0.0f, 0.0f, 1.0f );
                break;
            case D3DCUBEMAP_FACE_POSITIVE_Z:
                vLookatPt = D3DVECTOR( 0.0f, 0.0f, 1.0f );
                vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
            case D3DCUBEMAP_FACE_NEGATIVE_Z:
                vLookatPt = D3DVECTOR( 0.0f, 0.0f,-1.0f );
                vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
        }

        D3DMATRIX matView;
        D3DXMatrixLookAtLH( matView, vEnvEyePt, vLookatPt, vUpVec );
        d3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

Once the camera is in position and the projection matrix set, you can render the scene. Each

object in the scene should be positioned as you would normally position them. The following code example, provided for completeness, outlines this task.

```
        //Get pointer to surface in order to render to it.
        LPDIRECT3DSURFACE8 pFace;
        pCubeMap->GetCubeMapSurface( (D3DCUBEMAP_FACES)i, 0, &pFace );
        d3dDevice->SetRenderTarget ( pFace , pZBuffer );
        pFace->Release();

        d3dDevice->BeginScene();
        // Render scene here.
        d3dDevice->EndScene();
    }

    // Change the render target back to the main back buffer.
    d3dDevice->SetRenderTarget( pBackBuffer, pZBuffer );
    pBackBuffer->Release();
    pZBuffer->Release();

    // Restore the original transformation matrices.
    d3dDevice->SetTransform( D3DTS_VIEW,       &matViewSave );
    d3dDevice->SetTransform( D3DTS_PROJECTION, &matProjSave );
}
```

Note the call to the **IDirect3DDevice8::SetRenderTarget** method. When rendering to the cube map faces, you must assign the face as the current render-target surface. Applications that use depth buffers can explicitly create a depth-buffer for the render-target, or reassign an existing depth-buffer to the render-target surface. The code sample above uses the latter approach.

Microsoft DirectX 8.1 (C++)

Spherical Environment Mapping

[This is preliminary documentation and is subject to change.]

Spherical environment maps, or sphere maps, are special textures that contain an image of the scene surrounding an object, or the lighting effects around the object. Unlike cubic environment maps, sphere maps don't directly represent an object's surroundings. The teapot image in Environment Mapping topic shows the reflection effects you can achieve with sphere mapping.

A sphere map is a 2-D representation of the full 360-degree view of the scene surrounding of an object, as if taken through a fish-eye lens. The following illustration is the sphere map used by the SphereMap Sample.

Texture Coordinates for Spherical Environment Maps

The texture coordinates that you specify for each vertex receiving an environment mapping should address the texture as a function of the reflective distortion created by the curvature of the surface. Applications must compute these texture coordinates for each vertex to achieve the desired effect. One simple and effective way to generate texture coordinates uses the vertex normal as input. Although several methods exist, the following formula is common among applications that perform environment mapping with sphere maps.

$$u = N_x/2 + 0.5$$
$$v = N_y/2 + 0.5$$

In these formulas, *u* and *v* are the texture coordinates being computed, and N_x and N_y are the *x* and *y* components of the camera-space vertex normal. The formula is simple but effective. If the normal has a positive *x* component, the normal points to the right, and the *u* coordinate is adjusted to address the texture appropriately. Likewise for the *v* coordinate: positive *y* indicates that the normal points up. The opposite is true for negative values in each component.

If the normal points directly at the camera, the resulting coordinates should receive no distortion. The +0.5 bias to both coordinates places the point of zero-distortion at the center of the sphere map, and a vertex normal of (0, 0, *z*) addresses this point. This formula doesn't account for the *z* component of the normal, but applications that use the formula can optimize computations by skipping vertices with a normal that has a positive *z* element. This works for flat-shaded objects because, in camera space, if the normal points away from the camera (positive *z*), the vertex is culled when the object is rendered. For Gouraud-shaded objects, a normal can point away from the camera (positive *x*), and the triangle containing the vertex can still be visible. If you don't compute *u* and *v* for this vertex, the face might still be used, resulting in unexpected behavior.

Applying Spherical Environment Maps

You apply an environment map to objects in the same manner as for any other texture, by setting the texture to the appropriate texture stage with the **IDirect3DDevice8::SetTexture** method. Set the first parameter to the index for the desired texture stage, and set the second parameter to the address of the **IDirect3DTexture8** interface returned when you created the

texture for the environment map. You can set the color and alpha blending operations and arguments as needed to achieve the desired texture blending effects.

Microsoft DirectX 8.1 (C++)

Geometry Blending

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® enables an application to increase the realism of its scenes by rendering segmented polygonal objects—especially characters—that have smoothly blended joints. These effects are often referred to as *skinning*. The system achieves this effect by applying additional world transformation matrices to a single set of vertices to create multiple results, and then performing a linear blend between the resultant vertices to create a single set of geometry for rendering. The following image of a banana illustrates this.

This image shows how you might imagine the geometry-blending process. In a single rendering call, the system takes the vertices for the banana, transforms them twice—once without modification, and once with a simple rotation—and blends the results to create a bent banana. The system blends the vertex position, as well as the vertex normal when lighting is enabled. Applications are not limited to two blending paths; Direct3D can blend geometry between as many as four world matrices, including the standard world matrix, D3DTS_WORLD.

Note When lighting is enabled, vertex normals are transformed by a corresponding inverse world-view matrix, weighted in the same way as the vertex position computations. The system normalizes the resulting normal vector if the D3DRS_NORMALIZENORMALS render state is set to TRUE.

Without geometry blending, dynamic articulated models are often rendered in segments. For instance, consider a 3-D model of the human arm. In the simplest view, an arm has two parts: the upper arm which connects to the body, and the lower arm, which connects to the hand. The two are connected at the elbow, and the lower arm rotates at that point. An application that renders an arm might retain vertex data for the upper and lower arm, each with a separate world transformation matrix. The following code example illustrates this.

```
typedef struct _Arm {
    VERTEX upper_arm_verts[200];
    D3DMATRIX matWorld_Upper;

    VERTEX lower_arm_verts[200];
    D3DMATRIX matWorld_Lower;
} ARM, *LPARM;

ARM MyArm; // This needs to be initialized.
```

To render the arm, two rendering calls are made, as shown in the following code.

```
// Render the upper arm.
d3dDevice->SetTransform( D3DTS_WORLD, &MyArm.matWorld_Upper );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, numFaces );
```

```
// Render the lower arm, updating its world matrix to articulate
// the arm by pi/4 radians (45 degrees) at the elbow.
MyArm.matWorld_Lower = RotateMyArm(MyArm.matWorld, pi/4);
d3dDevice->SetTransform( D3DTS_WORLD, &MyArm.matWorld_Lower );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, numFaces );
```

The following image is a banana, modified to use this technique.

The differences between the blended geometry and the nonblended geometry are obvious. This example is somewhat extreme. In a real-world application, the joints of segmented models are designed so that seams are not as obvious. However, seams are visible at times, which presents constant challenges for model designers.

Geometry blending in Direct3D presents an alternative to the classic segmented-modeling scenario. However, the improved visual quality of segmented objects comes at the cost of the blending computations during rendering. To minimize the impact of these additional operations, the Direct3D geometry pipeline is optimized to blend geometry with the least possible overhead. Applications that intelligently use the geometry blending services offered by Direct3D can improve the realism of their characters while avoiding serious performance repercussions.

Blending Transform and Render States

The **IDirect3DDevice8::SetTransform** method recognizes the **D3DTS_WORLD** and **D3DTS_WORLD*n*** macros, which correspond to values that can be defined by the **D3DTS_WORLDMATRIX** macro. These macros are used to identify the matrices between which geometry will be blended.

The **D3DRENDERSTATETYPE** enumerated type includes the **D3DRS_VERTEXBLEND** render state to enable and control geometry blending. Valid values for this render state are defined by the **D3DVERTEXBLENDFLAGS** enumerated type. If geometry blending is enabled, the vertex format must include the appropriate number of blending weights.

Blending Weights

A blending weight, sometimes called a beta weight, controls the extent to which a given world matrix affects a vertex. Blending weights are floating-point values that range from 0.0 to 1.0, encoded in the vertex format, where a value of 0.0 means the vertex is not blended with that matrix, and 1.0 means that the vertex is affected in full by the matrix.

Geometry blending weights are encoded in the vertex format, appearing immediately after the position for each vertex, as described in Vertex Formats. You communicate the number of blending weights in the vertex format by including one of the D3DFVF_XYZB1 through D3DFVF_XYZB5 Flexible Vertex Format Flags in the vertex description that you provide to the Microsoft® Direct3D® rendering methods.

The system performs a linear blend between the weighted results of the blend matrices. The following is the complete blending formula.

In the preceding formula, *vBlend* is the output vertex, the v-elements are the vertices produced by the applied world matrix (**D3DTS_WORLD*n***). The W elements are the corresponding weight values within the vertex format. A vertex blended between *n* matrices can have *n–1* blending weight values, one for each blending matrix, except the last. The system automatically generates the weight for the last world matrix so that the sum of all weights is 1.0, expressed in sigma notation here. This formula can be simplified for each of the cases supported by Direct3D.

These are the simplified forms of the complete blending formula for the two, three, and four blend matrix cases.

Note Although Direct3D includes flexible vertex format descriptors to define vertices that contain up to five blending weights, only three can be used in this release of Microsoft DirectX®.

Additional information is contained in the following topic.

- Using Geometry Blending

Microsoft DirectX 8.1 (C++)

Using Geometry Blending

[This is preliminary documentation and is subject to change.]

The following user-defined structure can be used for vertices that will be blended between two matrices.

```
//
// The flexible vertex format descriptor for this vertex would be:
//
//     FVF = D3DFVF_XYZB1 | D3DFVF_NORMAL | D3DFVF_TEX1;
//
struct D3DBLENDVERTEX {
    D3DVECTOR v;
    FLOAT     blend;
    D3DVECTOR n;
    FLOAT     tu, tv;
};
```

The blend weight must appear after the position and RHW data in the flexible vertex format, and before the vertex normal.

Notice that the preceding vertex format contains only one blending weight value. This is because there will be two world matrices, defined in the **D3DTS_WORLDMATRIX**(0) and **D3DTS_WORLDMATRIX**(1) transform states. The system blends each vertex between these two matrices using the single weight value. For three matrices, only two weights are required, and so on.

Note Defining skin weights is easy. Using a linear function of the distance between joints is good start, but a smoother sigmoid function looks better. Choosing a skin-weight distribution function can result in sharp creases at joints, if desired, by using a large variation in skin weight over a short distance.

Setting Blending Matrices

You set the transformation matrices between which the system blends by calling the **IDirect3DDevice8::SetTransform** method. Set the first parameter to a value defined by the **D3DTS_WORLDMATRIX** macro, and set the second parameter to the address of the matrix to be set.

The following C++ code example sets two world matrices, between which geometry is blended to create the illusion of a jointed arm.

```
// For this example, the d3dDevice variable is assumed to be a
// valid pointer to an IDirect3DDevice8 interface for an initialized
// 3-D scene.
float     BendAngle = 3.1415926f / 4.0f; // 45 degrees
D3DMATRIX matUpperArm, matLowerArm;

// The upper arm is immobile. Use the identity matrix.
D3DXMatrixIdentity( matUpperArm );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(0), &matUpperArm );

// The lower arm rotates about the x-axis, attached to the upper arm.
D3DXMatrixRotationX( matLowerArm, -BendAngle );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(1), &matLowerArm );
```

Setting a blending matrix merely causes the system to cache the matrix for later use; it doesn't instruct the system to begin blending vertices.

Enabling Geometry Blending

Geometry blending is disabled by default. To enable geometry blending, call the **IDirect3DDevice8::SetRenderState** method to set the **D3DRS_VERTEXBLEND** render state to a value from the **D3DVERTEXBLENDFLAGS** enumerated type. The following code example shows what this call might look like when setting the render state for a blend between two world matrices.

```
d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_1WEIGHTS );
```

When D3DRS_VERTEXBLEND is set to any value other than D3DVBF_DISABLE, the system assumes that the appropriate number of blending weights will be included in the vertex format. It is your responsibility to provide a compliant vertex format, and to provide a proper description of that format to the Microsoft® Direct3D® rendering methods.

When enabled, the system performs geometry blending for all objects rendered by the DrawPrimitive rendering methods.

See Also

Vertex Formats

Microsoft DirectX 8.1 (C++)

Indexed Vertex Blending

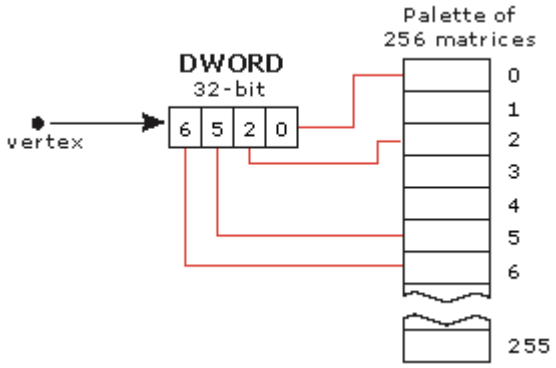
[This is preliminary documentation and is subject to change.]

Indexed vertex blending extends the vertex blending support in Microsoft® Direct3D® to allow matrices to be used for blending. These matrices are referred to by using a matrix index. These indices are supplied on a per-vertex basis and refer to a palette of up to 256 matrices. Each index is 8 bits and each vertex can have up to four indices, which allows four matrices to be blended per vertex. The indices are packed into a **DWORD**. Because indices are specified on a per-vertex basis, up to 12 matrices can affect a single triangle, and any matrix in the palette can affect the vertices of one draw call. This approach has the following advantages.

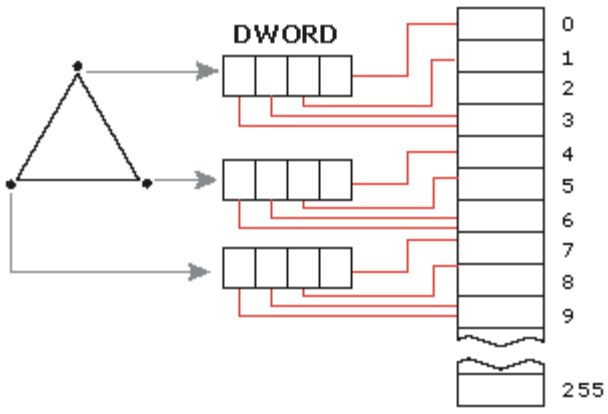
- It enables more matrices to affect a single triangle.
- It enables more blended triangles to be passed in the same draw call.
- It makes vertex blending independent of triangle indices. This enables progressive meshes to work in conjunction with vertex blending.

One disadvantage of this approach is that it does not work with curved-surface primitives when tessellation occurs before vertex processing.

The following illustration demonstrates how four matrices can affect a vertex. Each vertex has up to four indices, so four matrices can be blended per vertex. The illustration below uses the matrices indexed at 0, 2, 5, and 6.



The illustration below demonstrates how up to 12 matrices can affect a triangle. Using indices specified on a per-vertex basis, up to 12 matrices can affect the triangle.



The following formula determines the general case for how matrices effect a vertex.

$$V_{world} = V_{model} * M[index0] * b_0 + V_{model} * M[index1] * b_1 + V_{model} * M[index2] * b_2 + V_{model} * M[index3] * (1 - b_0 - b_1 - b_2)$$

V_{model} is the input model space vertex position. *Index0..Index3* are the per-vertex matrix

indices packed into a **DWORD**. *M[]* is the array of world matrices that get indexed into. $b_0..b_2$ are the blend weights. V_{world} is the output world space vertex position.

Additional information is contained in the following topic.

- Using Indexed Vertex Blending

Microsoft DirectX 8.1 (C++)

Using Indexed Vertex Blending

[This is preliminary documentation and is subject to change.]

Transform and Render States

Transform states 256-511 are reserved to store up to 256 matrices that can be indexed using 8-bit indices. Use the macro **D3DTS_WORLDMATRIX** to map indices 0-255 to the corresponding transform states. The following code example shows how to use the **IDirect3DDevice8::SetTransform** method to set the matrix at transform state number 256 to an identity matrix.

```
D3DMATRIX matBlend1;

D3DXMatrixIdentity( &matBlend1 );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(0), &matBlend );
```

To enable or disable indexed vertex blending, set the **D3DRS_INDEXEDVERTEXBLENDENABLE** render state to TRUE. When the render state is enabled ,you must pass matrix indices as packed **DWORD**s with every vertex. When this render state is disabled and vertex blending is enabled, it is equivalent to having the matrix indices 0, 1, 2, and 3 in every vertex. The code example below uses the **IDirect3DDevice8::SetRenderState** method to enable indexed vertex blending.

```
d3dDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE );
```

To enable or disable vertex blending, set the **D3DRS_VERTEXBLEND** render state to a value other than D3DRS_DISABLE from the **D3DVERTEXBLENDFLAGS** enumerated type. If this render state is not set to D3DRS_DISABLE, then you must pass the required number of weights for each vertex. The following code example uses **SetRenderState** to enable vertex blending with three weights for each vertex.

```
d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_3WEIGHTS );
```

Determining Indexed Vertex Blending Support

To determine the maximum size for the indexed vertex blending matrix, check the **MaxVertexBlendMatrixIndex** member of the **D3DCAPS8** structure. The code example below uses the **IDirect3DDevice8::GetDeviceCaps** method to retrieve this size.

```
D3DCAPS8 d3dCaps;

d3dDevice->GetDeviceCaps( &d3dCaps );
IndexedMatrixMaxSize = d3dCaps.MaxVertexBlendMatrixIndex;
```

If the value set in **MaxVertexBlendMatrix** is 0, then the device does not support indexed matrices.

Note When software vertex processing is used, 256 matrices can be used for indexed vertex blending, with or without normal blending.

Passing Matrix Indices to Direct3D

World matrix indices can be passed to Microsoft® Direct3D® by using legacy vertex shaders (FVF) or declarations.

The code example below shows how to use legacy vertex shaders.

```
struct VERTEX
{
    float x,y,z;
    float weight;
    DWORD matrixIndices;
    float normal[3];
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZB2 | D3DFVF_LASTBETA_UBYTE4 |
                             D3DFVF_NORMAL);
```

When a legacy vertex shader is used, matrix indices are passed together with vertex positions using **D3DFVF_XYZBn** flags. Matrix indices are passed as bytes inside a **DWORD** and must be present immediately after the last vertex weight. Vertex weights are also passed using **D3DFVF_XYZBn**. A packed **DWORD** contains index3, index2, index1, and index0, where index0 is located in the lowest byte of the **DWORD**. The number of used world-matrix indices is equal to the number passed to the number of matrices used for blending as defined by **D3DRS_VERTEXBLEND**.

When a declaration is used, **D3DVSDE_BLENDINDICES** defines the input vertex register to get matrix indices from. Matrix indices must be passed as **D3DVSDT_UBYTE4**.

The code example below shows how to use declarations. Note that the order of the components is no longer important unless using a fixed-function vertex shader.

```
struct VERTEX
{
    float x,y,z;
    float weight;
    DWORD matrixIndices;
    float normal[3];
}

DWORD decl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
    D3DVSD_REG(D3DVSDE_BLENDWEIGHT, D3DVSDT_FLOAT1),
    D3DVSD_REG(D3DVSDE_BLENDINDICES, D3DVSDT_UBYTE),
    D3DVSD_REG(D3DVSDE_NORMAL, D3DVSDT_FLOAT3),
    D3DVSD_END()
};
```

Matrix Stacks

[This is preliminary documentation and is subject to change.]

The Direct3DX utility library provides the **ID3DXMatrixStack** interface. It supplies a mechanism to enable matrices to be pushed onto and popped off of a matrix stack. Implementing a matrix stack is an efficient way to track matrices while traversing a transform hierarchy.

The Direct3DX utility library uses a matrix stack to store transformations as matrices. The various methods of the **ID3DXMatrixStack** interface deal with the current matrix, or the matrix located on top of the stack. You can clear the current matrix with the **ID3DXMatrixStack::LoadIdentity** method. To explicitly specify a certain matrix to load as the current transformation matrix, use the **ID3DXMatrixStack::LoadMatrix** method. Then you can call either the **ID3DXMatrixStack::MultMatrix** method or the **ID3DXMatrixStack::MultMatrixLocal** method to multiply the current matrix by the specified matrix.

The **ID3DXMatrixStack::Pop** method enables you to return to the previous transformation matrix and the **ID3DXMatrixStack::Push** method adds a transformation matrix to the stack.

The individual matrices on the matrix stack are represented as 4×4 homogeneous matrices, defined by the Direct3DX utility library **D3DXMATRIX** structure.

The Direct3DX utility library provides a matrix stack through a COM object.

Implementing a Scene Hierarchy

A matrix stack simplifies the construction of hierarchical models, in which complicated objects are constructed from a series of simpler objects.

A scene, or transform, hierarchy is usually represented by a tree data structure. Each node in the tree data structure contains a matrix. A particular matrix represents the change in coordinate systems from the node's parent to the node. For example, if you are modeling a human arm, you might implement the following hierarchy.

In this hierarchy, the Body matrix places the body in the world. The UpperArm matrix contains the rotation of the shoulder, the LowerArm matrix contains the rotation of the elbow, and the Hand matrix contains the rotation of the wrist. To determine where the hand is relative to the world, you simply multiply all the matrices from Body down through Hand together.

The previous hierarchy is overly simplistic, because each node has only one child. If you begin to model the hand in more detail, you will probably add fingers and a thumb. Each digit can be added to the hierarchy as children of Hand.

If you traverse the complete graph of the arm in depth-first order—traversing as far down one path as possible before moving on to the next path—to draw the scene, you perform a sequence of segmented rendering. For example, to render the hand and fingers, you implement

the following pattern.

1. Push the Hand matrix onto the matrix stack.
2. Draw the hand.
3. Push the Thumb matrix onto the matrix stack.
4. Draw the thumb.
5. Pop the Thumb matrix off the stack.
6. Push the Finger 1 matrix onto the matrix stack.
7. Draw the first finger.
8. Pop the Finger 1 matrix off the stack.
9. Push the Finger 2 matrix onto the matrix stack. You continue in this manner until all the fingers and thumb are rendered.

After you have completed rendering the fingers, you would pop the Hand matrix off the stack.

You can follow this basic process in code with the following examples. When you encounter a node during the depth-first search of the tree data structure, push the matrix onto the top of the matrix stack.

```
MatrixStack->Push();
MatrixStack->MultMatrix(pNode->matrix);
```

When you are finished with a node, pop the matrix off the top of the matrix stack.

```
MatrixStack->Pop();
```

In this way, the matrix on the top of the stack always represents the world-transform of the current node. Therefore, before drawing each node, you should set the Microsoft® Direct3D® matrix.

```
pD3DDevice->SetTransform(D3DTS_WORLDMATRIX(0), *MatrixStack->GetTop());
```

For more information on the specific methods that you can perform on a Direct3DX matrix stack, see the **ID3DXMatrixStack** reference topic.

Microsoft DirectX 8.1 (C++)

Stencil Buffer Techniques

[This is preliminary documentation and is subject to change.]

Applications use the stencil buffer to mask pixels in an image. The mask controls whether the pixel is drawn.

The stencil buffer enables or disables drawing to the rendering target surface on a pixel-by-pixel basis. At its most fundamental level, it enables applications to mask sections of the rendered image so that they are not displayed. Applications often use stencil buffers for special effects such as dissolves, decaling, and outlining.

Stencil buffer information is embedded in the z-buffer data. Your application can use the **IDirect3D8::CheckDeviceFormat** method to check for hardware stencil support, as shown in the following code example.

```
// Reject devices that cannot perform 8-bit stencil buffering.
```

```
// The following example assumes that pCaps is a valid pointer
// to an initialized D3DCAPS8 structure.

if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                       pCaps->DeviceType,
                                       Format,
                                       D3DUSAGE_DEPTHSTENCIL,
                                       D3DRTYPE_SURFACE,
                                       D3DFMT_D24S8 ) ) )
        return E_FAIL;
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 8-bit stencil buffers are rejected. Note that this is only one possible use for **CheckDeviceFormat**; for details see Determining Hardware Support.

To determine the stencil buffer limitations of a device, query the **StencilCaps** member of the **D3DCAPS8** structure for its supported stencil buffer operations.

How the Stencil Buffer Works

Microsoft® Direct3D® performs a test on the contents of the stencil buffer on a pixel-by-pixel basis. For each pixel in the target surface, it performs a test using the corresponding value in the stencil buffer, a stencil reference value, and a stencil mask value. If the test passes, Direct3D performs an action. The test is performed using the following steps.

1. Perform a bitwise **AND** operation of the stencil reference value with the stencil mask.
2. Perform a bitwise **AND** operation of the stencil buffer value for the current pixel with the stencil mask.
3. Compare the result of step 1 to the result of step 2, using the comparison function.

These steps are shown in the following code example.

```
(StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)
```

StencilBufferValue is the contents of the stencil buffer for the current pixel. This code example uses the ampersand (&) symbol to represent the bitwise **AND** operation. *StencilMask* represents the value of the stencil mask, and *StencilRef* represents the stencil reference value. *CompFunc* is the comparison function.

The current pixel is written to the target surface if the stencil test passes, and is ignored otherwise. The default comparison behavior is to write the pixel, no matter how each bitwise operation turns out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the **D3DRS_STENCILFUNC** render state, passing a member of the **D3DCMPFUNC** enumerated type to identify the desired comparison function.

Your application can customize the operation of the stencil buffer. It can set the comparison function, the stencil mask, and the stencil reference value. It can also control the action that Microsoft® Direct3D® takes when the stencil test passes or fails. For more information, see Stencil Buffer State.

Microsoft® Direct3D® applications can achieve a wide range of special effects with the stencil buffer. Some of the more common effects are discussed in this section.

- Dissolves, Fades, and Swipes

- Decaling
- Compositing
- Outlines and Silhouettes

Microsoft DirectX 8.1 (C++)

Dissolves, Fades, and Swipes

[This is preliminary documentation and is subject to change.]

Increasingly, applications employ special effects that are commonly used in movies and video, such as dissolves, swipes, and fades.

In a dissolve, one image is gradually replaced by another in a smooth sequence of frames. Although Microsoft® Direct3D® provides methods of using multiple texture blending to achieve the same effect, applications that use the stencil buffer for dissolves can use texture-blending capabilities for other effects while they do a dissolve.

When your application performs a dissolve, it must render two different images. It uses the stencil buffer to control which pixels from each image are drawn to the rendering target surface. You can define a series of stencil masks and copy them into the stencil buffer on successive frames. Alternately, you can define a base stencil mask for the first frame and alter it incrementally.

At the beginning of the dissolve, your application sets the stencil function and stencil mask so that most of the pixels from the starting image pass the stencil test. Most of the pixels from the ending image should fail the stencil test. On successive frames, the stencil mask is updated so that fewer and fewer of the pixels in the starting image pass the test. As the frames progress, fewer and fewer of the pixels in the ending image fail the test. In this manner, your application can perform a dissolve using any arbitrary dissolve pattern.

Fading in or fading out is a special case of dissolving. When fading in, the stencil buffer is used to dissolve from a black or white image to a rendering of a 3-D scene. Fading out is the opposite, your application starts with a rendering of a 3-D scene and dissolves to black or white. The fade can be done using any arbitrary pattern you want to employ.

Direct3D applications use a similar technique for swipes. For example, when an application performs a left-to-right swipe, the ending image appears to slide gradually on top of the starting image from left to right. As in a dissolve, you must define a series of stencil masks that are loaded into the stencil buffer on successive frames, or successively modify the starting stencil mask. The stencil masks are used to disable the writing of pixels from the starting image and to enable the writing of pixels from the ending image.

A swipe is somewhat more complex than a dissolve in that your application must read pixels from the ending image in the reverse order of the swipe. That is, if the swipe is moving from left to right, your application must read pixels from the ending image from right to left.

Microsoft DirectX 8.1 (C++)

Decaling

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® applications use decaling to control which pixels from a particular primitive image are drawn to the rendering target surface. Applications apply decals to the images of primitives to enable coplanar polygons to render correctly.

For instance, when applying tire marks and yellow lines to a roadway, the markings should appear directly on top of the road. However, the z values of the markings and the road are the same. Therefore, the depth buffer might not produce a clean separation between the two. Some pixels in the back primitive may be rendered on top of the front primitive and vice versa. The resulting image appears to shimmer from frame to frame. This effect is called *Z Fighting* or *flimmering*.

To solve this problem, use a stencil to mask the section of the back primitive where the decal will appear. Turn off z-buffering and render the image of the front primitive into the masked-off area of the render-target surface.

Although multiple texture blending can be used to solve this problem, doing so limits the number of other special effects that your application can produce. Using the stencil buffer to apply decals frees up texture blending stages for other effects.

Microsoft DirectX 8.1 (C++)

Compositing

[This is preliminary documentation and is 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 with the driver's forward view.

Microsoft DirectX 8.1 (C++)

Outlines and Silhouettes

[This is preliminary documentation and is subject to change.]

You can use the stencil buffer for more abstract effects, such as outlining and silhouetting.

If your application applies a stencil mask to the image of a primitive that is the same shape but slightly smaller, the resulting image contains only the primitive's outline. The application can then fill the stencil-masked area of the image with a solid color, giving the primitive an embossed look.

If the stencil mask is the same size and shape as the primitive you are rendering, the resulting image contains a hole where the primitive should be. Your application can then fill the hole with black to produce a silhouette of the primitive.

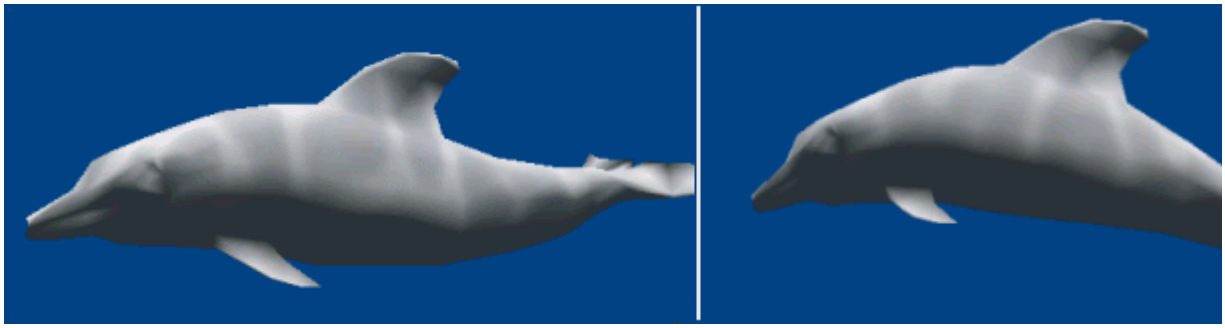
Microsoft DirectX 8.1 (C++)

Vertex Tweening

[This is preliminary documentation and is subject to change.]

Vertex tweening is used to blend two user-provided positions or normal streams. It can be used only with declarations. Tweening is enabled by setting D3DRS_VERTEXBLEND to D3DVBF_TWEENING. Vertex blending using weights and vertex blending tweening are mutually exclusive.

The C++ DolphinVS Sample uses vertex tweening to animate a dolphin so that it appears to move through the water. The images below show the dolphin at two different points. Notice that the dolphin's tail has gone from an up position to a down position and its nose points downward more in the right image.



Tweening is performed before lighting and clipping. The resulting vertex position (normal) is computed as follows:

POSITION = POSITION1 * (1.0 - TWEENFACTOR) + POSITION2 * TWEENFACTOR, and likewise for normals.

Additional information is contained in the following topic.

- Using Vertex Tweening

Vertex Shader Tweening

The code for using vertex shader tweening is similar to the code for the fixed-function tweening shown in the Using Vertex Tweening topic. The only difference is the call to **IDirect3DDevice8::CreateVertexShader** as shown below.

```
m_d3dDevice->CreateVertexShader( decl, &vsFunction, &handle, 0 );
```

The second parameter for **CreateVertexShader** takes a pointer to a vertex shader function token array. This function can be created by calling any of the following functions.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFile**

The code example below assembles a vertex shader from the file AppTween.vsh.

```
D3DXAssembleShaderFromFile( "AppTween.vsh", 0, NULL, &vsFunction, NULL
```

Microsoft DirectX 8.1 (C++)

Using Vertex Tweening

[This is preliminary documentation and is subject to change.]

Determining Support for Vertex Tweening

To determine if Microsoft® Direct3D® supports vertex tweening, check for the D3DVTXPCAPS_TWEENING flag in the **VertexProcessingCaps** member of the **D3DCAPS8** structure. The following code example uses the **IDirect3DDevice8::GetDeviceCaps** method to determine if tweening is supported.

```
//
// This example assumes that m_d3dDevice is
// a valid pointer to a IDirect3DDevice8 interface.
//

D3DCAPS8 d3dCaps;

m_d3dDevice->GetDeviceCaps( &d3dCaps );
if( 0 != (d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_TWEENING) )
    //Vertex tweening is supported.
```

Setting Vertex Declaration

To use vector tweening, you must first set up a custom vertex type that uses a second normal or a second position. The following code example shows a sample declaration that includes both a second point and a second position.

```
struct TEX_VERTEX
{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DVECTOR position2;
    D3DVECTOR normal2;
};

//Create a vertex buffer with the type TEX_VERTEX.
```

The next step is to set the current declaration. The code example below shows how to do this.

```
DWORD decl[]
{
    D3DVSD_STREAM(0),
    D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ) // Position 1
    D3DVSD_REG( D3DVSDE_NORMAL, D3DVSDT_FLOAT3 )   // Normal 1
    D3DVSD_REG( D3DVSDE_POSITION2, D3DVSDT_FLOAT3) // Position 2
    D3DVSD_REG( D3DVSDE_NORMAL2, D3DVSDT_FLOAT3 )  // Normal 2
    D3DVSD_END()
};
```

For more information on creating a custom vertex type and a vertex buffer, see Creating a Vertex Buffer.

Notes When vertex tweening is enabled, a second position or a second normal must be present in the current declaration.

Fixed-Function Tweening

The code example below shows how to implement fixed-function vertex tweening after a vertex type and declaration are set.

```
//Variables used for this example
DWORD handle;
float TweenFactor = 0.3f;
```

The first step is to use the **IDirect3DDevice8::CreateVertexShader** method to create a vertex shader, as shown in the code example below.

```
m_d3dDevice->CreateVertexShader( decl, NULL, &handle, 0 );
```

The first parameter accepted by **CreateVertexShader** is a pointer to a vector shader declaration. This example uses the declaration declared above. The second parameter accepts a pointer to a vertex shader function array. This example does not use a vertex shader function, so this parameter is set to NULL. The third parameter accepts a pointer to a vertex shader handle representing the returned vertex. The fourth parameter specifies the usage controls for the vertex shader. You can use the D3DUSAGE_SOFTWAREPROCESSING flag to indicate that the vertex shader should use software vertex processing. This example sets this parameter to zero to indicate that vertex processing should be done in hardware.

The next step is to set the current vertex shader by calling the **IDirect3DDevice8::SetVertexShader** method as shown in the code example below.

```
m_d3dDevice->SetVertexShader( handle );
```

SetVertexShader accepts a handle to a vertex shader. The value for this parameter can be a handle returned by **CreateVertexShader** or an FVF code. This example uses the handle returned from **CreateVertexShader** called in the last step.

The next step is to set the current render state to enable vertex tweening, set the tween factor, and set the stream source. The code example below uses **IDirect3DDevice8::SetRenderState** and **IDirect3DDevice8::SetStreamSource** to accomplish this.

```
m_d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_TWEENING );
m_d3dDevice->SetRenderState( D3DRS_TWEENFACTOR, *(DWORD*) &TweenFactor
m_d3dDevice->SetStreamSource( 0 ,vb, 12*sizeof(float));
```

If you have more than one stream specified for the current vertex shader, you need to call **SetStreamSource** for each stream that will be used for the tweening effect. At this point, vertex tweening is enabled. A call to any rendering function automatically has vertex tweening applied to it.

Microsoft DirectX 8.1 (C++)

Object Geometry

[This is preliminary documentation and is subject to change.]

Direct3D supports several higher leverl abstractions for handling geometry. These are broken down into the following topics.

- Index Buffers
- Vertex Buffers
- Higher-Order Primitives
- Point Sprites

Microsoft DirectX 8.1 (C++)

Index Buffers

[This is preliminary documentation and is subject to change.]

Index buffers, represented by the **IDirect3DIndexBuffer8** interface, are memory buffers that contain index data. Index data, or indices, are integer offsets into vertex buffers and are used to render primitives using the **IDirect3DDevice8::DrawIndexedPrimitive** method.

An index buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of indices it can contain. These traits are held in a **D3DINDEXBUFFER_DESC** structure.

Index buffer descriptions tell your application how an existing buffer was created. You provide an empty description structure for the system to fill with the capabilities of a previously created index buffer. For more information about this task, see Access the Contents of an Index Buffer.

The **Format** member describes the surface format of the index buffer data.

The **Type** identifies the resource type of the index buffer.

The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the index buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the index buffer memory is used only for write operations. This frees the driver to place the index data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data in the index buffer.

Pool specifies the memory class allocated for the index buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the index buffer in system memory.

The **Size** member simply stores the size, in bytes, of the vertex buffer data.

Additional information is contained in the following topics.

- Create an Index Buffer
- Access the Contents of an Index Buffer
- Render from an Index Buffer

Microsoft DirectX 8.1 (C++)

Create an Index Buffer

[This is preliminary documentation and is subject to change.]

Create an index buffer object by calling the **IDirect3DDevice8::CreateIndexBuffer** method, which accepts five parameters. The first parameter specifies the index buffer length, in bytes.

The second parameter is a set of usage controls. Among other things, its value determines whether the vertices being referred to by the indices are capable of containing clipping information. To improve performance, specify D3DUSAGE_DONOTCLIP when clipping is not required.

The D3DUSAGE_SOFTWAREPROCESSING flag can be set when mixed-mode or software vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING) is enabled for that device. D3DUSAGE_SOFTWAREPROCESSING must be set for buffers to be used with software vertex processing in mixed mode, but it should not be set for the best possible performance when using hardware index processing in mixed mode (D3DCREATE_HARDWARE_VERTEXPROCESSING). However, setting D3DUSAGE_SOFTWAREPROCESSING is the only option when a single buffer is used with both hardware and software vertex processing. D3DUSAGE_SOFTWAREPROCESSING is allowed for mixed and software devices.

It is possible to force vertex and index buffers into system memory by specifying D3DPOOL_SYSTEMMEM, even when the index processing is being done in hardware. This is a way to avoid overly large amounts of page-locked memory when a driver is putting these buffers into AGP memory.

The third parameter is either the D3DFMT_INDEX16 or D3DFMT_INDEX32 member of the **D3DFORMAT** enumerated type that specifies the size of each index.

The fourth parameter is a member of the **D3DPOOL** enumerated type that tells the system where in memory to place the new index buffer.

The final parameter that **CreateIndexBuffer** accepts is the address of a variable that is filled with a pointer to the new **IDirect3DIndexBuffer8** interface of the vertex buffer object, if the call succeeds.

The following C++ code example shows what creating a index buffer might look like in code.

```
/*
 * For the purposes of this example, the d3dDevice variable is the
 * address of an IDirect3DDevice8 interface exposed by a
 * Direct3DDevice object, g_IB is a variable of type
 * LPDIRECT3DINDEXBUFFER8.
 */
```

```
if( FAILED( d3dDevice->CreateIndexBuffer( 16384 *sizeof(WORD),
                                          D3DUSAGE_WRITEONLY, D3DFMT_
                                          D3DPOOL_DEFAULT, &g_IB ) )
    return E_FAIL;
```

Index Processing Requirements

The performance of index processing operations depends heavily on where the index buffer exists in memory and what type of rendering device is being used. Applications control the memory allocation for index buffers when they are created. When the D3DPOOL_SYSTEMMEM memory flag is set, the index buffer is created in system memory. When the D3DPOOL_DEFAULT memory flag is used, the device driver determines where the memory for the index buffer is best allocated, often referred to as driver-optimal memory. Driver-optimal memory can be local video memory, nonlocal video memory, or system memory.

Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **IDirect3DDevice8::CreateIndexBuffer** method specifies that the index buffer is to be used with software vertex processing. This flag is required in mixed mode vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING) when software vertex processing is used.

The application can directly write indices to a index buffer allocated in driver-optimal memory. This technique prevents a redundant copy operation later. This technique does not work well if your application reads data back from an index buffer, because read operations done by the host from driver-optimal memory can be very slow. Therefore, if your application needs to read during processing or writes data to the buffer erratically, a system-memory index buffer is a better choice.

Note Always use D3DPOOL_DEFAULT, except when you don't want to expend video memory or expand large amounts of page-locked RAM when the driver is putting vertex or index buffers into AGP memory.

Microsoft DirectX 8.1 (C++)

Access the Contents of an Index Buffer

[This is preliminary documentation and is subject to change.]

Index buffer objects enable applications to directly access the memory allocated for index data. You can retrieve a pointer to index buffer memory by calling the **IDirect3DIndexBuffer8::Lock** method, and then accessing the memory as needed to fill the buffer with new index data or to read any data it contains. The **Lock** method accepts four parameters. The first, *OffsetToLock*, is the offset into the index data. The second parameter is the size, measured in bytes, of the index data. The third parameter accepted by the **Lock** method, *ppbData*, is the address of a **BYTE** pointer filled with a pointer to the index data, if the call succeeds.

The last parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application accesses the data in the buffer. Specify constants for the *Flags* parameter according to the way the index data will be accessed by your application. This allows the driver to lock the memory and provide the best performance given the requested access type. Use D3DLOCK_READONLY flag if your application

will read only from the index buffer memory. Including this flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

After you fill or read the index data, call the **IDirect3DIndexBuffer8::Unlock** method, as shown in the following code example.

```
// This code example assumes the IB is a variable of type
// LPDIRECT3DINDEXBUFFER8 and that g_Indices has been properly
// initialized with indices.

// To fill the index buffer, you must lock the buffer to gain
// access to the indices. This mechanism is required because index
// buffers may be in device memory.

VOID* pIndices;

if( FAILED( IB->Lock( 0,                 // Fill from start of the buff
                      sizeof(g_Indices), // Size of the data to load.
                      (BYTE**)&pIndices, // Returned index data.
                      0 ) ) )            // Send default flags to the l
    return E_FAIL;

memcpy( pIndices, g_Indices, sizeof(g_Indices) );
IB->Unlock();
```

Note If you create an index buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the index buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

In C++, because you directly access the memory allocated for the index buffer, make sure your application properly accesses the allocated memory. Otherwise, you risk rendering that memory invalid. Use the stride of the index format your application uses to move from one index in the allocated buffer to another.

Retrieving Index Buffer Descriptions

Retrieve information about an index buffer by calling the **IDirect3DIndexBuffer8::GetDesc** method. This method fills the members of the **D3DINDEXBUFFER_DESC** structure with information about the vertex buffer.

Microsoft DirectX 8.1 (C++)

Render from an Index Buffer

[This is preliminary documentation and is subject to change.]

Rendering index data from an index buffer requires a few steps. First, you need to set the stream source by calling the **IDirect3DDevice8::SetStreamSource** method.

```
d3dDevice->SetStreamSource( 0, VB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to call the **IDirect3DDevice8::SetIndices** method to set the source of the index data.

```
d3dDevice->SetIndices( IB, 0 );
```

The first parameter that **SetIndices** accepts is the address of the index buffer to set. The second parameter is the starting point in the vertex stream.

After setting the stream and indices source, use the **IDirect3DDevice8::DrawIndexedPrimitive** method to render vertices that use indices from the index buffer.

```
d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0,
                                 dwVertices, 0, dwIndices / 3);
```

The second parameter that **DrawPrimitive** accepts is the minimum vertex index for vertices used during this call. The third parameter is the number of indices to use during this call starting from *BaseVertexIndex* + *MinIndex*. The fourth parameter is the location in the index array to start reading indices. The final parameter that **DrawPrimitive** accepts is the number of primitives to render. This parameter is a function of the primitive count and the primitive type. The code sample above uses triangles, so the number of primitives to render is the number of indices divided by three.

Microsoft DirectX 8.1 (C++)

Vertex Buffers

[This is preliminary documentation and is subject to change.]

Vertex buffers, represented by the **IDirect3DVertexBuffer8** interface, are memory buffers that contain vertex data. Vertex buffers can contain any vertex type—transformed or untransformed, lit or unlit—that can be rendered through the use of the rendering methods in the **IDirect3DDevice8** interface. You can process the vertices in a vertex buffer to perform operations such as transformation, lighting, or generating clipping flags. Transformation is always performed.

The flexibility of vertex buffers make them ideal staging points for reusing transformed geometry. You could create a single vertex buffer, transform, light, and clip the vertices in it, and render the model in the scene as many times as needed without re-transforming it, even with interleaved render state changes. This is useful when rendering models that use multiple textures: the geometry is transformed only once, and then portions of it can be rendered as needed, interleaved with the required texture changes. Render state changes made after vertices are processed take effect the next time the vertices are processed.

Description

A vertex buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of vertices it can contain. All these traits are held in a **D3DVERTEXBUFFER_DESC** structure.

Vertex buffer descriptions tell your application how an existing buffer was created and if it has been optimized since being created. You provide an empty description structure for the system to fill with the capabilities of a previously created vertex buffer. For more information about this task, see Accessing the Contents of a Vertex Buffer.

The **Format** member is set to D3DFMT_VERTEXDATA to indicate that this is a vertex buffer. The **Type** identifies the resource type of the vertex buffer. The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the vertex buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data within the vertex buffer.

Pool specifies the memory class that is allocated for the vertex buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the vertex buffer in system memory.

The **Size** member simply stores the size, in bytes, of the vertex buffer data. The **FVF** member contains a combination of Flexible Vertex Format Flags that identify the type of vertices that the buffer contains.

Memory Pool and Usage

You can create vertex buffers with the **IDirect3DDevice8::CreateVertexBuffer** method, which takes pool (memory class) and usage parameters. **CreateVertexBuffer** can also be created with a specified flexible vertex format (FVF) code for use in fixed function vertex processing, or as the output of process vertices. For details, see FVF Vertex Buffers.

The D3DUSAGE_SOFTWAREPROCESSING flag can be set when mixed-mode or software vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING) is enabled for that device. D3DUSAGE_SOFTWAREPROCESSING must be set for buffers to be used with software vertex processing in mixed mode, but it should not be set for the best possible performance when using hardware vertex processing in mixed mode. (D3DCREATE_HARDWARE_VERTEXPROCESSING). However, setting D3DUSAGE_SOFTWAREPROCESSING is the only option when a single buffer is to be used with both hardware and software vertex processing. D3DUSAGE_SOFTWAREPROCESSING is allowed for mixed as well as for software devices.

It is possible to force vertex and index buffers into system memory by specifying D3DPOOL_SYSTEMMEM, even when the vertex processing is done in hardware. This

is a way to avoid overly large amounts of page-locked memory when a driver is putting these buffers into AGP memory.

This section introduces the concepts necessary to understand and use vertex buffers in a Microsoft® Direct3D® application. Information is divided into the following sections.

- Creating a Vertex Buffer
- Accessing the Contents of a Vertex Buffer
- Rendering from a Vertex Buffer
- FVF Vertex Buffers
- Fixed Function Vertex Processing
- Programmable Vertex Processing
- Device Types and Vertex Processing Requirements

Microsoft DirectX 8.1 (C++)

Creating a Vertex Buffer

[This is preliminary documentation and is subject to change.]

You create a vertex buffer object by calling the **IDirect3DDevice8::CreateVertexBuffer** method, which accepts five parameters. The first parameter specifies the vertex buffer length, in bytes. Use the **sizeof** operator to determine the size of a vertex format, in bytes. Consider the following custom vertex format.

```
struct CUSTOMVERTEX {
        FLOAT x, y, z;
        FLOAT rhw;
        DWORD color;
        FLOAT tu, tv;   // The texture coordinates.
};

// Custom FVF, which describes the custom vertex structure.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TE
```

To create a vertex buffer to hold four CUSTOMVERTEX structures, specify [4*sizeof (CUSTOMVERTEX)] for the *Length* parameter.

The second parameter is a set of usage controls. Among other things, its value determines whether the vertex buffer is capable of containing clipping information—in the form of clip flags—for vertices that exist outside the viewing area. To create a vertex buffer that cannot contain clip flags, include the D3DUSAGE_DONOTCLIP flag for the *Usage* parameter. The D3DUSAGE_DONOTCLIP flag is applied only if you also indicate that the vertex buffer will contain transformed vertices—the D3DFVF_XYZRHW flag is included in the *FVF* parameter. The **CreateVertexBuffer** method ignores the D3DUSAGE_DONOTCLIP flag if you indicate that the buffer will contain untransformed vertices (the D3DFVF_XYZ flag). Clipping flags occupy additional memory, making a clipping-capable vertex buffer slightly larger than a vertex buffer incapable of containing clipping flags. Because these resources are allocated when the vertex buffer is created, you must request a clipping-capable vertex buffer ahead of time.

The third parameter, *FVF*, is a combination of Flexible Vertex Format Flags that describe the vertex format of the vertex buffer. If you specify 0 for this parameter, then

the vertex buffer is a non-FVF vertex buffer. For more information, see FVF Vertex Buffers. The fourth parameter describes the memory class into which to place the vertex buffer.

The final parameter that **CreateVertexBuffer** accepts is the address of a variable that will be filled with a pointer to the new **IDirect3DVertexBuffer8** interface of the vertex buffer object, if the call succeeds.

Note You cannot produce clip flags for a vertex buffer that was created without support for them.

The following C++ code example shows what creating a vertex buffer might look like in code.

```
    // For the purposes of this example, the d3dDevice variable is
    // the address of an IDirect3DDevice8 interface exposed by a
    // Direct3DDevice object, g_pVB is a variable of type
    // LPDIRECT3DVERTEXBUFFER8.

// The custom vertex type
struct CUSTOMVERTEX {
        FLOAT x, y, z;
        FLOAT rhw;
        DWORD color;
        FLOAT tu, tv;   // The texture coordinates
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TE

    // Create a clipping-capable vertex buffer. Allocate enough memory
    // in the  default memory pool to hold three CUSTOMVERTEX
    // structures.
    if( FAILED( d3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                               0 /* Usage */, D3DFVF
                                               D3DPOOL_DEFAULT, &g_p
        return E_FAIL;
```

Microsoft DirectX 8.1 (C++)

Accessing the Contents of a Vertex Buffer

[This is preliminary documentation and is 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 **IDirect3DVertexBuffer8::Lock** method, and then accessing the memory as needed to fill the buffer with new vertex data or to read any data it already contains. The **Lock** method accepts four parameters. The first, *OffsetToLock*, is the offset into the vertex data. The second parameter is the size, measured in bytes, of the vertex data. The third parameter accepted by the **Lock** method, *ppbData*, is the address of a **BYTE** pointer that points to the vertex data, if the call succeeds.

The last parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application will access the data in the buffer. Specify constants for the *Flags* parameter according to the way the vertex data will be accessed. This allows the driver to lock the memory and provide the best performance, given the

requested access type. Use D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

After you finish filling or reading the vertex data, call the **IDirect3DVertexBuffer8::Unlock** method, as shown in the following code example.

```
// This code example assumes the g_pVB is a variable of type
// LPDIRECT3DVERTEXBUFFER8 and that g_Vertices has been properly
// initialized with vertices.

// To fill the vertex buffer, you need to lock the buffer to
// gain access to the vertices. This mechanism is required because
// vertex buffers may be in device memory.
VOID* pVertices;

if( FAILED( g_pVB->Lock( 0,                 // Fill from the start of
                                            // the buffer.
                         sizeof(g_Vertices), // Size of the data to
                                            // load.
                         (BYTE**)&pVertices, // Returned vertex data.
                         0 ) ) )            // Send default flags to
                                            // the lock.
    return E_FAIL;

memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();
```

Performance Notes If you create a vertex buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

In C++, because you directly access the memory allocated for the vertex buffer, make sure your application properly accesses the allocated memory. Otherwise, you risk rendering that memory invalid. Use the stride of the vertex format that your application uses to move from one vertex in the allocated buffer to another. The vertex buffer memory is a simple array of vertices specified in flexible vertex format. Use the stride of whatever vertex format structure you define. You can calculate the stride of each vertex at run time by examining the Flexible Vertex Format Flags contained in the vertex buffer description. The following table shows the size for each vertex component.

Vertex Format Flag	Size
D3DFVF_DIFFUSE	**sizeof**(**DWORD**)
D3DFVF_NORMAL	**sizeof**(**float**) × 3
D3DFVF_SPECULAR	**sizeof**(**DWORD**)
D3DFVF_TEX*n*	**sizeof**(**float**) × *n* × *t*
D3DFVF_XYZ	**sizeof**(**float**) × 3
D3DFVF_XYZRHW	**sizeof**(**float**) × 4

The number of texture coordinates present in the vertex format is described by the D3DFVF_TEX*n* flags (where *n* is a value from 0 to 8). Multiply the number of texture coordinate sets by the size of one set of texture coordinates, which can range from one to four floats, to calculate the memory required for that number of texture coordinates.

Use the total vertex stride to increment and decrement the memory pointer as needed to access particular vertices.

Retrieving Vertex Buffer Descriptions

You can retrieve information about a vertex buffer by calling the **IDirect3DVertexBuffer8::GetDesc** method. This method fills the members of the **D3DVERTEXBUFFER_DESC** structure with information about the vertex buffer.

Microsoft DirectX 8.1 (C++)

Rendering from a Vertex Buffer

[This is preliminary documentation and is subject to change.]

Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source by calling the **IDirect3DDevice8::SetStreamSource** method, as shown in the following example.

```
d3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to inform Direct3D which vertex shader to use by calling the **IDirect3DDevice8::SetVertexShader** method. The following sample code sets an FVF code for the vertex shader. This informs Direct3D of the types of vertices it is dealing with.

```
d3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
```

After setting the stream source and vertex shader, any draw methods will use the vertex buffer. The code example below shows how to render vertices from a vertex buffer with the **IDirect3DDevice8::DrawPrimitive** method.

```
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

The second parameter that **DrawPrimitive** accepts is the index of the first vector in the vertex buffer to load.

Microsoft DirectX 8.1 (C++)

FVF Vertex Buffers

[This is preliminary documentation and is subject to change.]

Setting the FVF parameter of the **IDirect3DDevice8::CreateVertexBuffer** method to a nonzero value, which must be a valid FVF code, indicates that the buffer content is to be characterized by an FVF code. A vertex buffer that is created with an FVF code is referred to as an FVF vertex buffer. Some methods or uses of **IDirect3DDevice8** require FVF vertex buffers, and others require non-FVF vertex buffers. FVF vertex buffers are required as the destination vertex buffer argument for **IDirect3DDevice8::ProcessVertices**.

FVF vertex buffers can be bound to a source data stream for any stream number. However, FVF vertex buffers are not allowed to be used for inputs to programmed vertex shaders

The presence of the D3DFVF_XYZRHW component on FVF vertex buffers indicates that the vertices in that buffer have been processed. Vertex buffers used for **ProcessVertices** destination vertex buffers must be post-processed. Vertex buffers used for fixed function shader inputs can be either pre- or post-processed. If the vertex buffer is post-processed, then the shader is effectively bypassed and the data is passed directly to the primitive clipping and setup module.

FVF vertex buffers can be used with vertex shaders. Also, vertex streams can represent the same vertex formats that non-FVF vertex buffers can. They do not have to be used to input data from separate vertex buffers. The additional flexibility of the new vertex streams enables applications that need to keep their data separate to work better, but it is not required. If the application can maintain interleaved data in advance, then that is a performance boost. If the application will only interleave the data before every rendering call, then it should enable the application programming interface (API) or hardware to do this with multiple streams.

The most important things with vertex performance is to use a 32-byte vertex, and to maintain good cache ordering.

Microsoft DirectX 8.1 (C++)

Fixed Function Vertex Processing

[This is preliminary documentation and is subject to change.]

In the fixed function vertex pipeline, processing the vertices in a vertex buffer applies the current transformation matrices for the device. Vertex operations such as lighting, generating clip flags, and updating extents can also be applied, optionally. When using fixed function vertex processing, modifying the elements in the destination vertex buffer is controlled by the D3DPV_DONOTCOPYDATA flag. This flag applies only to fixed function vertex processing. The **IDirect3DDevice8** interface exposes the **IDirect3DDevice8::ProcessVertices** method to process vertices. You process vertices from a vertex shader to the set of input data streams, generating a single stream of interleaved vertex data to the destination vertex buffer by calling the **ProcessVertices** method. The method accepts five parameters that describe the location and quantity of vertices that the method targets, the destination vertex buffer, and the processing options. After the call, the destination buffer contains the processed vertex data.

The first, second, and third parameters, *SrcStartIndex*, *DestIndex*, and *VertexCount*,

reflect the index of the first vertex to load, the index within the destination buffer at which the vertices will be placed, and the total number of vertices to process and place in the destination buffer. The fourth parameter, *pDestBuffer*, should be set to the address of the **IDirect3DVertexBuffer8** of the vertex buffer object that will receive the source vertices. The *SrcStartIndex* specifies the index at which the method should start processing vertices.

The final parameter, *Flags*, determines special processing options for the method. You can set this parameter to 0 for default vertex processing, or to D3DPV_DONOTCOPYDATA to optimize processing in some situations. When you set *Flags* to 0, vertex components of the destination vertex buffer's vertex format that are not affected by the vertex operation are still copied from the vertex shader or set to 0. However, when using D3DPV_DONOTCOPYDATA, **ProcessVertices** does not overwrite color and texture coordinate information in the destination buffer unless this data is generated by Microsoft® Direct3D®. Diffuse color is generated when lighting is enabled, that is, D3DRS_LIGHTING is set to TRUE. Specular color is generated when lighting is enabled and specular is enabled, that is, D3DRS_SPECULARENABLE and D3DRS_LIGHTING are set to TRUE. Specular color is also generated when fog is enabled. Texture coordinates are generated when texture transform or texture generation is enabled. **ProcessVertices** uses the current render states to determine what vertex processing should be done.

FVF Usage Settings for Destination Vertex Buffers

The **IDirect3DDevice8::ProcessVertices** method requires specific settings for the Flexible Vertex Format Flags of the destination vertex buffer. The flexible vertex format (FVF) usage settings must be compatible with the current settings for vertex processing.

For fixed function vertex processing, **ProcessVertices** requires the following FVF settings.

- Position type is always D3DFVF_XYZRHW; so, D3DFVF_XYZ and D3DFVF_XYZB1 through D3DFVF_XYZB5 are invalid.
- The D3DFVF_NORMAL, D3DFVF_RESERVED0, and D3DFVF_RESERVED2 flags must not be set.
- The D3DFVF_DIFFUSE flag must be set if an **OR** operation of the following conditions returns true.
  - Lighting is enabled, that is, D3DRS_LIGHTING is true.
  - Lighting is disabled, and diffuse color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
- The D3DFVF_SPECULAR flag must be set if an **OR** operation of the following conditions returns true.
  - Lighting is enabled and specular color is enabled, that is, D3DRS_SPECULARENABLE is true.
  - Lighting is disabled, and specular color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
  - Vertex fog is enabled, that is, D3DRS_FOGVERTEXMODE is not set to D3DFOG_NONE.

In addition, the texture coordinate count must be set in the following manner.

- If texture transform and texture generation are disabled for all active texture stages, and the D3DPV_DONOTCOPYDATA is not set, then the number and type of output texture coordinates are required to match those of the input vertex

texture coordinates. If D3DPV_DONOTCOPYDATA is set and texture transform and texture generation are disabled, then the output texture coordinates are ignored.

- If texture transform or texture generation is enabled for any active texture stages, the output vertex might need to contain more texture coordinate sets than the input vertex. This is due to the proliferation of texture coordinates from those being generated by texture generation or derived by texture transforms. Note that a similar proliferation of texture coordinates occurs during **DrawPrimitive** calls, but is not visible to the application programmer. In this case, Microsoft® Direct3D® generates a new set of texture coordinates. The new set of texture coordinates is derived by stepping through the texture stages and analyzing the settings for texture generation, texture transformation, and texture coordinate index to determine if a unique set of texture coordinates is required for that stage. Each time a new set is required it is allocated in increasing order. Note that the maximum, and typical, requirement is one set per stage, although it might be less due to sharing of nontransformed texture coordinates through D3DTSS_TEXCOORDINDEX.

Thus, for each texture stage, a new set of texture coordinates is generated if a texture is bound to that stage and any of the following conditions are true.

- Texture generation is enabled for that stage.
- Texture transformation is enabled for that stage.
- Nontransformed input texture coordinates are referenced through D3DTSS_TEXCOORDINDEX for the first time.

When Direct3D is generating texture coordinates, the application is required to perform the following actions.

1. Use a destination vertex buffer with the appropriate FVF usage.
2. Reprogram the D3DTSS_TEXCOORDINDEX of the texture stage according to the placement of the post -processed texture coordinates. Note that the reprogramming of the D3DTSS_TEXCOORDINDEX setting occurs when the processed vertex buffer is used in subsequent **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** calls.

Finally, texture coordinate dimensionality (D3DFVF_TEX0 through D3DFVF_TEX8) must be set in the following manner.

- For each texture coordinate set, if texture transform and texture generation are disabled, then the output texture coordinate dimensionality must match the input. If the texture transform is enabled, then the output dimensionality must match the count defined by the D3DTTFF_COUNT1, D3DTTFF_COUNT2, D3DTTFF_COUNT3, or D3DTTFF_COUNT4 settings. If the texture transform is disabled and texture generation is enabled, then the output dimensionality must match the settings for the texture generation mode; currently all modes generate 3 float values.

When **ProcessVertices** fails due to an incompatible destination vertex buffer FVF code, the expected code is printed to the debug output (debug builds only).

Microsoft DirectX 8.1 (C++)

Programmable Vertex Processing

[This is preliminary documentation and is subject to change.]

When using a programmed vertex shader, the elements updated in the destination vertex buffer are controlled by the shader function program. When the application writes to one of the destination vertex registers, the corresponding element within each vertex of the destination vertex buffer is updated. Elements in the destination vertex buffer that are not written to by the application are not modified. **IDirect3DDevice8::ProcessVertices** will fail if the application writes to elements that are not present in the destination vertex buffer.

Microsoft DirectX 8.1 (C++)

Device Types and Vertex Processing Requirements

[This is preliminary documentation and is subject to change.]

The performance of vertex processing operations, including transformation and lighting, depends heavily on where the vertex buffer exists in memory and what type of rendering device is being used. Applications control the memory allocation for vertex buffers when they are created. When the D3DPOOL_SYSTEMMEM memory flag is set, the vertex buffer is created in system memory. When the D3DPOOL_DEFAULT memory flag is used, the device driver determines where the memory for the vertex buffer is best allocated, often referred to as driver-optimal memory. Driver-optimal memory can be local video memory, nonlocal video memory, or system memory.

Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **IDirect3DDevice8::CreateVertexBuffer** method specifies that the vertex buffer is to be used with software vertex processing. This flag is required for software vertex processing in mixed vertex processing mode. This flag is allowed in software vertex processing mode and disallowed in hardware vertex processing mode. Vertex buffers used with software vertex processing include the following:

- All input streams for **IDirect3DDevice8::ProcessVertices**.
- All input streams for **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** when software vertex processing. For more information, see D3DRS_SOFTWAREVERTEXPROCESSING.

The reasoning you use to determine the memory location—system or driver optimal—for vertex buffers is the same as that for textures. Vertex processing, including transformation and lighting, in hardware works best when the vertex buffers are allocated in driver-optimal memory, while software vertex processing works best with vertex buffers allocated in system memory. For textures, hardware rasterization works best when textures are allocated in driver-optimal memory, while software rasterization works best with system-memory textures.

Note Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports standalone processing of vertices, without rendering any primitive with the **ProcessVertices** method. This standalone vertex processing is always performed in software on the host processor. Because of this, vertex buffers used as sources set with **IDirect3DDevice8::SetStreamSource** must be created with the D3DUSAGE_SOFTWAREPROCESSING flag. The functionality provided by

ProcessVertices is identical to that of the **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** methods while using software vertex processing.

If your application performs its own vertex processing and passes transformed, lit, and clipped vertices to rendering methods, then the application can directly write vertices to a vertex buffer allocated in driver-optimal memory. This technique prevents a redundant copy operation later. Note that this technique will not work well if your application reads data back from a vertex buffer, because read operations done by the host from driver-optimal memory can be very slow. Therefore, if your application needs to read during processing or writes data to the buffer erratically, a system-memory vertex buffer is a better choice.

When using the Direct3D vertex processing features (by passing untransformed vertices to vertex-buffer rendering methods), processing can occur in either hardware or software depending on the device type and device creation flags. It is recommended that vertex buffers be allocated in pool D3DPOOL_DEFAULT for best performance in virtually all cases. When a device is using hardware vertex processing, there is a number of additional optimizations that may be done based on the flags D3DUSAGE_DYNAMIC and D3DUSAGE_WRITEONLY. See **IDirect3DDevice8::CreateVertexBuffer** for more information on using these flags.

Microsoft DirectX 8.1 (C++)

Higher-Order Primitives

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® for Microsoft DirectX® 8.1 supports points, lines, triangles, and grid primitives. These have been extended to support higher-order interpolation beyond linear. While triangles and lines have spatial extent, until now they were both rendered using linear interpolation. In DirectX 8.1, Direct3D supports rendering of these primitive types using higher order, up to quintic, interpolation. Furthermore, a new quad primitive type is now supported. This new type can also be rendered with higher-order interpolation. This feature is primarily driven by requirements for animation and rendering of characters. It can also be used for other surfaces such as terrain or water.

Higher-order primitives support higher-order interpolation when transmitted to the application programming interface (API) as lists, strips, fans, or indexed meshes. This is achieved by using additional information encoded in the vertices themselves. For example, normal vectors can be used to define tangent planes at the vertices to enable cubic interpolation. Most implementations support higher-order interpolation by tessellation into planar triangles. The tessellation step is applied logically before the vertex shader stage. Because the vertex shader API does not impose semantics on its input data, a special mechanism is provided to identify the vertex stream component that represents the position, and optionally the normal vector. All other components are interpolated accordingly.

This section introduces higher-order primitives and discusses how they can be used in your applications. Information is divided into the following topics.

Improved Quality through Resolution Enhancement

Current primitives are not ideal for representing smooth surfaces. Higher-order interpolation methods, such as cubic polynomials, allow more accurate calculations in rendering curved shapes. This provides increased realism by reducing or eliminating faceting artifacts visible on silhouette edges or on specular surface lighting. Furthermore, when tessellation occurs on the chip, the tessellated triangles do not impact the bus bandwidth. In many cases, a small amount of tessellation can provide improvements in image quality with minimal performance impact.

Microsoft® Direct3D® for Microsoft DirectX® 8.1 provides a simple way to apply resolution enhancement to content created by existing polygon-oriented tools and art pipelines. The application need only provide a desired level of tessellation, and transmit the data using standard triangle syntax that includes normal vectors.

Direct Mapping from Spline-Based Tools

Many current authoring tools support higher-order primitives to enable more powerful modeling operations than are commonly provided for with planar triangle meshes. When used efficiently, so that the number of patches generated is reasonable, such tools can produce content that can be rendered directly by the application programming interface (API). To meet this requirement, a new entry point has been added that interprets the incoming vertex data stream as a 2-D array of control points and tessellates it to the desired resolution.

- Using Higher-Order Primitives

Microsoft DirectX 8.1 (C++)

Using Higher-Order Primitives

[This is preliminary documentation and is subject to change.]

This section shows you how to use higher-order primitives in your application.

Determining Higher-Order Primitive Support

The **DevCaps** member of **D3DCAPS8** can be queried to determine the level of support for operations involving higher-order primitives. The following table lists the device capabilities related to higher-order primitives in Microsoft® DirectX® 8.1.

Device capability	Description
D3DDEVCAPS_NPATCHES	Device supports N-patches.
D3DDEVCAPS_QUINTICRTPATCHES	Device supports quintic béziers and B-splines.
D3DDEVCAPS_RTPATCHES	Device supports rectangular and triangular (RT) patches.
D3DDEVCAPS_RTPATCHHANDLEZERO	RT patches might be drawn efficiently using handle zero.

Drawing Patches

DirectX 8.1 supports two types of higher order primitives, or patches. These are referred to as N-Patches and Rect/Tri patches. N-Patches can be rendered using any triangle

rendering call by enabling D3DRS_PATCHSEGMENTS to a value greater than 1.0. Rect/Tri patches must be rendered using the following explicit entry points.

You can use the following methods to draw patches in Microsoft® Direct3D® for DirectX.

- **IDirect3DDevice8::DrawRectPatch**. To better understand how the patch data is referenced in the vertex buffer, see **D3DRECTPATCH_INFO**.
- **IDirect3DDevice8::DrawTriPatch**. To better understand how the patch data is referenced in the vertex buffer, see **D3DTRIPATCH_INFO**.

DrawRectPatch draws a rectangular high-order patch specified by the *pRectPatchInfo* parameter using the currently set streams. The *Handle* parameter is used to associate the patch with a handle, so that the next time the patch is drawn, there is no need to respecify *pRectPatchInfo*. This makes it possible to precompute and cache forward difference coefficients or other information, which in turn enables subsequent calls to **DrawRectPatch** using the same handle to execute efficiently.

It is intended that for static patches, an application would set the vertex shader and appropriate streams, supply patch information in the *pRectPatchInfo* parameter, and specify a handle so that Direct3D can capture and cache information. The application can then call **DrawRectPatch** subsequently with *pRectPatchInfo* set to NULL to efficiently draw the patch. When drawing a cached patch, the currently set streams are ignored. However, it is possible to override the cached *pNumSegs* by specifying new values for *pNumSegs*. Also, it is required to set the same vertex shader when rendering a cached patch as was set when it was captured.

For dynamic patches, the patch data changes for every rendering of the patch, so it is not efficient to cache information. The application can convey this to Direct3D by setting *Handle* to 0. In this case, Direct3D draws the patch using the currently set streams and the *pNumSegs* values and does not cache any information. It is not valid to simultaneously set *Handle* to 0 and *pPatch* to NULL.

By respecifying *pRectPatchInfo* for the same handle, the application can overwrite the previously cached information.

If *pNumSegs* is set to NULL, then the tessellator uses D3DRS_PATCHSEGMENTS to control the amount of tessellation. In this case, obviously, the same tessellation is used for all sides. There are no special methods for N-patches. Tessellation is simply controlled by D3DRS_PATCHSEGMENTS. It is necessary to supply normals.

DrawTriPatch is similar to **DrawRectPatch** except that it draws a triangular high-order patch.

Microsoft DirectX 8.1 (C++)

Point Sprites

[This is preliminary documentation and is subject to change.]

Support for point sprites in Microsoft® Direct3D® for Microsoft DirectX® 8.1 enables the high-performance rendering of points (particle systems). Point sprites are

generalizations of generic points that enable arbitrary shapes to be rendered as defined by textures.

Point Primitive Rendering Controls

Microsoft® Direct3D® for Microsoft DirectX® 8.1 supports additional parameters to control the rendering of point sprites (point primitives). These parameters enable points to be of variable size and have a full texture map applied. The size of each point is determined by an application-specified size combined with a distance-based function computed by Direct3D. The application can specify point size either as per-vertex or by setting D3DRS_POINTSIZE, which applies to points without a per-vertex size. The point size is expressed in camera space units, with the exception of when the application is passing post-transformed flexible vertex format (FVF) vertices. In this case, the distance-based function is not applied and the point size is expressed in units of pixels on the render target.

The texture coordinates computed and used when rendering points depends on the setting of D3DRS_POINTSPRITEENABLE. When this value is set to TRUE, the texture coordinates are set so that each point displays the full-texture. In general, this is only useful when points are significantly larger than one pixel. When D3DRS_POINTSPRITEENABLE is set to FALSE, each point's vertex texture coordinate is used for the entire point.

Point Size Computations

Point size is determined by D3DRS_POINTSCALEENABLE. If this value is set to FALSE, the application-specified point size is used as the screen-space (post-transformed) size. Vertices that are passed to Microsoft® Direct3D® in screen space do not have point sizes computed; the specified point size is interpreted as screen-space size.

If D3DRS_POINTSCALEENABLE is TRUE, Direct3D computes the screen-space point size according to the following formula. The application-specified point size is expressed in camera space units.

$$S_s = V_h * S_i * \mathrm{sqrt}(1/(A + B*D_e + C*(D_e^2)))$$

In this formula, the input point size, S_i, is either per-vertex or the value of the D3DRS_POINTSIZE render state. The point scale factors, D3DRS_POINTSCALE_A, D3DRS_POINTSCALE_B, and D3DRS_POINTSCALE_C, are represented by the points A, B, and C. The height of the viewport, V_h, is the **Height** member of the **D3DVIEWPORT8** structure representing the viewport. D_e, the distance from the eye to the position (the eye at the origin), is computed by taking the eye space position of the point (X_e, Y_e, Z_e) and performing the following operation.

$$D_e = \mathrm{sqrt}\ (X_e^2 + Y_e^2 + Z_e^2)$$

The maximum point size, P_{max}, is determined by taking the smaller of either the **MaxPointSize** member of **D3DCAPS8** or the D3DRS_POINTSIZE_MAX render state. The minimum point size, P_{min}, is determined by querying the value of D3DRS_POINTSIZE_MIN. Thus the final screen-space point size, S, is determined in

the following manner.

- If $S_s > P_{max}$, then $S = P_{max}$
- if $S_s < P_{min}$, then $S = P_{min}$
- Otherwise, $S = S_s$

Point Rendering

A screen-space point,

P = (**X**, **Y**, **Z**, **W**)

of screen-space size **S** is rasterized as a quadrilateral of the following four vertices.

((**X**+**S**/2, **Y**+**S**/2, **Z**, **W**), (**X**+**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**+**S**/2, **Z**, **W**))

The vertex color attributes are duplicated at each vertex; thus each point is always rendered with constant colors.

The assignment of texture indices is controlled by the D3DRS_POINTSPRITEENABLE render state setting. If D3DRS_POINTSPRITEENABLE is set to FALSE, then the vertex texture coordinates are duplicated at each vertex. If D3DRS_POINTSPRITEENABLE is set to TRUE, then the texture coordinates at the four vertices are set to the following values.

(0.F, 0.F), (0.F, 1.F), (1.F, 0.F), (1.F, 1.F)

This is shown in the following diagram.

When clipping is enabled, points are clipped in the following manner. If the vertex exceeds the range of depth values—**MinZ** and **MaxZ** of the **D3DVIEWPORT8** structure—into which a scene is to be rendered, the point exists outside of the view frustum and is not rendered. If the point, taking into account the point size, is completely outside the viewport in **X** and **Y**, then the point is not rendered; the remaining points are rendered. It is possible for the point position to be outside the viewport in **X** or **Y** and still be partially visible.

Points may or may not be correctly clipped to user-defined clip planes. If D3DPMISCCAPS_CLIPPLANESCALEDPOINTS is not set in the **PrimitiveMiscCaps** member of **D3DCAPS8**, then points are clipped to user-defined clip planes based only on the vertex position, ignoring the point size. In this case, scaled points are fully rendered when the vertex position is inside the clip planes, and discarded when the vertex position is outside a clip plane. Applications can prevent potential artifacts by adding a border geometry to clip planes that is as large as the maximum point size.

If the D3DPMISCCAPS_CLIPPLANESCALEDPOINTS bit is set, then the scaled points are correctly clipped to user-defined clip planes.

Hardware vertex processing may or may not support point size. For example, if a device

is created with D3DCREATE_HARDWARE_VERTEXPROCESSING on a HAL Device(D3DDEVTYPE_HAL) that has the **MaxPointSize** member of its **D3DCAPS8** structure set to 1.0 or 0.0, then all points are a single pixel. To render pixel point sprites less than 1.0, you must use either flexible vertex format (FVF) TL (transformed and lit) vertices or software vertex processing (D3DCREATE_SOFTWARE_VERTEXPROCESSING), in which case the Microsoft® Direct3D® run time emulates the point sprite rendering. For details on FVF TL vertices, see Transformed and Lit Vertex Functionality.

A hardware device that does vertex processing and supports point sprites—**MaxPointSize** set to greater than 1.0f—is required to perform the size computation for nontransformed sprites and is required to properly set the per-vertex or D3DRS_POINTSIZE for TL vertices.

Microsoft DirectX 8.1 (C++)

Samples

[This is preliminary documentation and is subject to change.]

DirectX Graphics C/C++ Samples

The following samples are built on a base class that includes Microsoft® Windows® and Microsoft® Direct3D® functionality. This base class provides many of the basic features in a Windows application, such as creating windows and handling messages. The samples include a derived class that overrides the methods necessary to add Direct3D features, such as bump maps, vertex blending, and volume textures. For more information about the sample architecture, see Sample Framework.

- Billboard Sample
- BumpEarth Sample
- BumpLens Sample
- BumpUnderwater Sample
- BumpWaves Sample
- Bump Self-Shadow Sample
- ClipMirror Sample
- CubeMap Sample
- Cull Sample
- DolphinVS Sample
- DotProduct3 Sample
- DXTex Tool
- Emboss Sample
- EnhancedMesh Sample
- FishEye Sample
- Lighting Sample
- MFCFog Sample
- MFCPixelShader Sample
- MFCTex Sample
- Moire Sample
- OptimizedMesh Sample
- Pick Sample

- PointSprites Sample
- ProgressiveMesh Sample
- RTPatch Sample
- ShadowVolume Sample
- SkinnedMesh Sample
- SphereMap Sample
- StencilDepth Sample
- StencilMirror Sample
- Text3D Sample
- VertexBlend Sample
- VertexShader Sample
- VolumeTexture Sample
- Water Sample

The Application Wizard is available to help generate Microsoft® DirectX® applications.

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see Compiling DirectX Samples and Other DirectX Applications.

See Also

DirectX Graphics C/C++ Tutorials

Microsoft DirectX 8.1 (C++)

Sample Framework

[This is preliminary documentation and is subject to change.]

The Microsoft® DirectX® 8.1 and Microsoft® Direct3D® Software Development Kit (SDK) graphics sample framework is an evolution from the DirectX 7.0 graphics sample framework. The SDK samples are installed by default in \Mssdk\Samples\Multimedia. The folders of interest are Common and Direct3D. The sample framework is contained in the Common folder and the Direct3D samples based on the graphics framework are contained in the Direct3D folder.

The graphics framework consists of five source modules.

- **d3dapp.cpp** exposes the application interface used for samples. Of particular interest is class CD3DApplication.
- **d3dfile.cpp** furnishes .x file support, to enable samples to load .x files. Of particular interest are classes CD3Dmesh and CD3DFrame.
- **d3dfont.cpp** furnishes basic font output support, to enable things like statistics views. Of particular interest is class CD3DFont.
- **d3dutil.cpp** provides generally useful 3 dimensional functions, such as material, light, and texture helper functions.
- **dxutil.cpp** provides generally useful DirectX functions, such as media, registry, and timer helper functions.

Corresponding header files are located in the Common\Include folder.

Each sample implements a subclass of CD3DApplication, which is typically named CMyD3DApplication, and set of overridables that are shown below.

```
// Overridable functions for the 3 dimensional scene created by the app
virtual HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT)   { return S_O
virtual HRESULT OneTimeSceneInit()                         { return S_O
virtual HRESULT InitDeviceObjects()                        { return S_O
virtual HRESULT RestoreDeviceObjects()                     { return S_O
virtual HRESULT FrameMove()                                { return S_O
virtual HRESULT Render()                                   { return S_O
virtual HRESULT InvalidateDeviceObjects()                  { return S_O
virtual HRESULT DeleteDeviceObjects()                      { return S_O
virtual HRESULT FinalCleanup()                             { return S_O

The prototypes for these methods are contained in d3dapp.h in the
 CD3Dapplication class. The samples create a new Direct3D application a
 those methods that are needed by the application.
```

Derived Class Example

This example uses a subset of the overrideable methods. The class CMyD3DApplication contains the following methods. Each of these methods is explained below.

```
class CMyD3DApplication : public CD3DApplication
{
public:
    CMyD3DApplication();

protected:
    HRESULT ConfirmDevice( D3DCAPS8*, DWORD, D3DFORMAT );
    HRESULT DeleteRestoreDeviceObjects();
    HRESULT RestoreDeviceObjects();
    HRESULT FrameMove();
    HRESULT Render();

private:
    LPDIRECT3DVERTEXBUFFER8 m_pVB;              // Vertex buffer to hol
};
```

Constructor

The constructor initializes the window title, enables depth buffering and initializes the vertex buffer.

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle    = _T("D3D Example");    // title bar string
    m_bUseDepthBuffer   = TRUE;                 // enable depth buffer
    m_pVB                = NULL;                // initialize
}
```

The window title is a wide character string that is visible in the title bar or the window class when the application is invoked. It is optional.

The base class contains a member variable for enabling depth buffering. The default value for this boolean variable is FALSE, which disables depth buffering.

The window title is a wide character string that is visible in the title bar or the window class when the application is invoked. It is optional.

ConfirmDeviceObjects

DeleteDeviceObjects

DeleteDeviceObjects is called when the application is exiting, or if the device is being changed. You use this method to delete device dependent objects, such as the vertex buffer.

```
HRESULT CVShader1::DeleteDeviceObjects()
{
    m_pQuadVB->Release();
    m_pQuadVB = NULL;
    return S_OK;
}
```

RestoreDeviceObjects

This method is called when the application needs to restore device memory objects and device states. This is required after a DirectX device is created or resized. This method does most of the work of creating objects and initializing render states.

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // Create the vertex buffer. Allocate enough memory (from the defau
        // to hold the custom vertices. Specify the flexible vertex for
        //      data it contains.
    if( FAILED( m_pd3dDevice->CreateVertexBuffer( NUM_VERTS*sizeof(CUST
                          0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &m_p
    {
        return E_FAIL;
    }

    // Fill the vertex buffer. First, lock the vertex buffer to get acc
        // vertices. This mechanism is required because vertex buffers
        // memory. Then use memcpy to do a fast data copy.
    VOID* pVertices;
    if( FAILED( m_pVB->Lock( 0, sizeof(g_Vertices),
     (BYTE**)&pVertices, 0 ) ) )
        return E_FAIL;
    memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
    m_pVB->Unlock();

    // Set the projection matrix. The size of the back buffer comes fro
        // class
    D3DXMATRIX matProj;
    FLOAT fAspect = m_d3dsdBackBuffer.Width /
                    (FLOAT)m_d3dsdBackBuffer.Height;
    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, fAspect,
                                1.0f, 100.0f );
    m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );

    // Set up the view matrix. A view matrix can be defined from an eye
    // point, a look at point and an up direction vector. In this examp
    // the eye position is (0,1,-4) the look at point is (0,0,0) and th
    // up vector is (0,1,0.
    D3DXMATRIX matView;
    D3DXMatrixLookAtLH( &matView, &D3DXVECTOR3( 0.0f, 1.0f,-4.0f ),
                                  &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ),
```

```
                                    &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
    m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );

    // Set up default texture states

    // Set up render states ( this is only one example renderstate )
    m_pd3dDevice->SetRenderState( D3DRS_CULLMODE,     D3DCULL_NONE );

    return S_OK;
}
```

This method creates the vertex buffer and copies the vertex data into it. It creates the view and projection matrices, which define the camera orientation to the object in the vertex buffer. Texture stage states can be set in this method although none are present in this example. Render states that are not likely to change are set. These determine how the scene renders.

FrameMove

This method contains actions that happen every frame such as animation. In this example, it adds a y axis rotation to the world transform.

```
HRESULT CMyD3DApplication::FrameMove()
{
    // For our world matrix, just rotate the object about the y-axis.
    D3DXMATRIX matWorld;
    D3DXMatrixRotationY( &matWorld, ::TimeGetTime()/150.0f );
    m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );

    return S_OK;
}
```

The Windows method ::TimeGetTime() is used to return the current time. Once it is divided by 150, this generates a incremental angle to rotate the object.

Render

This method is called when it is time to render the output. This function clears the view port and render the scene. It also renders state changes.

```
HRESULT CMyD3DApplication::Render()
{
    // Clear the viewport
    m_pd3dDevice->Clear( 0L, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                D3DCOLOR_XRGB(0,0,0), 1.0f, 0L );

    // Begin the scene
    if( SUCCEEDED( m_pd3dDevice->BeginScene() ) )
    {
        m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) )
            m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
        m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, NUM_TRIS );

        m_pd3dDevice->EndScene();
    }

    return S_OK;
}
```

The Render method first clears the viewport using Clear. Then, within the BeginScne/EndScene pair it uses SetStreamSource to inform the runtime that it uses vertex buffer m_pVB with a stride of the size of the custom vertex type. Then, it informs the runtime that it uses a flexible vertex format (FVF) shader, the simplest type. Finally it invokes DrawPrimitive to render the quad.

Other functions

DeleteDeviceObjects is called when the application exits, or if the device changes. You use this method to delete device dependent objects.

ConfirmDevice checks the device for some minimum set of capabilities. It is called during the device initialization.

InvalidateDeviceObjects is called when the device dependent objects might be removed. Device dependent objects such as vertex buffers are usually added to this method.

OneTimeSceneInit is provided for code that needs to run during the initial application startup.

DirectXDev for graphics, networking, and input at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXDEV.html.

Microsoft DirectX 8.1 (C++)

Billboard Sample

[This is preliminary documentation and is subject to change.]

Description

The Billboard sample illustrates the billboarding technique. Rather than rendering complex 3-D models, such as a high-polygon tree model, billboarding renders a 2-D image of the model and rotates it to always face the eyepoint. This technique is commonly used to render trees, clouds, smoke, explosions, and more. For more information, see Billboarding.

In this sample, a camera flies around a 3-D scene with a tree-covered hill. The trees look like 3-D objects, but they are actually 2-D billboarded images that are rotated towards the eyepoint. The hilly terrain and the skybox, a six-sided cube containing sky textures, are objects loaded from .x files. They are used for visual effect and are unrelated to the billboarding technique.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Billboard

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The billboarding technique is the focus of this sample. The camera is moved in each frame, so the viewpoint changes accordingly. As the viewpoint changes, a rotation matrix is generated to rotate the billboards about the y-axis so that they face the new viewpoint. The computation of the billboard matrix occurs in the **FrameMove** function. The trees are also sorted in this function, as required for proper alpha blending, because billboards typically have some transparent pixels. The trees are rendered from a vertex buffer in the **DrawTrees** function.

The billboards in this sample are constrained to rotate about the y-axis only. Otherwise, the tree trunks would appear to not be fixed to the ground. For effects such as explosions in a 3-D flight simulator or space shooter, billboards are typically not constrained to one axis.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the source code in (SDK root)\Samples\Multimedia\VBSamples.

Microsoft DirectX 8.1 (C++)

BumpEarth Sample

[This is preliminary documentation and is subject to change.]

Description

The BumpEarth sample demonstrates the bump mapping capabilities of Microsoft® Direct3D®. Bump mapping is a texture-blending technique that renders the appearance of rough, bumpy surfaces. This sample renders a rotating, bump-mapped planet Earth.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\BumpEarth

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

BumpLens Sample

[This is preliminary documentation and is subject to change.]

Description

The BumpLens sample demonstrates a lens effect that can be achieved using bump mapping. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces, but can also be used for other effects, as shown here.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\BumpLens

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.

ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses bump mapping in a nontraditional fashion. Because bump mapping only perturbs an environment map, it can be used for other effects. In this case, it is used to perturb a background image, which can be rendered on the fly, to make a lens effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in the following directory: (SDK root) \Samples\Multimedia\Common

Microsoft DirectX 8.1 (C++)

BumpUnderwater Sample

[This is preliminary documentation and is subject to change.]

Description

The BumpUnderwater sample demonstrates an underwater effect that can be achieved using bump mapping. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces, but can also be used for other effects as shown here.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\BumpUnderWater

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.

ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses bump mapping in a nontraditional fashion. Because bump mapping only perturbs an environment map, it can be used for other effects. In this case, it is used to perturb a background image, which can be rendered on the fly, to make an underwater effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

BumpWaves Sample

[This is preliminary documentation and is subject to change.]

Description

The BumpWaves sample demonstrates the bump mapping capabilities of Microsoft® Direct3D®. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces. This sample renders a waterfront scene with only four triangles. The waves in the scene are completely fabricated with a bump map.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, refer to the Microsoft DirectX® SDK documentation.

This sample also uses a technique called projected textures, which is a texture-coordinate generation technique and is not the focal point of the sample. For more information on texture-coordinate generation, see Bump Mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\BumpWaves

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

In this sample, bump mapping is used to generate waves in a scene. The backdrop is used as a projective texture for the environment map, so it reflects in the waves. The waves themselves appear to be generated with many polygons. However, it is one large quad.

This sample uses common Microsoft DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Bump Self-Shadow Sample

[This is preliminary documentation and is subject to change.]

Description

This project includes all source and content for the self-shadowing bump-map algorithm presented at the 2001 Game Developers Conference (GDC). This application will run without pixel shaders, as long as the hardware has render targets and DOT3. However, it runs with better visual results and much more efficiently with pixel shaders.

Not all cards support all features for bump-mapping techniques. Some hardware has no, or limited, bump-mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\BumpSelfShadow

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
S	Zoom out
W	Zoom in
Arrow keys	Move camera
Number pad keys	Pitch camera (Num Lock should be on)
Number pad 7 and 9	Roll camera
2	Toggle self-shadowing
3	Toggle diffuse bump mapping
4	Reset position
5	Automatically rotate

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contains values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX software development kit (SDK). You can find the common headers and source code in (SDK root) \Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

ClipMirror Sample

[This is preliminary documentation and is subject to change.]

Description

The ClipMirror sample demonstrates the use of custom-defined clip planes. A 3-D scene is rendered normally, and then in a second pass as if reflected in a planar mirror. Clip planes are used to clip the reflected scene to the edges of the mirror.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\ClipMirror

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

The mouse is also used in this sample to control the viewing position.

Programming Notes

The main feature of this sample is the use of clip planes. The rectangular mirror has four edges, so four clip planes are used. Each plane is defined by the eyepoint and two vertices of one edge of the mirror. With the clip planes in place, the view matrix is reflected in the mirror's plane, and then the scene geometry (the teapot object) can be rendered as normal. Afterward, a semi-transparent rectangle is drawn to represent the mirror itself.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

CubeMap Sample

[This is preliminary documentation and is subject to change.]

Description

The CubeMap sample demonstrates an environment mapping technique called cube mapping. Environment mapping is a technique in which the environment surrounding a 3-D object, such as the lights, is put into a texture map, so that the object can have complex lighting effects without expensive lighting calculations.

Not all cards support all features for environment mapping techniques, such as cube mapping and projected textures. For more information on environment mapping, cube mapping, and projected textures, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\EnvMapping\CubeMap

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action

ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Cube mapping is a technique that employs a six-sided texture. To visualize the effects of this technique, imagine a wall-papered room in which the wallpaper has been shrink-wrapped around an object. Cube mapping is superior to sphere-mapping because the latter is inherently view-dependent; sphere maps are constructed for one particular viewpoint in mind. Cube maps also have no geometry distortions, so they can be generated on the fly using **IDirect3DDevice8::SetRenderTarget** for all six cube map faces.

Cube mapping works with Microsoft Direct3D® texture coordinate generation. By setting D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR, Direct3D generates cube map texture coordinates from the reflection vector for a vertex, thereby making this technique easy for environment mapping effects where the environment is reflected in the object.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Cull Sample

[This is preliminary documentation and is subject to change.]

Description

This sample illustrates how to cull objects whose object bounding box (OBB) does not intersect the viewing frustum. By not passing these objects to Microsoft® Direct3D®, you save the time that would be spent by Direct3D transforming and lighting these objects, which will never be visible. The time saved could be significant if there are many such objects or if the objects contain many vertices.

More elaborate and efficient culling can be done by creating hierarchies of objects, with bounding boxes around groups of objects, so that not every object's OBB has to be compared to the view frustum.

It is more efficient to do this OBB/frustum intersection calculation in your own code than to use Direct3D to transform the OBB and check the resulting clip flags.

You can adapt the culling routines in this sample meet the needs of programs that you write.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Cull

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

When you run this program, you'll see a scene with teapots rendered into two viewports. The right viewport uses the view frustum that the code will cull against. The left viewport has an independent camera, and shows the right viewport's frustum as a visible object, so you can see where culling should occur. Fifty teapots are randomly placed in the scene and they are rendered along with their semitransparent OBBs.

The teapots are colored as follows to indicate their cull status.

Color	Description	Cull status
Dark green	The object was quickly determined to be inside the frustum.	CS_INSIDE
Light green	The object was determined after some work to be inside the frustum.	CS_INSIDE_SLOW
Dark red	The object was quickly determined to be outside the frustum.	CS_OUTSIDE
Light red	The object was determined after some work to be outside the frustum	CS_OUTSIDE_SLOW

You should see only green teapots in the right window. Note that most teapots are either dark green or dark red, indicating that the slower tests are not needed for the majority of cases.

To move the camera for the right viewport, click on the right side of the window, then use the camera keys listed below to move around.

To move the camera for the left viewport, click on the left side of the window, then use the camera keys listed below to move around.

You can also rotate the teapots to set up particular relationships against the view frustum. You cannot move the teapots, but you can get the same effect by moving the frustum instead.

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
F1	Shows help or available commands
F2	Prompts the user to select a new rendering device or display mode
ALT+ENTER	Toggles between full-screen and windowed modes
ESC	Exits the application
W, S, ARROW KEYS	Moves the camera
Q, E, A, Z	Rotates the camera
Y, U, H, J	Rotates the teapots
N	Moves the left camera to match the right camera
M	Moves the right camera to its original position

Programming Notes

The OBB/viewport intersection algorithm used by this program is:

1. If any OBB corner point is inside the frustum, return CS_INSIDE.
2. Else if all OBB corner points are outside a single frustum plane, return CS_OUTSIDE.
3. Else if any frustum edge penetrates a face of the OBB, return CS_INSIDE_SLOW.
4. Else if any OBB edge penetrates a face of the frustum, return CS_INSIDE_SLOW.
5. Else if any point in the frustum is outside any plane of the OBB, return CS_OUTSIDE_SLOW.
6. Else return CS_INSIDE_SLOW.

The distinction between CS_INSIDE and CS_INSIDE_SLOW, and between CS_OUTSIDE and CS_OUTSIDE_SLOW, is provided here only for educational purposes. In a shipping application, you probably would combine the cull states into just CS_INSIDE and CS_OUTSIDE, because usually you need to know only whether the OBB is inside or outside the frustum.

The culling code shown here is written in a straightforward way for readability. It is not optimized for performance. Additional optimizations can be made, especially if the bounding box is a regular box (for example, the front and back faces are parallel). This algorithm could be generalized to work for arbitrary convex bounding hulls to allow tighter fitting against the underlying models.

This sample uses common DirectX code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

DolphinVS Sample

[This is preliminary documentation and is subject to change.]

Description

The DolphinVS sample shows an underwater scene of a dolphin swimming, with caustic effects on the dolphin and sea floor. The dolphin is animated using a technique called tweening. The underwater effect simply uses fog, and the water caustics use an animated set of textures. These effects are achieved using vertex shaders.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\DolphinVS

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.

SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Several things are happening in this sample.

- Two vertex shaders are used: one for the dolphin and one for the sea floor. The vertex shaders are assembly instructions which are assembled from two files, DolphinTween.vsh and SeaFloor.vsh.
- The dolphin is tweened—a form of morphing—by passing three versions of the dolphin down in multiple vertex streams. The vertex shader takes the weighted positions of each mesh and produces an output position. The vertex is then lit, texture coordinates are computed, and fog is added.
- The sea floor is handled similarly, just with no need for tweening. The caustic effects are added in a separate alpha-blended pass, using an animated set of 32 textures. The texture coordinates for the caustics are generated in the vertex shaders, stored as TEXCOORDINDEX stage 1.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

DotProduct3 Sample

[This is preliminary documentation and is subject to change.]

Description

The DotProduct3 sample demonstrates an alternative approach to Microsoft® Direct3D® bump mapping. This technique is named after the mathematical operation that combines a light vector with a surface normal. The normals for a surface are traditional (x,y,z) vectors stored in RGBA format in a texture map—called a normal map for this technique.

Not all cards support DotProduct3 blending texture stages, and not all cards support Direct3D bump mapping.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\BumpMapping\DotProduct3

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The lighting equation for simulating bump mapping involves using the dot product of the surface normal and the lighting vector. The lighting vector is passed into the texture factor, and the normals are encoded in a texture map. The blend stages look like this:

```
SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_DOTPRODUCT3 );
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
```

The next step is to store the normals in the texture. To do this, the components of a vector (XYZW) are each turned from a 32-bit floating value into a signed 8-bit integer and packed into a texture color (RGBA). The code shows how to do this using a custom-generated normal map, as well as one built from a bump mapping texture image.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, refer to the Microsoft® DirectX® SDK documentation.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

DXTex Tool

[This is preliminary documentation and is subject to change.]

Description

The DXTex tool enables Microsoft® DirectX® software development kit (SDK) users to create texture maps that use the new DXTn compression formats. Creating a DXTn-compressed texture is not difficult. However, Microsoft® Direct3D® can do the conversion for you when using the **IDirect3DTexture8** interface. Advanced developers will probably want to write their own tools that meet their specific needs, but the DXTex tool provides useful basic functionality.

Functionality

- Opens .bmp and Microsoft® DirectDraw® surface (.dds) files. For more information, see DDS File Format, below.
- Opens .bmp files as alpha channel, either explicitly or implicitly, using a <*filename*>_a.bmp naming convention.

- Saves textures in .dds format.
- Supports conversion to all five DXTn compression formats.
- Supports generation of mipmaps, using a box filter.
- Supports visualization of alpha channel as a grayscale image or through a user-selectable background color.
- Supports easy visual comparison of image quality between formats.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\DXTex

Executable: (SDK root)\bin\dxutils

User Interface

DxTex uses a fairly traditional user interface (UI) in that each texture map is a document, and several documents can be open at one time. However, each document can hold the texture in either one or two formats simultaneously while the document is open in DxTex. For example, you can import a .bmp file, which automatically creates a 32-bit ARGB texture. You can then choose to convert the texture to DXT1 format. The document now has the image open in both formats, and you can switch between the formats by clicking the window. You can also switch by clicking **Original View** or **New View** on the **View** menu. This makes it easy for you to observe any artifacts introduced by image compression, and to try different compression formats without progressive degradation of the image. For example, if this technique is not used and you convert an image from ARGB to DXT1, all but one bit of alpha is lost.

If you then decide to convert to DXT2, there are still only two alpha levels. Using the DxTex system, the second conversion is done from the original ARGB format, and the DXT2 image will contain all 16 levels of alpha supported by DXT2. When the image is saved, the original format is discarded and only the new format is stored.

Keep in mind the following when using the DxTex interface.

- If *no* format conversion has been requested since the document was opened, the one format is the *original* format—that is, the format in which the image will be written when the document is saved.
- If format conversion *has* been requested since the document was opened, the converted format is the *new* format—that is, the format in which the image will be written when the document is saved.
- If a second format conversion is requested, the new format from the first conversion is replaced with the second format. The original format will be unchanged.
- Generating mipmaps applies to both formats simultaneously.
- If the original format has multiple mipmaps and conversion to a new format is requested, the new format will automatically have multiple mipmaps as well.

Performance

DxTex uses the Direct3D Reference Rasterizer to draw the textures, regardless of what three-dimensional (3-D) hardware is available. So with larger textures (greater than 256-by-256 pixels), the application may be somewhat slow, depending on your CPU speed.

DDS File Format

See DDS File Format for more detailed information on the .dds file format.

Mipmaps

Mipmapping is a technique that improves image quality and reduces texture memory bandwidth by providing prefiltered versions of the texture image at multiple resolutions.

To generate mipmaps in DxTex, the width and height of the source image must both be powers of 2. To do this, go to the **Format** menu and click **Generate Mip Maps**. Filtering is done through a simple box filter. That is, the four nearest pixels are averaged to produce each destination pixel.

Alpha

Many texture formats include an alpha channel, which provides opacity information at each pixel. DxTex fully supports alpha in textures. When you import a .bmp file, if there is a file of the same size with a name that ends in "_a"—for example, Sample.bmp and Sample_a.bmp—the second file is loaded as an alpha channel. The blue channel of the second .bmp is stored in the alpha channel. Once a document is open, you can explicitly load a .bmp file as the alpha channel by clicking **Open As Alpha Channel** on the **File** menu.

To view the alpha channel directly, without the RGB channels, on the **View** menu, click **Alpha Channel Only**. The alpha channel appears as a grayscale image. If no alpha channel has been loaded, every pixel has a full alpha channel and the image appears solid white when viewing "Alpha Channel Only." To turn off alpha channel viewing, click the **Alpha Channel Only** command a second time.

In the usual view, the effect of the alpha channel is visible because the window has a solid background color that shows through the texture where the alpha channel is less than 100 percent. You can change this background color by clicking **Change Background Color** on the **View** menu. This choice does not affect the texture itself or the data that is written when the file is saved.

The DXT2 and DXT4 formats use premultiplied alpha. This means that the red, green, and blue values stored in the surface are already multiplied by the corresponding alpha value. Direct3D cannot copy from a surface that contains premultiplied alpha to one that contains non-premultiplied alpha, so some DxTex operations (Open as Alpha Channel, conversion to DXT3, and conversion to DXT5) are not possible on DXT2 and DXT4 formats. Supporting textures using these formats is difficult on Direct3D devices that do not support DXTn textures. This is because Direct3D cannot copy them to a traditional ARGB surface either, unless that ARGB surface uses premultiplied alpha as well, which is rare. For this reason, you might find it easier to use DXT3 rather than DXT2, and DXT5 rather than DXT4 when possible.

Command-Line Options

You can use command-line options to pass input files, an output file name, and processing options to DxTex. If an output file name is specified, the application exits automatically after writing the output file, and no user interface is presented.

```
dxtex [infilename] [-a alphaname] [-m] [DXT1|DXT2|DXT3|DXT4|DXT5] [outfilena

infilename:               Name of the file to load.  This can be a
                          .bmp or .dds file.

-a alphaname:             The next parameter is the name of a .bmp
                          file to load as the alpha channel. If no
```

```
                          alpha filename is specified, DxTex still
                          looks for a file named Infilename_a.bmp. If
                          it exists, use that file as the alpha
                          channel.

-m:                       Mipmaps are generated.

DXT1|DXT2|DXT3|DXT4|DXT5: Compression format. If no format is
                          specified, the image will be in ARGB-8888.

outfilename:              Name of the destination file. If this
                          option is not specified, the user interface
                          shows the current file and all requested
                          operations.  If an outfilename is specified,
                          the application exits after saving the
                          processed file without presenting a user
                          interface.
```

Microsoft DirectX 8.1 (C++)

Emboss Sample

[This is preliminary documentation and is subject to change.]

Description

The Emboss sample demonstrates an alternative approach to Microsoft® Direct3D® bump mapping. Embossing is done by subtracting the height map from itself and having texture coordinates that are slightly changed.

Not all cards support Direct3D bump mapping. Refer to the Microsoft DirectX® SDK documentation for more information.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Emboss

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-E	Turns emboss effect on/off.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

In this sample, bump mapping is used to emboss the surface of the tiger.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

EnhancedMesh Sample

[This is preliminary documentation and is subject to change.]

Description

The EnhancedMesh sample shows how to use Microsoft® Direct3DX to load and enhance a mesh. The mesh is enhanced by increasing the vertex count.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\EnhancedMesh

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

FishEye Sample

[This is preliminary documentation and is subject to change.]

Description

The FishEye sample shows a fish-eye lens effect that can be achieved using cube maps.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\EnvMapping\FishEye

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The FishEye sample scene is rendered into the surfaces of a cube map, rather than the back buffer. Then the cube map is used as an environment map to reflect the scene in some distorted geometry—in this case, some geometry approximating a fish eye lens. See the CubeMap Sample for more information about cube mapping.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Lighting Sample

[This is preliminary documentation and is subject to change.]

Description

The lighting sample show how to use Microsoft® Direct3D® lights when rendering. It shows the difference between the various types of lights—ambient, point, directional, and spot—how to configure these lights, and how to enable and disable them.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Lighting

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Direct3D lights can be used to shade and color objects that are rendered. They modify the color at each vertex of the primitives rendered while the lights are active. Note that the walls and floor of this sample contain many vertices, so the lighting is fairly detailed. If the vertices were only at the corners of the walls and floor, the lighting effects would be much rougher because of the reduced number of vertices. You can modify the m_m and m_n member variables in this program to see how the vertex count affects the lighting of the surfaces.

There are ways to generate lighting effects other than by using Direct3D lights. Techniques like light mapping can create lighting effects that are not limited to being computed only at each vertex. Vertex shaders can generate more unusual, realistic, or customized lighting at each vertex. Pixel shaders can perform lighting computation at each pixel of the polygons being rendered.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX software development kit (SDK). You can find the common headers and source code in (SDK root) \Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

MFCFog Sample

[This is preliminary documentation and is subject to change.]

Description

The MFCFog sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to control fog parameters for the 3-D scene.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\MFCFog

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls—sliders, radio buttons, and so on. You are encouraged to play with the controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and Direct3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

MFCPixelShader Sample

[This is preliminary documentation and is subject to change.]

Description

The MFCPixelShader sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to the control the pixel shader used for the 3-D scene.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\MFCPixelShader

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls—sliders, radio buttons, and so on. You are encouraged to play with controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and Direct3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

MFCTex Sample

[This is preliminary documentation and is subject to change.]

Description

The MFCTex sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to the control the texture appearance.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\MFCTex

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls—sliders, radio buttons, and so on. You are encouraged to play with controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and Direct3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Moire Sample

[This is preliminary documentation and is subject to change.]

Description

The moire sample shows how to use the Microsoft® DirectX® software development kit (SDK) screen saver framework to write a screen saver that uses Microsoft® Direct3D®. The screen saver framework is similar to the sample application framework, using many methods and variables with the same names. After writing a program with the screen saver framework, you end up with a fully-functional Microsoft® Windows® screen saver, rather than with a regular Windows application.

The moire screen saver appears as a mesmerizing sequence of spinning lines and colors. It uses texture transformation and alpha blending to create a highly animated scene, even though the polygons that make up the scene do not move at all.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\ScreenSavers\Moire

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

Moire.scr can be started in five modes: configuration, preview, full, test, and password-change. You can choose some modes by clicking the right mouse button (right-click) on the moire.scr file and choosing Configure or Preview. Or you can start moire.scr from the command line with the following command-line parameters:

-c Configuration mode
-t Test mode
-p Preview mode
-a Password-change mode
-s Full mode

When the screen saver is running in full mode, press any key or move the mouse to exit.

Programming Notes

Programs that use the screen saver framework are very similar to programs that use the Direct3D sample application framework. Each screen saver needs to create a class derived from the main application class, **CD3DScreensaver**. To provide functionality specific to each screen saver, the screen saver implements its own versions of the virtual functions **FrameMove**, **Render**, **InitDeviceObjects**, and so forth.

Screen savers can be written to be multimonitor-compatible, without much extra effort. If you do not want your screen saver to run on multiple monitors, you can just set the *m_bOneScreenOnly* variable to TRUE. This value is set to FALSE by default. The function **SetDevice** will be called each time the device changes. The way that moire deals with this is to create a structure called *DeviceObjects*, which contains all device-specific pointers and values. *CMoireScreensaver* holds an array of *DeviceObjects* structures, called *m_DeviceObjectsArray*. When **SetDevice** is called, *m_pDeviceObjects* is changed to point to the *DeviceObjects* structure for the specified device. When rendering, *m_rcRenderTotal* refers to the rendering area that spans all monitors, and *m_rcRenderCurDevice* refers to the rendering area for the current device's monitor. The function **SetProjectionMatrix** shows one way to set up a projection matrix that makes proper use of these variables to either render a scene that spans all the monitors, or display a copy of the scene on each monitor. The projection matrix used

depends on the value of *m_bAllScreensSame*, which you can enable the user to control in the configuration dialog.

The **ReadSettings** function is called by the screen saver framework at program startup time, to read various screen saver settings from the registry. **DoConfig** is called when the user wants to configure the screen saver settings. The program should respond to this by creating a dialog box with controls for the various screen saver settings. This dialog box should also have a button called **Display Settings** which, when pressed, should call **DoScreenSettingsDialog**. This common dialog box allows the user to configure what renderer and display mode should be used on each monitor. You should set the member variable *m_strRegPath* to a registry path that will hold the screen saver's settings. You can use this variable in your registry read/write functions. The screen saver framework will also use this variable to store information about the default display mode in some cases.

This sample uses common DirectX code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

OptimizedMesh Sample

[This is preliminary documentation and is subject to change.]

Description

The OptimizedMesh sample illustrates how to load and optimize a file-based mesh using the Microsoft® Direct3DX mesh utility functions.

For more information on Direct3DX, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\OptimizedMesh

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-O	Opens mesh file.
CTRL-M	Toggles optimized mesh.

Programming Notes

Many Microsoft Direct3D® samples in the DirectX SDK use file-based meshes. However, the OptimizedMesh sample is a good example of the basic code necessary for loading a mesh. The D3DX mesh loading functionality collapses the frame hierarchy of an .x file into one mesh.

For other samples, the bare bones D3DX mesh functionality is wrapped in a common class CD3DMesh. If you want to keep the frame hierarchy, you can use the common class CD3DFile.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Pick Sample

[This is preliminary documentation and is subject to change.]

Description

The Pick sample shows how to implement picking; that is, finding which triangle in a mesh is intersected by a ray. In this case, the ray comes from mouse coordinates.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Pick

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Use the mouse to pick any spot in the mesh to see that triangle.

Programming Notes

When you click the mouse, the code reads the screen coordinates of the cursor. These coordinates are converted, through the projection and view matrices, into a ray that goes from

the eyepoint through the point clicked on the screen and into the scene. This ray is passed to IntersectTriangle along with each triangle of the loaded model to determine which triangles, if any, are intersected by the ray. The texture coordinates of the intersected triangle is also determined.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

PointSprites Sample

[This is preliminary documentation and is subject to change.]

Description

The PointSprites sample shows how to use the new Microsoft® Direct3D® point sprites feature. A point sprite is simply a forward-facing, textured quad that is referenced only by (x,y,z) position coordinates. Point sprites are most often used for particle systems and related effects.

Not all cards support all features for point sprites. For more information on point sprites, refer to the Microsoft DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\PointSprites

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
F3	Animates the emitter.
F4	Changes color.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Without Direct3D support, point sprites can be implemented with four vertices that are oriented each frame towards the eyepoint, much like a billboard. With Direct3D, though, you can refer to each point sprite just by its center position and a radius. This saves processor computation time and bandwidth when uploading vertex information to the graphics card.

In this sample, each particle is implemented using multiple alpha-blended point sprites, giving the particle a motion-blur effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

ProgressiveMesh Sample

[This is preliminary documentation and is subject to change.]

Description

The ProgressiveMesh sample illustrates how to load and optimize a file-based mesh using the Microsoft® Direct3DX mesh utility functions. A progressive mesh is one in which the vertex information is stored internally in a special tree that can be accessed to render the mesh with any number of vertices. This procedure is fast, so progressive meshes are ideal for level-of-detail scenarios, where objects in the distance are rendered with fewer polygons.

For more information on Direct3DX, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\ProgressiveMesh

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
UP ARROW	Adds one vertex to the progressive mesh.

DOWN ARROW	Subtracts one vertex from the progressive mesh.
PAGE UP	Adds 100 vertices to the progressive mesh.
PAGE DOWN	Subtracts 100 vertices from the progressive mesh.
HOME	Displays all available vertices for the progressive mesh.
END	Displays the minimum vertices for the progressive mesh.

Programming Notes

Many Microsoft Direct3D® samples in the DirectX SDK use file-based meshes. However, the ProgressiveMesh sample is a good example of the basic code necessary for loading a mesh. The Direct3DX mesh loading functionality collapses the frame hierarchy of an .x file into one mesh.

The primary reason to use progressive meshes is the call to **ID3DXPMesh::SetNumVertices** for the mesh.

For other samples, the basic Direct3DX mesh functionality is wrapped in a common class CD3DMesh. To keep the frame hierarchy, you can use the common class CD3DFile.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

RTPatch Sample

[This is preliminary documentation and is subject to change.]

Description

The RTPatch sample shows how uses patches in Direct3D.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\RTPatch

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.

ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
UP ARROW	Increase number of patches per segment.
DOWN ARROW	Decrease number of patches per segment.
W	Toggles the wireframe.

Programming Notes

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

ShadowVolume Sample

[This is preliminary documentation and is subject to change.]

Description

The ShadowVolume sample uses stencil buffers to implement real-time shadows. In the sample, a complex object is rendered and used as a shadow caster to cast real-time shadows on itself and on the terrain below.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Not all cards support all features for stencil-buffer techniques. Some hardware has no, or limited, stencil-buffer support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\StencilBuffer\ShadowVolume

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.

ESC	Exits the application.

Programming Notes

Creating real-time shadows is a fairly advanced technique. In each frame, or as the geometry or lights in the scene are moved, an object called a shadow volume is computed. A shadow volume is a 3-D object that is the silhouette of the shadowcasting object, as protruded away from the light source.

In this sample, the 3-D object that casts shadows is a bi-plane. The silhouette of the plane is computed in each frame. This technique uses an edge-detection algorithm in which silhouette edges are found. This can be done because the normals of adjacent polygons will have opposing normals with respect to the light vector. The resulting edge list (the silhouette) is protruded into a 3-D object away from the light source. This 3-D object is known as the shadow volume, as every point inside the volume is inside a shadow.

Next, the shadow volume is rendered into the stencil buffer twice. First, only forward-facing polygons are rendered, and the stencil-buffer values are incremented each time. Then the back-facing polygons of the shadow volume are drawn, decrementing values in the stencil buffer. Normally, all incremented and decremented values cancel each other out. However, because the scene was already rendered with normal geometry, in this case the plane and the terrain, some pixels fail the zbuffer test as the shadow volume is rendered. Any values left in the stencil buffer correspond to pixels that are in the shadow.

Finally, these remaining stencil-buffer contents are used as a mask, as a large all-encompassing black quad is alpha-blended into the scene. With the stencil buffer as a mask, only pixels in shadow are darkened

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

SkinnedMesh Sample

[This is preliminary documentation and is subject to change.]

Description

The SkinnedMesh sample shows how to load and render a skinned mesh.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\SkinnedMesh

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the

same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

SphereMap Sample

[This is preliminary documentation and is subject to change.]

Description

The SphereMap sample demonstrates an environment mapping technique called sphere mapping. Environment mapping is a technique in which the environment surrounding a 3-D object, such as the lights, are put into a texture map, so that the object can have complex lighting effects without expensive lighting calculations.

Not all cards support all features for environment mapping techniques, such as cube mapping and projected textures. For more information on environment mapping, cube mapping, and projected textures, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\EnvMapping\SphereMap

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Sphere-mapping uses a precomputed (at model time) texture map that contains the entire environment as reflected by a chrome sphere. The idea is to consider each vertex, compute its normal, find where the normal matches on the chrome sphere, and then assign that texture coordinate to the vertex.

Although the math is not complicated, this involves computations for each vertex for every frame. Direct3D has a texture-coordinate generation feature that you can use to do this. The relevant render state operation is D3DTSS_TCI_CAMERASPACENORMAL, which takes the normal of the vertex in camera space and puts it through a texture transform to generate texture coordinates. Then you set up the texture matrix to do the rest. In this simple case, the matrix only has to scale and translate the texture coordinates to get from camera space (-1, +1) to texture space (0,1).

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

StencilDepth Sample

[This is preliminary documentation and is subject to change.]

Description

The StencilDepth sample uses stencil buffers to display the depth complexity of a scene. The depth complexity of a scene is defined as the average number of times each pixel is rendered.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\StencilBuffer\StencilDepth

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.

ESC	Exits the application.

Programming Notes

Displaying depth complexity is a valuable tool to analyze the performance of a scene. Scenes with high amounts of overdraw can benefit from some scene optimization such as sorting the geometry in a front-to-back order.

Not all cards support all features for stencil-buffer techniques. Some hardware has no, or limited, stencil buffer-support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

StencilMirror Sample

[This is preliminary documentation and is subject to change.]

Description

The StencilMirror sample uses stencil buffers to implement a mirror effect. In the sample, a watery terrain scene is rendered with the water reflecting a helicopter that flies above.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Not all cards support all features for stencil buffer-techniques. Some hardware has no, or limited, stencil buffer-support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\StencilBuffer\StencilMirror

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.

ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

In this sample, a stencil buffer is used to create the effect of a reflection coming off the water. The geometry of the water is rendered into the stencil buffer. Then the stencil buffer is used as a mask to render the scene again, this time with the geometry translated and rendered upside down, to appear as if it were reflected in the mirror.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Text3D Sample

[This is preliminary documentation and is subject to change.]

Description

The Text3D sample shows how to draw 2-D text in a 3-D scene. This is useful for displaying statistics or game menus, and so on.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Text3D

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses the common class, CD3DFont, to display 2-D text in a 3-D scene. The source code for the class is of most interest to this sample. The class uses GDI to load a font and output each letter to a bitmap. That bitmap, in turn, is used to create a texture.

When the CD3DFont class's **DrawText** function is called, a vertex buffer is filled with

polygons that are textured using the font texture created as mentioned above. The polygons may be drawn as a 2-D overlay—useful, for example, for printing statistics—or truly integrated in the 3-D scene.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

VertexBlend Sample

[This is preliminary documentation and is subject to change.]

Description

The VertexBlend sample demonstrates a technique called vertex blending, also known as surface skinning. It displays a file-based object that is made to bend in various spots.

Vertex blending is used for creating effects such as smooth joints and bulging muscles in character animations.

Not all cards support all features for vertex blending. For more information on vertex blending, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\VertexBlend

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-V	Switches between a custom vertex shader and Direct3D vertex blending.

Programming Notes

Vertex blending requires each vertex to have an associated blend weight. Multiple world transforms are set up using **SetTransformState** and the blend weights determine how much contribution each world matrix has when positioning each vertex.

In this sample, a mesh is loaded using the common helper code. Note how a custom vertex and a custom FVF is declared and used to build the mesh; see the **SetFVF** call for the mesh object. Without using the mesh helper code, the technique is the same. Create a vertex buffer full of vertices that have a blend weight, and use the appropriate FVF.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

VertexShader Sample

[This is preliminary documentation and is subject to change.]

Description

The VertexShader sample shows some effects that can be achieved using vertex shaders. Vertex shaders use a set of instructions, executed by the 3-D device on a per-vertex basis, that can affect the properties of the vertex—positions, normal, color, texture coordinates, and so on—in interesting ways.

Not all cards support all features for vertex shaders. For more information on vertex shaders, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\VertexShader

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Programming vertex shaders is not a trivial task. Refer to the Microsoft DirectX® SDK documentation for more information.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

VolumeTexture Sample

[This is preliminary documentation and is subject to change.]

Description

The VolumeTexture sample illustrates how to use the new volume textures in Microsoft® Direct3D®. Normally, a texture is thought of as a 2-D image, which has a width and a height and whose texels are addressed with two coordinate, tu and tv. Volume textures are the 3-D counterparts, with a width, height, and depth, are addressed with three coordinates, tu, tv, and tw.

You can use volume textures for interesting effects such as patchy fog, explosions, and so on.

Not all cards support all features for volume textures. For more information on volume textures, refer to the Microsoft DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\VolumeTexture

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Volume textures are no more difficult to use than 3-D textures. In this sample source code, note the vertex declaration, which has a third texture coordinate; texture creation, which also takes a depth dimension; and texture locking, again with the third dimension. The 3-D rasterizer interpolates texel values much as for 2-D textures.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Water Sample

[This is preliminary documentation and is subject to change.]

Description

The Water sample illustrates using Microsoft® Direct3DX techniques stored in shader files.

The sample shows a square pond inside a building, with rippling water effects including water caustics.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\Water

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
D	Starts a water drop.
PAGE DOWN	Displays next technique.
PAGE UP	Displays previous technique.
SHIFT+PAGE DOWN	Displays next technique with no validation.
SHIFT+PAGE UP	Displays previous technique with no validation.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The Direct3DX shaders technique is the focus of this sample.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

Microsoft DirectX 8.1 (C++)

Direct3D Appendix

[This is preliminary documentation and is subject to change.]

This section contains additional material that covers topics such as file formats and tips for performance improvements.

- DDS File Format

  This section explains the DDS file format in detail.

- Device States

  This section explains device states, which are used to set rendering and texturing attributes.

- Programming Tips

  These tips are derived from lessons learned from programming topics such as troubleshooting a program, implementing multithreading, or optimizing code for performance.

- X Files

  This section explains X files in depth, including their architecture, the file format, and some samples of file loading and saving.

- Mesh View Help

  This section outlines the functionality that is available in the mesh view executable. This handy executable can be used to experiment with meshes by applying different mesh utility operations. This application is part of the SDK install.

Microsoft DirectX 8.1 (C++)

DDS File Format

[This is preliminary documentation and is subject to change.]

The Microsoft® DirectDraw® Surface (DDS) file format is used to store textures and cubic environment maps, both with and without mipmap levels. This format can store uncompressed and compressed pixel formats, and is the preferred file format for storing DXTn compressed data. This file format is supported by the Microsoft® DirectX® Texture tool (DXTex Tool), as

well as some third party tools, and by the Microsoft Direct3DX library.

This format was introduced with DirectX 7. In DirectX 8, support for volume textures was added.

File Layout

The basic structure of a DDS file is a header, and one or more surfaces written to a binary file. The header consists of a four-character code and a DDSURFACEDESC2 structure. This header contains all the information needed to determine the contents of the entire file. The image below shows the layout of a DDS file.

Surface Format Header

The DDSURFACEDESC2 structure describes the file contents using the standard flags and values defined in the DirectDraw documentation, but files should use a restricted set of values to ensure full compatibility. A robust reader should verify key values and a robust writer should ensure all required flags are set for the different fields and options to ensure easy loading by the application or other tool. Also, a robust reader should not use a field unless the corresponding flag is set, and a robust writer should set all undefined fields to 0.

The table below shows the members of the DDSURFACEDESC2 structure.

Member	Description
DWORD dwSize	Size of structure. This member must be set to 124.
DWORD dwFlags	Flags to indicate valid fields. Always include DDSD_CAPS, DDSD_PIXELFORMAT, DDSD_WIDTH, DDSD_HEIGHT and either DDSD_PITCH or DDSD_LINEARSIZE.
DWORD dwHeight	Height of the main image in pixels
DWORD dwWidth	Width of the main image in pixels
DWORD dwPitchOrLinearSize	For uncompressed formats, this is the number of bytes per scan line (DWORD aligned) for the main image. *dwFlags* should include DDSD_PITCH in this case. For compressed formats, this is the total number of bytes for the main image. *dwFlags* should be include DDSD_LINEARSIZE in this case.
DWORD dwDepth	For volume textures, this is the depth of the volume. *dwFlags* should include DDSD_DEPTH in this case.
DWORD dwMipMapCount	For items with mipmap levels, this is the total number of levels in the mipmap chain of the main image. *dwFlags* should include DDSD_MIPMAPCOUNT in this case
DWORD dwReserved1 [11]	
DDPIXELFORMAT ddpfPixelFormat	A 32-byte value that specifies the pixel format structure.
DDCAPS2 ddsCaps	A 16-byte value that specifies the capabilities structure.
DWORD dwReserved2	

Note that the field names used in this description do not correspond exactly with the DDSURFACEDESC2 fields due to the restrictions placed on the DDS file format, but a standard DDSURFACEDESC2 structure can and should be used. Use the magic DDS value and the header *dwSize* value to validate the file.

The pixel format of the image is given in the *ddpfPixelFormat* field of the header, and can describe all the formats supported by Microsoft® Direct3D®. DDS files are usually restricted to either RGB or FOURCC formats, and the use of other formats is not generally supported. Restrict usage to RGB formats A8R8G8B8, A1R5G5B5, A4R4G4B4, R8G8B8, R5G6B5; and FOURCC formats DXT1, DXT2, DXT3, DXT4, and DXT5 to ensure the best support. A well-written reader should handle more formats, if possible.

The following table shows the layout of the DDPIXELFORMAT structure.

Member	Description
DWORD dwSize	Size of structure. This member must be set to 32.
DWORD dwFlags	Flags to indicate valid fields. Uncompressed formats will usually use DDPF_RGB to indicate an RGB format, while compressed formats will use DDPF_FOURCC with a four-character code.
DWORD dwFourCC	This is the four-character code for compressed formats. *dwFlags* should include DDPF_FOURCC in this case. For DXTn compression, this is set to "DXT1", "DXT2", "DXT3", "DXT4", or "DXT5".
DWORD dwRGBBitCount	For RGB formats, this is the total number of bits in the format. *dwFlags* should include DDPF_RGB in this case. This value is usually 16, 24, or 32. For A8R8G8B8, this value would be 32.
DWORD dwRBitMask DWORD dwGBitMask DWORD dwBBitMask	For RGB formats, this contains the masks for the red, green, and blue channels. For A8R8G8B8, these values would be 0x00ff0000, 0x0000ff00, and 0x000000ff respectively.
DWORD dwRGBAlphaBitMask	For RGB formats, this contains the mask for the alpha channel, if any. dwFlags should include DDPF_ALPHAPIXELS in this case. For A8R8G8B8, this value would be 0xff000000.

The final details of the format are inferred from the capabilities bits set in the *ddsCaps* field of the header. The layout of the *ddsCaps* structure is shown in the table below.

Member	Description
DWORD dwCaps1	DDS files should always include DDSCAPS_TEXTURE. If the file contains mipmaps, DDSCAPS_MIPMAP should be set. For any DDS file with more than one main surface, such as a mipmaps, cubic environment map, or volume texture, DDSCAPS_COMPLEX should also be set.
DWORD dwCaps2	For cubic environment maps, DDSCAPS2_CUBEMAP should be included as well as one or more faces of the map (DDSCAPS2_CUBEMAP_POSITIVEX, DDSCAPS2_CUBEMAP_NEGATIVEX, DDSCAPS2_CUBEMAP_POSITIVEY, DDSCAPS2_CUBEMAP_NEGATIVEY, DDSCAPS2_CUBEMAP_POSITIVEZ, DDSCAPS2_CUBEMAP_NEGATIVEZ). For volume textures, DDSCAPS2_VOLUME should be included.
DWORD Reserved [2]	

Note that as of DirectX 8, cubic environment maps are always written with all faces defined.

The following topics show the layout.

- DDS File Layout for Textures
- DDS File Layout for Cubic Environment Maps
- DDS File Layout for Volume Textures
- DDS Bit Flag Values

Microsoft DirectX 8.1 (C++)

DDS File Layout for Textures

[This is preliminary documentation and is subject to change.]

For simple uncompressed textures, the file should contain DDSD_PITCH and a DDPF_RGB pixel format surface. The pitch value should be DWORD-aligned. In this case, the main image should be (*dwPitchOrLinearSize* × *dwHeight*) bytes total. If the file also contains mipmaps, the DDSD_MIPMAPCOUNT, DDSCAPS_MIPMAP, and DDSCAPS_COMPLEX flags should be set, and the dwMipMapCount field should contain the total number of levels defined in the file (for a 256-by-256 image, this field would be 9). The pitch for each mipmap level image is computed from *dwWidth* and the pixel format, which is then DWORD aligned. For example, a 256-by-256 main image with a pixel format of R8G8B8 (three bytes per pixel) and all mipmap levels would contain the following:

"DDS ", (256, 256), (pitch=768), (pixel format=R8G8B8, bitcount=24), (mipmapcount=9), (TEXTURE, PITCH, COMPLEX, MIPMAP, RGB)	128 bytes
256 x 256 main image.	196608 bytes
128 x 128 mipmap image.	49152 bytes
64 x 64 mipmap image.	12288 bytes
32 x 32 mipmap image.	3072 bytes
16 x 16 mipmap image.	768 bytes
8 x 8 mipmap image.	192 bytes
4 x 4 mipmap image.	48 bytes
2 x 2 mipmap image.	12 bytes
1 x 1 mipmap image.	4 bytes

For simple compressed textures, the file should contain DDSD_LINEARSIZE and a DDPF_FOURCC pixel format surface. In this case, the image should be *dwPitchOrLinearSize* bytes total. If the file contains mipmaps, the DDSD_MIPMAPCOUNT, DDSCAPS_COMPLEX, DDSCAPS_MIPMAP, and *dwMipMapCount* fields should be set. For DXTn compressed formats, the size of each mipmap level image is one-fourth the size of

the previous, with a minimum of 8 (DXT1) or 16 (DXT2-5) bytes (this is only true for square textures). For example, a 256-by-256 main image with a pixel format of DXT1 and all mipmap levels would contain the following:

"DDS ", (256, 256), (linearsize=32768), (pixel format=DXT1), (mipmapcount=9), (TEXTURE, LINEARSIZE, COMPLEX, MIPMAP, FOURCC)	128 bytes
256 x 256 main image.	32768 bytes
128 x 128 mipmap image.	8192 bytes
64 x 64 mipmap image.	2048 bytes
32 x 32 mipmap image.	512 bytes
16 x 16 mipmap image.	128 bytes
8 x 8 mipmap image.	32 bytes
4 x 4 mipmap image.	8 bytes
2 x 2 mipmap image.	8 bytes
1 x 1 mipmap image.	8 bytes

If mipmaps are generated, all levels down to 1-by-1 are usually written. Some tools may not support partial mipmap chains.

When computing DXTn compressed sizes for non-square textures, the following formula should be used at each mipmap level:

max(1, *width* ÷ 4) × **max**(1, *height* ÷ 4) × 8 (DXT1) or 16 (DXT2-5)

A 256-by-64 main image with a pixel format of DXT1 and all mipmap levels would contain the following:

"DDS ", (256, 64), (linearsize=8192), (pixel format=DXT1), (mipmapcount=9), (TEXTURE, LINEARSIZE, COMPLEX, MIPMAP, FOURCC)	128 bytes
256 x 64 main image.	8192 bytes
128 x 32 mipmap image.	2048 bytes
64 x 16 mipmap image.	512 bytes
32 x 8 mipmap image.	128 bytes
16 x 4 mipmap image.	32 bytes
8 x 2 mipmap image.	16 bytes
4 x 1 mipmap image.	8 bytes
2 x 1 mipmap image.	8 bytes
1 x 1 mipmap image.	8 bytes

Microsoft DirectX 8.1 (C++)

DDS File Layout for Cubic Environment Maps

[This is preliminary documentation and is subject to change.]

For cubic environment maps, one or more faces of a cube are written to the file using either uncompressed or compressed formats, and all faces must be the same size. Each face can have mipmaps defined, although all faces must have the same number of mipmap levels. If a file contains a cube map, DDSCAPS_COMPLEX, DDSCAPS2_CUBEMAP, and one or more of DSCAPS2_CUBEMAP_POSITIVEX/Y/Z and/or DDSCAPS2_CUBEMAP_NEGATIVEX/Y/Z should be set. The faces are written in the order POSITIVEX, NEGATIVEX, POSITIVEY, NEGATIVEY, POSITIVEZ, NEGATIVEZ, with any missing faces omitted. Each face is written with its main image, followed by any mipmap levels.

For example, a 256-by-256 cube map with Positive X, Negative Y, and Positive Z faces, a pixel format of DXT1, and all mipmap levels would contain the following:

"DDS ", (256, 256), (linearsize=32768), (pixel format=DXT1), (mipmapcount=9), (TEXTURE, LINEARSIZE, COMPLEX, MIPMAP, CUBEMAP, POSITIVEX, NEGATIVEY, POSITIVEZ, FOURCC)	128 bytes
256 x 256 Positive X main image.	32768 bytes
128 x 128 Positive X mipmap image.	8192 bytes
64 x 64 Positive X mipmap image.	2048 bytes
32 x 32 Positive X mipmap image.	512 bytes
16 x 16 Positive X mipmap image.	128 bytes
8 x 8 Positive X mipmap image.	32 bytes
4 x 4 Positive X mipmap image.	8 bytes
2 x 2 Positive X mipmap image.	8 bytes
1 x 1 Positive X mipmap image.	8 bytes
256 x 256 Negative Y main image.	32768 bytes
128 x 128 Negative Y mipmap image.	8192 bytes
64 x 64 Negative Y mipmap image.	2048 bytes
32 x 32 Negative Y mipmap image.	512 bytes
16 x 16 Negative Y mipmap image.	128 bytes
8 x 8 Negative Y mipmap image.	32 bytes
4 x 4 Negative Y mipmap image.	8 bytes
2 x 2 Negative Y mipmap image.	8 bytes
1 x 1 Negative Y mipmap image.	8 bytes
256 x 256 Positive Z main image.	32768 bytes
128 x 128 Positive Z mipmap image.	8192 bytes
64 x 64 Positive Z mipmap image.	2048 bytes
32 x 32 Positive Z mipmap image.	512 bytes

16 x 16 Positive Z mipmap image.	128 bytes
8 x 8 Positive Z mipmap image.	32 bytes
4 x 4 Positive Z mipmap image.	8 bytes
2 x 2 Positive Z mipmap image.	8 bytes
1 x 1 Positive Z mipmap image.	8 bytes

Note that as of Microsoft® DirectX® 8, cube maps are always written with all faces defined.

Microsoft DirectX 8.1 (C++)

DDS File Layout for Volume Textures

[This is preliminary documentation and is subject to change.]

A volume texture is an extension of the standard texture for Microsoft® DirectX® 8, and can be defined with or without mipmaps. If a file contains a volume texture, DDSCAPS_COMPLEX, DDSCAPS2_VOLUME, DDSD_DEPTH, and *dwDepth* should be set. For volumes without mipmaps, each depth slice is written to the file in order. If mipmaps are included, all depth slices for a given mipmap level are written together, with each level containing half as many slices as the previous level with a minimum of 1. Volume textures do not support DXTn compression as of DirectX 8.

For example, a 64 x 64 x 4 volume map, a pixel format of R8G8B8 (3 bytes per pixel), and all mipmap levels would contain:

"DDS ", (64, 64, 4), (pitch=768), (pixel format=R8G8B8, bitcount=24), (mipmapcount=7), (DEPTH, TEXTURE, PITCH, COMPLEX, MIPMAP, VOLUME, RGB)	128 bytes
64 x 64 slice 1 of 4 main image.	12288 bytes
64 x 64 slice 2 of 4 main image.	12288 bytes
64 x 64 slice 3 of 4 main image.	12288 bytes
64 x 64 slice 4 of 4 main image.	12288 bytes
32 x 32 slice 1 of 2 mipmap image.	3072 bytes
32 x 32 slice 2 of 2 mipmap image.	3072 bytes
16 x 16 slice 1 of 1 mipmap image.	768 bytes
8 x 8 slice 1 of 1 mipmap image.	192 bytes
4 x 4 slice 1 of 1 mipmap image.	48 bytes
2 x 2 slice 1 of 1 mipmap image.	12 bytes
1 x 1 slice 1 of 1 mipmap image.	4 bytes

Microsoft DirectX 8.1 (C++)

DDS Bit Flag Values

[This is preliminary documentation and is subject to change.]

A number of fields make use of the standard Microsoft® DirectDraw® bit flags. Those that are used for DDS files are included here for reference.

The *dwFlags* member of the original **DDSURFACEDESC2** structure can be set to one or more of the following values.

The *dwFlags* member of the modified **DDSURFACEDESC2** structure can be set to one or more of the following values.

Flag	Value
DDSD_CAPS	0x00000001
DDSD_HEIGHT	0x00000002
DDSD_WIDTH	0x00000004
DDSD_PITCH	0x00000008
DDSD_PIXELFORMAT	0x00001000
DDSD_MIPMAPCOUNT	0x00020000
DDSD_LINEARSIZE	0x00080000
DDSD_DEPTH	0x00800000

Flag	Value
DDPF_ALPHAPIXELS	0x00000001
DDPF_FOURCC	0x00000004
DDPF_RGB	0x00000040

The *dwCaps1* member of the **DDSCAPS2** structure can be set to one or more of the following values.

Flag	Value
DDSCAPS_COMPLEX	0x00000008
DDSCAPS_TEXTURE	0x00001000
DDSCAPS_MIPMAP	0x00400000

The *dwCaps2* member of the **DDSCAPS2** structure can be set to one or more of the following values.

Flag	Value
DDSCAPS2_CUBEMAP	0x00000200
DDSCAPS2_CUBEMAP_POSITIVEX	0x00000400
DDSCAPS2_CUBEMAP_NEGATIVEX	0x00000800
DDSCAPS2_CUBEMAP_POSITIVEY	0x00001000
DDSCAPS2_CUBEMAP_NEGATIVEY	0x00002000
DDSCAPS2_CUBEMAP_POSITIVEZ	0x00004000
DDSCAPS2_CUBEMAP_NEGATIVEZ	0x00008000
DDSCAPS2_VOLUME	0x00200000

Microsoft DirectX 8.1 (C++)

Device States

[This is preliminary documentation and is subject to change.]

A Microsoft® Direct3D® device is a state computer. Applications set up the state of the lighting, rendering, and transformation modules and then pass data through them during rendering.

This section describes render states, provides details about each render state used in Direct3D, and contains information about device state blocks, which applications can use to manipulate groups of devices states in a single call. The following topics are discussed.

- Render States
- Texture Stage States
- State Blocks

Microsoft DirectX 8.1 (C++)

Alpha Blending State

[This is preliminary documentation and is 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 Texture Blending.

Applications written in C++ use the D3DRS_ALPHABLENDENABLE render state to enable alpha transparency blending. The Microsoft® Direct3D® API allows many types of alpha blending. However, it is important to note the user's 3-D hardware might not support all the blending states allowed by Direct3D.

The type of alpha blending that is done depends on the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Source and destination blend states are used in pairs. The following code example demonstrates how the source blend state is set to D3DBLEND_SRCCOLOR and the destination blend state is set to D3DBLEND_INVSRCCOLOR.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Set the source blend state.
d3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);

// Set the destination blend state.
d3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR);
```

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

Altering the source and destination blend states can give the appearance of emissive objects in a foggy or dusty atmosphere. For instance, if your application models flames, force fields,

plasma beams, or similarly radiant objects in a foggy environment, set the source and destination blend states to D3DBLEND_ONE.

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

You can achieve color light mapping by setting the source alpha blending state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCCOLOR.

Microsoft DirectX 8.1 (C++)

Alpha Testing State

[This is preliminary documentation and is subject to change.]

C++ applications can use alpha testing to control when pixels are written to the render-target surface. By using the D3DRS_ALPHATESTENABLE render state, your application sets the current Microsoft® Direct3D® device so that it tests each pixel according to an alpha test function. If the test succeeds, the pixel is written to the surface. If it does not, Direct3D ignores the pixel. Select the alpha test function with the D3DRS_ALPHAFUNC render state. Your application can set a reference alpha value for all pixels to compare against by using the D3DRS_ALPHAREF render state.

The most common use for alpha testing is to improve performance when rasterizing objects that are nearly transparent. If the color data being rasterized is more opaque than the color at a given pixel (D3DPCMPCAPS_GREATEREQUAL), then the pixel is written. Otherwise, the rasterizer ignores the pixel altogether, saving the processing required to blend the two colors. The following code example checks if a given comparison function is supported and, if so, it sets the comparison function parameters required to improve performance during rendering.

```
// This code example assumes that pCaps is a
// D3DCAPS8 structure that was filled with a
// previous call to IDirect3D8::GetDeviceCaps.
if (pCaps.AlphaCmpCaps & D3DPCMPCAPS_GREATEREQUAL)
{
    dev->SetRenderState(D3DRS_ALPHAREF, (DWORD)0x00000001);
    dev->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
    dev->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
}

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

Not all hardware supports all alpha-testing features. You can check the device capabilities by calling the **IDirect3D8::GetDeviceCaps** method. After retrieving the device capabilities, check the **AlphaCmpCaps** member of the associated **D3DCAPS8** structure for the desired comparison function. If the **AlphaCmpCaps** member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests.

Microsoft DirectX 8.1 (C++)

Ambient Lighting State

[This is preliminary documentation and is subject to change.]

Ambient light is surrounding light that radiates from all directions.

For specific information on how Microsoft® Direct3D® uses ambient light, see Direct Light vs. Ambient Light, and Mathematics of Lighting.

A C++ application sets the color of ambient lighting by invoking the **IDirect3DDevice8::SetRenderState** method and passing the enumerated value D3DRS_AMBIENT as the first parameter. The second parameter is a color value. The default value is zero.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Set the ambient light.
d3dDevice->SetRenderState(D3DRS_AMBIENT, 0x00202020);
```

Microsoft DirectX 8.1 (C++)

Antialiasing State

[This is preliminary documentation and is subject to change.]

Antialiasing is a method of making lines and edges appear smoother on the screen. Microsoft® Direct3D® supports two antialiasing methods: edge antialiasing and full-scene antialiasing.

For details about these techniques, see Antialiasing.

By default, Direct3D does not perform antialiasing. To enable edge-antialiasing, which requires a second rendering pass, set the D3DRS_EDGEANTIALIAS render state to TRUE. To disable it, set D3DRS_EDGEANTIALIAS to FALSE.

To enable full-scene antialiasing, set the D3DRS_MULTISAMPLEANTIALIAS render state to TRUE. To disable it, set D3DRS_MULTISAMPLEANTIALIAS to FALSE.

Microsoft DirectX 8.1 (C++)

Culling State

[This is preliminary documentation and is subject to change.]

As Microsoft® Direct3D® renders primitives, it culls primitives that are facing away from the user.

C++ applications set the culling mode by using the D3DRS_CULLMODE render state, which can be set to a member of the **D3DCULL** enumerated type. By default, Direct3D culls back

faces with counterclockwise vertices.

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

```
// This code example assumes that d3dDevice is a valid
// pointer to an IDirect3DDevice8 interface.

// Set the culling state.
d3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
```

Microsoft DirectX 8.1 (C++)

Depth Buffering State

[This is preliminary documentation and is subject to change.]

Depth buffering is a method of removing hidden lines and surfaces. By default, Microsoft® Direct3D® does not use depth buffering.

For a conceptual overview of depth buffers, see Depth Buffers.

C++ applications update the depth-buffering state with the D3DRS_ZENABLE render state, using a member of the **D3DZBUFFERTYPE** enumeration to specify the new state value.

If your application needs to prevent Direct3D from writing to the depth buffer, it can use the D3DRS_ZWRITEENABLE enumerated value, specifying D3DZB_FALSE as the second parameter for the call to **IDirect3DDevice8::SetRenderState**.

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

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Enable z-buffering.
d3dDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Your application can also use the D3DRS_ZFUNC render state to control the comparison function that Direct3D uses when performing depth buffering.

Z-biasing is a method of displaying one surface in front of another, even if their depth values are the same. You can use this technique for a variety of effects. A common example is rendering shadows on walls. Both the shadow and the wall have the same depth value. However, you want your application to show the shadow on the wall. Giving a z-bias to the shadow makes Direct3D display them properly (see D3DRS_ZBIAS).

Microsoft DirectX 8.1 (C++)

Fog State

[This is preliminary documentation and is subject to change.]

Fog effects can give a 3-D scene greater realism. You can use fog effects for more than simulating fog. They can also decrease the clarity of a scene with distance. This mirrors what happens in the real world; as objects become more distant from the user, their detail is less distinct.

For more information about using fog in your application, see Fog.

A C++ application controls fog through device rendering states. The **D3DRENDERSTATETYPE** enumerated type includes states to control whether pixel (table) or vertex fog is used, what color it is, the fog formula the system applies, and the parameters of the formula.

You enable fog by setting the D3DRS_FOGENABLE render state to TRUE. The fog color can be set to any color value by using the D3DRS_FOGCOLOR render state; the alpha component of the fog color is ignored, .

The D3DRS_FOGTABLEMODE and D3DRS_FOGVERTEXMODE render states control the fog formula applied for fog calculations, and they indirectly control which type of fog is applied. Both render states can be set to a member of the **D3DFOGMODE** enumerated type. Setting either render state to D3DFOG_NONE disables pixel or vertex fog, respectively. If both render states are set to valid modes, the system applies only pixel fog effects.

The D3DRS_FOGSTART and D3DRS_FOGEND render states control fog formula parameters for the D3DFOG_LINEAR mode. The D3DRS_FOGDENSITY render state controls fog density in the exponential fog modes.

For more information, see Fog Parameters.

Microsoft DirectX 8.1 (C++)

Lighting State

[This is preliminary documentation and is subject to change.]

Applications that use the Microsoft® Direct3D® geometry pipeline can enable or disable lighting calculations. Only vertices that contain a vertex normal are properly lit; vertices with no normal will use a dot product of zero in all lighting computations. Therefore, a vertex that does not use a normal receives no light.

For more information, see Mathematics of Lighting.

Applications enable Direct3D lighting by setting the D3DRS_LIGHTING render state to TRUE, which is the default setting, and they disable Direct3D lighting by setting the render state to FALSE.

The lighting render state is entirely independent of lighting computations that can be performed on vertices within a vertex buffer. The **IDirect3DDevice8::ProcessVertices** method accepts its own flags to control lighting calculations during vertex processing.

Microsoft DirectX 8.1 (C++)

Outline and Fill State

[This is preliminary documentation and is 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. You can select the method to fill them by specifying a value defined by the **D3DFILLMODE** enumerated type for the D3DRS_FILLMODE render state.

To enable dithering, your application must pass the D3DRS_DITHERENABLE enumerated value as the first parameter to **IDirect3DDevice8::SetRenderState**. It must set the second parameter to TRUE to enable dithering, and FALSE to disable it.

At times, drawing the last pixel in a line can cause unsightly overlap with surrounding primitives. You can control this using the D3DRS_LASTPIXEL enumerated value. However, do not alter this setting without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.

By default, Microsoft® Direct3D® devices use a solid outline for primitives. The outline pattern can be changed using the **D3DLINEPATTERN** structure. For details, see the D3DRS_LINEPATTERN render state.

Microsoft DirectX 8.1 (C++)

Per-Vertex Color State

[This is preliminary documentation and is subject to change.]

When using flexible vertex format (FVF) codes, vertices can contain both vertex color and vertex normal information. By default, Microsoft® Direct3D® uses this information when it calculates lighting. To disable use of vertex color lighting information, invoke the **IDirect3DDevice8::SetRenderState** method and pass D3DRS_COLORVERTEX as the first parameter. Set the second parameter to FALSE to disable vertex color lighting, or TRUE to enable it.

If per-vertex color is enabled, applications can configure the source from which the system retrieves color information for a vertex. The D3DRS_AMBIENTMATERIALSOURCE, D3DRS_DIFFUSEMATERIALSOURCE, D3DRS_EMISSIVEMATERIALSOURCE, and D3DRS_SPECULARMATERIALSOURCE render states control the ambient, diffuse, emissive, and specular color component sources, respectively. Each state can be set to members of the **D3DMATERIALCOLORSOURCE** enumerated type, which defines constants that instruct the system to use the current material, diffuse color, or specular color as the source for the specified color component.

Microsoft DirectX 8.1 (C++)

Primitive Clipping State

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® can clip primitives that render partially outside the viewport. When

using C++, Direct3D clipping is controlled by the D3DRS_CLIPPING render state. Set this render state to TRUE (the default value) to enable primitive clipping. Set it to FALSE to disable Direct3D clipping services.

The primitive clipping render state is entirely independent of clipping operations that can be performed on vertices within a vertex buffer. The **IDirect3DDevice8::ProcessVertices** method accepts its own flags to control primitive clipping during vertex processing.

Microsoft DirectX 8.1 (C++)

Shading State

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® supports both flat and Gouraud shading. The default is Gouraud shading. To control the current shading mode, your C++ application specifies a member of the **D3DSHADEMODE** enumerated type for the D3DRS_SHADEMODE render state.

The following C++ code example demonstrates the process of setting the shading state to flat shading mode.

```
// This code example assumes that d3dDevice is a
// valid pointer to a IDirect3DDevice8 interface.

// Set the shading state.
d3dDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
```

Microsoft DirectX 8.1 (C++)

Stencil Buffer State

[This is preliminary documentation and is subject to change.]

Applications use the stencil buffer to determine whether a pixel is written to the rendering target surface.

For details, see Stencil Buffer Techniques.

Applications written in C++ enable or disable stenciling by calling the **IDirect3DDevice8::SetRenderState** method. Pass D3DRS_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to TRUE or FALSE to enable or disable stenciling, respectively.

Set the comparison function that Microsoft® Direct3D® uses to perform the stencil test by calling **SetRenderState**. Set the value of the first parameter to D3DRS_STENCILFUNC. Pass a member of the **D3DCMPFUNC** enumerated type as the value of the second parameter.

The stencil reference value is the value in the stencil buffer that the stencil function uses for its test. By default, the stencil reference value is zero. Your application can set the value by calling **SetRenderState**. Pass D3DRS_STENCILREF as the value of the first parameter. Set the value of the second parameter to the new reference value.

Before the Direct3D module performs the stencil test for any pixel, it performs a bitwise **AND** operation of the stencil reference value and a stencil mask value. The result is compared against the contents of the stencil buffer using the stencil comparison function. Your application can set the stencil mask by calling **SetRenderState**. Pass D3DRS_STENCILMASK as the value of the first parameter. Set the value of the second parameter to the new stencil mask.

To set the action that Direct3D takes when the stencil test fails, call **SetRenderState** and pass D3DRS_STENCILFAIL as the first parameter. The second parameter must be a member of the **D3DSTENCILOP** enumerated type.

Your application can also control how Direct3D responds when the stencil test passes but the z-buffer test fails. Call **SetRenderState** and pass D3DRS_STENCILZFAIL as the first parameter and use a member of the **D3DSTENCILOP** enumerated type for the second parameter.

In addition, your application can control what Direct3D does when both the stencil test and the z-buffer test pass. Call **SetRenderState** and pass D3DRS_STENCILPASS as the first parameter. Again, the second parameter must be a member of the **D3DSTENCILOP** enumerated type.

Microsoft DirectX 8.1 (C++)

Texture Wrapping State

[This is preliminary documentation and is subject to change.]

The D3DRS_WRAP0 through D3DRS_WRAP7 render states enable and disable u- and v-wrapping for various textures in the device's multitexture cascade. You can set these render states to a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags to enable wrapping in first, second, third, and fourth directions of the texture. Use a value of zero to disable wrapping altogether. Texture wrapping is disabled in all directions for all texture stages by default.

For a conceptual overview, see Texture Wrapping.

Microsoft DirectX 8.1 (C++)

Texture Stage States

[This is preliminary documentation and is subject to change.]

Texture stage states control the style of texturing and how texture filtering is done.

Applications written in C++ control the characteristics of the texture-related render states by invoking the **IDirect3DDevice8::SetTextureStageState** method. The **D3DTEXTURESTAGESTATETYPE** enumerated type specifies all the possible texture-related rendering states. Your application passes a value from the **D3DTEXTURESTAGESTATETYPE** enumeration as the first parameter to the **SetTextureStageState** method.

Applications set the texture for a stage by calling the **IDirect3DDevice8::SetTexture** method.

Additional information is contained in the following topics.

- Texture Addressing State
- Texture Border State
- Texture Filtering State
- Texture Blending State

Microsoft DirectX 8.1 (C++)

Texture Addressing State

[This is preliminary documentation and is subject to change.]

C++ applications use the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, and D3DTSS_ADDRESSW texture stage states to set the texture addressing.

Microsoft DirectX 8.1 (C++)

Texture Border State

[This is preliminary documentation and is subject to change.]

C++ applications use the D3DTSS_BORDERCOLOR texture stage state to set border color texture addressing.

For more information, see Border Color Texture Address Mode.

Microsoft DirectX 8.1 (C++)

Texture Filtering State

[This is preliminary documentation and is subject to change.]

C++ applications use the D3DTSS_MAGFILTER, D3DTSS_MINFILTER, and D3DTSS_MIPFILTER, and texture-stage states to control texture filtering.

Microsoft DirectX 8.1 (C++)

Texture Blending State

[This is preliminary documentation and is subject to change.]

C++ applications use the texture blending states defined by the D3DTSS_COLOROP and D3DTSS_ALPHAOP texture stage states to control texture blending.

For more information, see Texture Blending.

Microsoft DirectX 8.1 (C++)

State Blocks

[This is preliminary documentation and is subject to change.]

A state block in Microsoft® Direct3D® is a group of device states—render states, lighting and material parameters, transformation states, texture stage states, and current texture information. The state block is a snapshot of the device's current state, or it is explicitly recorded. The snapshot can be applied to a device in a single call. Device-state blocks can be optimized by the rendering device to accelerate the common sequences of state changes that your application requires, or they can simply make applying device states easier.

In C++, you receive a state-block handle when you finish recording a state block by calling the **IDirect3DDevice8::EndStateBlock** method, and when you capture a predefined set of device state data by calling the **IDirect3DDevice8::CreateStateBlock** method.

Additional information is contained in the following topics.

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

Microsoft DirectX 8.1 (C++)

Recording State Blocks

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8** interface provides the **IDirect3DDevice8::BeginStateBlock** method to record device states in a state block as your application calls for them. The **BeginStateBlock** method causes the system to start recording device state changes in a state block, rather than applying them to the device. After you call **BeginStateBlock**, calls to any of the following methods are recorded in a device state block.

- **IDirect3DDevice8::LightEnable**
- **IDirect3DDevice8::SetClipPlane**
- **IDirect3DDevice8::SetLight**
- **IDirect3DDevice8::SetMaterial**
- **IDirect3DDevice8::SetRenderState**
- **IDirect3DDevice8::SetTexture**
- **IDirect3DDevice8::SetTextureStageState**
- **IDirect3DDevice8::SetTransform**
- **IDirect3DDevice8::SetViewport**

When you're done recording the state block, notify the system to stop recording by calling the **IDirect3DDevice8::EndStateBlock** method. The **EndStateBlock** method places the handle of the state block in the variable whose address you pass in the *pToken* parameter. Your application uses this handle to apply the state block to the device as needed, to capture new state data into the block, and to delete the state block when it is no longer required.

Performance Note For best results, group all state changes tightly between a **BeginStateBlock** and **EndStateBlock** pair.

It is important to check the error code from the **EndStateBlock** method. If the method fails, it is likely because the display mode has changed. Design your application to recover from this type of failure by recreating its surfaces and recording the state block again.

Microsoft DirectX 8.1 (C++)

Capturing State Blocks

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8::CaptureStateBlock** method updates the values within an existing state block to reflect the current state of the device. The method accepts a single parameter, *Token*, that identifies the state block that will receive the current state of the device if the call succeeds.

The **CaptureStateBlock** method is especially useful to update a state block that your application has already recorded to include slightly different state settings. To do so, apply the existing state block, change the necessary settings, then capture the state of the system back to the state block, as shown in the following code example.

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

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

// Change device states as needed.
.
.
.

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

The **CaptureStateBlock** method doesn't capture the entire state of the device; it only updates the values for the states already in the state block. For example, imagine a state block that contains the two operations shown in the following code example.

```
d3dDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
d3dDevice->SetTexture(0, pTexture);
```

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

Microsoft DirectX 8.1 (C++)

Applying State Blocks

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8::ApplyStateBlock** method applies an existing state block to the device. The **ApplyStateBlock** method accepts a single parameter, *Token*, that identifies the state block to apply.

Note You cannot apply a state block while recording another state block—that is, between calls to **IDirect3DDevice8::BeginStateBlock** and **IDirect3DDevice8::EndStateBlock**. Attempts to do so will fail.

Microsoft DirectX 8.1 (C++)

Creating Predefined State Blocks

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8::CreateStateBlock** method creates a new state block that contains the entire set of device states or only those device states related to vertex or pixel processing. The **CreateStateBlock** method accepts two parameters. The first parameter identifies the type of state information to capture in the new state block, and the second parameter is the address of a variable that will receive a valid state-block handle if the call succeeds.

Valid values for the first parameter are defined by the **D3DSTATEBLOCKTYPE** enumerated type, which includes members that you can use to capture the entire set of device states (D3DSBT_ALL), or only those states that pertain to vertex or pixel processing (D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE). The following list summarizes the states that the system captures when you pass the D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE values.

- **Vertex-related states (D3DSBT_VERTEXSTATE)**
- **State (enabled or disabled) of all lights**
- **Transformation matrices**
- **User-defined clipping planes**
  These values are set for the following render states.

  D3DRS_AMBIENT

  D3DRS_AMBIENTMATERIALSOURCE

  D3DRS_CLIPPING

  D3DRS_CLIPPLANEENABLE

  D3DRS_COLORVERTEX

  D3DRS_CULLMODE

  D3DRS_DIFFUSEMATERIALSOURCE

  D3DRS_EMISSIVEMATERIALSOURCE

  D3DRS_FOGCOLOR

  D3DRS_FOGDENSITY

D3DRS_FOGENABLE

D3DRS_FOGEND

D3DRS_FOGSTART

D3DRS_FOGTABLEMODE

D3DRS_FOGVERTEXMODE

D3DRS_INDEXEDVERTEXBLENDENABLE

D3DRS_LIGHTING

D3DRS_LOCALVIEWER

D3DRS_NORMALIZENORMALS

D3DRS_POINTSCALE_A

D3DRS_POINTSCALE_B

D3DRS_POINTSCALE_C

D3DRS_POINTSIZE

D3DRS_POINTSIZE_MAX

D3DRS_POINTSIZE_MIN

D3DRS_POINTSCALEENABLE

D3DRS_POINTSPRITEENABLE

D3DRS_PATCHEDGESTYLE

D3DRS_PATCHSEGMENTS

D3DRS_RANGEFOGENABLE

D3DRS_SHADEMODE

D3DRS_SOFTWAREVERTEXPROCESSING

D3DRS_SPECULARENABLE

D3DRS_SPECULARMATERIALSOURCE

D3DRS_TWEENFACTOR

D3DRS_VERTEXBLEND

These values are set for the following texture-stage states.

D3DTSS_TEXCOORDINDEX

D3DTSS_TEXTURETRANSFORMFLAGS

- **Pixel-related states (D3DSBT_PIXELSTATE)**
  These values are set for the following render states.

  D3DRS_ALPHABLENDENABLE

  D3DRS_ALPHAFUNC

  D3DRS_ALPHAREF

  D3DRS_ALPHATESTENABLE

  D3DRS_COLORWRITEENABLE

  D3DRS_DESTBLEND

  D3DRS_DITHERENABLE

  D3DRS_EDGEANTIALIAS

  D3DRS_FILLMODE

  D3DRS_FOGDENSITY

  D3DRS_FOGEND

  D3DRS_FOGSTART

  D3DRS_LASTPIXEL

  D3DRS_LINEPATTERN

  D3DRS_MULTISAMPLEANTIALIAS

  D3DRS_MULTISAMPLEMASK

  D3DRS_SHADEMODE

  D3DRS_SRCBLEND

  D3DRS_STENCILENABLE

  D3DRS_STENCILFAIL

  D3DRS_STENCILFUNC

  D3DRS_STENCILMASK

  D3DRS_STENCILPASS

  D3DRS_STENCILREF

D3DRS_STENCILWRITEMASK

D3DRS_STENCILZFAIL

D3DRS_TEXTUREFACTOR

D3DRS_WRAP0 through D3DRS_WRAP7

D3DRS_ZBIAS

D3DRS_ZENABLE

D3DRS_ZFUNC

D3DRS_ZWRITEENABLE

These values are set for the following texture-stage states.

D3DTSS_ADDRESSU

D3DTSS_ADDRESSV

D3DTSS_ADDRESSW

D3DTSS_ALPHAARG1

D3DTSS_ALPHAARG2

D3DTSS_ALPHAOP

D3DTSS_BORDERCOLOR

D3DTSS_BUMPENVLOFFSET

D3DTSS_BUMPENVLSCALE

D3DTSS_BUMPENVMAT00

D3DTSS_BUMPENVMAT01

D3DTSS_BUMPENVMAT10

D3DTSS_BUMPENVMAT11

D3DTSS_COLORARG1

D3DTSS_COLORARG2

D3DTSS_COLOROP

D3DTSS_MAGFILTER

D3DTSS_MAXANISOTROPY

D3DTSS_MAXMIPLEVEL

D3DTSS_MINFILTER

D3DTSS_MIPFILTER

D3DTSS_MIPMAPLODBIAS

D3DTSS_TEXCOORDINDEX

D3DTSS_TEXTURETRANSFORMFLAGS

Note It is important to check the error code from the **CreateStateBlock** method. If the method fails, it is likely because the display mode has changed. Your application should recover from this type of failure by recreating its surfaces, and then recreating the state block.

Microsoft DirectX 8.1 (C++)

Deleting State Blocks

[This is preliminary documentation and is subject to change.]

The **IDirect3DDevice8::DeleteStateBlock** deletes a state block, deallocating the memory used to contain it. The **DeleteStateBlock** method accepts the handle to the state block to delete at its only parameter. After deleting a state block, any handles to it are invalid, and can no longer be used.

Microsoft DirectX 8.1 (C++)

Programming Tips

[This is preliminary documentation and is subject to change.]

This section contains information to help you develop Microsoft® DirectX® Graphics applications efficiently. The following topics are covered.

- Performance Optimizations
- Troubleshooting
- Multithreading Issues
- Working with Device Windows
- Working with Earlier Drivers
- Working with Multiple Monitor Systems
- Presenting Multiple Views in Windowed Mode

Microsoft DirectX 8.1 (C++)

Performance Optimizations

[This is preliminary documentation and is subject to change.]

Every developer who creates real-time applications that use three-dimensional (3-D) graphics is concerned about performance optimization. This section provides you with guidelines about getting the best performance from your code.

General Performance Tips

Follow these general guidelines to increase the performance of your application.

- Clear only when you must.
- Minimize state changes and group the remaining state changes.
- Use smaller textures, if you can do so.
- Draw objects in your scene from front to back.
- Use triangle strips instead of lists and fans. For optimal vertex cache performance, arrange strips to reuse triangle vertices sooner, rather than later.
- Gracefully degrade special effects that require a disproportionate share of system resources.
- Constantly test your application's performance.
- Minimize vertex buffer switches.
- Use static vertex buffers where possible.
- Use one large static vertex buffer per FVF for static objects, rather than one per object.
- If your application needs random access into the vertex buffer, choose a vertex format size that is a multiple of 32 bits. Otherwise, select the smallest appropriate format.
- Draw using indexed primitives. This may allow for more efficient vertex caching within hardware.
- If the depth buffer format contains a stencil channel, always clear the depth and stencil channels at the same time.
- Do not copy to output registers unless necessary in shaders. For example:

  ```
  mad oD0, r1, v0, c[3]
  ```

  rather than:

  ```
  mad r1, v0, c0
  mov oD0, r1
  ```

Databases and Culling

Building a reliable database of the objects in your world is key to excellent performance in Microsoft® Direct3D®. It is more important than improvements to rasterization or hardware.

You should maintain the lowest polygon count you can possibly manage. Design for a low-polygon count by building low-polygon models from the start. Add polygons if you can do so without sacrificing performance later in the development process. Remember, the fastest polygons are the ones you don't draw.

Batching Primitives

To get the best rendering performance during execution, try to work with primitives in batches and keep the number of render-state changes as low as possible. For example, if you have an object with two textures, group the triangles that use the first texture and follow them with the necessary render state to change the texture. Then group all the triangles that use the second texture. The simplest hardware support for 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

Because lights add a per-vertex cost to each rendered frame, you can achieve significant performance improvements by being careful about how you use them in your application. Most of the following tips derive from the maxim, the fastest code is code that is never called.

- Use as few light sources as possible. If you only need to increase the overall lighting level, use the ambient light instead of adding a new light source. It's much cheaper.
- Directional lights are cheaper than point lights or spotlights. For directional lights, the direction to the light is fixed and doesn't need to be calculated on a per-vertex basis.
- Spotlights can be cheaper than point lights, because the area outside the cone of light is calculated quickly. Whether spotlights are cheaper depends on how much of your scene is lit by the spotlight.
- Use the range parameter to limit your lights to only the parts of the scene you need to illuminate. All the light types exit fairly early when they are out of range.
- Specular highlights almost double the cost of a light. Use them only when you must. Set the D3DRS_SPECULARENABLE render state to 0, the default value, whenever possible. When defining materials, you must set the specular power value to zero to turn off specular highlights for that material; simply setting the specular color to 0,0,0 is not enough.

Texture Size

Texture-mapping performance is heavily dependent on the speed of memory. There are a number of ways to maximize the cache performance of your application's textures.

- Keep the textures small. The smaller the textures are, the better chance they have of being maintained in the main CPU's secondary cache.
- Do not change the textures on a per-primitive basis. Try to keep polygons grouped in order of the textures they use.
- Use square textures whenever possible. Textures whose dimensions are 256×256 are the fastest. If your application uses four 128×128 textures, for example, try to ensure that they use the same palette and place them all into one 256×256 texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256×256 textures unless your application requires that much texturing because, as mentioned, textures should be kept as small as possible.

Using Dynamic Textures

Dynamic textures are a new Microsoft® DirectX® 8.1 feature. To find out if the driver supports dynamic textures, check the D3DCAPS2_DYNAMICTEXTURES flag of the **D3DCAPS8** structure.

Keep the following things in mind when working with dynamic textures.

- They cannot be managed. For example, their pool cannot be D3DPOOOL_MANAGED.
- Dynamic textures can be locked, even if they are created in D3DPOOL_DEFAULT.
- D3DLOCK_DISCARD is a valid lock flag for dynamic textures.

It is a good idea to create only one dynamic texture per format and possibly per size. Dynamic mipmaps, cubes, and volumes are not recommended because of the additional overhead in locking every level. For mipmaps, LOCK_DISCARD is allowed only on the top level. All levels are discarded by locking just the top level. This behavior is the same for volumes and cubes. For cubes, the top level and face 0 are locked.

The following pseudocode shows an example of using a dynamic texture.

```
DrawProceduralTexture(pTex)
{
    // pTex should not be very small since overhead of calling driver every
    // will not justify the performance gain. Experimentation is encouraged.
    pTex->Lock(DISCARD);
    <Overwrite *entire* texture>
    pTex->Unlock();
    pDev->SetTexture();
    pDev->DrawPrimitive();
}
```

Using Dynamic Vertex and Index Buffers

Dynamic vertex and index buffers have a difference in performance based the size and usage. The usage styles below help to determine whether to use D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

Usage Style 1:

```
for loop()
{
    pBuffer->Lock(...D3DLOCK_DISCARD...); //Ensures that hardware
                                          //doesn't stall by returning
                                          //a new pointer.
    Fill data (optimally 1000s of vertices/indices, no fewer) in pBuffer.
    pBuffer->Unlock()
    Change state(s).
    DrawPrimitive() or DrawIndexedPrimitive()
}
```

Usage Style 2:

```
for loop()
{
    pVB->Lock(...D3DLOCK_DISCARD...); //Ensures that hardware doesn't
                                      //stall by returning a new
                                      //pointer.
    Fill data (optimally 1000s of vertices/indices, no fewer) in pBuffer.
    pBuffer->Unlock
    for loop( 100s of times )
    {
        Change State
        DrawPrimitive() or DrawIndexPrimitives() //Tens of primitives
    }
}
```

Usage Style 3:

```
for loop()
{
    If there is space in the buffer
    {
        // Append vertices/indices.
        pBuffer->Lock(…D3DLOCK_NOOVERWRITE…);
    }
    Else
    {
        // Reset to beginning.
```

```
    pBuffer->Lock(…D3DLOCK_DISCARD…);
}
Fill few 10s of vertices/indices in pBuffer
pBuffer->Unlock
Change State
DrawPrimitive() or DrawIndexedPrimitive() // A few primitives

}
```

Style 1 is faster than either style 2 or 3, but is generally not very practical. Style 2 is usually faster than style 3, provided that the application fills at least a couple thousand vertices/indices for every **Lock,** on average. If the application fills fewer than that on average, then style 3 is faster. There is no guaranteed answer as to which lock method is faster and the best way to find out is to experiment.

Using Meshes

You can optimize meshes by using Direct3D indexed triangles instead of indexed triangle strips. The hardware will discover that 95 percent of successive triangles actually form strips and adjust accordingly. Many drivers do this for legacy hardware also.

Direct3DX mesh objects can have each triangle, or face, tagged with a **DWORD**, called *the attribute* of that face. The semantics of the **DWORD** are user-defined. They are simply used by Direct3DX to classify the mesh into subsets. The application sets per-face attributes using the **LockAttributeBuffer** call. The **Optimize** method has an option to group the mesh vertices and faces on attributes using the D3DXMESHOPT_ATTRSORT option. When this is done, the mesh object calculates an attribute table that can be obtained by the application by calling **GetAttributeTable**. This call returns 0 if the mesh is not sorted by attributes. There is no way for an application to set an attribute table because it is generated by the **Optimize method**. The attribute sort is data sensitive, so if the application knows that a mesh is attribute sorted, it still needs to call **Optimize** to generate the attribute table.

The following topics describes the different attributes of a mesh.

Attribute ID

An attribute ID is a value that associates a group of faces with an attribute group. This ID describes which subset of faces **DrawSubset** should draw. Attribute IDs are specified for the faces in the attribute buffer. The actual values of the attribute IDs can be anything that fits in 32bits, but it is common to use 0 to *n* where *n* is the number of attributes.

Attribute Buffer

The attribute buffer is an array of **DWORD**s (one per face) that specifies which attribute group each face belongs in. This buffer is initialized to zero on creation of a mesh, but is either filled by the load routines or must be filled by the user if more than one attribute with ID 0 is desired. This buffer contains the information that is used to sort the mesh based on attributes in **Optimize**. If no attribute table is present, **DrawSubset** scans this buffer to select the faces of the given attribute to draw.

Attribute Table

The attribute table is a structure owned and maintained by the mesh. The only way for one to be generated is by calling **Optimize** with attribute sorting or stronger optimization enabled.

The attribute table is used to quickly initiate a single draw primitive call to **DrawSubset**. The only other use is that progressing meshes also maintain this structure, so it is possible to see what faces and vertices are active at the current level of detail.

Z-Buffer Performance

Applications can increase performance when using z-buffering and texturing by ensuring that scenes are rendered from front to back. Textured z-buffered primitives are pretested against the z-buffer on a scan line basis. If a scan line is hidden by a previously rendered polygon, the system rejects it quickly and efficiently. Z-buffering can improve performance, but the technique is most useful when a scene includes a great deal of *overdraw*. Overdraw is the average number of times that a screen pixel is written to. Overdraw is difficult to calculate exactly, but you can often make a close approximation. If the overdraw averages less than 2, you can achieve the best performance by turning z-buffering off and rendering the scene from back to front.

On faster personal computers, software rendering to system memory is often faster than rendering to video memory although it has the disadvantage of not being able to use double buffering or hardware-accelerated clear operations. If your application can render to either system or video memory, and if you include a routine that tests which is faster, you can take advantage of the best approach on the current system. The Direct3D sample code in this software development kit (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.

Microsoft DirectX 8.1 (C++)

Troubleshooting

[This is preliminary documentation and is subject to change.]

This topic lists common categories of problems that you might encounter when writing Microsoft® Direct3D® applications, and how to prevent them.

Device Creation

If your application fails during device creation, check for the following common errors.

- Make sure you check the device capabilities, particularly the render depths.
- Examine the error code. **D3DERR_OUTOFVIDEOMEMORY** is always possible.
- Use the debug DirectX DLLs and review output messages under the debugger.

Using Lit Vertices

Applications that use lit vertices should disable the Microsoft® Direct3D® lighting engine by setting the D3DRS_LIGHTING render state to FALSE. By default, when lighting is enabled, the system sets the color for any vertex that doesn't contain a normal vector to 0 (black), even if the input vertex contained a nonzero color value. Because lit-vertices don't, by their nature, contain a vertex normal, any color information passed to Direct3D is lost during rendering if the lighting engine is enabled.

Obviously, vertex color is important to any application that performs its own lighting. To prevent the system from imposing the default values, make sure you set D3DRS_LIGHTING to FALSE.

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

- Ensure that your triangles are not degenerate.
- Ensure that your triangles are not being culled.
- Make sure that your transformations are internally consistent.
- Check the viewport settings to be sure they allow your triangles to be seen.

Debugging

Debugging a Microsoft® Direct3D® application can be challenging. Try the following techniques, in addition to checking all the return values—a particularly important piece of advice in Direct3D programming, which is dependent on very different hardware implementations.

- Switch to debug DLLs.
- Force a software-only device, turning off hardware acceleration even when it is available.
- Force surfaces into system memory.
- Create an option to run in a window, so that you can use an integrated debugger.

The second and third options in this list can help you avoid the Win16 lock which can otherwise cause your debugger to hang.

Also, try adding the following entries to Win.ini.

```
[Direct3D]
debug=3
[DirectDraw]
debug=3
```

Borland Floating-Point Initialization

Compilers from Borland report floating-point exceptions in a manner that is incompatible with Microsoft® Direct3D®. To solve this problem, include a _matherr exception handler like the following:

```
// Borland floating point initialization
#include <math.h>
#include <float.h>

void initfp(void)
{
    // Disable floating point exceptions.
    _control87(MCW_EM,MCW_EM);
}

int _matherr(struct _exception  *e)
{
    e;              // Dummy reference to catch the warning.
    return 1;       // Error has been handled.
}
```

Parameter Validation

For performance reasons, the debug version of the Microsoft® Direct3D® Immediate Mode run time performs more parameter validation than the retail version, which sometimes performs no validation at all. This enables applications to perform robust debugging with the slower debug run-time component before using the faster retail version for performance tuning and final release.

Although several Direct3D Immediate Mode methods impose limits on the values that they can accept, these limits are often only checked and enforced by the debug version of the Direct3D Immediate Mode run time. Applications must conform to these limits, or unpredictable and undesirable results can occur when running on the retail version of Direct3D. For example, the **IDirect3DDevice8::DrawPrimitive** method accepts a parameter (*PrimitiveCount*) that indicates the number of primitives that the method will render. The method can only accept values between 0 and D3DMAXNUMPRIMITIVES. In the debug version of Direct3D, if you pass more than D3DMAXNUMPRIMITIVES primitives, the method fails gracefully, printing an error message to the error log, and returning an error value to your application. Conversely, if your application makes the same error when it is running with the retail version of the run time, behavior is undefined. For performance reasons, the method does not validate the parameters, resulting in unpredictable and completely situational behavior when they are not valid. In some cases the call might work, and in other cases it might cause a memory fault in Direct3D. If an invalid call consistently works with a particular hardware configuration and DirectX version, there is no guarantee that it will continue to function on other hardware or with later releases of DirectX.

If your application encounters unexplained failures when running with the retail Direct3D Immediate Mode run-time file, test against the debug version and look closely for cases where your application passes invalid parameters.

Microsoft DirectX 8.1 (C++)

Multithreading Issues

[This is preliminary documentation and is subject to change.]

Full-screen Microsoft® Direct3D® applications provide a window handle to the Direct3D run time. The window is hooked by the run time. This means that all messages passed to the application's window message procedure have first been examined by the Direct3D run time's own message-handling procedure.

Display mode changes are affected by support routines built into the underlying operating system. When mode changes occur, the system broadcasts several messages to all applications. In Direct3D applications, the messages are received on the window procedure thread, which is not necessarily the same thread that called **IDirect3DDevice8::Reset** or **IDirect3D8::CreateDevice** (or the final **IUnknown::Release** of **IDirect3DDevice8**, which can cause a display mode change). The Direct3D run time maintains several critical sections internally. Because at least one of these critical sections is held across the mode switch caused by **Reset** or **CreateDevice**, these critical sections are still held when the application receives the mode-change related window messages.

This design has some implications for multithreaded applications. In particular, an application must be sure to strongly segregate its window message handling threads from its Direct3D

threads. An application that causes a mode change on one thread but makes Direct3D calls on a different thread in its window procedure is in danger of deadlock.

For these reasons, Direct3D is designed so that the methods **Reset**, **CreateDevice**, **TestCoorperativeLevel**, or the final **Release** of **IDirect3DDevice8** can only be called from the same thread that handles window messages.

Microsoft DirectX 8.1 (C++)

Working with Device Windows

[This is preliminary documentation and is subject to change.]

This section lists issues that you might encounter when working with device windows in Microsoft® Direct3D® applications.

- Direct3D only hooks up the focus windows instead of the device window with the Direct3D message processing function, and only processes the focus window messages. So, the focus window should be the parent of any device window.
- For any application, multiple monitor or single monitor, at least one device window must be the focus window. In practical terms, this means that you cannot use a child window of your focus window as the device window in single monitor systems. You must use the parent focus window as both the focus and device windows. For a multiple monitor system, at least one monitor must use the focus window as its device window.

Microsoft DirectX 8.1 (C++)

Working with Earlier Drivers

[This is preliminary documentation and is subject to change.]

This section lists issues that may be encountered when working with Microsoft® DirectX® 8.0 on drivers written for versions of DirectX earlier than DirectX 8.0.

- When working with a TnLHAL device, the fog intensity is computed but the absolute value operation is not applied to this value. Rather, the value is left negative if that is what is computed. The best way to avoid this situation is to set up transforms appropriately. A less-preferred method is to make the fog-start and fog-end values negative to match.

You can detect a driver not written for DirectX 8.0 by querying the **MaxStream** member of **D3DCAPS8**. If this value is 0, then it is not a DirectX 8.0 driver.

Microsoft DirectX 8.1 (C++)

Working with Multiple Monitor Systems

[This is preliminary documentation and is subject to change.]

The concept of exclusive full-screen mode is retained in Microsoft® DirectX® 8.0, but it is kept entirely implicit in the **IDirect3D8::CreateDevice** and **IDirect3DDevice8::Reset** method calls. Whenever a device is successfully reset or created in full-screen operation, the Microsoft® Direct3D® object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and at this point the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D8 object can neither assume full-screen operation nor allocate video memory. In addition, when a Direct3D8 object assumes exclusive mode, all devices other than the device that went full-screen are placed into the lost state. For information on how to handle lost devices, see Lost Devices.

Along with exclusive mode, the Direct3D8 object is informed of the focus window to be used by the device. Exclusive mode is released when the last full-screen device owned by that Direct3D8 object is either reset to windowed mode or destroyed.

Devices can be divided into two categories when a Direct3D8 object owns exclusive mode. The first category of devices are those that were created by the same Direct3D8 object that created the device that is already full-screen, have the same focus window as the device that is already full-screen, and represent a different adapter from any full-screen device. Devices in this category have no restrictions concerning their ability to be reset or created and are not placed into the lost state. Devices in this category can even be placed into full-screen mode.

Devices not in this category, which would be those created by a different Direct3D8 object, or with a different focus window, or for some adapter with a device already full-screen cannot be reset and remain in the lost state until exclusive mode is lost.

The practical implication is that a multiple monitor application can place several devices in full-screen mode, but only if all these devices are for different adapters, were created by the same Direct3D8 object, and all share the same focus window.

Note When you create a new device using the same **IDirect3D8** object and focus window, your original device will lose it's surfaces. You will need to call **IDirect3DDevice8::Reset** on the first device in order for your application to use it. For example, to create two devices, do the following:

1. Create Device 1.
2. Create Device 2.
3. Reset Device 1.

Microsoft DirectX 8.1 (C++)

Presenting Multiple Views in Windowed Mode

[This is preliminary documentation and is subject to change.]

In addition to the swap chain that is owned and manipulated through the **IDirect3DDevice8** interface, an application can create additional swap chains in order to present multiple views from the same device. The application typically creates one swap chain per view by using the **IDirect3DDevice8::CreateAdditionalSwapChain** method, and associates each swap chain with a particular window. The application renders images into the back buffers of each swap chain, and then presents them individually.

Only one swap chain at a time can be full-screen on each adapter.

Microsoft DirectX 8.1 (C++)

X Files

[This is preliminary documentation and is subject to change.]

The *X file* format refers to files with the .x file name extension. X files were introduced with Microsoft® DirectX® 2.0. A binary version of this format was subsequently released with DirectX 3.0, which is also described in this documentation. DirectX 6.0 introduced interfaces and methods that enable reading from and writing to .x files.

X files provide a rich, template-driven format that enables the storage of meshes, textures, animations, and user-definable objects. Support for animation sets enables you to store predefined paths for playback in real time. Instancing and hierarchies are also supported. Instancing enables multiple references to an object, such as a mesh, while storing its data only once per file. Hierarchies are used to express relationships between data records.

The .x file format provides low-level data primitives on which applications define higher-level primitives through templates.

This section discusses the structure of .x files and how to use them in your applications.

Information is divided into the following topics.

- X File Interface Hierarchy
- X File Architecture
- Using X Files
- X File Format Reference

Microsoft DirectX 8.1 (C++)

X File Interface Hierarchy

[This is preliminary documentation and is subject to change.]

The following tables illustrate the relationship between the .x file interfaces.

Interface	Derives from	Derives from
IDirectXFileBinary	IDirectXFileObject	IUnknown
IDirectXFileData	IDirectXFileObject	IUnknown
IDirectXFileReference	IDirectXFileObject	IUnknown

Interface	Derives from
IDirectXFileSave	IUnknown

Microsoft DirectX 8.1 (C++)

X File Architecture

[This is preliminary documentation and is subject to change.]

The Microsoft® DirectX® .x file format is not specific to any application. It uses templates that don't depend on how the file is used. This allows the .x file format to be used by any client application.

This following sections deal with the content and syntax of the file format, which uses the .x extension in the DirectX Software Development Kit (SDK).

- Reserved Words, Header, and Comments
- Templates
- Data
- Use of Commas and Semicolons

Microsoft DirectX 8.1 (C++)

Reserved Words, Header, and Comments

[This is preliminary documentation and is subject to change.]

The table below shows which words are reserved and must not be used.

- ARRAY
- BINARY
- BINARY_RESOURCE
- CHAR
- CSTRING
- DOUBLE
- DWORD
- FLOAT
- SDWORD
- STRING
- SWORD
- TEMPLATE
- UCHAR
- ULONGLONG
- UNICODE
- WORD

The variable-length header is compulsory and must be at the beginning of the data stream. The header contains the following data.

Type	Size	Contents	Content meaning
Magic Number (required)	4 bytes	"xof"	
Version Number (required)	2 bytes	03	Major Version 3
	2 bytes	02	Major Version 2
Format Type (required)	4 bytes	"txt"	Text File
		"bin"	Binary File
		"tzip"	MSZip Compressed Text File
		"bzip"	MSZip Compressed Binary File
Float Size (required)	4 bytes	0064	64-bit floats
		0032	32-bit floats

The following example shows a valid header.

```
xof 0302txt 0064
```

Comments are applicable only in text files. Comments can occur anywhere in the data stream.

A comment begins with either C++ style double-slashes (//), or a number sign (#). The comment runs to the next new line. The following example shows valid comments.

```
#  This is a comment.
// This is another comment.
```

Microsoft DirectX 8.1 (C++)

Templates

[This is preliminary documentation and is subject to change.]

A template has the following syntax definition.

```
template              : TOKEN_TEMPLATE name TOKEN_OBRACE
                            class_id
                            template_parts
                            TOKEN_CBRACE

template_parts        : template_members_part TOKEN_OBRACKET
                        template_option_info
                        TOKEN_CBRACKET
                      | template_members_list

template_members_part : /* Empty */
                      | template_members_list

template_option_info  : ellipsis
                      | template_option_list

template_members_list :    template_members
                      | template_members_list template_members

template_members      : primitive
                      | array
                      | template_reference

primitive             : primitive_type optional_name TOKEN_SEMICOLON

array                 : TOKEN_ARRAY array_data_type name dimension_list
                        TOKEN_SEMICOLON

template_reference    : name optional_name YT_SEMICOLON

primitive_type        : TOKEN_WORD
                      | TOKEN_DWORD
                      | TOKEN_FLOAT
                      | TOKEN_DOUBLE
                      | TOKEN_CHAR
                      | TOKEN_UCHAR
                      | TOKEN_SWORD
                      | TOKEN_SDWORD
                      | TOKEN_LPSTR
                      | TOKEN_UNICODE
                      | TOKEN_CSTRING

array_data_type       : primitive_type
                      | name

dimension_list        : dimension
```

```
                      | dimension_list dimension

dimension             : TOKEN_OBRACKET dimension_size TOKEN_CBRACKET

dimension_size        : TOKEN_INTEGER
                      | name

template_option_list  : template_option_part
                      | template_option_list template_option_part

template_option_part  : name optional_class_id

name                  : TOKEN_NAME

optional_name         : /* Empty */
                      | name

class_id              : TOKEN_GUID

optional_class_id     : /* Empty */
                      | class_id

ellipsis              : TOKEN_DOT TOKEN_DOT TOKEN_DOT
```

Microsoft DirectX 8.1 (C++)

Data

[This is preliminary documentation and is subject to change.]

A data object has the following syntax definition.

```
object                : identifier optional_name TOKEN_OBRACE
                            optional_class_id
                            data_parts_list
                            TOKEN_CBRACE
data_parts_list       : data_part
                      | data_parts_list data_part

data_part             : data_reference
                      | object
                      | number_list
                      | float_list
                      | string_list

number_list           : TOKEN_INTEGER_LIST

float_list            : TOKEN_FLOAT_LIST

string_list           : string_list_1 list_separator

string_list_1         : string
                      | string_list_1 list_separator string

list_separator        : comma
                      | semicolon

string                : TOKEN_STRING

identifier            : name
                      | primitive_type
```

```
data_reference        : TOKEN_OBRACE name optional_class_id TOKEN_CBRACE
```

Note that in number_list and float_list data in binary files, TOKEN_COMMA and TOKEN_SEMICOLON are not used. The comma and semicolon are used in string_list data. Also note that you can only use data_reference for optional data members.

Microsoft DirectX 8.1 (C++)

Use of Commas and Semicolons

[This is preliminary documentation and is subject to change.]

Using commas and semicolons can be the most complex syntax issue in the file format, and this usage is very strict. Commas are used to separate array members; semicolons terminate each data item.

For example, if a template is defined in the following manner:

```
template mytemp {
DWORD myvar;
}
```

Then an instance of this template looks like the following:

```
mytemp dataTemp {
1;
}
```

If a template containing another template is defined in the following manner;

```
template mytemp {
DWORD myvar;
DWORD myvar2;
}
template container {
FLOAT aFloat;
mytemp aTemp;
}
```

Then an instance of this template looks like the following:

```
container dataContainer {
1.1;
2; 3;;
}
```

Note that the second line that represents the *mytemp* inside *container* has two semicolons at the end of the line. The first semicolon indicates the end of the data item, *aTemp* (inside container), and the second semicolon indicates the end of the *container*.

If an array is defined in the following manner:

```
Template mytemp {
array DWORD myvar[3];
}
```

Then an instance of this looks like the following:

```
mytemp aTemp {
1, 2, 3;
}
```

In the array example, there is no need for the data items to be separated by semicolons because they are delineated by commas. The semicolon at the end marks the end of the array.

Consider a template that contains an array of data items defined by a template.

```
template mytemp {
DWORD myvar;
DWORD myvar2;
}
template container {
DWORD count;
array mytemp tempArray[count];
}
```

An instance of this would look like the following example.

```
container aContainer {
3;
1;2;,3;4;,5;6;;
}
```

Microsoft DirectX 8.1 (C++)

Using X Files

[This is preliminary documentation and is subject to change.]

This section contains information about using .x files in a Microsoft® Direct3D® application. Information is divided into the following topics.

- Loading an X File
- Saving to an X File
- Creating a Cube
- Converting and Exporting 3-D Models to X Files

Microsoft DirectX 8.1 (C++)

Loading an X File

[This is preliminary documentation and is subject to change.]

Use the following procedure to load an .x file.

1. Use the **DirectXFileCreate** function to create an **IDirectXFile** object.
2. If templates are present in the Microsoft® DirectX® file that you will load, use the **IDirectXFile::RegisterTemplates** method to register those templates.
3. Use the **IDirectXFile::CreateEnumObject** method to create an **IDirectXFileEnumObject** enumerator object.

4. Loop through the objects in the file. For each object, perform the following steps.
    a. Use the **IDirectXFileEnumObject::GetNextDataObject** method to retrieve each **IDirectXFileData** object.
    b. Use the **IDirectXFileData::GetType** method to retrieve the data's type.
    c. Load the data using the **IDirectXFileData::GetData** method.
    d. If the object has optional members, retrieve the optional members by calling the **IDirectXFileData::GetNextObject** method.
    e. Release the **IDirectXFileData** object.
5. Release the **IDirectXFileEnumObject** object.
6. Release the **IDirectXFile** object.

Microsoft DirectX 8.1 (C++)

Saving to an X File

[This is preliminary documentation and is subject to change.]

Use the following procedure to save .x file templates and data to an .x file.

1. Use the **DirectXFileCreate** function to create an **IDirectXFile** object.
2. Use the **IDirectXFile::RegisterTemplates** method to inform the Microsoft® DirectX® file system about any templates that you will use.
3. Use the **IDirectXFile::CreateSaveObject** method to create an **IDirectXFileSaveObject** object.
4. Use the **IDirectXFileSaveObject::SaveTemplates** method to save templates, if desired.
5. Loop through the objects to save. For each top-level object, perform the following steps.
    a. Use the **IDirectXFileSaveObject::CreateDataObject** method to create an **IDirectXFileData** object as a top-level object in the file. If the top-level data object has optional child objects, add them to the object by using the appropriate method from the next step.
    b. Each **IDirectXFileData** object can have optional child objects if its template allows it. The child objects can be any of the three types of objects: **IDirectXFileData**, **IDirectXFileDataReference**, or **IDirectXFileBinary**. Loop through the objects you need to save, adding each optional child member to the object list in the manner appropriate to its type, as illustrated in the following steps. Then, if the object type is Data, call the **IDirectXFileSaveObject::CreateDataObject** method to create an **IDirectXFileData** object, and then call the **IDirectXFileData::AddDataObject** method to add it as a child of the object. If the object type is Data Reference, call the **IDirectXFileData::AddDataReference** method to create and add the data reference object as a child of the object. Or, if the object type is Binary, call the **IDirectXFileData::AddBinaryObject** method to create and add the binary object as a child of the object.
    c. Call the **IDirectXFileSaveObject::SaveData** method to save the data object and its children.
    d. Release the **IDirectXFileData** object.
6. Release the **IDirectXFileSaveObject** object.
7. Release the **IDirectXFile** object.

Microsoft DirectX 8.1 (C++)

Creating a Cube

[This is preliminary documentation and is subject to change.]

This section describes a simple cube and shows how to add textures, frames, and animations.

- Defining a Simple Cube
- Adding Textures
- Adding Frames and Animations

Microsoft DirectX 8.1 (C++)

Defining a Simple Cube

[This is preliminary documentation and is subject to change.]

The following file defines a simple cube that has four red sides and two green sides. In this file, optional information is used to add information to the data object defined by the **Mesh** template.

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;     // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
}
// Define a mesh with 8 vertices and 12 faces (triangles). Use
// optional data objects in the mesh to specify materials, normals,
// and texture coordinates.
Mesh CubeMesh {
8;                                  // 8 vertices.
1.000000;1.000000;-1.000000;,       // Vertex 0.
-1.000000;1.000000;-1.000000;,      // Vertex 1.
-1.000000;1.000000;1.000000;,       // And so on.
1.000000;1.000000;1.000000;,
1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;1.000000;,
1.000000;-1.000000;1.000000;;

12;                         // 12 faces.
3;0,1,2;,                   // Face 0 has three vertices.
3;0,2,3;,                   // And so on.
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;

// All required data has been defined. Now define optional data
// using the hierarchical nature of the file format.
MeshMaterialList {
2;                      // Number of materials used.
12;                     // A material for each face.
```

```
0,                    // Face 0 uses the first material.
0,
0,
0,
0,
0,
0,
0,
1,                    // Face 8 uses the second material.
1,
1,
1;;
{RedMaterial}         // References to the definitions
{GreenMaterial}       // of material 0 and 1.
}
MeshNormals {
8;                    // Define 8 normals.
0.333333;0.666667;-0.666667;,
-0.816497;0.408248;-0.408248;,
-0.333333;0.666667;0.666667;,
0.816497;0.408248;0.408248;,
0.666667;-0.666667;-0.333333;,
-0.408248;-0.408248;-0.816497;,
-0.666667;-0.666667;0.333333;,
0.408248;-0.408248;0.816497;;
12;                   // For the 12 faces, define the normals.
3;0,1,2;,
3;0,2,3;,
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;
}
MeshTextureCoords {
8;                        // Define texture coords for each vertex.
0.000000;1.000000;
1.000000;1.000000;
0.000000;1.000000;
1.000000;1.000000;
0.000000;0.000000;
1.000000;0.000000;
0.000000;0.000000;
1.000000;0.000000;;
}
}
```

Microsoft DirectX 8.1 (C++)

Adding Textures

[This is preliminary documentation and is subject to change.]

To add textures, use the hierarchical nature of the file format and add an optional **TextureFilename** data object to the **Material** data objects. The **Material** objects now read as follows:

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"tex1.ppm";
}
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;     // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"win95.ppm";
}
}
```

Microsoft DirectX 8.1 (C++)

Adding Frames and Animations

[This is preliminary documentation and is subject to change.]

This section shows how to add frames and animations to a simple cube.

Working with Frames

A frame is expected to take the following structure.

```
Frame Aframe {        // The frame name is chosen for convenience.
FrameTransformMatrix {
...transform data...
}
[ Meshes ] and/or [ More frames]
}
```

Place the defined cube mesh inside a frame with an identity transform. Then apply an animation to this frame.

```
Frame CubeFrame {
FrameTransformMatrix {
1.000000, 0.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000,
0.000000, 0.000000, 1.000000, 0.000000,
0.000000, 0.000000, 0.000000, 1.000000;;
}
{CubeMesh}        // You could have the mesh inline, but this
                  // uses an object reference instead.
}
```

Working with AnimationSets and Animations

An animation is defined by a set of keys. A key is a time value associated with a scaling operation, an orientation, or a position.

```
Animation Animation0 {        // The name is chosen for convenience.
```

```
{ Frame that it applies to&em;normally a reference }
AnimationKey {
...animation key data...
}
{ ...more animation keys... }
}
```

Animations are then grouped into AnimationSets:

```
AnimationSet AnimationSet0 { // The name is chosen for convenience.
{ an animation—could be inline or a reference }
{ ... more animations ... }
}
```

Now take the cube through an animation.

```
AnimationSet AnimationSet0 {
Animation Animation0 {
{CubeFrame}    // Use the frame containing the cube.
AnimationKey {
2;             // Position keys
9;             // 9 keys
10; 3; -100.000000, 0.000000, 0.000000;;,
20; 3; -75.000000, 0.000000, 0.000000;;,
30; 3; -50.000000, 0.000000, 0.000000;;,
40; 3; -25.500000, 0.000000, 0.000000;;,
50; 3; 0.000000, 0.000000, 0.000000;;,
60; 3; 25.500000, 0.000000, 0.000000;;,
70; 3; 50.000000, 0.000000, 0.000000;;,
80; 3; 75.500000, 0.000000, 0.000000;;,
90; 3; 100.000000, 0.000000, 0.000000;;;
}
}
}
```

For more information, see the **Animation** and **AnimationSet** templates.

Microsoft DirectX 8.1 (C++)

Converting and Exporting 3-D Models to X Files

[This is preliminary documentation and is subject to change.]

Microsoft® Direct3D® supplies the following tools to convert and export 3-D models to the .x file format.

- Conv3ds.exe
- XSkinExp.dle

Microsoft DirectX 8.1 (C++)

Conv3ds.exe

[This is preliminary documentation and is subject to change.]

The Conv3ds.exe utility converts three-dimensional (3-D) models produced by Autodesk 3-D

Studio and other modeling packages to the Microsoft® DirectX® file format. By default, the utility produces binary .x files with no templates.

Running Conv3ds.exe

You can run Conv3ds.exe with no options, and it will produce an .x file containing a hierarchy of frames. For example, the following command produces an .x file called File.x from the .3ds file.

```
conv3ds File.3ds
```

To run Conv3ds.exe with options, see Specifying Optional Arguments for Conv3ds.exe.

Information is divided into the following sections.

- Specifying Optional Arguments for Conv3ds.exe
- Producing 3DS Files from Lightwave Objects
- Hints and Tips

Microsoft DirectX 8.1 (C++)

Specifying Optional Arguments for Conv3ds.exe

[This is preliminary documentation and is subject to change.]

If the .3ds file contains key frame data, you can use the **-A** option to produce an .x file that contains an animation set. The following command would do this.

```
conv3ds -A File.3ds
```

The *File.3ds* parameter is the name of the file to be converted.

Use the **-m** option to make an .x file that contains a single mesh made from all the objects in the .3ds file.

```
conv3ds -m File.3ds
```

Use the **-T** option to wrap all the objects and frame hierarchies in a single top-level frame. Using this option, all the frames and objects in the .3ds file can be loaded with a single call. The first top-level frame hierarchy in the .x file will be loaded. The frame containing all the other frames and meshes is called x3ds_filename, without the .3ds extension. The **-T** option will have no effect if it is used with the **-m** option.

The **-s** option enables you to specify a scale factor for all the objects converted in the .3ds file. For example, the following command makes all objects ten times bigger.

```
conv3ds -s10 File.3ds
```

The following command makes all objects ten times smaller:

```
conv3ds -s0.1 File.3ds
```

The **-r** option reverses the winding order of the faces when the .3ds file is converted. If, after converting the .3ds file and viewing it in Microsoft® Direct3D®, the object appears inside-

out, try converting it with the **-r** option. All Lightwave models exported as .3ds files need this option. For details, see Producing 3DS Files from Lightwave Objects.

The **-v** option turns on verbose output mode. Specify an integer with it. The following table shows the currently supported integers.

Option	Meaning
-v0	Default. Verbose mode off.
-v1	Prints warnings about bad objects, and prints general information about what the converter is doing.
-v2	Prints basic key frame information, the objects being included in the conversion process, and information about the objects being saved.
-v3	Very verbose. Most useful for debugging information.

The **-e** option enables you to change the extension for texture map files, as shown in the following example.

```
conv3ds -e"ppm" File.3ds
```

If File.3ds contains objects that reference the texture map file Brick.gif, the .x file will reference the texture map file Brick.ppm. The converter does not convert the texture map file. The texture map files must be in the D3DPATH when the resulting .x file is loaded. The D3DPath is an environment variable that sets the default search path.

The **-x** option forces the Conv3ds.exe utility to produce a text .x file, instead of a binary .x file. Text files are larger but can be easily edited by hand.

The **-X** option forces the Conv3ds.exe utility to include the .x file templates in the file. By default, the templates are not included.

The **-t** option specifies that the .x file produced will not contain texture information.

The **-N** option specifies that the .x file produced will not contain normal vector information. All the load calls will generate normal vectors for objects with no normal vectors in the .x file.

The **-c** option specifies that the .x file produced should not contain texture coordinates. By default, if you use the **-m** option, the mesh that is output will contain (0,0) uv texture coordinates if the .3ds object had no texture coordinates.

The **-f** option specifies that the .x file produced should not contain a frame transformation matrix.

The **-z** and **-Z** options enable you to adjust the alpha face color value of all the materials referenced by objects in the .x file. For example, the following command causes Conv3ds.exe to add 0.1 to all alpha values under 0.2.

```
conv3ds -z0.1 -Z0.2 File.3ds
```

The following command causes Conv3ds.exe to subtract 0.2 from the alpha values for all alphas.

```
conv3ds-z"-0.2" -z1 File.3ds
```

The **-o** option enables you to specify the file name for the .x file produced.

The **-h** option tells the converter not to try to resolve any hierarchy information in the .3ds file, usually produced by the key framer. Instead, all the objects are output in top-level frames if the **-m** option is not used.

Microsoft DirectX 8.1 (C++)

Producing 3DS Files from Lightwave Objects

[This is preliminary documentation and is subject to change.]

There are several issues with .3ds files exported by the Trans3d plug-in for Lightwave. These are best handled using the following Conv3ds.exe command.

```
conv3ds -r -N -f -h -T|m trans3dfile.3ds
```

All the .3ds objects produced by Trans3d and the Lightwave object editor need their winding order reversed. Otherwise, they appear inside-out when displayed. They contain no surface normal vector information.

Microsoft DirectX 8.1 (C++)

Hints and Tips

[This is preliminary documentation and is subject to change.]

If you cannot see objects produced by Conv3ds.exe after loading them, use the scale option **-s** with a scale factor of approximately 100. This increases the scale of the objects in the .x file.

If, after loading the object into the viewer and switching from flat shading into Gouraud shading, the object turns dark gray, try converting with the **-N** option.

If the textures are not loaded after the object is converted, check whether the object is referencing either .ppm or .bmp files by using the **-e** option. Also check whether the texture widths and heights are a power of 2. Make sure the textures are stored in a directory in your D3DPATH.

Currently, Conv3ds.exe cannot handle dummy frames used in .3ds animations. It ignores them. However, it converts any child objects.

Microsoft DirectX 8.1 (C++)

XSkinExp.dle

[This is preliminary documentation and is subject to change.]

XSkinExp.dle is used as an exporter for 3D Studio Max and is located on the DirectX8 SDK CD.

The path for this file is: (*CD Drive*):\DXF\extras\direct3d\tools\3dsmax.

See the readme file provided with XSkinExp.dle for information on how to use the .dle.

Microsoft DirectX 8.1 (C++)

X File Format Reference

[This is preliminary documentation and is subject to change.]

This section contains reference information for the Microsoft® DirectX® .x file format.

- X File Templates
- Binary Format

Microsoft DirectX 8.1 (C++)

X File Templates

[This is preliminary documentation and is subject to change.]

This section lists and details the following .x file templates.

- **Animation**
- **AnimationKey**
- **AnimationOptions**
- **AnimationSet**
- **Boolean**
- **Boolean2d**
- **ColorRGB**
- **ColorRGBA**
- **Coords2d**
- **FloatKeys**
- **Frame**
- **FrameTransformMatrix**
- **Header**
- **IndexedColor**
- **Material**
- **Matrix4x4**
- **Mesh**
- **MeshFace**
- **MeshFaceWraps**
- **MeshTextureCoords**
- **MeshMaterialList**
- **MeshNormals**
- **MeshVertexColors**
- **Patch**
- **PatchMesh**
- **Quaternion**
- **SkinWeights**
- **TextureFilename**
- **TimedFloatKeys**
- **Vector**

- **VertexDuplicationIndices**
- **XSkinMeshHeader**

Microsoft DirectX 8.1 (C++)

Animation

[This is preliminary documentation and is subject to change.]

Contains animations referencing a previous frame. It should contain one reference to a frame and at least one set of **AnimationKey** templates. It also can contain an **AnimationOptions** data object.

UUID

```
<3D82AB4F-62DA-11cf-AB39-0020AF71E433>
```

Member name Type	Optional array size	Optional data objects
None		Any

Optional Data Elements

The following optional data elements are used.

Data element	Description
AnimationKey	All animations require AnimationKeys.
AnimationOptions	If this element is not present, an animation is closed.

Microsoft DirectX 8.1 (C++)

AnimationKey

[This is preliminary documentation and is subject to change.]

Defines a set of animation keys. The **keyType** member specifies whether the keys are rotation, scale position, or matrix keys (using the integers 0, 1, 2, or 3 respectively). A matrix key is useful for sets of animation data that need to be represented as transformation matrices.

UUID

```
<10DD46A8-775B-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
keyType	**DWORD**		None
nKeys	**DWORD**		
keys	array **TimedFloatKeys**	**nKeys**	

Microsoft DirectX 8.1 (C++)

AnimationOptions

[This is preliminary documentation and is subject to change.]

Enables you to set the animation options. The **openclosed** member can be either 0 for a closed or 1 for an open animation. The **positionquality** member is used to set the position quality for any position keys specified and can either be 0 for spline positions or 1 for linear positions. By default, an animation is closed.

UUID

<E2BF56C0-840F-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
openclosed	**DWORD**		None
positionquality	**DWORD**		

Microsoft DirectX 8.1 (C++)

AnimationSet

[This is preliminary documentation and is subject to change.]

Contains one or more **Animation** objects. Each animation within an animation set has the same time at any given point. Increasing the animation set's time increases the time for all the animations it contains.

UUID

<3D82AB50-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
None			**Animation**

Microsoft DirectX 8.1 (C++)

Boolean

[This is preliminary documentation and is subject to change.]

Defines a simple Boolean type. This template should be set to 0 or 1.

UUID

<4885AE61-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
truefalse	**DWORD**		None

Microsoft DirectX 8.1 (C++)

Boolean2d

[This is preliminary documentation and is subject to change.]

Defines a set of two Boolean values used in the **MeshFaceWraps** template to define the texture topology of an individual face.

UUID

<4885AE63-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
u	**Boolean**		None
v	**Boolean**		

Microsoft DirectX 8.1 (C++)

ColorRGB

[This is preliminary documentation and is subject to change.]

Defines the basic RGB color object.

UUID

<D3E16E81-7835-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
red	**FLOAT**		None
green	**FLOAT**		
blue	**FLOAT**		

Microsoft DirectX 8.1 (C++)

ColorRGBA

[This is preliminary documentation and is subject to change.]

Defines a color object with an alpha component. This is used for the face color in the material template definition.

UUID

<35FF44E0-6C7C-11cf-8F52-0040333594A3>

Member name	Type	Optional array Size	Optional data objects

red	**FLOAT**	None
green	**FLOAT**	
blue	**FLOAT**	
alpha	**FLOAT**	

Microsoft DirectX 8.1 (C++)

Coords2d

[This is preliminary documentation and is subject to change.]

A two dimensional vector used to define a mesh's texture coordinates.

UUID

```
<F6F23F44-7686-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
u	**FLOAT**		None
v	**FLOAT**		

Microsoft DirectX 8.1 (C++)

FloatKeys

[This is preliminary documentation and is subject to change.]

Defines an array of floating-point numbers (floats) and the number of floats in that array. This is used for defining sets of animation keys.

UUID

```
<10DD46A9-775B-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
nValues	**DWORD**		None
values	array **FLOAT**	**nValues**	

Microsoft DirectX 8.1 (C++)

Frame

[This is preliminary documentation and is subject to change.]

Defines a frame. Currently, the frame can contain objects of the type **Mesh** and a **FrameTransformMatrix**.

UUID

```
<3D82AB46-62DA-11cf-AB39-0020AF71E433>
```

Member name	Type	Optional array size	Optional data objects
None			Any

Optional Data Elements

The following optional elements can be used.

Data element	Description
FrameTransformMatrix	If this element is not present, no local transform is applied to the frame.
Mesh	Any number of mesh objects that become children of the frame. These objects can be specified inline or by reference.

Microsoft DirectX 8.1 (C++)

FrameTransformMatrix

[This is preliminary documentation and is subject to change.]

Defines a local transform for a frame (and all its child objects).

UUID

```
<F6F23F41-7686-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
frameMatrix	**Matrix4x4**		None

Microsoft DirectX 8.1 (C++)

Header

[This is preliminary documentation and is subject to change.]

The following definitions should be used when directly reading and writing the binary header. Note that compressed data streams are not currently supported and are therefore not detailed here.

```
#define XOFFILE_FORMAT_MAGIC \
  ((long)'x' + ((long)'o' << 8) + ((long)'f' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_VERSION \
  ((long)'0' + ((long)'3' << 8) + ((long)'0' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_BINARY \
  ((long)'b' + ((long)'i' << 8) + ((long)'n' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_TEXT   \
  ((long)'t' + ((long)'x' << 8) + ((long)'t' << 16) + ((long)' ' << 24))
```

```
#define XOFFILE_FORMAT_COMPRESSED \
  ((long)'c' + ((long)'m' << 8) + ((long)'p' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_32 \
  ((long)'0' + ((long)'0' << 8) + ((long)'3' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_64 \
  ((long)'0' + ((long)'0' << 8) + ((long)'6' << 16) + ((long)'4' << 24))
```

Microsoft DirectX 8.1 (C++)

IndexedColor

[This is preliminary documentation and is subject to change.]

Consists of an index parameter and an RGBA color and is used for defining mesh vertex colors. The index defines the vertex to which the color is applied.

UUID

```
<1630B820-7842-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size
index	**DWORD**	
indexColor	**ColorRGBA**	

Microsoft DirectX 8.1 (C++)

Material

[This is preliminary documentation and is subject to change.]

Defines a basic material color that can be applied to either a complete mesh or a mesh's individual faces. The power is the specular exponent of the material. Note that the ambient color requires an alpha component.

TextureFilename is an optional data object. If this object is not present, the face is untextured.

UUID

```
<3D82AB4D-62DA-11cf-AB39-0020AF71E433>
```

Member name	Type	Optional array size	Optional data objects
faceColor	**ColorRGBA**		Any
power	**FLOAT**		
specularColor	**ColorRGB**		
emissiveColor	**ColorRGB**		

Microsoft DirectX 8.1 (C++)

Matrix4x4

[This is preliminary documentation and is subject to change.]

Defines a 4×4 matrix. This is used as a frame transformation matrix.

UUID

```
<F6F23F45-7686-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
matrix	array **FLOAT**	16	None

Microsoft DirectX 8.1 (C++)

Mesh

[This is preliminary documentation and is subject to change.]

Defines a simple mesh. The first array is a list of vertices, and the second array defines the faces of the mesh by indexing into the vertex array.

UUID

```
<3D82AB44-62DA-11cf-AB39-0020AF71E433>
```

Member name	Type	Optional array size	Optional data objects
nVertices	**DWORD**		Any
vertices	array **Vector**	**nVertices**	
nFaces	**DWORD**		
faces	array **MeshFace**	**nFaces**	

Optional Data Elements

The following optional data elements can be used.

Data element	Description
MeshFaceWraps	If this is not present, wrapping for both u and v defaults to false.
MeshTextureCoords	If this is not present, there are no texture coordinates.
MeshNormals	If this is not present, normals are generated using the **GenerateNormals** method of the API.
MeshVertexColors	If this is not present, the colors default to white.
MeshMaterialList	If this is not present, the material defaults to white.

Microsoft DirectX 8.1 (C++)

MeshFace

[This is preliminary documentation and is subject to change.]

Used by the **Mesh** template to define a mesh's faces. Each element of the **nFaceVertexIndices** array references a mesh vertex used to build the face.

UUID

<3D82AB5F-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
nFaceVertexIndices	**DWORD**		None
faceVertexIndices	array **DWORD**	**nFaceVertexIndicies**	

Microsoft DirectX 8.1 (C++)

MeshFaceWraps

[This is preliminary documentation and is subject to change.]

Used to define the texture topology of each face in a wrap. The value of the **nFaceWrapValues** member should be equal to the number of faces in a mesh.

UUID

<4885AE62-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nFaceWrapValues	**DWORD**		None
faceWrapValues	**Boolean2d**		

Microsoft DirectX 8.1 (C++)

MeshMaterialList

[This is preliminary documentation and is subject to change.]

Used in a mesh object to specify which material applies to which faces. The **nMaterials** member specifies how many materials are present, and materials specify which material to apply.

UUID

<F6F23F42-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nMaterials	**DWORD**		**Material**
nFaceIndexes	**DWORD**		
FaceIndexes	array **DWORD**	**nFaceIndexes**	

Microsoft DirectX 8.1 (C++)

MeshNormals

[This is preliminary documentation and is subject to change.]

Defines normals for a mesh. The first array of vectors is the normal vectors themselves, and the second array is an array of indexes specifying which normals should be applied to a given face. The value of the **nFaceNormals** member should be equal to the number of faces in a mesh.

UUID

<F6F23F43-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nNormals	**DWORD**		None
normals	array **Vector**	**nNormals**	
nFaceNormals	**DWORD**		
faceNormals	array **MeshFace**	**nFaceNormals**	

Microsoft DirectX 8.1 (C++)

MeshTextureCoords

[This is preliminary documentation and is subject to change.]

Defines a mesh's texture coordinates.

UUID

<F6F23F40-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nTextureCoords	**DWORD**		None
textureCoords	array **Coords2d**	**nTextureCoords**	

Microsoft DirectX 8.1 (C++)

MeshVertexColors

[This is preliminary documentation and is subject to change.]

Specifies vertex colors for a mesh, instead of applying a material per face or per mesh.

UUID

<1630B821-7842-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nVertexColors	**DWORD**		None
vertexColors	array **IndexedColor**	**nVertexColors**	

Microsoft DirectX 8.1 (C++)

Patch

[This is preliminary documentation and is subject to change.]

Defines a bézier control patch. The array defines the control points for the patch.

UUID

<A3EB5D44-FC22-429D-9AFB-3221CB9719A6>

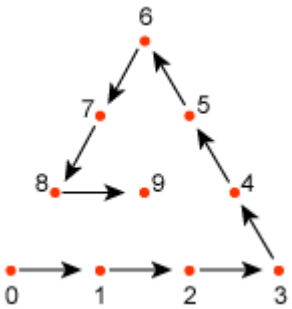
Member name	Type	Optional array size	Optional data objects
nControlIndices	**DWORD**		None
controlIndices	array **DWORD**	**nControlIndices**	

The type of patch is defined by the number of control points, as shown in the following table.

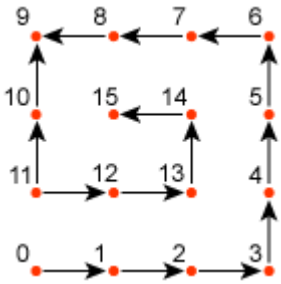
Number of control points	Type
10	Cubic Bézier Triangular Patch
15	Quartic Bézier Triangular Patch
16	Cubic Bézier Quad (Rectangular) Patch

The order of the control points are given in a spiral pattern, as shown in the following diagrams for triangular and rectangular patches.

Triangular patches used the following pattern.



Rectangular patches use the following pattern.



Microsoft DirectX 8.1 (C++)

PatchMesh

[This is preliminary documentation and is subject to change.]

Defines a mesh defined by bézier patches. The first array is a list of vertices, and the second array defines the patches for the mesh by indexing into the vertex array.

UUID

```
<D02C95CC-EDBA-4305-9B5D-1820D7704BBF>
```

Member name	Type	Optional array size	Optional data objects
nVertices	**DWORD**		None
vertices	array **Vector**	**nVertices**	
nPatches	**DWORD**		
patches	array **Patch**	**nPatches**	

The patches use the vertices in the array of vertices as the control points for each patch.

Microsoft DirectX 8.1 (C++)

Quaternion

[This is preliminary documentation and is subject to change.]

Currently unused.

UUID

```
<10DD46A3-775B-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
s	**FLOAT**		None
v	**Vector**		

Microsoft DirectX 8.1 (C++)

SkinWeights

[This is preliminary documentation and is subject to change.]

This template is instantiated on a per-mesh basis. Within a mesh, a sequence of n instances of this template will appear, where *n* is the number of bones (X file frames) that influence the vertices in the mesh. Each instance of the template basically defines the influence of a particular bone on the mesh. There is a list of vertex indices, and a corresponding list of weights.

UUID

```
<6F0D123B-BAD2-4167-A0D0-80224F25FABB>
```

Member name	Type	Optional array size	Optional data objects
transformNodeName	**STRING**		None
nWeights	**DWORD**		
vertexIndices	array **DWORD**	**nWeights**	
weights	array float	**nWeights**	
matrixOffset	**Matrix4x4**		

The name of the bone whose influence is being defined is **transformNodeName**, and **nWeights** is the number of vertices affected by this bone. The vertices influenced by this bone are contained in **vertexIndices**, and the weights for each of the vertices influenced by this bone are contained in **weights**.

The matrix **matrixOffset** transforms the mesh vertices to the space of the bone. When concatenated to the bone's transform, this provides the world space coordinates of the mesh as affected by the bone.

Microsoft DirectX 8.1 (C++)

TextureFilename

[This is preliminary documentation and is subject to change.]

Enables you to specify the file name of a texture to apply to a mesh or a face. This template should appear within a material object.

UUID

```
<A42790E1-7810-11cf-8F52-0040333594A3>
```

Member name	Type	Optional array size	Optional data objects
filename	**STRING**		None

Microsoft DirectX 8.1 (C++)

TimedFloatKeys

[This is preliminary documentation and is subject to change.]

Defines a set of floats and a positive time used in animations.

UUID

<F406B180-7B3B-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
time	**DWORD**		None
tfkeys	**FloatKeys**		

Microsoft DirectX 8.1 (C++)

Vector

[This is preliminary documentation and is subject to change.]

Defines a vector.

UUID

<3D82AB5E-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
x	**FLOAT**		None
y	**FLOAT**		
z	**FLOAT**		

Microsoft DirectX 8.1 (C++)

VertexDuplicationIndices

[This is preliminary documentation and is subject to change.]

This template is instantiated on a per-mesh basis, holding information about which vertices in the mesh are duplicates of each other. Duplicates result when a vertex sits on a smoothing group or material boundary. The purpose of this template is to allow the loader to determine which vertices exhibiting different peripheral parameters are actually the same vertexes in the model. Certain applications (mesh simplification for example) can make use of this information.

UUID

<B8D65549-D7C9-4995-89CF-53A9A8B031E3>

Member name	Type	Optional array size	Optional data objects

nIndices	**DWORD**		None
nOriginalVertices	**DWORD**		
indices	array **DWORD**	**nIndices**	

The number of vertices in the mesh before any duplication occurred is **nOriginalVertices**. The value **indices**[*n*] holds the vertex index that vertex[*n*] in the vertex array for the mesh would have had if no duplication had occurred. So any indices in this array that are the same indicate duplicate vertices.

Microsoft DirectX 8.1 (C++)

XSkinMeshHeader

[This is preliminary documentation and is subject to change.]

This template is instantiated on a per-mesh basis only in meshes that contain exported skinning information. The purpose of this template is to provide information about the nature of the skinning information that was exported.

UUID

```
<3CF169CE-FF7C-44ab-93C0-F78F62D172E2>
```

Member name	Type	Optional array size	Optional data objects
nMaxSkinWeightsPerVertex	**WORD**		None
nMaxSkinWeightsPerFace	**WORD**		
nBones	**WORD**		

The maximum number of transforms that affect a vertex in the mesh is **nMaxSkinWeightsPerVertex**. The maximum number of unique transforms that affect the three vertices of any face is **nMaxSkinWeightsPerFace**. The number of bones that affect vertices in this mesh is **nBones**.

Microsoft DirectX 8.1 (C++)

Binary Format

[This is preliminary documentation and is subject to change.]

This section details the binary version of the Microsoft® DirectX® .x file format as introduced with the release of DirectX 3.0. The information presented in this section should be read in conjunction with the X File Architecture section.

The binary format is a tokenized representation of the text format. Tokens can be stand-alone or accompanied by primitive data records. Stand-alone tokens give grammatical structure, and record-bearing tokens supply the necessary data.

Note that all data is stored in little-endian format. A valid binary data stream consists of a header followed by templates and/or data objects.

This section discusses the following components of the binary file format and provides an example template and binary data object.

- Header
- Tokens
- Token Records
- Templates
- Data
- Examples

Microsoft DirectX 8.1 (C++)

Tokens

[This is preliminary documentation and is subject to change.]

Tokens are written as little-endian **WORD**s. The following list of token values is divided into record-bearing and stand-alone tokens.

The record-bearing tokens are defined as follows.

```
#define TOKEN_NAME 1
#define TOKEN_STRING 2
#define TOKEN_INTEGER 3
#define TOKEN_GUID 5
#define TOKEN_INTEGER_LIST 6
#define TOKEN_FLOAT_LIST 7
```

The stand-alone tokens are defined as follows.

```
#define TOKEN_OBRACE 10
#define TOKEN_CBRACE 11
#define TOKEN_OPAREN 12
#define TOKEN_CPAREN 13
#define TOKEN_OBRACKET 14
#define TOKEN_CBRACKET 15
#define TOKEN_OANGLE 16
#define TOKEN_CANGLE 17
#define TOKEN_DOT 18
#define TOKEN_COMMA 19
#define TOKEN_SEMICOLON 20
#define TOKEN_TEMPLATE 31
#define TOKEN_WORD 40
#define TOKEN_DWORD 41
#define TOKEN_FLOAT 42
#define TOKEN_DOUBLE 43
#define TOKEN_CHAR 44
#define TOKEN_UCHAR 45
#define TOKEN_SWORD 46
#define TOKEN_SDWORD 47
#define TOKEN_VOID 48
#define TOKEN_LPSTR 49
#define TOKEN_UNICODE 50
#define TOKEN_CSTRING 51
#define TOKEN_ARRAY 52
```

Microsoft DirectX 8.1 (C++)

Token Records

[This is preliminary documentation and is subject to change.]

This section describes the format of the records for each of the record-bearing tokens. Information is divided into the following sections.

TOKEN_NAME

A variable-length record. The token is followed by a *count* value that specifies the number of bytes that follow in the *name* field. An ASCII name of length *count* completes the record.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_NAME**
count	**DWORD**	4	Length of name field, in bytes
name	**BYTE** array	**count**	ASCII name

TOKEN_STRING

A variable-length record. The token is followed by a count value that specifies the number of bytes that follow in the string field. An ASCII string of length count continues the record, which is completed by a terminating token. The choice of terminator is determined by syntax issues discussed elsewhere.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_STRING**
count	**DWORD**	4	Length of string field in bytes
string	**BYTE** array	**count**	ASCII string
terminator	**DWORD**	4	**TOKEN_SEMICOLON** or **TOKEN_COMMA**

TOKEN_INTEGER

A fixed length record. The token is followed by the integer value required.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_INTEGER**
value	**DWORD**	4	Single integer

TOKEN_GUID

A fixed-length record. The token is followed by the four data fields as defined by the OSF DCE standard.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_GUID**

data1	**DWORD**	4	UUID data field 1
data2	**WORD**	2	UUID data field 2
data3	**WORD**	2	UUID data field 3
data4	**BYTE** array	8	UUID data field 4

TOKEN_INTEGER_LIST

A variable-length record. The token is followed by a count value that specifies the number of integers that follow in the list field. For efficiency, consecutive integer lists should be compounded into a single list.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_INTEGER_LIST**
count	**DWORD**	4	Number of integers in list field
list	**DWORD**	4 × **count**	Integer list

TOKEN_FLOAT_LIST

A variable-length record. The token is followed by a count value that specifies the number of floats or doubles that follow in the list field. The size of the floating point value (float or double) is determined by the value of float size specified in the file header. For efficiency, consecutive **TOKEN_FLOAT_LIST**s should be compounded into a single list.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_FLOAT_LIST**
count	**DWORD**	4	Number of floats or doubles in list field
list	float/double array	4 or 8 × **count**	Float or double list

Microsoft DirectX 8.1 (C++)

Mesh View Help

[This is preliminary documentation and is subject to change.]

Menu Descriptions

File

- **Open Mesh File**

  Opens a dialog to select a file in the .x or .m file format to be loaded and viewed.

- **Open PMesh File**

  Opens a dialog to select a file in the progressive mesh format to be loaded and viewed.

- **Create Shape**

Opens a sub-dialog to create some basic shapes that are defined programmatically (text, polygon, box, cylinder, torus, teapot, sphere, cone).

- **Save Mesh As**

  Opens a dialog to save the selected mesh to a file. The file can be written as a text or binary file.

- **Close Selected**

  Closes and deletes the currently selected mesh.

- **Close Non Selected**

  Closes and deletes the meshes that are not currently selected.

View

- **Wireframe**

  View all content in wireframe mode.

- **Edges**

  View all content in solid-shaded mode with the edges drawn in black.

- **Creases**

  Highlight the creases on the visible meshes. A crease is an edge with a vertex that has a different piece of data on it for multiple faces that refer to it, that is, a different normal for the vertex per face.

- **Strips**

  Show the strips that are generated by this mesh in blue. The blue line goes from the center of each triangle to the next triangle in the triangle strip.

- **Adjacency**

  Show the adjacency of the polygons in a mesh by drawing a line from the center of a polygon to the center of the adjacent polygon.

- **Bounding Box**

  Draw the bounding boxes for the visible meshes.

- **Normals**

  Draw the normals of the vertices on the visible meshes in yellow.

- **Texture Coords**

  Show the texture coordinates for the viewed geometry as rays projecting from the

vertices. Because a vertex in Microsoft® Direct3D® can have up to eight texture coordinates, users must specify which sets they would like to view. This viewing mode is especially useful when the texture coordinates are filled with tangents for use in pixel shaders for example.

- **Textures**

  Display the textures on the visible geometry.

- **Lighting**

  Show the geometry in the scene with lighting calculations still on.

- **Culling**

  Perform back face culling on the visible geometry when disabled polygons facing away from the camera are not drawn.

- **Hierarchy**

  Display the frame hierarchy of the meshes that are currently loaded. This is displayed in a separate floating window. To make the window disappear, on the **View** menu, clear the **Hierarchy** command.

- **Play Animation**

  Play the current animation for the currently loaded geometry if one exists.

- **Pause Animation**

  Stop the animation at the current frame when playing.

- **Normal Speed**

  Interpret the time value in the animation from an X file as 4800 units per second. Otherwise, the interpreted value is 30 units per second.

MeshOps

- **Optimize**

  Optimizes the currently selected mesh with the selected optimization method. See the Microsoft® Direct3DX reference pages to see the differences in methods.

- **Weld Vertices**

  Removes duplicate vertices and makes polygons that use these vertices use the nondeleted vertex.

- **Split Mesh**

  Splits the selected mesh into multiple meshes that are less than specified size in vertices and faces.

- **Collapse Meshes**

  Collapses the currently selected meshes into a single mesh.

- **Reset Matrices**

  Resets the matrices for the frames that are loaded to their initial position.

- **Mesh Properties**

  Shows the selected mesh's FVF render states and whether or not it is a 32 bit mesh.

- **Skinning Method**

  Allows the user to select the skinning method while animating a skinned mesh. The choices are nonindexed, indexed, and software skinning.

- **Face Selection**

  Enters a mode for the user to select an individual face on the current mesh.

- **Vertex Selection**

  Enters a mode for the user to select an individual vertex on the current mesh.

PMeshes

- **Convert Selected to PM**

  Convert the selected mesh to a progressive mesh. The conversion uses the error parameters that are entered in a dialog box. For more information on these parameters refer to the Direct3DX documentation of progressive meshes.

- **Snapshot to Mesh**

  Convert the current progressive mesh object to a static mesh object using the current settings of the progressive mesh.

- **Set number of Faces**

  Set the current number of faces in the progressive mesh to a specific number.

- **Set number of Vertices**

  Set the current number of vertices in the progressive mesh to a specific number.

- **Trim**

  Set the minimum and maximum number of faces for a progressive mesh. Once the user has set the trim values to the desired minimum and maximum, the progressive mesh can be trimmed to the selected values, thereby reducing the dynamic range of the progressive mesh.

N-Patches

- **N-Patch Selected**

  Draw the current object as an N-Patches object. The scroll bar in this mode selects the amount of N-Patch iterations for the current object.

- **SnapShot to mesh**

  Convert the selected object to a static mesh based on the current N-Patch settings to create a high-resolution static mesh.

Icons and Usage

Icons

- **Selection Modes**

  The first three icons are easy ways for the user to select the selection mode. They are Mesh Selection Mode (Arrow), Face Selection Mode (yellow outlined triangle), and Vertex Selection Mode (Red point highlighted triangle). These are the same modes that are available from the menus in MeshOps.

- **Display Modes**

  The next icons are easy ways to select the most common display modes for geometry. They are Shaded mode (nonoutlined tri-color cube), Wire frame mode (wire frame cube), and Edge mode (outlined tri-color cube). These are the same modes that are available from the menus in View.

- **Topology Display modes**

  The next icons display specific topological information about the geometry displayed. They are adjacency (A), Strips (S), Creases (C), and Normals (N). These are the same modes that are available from the menus in View.

- **Info**

  The next icon is the info button that will display information about the currently selected element.

- **Animation Controls**

  The last two icons are animation controls for playing and pausing the animation for the currently visible mesh.

Status Bar

The status bar in MView displays the current status of the visible geometry. The order from left to right of the displayed information is currently selected element (face or vertex only), Mesh mode (polygon, Pmesh, or pMesh), display frames per second, display triangles per second, number of displayed triangles, and number of displayed vertices.

Scroll Bar

The scroll bar will appear in two of the three mesh modes, pMesh and nPatch mode. In pMesh mode, the scroll bar indicates the range of displayed triangles for the progressive mesh. You can slide the scroll bar up or down to change the number of triangles displayed. In nPatch mode, the scroll bar indicates how many nPatch levels are being used. As the scroll bar is moved up or down, the number of nPatch interactions performed are adjusted accordingly.

More Information

For more information on any of the functions used by MView, refer to the Direct3DX documentation included in the Microsoft® DirectX® software development kit (SDK).