D3D11 Review

Main D3D11 Objects

In order to perform standard rendering processes with D3D11 we will need the following objects:

ID3D11Device - Used to create resources, ex. ID3D11Device::CreateBuffer

ID3D11DeviceContext - Used to generate rendering commands, ex. ID3D11DeviceContext::DrawIndexed

IDXGISwapChain - Handles how we present the back buffer to the output, ex. IDXGISwapChain::Present

Main D3D11 Objects continued

Under typical usage we can create all three of these objects with one function call to D3D11CreateDeviceAndSwapChain. One complex, 12 parameter function call...we will go over the trickier parts.

```
HRESULT D3D11CreateDeviceAndSwapChain(
  _In_   IDXGIAdapter *pAdapter,
  _In_   D3D_DRIVER_TYPE DriverType,
  _In_   HMODULE Software,
  _In_   UINT Flags,
  _In_   const D3D_FEATURE_LEVEL *pFeatureLevels,
  _In_   UINT FeatureLevels,
  _In_   UINT SDKVersion,
  _In_   const DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
  _Out_  IDXGISwapChain **ppSwapChain,
  _Out_  ID3D11Device **ppDevice,
  _Out_  D3D_FEATURE_LEVEL *pFeatureLevel,
  _Out_  ID3D11DeviceContext**ppImmediateContext
```

D3D11CreateDeviceAndSwapChain

_In_ const D3D_FEATURE_LEVEL *pFeatureLevels - Should point to an array of the feature levels you wish to support. This could just be just D3D_FEATURE_LEVEL_11_0 or a collection of viable options. Null defaults to try all from DX11 to DX9.1.

Out D3D_FEATURE_LEVEL *pFeatureLevel - The feature level supported. This will be the highest available level provided in pFeatureLevels. Use this to test if the needed feature level is supported.

_In_ const DXGI_SWAP_CHAIN_DESC *pSwapChainDesc - You will need to fully define an instance of the DXGI_SWAP_CHAIN_DESC structure type to the desired settings. You will very likely need to look up this object on MSDN to find the proper settings for your particular needs.

Additional typical use initialization

We need to tell D3D11 where to render. We do this with the OMSetRenderTargets method. This method will need a ID3D11RenderTargetView and a ID3D11DepthStencilView to be used as the active targets.

OMSetRenderTargets actually takes in an array of ID3D11RenderTargetView pointers, as opposed to just a single target. We can use this functionality to allow us to write to multiple targets at one time. We will use this feature when we get to the deferred shading topic.

Getting the main render target

1. Get a ID3D11Texture2D pointer to the back buffer.
   a. ID3D11Texture2D *backBufferPtr = nullptr;
   b. theSwapChainPtr->GetBuffer(0, __uuidof (theBackBufferPtr), reinterpret_cast<void**>(&theBackBufferPtr));

2. Get the main render target via the ID3D11Device:: CreateRenderTargetView method, passing theBackBufferPtr as the first parameter.

3. Move on to getting the depth-stencil view.

Getting the main depth-stencil view

- Create a ID3D11Texture2D interface instance through the use of the ID3D11Device::CreateTexture2D method.

  a. You will need to fill out a D3D11_TEXTURE2D_DESC to describe the usage of this texture as the depth-stencil buffer.

     i. You may wish to call GetDesc from your swap chain interface to match parts of the description to.

     ii. BindFlags are what we set to define if this is to be used as a depth and/or stencil buffer.

     iii. Use a typeless format, such as DXGI_FORMAT_R24G8_TYPELESS, if you will later need to read from this asset in a shader. We will need this support in our rendering engine.

Getting the main depth-stencil view continued

- Create a ID3D11DepthStencilView interface instance through the use of the ID3D11Device::CreateDepthStencilView method.

    a. You will need to fill out a D3D11_DEPTH_STENCIL_VIEW_DESC to describe the usage of this view.

        i. Since we used a typeless format when creating the texture, we will need to use a more fleshed out compatible format, such as DXGI_FORMAT_D24_UNORM_S8_UINT

        ii. ViewDimension - use D3D11_DSV_DIMENSION_TEXTURE2D, to create a view based on a 2d texture.

- We now have the views for our render target and our depth-stencil target, so we can call OMSetRenderTargets

Viewport

- The last thing we need to define and set is the viewport.

- In order to set the viewport we call the ID3D11DeviceContext::RSSetViewports method, passing a filled out D3D11_VIEWPORT instance.
    - Remember, you can use the GetDesc method of your swap chain to get information like the size of your back buffer.

That was a lot of text

Here is a picture of your teacher from the 1800s.

The Render Loop

Each frame we will do the following tasks:

1. Clear the targets we will be using for rendering.

2. ...Render stuff...

3. Present what we have rendered

We will focus on step 1 and 3 first, since they are easier and testable.

Clearing our render targets

- Each render target can be cleared to a defined color.
    - We clear a render target by calling the ID3D11DeviceContext::ClearRenderTargetView method, passing it the render target view to be cleared and the color the target should be cleared to.
- Each depth-stencil target can be cleared to defined values.
    - We clear depth-stencil buffers with the ID3D11DeviceContext::ClearDepthStencilView method, passing the view to be cleared, flags defining depth and or stencil, then values to clear to.

Presenting the rendered image to the screen

- In order to present, call the IDXGISwapChain::Present method. Pass this method 0s for both parameters for our general use. These parameters control some aspects of presentation such as the ability to wait for vertical refreshes before presenting, aka Vsync.

- At this point we should be able to run and see our clear color. If you are not testing this, you are wrong. =)

"...Render stuff..." we need stuff

For basic rendering we will need the following:

1. ID3D11Buffer vertexBuffer - the buffer with the vertex data in it
2. ID3D11Buffer indexBuffer - the buffer with the index data in it
3. ID3D11VertexShader vertexShader - the interface to the vertex shader to be used
4. ID3D11PixelShader pixelShader - the interface to the pixel shader to be used
5. ID3D11InputLayout inputLayout - this object will define how a vertex is laid out

ID3D11Buffer

- Provides an interface to unstructured data on the video card.
- The most common use of the ID3D11Buffer interface is to store and act on vertex and index data. This interface will also be used to interact with constant buffers in shaders.
- ID3D11Buffer usage is abstracted so the code for an index buffer or a vertex buffer will be almost identical.

Creating buffer with data

- To create an ID3D11Buffer we use the ID3D11Device::CreateBuffer method.
- The D3D11_BUFFER_DESC parameter will define whether this is index or vertex data, how big a buffer to create, as well as how we will be accessing the data from the CPU if at all.
- The optional D3D11_SUBRESOURCE_DATA parameter can be used to set the data in the buffer upon creation. Simply set the pSysMem member to point at contiguous data you want to be written to the buffer upon creation.
- D3D11_SUBRESOURCE_DATA data;
- data.pSysMem = &someStlVectorOVertsOrIndices [0];

Changing data in ID3D11Buffer

- Setting the data in a buffer when it is created allows us to disable CPU access to the buffer which is an optimization we can use for static geometry and other data sets that will not change.

- For dynamic ID3D11Buffer interface's we can use the Map method to gain access to the data, read from or write to the data, then use the Unmap method to update the GPU with our changes.

- If creating a dynamic buffer, be sure to set the CPUAccessFlags appropriately to your usage when creating the buffer.

Vertex and Pixel shader code

- Before we can create interfaces to our shader code we need... shader code.
- Visual Studio 2012 can add, color and compile HLSL shaders by default.
- Simply add a new item to your project through the right-click menu as normal, but select the HLSL tab, then the type of shader desired.
- By default this new shader will be compiled with your project and will output a binary of the same name with the .cso extension, if there are no errors.
- Many compile and output setting can be adjusted per-file through the properties of the file.

Loading a shader from a compiled file (.cso)

- To create interfaces to shaders we will use the ID3D11Device::CreateVertexShader or CreatePixelShader methods. These methods will create our pixelShader and vertexShader instances.

- The first parameter of either of these methods is the array of data representing our shader code. We will need to read our compiled shader binary file (.cso) into an array for this parameter. The second parameter is the size of the data in the first.

- Pass null for the pClassLinkage parameter. At the time of this writing I do not fully understand its use. =)

ID3D11InputLayout

- To create an ID3D11InputLayout call the ID3D11Device::CreateInputLayout method.
- The pInputElementDescs paramter should be set to an array of D3D11_INPUT_ELEMENT_DESC that define your vertex. Here is an example with positions, normals and UVs:
- D3D11_INPUT_ELEMENT_DESC vertexPosNormTexDesc[] =

```
{
          {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0},
          {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0},
          {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0},
};
```

- SemanticName, SemanticIndex, Format, InputSlot, AlignedByteOffset, InputSlotClass, InstanceDataStepRate

ID3D11InputLayout continued...

- The ID3D11Device::CreateInputLayout method also takes in an array of shader byte code to validate the desired input layout against.
- This "feature", to my understanding, is all but pointless.
- You cannot skip this, you will have to provide the byte code to a vertex shader when creating these layouts.
- You are not locked into using the same shader file you used for validation when you use this input layout for rendering.
- The validation is extremely loose. Try making an input layout with positions and colors, then validate it against a vertex shader that only takes in positions. Yeah that will pass validation just fine even though the layout and input vertex do not match.
- For my use, I create a set of very simple vertex shaders that have no purpose other than for use during validation. Ex:

```
float4 main(float3 pos:POSITION0)
{
	return float4(pos, 1);
}
```

"...Render stuff..." we have stuff

We should have what we need to render now, but first another silly picture of your teacher.

Steps to render

- In order to render we will need to do the following steps:
  - Set input layout to be used with the ID3D11DeviceContext::IASetInputLayout
  - Set the type of primitive being used with ID3D11DeviceContext::IASetPrimitiveTopology
  - Set the vertex buffer to be used through the ID3D11DeviceContext::IASetVertexBuffers method
  - Set shaders to be used through the ID3D11DeviceContext::VSSetShader and PSSetShader methods.
  - Set index buffer to be used with the ID3D11DeviceContext::IASetIndexBuffer method.

Steps to render continued

- Continued from previous slide:
  - Set shader values -
    - All shader variables you wish to access CPU side will need to be in constant buffers in your shader code.
    - Once you have a const buffer in your shader, a ID3D11Buffer object will need to be created with the same size as the constant buffer.
    - Each time we want to change a shader variable we can use the ID3D11DeviceContext::Map and Unmap methods.
    - Once the data is set in our ID3D11Buffer instance, we can write it to the constant buffer shader side through the ID3D11DeviceContext::VSSetConstantBuffers and PSSetConstantBuffers methods
  - We are (finally) ready to make a draw call like ID3D11DeviceContext::DrawIndexed

Visual Studio 2012 Graphics Debugger

- VS2012 comes with powerful tools for debugging and profiling graphics code
- To use these tools, under the debug tab, open the graphics tab then select Start Diagnostic, usually alt+f5
- Your application should run as normal, but whenever you press Print Screen a frame capture will occur that can then be analyzed.
- Under Debug->Graphics you will have the option to open a few graphics debugging windows

Visual Studio 2012 Graphics Debugger

- Your Graphics Experiment window should be open by default and have a screenshot of your application. You can zoom and pan this image as well as select specific pixels to evaluate and debug.
- Event List shows all the DX methods that have been run this frame. Selecting one of these events should change the experiment window to show what your rendering looked like when that method was called.
- Pixel History will show each time the selected pixel has changed color, as well as letting you step through the shaders involved in that change.
- Pipeline Stages will show, with pictures, the results of each stage of the rendering pipeline for the currently selected draw call. You can debug these stages.

Visual Studio 2012 Graphics Debugger

- Graphics Object Table - This window will show a list of all the DX objects that existed during the life of the frame. You can double click these items to see them or at least see information about them.
    - You can set the "name" that shows up in this table by calling the following method on most D3D objects:
    - d3dObjInterface->SetPrivateData (WKPDID_D3DDebugObjectName, length of name, name);
    - I'd make a macro for this...
- Graphics Event Call Stack - This window will show your CPU side call stack at the time of a selected even from the Event List
- Holy crap do you have it good. In my day we wrote shaders in notepad then crossed our fingers.