Vulkan: An Introduction, The Process, and Takeaways

CSE 480: Departmental Honors

Brad Schmitz

CSE 480 - Departmental Honors

Vulkan: An Introduction, The Process, and Takeaways

TABLE OF CONTENTS

CSE 480 - Departmental Honors

Vulkan: An Introduction, The Process, and Takeaways

PREFACE

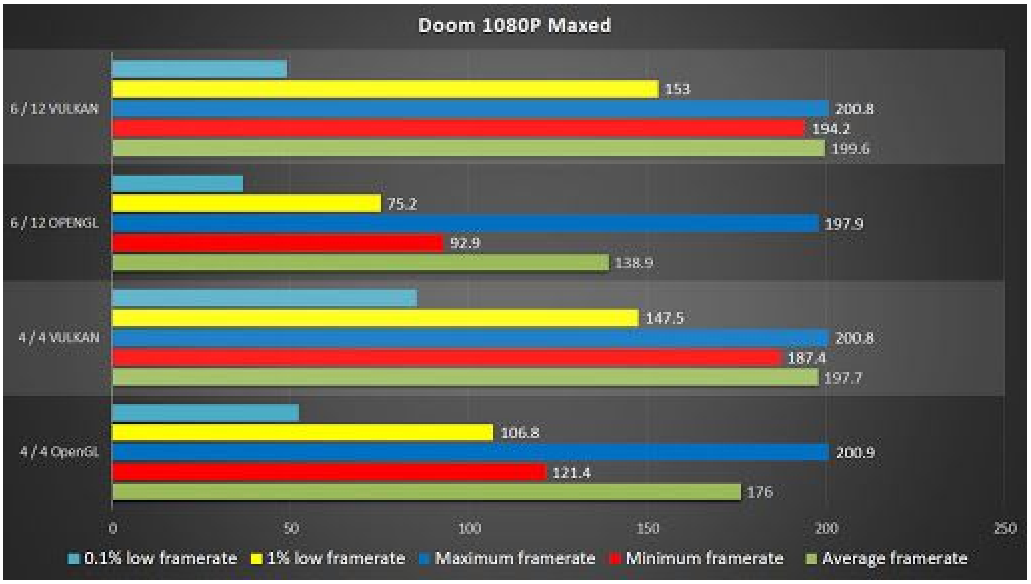
This paper is part of the submission for my CSE Departmental Honors (CSE 480) project. Originally, the scope of this project was to convert the game engine utilized in CSE 489 (Advanced Graphics and Game Engine Design) from OpenGL to Vulkan; however, due to difficulties of Vulkan and time constraints, this paper is serving as the “final project”. The current repository for this project is located **HERE**. Time permitting, I would still like to figure out how to get the game engine working in Vulkan. However, that will take some time I simply don’t have right now.

WHAT IS VULKAN?

Vulkan is a low-level and cross-platform graphics API, mainly utilized for 3D graphics applications like game engines and render engines. According to the Kronos Group (the makers of Vulkan), Vulkan is “a new generation graphics and compute API that provides high-efficiency, cross-platform access to modern GPUs used in a wide variety of devices from PCs and consoles to mobile phones and embedded platforms”. Vulkan is meant to give users the ability to have greater control over CPU / GPU usage, in exchange for being much lower-level and more verbose than alternative graphics APIs. In other words, Vulkan’s in-depth and low-level nature gives developers the ability to have better control over their graphics cards.

WHY / WHY NOT VULKAN?

As stated above, Vulkan provides a much better abstraction of modern graphics cards, allowing developers to utilize the power of their GPUs to a greater extent. Some of the most notable features include parallel processing, less driverhead, direct memory management, and more. Many of these features are discussed in greater detail in the **Vulkan’s Capabilities and Features** section. Because of this, Vulkan can offer significant improvements on performance compared to other graphics APIs like OpenGL and Direct3D, if utilized properly. For example, below is a comparison of OpenGL and Vulkan on the video game Doom (2016).



Comparing the performance of OpenGL and Vulkan in Doom (2016).

As can be seen, Vulkan offers a much higher average framerate compared to OpenGL, making for quicker and more engaging gameplay.

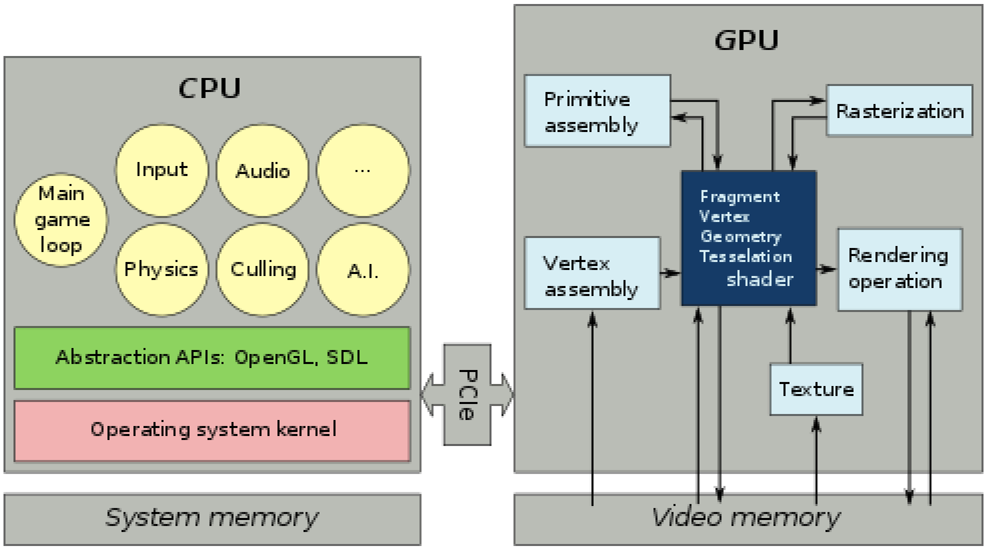
However, Vulkan is also meant for people who have experience in computer graphics and understand the fundamentals of computer rendering. If you are a beginner to computer graphics, you may find Vulkan to be unfriendly and unforgiving; in that case, it is recommended to learn OpenGL or Direct3D first to learn the fundamentals. Additionally, as stated above, Vulkan is also much more verbose than alternative graphics APIs. Many introductory Vulkan programs require hundreds of lines of code, which can be rather difficult to debug and manage if you have little experience with computer graphics.

To answer the question of “Should I learn Vulkan?” really depends on a few things: your willingness to learn new concepts, your current knowledge of computer graphics, and your intended usage for learning Vulkan. If you have some knowledge of computer graphics and are willing to put in the time and effort to learn Vulkan, go for it! If not, a higher-level graphics API or a general graphics / game engine (like Unity, Unreal Engine, etc.) might be more up your alley.

VULKAN'S CAPABILITIES AND FEATURES

Some of Vulkan's most notable capabilities include:

- **Architecture**: Vulkan's architecture is akin to OpenGL and Direct3D - the CPU and GPU split work, with the GPU doing graphics-related work (specified in the shaders) and the CPU doing mostly everything else.



The rendering system for Vulkan applications, such as game engines. The GPU executes shaders, while the CPU executes everything else.

- **Cross-Platform**: Vulkan code is transferable between operating systems, including Android, Linux, Windows, and macOS/iOS (through third-party support). Direct3D and OpenGL are both locked to specific operating systems
- **Reduced Driver Overhead**: Less code needs to be run for drivers, allowing for the CPU to focus on other computations.
- **Parallel Processing**: Vulkan has direct support for parallel processing, including running multithreaded / concurrent command execution. Vulkan also has built-in support for synchronization objects, like semaphores and fences, which help to control the concurrent execution of commands. Lastly, Vulkan has greater support for multi-core CPUs.
- **Shaders**: Unlike other graphics APIs, Vulkan requires that its shader be pre-compiled into an intermediate binary format called **SPIR-V** before program execution. This helps

to speed up the program initialization process and allows for more shaders per scene. However, this means that any changes to the shader need to be re-compiled, as Vulkan doesn't do it dynamically. However, there are tools that allow for runtime shader compilation, such as **SHADERC**, which comes with the Vulkan SDK. More info on Vulkan shaders **BELOW**.

- **Memory Management**: Vulkan has explicit control over memory management for many objects. For example, memory buffers require the creation of both a buffer object and a buffer memory object to store the data. This allows for experienced programs to dynamically manage their memory, which can help to speed up the program even more.
- **Ray Tracing**: Vulkan has extended and direct support for ray tracing, released in November 2020. For more information, visit **THIS LINK**.

CODING IN VULKAN: GENERAL SYNTAX

Vulkan has a very standard syntax, with all objects and commands having the same general "look". **OBJECTS** in Vulkan are in camel-case and start with the "Vk" prefix - like "VkObjectName". Examples of this include VkBuffer, VkInstance, VkQueue, etc. Similarly, **COMMANDS** in Vulkan are in camel-case and start with the "vk" prefix, but the "v" in "vk" is lowercase instead of uppercase. Additionally, many commands revolve around creating/destroying or allocating/deallocating information about an object - this is reflected in the command name. Examples of Vulkan commands include vkCreateBuffer, vkCreateSampler, vkDestroyBuffer, and vkDestroyInstance.

The general process for creating Vulkan objects goes like this:

1. Create a Vulkan object variable: **VkObjectName**
2. Specify info about the object in a CreateInfo struct: **VkObjectNameCreateInfo**
    a. This struct always has an **sType** (struct type) parameter, which tells Vulkan which type of create struct this struct is
3. Run the command to create the object: **vkCreateObjectName**
    a. Takes in the object and create info struct as references

This process is extremely standard - nearly every object is created using this methodology. The complexity of the CreateInfo depends on the object, but if you're making an object, there's a very good chance it uses this process. If memory-related, you may need to use "Allocate" instead of "Create".

THE VULKAN RENDER PROCESS AND OBJECTS

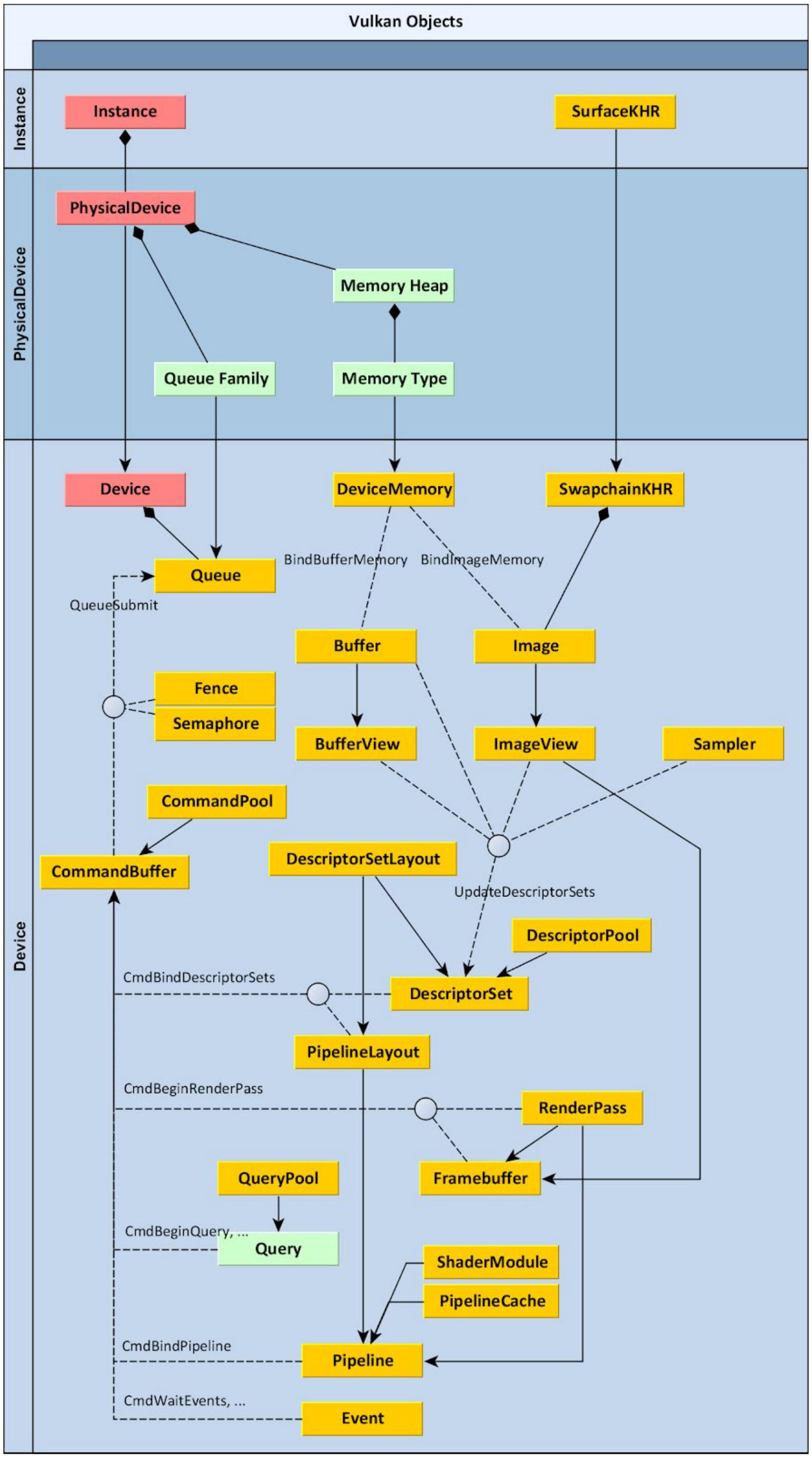
Now we get to the good stuff - the entire process for rendering in Vulkan! This section is going to be very verbose, and talk-in detail about what each object does. For the sake of some brevity, most code will be excluded, and instead references to line numbers in the above repository will be provided. Buckle up - it's a crazy ride.

Also, as a note: the **VULKAN SPECIFICATION GUIDE** has all sorts of information in more detail on many of the commands. If you want more information on specific parameters and commands, reference it!

On the next page is a reference diagram of how everything in Vulkan interacts. I will go more in-depth on each object later, but this just serves to illustrate how complex Vulkan is! There are a ton of moving parts, each with their own crazy hierarchy.

Additionally, if you would like more information on Vulkan objects and hierarchies, view the **VULKAN GLOSSARY** section below.

LASTLY - these are not exact steps that need to be performed in this specific order. For example, the descriptor set layouts don't necessarily need to be the 7th step - they just need to be created at some point before the pipeline, descriptor pool, and descriptor sets.



The Vulkan render process. This image demonstrates the relationship between many Vulkan components.

PRE.) BEFORE RENDERING

Before you do anything, you need to download the Vulkan SDK from the LunarG website using **THIS LINK**. Included in the Vulkan SDK are tools like **SHADERC** (runtime C++ shader compiling), **GLSLC** (for pre-compiling shaders), and many other awesome tools for improving your Vulkan programs. If you intend to use it in a game engine, place it close to your C++ project (probably in an "External" folder, much like other tools used in the CSE 489 game engine); else, just install it in its default location.

Now, let's create a C++ program that actually uses the Vulkan SDK. These steps are in greater detail in the Lab #1 document :

1. Create a new Visual Studio project as a "C++ Windows Desktop Wizard" empty project
2. Include the Vulkan SDK as a dependency / include for the project
    a. For the "include": use the "Include" directory
    b. For the linking: use the "Lib32" for 32-bit programs, and "Lib" for 64-bit programs
    c. It's not necessary to run post-build commands for Vulkan, unlike other dependencies
3. Get all other necessary dependencies and link them in a similar (Bullet, ASSIMP, etc.)
4. Add GLFW as a NuGet package

1.) CREATE A VULKAN INSTANCE

The first Vulkan object you will want to create is a Vulkan **INSTANCE**, which essentially serves as a link between the application and the Vulkan library. Think of it as a big capsule, which holds all the Vulkan devices and commands. This is where you specify information about your application, such as the name and version.

Creating a Vulkan instance uses the general object creation syntax:

1. Create a Vulkan instance object: **VkInstance**
2. Specify info about the application: **VkApplicationInfo**

3. Use this to specify information about the instance: **VkInstanceCreateInfo**
4. Create the instance: **vkCreateInstance**

REFERENCE CODE: ***VULKAN INSTANCE***

2.) (OPTIONAL) CREATE VALIDATION LAYERS

As stated above, Vulkan is designed to have minimal driver overhead. This is great for speeding up programs via less CPU computations, but results in one of the great annoyances of Vulkan: it doesn't have extensive, built-in error checking in the same way something like OpenGL does. However, what it does have instead are something called **VALIDATION LAYERS**. Validation layers are optional attachments to commands, which validate the parameters passed to them before running the command. When it comes to Vulkan debugging, validation layers are extremely helpful (maybe borderline necessary), as they help you figure out crazy memory leaks and C++ errors. However, they require a bit more setup - for a better explanation, reference the **FOLLOWING LINK** from the major Vulkan tutorial.

To create validation layers:

1. Check for validation layer support by enumerating through a list of Vulkan layers and checking to see if they support validation layers: **vkEnumerateInstanceLayerProperties**
2. If validation layers were found, pass them to the **VkInstanceCreateInfo** struct in Instance creation. For example:

```
if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
        createInfo.enabledLayerCount = 0;
}
```

3. Create a message callback / debug messenger *(there's a lot here, just look at the tutorial above)*

REFERENCE CODE: ***VULKAN VALIDATION LAYERS*** *(also see Vulkan Instance code)*

3.) CREATE A VULKAN SURFACE

This is an easy step, but requires you to make a window for your application to use. For the sake of simplicity, I recommend using GLFW (which the CSE 489 game engine already uses). GLFW can easily be acquired as a NuGet package in Visual Studio, and is simple enough to set up. The GLFW window then serves as a **SURFACE** for Vulkan; in other words, the GLFW window becomes the "surface" Vulkan draws to. To put this into a metaphor, imagine the GLFW window as a chalkboard. In order for us to see anything, we have to tell Vulkan "you can draw on this chalkboard" by specifying the GLFW window as its surface. If we don't, then Vulkan won't have anything to draw on.

The steps for creating a Vulkan surface (including creating the window) are as follows

1. Initialize GLFW: **glfwInit()**
2. Set the GLFW window to use "no API". GLFW is meant for OpenGL direct support, so specifying "no API" allows Vulkan to essentially take control of the window without it having to worry about OpenGL formatting: **glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API)**
3. Create the window: **glfwCreateWindow(width, height, title, nullptr, nullptr)**
4. Bind the callbacks: **glfwSetXXXXXCallback(window, callbackMethodName)**

REFERENCE CODE: ***CREATE GLFW WINDOW***

5. Create a Vulkan surface object: **VkSurfaceKHR**
6. Create the window surface: **glfwCreateWindowSurface**

REFERENCE CODE: ***VULKAN SURFACE***

4.) CREATE A VULKAN DEVICE

Our next step is split into two parts: selecting a **PHYSICAL DEVICE**, then using that to create a **LOGICAL DEVICE** for Vulkan to use. A Vulkan physical device is essentially the Vulkan term for a piece of hardware, typically a graphics card. Depending on what features and extensions we want our program to have, we may need to choose between multiple graphics cards. Whichever graphics card we choose is then interfaced to Vulkan via a logical device, which basically serves as a "software construct" for a physical device and its resources. In other words, the logical device is a link between Vulkan and the graphics card.

The steps for creating a Vulkan device are:

1. Create a Vulkan physical device object: **VkPhysicalDevice**
2. Get a list of physical devices: **vkEnumeratePhysicalDevices**
3. From the list, choose a device. Can either be chosen arbitrarily or based off supported extensions / features
    a. May need to create a "isDeviceSuitable" method for checking - see the repo for more information: **vkGetPhysicalDeviceProperties** and **vkGetPhysicalDeviceFeatures**

REFERENCE CODE: ***VULKAN PHYSICAL DEVICE***

4. Create a Vulkan logical device object: **VkDevice**
5. Get information about the queue families: **vkGetPhysicalDeviceQueueFamilyProperties**
    a. The tutorial I used made a method for finding the queue families. More info on how to implement that method **HERE**.
    b. Create a list of VkDeviceQueueCreateInfo structs
    c. For each unique queue family, create a new struct specifying its priority and index: **VkDeviceQueueCreateInfo**
6. Specify info about the logical device to be created: **VkDeviceCreateInfo**
    a. Pass in info about device features, render queues, etc.
7. Create the logical device: **vkCreateDevice**

REFERENCE CODE: ***VULKAN LOGICAL DEVICE***

5.) CREATE A VULKAN SWAP CHAIN AND SWAP CHAIN IMAGES

The next thing we're going to create is a Vulkan **SWAP CHAIN**. A swap chain is essentially a queue of images that are waiting to be presented to the screen, which presents rendered images to the screen synchronized with the screen's refresh rate. The swap chain is the equivalent of the "front" and "back" buffers of Vulkan, but is meant to improve performance by using a chain of already-rendered images, instead of waiting for the back buffer to render.

Two notable things about building swap chains in Vulkan though - first off, swap chains aren't actually built into core Vulkan, as they are heavily tied to the window / surface, and need to be made through an extension in the Vulkan SDK. Thus, many of the commands for the swap chain have the "KHR" extension. Second, not every graphics card supports swap chains or direct image presentation. As a result, we need to check if the graphics card supports swap chains when we are selecting it (see the **isDeviceSuitable** method).

Once we build the swap chain, we have to create a list of Vulkan **IMAGES** for the swap chain to use. In Vulkan, any sort of rendered "picture" (scenes, textures, depth, etc.) needs to have an image object associated with it. This basically allows Vulkan to store information about the image in question. However, to then view the image, we need to create **IMAGE VIEWS**. Images aren't directly accessible by pipeline shaders for reading / writing, so we need to create image views, which are pretty much literally "views" into an image: they describe how to access the image, what parts of the image to access, the image format, etc.

To tie everything together, the swap chain is essentially an object that controls the presentation (or display on a window surface) of images. The images are stored in a list, and are managed by Vulkan. However, to actually use and "see" the images, we need to have an image view for each image.

The steps for creating a Vulkan swap chain and its related components are:

1. Create a Vulkan swap chain object: **VkSwapChainKHR**
2. Get information about the chosen physical device's (graphics card's) swap chain support
    a. The tutorial I used created a method, which took a custom SwapChainSupportDetails struct and populated it with information about the surface capabilities, surface formats, and surface present modes: **vkGetPhysicalDeviceSurfaceCapabilitiesKHR**,

vkGetPhysicalDeviceSurfaceFormatsKHR, and **vkGetPhysicalDeviceSurfacePresentModesKHR**

   i. *REFERENCE CODE:* ***QUERY SWAP CHAIN SUPPORT***

b. The **SwapChainSupportDetails** struct just contains a **VkSurfaceCapabilitiesKHR** object, a list of **VkSurfaceFormatKHR** objects, and a list of **VkPresentModeKHR** objects

   i. *REFERENCE CODE:* ***SWAP CHAIN SUPPORT DETAILS***

3. Use the information acquired from the swap chain support method to set the surface format, present mode, and extent of the application: **VkSurfaceFormatKHR**, **VkPresentModeKHR**, and **VkExtent2D**
    a. **Surface Format**: The color depth. Specifies how we will be storing each color channel and the color space.
    b. **Presentation Mode**: Represents the conditions for showing the image to the screen. This area is kind of hard to describe, so more info **HERE**.
    c. **Extent**: The resolution of the swap chain images. Pretty much always equal to the resolution of the presentation window.
    d. The tutorial I used created separate method for choosing each option, based on certain parameters we want:
        i. *REFERENCE CODE:* ***CHOOSE SURFACE FORMAT***
        ii. *REFERENCE CODE:* ***CHOOSE PRESENTATION MODE***
        iii. *REFERENCE CODE:* ***CHOOSE EXTENT***
4. Specify the size of the swap chain. You'll need to get this from the **SwapChainSupportDetails**.
5. Create a struct for creating the swap chain: **VkSwapchainCreateInfoKHR**
    a. Need to specify the surface, image format, image extent, image usage, present mode, etc. See the code / Vulkan specification docs for more information.
    b. Will need to specify the image sharing mode, based on whether the queue family for the graphics and present queues are the same. Again, see the code.
6. Create the swap chain: **vkCreateSwapchainKHR**

7. Resize the swap chain image list to match the size of the swap chain: **vkGetSwapchainImagesKHR**

REFERENCE CODE: ***VULKAN SWAP CHAIN***

8. Resize the list of swap chain image views to match the size of the swap chain images
9. For each swap chain image, create a swap chain image view
   a. Create an image view create struct: **VkImageViewCreateInfo**
   b. Create the image view: **vkCreateImageView**
   c. **Note:** I made a general method for creating image views. The only real difference between swap chain images and other images is the image itself, the image format, and the image aspect flags - those are passed into the general method. See the general method **HERE**.

REFERENCE CODE: ***VULKAN SWAP CHAIN IMAGE VIEWS***

6.) CREATE A VULKAN RENDER PASS

This is one of the more vague Vulkan objects. In short, a Vulkan **RENDER PASS** is a container object, which stores information that is used during rendering. This is mainly for attaching information about render buffers (color, depth, etc.) to the pipeline, including the number of buffers, their samples, and how their contents should be handled. These are done by creating **ATTACHMENTS**, which are then bound to **SUBPASSES**, which are then attached to the render pass proper. Using a render pass allows Vulkan to group / reorder render operation, which can result in potentially-better rendering performance.

We'll get more to Vulkan render passes when we get to the main render loop. For more information on all the various enums needed, view **HERE**.

The steps for creating a Vulkan render pass are:

1. Create a Vulkan render pass object: **VkSwapChainKHR**
2. For each attachment we want to make, create an attachment struct: **VkAttachmentDescription**
   a. In my case, I only have two: one for color, and one for depth

   b. Need to specify stuff like format, sample rate, load / store operation, etc.

3. For each attachment, create an attachment reference struct: **VkAttachmentReference**
   a. Need to increment the attachment number!
4. Create a subpass description struct, and attach all attachment reference structs: **VkSubpassDescription**
5. Create a subpass dependency struct: **VkSubpassDependency**
   a. Is used in image layout transitions in our project - see **BELOW** for more details on subpass dependencies
6. Create a struct for specifying info about the render pass creation: **VkRenderPassCreateInfo**
   a. Add the attachments, subpasses, and subpass dependencies to this object!
7. Create the render pass: **vkCreateRenderPass**

REFERENCE CODE: ***VULKAN RENDER PASS***

7.) CREATE VULKAN DESCRIPTOR SET LAYOUTS

In Vulkan, any sort of resources that need to be utilized by the shaders need to be bound to **DESCRIPTOR SETS**. However, to make descriptor sets for the shaders to use, we need to first create two objects, the first of which are **DESCRIPTOR SET LAYOUTS**. Descriptor set layouts tell the pipeline how descriptor sets are formatted, including the descriptor type and the binding. Descriptor set layouts are also used as templates when creating descriptor sets.

Just as a note: this process will need to be repeated per descriptor set layout! For my project, I had to make two descriptor set layouts: one for uniform buffer objects, and one for texture samplers.

The steps for creating a Vulkan descriptor set layout are:

1. Create a Vulkan descriptor set layout object: **VkDescriptorSetLayout**
2. Create a Vulkan descriptor set layout binding struct (or list of structs): **VkDescriptorSetLayoutBinding**

   a. Used to specify the type of descriptor (uniform buffer, texture sampler, etc.) and the pipeline stage of use (vertex, fragment, etc.)

3. Create a struct to specify info used for the descriptor set layout creation: **VkDescriptorSetLayoutCreateInfo**
   a. Attach the descriptor set layout binding to this!
4. Create the descriptor set layout: **vkCreateDescriptorSetLayout**

*REFERENCE CODE: **VULKAN DESCRIPTOR SET LAYOUTS***

8.) CREATE A VULKAN DESCRIPTOR POOL

As mentioned above, descriptor sets need two objects to be created before they are allocated. The first, descriptor set layouts, were just created. The second of these are **DESCRIPTOR POOLS**. Descriptor sets cannot be created directly; instead, their memory must be allocated from "memory pools". A descriptor pool is simply a memory pool used for allocating descriptor sets. The architecture of your descriptor pools may depend on your program. For example, if you are making a game engine with dynamic object creation / destruction, you will want multiple descriptor pools to accommodate this (descriptor pools are limited in size and will throw errors if they run out of memory).

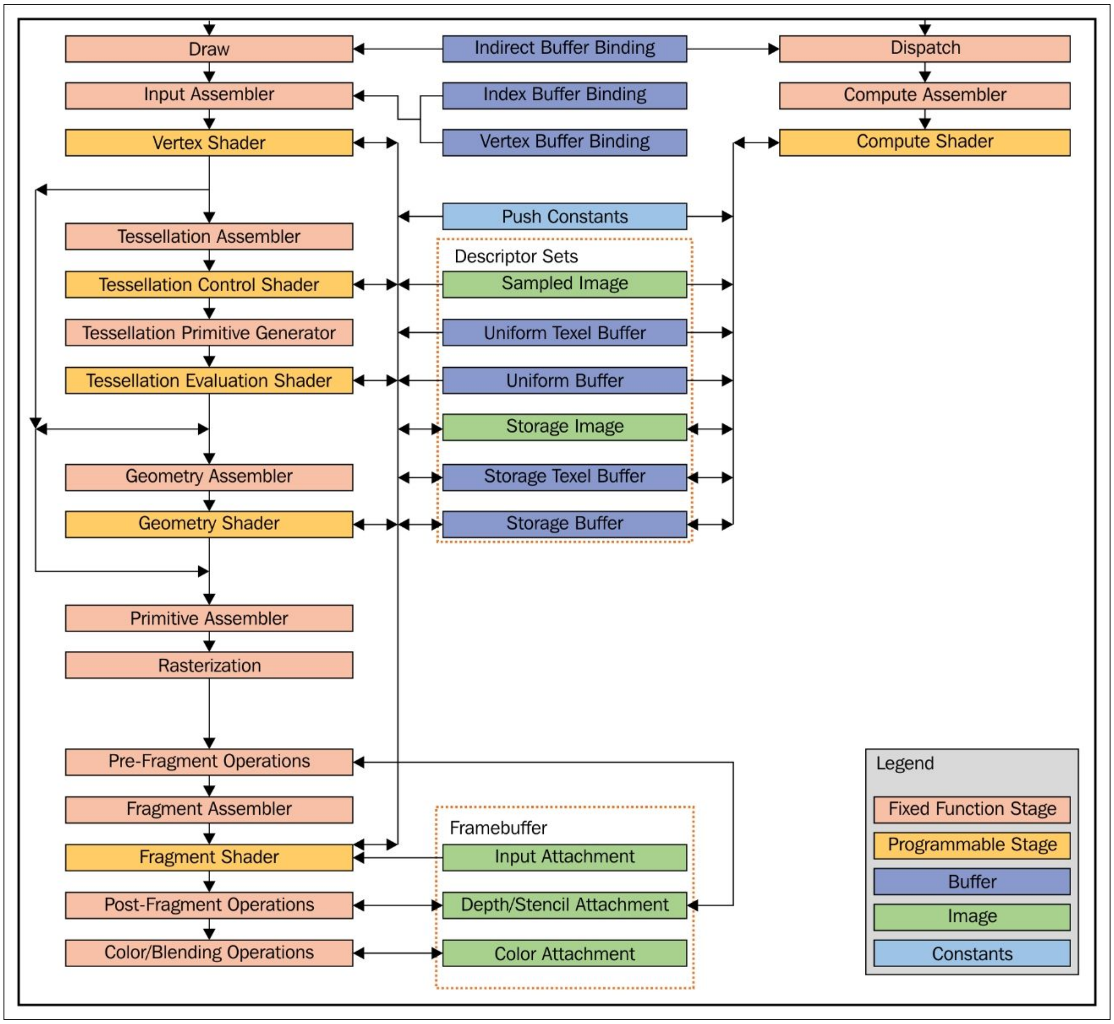
The steps for creating a Vulkan descriptor pool are:

1. Create a Vulkan descriptor pool object: **VkDescriptorPool**
2. Create a descriptor pool size struct / list of descriptor pool size structs: **VkDescriptorPoolSize**
   a. Has parameters for the descriptor pool type and descriptor count
3. Create a struct for specifying info about the descriptor pool's creation: **VkDescriptorPoolCreateInfo**
   a. Attach the descriptor pool size(s) above
4. Create the descriptor pool: **vkCreateDescriptorPool**

*REFERENCE CODE: **VULKAN DESCRIPTOR POOLS***

9.) CREATE A VULKAN GRAPHICS PIPELINE

This step is perhaps the most important step in this entire process, but also the most verbose. Like in OpenGL, a **GRAPHICS PIPELINE** is simply a set of steps and transformations used to render objects in a scene. However, unlike OpenGL, the pipeline is an object that has to be specifically created in Vulkan. This requires lots of different components, but allows for greater flexibility in the graphics pipeline itself. Below is a low-level overview of of the graphics pipeline steps in Vulkan:



A low-level overview of the Vulkan graphics pipeline.

When I say that the Vulkan graphics pipeline is more flexible, this is both true and untrue. Like OpenGL, it has MANY fixed stages, with the "programmable" stages being done in shader modules. I won't go into specifics about what each shader stage does (see **HERE** for that info), but will talk more about the various components that can be attached to a Vulkan pipeline.

The steps for creating a Vulkan graphics pipeline are as follows:

1. Create a Vulkan graphics pipeline layout object: **VkPipelineLayout**
2. Create a Vulkan graphics pipeline object: **VkPipeline**
3. For each shader, read in the code and create a shader module: **VkShaderModule**
    a. A shader module is basically how shaders are attached to the pipeline - more info in the **GLOSSARY**
    b. Create a struct for specifying creation info about the shader module: **VkShaderModuleCreateInfo**
        i. Pass the code as a reference in the shader module
    c. Create the shader module: **vkCreateShaderModule**
4. For each shader module, create a struct for creating the pipeline stage: **VkPipelineShaderStageCreateInfo**
    a. Need to specify the stage type and pass in the shader module!
5. Specify information about the vertices should be input to the pipeline: **VkPipelineVertexInputStateCreateInfo**
    a. Requires getting the binding (size and input rate) and attribute () descriptions of the vertex object - need to create methods in the vertex for acquiring those!
        i. Binding needs to be a **VkVertexInputBindingDescription** struct
        ii. Attribute descriptions need to be a list of **VkVertexInputAttributeDescription** structs, which specify the binding, location, format, and offset of each attribute of a vertex (such as position, normal, and texture coordinate)
6. Create an input assembly pipeline stage (the construction of triangles / lines / etc.): **VkPipelineInputAssemblyStateCreateInfo**

7. Create a viewport object and specify its position, width, height, and depth: **VkViewport**
8. Create a scissor rectangle (for getting a subsection of a viewport): **VkRect2D**
9. Combine the viewport and scissor rectangle into a viewport pipeline stage: **VkPipelineViewportStateCreateInfo**
10. Create a rasterizer for the pipeline: **VkPipelineRasterizationStateCreateInfo**
    a. Here is where we specify the **polygon mode** (wireframe, fill, etc.), the **cull mode**, the **front face**, etc.
11. Create a multi-sampling pipeline stage: **VkPipelineMultisampleStateCreateInfo**
    a. NOTE: I didn't actually get the chance to implement multi-sampling (for anti-aliasing) in my project. The tutorial for implementing that can be found **HERE**.
12. Create a depth and stencil testing pipeline stage: **VkPipelineDepthStencilStateCreateInfo**
    a. Has instructions for how to perform depth and stencil testing
13. Create a color blending attachment: **VkPipelineColorBlendAttachmentState**
    a. Specifies how colors should be blended!
14. Create a color blending pipeline stage: **VkPipelineColorBlendStateCreateInfo**
    a. Uses the color blend attachment from the previous state, as well as blend constants
15. Create a struct used in creating the pipeline layout: **VkPipelineLayoutCreateInfo**
    a. Requires the descriptor set layouts from earlier as attachments!
16. Create the pipeline layout: **vkCreatePipelineLayout**
17. Bind ALL the above steps to a struct for creating a graphics pipeline: **VkGraphicsPipelineCreateInfo**
    a. Requires pointers to shader stages, vertex input state, input assembly state, viewport state, rasterization state, and other misc. stages (multisampling, depth / stencil testing, color blending, etc.), as well as the pipeline layout and render pass from earlier!
18. FINALLY, create the graphics pipeline: **vkCreateGraphicsPipelines**

19. Since we no longer need the shader modules, destroy them:
    vkDestroyShaderModule

REFERENCE CODE: ***VULKAN GRAPHICS PIPELINE***

10.) CREATE VULKAN DEPTH RESOURCES

One of the annoyances with Vulkan is that depth testing isn't built directly into the pipeline. Thus, we need to create a **DEPTH BUFFER** to manage depth testing of multiple vertices. A depth buffer is an attachment that stores the depth of every position, then performs depth testing every time the rasterizer produces a new fragment (exactly how color attachments work). We already made the render pass step for depth buffering, but need to make some additional components. The main things we need to create are a single **DEPTH IMAGE**, which, like most other images, requires an **IMAGE VIEW**. However, since we want to have this object in permanent storage (rather than just presenting it to the window, like we do with the swap chain images), we need to create **DEVICE MEMORY** for the depth image. Only a single depth image is need, since only one draw operation will be running at once.

The steps for creating the Vulkan depth resources are as follows:

1. Create a Vulkan depth image: **VkImage**
2. Create Vulkan device memory for the depth image: **VkDeviceMemory**
3. Create a Vulkan image view for the depth image: **VkImageView**
4. Select an appropriate depth format for the image
    a. The tutorial I used created a method for finding the best depth format available. For more information on that method, view **HERE**.
5. Use the VkFormat we just selected to create the Vulkan depth image
    a. This is another example of a general method I made to reduce the amount of code; here, I made a general createImage method for creating any type of Vulkan image. See the **CREATEIMAGE** method for how Vulkan image creation works. Additionally, see the **OTHER CONSIDERATIONS** section for my explanation of image creation.

   b. The main difference between this image and other Vulkan images is its formatting: it uses a depth **VkFormat,** indicated by **VK_FORMAT_D#** instead of **VK_FORMAT_R#G#B#A#** or variations
6. Use the image view creation method from before to create the depth image view
7. Transition the image layout to be depth / stencil attachment-optimal, so it can be used as a render pass attachment
   a. This is one of the more confusing areas of the code - transitioning image layouts. More on that in the **OTHER CONSIDERATIONS** section.
8. We will use the objects we just made when creating framebuffers - don't worry about not using them just yet.

REFERENCE CODE: ***VULKAN DEPTH BUFFERING***

11.) CREATE VULKAN FRAMEBUFFERS

Framebuffers are also a bit of a weird concept in Vulkan. According to a definition I found online, **FRAMEBUFFERS** "represent a collection of memory attachments that are used by a render pass instance. Examples of these memory attachments include color buffers and depth buffers. A framebuffer provides the attachments that a render pass needs while rendering." In other words, a framebuffer is a storage object, which contains the buffers needed in render pass operations.

For our application, we will be creating a framebuffer for each swap chain image. To create a Vulkan framebuffer...

1. Create a Vulkan framebuffer (or list of Vulkan framebuffers): **VkFramebuffer**
   a. Resize the framebuffer list to be equal to the swap chain image view list size!
2. Create a list of Vulkan image views to serve as our attachments
   a. In our case, this should be the swap chain image view corresponding to the current framebuffer, as well as the depth image view
3. Create a struct for specifying frame buffer creation info: **VkFramebufferCreateInfo**
   a. Need to add the list of images views as an attachment, as well as pass in the render pass and the height / width of the application

4. Create the framebuffers: **vkCreateFramebuffer**

REFERENCE CODE: ***VULKAN FRAMEBUFFERS***

12.) CREATE A VULKAN COMMAND POOL

The next thing we need to create is something similar to the descriptor pool from earlier: a **COMMAND POOL**. Command pools manage the memory used to store buffers and command buffers allocated from them. It's exactly like descriptor pools from earlier: we create one or more pools, then allocate memory from those pools to create command buffers (more on that later).

In some cases, it may be wise to have multiple command pools. More on that below in the **EXTRA CONSIDERATIONS** section.

The process for creating a Vulkan command pool is as follows:

1. Create a Vulkan command pool: **VkCommandPool**
2. Retrieve the indices of the queue families
    a. See the more about the method in the **DEVICE** section!
3. Create a struct for holding command pool creation information: **VkCommandPoolCreateInfo**
4. Create the command pool: **vkCreateCommandPool**

REFERENCE CODE: ***VULKAN COMMAND POOL***

13.) CREATE A VULKAN COMMAND BUFFER

Now we get to one of the more important Vulkan objects: the **COMMAND BUFFER**. The command buffer is a memory region, which stores the drawing commands we will be submitting to the drawing (rendering) for execution. The command buffer will use many of the objects we made before, including the pipeline, render pass, and framebuffers.

We'll touch on this more in the **EXTRA CONSIDERATIONS** section, but there are two methods for recording to command buffers: pre-recording, and runtime-recording. For something like a game engine, where the number of objects are constantly shifting, we will need to do runtime-recording. This is one of the areas I ran into issues - the main Vulkan tutorial I found used pre-recorded command buffers. Thus, I had to find an alternate tutorial for these steps.

Additionally, we will also talk about the number of command buffers we should use in the same section. Many different sources argue to use either one command buffer overall, or one command buffer per swap chain image.

The process for creating a Vulkan command pool is as follows:

1. Create a Vulkan command buffer: **VkCommandBuffer**
2. Create a struct for holding command buffer allocation information: **VkCommandBufferAllocateInfo**
    a. Need to specify which command pool to allocate from!
3. Allocate the command buffer: **vkAllocateCommandBuffers**

REFERENCE CODE: ***VULKAN COMMAND BUFFER***

14.) CREATE VULKAN SYNC OBJECTS

“Sync objects” is a bit of a shorthand here for synchronization tools for multithreading in Vulkan. Part of Vulkan’s major efficiency is that it relies on synchronization tools, such as **SEMAPHORES** and **FENCES**. These help to signal when certain queues are available to be submitted to for rendering, presentation, etc.

According to the Vulkan tutorial, “The difference [between fences and semaphores] is that the state of fences can be accessed from your program using calls like vkWaitForFences and semaphores cannot be. Fences are mainly designed to synchronize your application itself with rendering operation, whereas semaphores are used to synchronize operations within or across command queues. We want to synchronize the queue operations of draw commands and presentation, which makes semaphores the best fit.”

Like command buffers, this is another area where the number of fences and semaphores is a little vague for building a general architecture. More on that in the **EXTRA CONSIDERATIONS** section.

The process for creating a Vulkan sync objects as follows:

1. Create a semaphore (or list of semaphores) for signaling when a new image is available from the swap chain: **VkSemaphore**
2. Create a semaphore (or list of semaphores) for signaling when a render is finished: **VkSemaphore**
3. Create a fence (or list of fences) for waiting on in-flight (currently-waiting) images: **VkFence**
4. Create a struct for specifying semaphore create info: **VkSemaphoreCreateInfo**
5. Create a struct for specifying fence create info: **VkFenceCreateInfo**
6. Create the fences and semaphores: **vkCreateSemaphore** and **vkCreateFence**

REFERENCE CODE: ***VULKAN SYNC OBJECTS***

15.) LOADING YOUR GAME OBJECTS AND COMPONENTS

Here's where stuff starts getting interesting. Now, we're going to use the loadScene method to load in game objects and components. When we load these items in, we need to initialize them and create buffer space for their properties.

In the case of game objects, we need to allocate buffer memory (see more on the process of this in **MORE CONSIDERATIONS**) for a uniform buffer object, which controls our modeling, viewing, and projection matrices. In our code, we do this through a custom **GameObjectUniformBufferObject** struct, which holds these members as glm::mat4 objects. After this, we then have to bind this uniform buffer object to a descriptor set (again, more **BELOW**) and update the descriptor set, so that way it can be used in the shaders.

The exact code for this goes something like:

1. Create a Vulkan buffer for the uniform buffer object: **VkBuffer**

2. Create Vulkan device memory for the uniform buffer object: **VkDeviceMemory**
3. Create a Vulkan descriptor set for the uniform buffer object: **VkDescriptorSet**
4. Allocate descriptor set memory for the uniform buffer object's descriptor set
5. Create two structs for use in updating the descriptor set: a **VkDescriptorBufferInfo** struct and a **VkWriteDescriptorSet** structs / list of **VkWriteDescriptorSet** structs
6. Update the descriptor set: **vkUpdateDescriptorSets**
7. Initialize all child game objects and components

*REFERENCE CODE: **GAME OBJECT INITIALIZATION***

Most components don't need any additional setup; they can simply be attached. However, mesh components are a different story. Like game objects, they have a descriptor set which needs to be bound and updated by Vulkan for use in the shaders. However, while you may use a uniform buffer object in the case of getting color properties (more **BELOW**). However, if you want to load in textures of any kind to use as a material, you will need to create a texture sampler, which is a separate type of descriptor set Vulkan can use. This requires using a program like **FREEIMAGE** or **STBIMAGE** to load in the image, buffer its memory, then bind it to a descriptor set.

The exact code for this goes something like:

1. Create a Vulkan image object: **VkImage**
2. Create Vulkan device memory for the image: **VkDeviceMemory**
3. Create a Vulkan descriptor set for each texture image to load: **VkDescriptorSet**
4. Use your image loader of choice to load the image and get its size
5. Create a temporary staging buffer (**VkBuffer** and **VkDeviceMemory**) to store the image
    a. This uses the general buffer creation method, found **BELOW**.
    b. **Why Use Staging Buffer?**: According to the internet, if loading the vertices into CPU memory only, "the memory type that allows us to access it from the CPU may not be the most optimal memory type for the graphics card itself to read from. The most optimal memory has the

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT flag and is usually not accessible by the CPU on dedicated graphics cards. In this chapter we're going to create two vertex buffers. One staging buffer in CPU accessible memory to upload the data from the vertex array to, and the final vertex buffer in device local memory. We'll then use a buffer copy command to move the data from the staging buffer to the actual vertex buffer."

6. Map the image to device memory: **vkMapMemory**, **memcpy**, **vkUnmapMemory**
    a. This is a pretty common method of binding data.
7. Unload the image file
8. Create a Vulkan image using the information acquired from loading in the image
    a. We're going to be using the general image creation method for this: more info **BELOW**.
9. Transfer the image layout to be more optimal for transfer-destination
    a. More information about transitioning image layouts **BELOW**.
10. Copy the image from the staging buffer to the actual image buffer
    a. This is done through a custom **copyBufferToImage** method!
11. Transition the image layout to be shader read-only optimal
12. Create a texture image view for the new texture image
    a. Create an image view create struct: **VkImageViewCreateInfo**
    b. Create the image view: **vkCreateImageView**
    c. **Note:** This uses the general image view struct I made before. See the general method **HERE**.

REFERENCE CODE: ***TEXTURE LOADING***

Additionally, meshes requires you to load in vertex / index data (either hand-made or from a 3D model file). Doing so requires creating a staging buffer, loading the data to said staging buffer, then copying the staging buffer memory to CPU-accessible memory, then finally moving said data to the actual vertex / index buffer and its associated memory. Vertex and index buffering are very similar, but need distinct buffers and memory regions for operation. For this section, I used the submesh struct, and loaded in vertices / indices per submesh.

The exact code for this (per submesh) goes something like:

1. Create separate Vulkan buffers for the vertices and indices: **VkBuffer**
2. Create separate Vulkan device memory objects for the vertices and indices: **VkDeviceMemory**
3. Create a temporary staging buffer (**VkBuffer** and **VkDeviceMemory**) to store the vertex / index data
4. Map the image to device memory: **vkMapMemory**, **memcpy**, **vkUnmapMemory**
5. Create the actual index / vertex storage buffer using the buffer object created above
    a. This uses the general buffer creation method, found **BELOW**.
6. Copy the data from the staging buffer to the actual buffer
7. Destroy the staging buffer and free its associated memory: **vkDestroyBuffer** and **vkFreeMemory**

*REFERENCE CODE: **BUILDING SUBESHES (VERTEX)***
BUILDING SUBESHES (INDEX)

16.) THE MAIN RENDER LOOP

Now here's where stuff gets fun - we finally get to run the game! Again for the sake of simplicity, stuff like updating the sound and physics engines will be left out. The main focus here will be the Vulkan-related components: starting the render loop, updating objects, and ending the render loop. As mentioned above, Vulkan command buffers can either be pre-recorded or dynamically recorded. Since the latter is more suitable for game engines, that's what we'll be doing

Beginning the render loop:

1. Wait for the fence for the next frame to become free: **vkWaitForFences**
2. Acquire the next image from the swap chain: **vkAcquireNextImageKHR**
3. If a previous frame is using this image, wait for its associated fence to free up: **vkWaitForFences**
4. Create a struct to begin recording to the command buffer: **VkCommandBufferBeginInfo**

5. Begin recording to the command buffer: **vkBeginCommandBuffer**
6. Create a struct to begin the render pass: **VkRenderPassBeginInfo**
    a. Need to pass in the render pass object, swap chain extent, frame buffer, and clear values (clear color and depth)
7. Begin using the render pass in the command buffer: **vkCmdBeginRenderPass**
8. Bind the graphics pipeline to the command buffer: **vkCmdBindPipeline**

*REFERENCE CODE: **BEGIN OF RENDER LOOP***

Per camera:

9. Set the viewing and projection matrices somewhere so the game objects can use them
    a. Calculations for these are pretty straightforward - see the code for the process
    b. Just a reminder that the current game engine also doesn't support multi-camera layouts!
    c. One thing to note here too is that we need to flip the Y-coordinate of the clip coordinates. This is because of some GLM weirdness - according to the Internet, "GLM was originally designed for OpenGL, where the Y coordinate of the clip coordinates is inverted. The easiest way to compensate for that is to flip the sign on the scaling factor of the Y axis in the projection matrix. If you don't do this, then the image will be rendered upside down."

*REFERENCE CODE: **SETTING CAMERA TRANSFORMATIONS***

Per mesh component (or submesh):

10. If we have vertices in our mesh, then bind them: **vkCmdBindVertexBuffers**
11. If we have indices in our mesh, then bind them: **vkCmdBindIndexBuffer**
12. If this mesh has a material, bind the descriptor sets for it:
    a. We need to update and bind the descriptor set for each texture / color material per frame! For uniform buffers, we don't need to do this!
    b. To update, we create two structs (a **VkDescriptorImageInfo** struct and a **VkWriteDescriptorSet** structs / list of **VkWriteDescriptorSet** structs), the pass those into the update method: **vkUpdateDescriptorSets**

   c. To bind, we simply call the method to bind the descriptor sets: **vkCmdBindDescriptorSets**

REFERENCE CODE: ***BINDING TEXTURE DESCRIPTOR SETS***

13. If we are using ordered rendering, draw the mesh using the vertices: **vkCmdDraw**
14. Else, we are using indexed rendering, so draw the mesh using indices: **vkCmdDrawIndexed**

REFERENCE CODE: ***RENDERING MESH***

Per game object:

15. Update the modeling transformation, then create a Vulkan uniform buffer object and set the modeling, viewing, and projection transformation matrices
    a. In our code, we do this through a custom **GameObjectUniformBufferObject** struct. This is what we created the buffer for, and what is bound to the descriptor sets for use in the shader. This struct stores the modeling, viewing, and projection matrices.
16. Copy the data from the struct to the uniform buffer object to device memory: **vkMapMemory**, **memcpy**, **vkUnmapMemory**
17. Bind the uniform descriptor set: **vkCmdBindDescriptorSets**
18. Update all child game objects

REFERENCE CODE: ***UPDATING GAME OBJECT***

Ending the render loop:

19. End the usage of the render pass: **vkCmdEndRenderPass**
20. End recording to the command buffer: **vkEndCommandBuffer**
21. Create a struct for information to submit to the graphics (render) queue: **VkSubmitInfo**
    a. Include the semaphore that we should wait on to submit, the semaphore to signal when this queue is open, and which command buffers to submit to
22. Reset the fence at the current frame, allowing another image to be rendered to: **vkResetFences**

23. Create a struct for information to submit to the presentation queue: **VkPresentInfoKHR**
    a. Include the semaphore to wait on and the swap chain
24. Send the image to the present queue: **vkQueuePresentKHR**

REFERENCE CODE: ***END OF RENDER LOOP***

Essentially, what we just did is start recording to the command buffer, bound the render pass and pipeline to the buffer, then inserted a ton of commands to update uniform buffers, descriptor sets, and render meshes. Just rinse and repeat as long as the render loop should run!

16.) CLEANING UP

In Vulkan, virtually every object that is created needs to be destroyed or deallocated once it is no longer in use. That means once the render loop is exited, we need to remove all our Vulkan-related objects to prevent memory-related leaks and errors. For the sake of simplicity, I'm not going to write every destroy / deallocation command since there are a lot (just look at the reference code); however, I will give general tips for the usage of cleanup commands.

First, a good tip with cleanup commands is to destroy objects in the opposite order that you created them. For example, our Vulkan instance was the first object we created, so it should be destroyed last to prevent it from causing errors on dependencies. Additionally, some objects don't need to be destroyed, and are destroyed implicitly by destroying their owning object. This mainly applies to pools, such as command pools and descriptor pools - destroying them implicitly destroys buffers made from the pools.

REFERENCE CODE: ***VULKAN CLEANUP***

OTHER CONSIDERATIONS AND NOTES

This section is reserved for any other considerations about Vulkan that haven't come up yet at this point.

ALLOCATING DESCRIPTOR SETS

Allocating descriptor sets is a bit of an interesting pickle, in my experience. Descriptor sets, as previously mentioned, are allocated from descriptor pools. However, descriptor pools are limited in size, and as such, attempting to allocate from a descriptor pool with no more room results in "out of memory errors". Thus, we need to dynamically create descriptor pools for whenever we add / delete new descriptor sets. I made a global method in the Vulkan static class, which allocates a given descriptor set from a pool, if memory exists. Else, it creates a new pool.

This method works as follows:

1. For each descriptor pool, attempt to allocate a descriptor set
    a. Create a struct for specifying descriptor set allocation information: **VkDescriptorSetAllocateInfo**
        i. Requires information about the pool, descriptor set layout, etc.
    b. Attempt to allocate the descriptor set: **vkAllocateDescriptorSets**
2. If no descriptor pool has enough memory, **CREATE A NEW ONE** and repeat the above steps using the new command pool

REFERENCE CODE: ***ALLOCATING DESCRIPTOR SETS***

RE-CREATING THE SWAP CHAIN

Unfortunately, one of the annoyances with Vulkan is that some simple operations, such as window resize, aren't as easy. If we resize the window and therefore the surface, then the swap chain will no longer be compatible with the window (due to different extents) and will throw an error. Thus, in circumstances where the swap chain becomes out of date or incompatible (such as window resize), we need to re-create the swap chain and all its related components, including image views, the render pass, the graphics pipeline, the framebuffers, and the command buffers.

The process for this is pretty straightforward: in our render loop, we simply have checks when we acquire the next image in the swap chain / present a rendered image that, if the swap chain

is out of date, call a method that destroys all swap chain components and re-builds them to match the new specifications of the window. However, this also means that each object has to have its descriptor set re-allocated and re-bound for the new swap chain and pipeline to use them.
REFERENCE CODE: ***RECREATING SWAP CHAIN***

CREATING IMAGES

As mentioned above, I create a general method for creating Vulkan images. Since this needed to be used for depth images, textures, and swap chain images, I figured it would be a good general method to have.

The process for creating a Vulkan image is as follows:

1. You should already have a **VkImage** object and **VkDeviceMemory** to copy to
2. Create a struct for specifying image creation info: **VkImageCreateInfo**
    a. Requires information, including the image type, image extent (width and height), format, tiling, usage, and samples. To differentiate between image types, many of these items are passed in via reference!
3. Create the Vulkan image: **vkCreateImage**
4. Get the memory requirements for the image: **vkGetImageMemoryRequirements**
5. Create a struct for allocating the image memory: **VkMemoryAllocateInfo**
6. Allocate the Vulkan image memory: **vkAllocateMemory**
7. Bind the Vulkan image memory: **vkBindImageMemory**

REFERENCE CODE: ***CREATING AN IMAGE***

TRANSITIONING IMAGE LAYOUTS

Buckle up - this method is a doozy. In Vulkan, when creating images, it's important to make sure they're in the correct image layout (**VkFImageLayout**). However, the image layout we import the image in isn't always the best for storing the image. As such, we need to transfer the layout of the image to be optimal for its intended purpose. Some of these possible image layouts include:

- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL**: This makes the image optimal as a depth image attachment to a pipeline
- **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**: This makes images optimized for transferring to a destination buffer
- **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL**: This makes the image optimized for shaders as a read-only image

To transition between these layouts takes a lot of work though, and makes use of some weird custom methods, which we will discuss below.

The process for transitioning a Vulkan image between various formats is as follows:

1. Create a single-time command buffer: **VkCommandBuffer**
   a. **Why use a single-time command buffer?**: Memory transfer operations are submitted using command buffers. However, since we only need them for one command at a time, it only needs to be temporary.
   b. Create a struct for storing command buffer allocation information: **VkCommandBufferAllocateInfo**
   c. Allocate the command buffer: **vkAllocateCommandBuffers**
   d. Create a struct of information for starting the command buffer: **vkBeginCommandBuffer**
2. Create a Vulkan image memory barrier struct: **VkImageMemoryBarrier**
   a. This takes in the image, old layout, new layout, and a couple other parameters
3. Create pipeline stage flags for the source and destination stages: **VkPipelineStageFlags**
4. Here's where stuff gets weird - we have to set various parameters (usually the **srcAccessMask** and **dstAccessMask** parameters, but sometimes the **subresourceRange.aspectMask** instead; also the **sourceStage** and destinationStage), depending on what the old and new layout are. See the code for more reference on these
5. Record the image transition to the pipeline barrier: **vkCmdPipelineBarrier**
6. Stop recording to the single-time command buffer: **vkEndCommandBuffer**

7. Create a struct for handling submission to the graphics queue: **VkSubmitInfo**
8. Submit the commands to the graphics queue and wait for them to complete: **vkQueueSubmit** and **vkQueueWaitIdle**
9. Free the single-time command buffer: **VkFreeCommandBuffer**

REFERENCE CODE: ***TRANSITIONING IMAGE LAYOUTS***

CREATING BUFFERS

As mentioned above, a general method was created for allocating buffers. A lot of different Vulkan objects use buffers (**VkBuffer**), so a general method for doing this process helps to make the code slightly less complicated.

The process for creating a Vulkan buffer is as follows:

1. A Vulkan command buffer object should have already been created: **VkBuffer**
2. Create a struct for specifying command buffer creation info: **VkBufferCreateInfo**
    a. Need to pass in the size and usage of the buffer!
3. Create the buffer: **vkCreateBuffer**
4. Get the memory requirements for the buffer: **vkGetBufferMemoryRequirements**
5. Create a struct for allocating buffer memory: **VkMemoryAllocateInfo**
6. Allocate the buffer memory: **vkAllocateMemory**
7. Bind the buffer memory: **vkBindBufferMemory**

REFERENCE CODE: ***TRANSITIONING IMAGE LAYOUTS***

SHADERS

As mentioned above, Vulkan shaders are a bit of a complex bunch. While Vulkan shaders can be written in languages like GLSL (OpenGL) and HLSL (Direct3D), they cannot be read by Vulkan in these formats. Instead, they need to be compiled into a bytecode format called **SPIR-V** before program execution. This is done for two reasons: 1.) it means the shaders don't have to be compiled during execution, which can speed up the program initialization time, and 2.) bytecode formats are easier to convert to native code than human-readable formats.

However, this means that any changes made to shaders won't be read by Vulkan unless the shaders are compiled. Luckily, Vulkan provides tools for compiling GLSL and HSLS shaders to SPIR-V. The most important of these is a little tool called **GLSLC**, which compiles GLSL shaders to SPIR-V. It operates on the command there are tools that allow for runtime shader compilation, and follows the general syntax of:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe {shader source} -o {output shader}.spv
```

For example, if you wanted to compile a vertex shader, you would use;

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
```

Note that these commands are for Windows; Linux / Mac simply requires "**/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc**" as the path.

Additionally, there are also tools in the Vulkan SDK for compiling shaders at runtime, such as **SHADERC**. However, after many long hours of attempts, I wasn't able to get shaderc working, so this might not be the best method. The shaders would have to be re-compiled every time the graphics pipeline is updated (such as window resize), so it may not be as efficient.

More information about shaders and shader modules can be found **HERE**. This specific page recommends creating a batch script for compiling the shaders, so that way if the shaders are updated, the only thing that needs to be done is run the script. I would also recommend this; it's much easier than having to mess around with shaderc.

REMOVED FEATURES AND POSSIBLE REIMPLEMENTATION METHODS

One of the annoying things about still not fully understanding Vulkan is that certain features in the game engine, such as multiple cameras, shared uniforms (colors, lights, etc.) and some other small features had to be stripped, in an attempt to get a minimally-viable product working. In this section, I want to discuss a few items that had to be removed from the game engine, and suggest possible ways to re-implement said features.

Multiple Cameras

The first I want to discuss is having multiple cameras in the scene. Currently, the architecture only supports one camera, due to viewport issues. In OpenGL, the viewport per camera can easily be set via glViewport. However, in Vulkan, the viewport needs to be stored in the graphics pipeline. As a result, it's much more for multiple viewports to exist in Vulkan. My initial idea was to create a graphics pipeline per camera in Vulkan; however, I definitely think that would be overkill and too much extra work per camera.

However, when I was digging around in the pipeline construction method, I found another potential solution - multiple viewports. The pipeline, in its viewport state (**VkPipelineViewportStateCreateInfo**) can accept multiple viewports and scissor rectangles. Thus, my current idea is to make a static list (array, vector, etc.) of **VkViewport** objects in the Vulkan renderer class that are used in the pipeline when it is built. Each camera could have its own separate VkViewport object, which is then passed into this list as a reference upon camera creation. When the pipeline is being built, it could simply pass this list ito the VkPipelineViewportStateCreateInfo struct's pViewports member. The only major consideration would be having to create the cameras before the pipeline, and rebuilding the pipeline whenever a viewport changes.

Lights / Colors / Shared Uniforms

The other major feature remove was shared uniforms, which includes lights and ambient / diffuse / specular colors. I believe the solution for this is easy - simply use uniform buffers to create these objects. While I'm not totally sure on how these could be implemented, I definitely feel that they wouldn't be super difficult to do. The uniform buffering for the game object's transformation matrices, or the texture sampler for the diffuse / specular textures would be a good place to start in figuring out how this might work. There also might be a way to just pass glm::vec4s to shaders, like there is in OpenGL - I'm not totally sure.

Mipmaps and Multisampling

One of the easiest features in OpenGL is generating mipmaps, which unfortunately is not as easy in Vulkan. The Vulkan Tutorial website has tutorials for both generating mipmaps and multisampling, both of which help to improve the look of meshes by a significant amount. However, due to time constraints, I wasn't able to complete these tutorials. I would definitely recommend looking at the **MIPMAPPING** and **MULTISAMPLING** tutorials in Vulkan, as they will help to improve the look of the game engine drastically.

HOW MANY COMMAND BUFFERS / COMMAND POOLS / SYNC OBJECTS?

This is one of the areas where my knowledge of Vulkan starts to fail me a little. One of the main challenges in the Vulkan is figuring out how many things are needed for rendering. This is further hampered by the fact that many different tutorials recommend different things. For example, should you have one overall command buffer, or one command buffer PER swap chain image? How many command pools should you have? How many fences / semaphores?

The Vulkan Tutorial (see **REFERENCES**), as previously mentioned, used pre-allocated command buffers. It one command buffer PER swap chain image, and swapped between them per frame. Additionally, it had two semaphores and one fence PER swap chain image, The other major tutorial I used, Vulkan API Tutorials (see **REFERENCES**), had dynamically-allocated command buffers, but only had a single command buffer for every frame, and two semaphores / one fence overall. I experimented with both architectures, but wasn't able to get the multi-command buffer setup working. However, due to Vulkan's multithreading support, I'm lead to believe that this method would be more efficient: having multiple command buffers (one per swap chain image) would help to ensure speediness.

Additionally, both tutorials only had one command pool, which managed the command buffer(s) in the tutorials. However, many Vulkan experts say you should have three command pools "per thread" (not sure what this means? I know multithreading but?), which would allow you to switch between them when submitting commands, rendering, and presenting. They also recommend never destroying a command pool, and instead simply resetting it (**vkResetCommandPool**) implicitly frees any associated buffers, which is more efficient than destroying and re-creating the command pool (side note - this also supposedly works for descriptor pools, but I don't know how efficient they are).

So what's the takeaway from all this? In short, the answer is: I don't know what the best architecture is. An answer for this would require someone smarter than me at Vulkan to answer properly. I think experimenting with these various layouts may be worth it, to see which works best while being the least complicated to code and maintain.

VULKAN GLOSSARY

- **INSTANCE**: The topmost-level Vulkan object, which stores all the information about the running Vulkan program. It essentially serves as a link between the application and the Vulkan library.
- **PHYSICAL DEVICE**: An object that represents a hardware component in the computer. Typically, this is an alias for a graphics card.
- **LOGICAL DEVICE**: An "interface" between the Vulkan API and a selected physical device, typically a graphics card. This object serves as the parent object of most Vulkan objects.
- **VALIDATION LAYERS**: Built-in Vulkan debugging "tools" that help to detect errors in Vulkan implementations.
- **QUEUE**: Quite literally like a queue data structure. Commands buffers are submitted to a queue, which then takes the commands in the command buffers and executes them. Queues belong to queue families, which dictate which type of operations the queue can run.
- **QUEUE FAMILY**: A set of queues that have common properties and support the same general functionality.
- **DESCRIPTOR SET LAYOUT**: An object that defines the set of resources within a descriptor set layout. It basically serves as a "template" for defining how a descriptor set will be allocated.

- **DESCRIPTOR SET**: An object stores information meant for use in shaders. Descriptor sets are bound to command buffers, which provide the shaders with information in the descriptor sets.
- **DESCRIPTOR POOL**: An object that manages memory for descriptors sets, and serves as the "owner" of said descriptor sets and their memory. According to the specification guide, "Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use."
- **WINDOW SURFACE**: A window which Vulkan can "draw" to. To draw, Vulkan needs a surface (think of it like a "canvas"), which manifests in the form of a window.
- **SWAP CHAIN**: A queue of images waiting to be presented to the screen. Swap chains are similar to front and back buffers in OpenGL, but are optimized for greater multithreading support.
- **IMAGE**: A Vulkan object, which quite literally represents an "image": a texture, a depth image, a swap chain image, etc. Images have a lot of properties that can vary their presentation and color scheme.
- **IMAGE VIEW**: Quite literally a view of a Vulkan image. Every image in Vulkan needs to have an associated "view" to be seen.
- **SPIR-V**: A bytecode format in which Vulkan shaders need to be converted to be bound to the graphics pipeline.
- **SHADER MODULE**: A compiled shader, has been read in and is meant to be attached to the graphics pipeline.
- **FRAMEBUFFER**: An object which represents a collection of memory attachments that are used by a render pass. This objects takes memory references to certain objects and bundles them for a render pass to use.
- **COMMAND POOL**: An object that manages memory for command buffers, and serves as the "owner" of said command buffers and their memory. According to the specification guide, "Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use."

- **COMMAND BUFFER**: An object that records commands to be submitted to a queue.
- **GRAPHICS RENDERING**: The actual act of "drawing" a scene. This action is done by a graphics queue, and involves taking information about vertices / indices, textures, etc. and rendering them.
- **PRESENTATION**: The act of placing a rendered image onto the window surface for display.
- **BUFFER**: A resource that represents a linear array of data in device memory.
- **DEVICE MEMORY**: An object that holds the actual memory associated with VkImage and VkBuffer files.
- **SAMPLER**: An object, which accesses textures and applies filtering / transformations to them to compute the final color. Such operations include bilinear filtering, anisotropic filtering, addressing mode (repeat, clamp, etc.), and more
- **SEMAPHORE**: A synchronization object used in Vulkan, which is meant to sync operations within or across command queues.
- **FENCE**: A synchronization object used in Vulkan, which is meant to sync the application with itself while rendering.
- **DEVICE FEATURE**: The "features" that can be supported by a Vulkan device. A full list can be found **HERE**.
- **DEVICE EXTENSION**: The extensions that are supported by Vulkan. Vulkan extensions come in two levels: instance-level and device-level. Examples of extensions include the swap chain and ray tracing.
- **IMAGE FORMAT**: The color depth (channels and bytes per channel - like R8G8B8) and color space of a given Vulkan image.
- **PRESENT MODE**: The "conditions" for showing an image to the screen. More specifics on the types of present modes can be found **HERE**.
- **EXTENT**: The resolution of a Vulkan image.
- **PIPELINE LAYOUT**: An object that holds descriptor set layouts and other uniforms. It is attached to the pipeline to give the pipeline access to said layouts.

- **PIPELINE**: A Vulkan object, which binds together various stages (including vertex input, shaders, viewport / scissor, depth / stencil, render pass, and more) and
- **RENDER PASS**: An object that represents a set of framebuffer attachments and subpasses (or phases) of rendering, which use said framebuffer attachments,
- **RENDER PASS ATTACHMENT**: An individual component of a render pass, which contain frambuffer information.
- **SUBPASS**: A phase of rendering in a render pass. A subpass consists of one or more attachments, which specifies what rendering operations should be performed at this step. Render passes can have multiple subpasses; if this is the case, then subpasses are performed in a chain, with their input being taken from the output of previous subpasses.
- **SUBPASS DEPENDENCY:** Specify execution and memory dependencies between two subpasses. They are automatically performed between supasses in a render pass.

REFLECTION AND TAKEAWAYS

There definitely was a lot to take away from this independent study into Vulkan, both in terms of learning about computer graphics and myself. While I ultimately was not able to complete the Vulkan game engine, I still feel like the takeaways from this project were worth the struggle. It was awesome to do a deep dive into the crazy world of Vulkan, and be able to still see Vulkan in action through various tutorials and other projects.

The major reflection of this whole experience is how I feel like I've barely scratched the surface of Vulkan. I spent an entire semester learning about Vulkan: its pipeline, its commands, its components. Yet, despite this, I feel like I barely scratched the surface into what Vulkan is totally capable of. For starters, the application itself is still in an incomplete state, showing that I still don't have my head wrapped around all the various Vulkan commands (how many command buffers should I have? How often should I update descriptor sets? etc.) Additionally, I didn't even get into some of the crazier aspects of Vulkan, like it's official support for ray tracing. I definitely feel like I could spend a whole additional semester or year on learning Vulkan, if I had the time.

Additionally, learning Vulkan made me begin to appreciate the succinctness and brevity of OpenGL. I remember thinking OpenGL was unintuitive when I first took CSE 387 / 489; however, after having learned about Vulkan, I feel that I now not only know more about what each OpenGL command does, I appreciate each command more. For example, depth buffering is something that comes naturally to OpenGL: simply call glEnable(DEPTH_TEST). However, Vulkan is a whole different ballgame: you have to make a depth image, bind it and its memory, then attach that to the graphics pipeline, framebuffers, and render pass. Even binding items to shaders is so much easier in OpenGL than in Vulkan.

But, like I said above, I really do think that learning Vulkan was super helpful in really helping me figure out the entire rendering process and fully understand it in the context of graphics cards and shaders. Sure, we did that stuff in CSE 287 / 386 and CSE 387 / 489, but I feel like it really started to click and all come together once I was able to devote an entire semester to independently studying how Vulkan works and interacts with the graphics card.

Lastly, I feel like I learned a bit about myself while doing this project. I found that I just DO NOT internalize anything unless I use it in an applicable fashion. For example, the first month or so consisted of me following the Vulkan Tutorial in an attempt to understand the components. Despite me doing the tutorial TWICE and doing both comments and hand-written notes, I really wasn't able to internalize and fully understand everything. It wasn't until I started trying to make the game engine I began to see how all the moving components came together.

In hindsight of the semester, I was that I didn't have such a busy time with all my classes and commitments; if I had just a few more hours a week, I feel I could have really absorbed more about Vulkan and finished the project. Maybe it would have been wiser to take this independent study in the spring? Alas, what's done is done, and I would say that while the game engine wasn't completed, it was still an overall success.

CONCLUSION

As an overall conclusion of the last 40+ pages of content, I would say the TL;DR of it all is that Vulkan is HARD, but has the potential for very rewarding results. During this independent study into Vulkan, I definitely learned a ton about computer graphics and the rendering process. Vulkan's low-level API was incredibly confusing at first, but when I began work on the game engine, I found all the interlocking components to be much more understandable. I guess it's just my learning style: I need to do something for myself independently for it to really "click". I'm definitely a little saddened that I wasn't able to finish the game engine, but as I said above, I hope to come back to Vulkan at some point post-graduation and finish the work I've started.

RESOURCES USED

- **Vulkan Tutorial**: This is the main Vulkan tutorial most people use. It's great for learning the basic concepts of Vulkan, and serves as a good baseline when it comes to making a game engine in Vulkan.
    - **Website Link**: https://vulkan-tutorial.com/
    - **GitHub Repository Link**: https://github.com/Overv/VulkanTutorial
- **Vulkan API Tutorials**: A series of YouTube tutorials covering the basics of Vulkan. Very similar to the Vulkan Tutorial above, but the overall repo seems more like what a game engine should be structured like, with dynamic command buffering.
    - **YouTube Playlist Link**: https://www.youtube.com/playlist?list=PLUXvZMiAqNbK8jd7s52BIDtCbZnKNGp0P
    - **GitHub Repository Link**: https://github.com/Noxagonal/Tutorial-Planning
- **Vulkan Subreddit**: Was a great resource for figuring out some of the more intricate details of Vulkan. Many great experts and questions which helped me debug certain issues that came up.
    - **Subreddit Link**: https://www.reddit.com/r/vulkan/
- **NVIDIA Resources**: NVIDIA had a number of great resources for learning how to program Vulkan applications, including webpages and videos. Definitely a good resource, but also a bit low-level for me,.

- **Vulkan Dos and Don'ts**: https://developer.nvidia.com/blog/vulkan-dos-donts/
- **Vulkan Overview Video**: https://on-demand.gputechconf.com/siggraph/2016/video/sig1625-tristan-lorach-vulkan-nvidia-essentials.mp4

- **Vulkan Specification Guide:** The definitive Vulkan guide! Has information about virtually EVERY aspect of Vulkan, and great for figuring out the intricacies of certain commands, parameters, and ideas.
  - **Specification Guide Link**: https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf
- **Vulkan Programming Guide** (Sellers): A book about programming in Vulkan. I didn't use this a ton, but it definitely would make for a good textbook. Lots of good info here too.