Vulkan Follow Along Guide

DESPITE Vulkan being known for its apparent complexity, the design that throughlines the technology is shared with other APIs. In this sense, while this guide can be appreciated to someone new to the Computer Graphics world, some base knowledge regarding general concepts of the area might come in handy. That said, the objective here is to provide a window to allow one to peek into Vulkan, and to that end, it was established that comparing the Vulkan way of doing things regarding something that the reader could already be familiar with, such as creating a simple "Hello Triangle" program in OpenGL, could be an effective way of accomplishing that goal. This translates to a step-by-step analysis of how to create the simplest visuals using Vulkan, all the while exploring the logic behind it.

Also, as it was previously mentioned, this exercise has two main Python projects: one containing a simple "Hello Triangle" accomplished using OpenGL, and another doing the exact same thing but using Vulkan. These are meant as a reference to be consulted along with this written guide, and as per usual, the commented code is intended to explain the code itself, and not the underlying technology that is being used, which will be expanded along these next paragraphs. Lastly, the OpenGL project might be useful to put into context the Vulkan one, especially if one is already familiar with the former.

I. Environment Setup

Jumping into action, the first thing one needs to do when experimenting with Vulkan, is setting up the proper environment, as per usual. Having *Python* installed and somewhere to edit code, such as *Visual Studio Code*, is a given, but when it comes to Vulkan, we need to download the SDK from the official source, at *LunarG*. That said, we still need a way to interact with the SDK, and for that we have a *Python* wrapper simply called *vulkan,* by *realitix*, available from *GitHub*. As a final step, we also need to download the *Python* package *glfw*, which allows access to the GLFW library, along with the packages from the OpenGL file, if you're also interested in those. GLFW is needed since agnostic APIs, such as Vulkan or OpenGL, don't operate using OS (Operating System) dependent structures, such as windows. That said, we still need a window to display whatever we may want to render using Vulkan, so a library such GLFW abstracts that layer and provides us with an OS related window, which can then be used to create a Vulkan surface. As a final note, we are using *Python* as our programming language, but we could be using something else. *Python* was chosen due to accessibility and ease of use, but the way Vulkan works is not dependent on the chosen language.

II. Hello Triangle

Before we even start exploring Vulkan, the first thing we need is a window, since without it we can't even display anything. Using GLFW, the process is pretty straightforward, and even though we won't be using it much since we'll be working with Vulkan's abstractions (*vk_surfaces*), we should still store it in a variable. Next, and regarding Vulkan proper, we need to create a Vulkan Instance, which is analogous with the concept of an OpenGL context. This works as an instance of Vulkan, and stores elements such as state, Vulkan version, name, etc. To create an instance, as well as most objects in Vulkan, we first need to get all the information that is needed in the creation of said object. We then bundle it all in a structure specifically made to hold the information needed for the creation of our intended object, so that when we call, for example, *vkCreateInstance(),*instead of cluttering the functions with every requirement, we just pass the info object as an argument.

Regarding the creation of the instance specifically, we need to specify the Vulkan version to use, the name we want to give to the application, the layers we want to use, as well as required extensions. Layers are a concept that Vulkan uses to provide additional functionality, mostly related to debugging. This is done since Vulkan strives to have as little overhead as possible, so if the user wants any feature, they need to explicitly state so. Since we are writing a simple program, we will choose to opt-out, leaving the array of layers empty. A similar concept is that of extensions. In this case though, for now, we only need to make sure our system supports the *VK_KHR_surface*, since GLFW will need that to create a *vk_surface*.

Next up, we create and store the vk_surface that we will be referencing throughout the rest of the program. Luckily, GLFW contemplates this, and allows us to do so in a hassle-free manner.

With those two cornerstones out of the way, we still need to setup one other component that makes up the general architecture of Vulkan, the logical device. Sometimes simply called *device*, a logical device is an instance/representation of a physical device, which is to say, hardware that processes graphics. On laptops for example, it is common to have two distinct physical devices: there is the actual GPU, but there is also the added bonus of having a CPU that can also handle tasks of this nature.

While there is a discrete number of physical devices present, with logical devices being instances, we can have as many as we like, although in this case we only need one. In order to create it though, we first need to pick our physical device. To do this, Vulkan allows us to query for the available physical devices, and in our simple case, we will pick the first one that complies with our required extensions. As before, we only need to make sure that the device supports the creation of *swapchains*, which are needed in order to render to a surface (when we created the instance, we also made sure that surfaces were supported).

Before creating the logical device though, we need to tackle the concept of queues, since queues need to already be set up at the moment of the logical device's creation. Queues are what Vulkan uses to issue commands, which means that when we want, for example, to issue a draw command, we first send that command into a queue. This is done so we can send a bunch of commands in one go, and then Vulkan can reorder them under the hood to execute them in a more optimal way. There is also a parallelism with actual GPU hardware, since these group functionalities together, so when we are setting up the queues in Vulkan, we are just defining what values we will be using to reference these functionality groups of the hardware. That is also why we need the physical device in order to query the available queues. Lastly, Vulkan distinguishes queues based on what they can do. Some queues support computation commands, others have the ability to transfer data, while others are used to render graphics. That said, a single queue can support one or more of these functionalities, so Vulkan groups queues into families. Each family contains a set number of available queues that can all preform the same operations, so we might have a family where all queues allow for data transferring and rendering graphics, whereas another family has queues that only support rendering.

With that in mind, we first want to query the physical device for the families of queues available, and then store the index of the first family that supports graphics operations, as well as the index of the first family that supports presentation to a given surface. We store these in different variables since they may or may not be the same family. As for the creation of the logical device, the process should be familiar by now, which, as usual, entails the caching of information surrounding queues as well as other properties into an info object, which is then fed into the creation function. After having our logical device, we store an actual reference to the queues we will be using instead of having just the indices for family groups.

Next in line, is the creation of the aforementioned *swapchain*. This process can also be quite detailed and makes use of custom classes, so just like the queues, we will use a separate *Python* file to handle this, grouping together similar functionality. First up though, the concept of a *swapchain* can be a bit hard to wrap one's head around, but by understanding it, the steps involved in its creation make a bit more sense. In a general sense, the *swapchain* is what allows Vulkan to interface with the display, and present images to it. It is an abstraction put into place since one can't simply output images directly from the graphics pipeline to the screen. Not only that, but image presentation is something that is heavily tied into the OS, so the *swapchain* acts as an intermediary of sorts, keeping reference to images/frames that will be displayed. It also works similarly to the *Double Buffer* in OpenGL, but a *swapchain* can have more than two buffers for example, maximizing productivity in cases where this scenario can be handled. Once again though, we will keep our *swapchain* simple due to the nature of the project.

Before jumping into the steps involved in the *swapchain* creation, it is important to note that a few of the functions required for this step aren't directly accessible through our Vulkan *Python* wrapper. That said, we can get a reference to these functions by using *vkGetInstanceProcAddrs()* and passing as a string the name of the desired function. With that out of the way, the first information we want to gather is the surface capabilities and the color space that we are going to use. The color space is self-explanatory, but the surface capabilities include useful information such as the width and height of our window, which will be used to set the extent of the *swapchain*. This makes sense since the images that our *swapchain* will reference should have, under normal circumstances, the size of the window (surface) we want to present to. The presentation mode also needs to be defined, and this particular setting refers to the way images are sent and synchronized with the screen, be it immediately, which can result in screen tearing, or through a queue, for example. Finally, the last thing we need to set up before creating the *swapchain*, is defining how different family queues handle a given image. This depends if in the queue set-up stage, a different family was assigned to the graphics and present family, but in most cases, GPUs have a queue that supports both operations, so the image sharing mode will most likely be set to *VK_SHARING_MODE_EXCLUSIVE*, or in other words, we don't need to set up how different families handle a single image since we are only using one family.

With all of that out of the way, we can finally create our *swapchain*. Immediately after doing this though, we need to create an *Image View* for each *Image* in our *swapchain*. In order to keep the project simple, the *swapchain* was created using the minimum number of images possible, which is two, so we also only need to create two *Image Views*. In order to keep things tidy, we are grouping structures such as *Images*, *Image Views*, and later down the line, *Framebuffers*, into a custom class called *SwapChainFrame*. This is done in this way since all of these concepts are related, in a way that *Images* just reference memory along with some extra information regarding format, for example, the *Image View* defines what part of the *Image* to use, and *Framebuffers* tie *Image Views* to *attachments*, which we will check out later.

Next, we need to create the pipeline. Taking into account that this is such a crucial component of any CG work, its is not surprising that once again this step gets its dedicated file. Unlike the *swapchain* though, the creation process might seem more natural to anyone familiar with a standard graphics pipeline since the concept is the same. In a general way though, graphics pipelines are used since it is a system tailor made to GPUs, which in turn are made to be overly competent in specific tasks. That said we will need to setup most of the elements that make up the pipeline. Starting with the *Vertex Input structure*, this describes the format of the vertex data in case any data is passed onto the vertex shader, and we will keep it default for the time being. Next, we have the *Input Assembly*, which dictates how geometry will be drawn with the given vertices, and once again, we just want that three vertices make up a triangle.

Not all pipeline elements can be created out of the box though. Shaders, for example, need to be custom made and in this first case we will handle the vertex shader, which processes vertex data individually. Unlike OpenGL though, this process is done a bit differently in Vulkan. We still write our custom shader using GLSL (OpenGL Shading Language), in this specific case, in a separate file, but then we need to compile it to a bytecode format called SPIR-V, also developed by *Khronos*. This is done for portability reasons, since when driver/GPU manufacturers were to interpret something like GLSL into native code, there was some flexibility in the way to do it, which could mean a shader could be interpreted differently across different GPU's. With SPIR-V, which doesn't have human readable syntax, the middleman is handled by *Khronos*, and the standard ensures compatibility across different vendors and manufacturers. This does imply some extra steps when working with Vulkan though.

After writing our shader code in GLSL, we use the GLSL compiler provided with the Vulkan SDK, and generate a *.spv* file for our shader. It is noteworthy to say that the GLSL file must have a specific extension for the compiler to recognize the shader as a vertex shader or fragment shader, but other than that, the process is pretty straightforward. The project already comes with compiled shaders, but in case you want to recompile them after doing some changes, be sure to also edit the bat file according to your system. So, after providing our pipeline creation function with the path for our compiled vertex shader file, we read the file and create a shader module from it, later specifying in the info structure that this is supposed to be the vertex shader.

Next, we need to define the viewport and scissor rectangle, and in most cases, these are left at 0.0 values, since they describe the region of the framebuffer that the pipeline output will be rendered to.

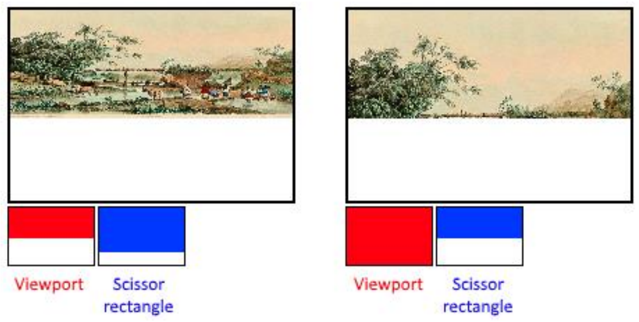

Fig. 1. Viewport and scissor representations.

Much like the viewport, the rasterizer, which is the element that creates fragments from the given geometry, will be created in the most stock way possible, since we don't want to do anything fancy with it. Consequentially, the fragment shader is also created in a similar manner to the vertex shader, so there isn't much to say regarding it. This leaves us only with the multisampling and color blending info left. Multisampling is used for anti-aliasing, and color blending is used to blend the output color with a color already on the frame buffer, and we don't want to do anything fancy with neither. That said, while these are all the structures needed for a pipeline, we still need to define a pipeline layout and *renderpass* to actually create one. The pipeline layout holds information regarding values that can be updated at draw time, but since we hard-coded the vertex information on the vertex shader, it is of little use to us. The *renderpass* is more complex and holds information regarding the previously mentioned *attachments*. In our project, the frame buffers will only have a *color attachment*, so this process is quite simple, but we could want our *renderpass* to have several *subpasses* that use several *attachments* in order to get post-processing effects, but in our case, we only need to create one *subpass* since a *renderpass* always needs to have at least one *subpass*. With the pipeline created, we can destroy the shader modules similarly to OpenGL, and move onto the next step.

Even though *framebuffers* were already touched upon, we still haven't created any, so in this next step we will create a *framebuffer* for each custom frame that we have associated with our *swapchain*. More complex than that, are *commandbuffers*, which are the structures used to issue commands to a given queue. The logic behind these, is that we first create a command pool that will manage the memory for our *commandbuffers*, and then in our case, create a *commandbuffer* for each frame. With this step done, we have almost put everything that is needed to render to screen into place.

The last thing we need to do before recording commands and submitting *commandbuffers* to a queue, is to setup a way to synchronize our GPU with the CPU. Since the GPU and CPU work independently, work that needs to be done in coordination between the two needs to be properly synchronized, which is something Vulkan leaves up to the user. In order to do this, we use both semaphores and fences. Semaphores are used to enforce an order of operations on the GPU, since calls to the GPU return immediately and run on the GPU asynchronously. This is done so we can guarantee that an image is rendered, only after being acquired, for example. The fence on the other hand, is used to keep the CPU in check, preventing it from calling the GPU indefinitely, as fast as it can. The actual creation of these structures is fairly simple though, since Vulkan already takes this into account and has custom functions for it.

With the sync structures into place, we are finally ready to bring it all together. First, we need a loop, in this case, managed by the close state of the GLFW window, and then in this loop, we just call a render function. In this render function we first need to check with the fence if the flow of operation can proceed, and if so, we get the index from the image that is next in line from the *swapchain*. With this index, we can also record the draw command in the respective *commandbuffer*, and while there are a few lines to it, it's mostly boilerplate code. We do need to reset the *commandbuffer* before recording anything on it just to be sure. After recording the command in the *commandbuffer*, we submit it to the graphics queue, and then, after the respective semaphore gets signaled, we can present the rendered image from the *swapchain*. With all of this put into place, the window should have a triangle drawn on it.

As is usual with lower-level programs, memory needs to be freed up, and in our case, this is done at the end of the program, which is to say, when we close the window surface. Since our render function has processes running asynchronously, we need to wait for any of those to finish, and after that we can start destroying the structures we have created along the way. Vulkan provides functions to handle each case though, so destroying a structure just involves calling the respective function.

V. Conclusion

Hopefully, in the end of it all you should have a multicolored triangle standing on top of an orange background, but more importantly, a better understanding of some of the basic building blocks of Vulkan. In case you want to go full gung-ho, best of luck, and better find some help somewhere else, because I for sure am not any sort of experienced voice on the matter. In any case, hopefully this was of some help, at least as a quick and dirty starter project!

VI. Appendix

In case you found this guide by itself and are wondering where the hell are the files I spent all this while talking about, you can check them in the following repo: https://github.com/luischavesdev/VulkanHelloTriangle

VII. References

[1] Toptal. *A Brief Overview of Vulkan API*. [Online]. Available: https://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api

[2] AMD. *AMD Vulkan Graphics API*. [Online]. Available: https://www.amd.com/en/technologies/vulkan

[3] Khronos. *Vulkan Overview*. [Online]. Available: https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf

[4] Alexander Overvoorde. *Vulkan Tutorial*. [Online]. Available: https://vulkan-tutorial.com/Introduction

[5] Brendan Galea. *Vulkan (c++) Game Engine Tutorials*. [Online]. Available: https://www.youtube.com/playlist?list=PL8327DO66nu9qYVKLDmdLW_84-yE4auCR

[6] GetIntoGameDev. *Vulkan with Python*. [Online]. Available: https://www.youtube.com/playlist?list=PLn3eTxaOtL2M4qgHpHuxY821C_oX0GvM7