DEVELOPMENT OF VULKAN BASE CODE FOR INTRODUCTION TO COMPUTER GRAPHICS

BY BOB LOTH

A SENIOR PROJECT SUBMITTED

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

Computer Science Department

California Polytechnic State University

San Luis Obispo, CA

MAY 2022

ABSTRACT:

short (1 paragraph) description of your project and your results

This project uses the Vulkan API to implement a double-buffered forward renderer of indexed three-dimensional geometry, with an interface over many lower-level details, intended for use as starter code for Introduction to Computer Graphics. The current starter code loads a number of different files in the Wavefront OBJ and the Khronos Group glTF/glb file formats, creates and modifies uniform data globally and for each loaded primitive shape to manage rendering an example scene showcasing the project's capabilities. A debug shader and a debug pipeline are included: pressing keys detailed in Figure 5: Key Bindings will render each object in the scene in the same style, exposing primitive data such as normals, texture coordinates, and wireframe geometry.

PROJECT INTRODUCTION AND MOTIVATION

PROBLEMS POSED BY OPENGL API USAGE IN INTRODUCTION TO COMPUTER GRAPHICS

Current Introduction to Computer Graphics curriculum includes the usage of base code developed using the OpenGL API, a cross-platform graphics API first released in 1992, also referred to as simply OpenGL. Today, the OpenGL API is managed by the Khronos Group. The Khronos Group consists of an association of technology groups and companies that develop, publish, and maintain standards and APIs such as OpenGL, Vulkan, and glTF. However, support and updates for OpenGL were deprecated on Apple's macOS devices with the release of macOS Mojave 10.14 in September 2018. Instead, Apple recommends that existing graphics applications using OpenGL instead use the Metal API. At time of writing, the latest supported version of OpenGL available on macOS devices is 4.1, an API version that was released on July 26th, 2010. This version and its capabilities are unlikely to change until Apple decides to remove support for OpenGL entirely. This version is more than 7 years older than the newest API version, 4.6, released on July 31st, 2017, and almost 12 years old at the time of writing. Since OpenGL version 4.1 was released 12 years ago, GPU hardware and features have changed significantly, and newer OpenGL versions support features such as Transform feedback (v4.2), Compute shaders (v4.3), asynchronous queries (v4.4), and precompiled SPIR-V shaders (v4.6), as well as many other such features.

Continued usage of OpenGL version 4.1 presents a few potential problematic situations. The potential situation with the most severe consequences is a potential removal of any and all support for OpenGL on macOS devices. This would prevent a sizable portion of Cal Poly's computer science students from using a personal computer with an operating system of their choice to directly run and modify the existing OpenGL base code. Another potential situation is that curriculum changes or a new class about computer graphics incorporates topics that include feature usage such as compute shaders, from OpenGL versions greater than macOS's last supported version of 4.1. Finally, in group projects consisting of students using a mix of operating systems, there is a high potential of students introducing code changes using newer API features that are not cross-platform.

INTRODUCTION TO VULKAN

The Vulkan API, or Vulkan, is a modern, cross-platform graphics and compute API, maintained by the same Khronos Group that maintains the OpenGL API.

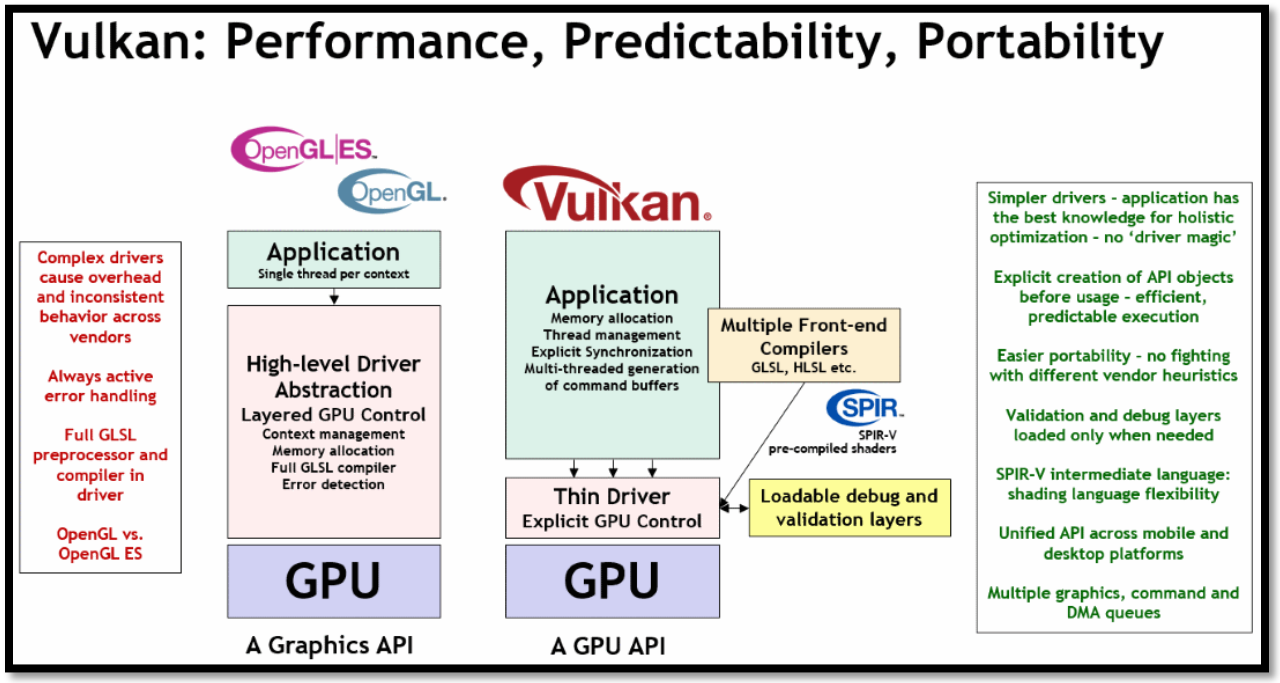

Figure 1: API Comparison of Vulkan to OpenGL

Vulkan is considered to be a lower-level API as compared to OpenGL and its more contemporary counterparts of Metal and Direct3D 12, as it is designed to run with minimal driver overhead. This decision moves the responsibility of many tasks once left up to the driver implementation, such as command batching and submission, memory allocation, and host-device synchronization, squarely onto the application developer. In addition, Vulkan is explicitly designed to support distribution of work across multiple CPU cores, as well as across multiple GPUs. It also takes full advantage of the parallel processing capabilities of modern GPUs, by allowing work to be submitted simultaneously from multiple CPU threads to multiple device queues and processed asynchronously. Multiple synchronization objects, which can coordinate memory access and command execution, are also included, which assists host to device, and device-only communication and throughput for these highly parallel workloads. Unlike OpenGL, there is no global state and no global context. State is managed through handles to opaque objects, many of which are thread-safe, but some must be created on a per-thread basis to avoid conflicts when using multiple threads. In summary, the Vulkan API provides an application programmer with tools that are highly configurable to the specific needs of the application, using lower-level constructs that more directly map to physical hardware.

The combination of performance capabilities, cross-platform support, and more modern features of this API as compared to OpenGL and other platform-specific graphics APIs makes Vulkan ideal for this project's motivation. However, the capabilities of this API, and its "minimal driver overhead" model, means that many operations typical performed by the driver must be instead performed by the application programmer. Vulkan is also an "explicit" API, meaning that many details concerning the programmer's intended usage of the API, such as the amount and type of memory to access, any pipeline and render loop stage dependencies, and many other facets of rendering setup, execution, and teardown must be specified explicitly, without relying on any automatic or default behavior.

With regards to macOS compatibility issues specifically, macOS does not have a native Vulkan driver. Instead, code using the Vulkan API can run on top of Metal with MoltenVK, a translation library that maps Vulkan commands to Metal commands. This does not depend on Apple's decision to support or not support a native Vulkan implementation. As of writing, Vulkan API features up to Vulkan version 1.1 are supported through MoltenVK. The latest Vulkan API version, 1.3, was released on January 25th, 2022.

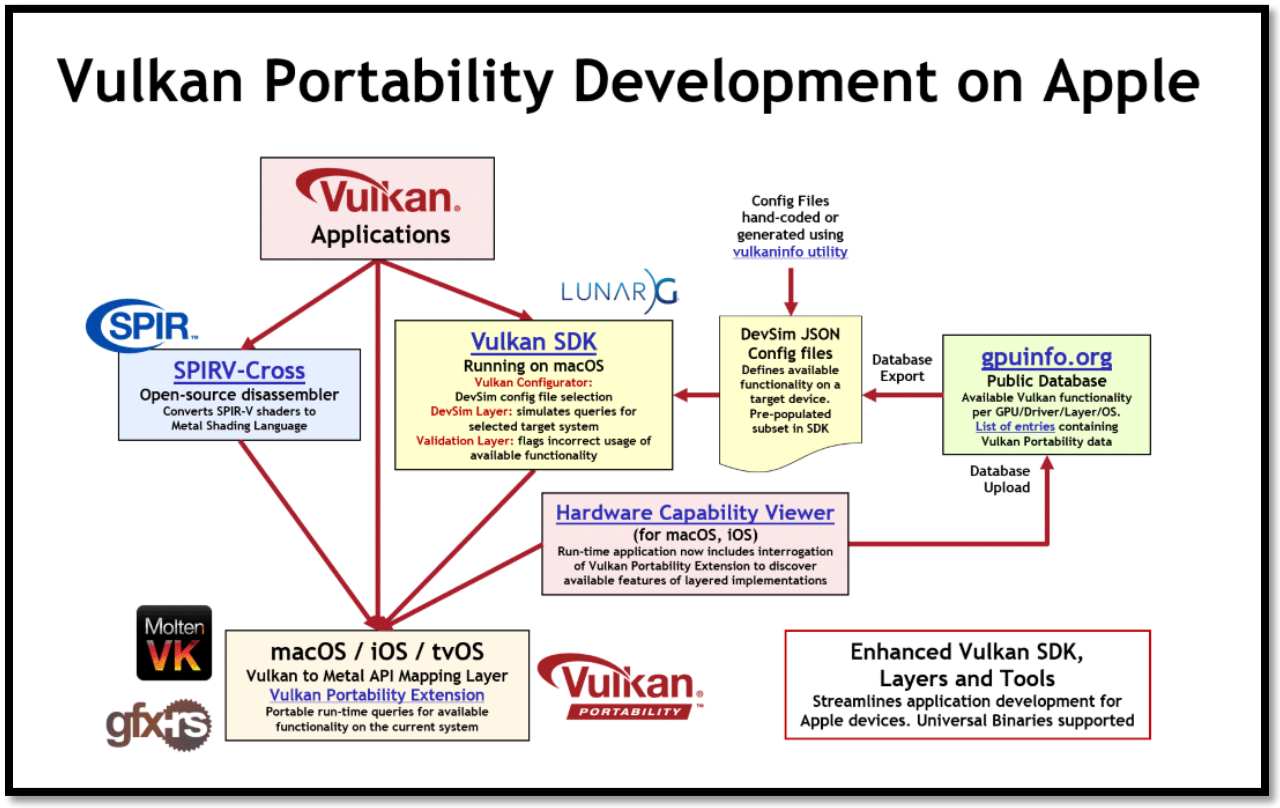

Figure 2: Portability Layers on Apple

The full breadth and depth of creating and running a Vulkan application is not something we want to introduce to students in an introduction to graphics course. Instead, learning the fundamentals of 3D graphics in an API-agnostic way is desired. To this end, base code was developed in order to abstract away some of the lower level Vulkan set up, and to facilitate students completing the learning objectives of the introduction to graphics course.

INTRODUCTION:

SUMMARY

In order to create a Vulkan graphics application that implements real-time double-buffered forward rendering of indexed three-dimensional geometry, you must:

- Create an instance
- Select a physical device
- Create a logical device
- Create a swapchain
- Create a surface
- Allocate vertex, index, and uniform buffers
- Allocate a depth buffer
- Create a descriptor set layout and a pipeline layout
- Create a descriptor pool, and allocate descriptor sets from it
- Create shader modules
- Create a render pass
- Create frame buffers
- Create a pipeline
- Create a command pool
- Record command buffers
- Create synchronization objects

Then you must manage your queue submissions, surface presentations, and descriptor set updates using synchronization objects in the render loop. Finally, you must destroy all created objects and allocated memory in the setup phase, often in the reverse order of their creation.

The following figures are diagrams of the Vulkan API objects and their relation to one another, that may be used as a visual aid when reading the following details section. It might also be helpful to first get acquainted with the various API objects before reading about their usage, by reading the glossary section at the end of this report.

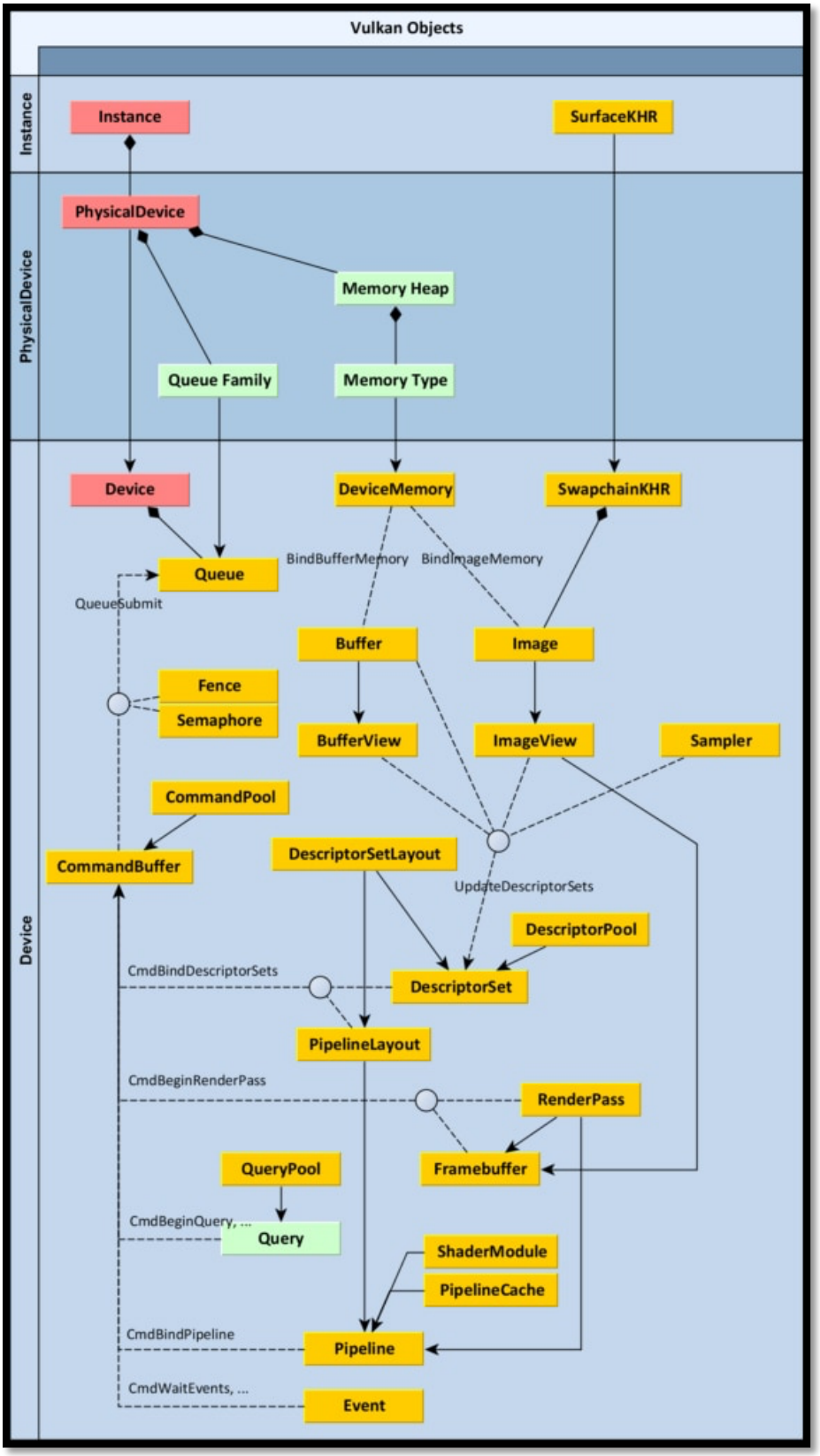

Figure 3: Vulkan Objects

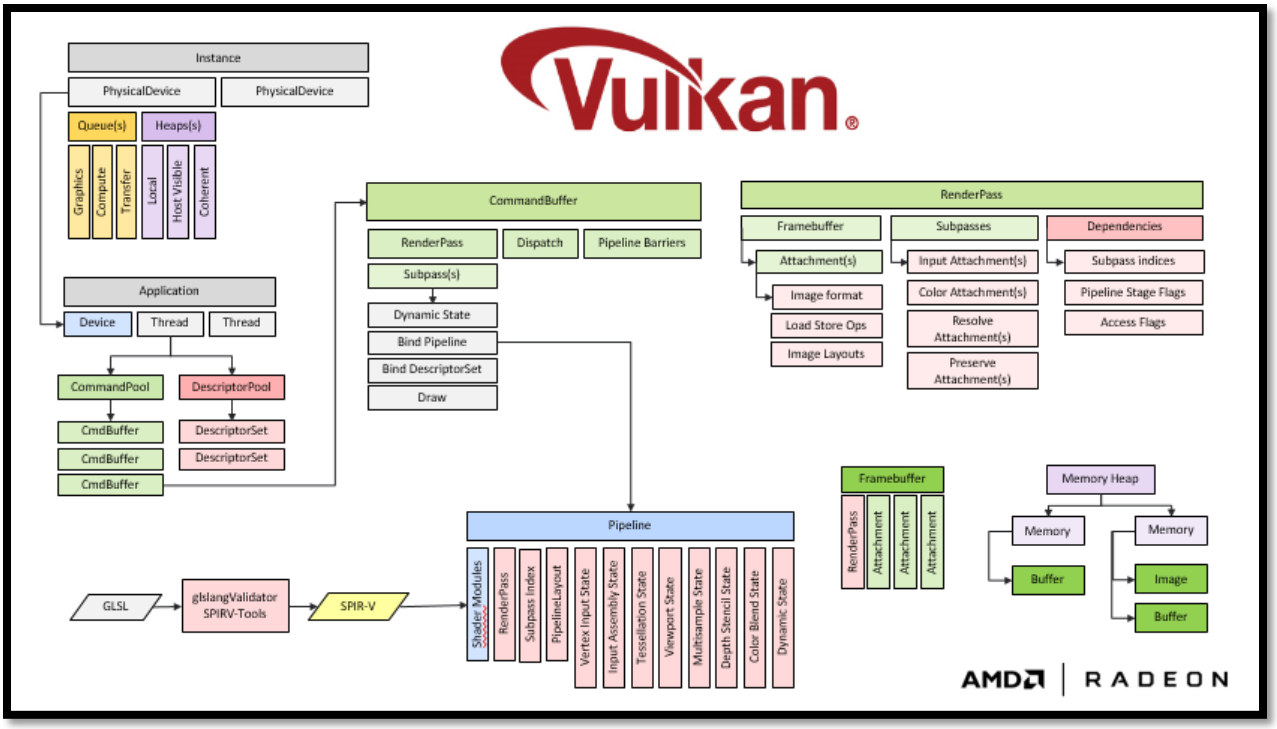

Figure 4: Vulkan Objects, alternate

DETAILS

The Vulkan API naming convention typically prefixes functions with `vk`, type names with `Vk`, and preprocessor defines and enumerated types with `VK_`. Creation of Vulkan objects typically involves constructing and initializing a creation struct, prefixed with `Vk` and suffixed with `CreateInfo`, as well as an uninitialized instance of the final object being created, and then passing the creation struct and the uninitialized instance into a `vkCreate`-prefixed function to retrieve a fully initialized object, as an opaque handle, upon success. An index for referencing any specific Vulkan object types or functions is included in the appendices section.

A typical forward renderer, which takes in model data and uniforms and uses a single pipeline with a vertex shader and a fragment shader, can be created in the following steps.

INSTANCE CREATION

Creating a Vulkan instance is the entry point of any application using the Vulkan API. `vkCreateInstance` is called, supplied with a properly initialized `VkInstanceCreateInfo` struct pointer, and a handle where the instance information will be stored upon successful invocation. `VkInstanceCreateInfo` will contain information about the requested and required layers and extensions, as well as a `VkApplicationInfo` structure that contains numerous fields, the most important being the Vulkan API version requested. For rendering to the screen, the `VK_KHR_surface` instance extension is required. Successfully creating an instance initializes the loader, which communicates with device drivers through any layers the instance has specified.

PHYSICAL DEVICE SELECTION

Once an instance has been initialized, the instance object can now be queried for information regarding the number of physical devices on the system, and their properties. This information can be queried by calling `vkEnumeratePhysicalDevices`, which will return a vector of `VkPhysicalDevice` handles for each physical device on the system. Each handle can be queried to retrieve a `VkPhysicalDeviceProperties` struct, that contains information about the device, including supported features, available extensions, device limitations vendor ID, and the API and driver versions. For presenting results to the screen, the `VK_KHR_swapchain` device extension is required. For most applications, a single physical device is all that is required, although a single instance can use multiple physical devices.

LOGICAL DEVICE CREATION

Once a physical device has been selected based on the needs of the application, a logical device must be created from that physical device, using `vkCreateDevice`. Vulkan logical device creation requires a `VkDeviceCreateInfo` struct that specifies the extensions enabled, and additional information about the number and type(s) of queues that will be used, contained in a `VkDeviceQueueCreateInfo` struct.

SWAPCHAIN AND SURFACE CREATION

For presenting images to the screen as they are rendered, a `VkSurface` object and a `VkSwapchain` object must be created. Surface objects are often operating system and platform-specific, but the creation functions are usually `vkCreate`-prefixed and `SurfaceKHR`-suffixed. A swapchain is created by calling `vkCreateSwapchainKHR`. A surface is an abstraction of a window to draw to, and a swapchain is an abstraction of a list of images that, for the purposes of a double-buffered window, are either render targets or currently displayed to the window. Retrieving target image handles is achieved by calling `vkGetSwapchainImagesKHR`, after which you can wrap the images in `VkImageViews` that describe the layers, format, and usage of the images.

VERTEX BUFFER CREATION

Vertex buffers are created with `vkCreateBuffer`, this time with a usage flag of `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`. Get the buffer memory requirements using `vkGetBufferMemoryRequirements` for the size of data to load, allocate memory using `vkAllocateMemory`, map the memory using `vkMapMemory`, and copy the data over. Vertex data is usually not written to every frame, so it may be beneficial to use a staging buffer approach, as described in the Buffers section of the glossary.

INDEX BUFFER CREATION

Index buffer creation is nearly identical to vertex buffer creation, although the appropriate usage flag is now `VK_BUFFER_USAGE_INDEX_BUFFER_BIT`.

UNIFORM BUFFER CREATION

In Vulkan, uniform buffers, or data available to all shader stages that remains constant for a single render pass, must be specifically allocated and filled in. Call `vkCreateBuffer` with an appropriately formed `vkBufferCreateInfo` struct that includes a usage flag of `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` and the size of the data, and begin the process of allocating memory for it by first calling `vkGetBufferMemoryRequirements`. As uniform data is often updated per frame, per material, or even per object, we will want this memory to reside in a device memory location that is easily accessible by the host and does not require explicit flushing. Request this type of memory allocation by using the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` flags. Finally, map the memory using `vkMapMemory` on the allocated `VkDeviceMemory`, which returns a pointer to a void pointer. This void pointer is the memory location to copy data to, using a standard library function such as memcpy. Unmapping memory is not required and can be more computationally expensive than leaving it mapped. However, if this is desired, calling `vkUnmapMemory` on the same `VkDeviceMemory` handle will unmap the memory.

DEPTH BUFFER CREATION

For 3D rendering that includes depth resolution, a depth buffer will need to be allocated, as Vulkan does not initialize one by default. First a depth image must be created, by calling `vkCreateImage`, with an appropriately formed `VkImageCreateInfo` struct. A depth buffer's device memory requirements must be obtained by calling `vkGetImageMemoryRequirements` with the `VkImage` handle passed in as a parameter. These returned memory requirements can be used to allocate and bind device memory for the depth buffer, using `vkAllocateMemory` and `vkBindImageMemory`, respectively.

DESCRIPTOR SET LAYOUT AND PIPELINE LAYOUT CREATION

Creation of a descriptor set, that will contain pointers to resources used by shader programs, will require specifying the layout of the descriptor set, using one or more `VkDescriptorSetLayoutBindings`. For a simple graphics application, one descriptor set layout, with a few descriptor set layout bindings should be appropriate. Supplying these layout bindings to a `VkDescriptorSetLayoutCreateInfo` struct, and then supplying that struct to `vkCreateDescriptorSetLayout` should return a handle to a created descriptor set layout.

A pipeline layout will contain a list of descriptor set layouts. Pass the number of descriptor set layouts created and their data to a `VkPipelineLayoutCreateInfo` struct, and call `vkCreatePipelineLayout`, passing the struct in as a parameter.

DESCRIPTOR POOL CREATION AND DESCRIPTOR SET ALLOCATION

Descriptors are allocated from a descriptor pool. Initialize a descriptor pool by calling `vkCreateDescriptorPool`, passing in a `VkDescriptorPoolCreateInfo` structure with `maxSets` indicating the maximum number of descriptor sets that can be allocated from the pool, and `pPoolSizes` as a pointer to an array of `VkDescriptorPoolSize` structs, each containing a descriptor type and number of descriptors of that type to be allocated in the pool. Allocate descriptor sets from this pool by calling `vkAllocateDescriptorSets`, using a descriptor set layout passed to `vkCreatePipelineLayout`.

DESCRIPTOR SET UPDATES

Allocation and updating of uniform buffers take place by mapping the memory and copying to it. In order for a shader to use this data, it must be written to a descriptor set. Start by creating a `VkWriteDescriptorSet` struct, which specifies the set and binding location to update, the buffer(s) to update the set and binding with, and the number of descriptors to update. Then call `vkUpdateDescriptorSets`, passing in the `VkWriteDescriptorSet` as a parameter. This will update the contents of the descriptor set.

SHADER MODULES

SPIR-V is the pre-compiled shader code representation for Vulkan. Shader modules are created by calling `vkCreateShaderModule`, passing in a create info struct pointing to the precompiled SPIR-V file. Shader modules are bound to specific shader stages in a pipeline layout.

RENDER PASS

Initialize `VkAttachmentDescriptions` for the color buffer and the depth buffer, and specify any format and image layout transitions that need to occur for each attachment. Specify a `VkSubpassDescription`, that defines a pipeline bind point, and include the color attachment and depth attachment descriptions. For a typical double-buffered rendering scheme, we will want to only write to the color and depth attachments if the image has been made available by the swapchain. Do so by specifying a subpass dependency: create a `VkSubpassDependency` struct, and specify the source and destination stage mask as `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`. Finally, create a render pass object using `vkCreateRenderPass`, passing in the attachment descriptions, the subpass, and subpass dependency. This object will be used in the render loop.

FRAME BUFFERS

Create a framebuffer for each swapchain image by calling `vkCreateFramebuffer`, supplying the related create info struct with a render pass handle, a pointer to an array of `VkImageView` handles, one for each framebuffer attachment, and the dimensions of the framebuffer.

PIPELINE CREATION

A simple graphics pipeline consists of a vertex shader stage, a fragment shader stage, a single pipeline layout with a single descriptor set layout, a render pass with a single subpass, state information defined in pipeline state objects, and the vertex and uniform buffers used in the pipeline. Configure the various pipeline state objects by creating the various `VkPipeline`-prefixed, `StateCreateInfo`-suffixed structs, initializing them, and finally calling `vkCreateGraphicsPipelines`, which will finally return a pipeline handle, to be bound and used in the render loop.

COMMAND POOL CREATION

Once a logical device has been created, we need a way to submit work to the device. This is done by creating a command pool and allocating command buffers from the pool. A command pool can only be associated with one queue family, so one pool per queue family used is required. For a simple application, and a device with a queue family that includes both graphics and transfer queues, one command pool is sufficient. Command pool creation is done through the `vkCreateCommandPool` function, which takes in a `VkCommandPoolCreateInfo` struct that contains the index of the queue family it is associated with.

COMMAND BUFFER RECORDING

Command buffers can contain multiple commands, which are placed into the buffer by calling `vkBeginCommandBuffer`, and then calling one or more `vkCmd`-prefixed functions, such as `vkCmdDrawIndexed`, for drawing indexed geometry, and then finally calling `vkEndCommandBuffer`. Finally, command buffers are sent to the GPU for asynchronous processing by calling `vkQueueSubmit`. Commands in a particular command buffer will be executed in the order they were recorded, and submissions to a single queue will typically be processed in the order they were submitted, but synchronization objects are often required to guarantee execution order across multiple command buffers and queues, as well as to synchronize work that is not contained within a command buffer. For a simple application, we will allocate a command buffer for each swapchain image, using `vkAllocateCommandBuffers`. Each command buffer will contain identical commands. We will begin a render pass using `vkCmdBeginRenderPass` and do some setup before drawing. We will record binding our created pipeline using `vkCmdBindPipeline`, binding our vertex and index buffers using `vkCmdBindVertexBuffers` and `vkCmdBindIndexBuffer`, respectively, and binding our descriptor sets using `vkCmdBindDescriptorSets`. Then finally, we will record our draw operation, `vkCmdDrawIndexed`. We will record ending our render pass using `vkCmdEndRenderPass`, and finally, end command buffer recording by using `vkEndCommandBuffer`. Now our command buffers, one for each swapchain image, contain a sequence of instructions that renders our scene.

RENDER SYNCHRONIZATION

Now that initialization is complete, we can finally begin our render loop. We can begin sending command buffers to the GPU, in order to draw a scene to the screen. However, because the Vulkan API was designed with both CPU and GPU parallelism in mind, there is no default synchronization scheme, and we must make our own.

The following paragraphs will describe a common usage of swapchain image synchronization used in a buffered display of rendered frames to the screen. We will be using two types of synchronization objects. Fences, to ensure that descriptor set data is updated on the GPU every frame before the next frame begins drawing again, and semaphores, to ensure that swapchain images are not being written to as they are being presented on the screen, and to ensure that a render pass has completed before presenting the image to the screen.

Firstly, we will use one fence per swapchain image, to ensure that any uniform data in our descriptor sets can be fully updated each frame. Call `vkWaitForFences`, specifying the fence to wait on. Fences are signaled as part of a queue submission by being supplied as a parameter to a `vkQueueSubmit` operation. In this case, we supply a fence as a parameter to the queue submit that contains one of our command buffers, the sequence of instructions that we specified to draw a single image. Once the command buffer in the submit operation has completed execution, the fence is signaled, `vkWaitForFences` unblocks, and operations after it can proceed. `vkResetFences` is called to unsignal or “reset” fences, indicating that operations will once again block on a `vkWaitForFences` call. In terms of actual usage in the render loop, the first function call in the loop is `vkWaitForFences`. Eventually, `vkResetFences` is called, to stop rendering the next frame. Directly after this, we make any changes to our uniform data on the CPU, map its corresponding memory, and push it to the GPU using memcpy. With the updated uniform data on the GPU, we then call `vkQueueSubmit` with our recorded command buffer, to draw using the updated data. Once that draw operation completes, the fence is signaled, and the next frame’s drawing and updating operations can begin.

Secondly, we will be using two semaphores for each swapchain image. One semaphore in the pair, our ”render complete” semaphore, will be used to signal on completion of our drawing command buffer, and the other, our “image available as render target” to signal on swapchain image availability to use as a render target. In terms of actual placement in the render loop, our “render complete” semaphore will be included in a `VkSubmitInfo` struct included in our `vkQueueSubmit` for draw operations, the same queue submit that contains our fence. This means that once that command buffer submitted to the queue has been fully executed on the GPU, the “render complete” semaphore signals, and a following `vkQueuePresentKHR`, which presents swapchain images to the screen, unblocks and presents a fully rendered image. The second, our “image available as render target” semaphore, is included in a call to `vkAcquireNextImageKHR`, which retrieves the index of the next available presentable image. It is also included in our `VkSubmitInfo` struct, instructing the color attachment output stage of our pipeline to not begin execution until the semaphore is signaled, meaning the image is available as a render target. The granularity of this semaphore means that almost all of the GPU-side render loop can continue, up to the point that it needs to write data into the color attachment.

To sum up our basic synchronization, we send prerecorded batches of command buffers to the GPU and swapchain provider as fast as possible, instruct them to do as much as possible before running into dependency issues, and if any dependency issues arise, such as swapchain image availability, rendering, or uniform updating, we complete that work and signal to the GPU that work can continue.

RENDER LOOP

Now that we have covered program initialization, and described our synchronization goals and methods, we can now describe the render loop.

- Wait on fence signal.
- Get the next available swapchain image.
- Reset fence to block before updating uniform data and drawing.
- Update uniform data.
- Submit our prerecorded “draw” command buffer to a graphics queue, which:
    - Begins a render pass.
    - Binds our pipeline.
    - For each object in the scene:
        - Binds its vertex buffer.
        - Binds its index buffer.
        - Binds its descriptor sets
        - Draws, going through all pipeline stages that do not explicitly write to the color attachment.
        - Writes to the color attachment once the image is signaled as a render target.
    - Ends the render pass, with the fence signaling the completed image as available for presentation, and the uniform data available to update.
- Present the completed image to the screen.

The synchronization scheme used in this render loop is used to prevent tearing and direct writes to the image being presented. More parallel execution schemes could separate non-dependent drawing operations into different command buffers and submit them to be executed in parallel, either across multiple threads of execution, or one-by-one from a single thread. This approach would be preferable for a deferred rendering pipeline, or an application using multiple GPU’s, each responsible for drawing a portion of an image.

TEARDOWN PHASE

Once the render loop has been exited and the program is to be terminated, all created Vulkan objects must be destroyed using `vkDestroy`-prefixed functions, and all memory allocations must be freed using `vkFree`-prefixed functions. There are only a few dependencies in the destruction order. The instance destruction command, `vkDestroyInstance`, should be the last Vulkan command executed, preceded by the device destruction command, `vkDestroyDevice`. `vkDestroyShaderModule` will destroy any state data associated with `VkShaderModule` handles. `vkDestroyBuffer` will destroy any buffer handle data for uniform, vertex, and index data, but this should be followed by calling `vkFreeMemory` to free the underlying device memory. Images that are used through `VkImageViews` must also be destroyed in a specific order. First, call `vkDestroyImageView` to destroy the handle data for the passed-in `VkImageView`, then call `vkDestroyImage` to destroy the handle data for the passed-in `VkImage`, then finally, free the memory used on the device for the image by calling `vkFreeMemory` on the corresponding `VkImageMemory` handle. Descriptor set layouts, descriptor pools, all synchronization objects, all framebuffers, and all pipelines, pipeline layouts and render passes created must be destroyed with their respective `vkDestroy` commands. `vkDestroyCommandPool` will destroy the passed-in command pool handle, freeing all command buffer objects allocated from the respective pool. For destroying the swapchain, first destroy all swapchain images, using the above order of destroy and free calls for `VkImageViews`. Then call `vkDestroySwapchainKHR` on the swapchain handle. Finally, once all objects have been destroyed, and all memory has been freed, destroy the logical device by calling `vkDestroyDevice` on its handle, and destroy the Vulkan instance by calling `vkDestroyInstance`. After doing all of the above and calling `vkDestroyInstance` with no errors, you are free to terminate the program.

PREVIOUS WORK/RELATED WORK:

Other options, in place of or in addition to using Vulkan exclusively for creating and/or maintaining a cross-platform solution either involve some form of continued usage of OpenGL, or using separate platform-specific graphics API "backends" and linking them through a common render hardware interface, or RHI.

For continued usage of OpenGL, the same problems described in the introduction apply, namely that OpenGL support is dependent on Apple supporting it, and Apple changing or removing OpenGL support to better support Metal is a nonzero possibility with consequences for continued OpenGL base code usage. The Vulkan implementation used by MoltenVK is not dependent on Apple supporting it and is likely a better option moving forward.

Using WebGL 2.0, an API based on OpenGL ES 3.0, would allow for OpenGL-based cross platform development using a web browser. However, as it is based on OpenGL ES 3.0, the available feature set is a subset of the full OpenGL API, even compared to macOS's last supported OpenGL version of 4.1. WebGPU, the working name of a future web-based API for hardware-accelerated graphics, could be an option in the future, but it is currently not widely available, and does not have the same breadth of documentation, tutorials and example code that the other options have. Additionally, the required curriculum for computer science students at Cal Poly supplies students with significantly more experience in C compared to web-based languages like HTML and JavaScript. For the purposes of learning about computer graphics programming concepts, programming in a familiar language, like C or C++, would be preferable to learning both the concepts and the language itself.

An additional option here is to implement separate hardware rendering "backends", potentially a Metal or Vulkan backend for macOS devices, potentially a DirectX, OpenGL, or Vulkan backend for Windows devices, and an OpenGL or Vulkan backend for all others. Switching backends would involve reading preprocessing flags set by the operating system, and using a common RHI to abstract away the API calls. The immediate advantage is that the existing OpenGL code can continue to be used on all devices, and a RHI can be designed on top of it, exposing just the features that are supported across all targeted graphics APIs. Then, the interface can check for existing graphics API support, and use a supported graphics API. For example, if the "__APPLE__" preprocessor flag is set, a Metal backend could be used in place of a "default" OpenGL backend, and using the RHI would use the Metal API. This RHI approach is very similar to the Vulkan base code's approach, except the Vulkan base code abstracts away the single Vulkan API. Compared to Vulkan, Metal is significantly less verbose and a backend using Metal could be easier to implement and update on its own, as compared to a Vulkan backend.

There are a few obvious disadvantages to this approach, both from a development and a maintenance perspective. From the development perspective, someone developing this approach would have to research multiple APIs and determine their similarities and differences in program setup and data management, hardware abstraction, and available features. They would also have to select the common features, map them to a common RHI, and confirm that their usage of all APIs through the RHI produces the same results across all targeted hardware. From the maintenance perspective, graphics APIs may change and diverge in their supported features with new updates. maintaining such a project would involve researching the relative changes in each supported API, potentially adding new features or removing deprecated features to and from the RHI entirely if features converge or diverge significantly, and debugging differences in RHI output across different API backends, some of which might not be available to test or fix by the instructor or project group member, due to differing operating systems or hardware. Compared to maintaining a single code base using a single cross-platform API such as Vulkan, the relative maintenance and support work involved could be enormous.

ALGORITHM:

OVERVIEW OF SOLUTION:

The additions to the base code mostly come in the form of support for texturing, debugging model input, loading different file types of model input, automatic loading of shape files using the standard library's `filesystem`, hierarchical transforms of multi-shape objects, a second pipeline used to render all scene geometry as wireframes, and a continuous integration testing script.

TEXTURING:

The texturing solution starts with a `TextureLoader` class, which manages loading images from disk into CPU memory, into a staging buffer on the GPU, and then finally into an image buffer, where they will be used as texture input during rendering. The `TextureLoader` class contains handles to the device bundle and the command pool, as well as a vector of Texture class instances. Using this class from the `VulkanGraphicsApp` class requires supplying its constructor with the device bundle, calling setup to supply it with the command pool, and then calling `createTexture`, with the path to an image to be loaded as a texture.

Internally, when `createTexture` is called, a Texture object is created, and the `stb_image` library is used to load the image's pixel data into an array of bytes, as well as collect information about the image width, height, and number of channels. The size of this array is used to create a `VkDeviceSize` object, which is used in staging buffer creation. The staging buffer is created, requesting memory that is either host-coherent or host-visible, that is to be used as a transfer source. We copy our image data from the CPU into this staging buffer, and then free the data on the CPU.

Image creation follows the staging buffer upload. We create the image, with the `Texture` object holding the handle. Then we query the device for a memory that is both of a sufficient size, and device-local, which typically provides faster access times than our staging buffer requirements of host-coherent or host-visible. We then allocate the memory and bind the memory to our image handle. The layout of bytes in the staging buffer is then transitioned to an optimal transfer layout specified by `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, then the image is copied from our staging buffer memory to its final location in device memory. The layout of bytes in the final buffer is finally transitioned to a layout optimal for shader read access, specified by `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`. Now that our staging buffer is no longer needed, we destroy it and free its memory.

A `VkImageView` is then created, that references the image memory, and specifies its format, type, and other configuration parameters left as defaults. Finally, a sampler is created, which specifies the type of texture filtering to use, the addressing mode for texture coordinates outside the range of [0, 1), whether to enable anisotropic filtering, and various other parameters that have been set to a simplified mode that should enable texture usage, regardless of what device features are available.

`createDebugTexture` does the exact same as the above createTexture, although instead of loading an image from disk, it initializes a 2x2 pixel image by creating an array of bytes in the same format returned by a call to `stbi_load`.

DEBUG SHADER:

The shaders in this project include a debug fragment shader, which should aid any debugging efforts associated with importing new textures and models. During rendering, holding 1-5 on the keyboard will change the shading behavior, or shading layer as it is described in the code, of all objects rendered using this debug shader. Releasing the number key will revert the shading behavior to its original form.

Internally, this behavior is managed through `observeCurrentShadingLayer` and `recordShadingLayers`. `recordShadingLayers` determines the shading layer per multi-shape object specified by the user during initialization and records the object's name and its user-specified shading layer, in order to revert back to this shading layer once there are no number keys being held down. `observeCurrentShadingLayer` examines the current global shading layer, determined by which keys are held down, and if no keys are held down, the shading layer of each multi-shape object is determined by the shading layer stored by `recordShadingLayers`. If a key is held down, the global shading layer determines the shading layer for all multi-shape objects.

Key	Description
1	Colors according to the material properties of the object, with Blinn-Phong shading.
2	Colors according to the normals of the object.
3	Colors according to the texture coordinates of the object.
4	Colors according to the texture mapped onto the object.
5	Colors according to the texture mapped onto the object, with Blinn-Phong shading.
6	Colors according to the user-specified shading layer. The same as not holding a key.
Z	Binds a pipeline that uses many of the same resources, the difference being that this pipeline renders polygon edges only (wireframe).
W	Dollies the camera forward.
A	Strafes the camera to its left.
S	Dollies the camera backward.
D	Strafes the camera to its right.
I	Moves the dummy forward.
J	Moves the dummy to its left.
K	Moves the dummy backward.
L	Moves the dummy to its right.

Figure 5: Key Bindings

GLTF LOADER:

INTRODUCTION TO THE GLTF FILE FORMAT:

the glTF file format was designed and specified by the Khronos Group for efficient transfer of 3D content over networks. As compared to the Wavefront OBJ file format, the file format itself is two files. One file is a binary file, which contains all data. The other file is a JSON object containing metadata, which specifies how to read the data contained in the binary file. There is also a companion format, glb, that is completely binary. Both glTF and glb can be smaller in size for similar geometry as compared to OBJ, due to most of its data being stored in binary. It can also specify additional rendering information, such as cameras, scene graphs, samplers, as well as material, skinning, and animation data, whereas the OBJ file format can only specify 3D geometry. The MTL file that often accompanies .obj files can additionally specify material information.

The new file format does have some drawbacks, as compared to the OBJ file format. It is much more difficult to create and edit glTF files, as OBJ files are in plaintext, and editing glTF files requires either editing binary directly, or using a dedicated program. The data access pattern is also more complicated for simple and small files, and the complexity only grows as more features of the glTF file format are used.

GLTF LOADER IMPLEMENTATION:

The glTF loader implementation is defined in load_gltf.cc. Overall, calling `load_gltf_to_vulkan` on a glTF or glb file will use TinyGLTF's loader implementation to load the contents of the metadata and binary file into its own in-memory data structure. Then, a scene graph of the file's default scene is constructed, the `tinyglTF` data structure is converted into something usable by the multi-shape object class, and the resulting multi-shape object is returned.

Internally, the helper function `process_gltf_contents` constructs a scene graph from the information in the file describing its default scene.

This is done by storing all TinyglTF nodes (analogous to glTF nodes) in the default scene into a tree structure. The SceneGraph class contains this tree structure, which is a vector of shared pointers to `TreeNodes`. Each `TreeNode` stores the `TinyglTF` node, a transformation matrix for the current node, a shared pointer to its parent TreeNode, and a vector of shared pointers to any children.

Each `tinyglTF` node may have a list of children associated with it. When rendering a scene, one or more root nodes are specified. All `tinyglTF` nodes may reference transform data, children and/or a mesh that contains one or more mesh primitives.

`ConstructCTMTree` is a recursive function that is called at top-level once for every root node: the nodes that are explicitly specified in the default scene. It collects any transform data at the current node and converts it into a matrix, calls itself on any children, specifying the current node as the parent, then assigns parent-child relationships for all nodes adjacent on the tree, and then finally stores the completed TreeNode in the `SceneGraph`.

`computeCTM` returns the current transformation matrix of the node, according to its placement in the default scene. This is constructed by making a copy of the current node's node-local matrix, looping up the hierarchy and multiplying the parent's node-local matrix by the current matrix, and finally returning the built current transformation matrix.

Once the scene graph is constructed, we iterate through all the nodes, compute the node's current transformation matrix, and if the node has a mesh, iterate through all its mesh primitives, acquiring accessors to the primitives' vertex attributes: their position data, their normal data if it exists, and the data of the first set of texture coordinates, if it exists. The glTF specification allows for a single mesh primitive to have multiple sets of texture coordinates, but this implementation only collects the first set.

Each of the primitive attribute accessors that are acquired describe the type and count of the data contained in the binary file, as well as a bufferview, which additionally describes the index of the buffer to access, the byte offset of the start of the data to read from the buffer, the length of the data to read in bytes, and a byte stride, used to support interleaved data, which specifies the distance in bytes between each value.

The helper functions `process_vertices`, `process_indices`, `process_normals`, and `process_texcoords`, each take in this accessor, and use it and the corresponding bufferview to extract and transform the data from the buffers and place it into the multi-shape geometry class. The vertices and normals are pre-transformed according to the current transformation matrix. Additionally, because indices of mesh primitives in glTF files always start at zero, process_indices and process_texcoords adds the cumulative number of indices added from the previous mesh primitives to the extracted index numbers, to replicate the way that the OBJ file formats store index data of multi-shape objects.

Finally, a bounding box center is calculated per primitive mesh, with the results being stored in a class member of the multi-shape object class. This allows for effective hierarchical modelling, even after the geometry data has been sent to the GPU memory and destroyed on the CPU memory.

HIERARCHICAL TRANSFORM SUPPORT:

Support for hierarchical modelling comes through the addition of bounding box center calculations in the geometry loading phase, described above, as well as the transition of assigning a descriptor set to each shape in the scene, rather than to each multi-shape object. Using a matrix stack implementation from CPE 471 base code that used OpenGL was immediately transferrable and resulted in an identical visual appearance using animation data imported from a project that used OpenGL.

AUTOMATIC ASSET LOADING:

This project was upgraded to C++17 to allow for the use of `filesystem`. Previously, the user had to specify all the shape data files that were being added to the project manually, name them, and then remember the assigned name when configuring shading and transform parameters. With the addition of the recursive function `loadShapeFilesFromPath`, all that is required is for the user to provide the name of the directory from which to load shape data files. From there, `loadShapeFilesFromPath` traverses the directory structure using depth-first search, locates .obj, .gltf, and .glb files, loads their contents into a multi-shape object, using the `load_obj_to_vulkan` and `load_gltf_to_vulkan` functions, and adds the created object to a map, with the key being the name of the file, without the extension. This reduces the confusion with naming conventions and encapsulates calls to the loading functions for every file to be included.

WIREFRAME PIPELINE:

A second pipeline object was created to support drawing with Vulkan's non-solid fill mode, `VK_POLYGON_MODE_LINE`. Unlike OpenGL, this mode is static, and not able to be changed at runtime for a given pipeline object. This second pipeline object uses many of the same resources as the first pipeline object. Specifically, the same render passes, the same framebuffers, the same depth buffer, the same pipeline layout, descriptor set layout, and descriptor sets are used. The only difference is the fill mode. A second set of command buffers, one per swapchain image, is initialized. This new set of command buffers is emplaced in the vector of command buffers behind the original set, the difference being that the newly created pipeline is bound during the command buffer recording. When Z is held during rendering, the starting index into the command buffer to submit is changed, to bind to the new pipeline. Releasing Z will revert the index, and the original command buffers that bind to the original pipeline are used.

CONTINUOUS INTEGRATION GITHUB ACTION SCRIPT:

This project's development goal of being cross-platform was assisted by a Github Action script, contained in the .github/workflows directory. In summary, the script runs on the latest version of Ubuntu, installs the Vulkan SDK, version 1.1.1, updates the apt package manager's package information, installs the required project dependencies of glfw(libglfw3, libglfw3-dev) glm(libglm-dev), configures CMake to output its products into a 'build' subdirectory, builds the project, and then runs tests defined by the CMake configuration. Further expansion of this project could include building on more architectures, and expanding the number of tests being run. This project was also manually tested to run on a 14-inch MacBook Pro, running macOS version 12.3.1, with an M1 Max processor with 10 CPU cores, 32 integrated GPU cores and 32 GB of LPDDR5 shared memory, and a custom-built Windows PC, running Windows 11 version 21H2, with an Intel i5-8600k CPU and 16GB of DDR4 system memory, with both an Intel UHD Graphics 630 integrated GPU that shares system memory, and an NVIDIA GTX 1070 Ti dedicated GPU with 8GB of GDDR5 device memory.

RESULTS:

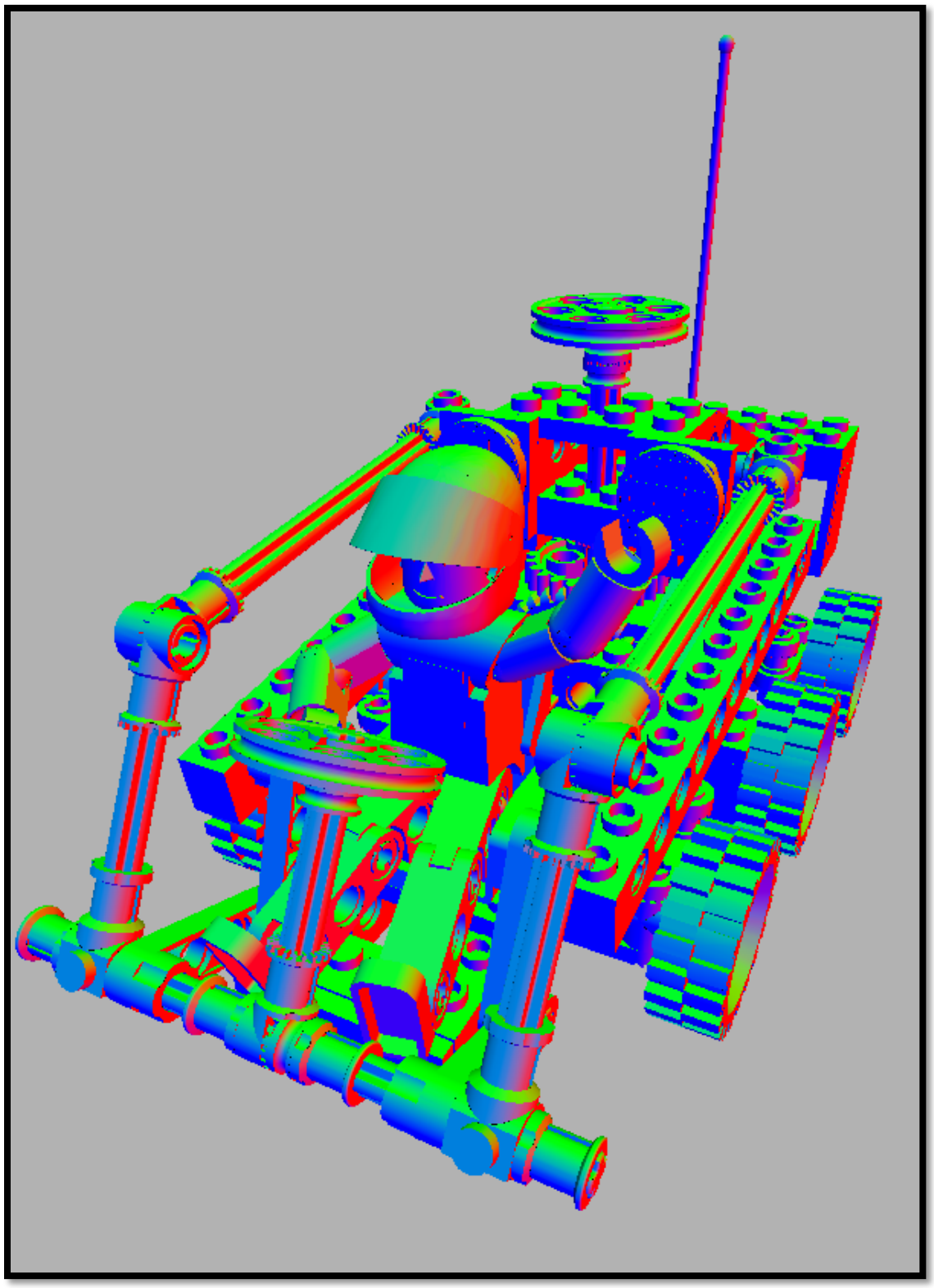

Figure 6: Normal Shading Layer of Buggy.glb

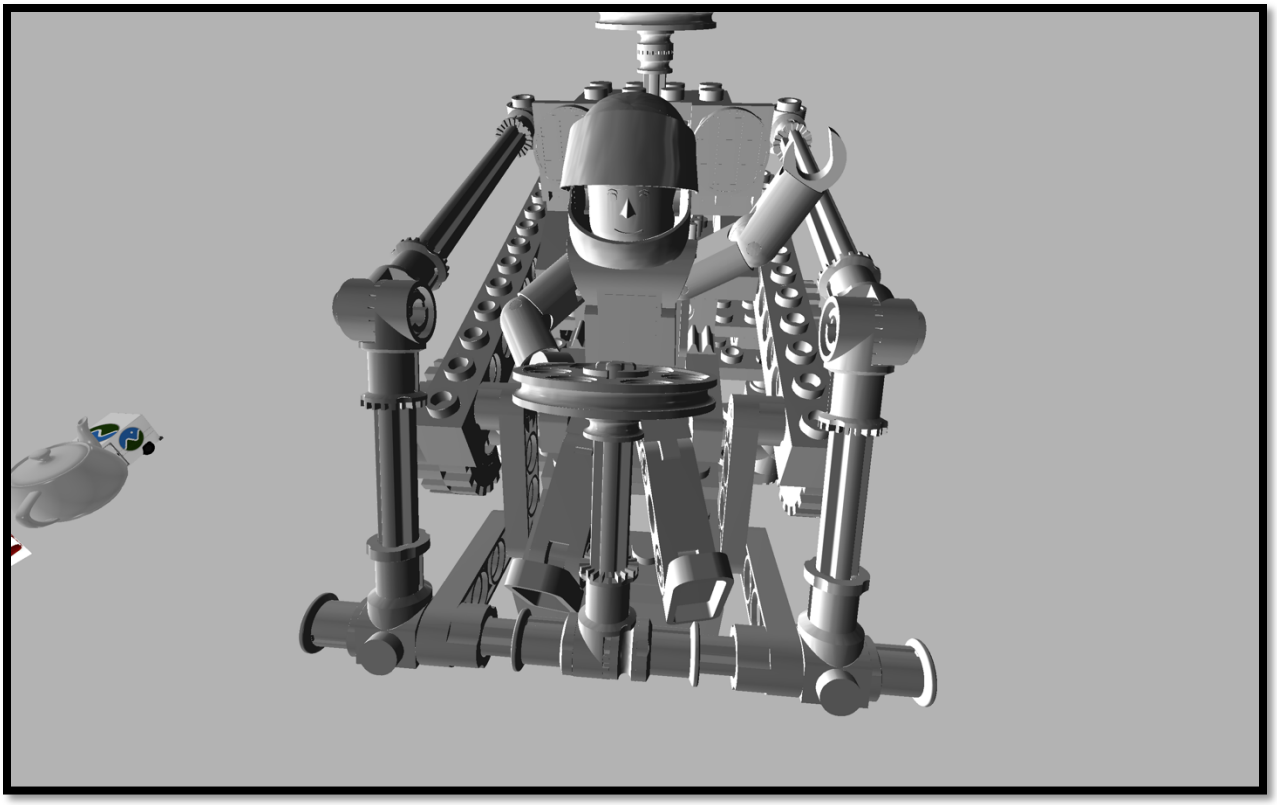

Figure 7: Blinn-Phong white material shading of Buggy.glb

Figure 8: Composite test scene, angle 1

Figure 9: Composite test scene, angle 2

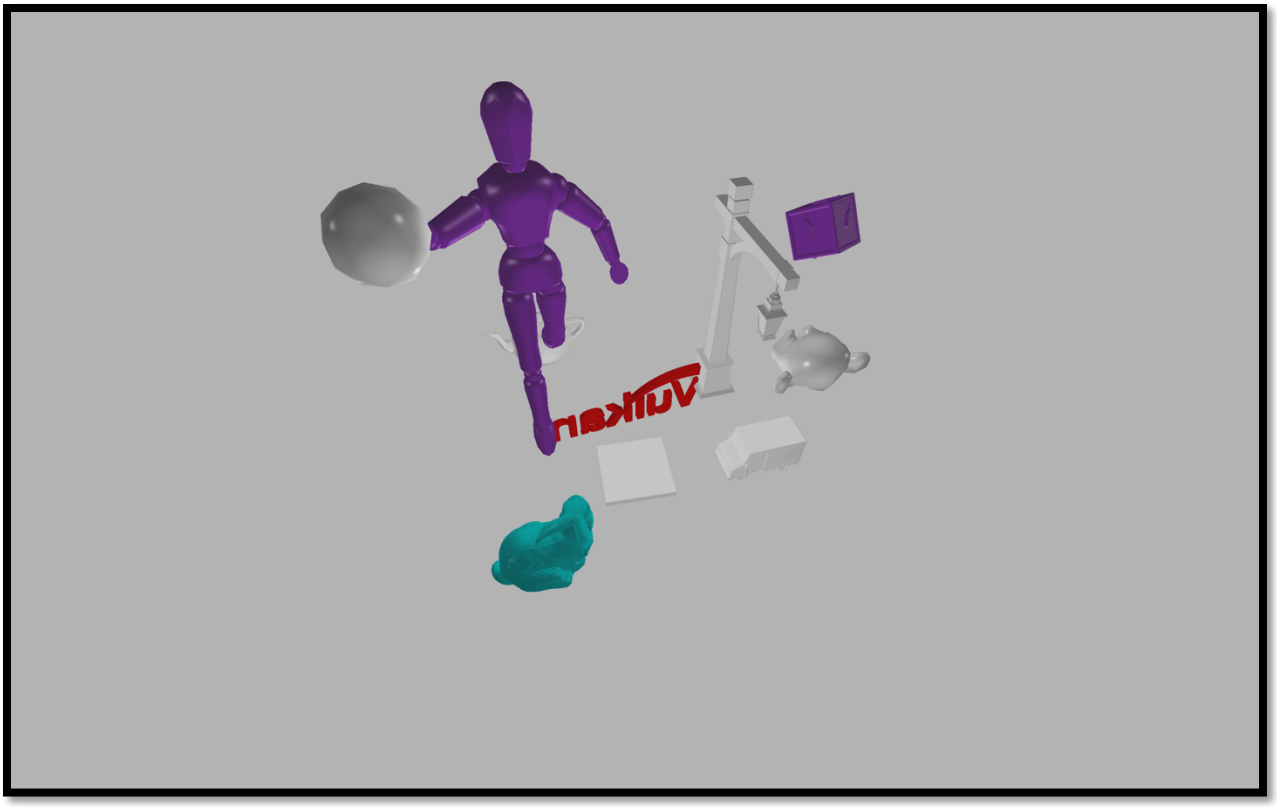

Figure 10: Blinn-Phong shading layer

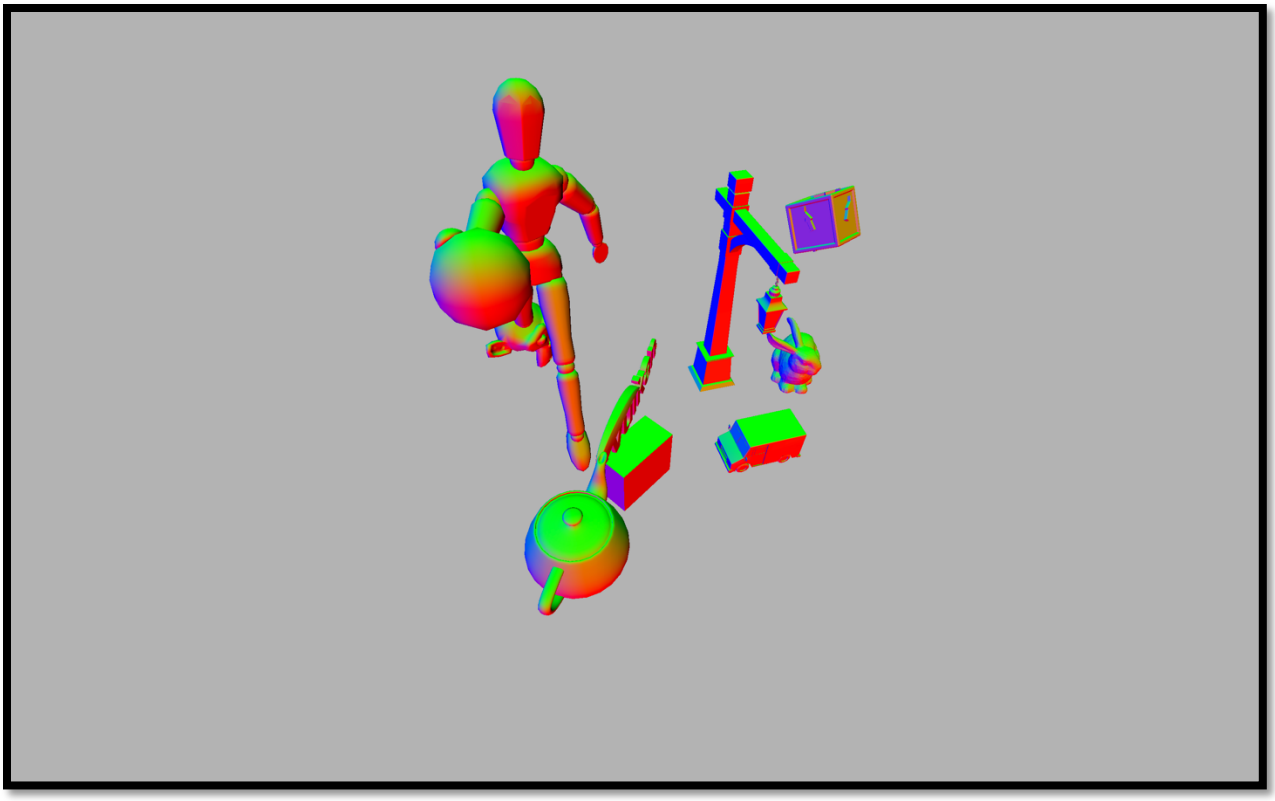

Figure 11: Normal shading layer

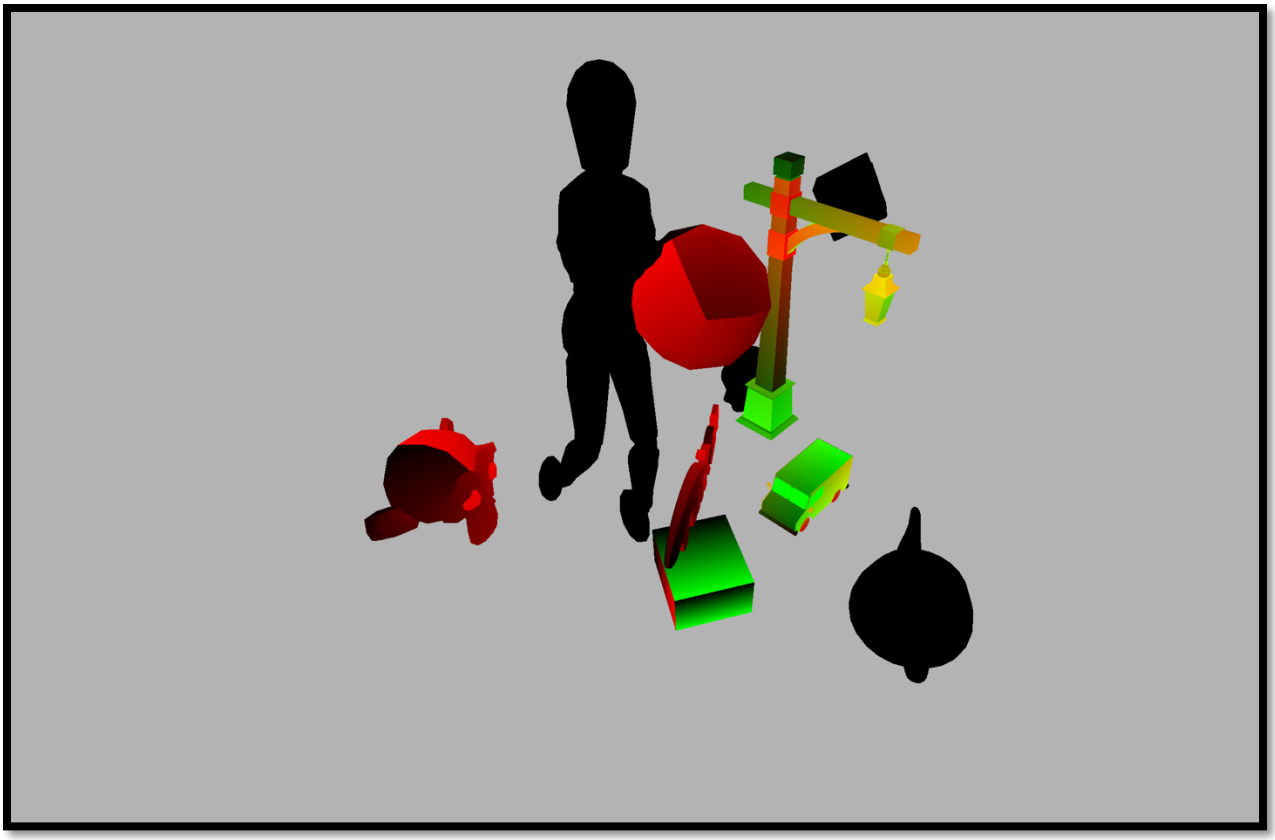

Figure 12: Texture coordinate shading layer

Figure 13: Unshaded textured shading layer

Figure 14: Shaded textured shading layer

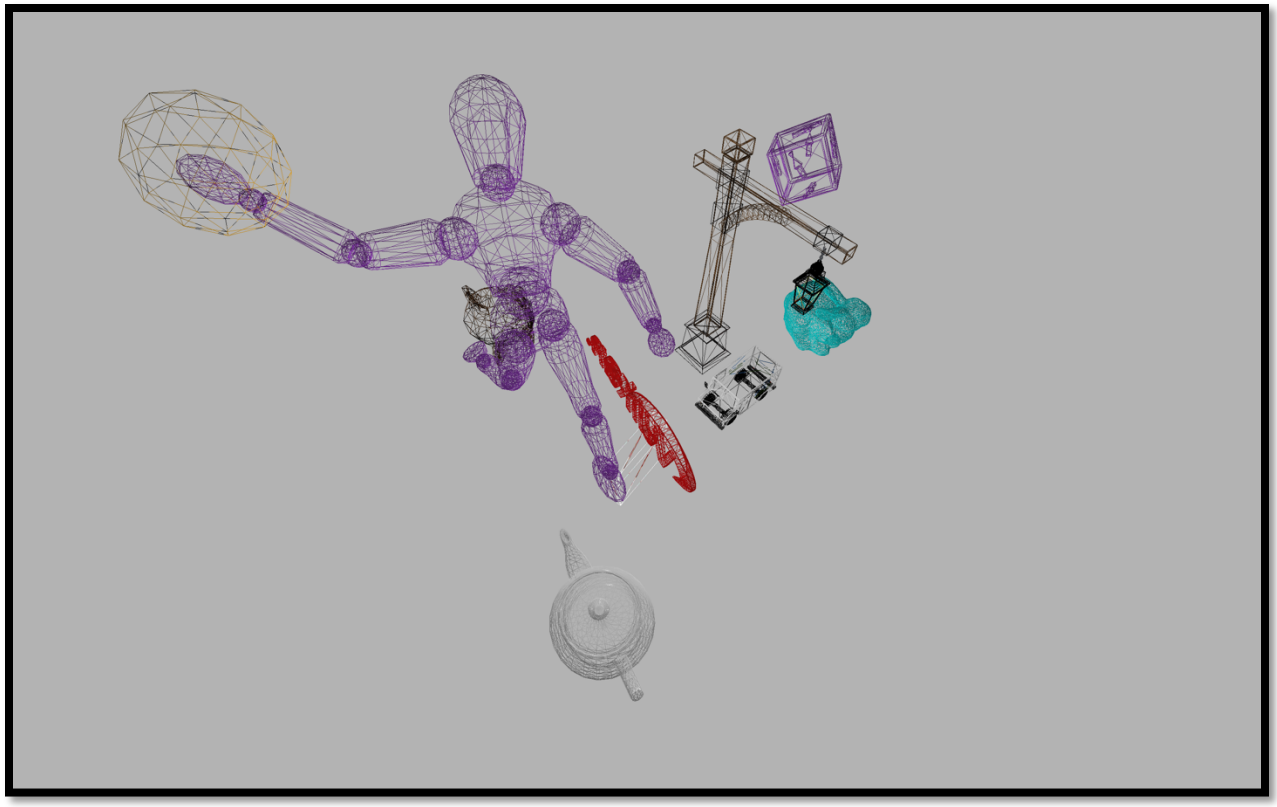

Figure 15: Composite test scene, wireframe

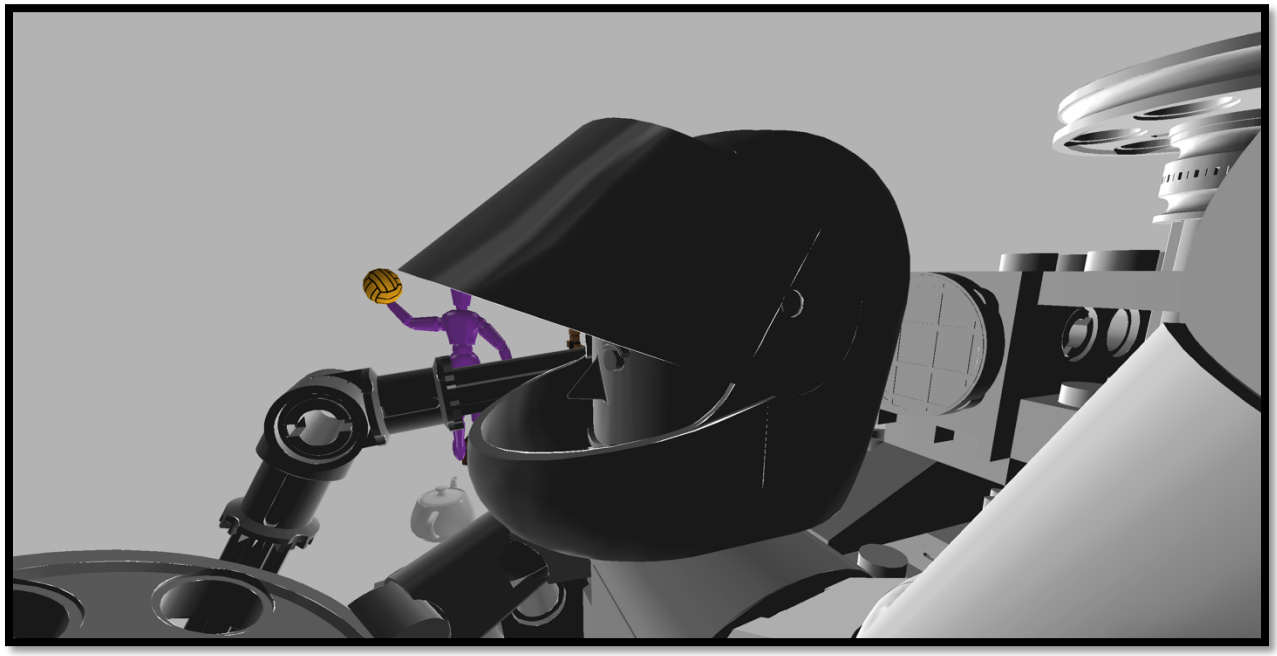

Figure 16: Buggy closeup

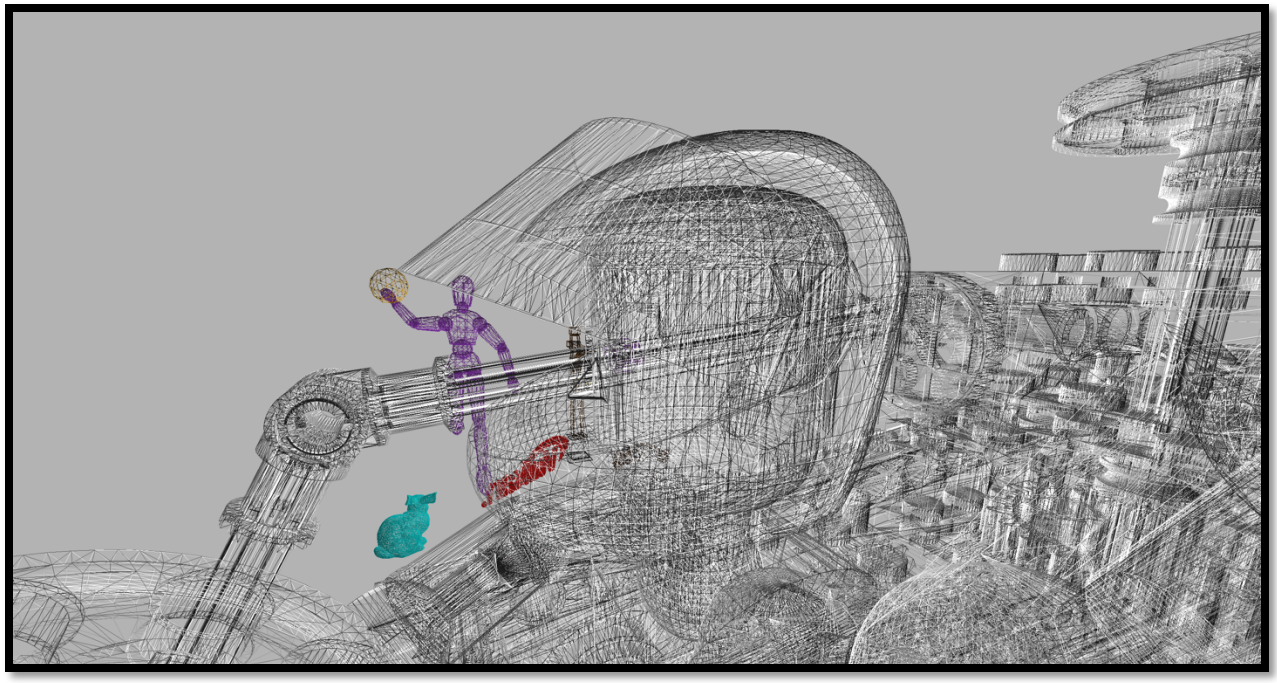

Figure 17: Buggy closeup, wireframe

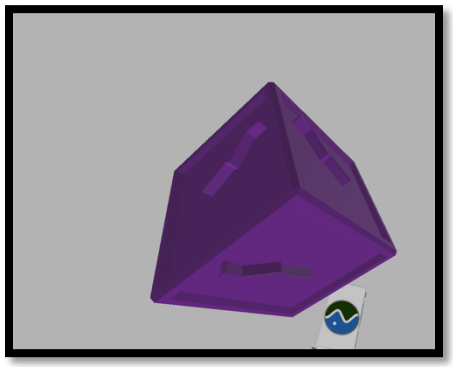

Figure 18: glTF Orientation Test cube

Figure 19: Applying descriptor set uniforms per-shape

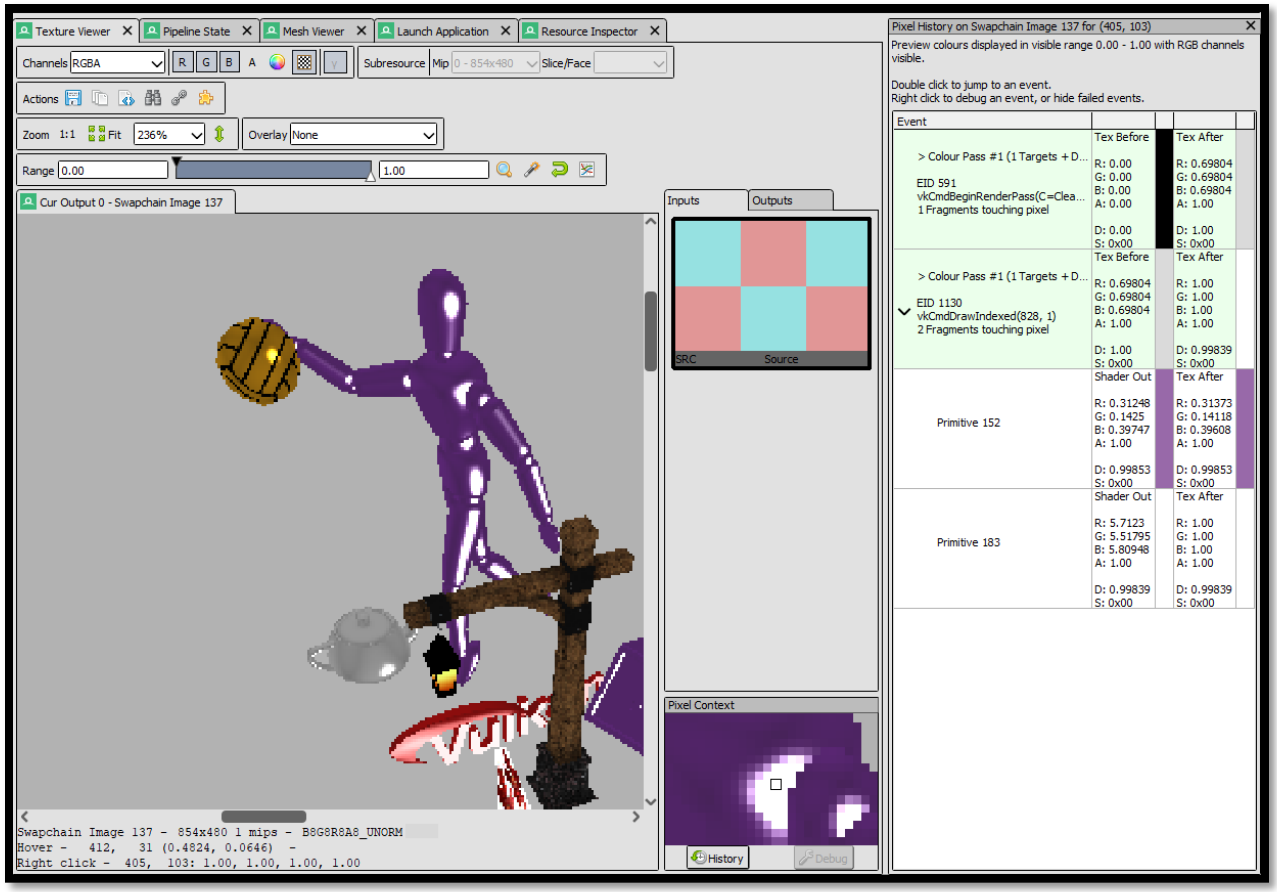

Figure 20: RenderDoc debugging, pixel history

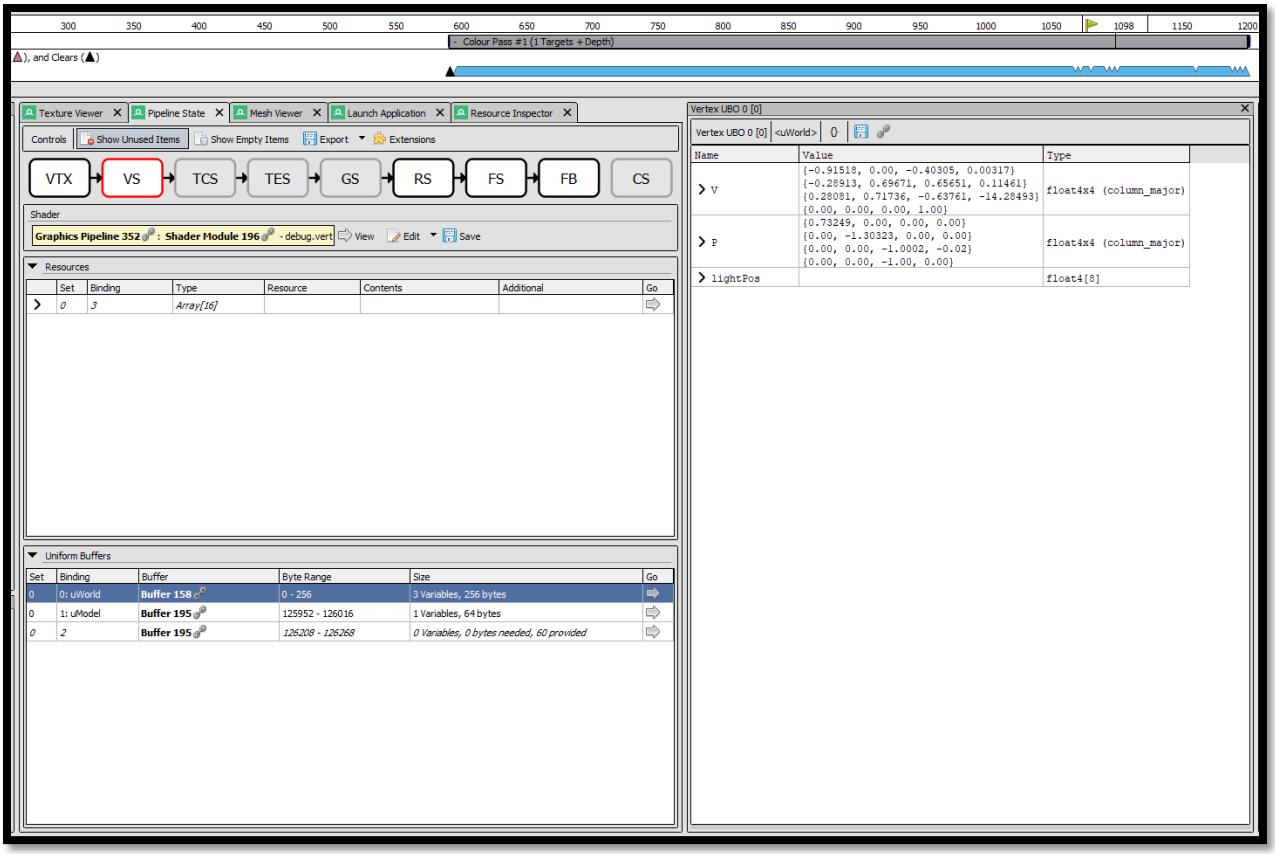

Figure 21: RenderDoc debugging, uniform buffer view

REFLECTIONS:

Overall, I am now comfortable enough with the state of the code, and with Vulkan as a whole, to be able to write graphics programs using Vulkan, either by improving and modifying the starter code, or by starting a project from scratch, if the performance goals or specific task requires it.

That was not always the case. This project was a continuation of a previous project, and it took a considerable amount of time to get acquainted with the existing base code, which was already massive, with deep inheritance structures, a myriad of third-party libraries, and subtle bugs. It took many weeks of deciphering the base code's own wrapper classes to finally feel comfortable enough to modify and extend on them without creating more headaches for myself. A first attempt was an attempted rewrite of the multi-instance uniform buffer class, to accept combined image samplers as well as generic struct data. That consumed about two and a half weeks of debugging time, with no real progress other than understanding what the multi-instance uniform buffer class actually did, and where to start adding support for combined image samplers in the descriptor set layout.

RenderDoc, an MIT-licensed graphics debugger, was an immense help to development. It has full support for debugging Vulkan API calls, determining state of Vulkan API objects, inspecting contents of GPU memory and pixel history for each shader stage, and many more useful features. It is available and officially supported on Windows and Linux operating systems, and building for macOS is an option, but it is not guaranteed to work.

Having support for precompiled shaders in every API version is immensely useful but can take some getting used to. Performing command-line compilation before building with an IDE of your choice avoids the typical OpenGL headache of not knowing about shader compilation errors before runtime. It also avoids the long string of OpenGL errors cluttering up console output. However, if you do not explicitly set the SPIR-V files to rebuild on editing their source in your respective IDE's project build settings, you will be running shader code that does not match up with a shader code file that is saved to disk, which can be frustrating if you don't set this up properly.

There is a lot more available power and control that comes with the state object programming paradigm, as opposed to the global context paradigm of OpenGL. The comparison is inversely analogous to my experience going from learning C++ as a first programming language to learning Python. At first, I was blown away at its simplicity and ease of development. It was only later when I started to run into the limitations of its global interpreter lock, and lack of the efficiency that comes with compiled languages. I think most scratch projects, feature tests, and small projects can do a lot with OpenGL, but I could definitely see myself doing more with Vulkan in the future, especially with highly-parallelizable work.

FUTURE WORK:

Work that could improve upon the current implementation could include the following:

CREATION OF MULTIPLE PIPELINES

Currently, the code creates two pipelines, with a shared vertex shader and fragment shader pair. One draws filled polygons, and the other draws polygons as wireframes. Swapping between them during runtime is achievable by holding and releasing Z. The related command buffers draw the entire scene using a single pipeline object. An improvement to this would be adding a user-friendly interface to create pipelines, change any relevant or commonly-changed settings, bind different shader modules and descriptor sets, and swap between them during runtime, in order to color individual objects, or the entire scene, differently. The debug shader is a naïve approach to this problem in the original codebase, and GPU performance could improve greatly with this addition.

DIFFERENT TYPES OF SHADERS OR PIPELINE SETTINGS

Vertex shaders and fragment shaders are sufficient for a simple graphics application, such as the student programs completed in CPE 471, but Vulkan supports many different types of shaders, including tessellation control and evaluation shaders, geometry shaders, and compute shaders. For raytracing or collision physics applications specifically, there are ray generation shaders, intersection shaders, any-hit, closest-hit, and miss shaders. An improvement to this code to make it sufficient for more advanced usage would be to create different pipeline initialization parameters, to link together arbitrary numbers and types of shaders, as well as providing parameters to change other pipeline-related settings, such as the polygon mode, in order to draw wireframes. Adding in a general-purpose compute pipeline would be another more advanced addition but could be instructive and potentially useful for highly parallelizable work unrelated to rendering.

TEXTURING

The most essential fix is to allow for different descriptor sets to bind different textures. Currently, each descriptor set binds all stored textures as an array of textures, and indexes into them on a per-shape basis. While the current implementation is set up to allow using multiple textures to color a single shape, this is not remotely sustainable at scale, as each descriptor set is limited in the amount of uniform data it can contain. Certain devices are more limited in this regard than others: the MoltenVK implementation used in macOS devices allows for sixteen simultaneous texture bindings, while a comparable Windows system running Vulkan directly allows for several thousand simultaneous texture bindings. This fix should be simple. During descriptor set updates, use some created mapping between the shape's descriptor set, and a `VkImageInfo` struct that contains the array of textures that will be used for that shape only. The user of such a change will have to specify which of the loaded textures will be included in which descriptor set.

This fix is not as essential as the previously described fix, but the staging buffer is currently being created and destroyed for each texture upload. A better implementation would be to allocate a staging buffer that is sufficiently large for all of the image uploads, preserve and use it throughout initialization of all textures, and then destroy it.

Similarly, each texture has its own sampler. This is an inefficient use of memory when each sampler has the same initialization parameters. It is more efficient for either all textures to use the same sampler, or to initialize a few different samplers, and swap between them as needed.

MORE EFFICIENT GLTF LOADER

A drawback of the current glTF loader is that it removes the efficiency of the glTF file format's usage of the node-mesh-primitive relationship. For example, a single mesh primitive may be included by reference, using an index, in several thousand nodes, each with a different place in the scene graph, with their own transformation matrix. The current implementation would create a copy of this primitive's data and assign that copy to the node, potentially resulting in wasted memory describing duplicate primitives. This results in a nice one-to-one mapping with how the OBJ file format describes shapes, but the glTF format can effectively describe instanced objects without this copying of data. An improvement to this would most likely involve using instanced rendering by default, by modifying the multi-shape object class to either load data onto the GPU if it hasn't seen it before and simply reference the data and the respective node transform, or if the primitive has been used before, to use the reference created upon the initial load onto the GPU, and the new transform. Creating a table per glTF file, mapping primitive IDs to GPU memory locations to keep track of this would most likely be useful.

Another fix to the glTF loader would be to traverse the tree structure depth-first, rather than element-by-element in the vector, so that the operations in computeCTM can be converted into a matrix stack calculation. This can lead to a modest performance improvement, for files with extremely deep node hierarchies in the default scene.

TEST HARNESS TO GATHER AND COMPARE PERFORMANCE METRICS

The tests run by in the CMake test configuration mainly test the behavior of the buffer classes, and don't measure performance at all. There is a timer class included in the code that was used during performance-related debugging that is a minimal wrapper over std::chrono that can do some of this. Something that is of interest during development of high-performance applications, graphics included, is whether any changes made to the code have any performance impact, either positive or negative. Something that could aid and inform future development of this code is to create a test harness that collects, reports, and optionally archives performance results. This could be measuring startup time, measuring frame times during rendering, or even the runtime of specific functions. Archiving this data could take the form of writing out to a CSV file, named by a timestamp, git commit hash, git commit name, or some identifier that represents the codebase at a particular point in time, along with an initialization block, for timings of operations during initialization, as well as frame times, stored as the frame number, followed by the frame time. You could then graph those results and compare across versions of code to inform design decisions.

REFERENCES/BIBLIOGRAPHY:

APPENDICES:

API Naming Conventions: https://www.khronos.org/registry/vulkan/specs/1.3/styleguide.html#naming

GLOSSARY

INSTANCE

The entry point of the Vulkan API. Creating an instance via calling `vkCreateInstance` with an appropriate `vkInstanceCreateInfo` struct pointer initializes the Vulkan library and allows execution of additional Vulkan API commands. Upon creation, the instance is supplied with information such as the API version number, and a list of layer names and extension names to enable. This instance can be queried to retrieve information about devices available to it.

PHYSICAL DEVICE

A physical device represents a single device. On a simple system, there would be one physical device: a single integrated or dedicated GPU. Computers with specialized hardware configurations or with multiple GPUs installed would have one physical device for every complete implementation of the Vulkan specification.

LOGICAL DEVICE

A logical device represents an application-level view of a physical device. There can be more than one logical device per physical device, each with its own state and resources. Logical devices are the objects through which all device resources are acquired, queried, modified, and destroyed.

PIPELINE

A pipeline object encompasses the programmable shader pipeline. Pipeline objects are immutable by default, but certain states of a pipeline can be declared as dynamic on pipeline creation, which indicates that their state can be changed at runtime with command buffers. In addition, different built pipelines can be created and bound at runtime. Vulkan uses precompiled SPIR-V shader modules, which are then encapsulated in shader module objects and supplied to pipeline creation. Descriptor set layouts and optional push constant information is additionally specified during pipeline creation. The decision to make pipeline objects immutable by default means that some API usage will require the creation of more pipelines than as compared to a similar workflow using OpenGL, but the device driver is free to make more optimizations using parameters known to not be modified for the current pipeline.

DESCRIPTORS/DESCRIPTOR SETS/DESCRIPTOR SET LAYOUTS/DESCRIPTOR POOLS

Descriptors are pointers to buffers used by shader programs. These are read-only blocks of data that remain constant for the duration of a single draw or dispatch. Descriptors are grouped into descriptor sets, which are allocated from a descriptor pool. When binding descriptor sets to a slot in a pipeline for a shader program to access these resources, you must specify a descriptor set layout. A descriptor set layout is a collection of bindings, indicating the types of resources being bound, the number of resources, and additional data such as which shader stages these data will be associated with.

QUEUES/QUEUE FAMILIES

To increase graphics device command throughput by taking full advantage of multicore and multi-CPU hardware, the Vulkan specification provides for a queued execution model. A given physical device may support one or more queue families, representing one or more queues of the following types: Graphics, for draw commands, Compute, for non-drawing computation, Transfer, for generic memory operations and Sparse, for sparse memory operations specifically. A given physical device will group its available queues into queue families, each that may support different types of queue types.

For example, a physical device may have three queue families: the first supporting all four queue types with sixteen queues in the family, the second supporting only transfer queues with eight queues in the family, and the third supporting only compute queues, with eight queues in the family. It is implied, but not guaranteed, that submitting command buffers to the most specialized queue family for the specific operation type (Graphics, Compute, Transfer, or Sparse) will lead to better performance. Device manufacturers will typically supply Vulkan API optimization guides and best practices for their specific hardware. In this example case, a multithreaded application submitting 8 simultaneous transfer operations should be sent to queue family #2, while operations past that limit should consider submitting their command buffers to queue family #1, to maximize command buffer throughput, and thus execution uptime on the physical device.

COMMAND POOLS/COMMAND BUFFERS

Submitting instructions to the GPU, takes the form of submitting command buffers to a device queue. Command buffers contain one or more commands to be executed by a device. There are two types of command buffers, primary and secondary. Primary command buffers are directly submitted to the queue. Secondary command buffers can be ordered and joined together inside of a primary command buffer, which is then submitted in a single submit operation.

Command buffers are not directly created and are instead allocated from a command pool. Each thread submitting command buffers must allocate its own command pool, and a command pool may only be associated with a single queue family. Once a command buffer is submitted, it is not destroyed, and can be resent without any re-computation or resource reallocation, if desired.

BUFFERS

Buffers are representations of memory that is accessible by a device. Vulkan differentiates between a few distinct kinds of device-visible memory, each with potentially different performance characteristics. In a typical desktop computer architecture, with a single CPU, system memory, and a single GPU with its own discrete memory, there will be host-local and device-local memory: memory that is directly connected to the CPU and GPU, respectively. Additionally, on the GPU, there could be memory that is visible and writable by the host, as well as memory that is not. For single-submission writes to device memory, it is not required, but is often recommended to use a staging buffer. A staging buffer is a memory region in to host-visible, device-local device memory, where data is "staged" before moving to a final host-not-visible, device-local memory region, where the device potentially has faster access to it. Use a command buffer containing a `vkCmdCopyBuffer`, `vkCmdCopyImage`, or related copy command to transfer data, and synchronize this memory access with a synchronization object, such as a barrier or a fence.

IMAGES/IMAGEVIEWS/SAMPLERS

Images are similar to buffers, but they specifically reference image objects, and can be used as render targets for graphics pipelines. ImageViews are pointers to Image objects, combined with format information to read the image. ImageViews may read from all or just part of an Image, as is the case with some texture atlas or virtual texture usage patterns. Additionally, Samplers may be used to specify sampling parameters, used to control sampling and filtering of raw image data to retrieve a final color per pixel. Images accessed in this way are referred to as combined image samplers.

RENDER PASS

Render pass handles contain information about attachments, the usage of attachments in subpasses, the subpasses themselves, and dependencies between subpasses. A subpass can read input attachments and write to a number of color, depth, or stencil output attachments. A subpass may also specify shader resolve or multisample resolve operations.

FRAME BUFFER

Frame Buffers include render pass information, and information about all of the attachments used in a single frame's output.

SYNCHRONIZATION

As the Vulkan API is explicitly designed with parallel processing in mind, both on the CPU and the GPU, various API synchronization objects that can order execution for dependent operations are made available.

BARRIERS

Pipeline barriers can specify which stages of the rendering pipeline are dependent on each other, such as only combining framebuffers for deferred shading once all source framebuffers have finished operations. They are separated into execution and memory barriers.

EVENTS

Events have a set-and-wait syntax and can be used from both the GPU as commands placed in command buffers, or from the CPU. Event barriers occur in two parts: setting an event and waiting for an event. Commands occurring between the set and wait commands can occur at any time, but execution will pause at a wait command until the corresponding set command has been executed. This is only available within the same device queue, so its usage is limited to interleaving unrelated work within the same queue.

SEMAPHORES

Semaphores can be used across multiple device queues, unlike the previous barriers and events. They are sent as parameters to a queue submission, are used to synchronize between GPU tasks, and cannot be used to synchronize between CPU and GPU tasks. Once the command buffer associated with that queue submission command completes, the semaphores are released, and other command buffers waiting on those semaphores can begin work.

FENCES

Fences are explicitly designed for GPU-to-CPU synchronization. They can attach to a queue submission, in a comparable manner to semaphores, but the CPU application may check the status of the fence or wait until the fence is released before continuing operations, such as waiting for a draw call to complete before presenting the completed image to the screen.

LIST OF FIGURES

INDEX