Tutorial 1: Creating a Device

To use Direct3D, you first create an application window, then you create and initialize Direct3D objects. You use the component object model (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.

Steps

- 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\C++\Direct3D\Tutorials\Tut01_CreateDevice

Step 1 - Creating a Window

The first thing any Windows application must do when it is run is to 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,
        "Direct3D Tutorial", NULL };

        RegisterClassEx( &wc );

        // Create the application's window.
        HWND hWnd = CreateWindow( "Direct3D Tutorial", "Direct3D 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 "Direct3D 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.) When the window is created, the code sample calls standard Win32 functions to display and update the window.

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

Step 2 - Initializing Direct3D

The CreateDevice sample project performs 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 the object, setting the presentation parameters, and finally creating the Direct3D device.

After creating a Direct3D object, use the **IDirect3D9::CreateDevice** method to create a device, and to enumerate devices, types, modes, and so on.

```
if( NULL == ( g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
        return E_FAIL;
```

The only parameter passed to **Direct3DCreate9** 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, **Direct3DCreate9** will fail.

By filling in the fields of **D3DPRESENT_PARAMETERS** you can specify how you want your 3D application to behave. The CreateDevice sample project sets Windowed to TRUE, SwapEffect to D3DSWAPEFFECT_DISCARD, and BackBufferFormat to D3DFMT_UNKNOWN.

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
```

The final step is to use the **IDirect3D9::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 object 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.

Step 3 - Handling System Messages

After you have created the application window and initialized Direct3D, you are ready to render the scene. In most cases, 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 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.

Step 4 - Rendering and Displaying a Scene

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 **IDirect3DDevice9::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 **IDirect3DDevice9::Clear** inform 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 zero 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 **IDirect3DDevice9::Clear** method because the corresponding flags are not present.

After clearing the viewport, the CreateDevice sample project informs Direct3D that rendering will begin, then 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 **IDirect3DDevice9::BeginScene** and **ID3DXRenderToSurface::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 **ID3DXRenderToSurface::EndScene** before calling **IDirect3DDevice9::BeginScene** Again.

After rendering the scene, you display it by using the **IDirect3DDevice9::Present** method.

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

The first two parameters 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.

Step 5 - Shutting Down

At some point during execution, your application must shut down. Shutting down a 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 Direct3D objects it uses by calling the IUnknown 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.