Cameron Schneider & Ben Hubner

Prof. Dan Buckstein

GPR-400 Adv. Rendering

12 February 2021

Project Proposal

What do we want to achieve?

We would like to create a rendering engine as an extension of Animal3D, at first using the Vulkan API. This interests us because neither of us has worked with Vulkan before and it is used by many professional studios, so understanding how it differs from the OpenGL API that we already have learned about is useful. From our initial research, we also know that Vulkan exposes a lot of powerful computational tools that OpenGL does not, which means we have the ability to use graphics hardware in ways other than rendering, and even use GPU multithreading to do so.

A topic that Ben considered before this was implementing ray tracing in Animal3D. He decided that this topic was a better choice because he wanted to learn more about how a rendering engine actually works at a low level.

Originally, Cameron was interested in either replacing or adding on to Unreal 4's 3D rendering module, as he would like to see more flexibility in its implementation and pipeline stages. There are also a few bugs in their post processing pipeline that he wanted to investigate. While this would be a bigger contribution to the industry as a whole, he really wanted to use this course as an entry point for low-level API work and general framework design, since he hasn't done it before. Additionally, creating a custom framework would let us use graphics APIs (like Vulkan) in ways other than rendering.

In the end, we are aiming to create a rendering engine that can display objects in real time, using the Vulkan API and HLSL/GLSL shaders. We would like to abstract various data structures in a way that would allow various APIs like Vulkan to utilize data in ways not necessarily applying to rendering, namely computational tasks, which means we are also targeting a basic interface for Animal3D to use Vulkan compute shaders.

Why is this relevant? What research must be done?

Innovation number one is our data abstraction. If we want to have the GPU perform physics calculations on a series of optimized Jacobian matrices (just an example) in LOD-style chunks, we need to have some method of translating a matrix as large as that into a data type that the API/hardware can deal with efficiently. Ideally, we should be able to utilize GPU hardware to perform batch processing for simulation, essentially just performing GPU-accelerated tasks. Additionally, Vulkan provides access to GPU multithreading, which we have never dealt with before. Can we create an easy-to-use interface to create and synchronize GPU threads? This should allow us to run computation tasks alongside the main rendering thread easily. All of the above tasks are becoming more and more useful in the larger industry as GPU hardware advances. Furthermore, Vulkan provides access to all graphics devices on a machine. We can use this to set up a dedicated card as a rendering card, and use a secondary (or integrated) card for additional computational tasks unrelated to rendering.

We need to figure out how various APIs treat common data types associated with rendering, such as vertices, textures, and uniform variables, and then figure out ways to abstract those that will allow developers to directly control what a “texture” even means. Is it a representation of transforms? Is it an image? Is it the result of physics calculations? Additionally, we need to research optimization of said data types and any operation that can be done on them. Since our initial target API is Vulkan, we will also need to research how multithreading works on a GPU, since it is different from CPU threading, as well as how to synchronize threads on a clock cycle using semaphores.

While graphics hardware and software have been used extensively in AI, we haven't seen much on how it can be used in physics or simulation. Nvidia's PhysX engine is one of the few commercial products that is able to utilize GPU acceleration to assist in physics simulations, but the vast majority of physics engines still run on the CPU. Why is that? Why not allow the GPU to do batch calculations based on object relevancy? If we are able to utilize the tools provided by Vulkan to do so, we think that could lead us towards higher-fidelity physics than standard CPU-only solutions, however, we need to pay attention to performance impacts on rendering, as well as resource management and device hardware restrictions.

As previously mentioned, neither of us has ever built a rendering engine, and neither of us has dealt with Vulkan, so pretty much all of the development we do will be brand new to us from a technical perspective. Since we're also planning to create these data abstractions and manipulation methods, we will also be able to work on our abilities to research low-level technical capabilities of platforms and hardware, as well as design high-level concepts around that research. Another common thing that will advance our professional development is that we will be creating an interface between Animal3D's pseudo-object-oriented structure, and Vulkan's data-oriented structure. It's common to create these sorts of adapter interfaces that allow new technologies to be compatible with existing software, and so this will be good practice.

How will we get there?

Since our framework is planned to target multiple APIs, but is starting with the most compatible one and will be built on C++, we will be able to work on multiple platforms in terms of OS and hardware. So far, we know that we will need mid to high-end GPU hardware in order to run Vulkan reliably, which should include any x64 CPU, and any GPU made within the last 4-5 years.

In order to just get Animal prepped for a new rendering engine, we will need to extensively research and discuss the implementation of Vulkan, as it is wildly different from the existing OpenGL implementation in Animal. We will also need to discuss Animal's basic structure, as it currently only supports a single rendering API and corresponding wrapper. If we want to be able to switch between OpenGl and Vulkan, the framework itself needs to be able to handle that swap cleanly. Next, we will need to actually create a new project and link it inside Animal, alongside hooking that project into Animal's callbacks. From there, we should be able to start basic work on the actual Vulkan implementation at the same time as we work on interfacing Animal's graphics data types with Vulkan's, which will be a considerable amount of work since Animal doesn't have existing support for the majority of Vulkan's informational structures. This way, we'll be able to test our interface with actual rendering code.