Justin Neft

Independent Study Post-Mortem

Github link: <https://github.com/jj0629/independent-study>

**Goals**

The goals of this independent study were twofold: to learn a low-level graphics API (in this case, DirectX12) and to understand/implement newer engine optimizations and graphics techniques to learn more facets of graphics development and further my own research skills.

To this end, I had some specific graphical and engine goals I wanted to meet: creating an asset manager to create and store all engine assets at runtime, and dive deep into 3D textures and volumetric lighting as a post-process since I thought it was an interesting effect.

The asset manager was meant to help streamline my own work later down the line, as it would allow for me to create and easily access shaders, textures, and other objects that various rendering passes will need at runtime.

Volumetric lighting is an effect that always interested me as it related to mood lighting and atmosphere, as it is an interesting visual technique, and utilized a few concepts I found intriguing: compute shaders, 3D textures and ray marching.

**Problems**

Throughout this independent study, I encountered several problems that ended up slowing-down my progress, or otherwise limited what I was able to complete for the study. These included bugs that were difficult to debug, or my own misunderstandings of DirectX12 to needing to spend time reworking parts of my engine to properly implement DirectX12 requirements. These different issues all made me change my priorities and kept me from fully completing my goals for the study.

**Learning DirectX12**

DirectX12 (DX12) is a low-level graphics API, which comes with its own challenges and unique structures to create a rendering engine. One of those structures is that the programmer must explicitly tell the API what you’re doing at ay given time. As such, this means that the programmer must write a lot more code and manage many more objects to get similar results to doing similar activities in DirectX11 (DX11). This more intense nature means DX12 has a lot more setup time to get the basics of a rendering engine created. Once those foundations are set though, things speed up quite a bit.

Some of the concepts of an explicit rendering pipeline took a bit of time for me to understand, namely root signatures and pipeline state objects. These concepts all pertain to how the programmer sends information from the CPU to the GPU for work and took quite a bit of time for me to learn.

Root Signatures were probably the most difficult, as they pertain to a part of the API that DX11 does automatically, so I had no idea what I was doing in the beginning. The root signature (RS) is essentially one big object that describes ALL the resources being sent to the shaders in your rendering pipeline. That includes samplers, textures, RTVs and cbuffers among other elements. One of the bigger things I needed to learn about the RS is that it describes elements in EVERY STAGE of the rendering pipeline together. That means if you have multiple cbuffers across your vertex and pixel shaders, you’ll need to make sure they’re all bound to different registers, despite being in different shader stages.

The next object, pipeline state objects (PSOs), was a bit more straightforward. It essentially says what shaders you’re using, what data structs you’re using between shaders, and what data is being passed into the rendering pipeline. This ends up being a powerful way to keep track of what your renderer is doing at any point, and essentially makes sure you’re passing the proper information in.

**Debugging**

Early on, I learned that the visual studio 2019 graphics debugger…didn’t work for DX12. This made it extremely difficult to debug my program and see what was going wrong in the pipeline. As such, debugging was hell for the first few weeks until I discovered PIX.

PIX is an extremely powerful graphics debugging tool that allows the user to debug the graphics pipeline and analyze the different stages of it. It ended up being essential to me for fixing issues and causing the sources of bugs. As such, I highly recommend anyone who wants to use DX12 also learn this program with it.

**DirectX12Helper**

This class is a singleton that holds a lot of miscellaneous helper methods to streamline my code for DirectX12. This includes things like adding fences to the command list for cpu/gpu synchronization, creating textures or RTVs and adding them to their respective heaps, or creating cbuffers of arbitrary data and copying them over to the GPU.

This class needed a lot of upgrades throughout the project, as I always needed new helper methods to create new objects or resources for the GPU to use. This could range anywhere from making RTVs, loading texture cubes in or even creating different types of cbuffers on the descriptor heap for my shaders to use. Thankfully these methods were extremely useful once created and ended up streamlining a lot of my code later down the line.

**The Asset Manager**

At first, the asset manager was meant to be a singleton object that any class in the code could access to pull resources such as textures and shaders to modify/reference them quickly and easily. The main benefit here is that, as the engine begins requiring more and more textures and shaders for more effects and game objects at run time, having a singleton to pull resources from heavily reduces the amount of code that needs to be written, and helps overall clarity of the project’s code.

This benefit is especially emphasized in DirectX12, where I needed to create many new types of assets for use throughout the engine. By keeping all this asset creation code contained to a single class. However, this quickly led to a different issue: since DirectX12 requires so many different types of resources and objects to make the rendering pipeline function, I needed to create reusable code that could make these objects (Root signatures, pipeline state objects, shader blobs, render target views, or even textures) through files that would contain this information. This helped me organize my project, as instead of having several hundred lines of code to crawl through for creating every single root signature, I could instead have a folder that contained json files with the relevant names and information within.

To make and read these json files, I needed to learn how to do file I/O in C++, a task I had never handled before. While I knew the principals of file I/O in C#, C++ ended up being a whole new beast with its own requirements. This involved learned a new library for json reading, rapidJson, to properly interpret the files I was writing for asset creation. I also needed to learn new data types like w\_strings so I could read and open all the file types needed to get my asset manager properly functioning.

Once I had all the file I/O completed and setup, the asset manager was properly functioning and extremely helpful, as it streamlined my workflow for creating new assets and including them in my project. This made it much easier to make new pipeline state objects and root signatures, which were necessary for when I wanted to implement new effects or object types for my engine.

**The Renderer**

The renderer class is meant to do a lot of the heavy lifting of the application, as that’s where most of the draw commands are sent to the command list, and where data typically gets passed-into the GPU. As such, the renderer needs access to a lot of the programs data and has several large methods that represent each rendering pass and effect that happens in the engine.

The renderer is also responsible for the program’s GUI, using the DearImGUI library as it is fast, easy to use, and extremely robust. The GUI itself requires a lot of data from the renderer, so it made sense to have it be within the renderer’s scope to help make updating data and information easy and efficient.

**Render Target Views (RTVs)**

When creating the renderer, a large part of that was making RTVs so that I could send the results of my renders to textures instead of directly to the screen. By doing this, it was possible to create post-process effects using the data those textures held. As such, I needed to learn how DX12 created and managed RTVs and do it myself.

In DX12, RTVs require their own memory heap on the GPU, that you must declare. In addition, you need to keep track of where in that heap you are, and make sure you are placing new RTVs in new spots within the heap. Beyond that, I needed to make sure all the proper pieces of my RTVs stayed together. To that end I made a small struct that held an RTV, it’s corresponding texture and some information about the RTV (like if it was screen sized or not). I called these RtvBundles and used them to keep references together and not let specific RTV textures get lost among my other textures for other uses.

By keeping track of what RTVs where screen sized, it made resizing them a lot easier. I just needed to go through my internal dictionary of RTVs and delete all the screen sized ones before recreating them with the new window size. However, this implementation was not efficient, as my engine had no way of tracking what previous heap slots opened through this process, meaning it would leave the heap with a lot of empty spots the engine wasn’t using. As such, I needed to make my RTV heap much larger to accommodate the inefficient use of resizing RTVs.

**Alternative Plans**

As I put more and more time into this project, it became clear quite quickly that I would need to shift priorities. Instead of focusing on actual graphics techniques like I wanted to, I instead needed to work on engine foundations (like the asset manager and dx12 helper), as well as research the inner workings of DX12 to get the basics of a functioning engine for doing more advanced techniques. These foundations and research ended up being significantly slower than I wanted to complete, and as such I didn’t have the time, I wanted for digging into volumetric lighting and other advanced post process effects.

**Known Issues**

This is just a small list of known issues in the repo that I want to go over here.

* When creating extra textures for IBL lighting, the cube maps aren’t created properly. This has the added effect of breaking later PBR lighting, so it has been disabled for now.
* When resizing RTVs, a lot of empty space is left in the RTV heap that isn’t being used. This can very quickly lead to an overflow in the RTV heap, which can result in a crash.

**Results**

Overall, I learned a lot not just about DX12, but how graphics APIs work in general from this study. I got the opportunity to learn how to manage, organize and expand my own code base as well as gained new skills in researching and problem solving from having to complete these tasks and resolve these issues on my own.

Beyond just that, seeing the lower-level graphics API also reinforced what I learned in DX11, and helped me to understand the underlying mechanics of that API. Overall, I feel much stronger in graphics programming after this independent study, and I plan on continuing my work with this into the future.