**Everything I Did to Render a Cube on a Platform using DirectX 12 on Windows**

**Error Handling**

**DirectX Error Handling**

Many DirectX functions return a HRESULT value. The value tells if you the function has failed or succeeded. To check if the function failed or succeeded, I made a DirectXException class, and a macro function called ThrowIfFailed(x). When you make an object of the class it stores a HRESULT value, the name of the function, the file and the line number where the error happened. The class has an errorMsg() function that I called to print out a the error message associated with the HRESULT value.

In the ThrowIfFailed(x) macro function if the function failed, I threw a DirectXException object and printed out the error message using a MessageBox.

**Windows Error Handling**

If a Windows API function fails you retrieve the error code by calling the function GetLastError(). I threw a runtime\_error object with the error code attached with a self-made error message.

I displayed the error message with a MessageBox.

**Window Creation**

**Entry Point for Win32**

int WINAPI wWinMain(\_In\_ HINSTANCE hInstance, \_In\_opt\_ HINSTANCE hPrevInstance, \_In\_ PWSTR pCmdline, \_In\_ int nCmdShow).

Most important parameter is the hInstance which is a value that identifies the executable when loaded in memory. It is needed for certain window functions.

**Making a Window**

I filled out a WNDCLASSEX struct to describe the window. I registered the window class with the operating system using the Windows API function RegisterClassEx() which takes in a pointer to a WNDCLASSEX object.

In a window there is the client area and non-client area. The non-client area is the title bar and borders of the window. The client area is below the title bar.

The client area is where scenes get rendered to. To get the client area to be a desired size I filled in a RECT object with the desired size and called the window API function AdjustWindowRect(). The function takes in a RECT object and in that RECT object it will store the size the of the window based on the desired client size.

I used that RECT object to create the window by calling the Windows API function CreateWindow(). The CreateWindow() function returns a handle to the window.

To display the window, you call the Windows API functions ShowWindow() which takes in a handle to a window.

**Window Messages**

In a WNDCLASSEX struct it stores a function to a window procedure. A window procedure is a callback function used to handle events. Every window has a message queue. Anytime an event happens (eg. Closing the window) it gets put into the windows message queue where the event happened. A message is retrieved from the windows message queue and sent to the window procedure.

The user must have a window procedure for every window they create. The function prototype for a window procedure is:

LRESULT CALLBACK windowProcedureName(HWND handle, UINT msg, WPARAM wParam, LPARAM lParam).

In the function I used a switch statement to check what event happened through the message that was passed to the window procedure and defined how to handle some messages.

To retrieve messages from the message queue, I called the Windows API function PeekMessage(). The function checks if there is a message in the queue. If there was a message in the queue, I called the Windows API functions TranslateMessage() and DispatchMessage(). TranslateMessage translates virtual-key messages into character messages. DispatchMessage calls your window procedure function.

**COM**

A component object model (COM) object is thought of as an interface and is used like a C++ class object. COM objects are reference counted like a shared pointer. When a COM objects reference count hits 0 we must call its Release method to free its memory. We use the ComPtr class from the wrl library to help us manage the lifetime of a COM object. The class is like smart pointers in C++. When a COM object goes out of scope the Release function for the COM object will automatically be called.

To retrieve a pointer to the COM object through a ComPtr we call the Get() function.

To retrieve a pointer to the pointer that points to the COM object through a ComPtr we call the GetAddressOf() functions.

To call Release and then retrieve a pointer to the pointer that points to the COM object we use & before the ComPtr object.

I used a ComPtr for every COM interface.

**Synchronization of the CPU and GPU**

The CPU and GPU are two different processors that work in parallel to render objects. Because they are in parallel and can access the same resources we have to synchronize them to make sure they access resources in the proper order. We don’t want to do to many synchronizations because we want to keep both processors as busy as possible, and when we synchronize it means one processor is waiting for the other to finish.

The GPU has a command queue. The CPU submits commands to the queue through the Direct3D API using a command list. Instructions are put into a command allocator through a command list. When we want to execute the commands in the command allocator, we close the command list and send the commands to the GPUs command queue.

The order is:

1. Put your commands in the command list.
2. Close the command list.
3. Send the commands to the GPU by calling ExecuteCommandLists() function.
4. Reset the command list to open it.
5. Go to step 1.

The commands in a command allocator are not executed by the GPU right away, so the CPU must wait until GPU has executed all the commands before putting new commands in the command allocator.

The way we make the CPU wait is through a fence. When the CPU is done putting all the commands in a command allocator, we put an extra command in the command queue called a fence. A fence is an instruction that the GPU needs to execute before the CPU can add more commands to a command allocator. This is called flushing the command queue when we make the CPU wait for the GPU to finish executing all the commands in a command allocator.

I made a function called flushCommandQueue() that increments a integer value and puts an instruction that tells the GPU to make a fence value the specified integer value. Until the GPU executes that command the CPU will wait.

We don’t want to make the CPU wait every single frame, so we make multiple command allocators and multiple resources that the GPU and CPU reference. If we made 3 command allocators, while the GPU is executing the commands in the nth allocator we can have CPU filling in the n+1 allocator with commands.

I implanted a circular array of 3 command allocators and 3 of every resource both the CPU and GPU reference. I also made a 4th command allocator used for initialization and resize commands. If the CPU is faster than the GPU it can get up to 2 frames ahead of the GPU.

**Initialization**

**Initialize Direct3D 12**

To initialize Direct3D 12, we need to:

1. Enable the debug layer.
2. Create a Direct 3D 12 device.
3. Create a DXGI Factory object.
4. Create a Fence object.
5. Query every descriptor size.
6. Create all our command objects.
7. Create a swap chain.
8. Create a render target view (descriptor) heap.
9. Create a depth/stencil view (descriptor) heap.

To initialize everything, I made a class called DeviceResources where the member variables were the COM interfaces needed to initialize Direct 3D 12. The header file DevicesResources was declared in also has all the necessary header files and linked all the necessary libraries to be able to initialize Direct3D 12.

**Scene Initialization**

To initialize a scene, I

1. Loaded in all my shader bytecodes for the scene.
2. Described all my shaders input elements (parameters).
3. Made a constant buffer view heap.
4. Described all the geometry in the scene.
5. Loaded all the vertices and indices into a vertex and index buffer respectively.
6. Created a constant buffer to hold each objects MVP matrix.
7. Defined each objects draw arguments.
8. Defined my camera properties.
9. Made all the root signatures I was going to use.
10. I made all of the pipeline state objects I was going to use.
11. I executed the initialization commands.
12. Flushed the command queue.

I created a class called RenderScene. It stored all the things needed to render a scene. It stored shader bytecodes, input element descriptions, rasterization state descriptions, pipeline state objects, root signatures, constant buffers, draw arguments, vertex buffers and index buffers. I used a hash map to pair a name with each thing, so I could access everything in constant time. The class also had member variables for a constant buffer descriptor heap, the constant buffer descriptor size and a descriptor range root parameter.

**Resize**

I made a resize function that:

1. Flushed the command queue.
2. Reset the command list.
3. Released all buffers that had a reference to the swap chain.
4. Created/Recreated the render target buffers and create views to each buffer.
5. Created/Recreated the depth stencil buffer and created a view to it.
6. Executed the resize commands.
7. Flushed the command queue.
8. Defined/Redefined the viewport and scissors rectangles.
9. Updated my camera’s aspect ratio.

I called the resize function after initializing Direct3D 12. I also reset the command list after doing the initial resizing.

I also called it every time the window was resized.

I declared and defined the resize function in the DeviceResources class.

**Update**

I made an update function that:

1. Checked if certain keys were pressed and if they were I moved the camera.
2. Updated the camera’s view and perspective projection matrices.
3. Updated the MVP matrix for each object and transposed each one because HLSL reads in matrices in column-major and my matrices are in row-major.
4. Went to the next frame in my circular array of frames.
5. Waited for the GPU if it hadn’t finished executing the commands for the current frame.
6. Stored the updated MVP matrices in a constant buffer.

**Draw**

I made a draw function that:

1. Reset the current command allocator.
2. Reset the command list.
3. Set the viewport.
4. Set the scissor rectangle.
5. Transitioned the current back buffer from present state to render target state.
6. Cleared the current back buffer.
7. Cleared the depth stencil buffer.
8. Specified which render target buffer to render to (which was the current back buffer).
9. Linked all the descriptor (view) heaps.
10. Drew all my objects.
11. Transitioned the current back buffer from render target state to present state.
12. Executed the draw commands.
13. Presented (swapped the front and back buffers).
14. Updated the current frames fence value.
15. Called signal to put a fence instruction into the command queue.

I implemented 3 functions in my RenderScene class to draw objects, beforeDraw(), afterDraw() and drawObjects(). The function beforeDraw() was called to do steps 1-9. The function drawObjects() was called for every object I wanted to draw. The function afterDraw() was called for steps 11-15.

**Game Loop**

In the game loop I:

1. Checked if there was a message for my window and if there was I translated it and sent it to my window procedure.
2. I got the time between each frame.
3. I called update function and passed it the time between each frame.
4. I called the draw function to draw all my objects.