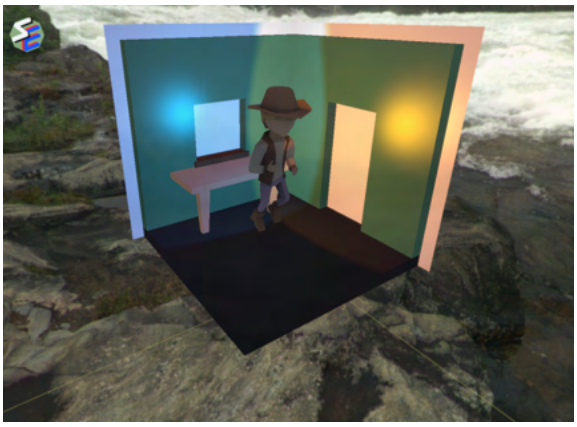
Vulkan 3D Renderer - Technical Takeaways

Charlie Sands - September, 2022 - Northville, Michigan

Overview

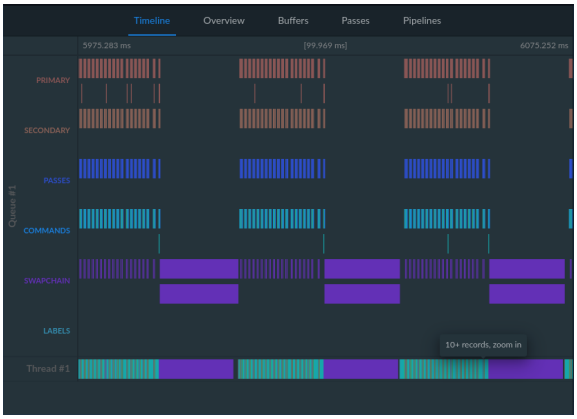


An animated cowboy running rendered in real time

I started my Vulkan 3D renderer during the pandemic in order to pass the time, improve my programming practices and learn more about high performance data processing. The engine is written in C++ and uses the C Vulkan bindings to interact with the API. My goal was to create an engine that has enough features to support the development of an A or AA game. Currently my engine supports rendering multiple objects, lighting with diffuse and point lights, on screen GUIs, multiple different shaders, particle systems, animated objects, cube-map rendering, audio, asset loading with ASSIMP and object transparency among other things. I do not use other libraries for memory management, I manage all V-RAM allocations and deallocations myself. Some issues still exist within the engine. Unfortunately, when school started I needed to begin work on my WiFi tracking system because it was a class project and I ran out of time to complete my renderer. It is about 95% complete, I still hope to finish it entirely one day. Many of my renderer's features were inspired by ThinMatrix's OpenGL 3D renderer, but implemented in Vulkan, not OpenGL and with C++, not Java. Many of his assets were used for testing in my renderer.

Programming Best Practices

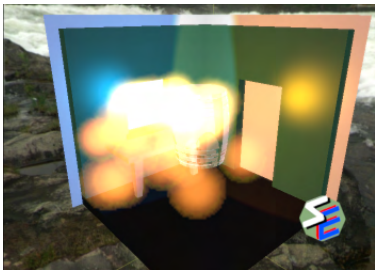
One of my main goals in creating my renderer was to improve how I wrote C++, and by extension C, code. I chose a rigid styling pattern enforced by clang-format. I used namespaces in order to segment my code from other libraries and code readability. I used tools such as Valgrind and RenderDoc often to profile CPU and GPU memory usage and ensure there were no growing memory leaks or performance bottlenecks in my code. I avoided the use of subclasses in my code in order to improve understandability and reduce time spent refactoring. I carefully planned my progress in Markdown files in order to keep the project on-track and consistently moving forward. I developed many time management and planning skills I have used in my other projects. Additionally, I learned about the effective use of documentation and abandoned my reliance on tutorials



A profiling output from RenderDoc used to improve my rederer's performance

Memory Management and API Interaction

I wanted to work with lower level hardware systems with this project. One aspect of this is memory management systems. Vulkan provides near direct access to the GPU driver which means any memory that it allocates must be managed by the program. Additionally, much like in C, all the memory must be manually deallocated when it is no longer needed. In order to ensure that no memory is gradually leaked from my application over time I created a global tracking system for all GPU memory that is able to effectively track and release all used VRAM at the end of execution. I created a similar system for more general access to the API. Each call to the API requires lots of, oftentimes redundant, information. I created an object tracking system to effectively manage this data through different classes, making creation of Vulkan API objects easier.



A flaming transparent barrel displayed by my renderer

Approximate Cost	$0
Approximate time spent	4 months
Skills developed	- Interaction with High Performance APIs - Userspace Linux Development - Code Reliability - Inverse Kinematics and Animation