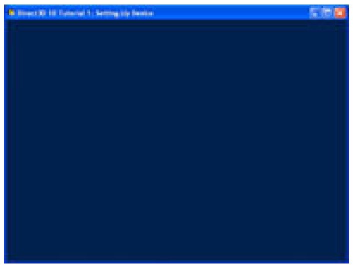
Tutorial 1: Direct3D 10 Basics

Collapse All



Summary

In this first tutorial, we will go through the elements necessary to create a minimal Direct3D 10 application. Every Direct3D 10 application must have these elements to function properly. The elements include setting up a window and a device object then displaying a color on the window.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial01

Setting Up The Direct3D 10 Device

Now that we have a window displaying, we can continue to set up a Direct3D 10 device, which is necessary if we are going to render any 3D scene. The first thing to do is to create two objects: a device and a swap chain.

The device object is used by the application to perform rendering onto a buffer. The device also contains methods to create resources.

The swap chain is responsible for taking the buffer that the device renders to and displaying the content on the actual monitor screen. The swap chain contains two or more buffers, mainly the front and the back. These are textures that the device renders to, for displaying on the monitor. The front buffer is what is currently being presented to the user. This buffer is read-only and cannot be modified. The back buffer is the render target that the device will draw to. Once it finishes the drawing operation, the swap chain will present the backbuffer, by swapping the two buffers. The back buffer becomes the front buffer, and vice versa.

To create the swap chain, we fill out a DXGI_SWAPCHAIN_DESC structure that describes the swap chain we are about to create. A few fields are worth mentioning. BackBufferUsage is a flag that tells the application how the back buffer will be used. In this case, we want to render to the back buffer, so we'll set BackBufferUsage to DXGI_USAGE_RENDER_TARGET_OUTPUT. The OutputWindow field represents the window that the swap chain will use to present images on the screen. SampleDesc is used to enable multi-sampling. Since this tutorial does not use multi-sampling, SampleDesc's Count is set to 1 and Quality to 0 to have multi-sampling disabled.

Once the description has been filled out, we can call the D3D10CreateDeviceAndSwapChaing function to create both the device and the swap chain for us. The code to create a device and a swap chain is listed below:

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

if( FAILED( D3D10CreateDeviceAndSwapChain( NULL, D3D10_DRIVER_TYPE_REFERENCE, NULL,
                 0, D3D10_SDK_VERSION, &sd, &g_pSwapChain, &g_pd3dDevice ) ) )
{
    return FALSE;
}
```

The next thing that we need to do is to create a render target view. A render target view is a type of resource view in Direct3D 10. A resource view allows a resource to be bound to the graphics pipeline at a specific stage. Think of resource views as typecast in C. A chunk of raw memory in C can be cast to any data type. We can cast that chunk of memory to an array of integers, an array of floats, a structure, an array of structures, so on. The raw memory itself is not very useful to us if we don't know its type. Direct3D 10 resource views act in a similar way. For instance a 2D texture, analogous to the raw memory chunk, is the raw underlying resource. Once we have that resource we can create different resource views to bind that texture to different stages in the graphics pipeline with different formats: as a render target to render to, as a depth stencil buffer that will receive depth information, or as a texture resource. Where C typecasts allow a memory chunk to be used in a different manner, so do Direct3D 10 resource views.

We need to create a render target view because we would like to bind the back buffer of our swap chain as a render target, so that Direct3D 10 can render onto it. We first call GetBuffer() to obtain the back buffer object. Optionally, we can fill in a D3D10_RENDERTARGETVIEW_DESC structure that describes the render target view to be created. This description is normally the second parameter to CreateRenderTargetView. However, for these tutorials, the default render target view will suffice. The default render target view can be obtained by passing NULL as the second parameter. Once we have created the render target view, we can call OMSetRenderTargets() to bind it to the pipeline so that the output that the pipeline renders gets written to the back buffer. The code to create and set the render target view is listed below:

```
// Create a render target view
ID3D10Texture2D *pBackBuffer;
if( FAILED( g_pSwapChain->GetBuffer( 0, __uuidof( ID3D10Texture2D ), (LPVOID*)&pBackBuffer ) ) )
    return FALSE;
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL, &g_pRenderTargetView );
pBackBuffer->Release();
if( FAILED( hr ) )
    return FALSE;
g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

The last thing that we need to set up before Direct3D 10 can render is initialize the viewport. The viewport maps clip space coordinates, where X and Y range from -1 to 1 and Z ranges from 0 to 1, to render target space, sometimes known as pixel space. In Direct3D 9, if the application does not set up a viewport, a default viewport is set up to be the same size as the render target. In Direct3D 10, no viewport is set by default. Therefore, we must do so before we can see anything on the screen. Since we would like to use the entire render target for the output, we set the top left point to (0, 0) and width and height to be identical to the render target's size. The code to do so is shown below:

```
D3D10_VIEWPORT vp;
vp.Width = 640;
vp.Height = 480;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pd3dDevice->RSSetViewports( 1, &vp );
```

Modifying the Message Loop

We have set up the window and Direct3D 10 device, and we are ready to render. However, there is still a problem with our message loop: it uses GetMessage() to obtain messages. The problem with GetMessage() is that if there is no message in the queue for the application window, GetMessage() blocks and does not return until a message is available. Thus, instead of doing something like rendering, our application is waiting within GetMessage() when the message queue is empty. We can solve this problem by using PeekMessage() instead of GetMessage(). PeekMessage() can retrieve a message like GetMessage() does, but when there is no message waiting, PeekMessage() returns immediately instead of blocking. We can then take this time to do some rendering. The modified message loop, which uses PeekMessage(), looks like this:

```
MSG msg = {0};
while( WM_QUIT != msg.message )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else
    {
        Render();  // Do some rendering
    }
}
```

The Rendering Code

Rendering is done in the Render() function. In this tutorial, we will render the simplest scene possible, which is to fill the screen with a single color. In Direct3D 10, an easy way to fill the render target with a single color is to use the device's ClearRenderTargetView() method. We first define a D3D10_COLOR structure that describes the color we would like to fill the screen with, then pass it to ClearRenderTargetView(). In this example, a shade of blue is chosen. Once we have filled our back buffer, we call the swap chain's Present() method to complete the rendering. Present() is responsible for displaying the swap chain's back buffer content onto the screen so that the user can see it. The Render() function looks like below:

```
void Render()
{
    //
    // Clear the backbuffer
    //
    float ClearColor[4] = { 0.0f, 0.125f, 0.6f, 1.0f }; // RGBA
    g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView, ClearColor );

    g_pSwapChain->Present( 0, 0 );
}
```