Vulkan Recap

# 1 | Instance

## 1.0 | Overview

Instances are what you use to interact with Vulkan, you need to create one to actually be able to use the API itself. You’re going to need access to a variable to do this, and in this case it’s stored as a VkInstance which you can declare as a VK\_NULL\_HANDLE which is more or less just null.

## 1.1 | Application Info

The first thing you’re going to need to do is fill this variable with information about the application itself, so firstly you’ll need a VkApplicationInfo to feed the following data into. The main and recurring theme with Vulkan is structure types (or sTypes). Everything uses them because everything is a structure here. They all have corresponding enums with very long names VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO is the name of this one.

You’re also going to need to set the application and engine name, and their corresponding versions, which is really up to you right now, VK\_MAKE\_API\_VERSION(variant, major, minor, patch). You should also set the api version using the same #define above which is determined by the api version of Vulkan you’re using.

## 1.2 | Instance info

The instance also needs some information fed into it, so you need to declare a VkInstanceCreateInfo, with an sType of VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO and feed a pointer to the app info you previously created into *pApplicationInfo*.

Alongside those you’re going to need 4 more variables that I haven’t mentioned yet. Those variables related to extensions and layers.

### 1.2.1 | Extensions

Extensions are more or less extra bits of functionality that aren’t part of the Vulkan API that you might want access to, whether it be related to the OS itself, or something else, you might need a few. You’re going to need to store the number of extensions, and their names in some sort of array.

In order to know these values you can either manually feed them all in using defines and just counting, or you can use vkEnumerateInstanceExtensionProperties() to find them all and push them into a vector, then pull the information from said vector.

Once you’ve done that you can set the instance info’s *enabledExtensionCount* as the counter you made, and the *ppEnabledExtensionNames* as the array you set before.

### 1.2.2 | Validation Layers

Vulkan is pretty low on the error checking front, though it’s done on purpose to avoid overhead, but if you want to deal with that stuff, you’re going to want validation layers. All the basic ones are all bundled into “VK\_LAYER\_KHRONOS\_validation” is the main one here I suppose.

In order to check if the layer itself is valid you can use a function called CheckLayersSupport() and see if it exists in there. Once it’s all checked and done you can feed it into the instance info under *enabledLayerCount* and *ppEnabledLayerNames*.

## 1.3 | Finishing Up

Once you’ve gotten all that out the way, all you really need to do is throw all that info into the instance, and there’s a very simple function to handle that called vkCreateInstance().

There’s 3 parameters within the function but you’re probably only gonna need the 1st and 3rd ones. Those being the pointer to the VkInstanceCreateInfo that you just filled up, and a pointer to the instance itself that you’re feeding all of this info into.

You don’t really need the allocator, I’m not even sure what it does tbh.

# 2 | Physical device

## 2.0 | Overview

Now that we have access to the library all set up through the instance, it’s about time to grab a GPU so all that graphics stuff we have access to doesn’t just get trapped in the void. This time around the variable required to access all that will be of the type VkPhysicalDevice once again set to null for now.

## 2.1 | Finding the Physical Device

So in order to set up our physical device variable, we need to start by picking a physical device since there can of course be multiple GPU’s in a system. We can make an integer to store the device count and similar to what we did before with extensions, there is the enumerate function vkEnumeratePhysicalDevices() which will now require our previously created instance.

We’re looking to fill up our device counter first and make sure it can find any of them, so you can leave the last parameter as nullptr for now.

Once you’ve run the function you can now create an array of physical devices in any sort of way, and then rerun the function but this time adding the new array into the final parameter.

## 2.2 | Checking Device Suitability

So you now have a list of physical devices, which is great, that means the screen is turned on and Vulkan has at least one leg to latch onto. Now it’s time to check if it’s the kinda leg that Vulkan likes. The device has to meet some requirements before it can be used and assuming we have more than one we might need to check which one is the ideal choice too.

### 2.2.1 | Gathering Device Details

We’re going to need to check the devices properties, and the devices features. Both are fairly straightforward and are declared and set using the following names.

VkPhysicalDeviceProperties | vkGetPhysicalDeviceProperties()

VkPhysicalDeviceFeatures | vkGetPhysicalDeviceFeatures()

And there’s also VkPhysicalDeviceMemoryProperties which follows the same pattern though maybe it’s not as required as the last 2, there’s no harm in setting it in more or less the same way.

### 2.2.2 | Choosing a Device

Once you’ve iterated through your devices, or perhaps while iterating if you wanna do it that way. You’re going to want to make use of that original device variable you made in the overview. You want to add the properties and features that you found in [2.2.1](#_2.2.1_|_Gathering) and insert them into a new set of variables you’re going to want, which are as follows:

VkPhysicalDeviceProperties

VkPhysicalDeviceFeatures

VkPhysicalDeviceMemoryProperties

And fill them with the new values that you found, and of course you’re going to want to set the variable that you created in the overview as the current device iteration in the loop.

## 2.3 | Queue Families

Queues are really important and everything that sends a command of any kind, sends it to a queue. Those queues are found in the queue families which all allow for their own types of commands. We want a queue that looks for graphics commands at the moment.

### 2.3.1 | Finding the Queue Family

Time for some déjà vu as we once again make a counter, and start using a very familiar looking function. vkGetPhysicalDeviceQueueFamilyProperties() of which you’re going to fill out the parameters in the same way, first using it to set a counter, followed by using it to fill an array made from VkQueueFamilyProperties.

Once we’ve got a list of every queue family, we’re gonna need to find a specific VkQueueFlags within the properties variable you just made that contains the property VK\_QUEUE\_GRAPHICS\_BIT. You can quickly check for that using the bitwise & since that should return 0 if the bit flag doesn’t match.

Once you do find a matching queue family you should check for surface support, I don’t think it’s crucial though.

### 2.3.2 | Surface support

You’re going to want to call vkGetPhysicalDeviceSurfaceSupportKHR() and you’ll need a VkBool32 for that last parameter, and if there is support you can continue to set the index and jump out of the loop.

However if the surface is going to be null to begin with, perhaps you just don’t need one, you can also set that as an early escape from the loop and just set the index.

## 2.4 | Device Extensions

As sort of another check, if you want to check if your device has support for the extension you require before you go ahead and check for it using vkEnumerateDeviceExtensionProperties() the same way you use other enumeration functions, filling a counter then filling an array, though there is an extra const char\* parameter called pLayerName which I pretty much just ignored.

Once you’ve gotten all that in order loop through them all as usual to find the extension you need which in my case I was looking for VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME and deleting the extensions array you filled up to loop through. We’ll come back to this extension [later](#_5.1.2_|_How).

## 2.5 | Finishing Up

So to recap on this section. You’re looking for a compatible physical device, before you can do that you can make sure it is compatible with the extensions you need, as well as compatible with the queue family so that it can send commands.

Once you figure out the preliminary stuff, you can check for the devices properties and features, and if it all matches, then set your main VkPhysicalDevice variable with the one that matches all the compatibility, and set device properties, and device features variables you made in [2.2.2](#_2.2.2_|_Choosing), and also set the queue family index it’s going to need.

# 3 | Logical device

## 3.0 | Overview

Once you’ve gotten the physical device set up, you need a logical device, the physical device is what you see when you look in the case, the logical device is what Vulkan sees when it looks into your motherboard. For this set of steps, we’re going to be working with a VkDevice to store it all in.

## 3.1 | Deciding the Queue’s

This is a queue that and sets how many queues we want to use in a chosen queue family. The queues being that thing we send instructions through so hardware knows what it needs to do next. That queue would be the graphics one at the moment.

To do this you’re going to want to create a VkDeviceQueueCreateInfo and give it an sType of VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO and pretty much fill it with 0 for the index, there’s only going to be 1 queue in this case, aka the graphics one, and the priority can be 1 as well.

Family index is the one you set when you were sorting out the physical device, and queue count is pretty self-explanatory, there’s only the one.

Queue priority is required but you can just set it to 1.0f because lets be real neither me nor you know what we’re doing right now.

## 3.2 | Creating It

So in order to create the logical device, it comes under the type VkDeviceCreateInfo with the sType VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO. The pQueueCreateInfos will be what you made just above, and it’s 1 because we only really set 1 of them.

Now for the enabled features, extension count, and extension names, we set them all up during the physical device part after we found the queue families, you can just set them as those variables.

### 3.2.1 | Actually Creating It

For real this time, we just need to call another VkResult function called *vkCreateDevice* and fill it with all the necessary bits and once again just ignoring the allocator, and of course, it’s all going into that VkDevice variable you made at the beginning of this part. You should know by now what to do with VkResults (I probably wrote it in here somewhere).

## 3.3 | Queue Handles

Just real quick, you might want to like, have a way to handle the queue, the way to do that is with a handle. Just make a VkQueue call it the name of the queue you want handled, and fill out the *vkGetDeviceQueue* form, using the device you created in the last subsection.

## 3.4 | In Summary

To round up what we’ve done so far, we created an info structure for a logical device, and filled it up using device and queue info that we gathered from the physical device in section 2, which included creating a VkDeviceQueueCreateInfo to put into the *pQueueCreateInfos* field.

Then using that structure, we created a VkDevice and populated it with a VkResult function, and then created a handle or handles for it in the form of VkQueue’s so that we can use them.

This in turn lets Vulkan talk to your graphics card, and lets you talk to Vulkan about your graphics card.

# 4 | Window surface

## 4.0 | Overview

So because Vulkan is so barebones, it doesn’t really even default to a particular OS, so it won’t directly just work with windows immediately, in order to get this working, we’re gonna need to make use of an extension, specifically one called the WSI (or Windows System Integration). The surface at hand here is the VK\_KHR\_SURFACE. Which gives us the type VkSurfaceKHR to present images to.

## 4.1 | Surface Creation

So we’re going to need to make this right after we’ve made the instance, this is because if we do it before, it apparently influences that physical device selection part we did a few pages ago.

In order to create a surface, we’re going to want to create an object under the VkSurfaceKHR type mentioned above. As a side note, surfaces might be platform agnostic when you use them, but you actually do need to consider it when creating the surface.

## 4.2 | Surface Info Setup

First step is creating a VkWin32SurfaceCreateInfoKHR struct to store some handles. It’s going to store an sType of VK\_STRUCTURE\_TYPE\_WIN32\_SURFACE\_CREATE\_INFO\_KHR and store 2 handles, one to the window, and one to the instance.

The window handle goes under hwnd and you can get it using this absolute mess window\_info.info.win.window. Meanwhile the instance handle comes under hinstance and you can grab it with a simple GetModuleHandle(nullptr).

To tie all this together and create the surface, you’re going to need to use either vkCreateWin32SurfaceKHR or in my case SDL\_Vulkan\_CreateSurface() because I used SDL to set this up. Don’t ask me what that means frankly I have no idea, I’ll probably figure it out later in the tutorial.

## 4.3 | Queue Alterations

So now we need to modify the part where we found the queue family, this is what section 2.3.2 was talking about, that surface parameter in the function call is the global one you made in here. Essentially what was happening was if there is surface support, then the queue would be selected.

## 4.4 | Presentation Queue

Remember 3.3? Basically we’re doing that again, straight after where we call the vkGetDeviceQueue() function. This time however we change the last parameter to a different VkQueue we’re going to call present\_queue. They’re probably going to be the same 2 values so yeah. I’ll get back to this part and figure out what presentation is meant to do, I’m guessing it’s the queue that puts stuff on the screen, otherwise my understanding of queue’s has gone horrible wrong somewhere.

# 5 | Swap chain

## 5.0 | Overview

So there’s thing called a framebuffer, essentially things are rendered into a buffer before they’re visualised all at once. The system that holds all that together is known as the swap chain and we need to make it in Vulkan. This chain of moving buffers is what helps sync with your screens refresh rate. For this section we’re gonna make a VkSwapchainKHR and get it all filled out.

## 5.1 | Checking For Support

### 5.1.1 | Why It Needs Support

Some graphics cards don’t present things directly to a screen, for example the server ones that come without display outputs. That means we need to search for support, but also it’s about how you put images on surfaces, and since our surfaces are tied to windows, you’re also looking for support based in windows.

### 5.1.2 | How to Check

So first up we need some extension names stored, that means you’ll need some strings, though in this case probably only the one called VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME but it’s best to keep an array of them. We checked for this during out extension checks earlier on.

Since we’ve already dealt with that we can start filling out the support details, so we’re gonna want to check 3 different types of variable VkSurfaceCapabilitiesKHR, and 2 pointers, VkSurfaceFormatKHR, and VkPresentModeKHR.

First you’re going to fill the capabilities with vkGetPhysicalDeviceSurfaceCapabilitiesKHR() and fill in the parameters accordingly with what you’ve made so far. Once again it’s a VkResult so you can check the result to VK\_SUCCESS.

Next you’re going to move over to the surface format. The function you’ll need is vkGetPhysicalDeviceSurfaceFormatsKHR() and it should fill the count variable, then once you’ve got your VkResult out of it you can go ahead and create that VkSurfaceFormatKHR array from that pointer, or just make it a vector it’s up to you really.

Once you’ve done that, literally just run it again, but this time put in that fresh new array you just made into the last parameter.

The VkPresentModeKHR is the same process but instead uses the vkGetPhysicalDeviceSurfacePresentModesKHR() function instead. Once all that’s out the way you now have the support set.

## 5.2 | Swap Chain Settings

Now that all that is out the way, there’s another set of things to confirm within the swap chain support has been checked. In particular 3 things that you need to know before continuing, that come with 3 fitting function names. Surface format, presentation mode, and swap extent.

So let’s go through them one by one, starting with the surface format. 2 of these you just dealt with when checking swap chain support.

### 5.2.1 | Surface Format

Surface format is about the colour settings, your VkSurfaceFormatKHR struct contains 2 values. Format and colour space. Format is for the channels and their sizes, so for example we’re going to outfit the struct with VK\_FORMAT\_B8G8R8A8\_UNORM for the format, which is blue, green, red, and alpha at 8 bits each, I forget what that unorm part is all about.

The colour space is going to be VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR. But I think it’s changed now to VK\_COLORSPACE\_SRGB\_NONLINEAR\_KHR. Regardless this means we want the SRGB colour space which is pretty standard stuff.

What you’re going to want to do is loop through all the VkSurfaceFormatKHR’s you have, and find the one with those 2 settings available, and return said format.

### 5.2.2 | Presentation Mode

Presentation mode is how your image will be presenting to the screen. The one we’re looking for is VK\_PRESENT\_MODE\_MAILBOX\_KHR which is essentially triple buffering.

If the mode isn’t there your safe bet is VK\_PRESENT\_MODE\_FIFO\_KHR which is similar to mailbox but is more like a queue of frames that of course goes first in first out, and the queue just waits if it gets filled up. There isn’t really much else to it, presentation mode is pretty simple.

### 5.2.3 | Extent

Extent is the image resolution of the swap chain buffers and is more or less the same as the window usually. It’s stored in the VkExtent2D variable and the details are stored within the VkSurfaceCapabilitiesKHR variable that was filled up earlier in this section.

Essentially you want to set it to the currentExtent of the capabilities variable if it’s not at the uint32 max, or if it goes over the capabilities boundaries, otherwise if it manages to go over the capable sizes, just either clamp the value or snap it back within the boundaries.

## 5.3 | Creation

So when creating a swap chain, it’s gotta start once we’ve set up the logical device. Now that we’ve set the colour, the buffer style, and the resolution, we now need to specify the minimum image count, which is within the previously mentioned capabilities. You can just store that number in a variable.

It’s recommended to get one more than that minimum though, almost like it’s a buffer when the latest one isn’t ready. On top of that you shouldn’t go over the limit either, capabilities once again has you covered on that front, you have to make a check so you don’t spill over. (Side note: check for 0 as well, when there’s 0 that means there’s no limit).

To begin with you need to make a VkSwapchainCreateInfoKHR struct to fill in.

### 5.3.1 | Filling In the Struct

Now time to put it into a structure as we usually do, this one in particular is a pretty big one. Starting with the sType which is a VK\_STRUCTURE\_TYPE\_SWAPCHAIN\_CREATE\_INFO\_KHR. Then we simply put in the values we created from section 5.2.

There’s also some manual variables you need to fill out that we haven’t gone over. imageArrayLayers specifies image layers which isn’t really relevant since you’re just going to put 1, imageUsage is another one of those bit fields that asks what we’re going to use the pixels for, we’re just putting colours on them so we use VK\_IMAGE\_USAGE\_COLOR\_ATTACHMENT\_BIT because it says something about colours.

After that there’s variables to fill in like clipped, which just means do you care about the colour of a pixel behind another pixel, like transparency and things which I just set to true for now. Then there’s oldSwapChain which you can just ignore null handle for now because frankly I don’t know what it’s deal is.

The next one imageSharingMode is all about queue families again, and how the images can be transferred across multiple queue families. There’s VK\_SHARING\_MODE\_EXCLUSIVE and also a concurrent mode. I just went for exclusive because the graphics and presentation queue’s I have are part of the same family. If there does end up being ownership transfer then it will end up being a bit of a drag but that’s a problem for future tutorial me to worry about. There’s also a pQueueFamilyIndexCount and pQueueFamilyIndices but those can be 0 and nullptr since we don’t need them in this case.

The variable preTransform is just checking if an extra transform needs applying to images before it’s output like a flip or something, but for now just making it the current one within capabilities is fine.

And finally compositeAlpha is all about making the window blend with other windows within the OS, and honestly unless you’re making one of those spicy looking gamer tools you should be fine to just make it VK\_COMPOSITE\_ALPHA\_OPAQUE\_BIT\_KHR because it doesn’t really matter.

## 5.4 | Putting It All Together

Now that we’ve gotten all the internals sorted, we can finally get our VkResult out using vkCreateSwapchainKHR() and filling out all the parameters, as usual making the allocator nullptr.

### 5.4.1 | Retrieving Swap Chain Images

The swap chain is made but we still need to retrieve the handles for the VkImage’s in them, which we’ll need later on. You’d need to make an array of some kind, (in my case using a unique\_ptr but vector is fine too).

We’ve done something similar to this before, where we called the same function twice, first time to receive the size. Time to check another result, this time being the vkGetSwapchainImagesKHR() function, the first time round ignoring the last parameter since we don’t have a number to fill it with yet. This does of course mean we need an image count variable, then we can set the array size for the image array you created before. That now properly sized array can then go in the last parameter.

## 5.5 | Summary

So what we’ve done in this fairly large chapter is create a way to put frames onto the screen not pixel by pixel. We checked for surface and presentation support, then once we found them, we started setting up what our chain would look like, we decided on the colour, the amount of links, and how bulky the chain would be, so in other words, colour, presentation mode, and resolution.

Once we had all the details down, we started working on creating the swap chain itself. So we did what we always do and create a struct, this time a rather large one, and started filling it with the info we have already created, alongside some new info that is more of a choice thing than a setup thing.

After that we created the result, and then fetched some image handles out of the swap chain we just made since we’ll need access to them later.

It might also be useful to store the format and the extent somewhere because apparently those are useful later.

# 6 | Image views

## 6.0 | Overview

So the image view is essentially just what it says on the tin, it lets us see into the image. The VkImageView tells us how we should be using the image and the general properties of it.

## 6.1 | Struct Setup

So first things first we’re pretty much carrying on from the last part directly, because image views require the images we created in the swap chain. Resize the VkImageView array that we made before to be the same size as the VkImage that we setup in those last 2 result functions.

Once that’s sorted you need to setup a loop to go through all of the images and create a VkImageViewCreateInfo struct for all of them.

## 6.2 | Filling the Struct

Firstly is the sType, you’re going to want VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_CREATE\_INFO this time around, and the image is going to be pretty much the image in the iteration of the loop you’re in.

Then there’s the view type which is going to be set as VK\_IMAGE\_VIEW\_TYPE\_2D the 2D one. The format is going to be the surfaces format that was set earlier, there’s then going to be 4 components of RGBA, you can either do VK\_COMPONENT\_SWIZZLE\_R with the corresponding channel at the end or default it to VK\_COMPONENT\_SWIZZLE\_IDENTITY, I don’t know why they used the word swizzle here.

Next up you have the subresourcerange which requires that 5 fields are filled in. The aspectMask, the levelCount, the baseArrayLayer, and the layerCount. These essentially let you know which part of the image is being accessed, but since we are just mapping colours to our images it doesn’t really matter. That means we’re not dealing with mipmapping, or multiple layers, the counters by extension can also just be 1. This does mean that the aspect mask will take a VK\_IMAGE\_ASPECT\_COLOR\_BIT.

## 6.3 | Getting the Result

Of course as usual, once we’ve set the struct up, we need to send it to one of those magical vulkan functions that returns a result while doing stuff with your designated structure. I think by now you know how this goes but the device needs your device, the createinfo will take the struct, nobody cares about the allocator and the last one takes the corresponding image view you just created in the loop which I completely forgot about.

## 6.4 | Summary

This was a fairly short section because most of the work was done in the last one, we were more or less just fetching the previous images swap chain images we set up and creating corresponding views for them so that we know how they’re meant to be displayed to the end user.

# A | Graphics Pipeline

## A.A | Overview

Now it’s time for the big part, this section is probably gonna consume a few pages because the graphics pipeline has a lot of steps to go through, lets quickly go through all the steps in section 7 before we start piecing it together.

To begin with, the graphics pipeline puts triangles on the screen. Now that sounds pretty simple, but remember it needs to get all that data from a mesh and put them into a set of pixels on the screen.

## A.B | Input Assembler

To begin with the pipeline starts at the input assembler, there will be a subsection for every step of this process btw, but I’ll make the full sections once we’ve overviewed them. The input assembler collects the vertex data necessary for the triangles we’re about to make.

## A.C | Vertex Shader

The vertex shader now takes all those things points that you lined up and lets you move them about, specifically so that they can be converted from points in a space to points on a screen, This is pretty much critical for literally anything 3D.

## A.D | Tessellation

This is less blood and guts and more visual spice, it lets you subdivide geometry as it gets closer so it doesn’t look flat, it’s that thing that made cobblestone floors look bumpy in medieval games. In fact I don’t think older games used tessellation since it didn’t exist yet.

## A.E | Geometry Shader

This one is another one like tessellation where it takes the triangles and lines and points (which are called primitives) and decide where it needs more or less of them, but again it’s like tessellation where it’s all about spice at this point.

## A.F | Rasterization

Now we’re back to blood and guts territory and rasterization is more or less how the triangles mentioned get turned into batches of pixels, based off of how they fall on the screen space. From here they can get placed into the frame buffer more or less.

## A.G | Fragment Shader

This now takes those rasterized pixels and decides what they should look like, so colour more or less. This also means lighting is set here, and usually it’s all set using co-ordinates of vertexes and normals from the faces.

## A.H | Colour Blending

From the rasterization stage some pixels might map to the same place on the frame buffer, in which case depending on transparency they can either overwrite or mix depending on transparency.

## A.I | Summary

This section was more or less just an outline of each stage, next up we’re going to go more in depth and individually make the stages, that way when we’re making them we know what we’re making.

Not all of these stages are necessary as you may have noticed, some of them are there to make the frame buffer prettier, and some aren’t even used like the geometry shader.

The only 3 that I believe are truly mandatory are the input assembler and the vertex shader since you need to them for triangles since that’s what gpu’s need, after that it will work without tessellation, most games don’t even use geometry shaders, nvidia is trying their damn best to get rid of rasterization, and doing that gets rid of the fragment shader and colour blending.

That being said it’s good to know these because ray tracing isn’t exactly common and most games like to use most of these steps anyway.

# 7 | Pipeline Implementation

## 7.1 | Layout Info

So we’re going to set up a graphics pipeline, step one to doing that is setting up the layout. The layout of course is set up the same way we set everything up, Using an sType. You also need to create a global VkPipelineLayout under a VK\_NULL\_HANDLE for now which you’re going to need when you fill up the layout.

What we want to start with is creating a variable of type VkPipelineLayoutCreateInfo which is of course a struct, and fill it with the sType VK\_STRUCTURE\_TYPE\_PIPELINE\_LAYOUT\_CREATE\_INFO. The next 4 variables you need to fill are setLayoutCount, pSetLayouts, pushConstantRangeCount, and pPushConstantRanges, all of which you can leave empty, so 0, nullptr, 0, 0.

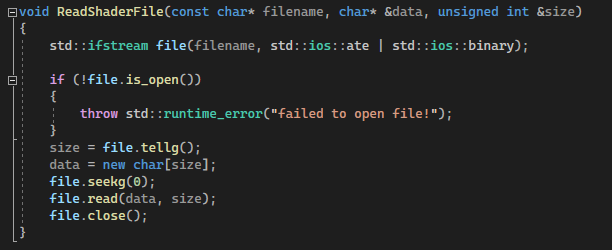
In order to fill the structure up you would VkResult it using vkCreatePipelineLayout() and then assert to make sure it all went through correctly. I’d tell you how but at this point I’m sure you can just hover over the function and look at the parameter names, while I once again just decide to ignore the allocator. We have now just set up the layout info, and pushed that data into the pipeline layout.

## 7.2 | Shader Layout

We also need to set up how many stages of shader we’re going to have. I’ve only really done 2 of them before and those 2 are vertex and fragment. Both pretty important stages though and we’re gonna set them up side by side because they look really similar in terms of setup, usually the main difference between the 2 in this section is changing the word “vertex” to “fragment”. Wild stuff.

Then we need to read some shader files. Create a char pointer for the vertex shader data, you can leave it as null for now, and make a size integer and set it to 0 as well. Next we need to fill these two variables up by reading it off of a shader file.

I wasn’t planning on putting any code examples in this doc, but here go nuts it’s just a file reader.



The parameters correspond with the 2 variables I just mentioned, they just need filling up.

To begin with, you need a VkShaderModuleCreateInfo which will determine… I guess how you create the shader. The sType would be VK\_STRUCTURE\_TYPE\_SHADER\_MODULE\_CREATE\_INFO. As for the other parameters, you need a codeSize and a pCode. The former is the size of the latter, and the latter is used to create the shader module, more specifically it’s pointing at code that is used to create the shader module.

What you’re going to need to do is make a reinterpret cast for that vertex shader data you filled up a few paragraphs ago, casting it to a <const uint32\_t\*>.

Then you just VkResult it as usual using the create info and the module itself, The result function is called vkCreateShaderModule().

So by now we’ve made 2 modules for the shaders, we now need to put this pipeline together. So we know that there are only 2 stages we’re working on at the moment, the vertex and the fragment shaders, we want to set up some VkPipelineShaderStageCreateInfo’s in a unique pointer array.

Because we’re storing creation info, that means we’re storing structs. Back to familiar territory we need to set the sType as VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO. The name I don’t think matters. The part where it’s going to differ is the stage and the module. The stage is called VK\_SHADER\_STAGE\_VERTEX\_BIT and the module is the one you created with that function I previously mentioned.

For the fragment shader do the same thing but just using the fragment shader bit instead, and the fragment module you made. With that your 2 shader stages should be filled in that array.

### 7.2.1 | Input Bindings i gess

Next we need to deal with how many sets of vertex data the pipeline is going to take at a time. In this case it’s once, but we’re going to make it another pointer array of type VkVertexInputBindingDescription of that size. This struct has 3 values to fill, a binding, an input rate, and a stride. As a rough explanation

Binding is a binding number, I don’t know what that means but it’s 0 in my case. Input rate is how often the attributes are pulled from the buffers into the pipeline. VK\_VERTEX\_INPUT\_RATE\_VERTEX will be what you set it as.

I think stride is just how many bytes does the element in the buffer take up before you get to the next element, so it’s just the sizeof() for the struct holding your vertex data. Which reminds me, you want to make a struct that will let you store the details of all your vertices. So for example I have a struct made that stores xyz values, colour values, and a padding value.

### 7.2.2 | Input attributes

So previously we dealt with input binding descriptions, this time we’re dealing with input attribute descriptions. The difference is now we’re defining what the vertex is made of, rather than defining how the data is sent in. Hence why this time we’re going to be filling parts of the vertex data struct we mentioned when filling in the stride.

I mentioned position, colour, and padding, we’re going to set up the first two. So once again we need to set up a pointer array of type VkVertexInputAttributeDescription for the 2 attributes and within each value of the array there’s 4 members that need setting up, binding which is just 0 again, location which is the position in the array I guess.

Format is actually more specific, VK\_FORMAT\_R32G32B32\_SFLOAT the reason for this is because I set my positional data to be 3 floats of xyz, and xyz and rgb are usually pretty interchangeable when it comes to data. This does indeed mean that when it comes to colour, you just copy paste it since that’s also just 3 floats in my case. The offset you can find using offsetof() which is filled using the struct and the value within the struct. This would also be the same for the colour description, but the offset would be of your structs colour value rather than position.

### 7.2.3 | Putting It Together

Now that we have both of our descriptions sorted, we need to put it together into that thing vulkan likes to see whenever it stores any data. That’s right, it’s a create info specifically of the type VkPipelineVertexInputStateCreateInfo which has 5 values that need filling. To get the sType out the way it’s VK\_STRUCTURE\_TYPE\_PIPELINE\_VERTEX\_INPUT\_STATE\_CREATE\_INFO.

These values are quite familiar, the 2 count values seem familiar because they are the pointer arrays we made earlier, binding description count and attribute description count. The other two are the binding and attribute descriptions themselves.

### 7.2.4 | Primitives

So the vertex was the long part, that sort of required a lot of values to be set up for the vertices going in, the rest is more a case of filling in the blanks in some data structures. To begin with let’s get the input assembly set up.

The input assembly is quite short, the actual typing is VkPipelineInputAssemblyStateCreateInfo and it requires an sType of VK\_STRUCTURE\_TYPE\_PIPELINE\_INPUT\_ASSEMBLY\_STATE\_CREATE\_INFO and then a topology and a primitiveRestartEnable. But “Aha!” you may be thinking, “what the hell do those things do?” And to that I say, I don’t know either so let me google it real quick.

It’s basically how the connected points will be sent into the vertex buffer. So for example we’re going to use VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_LIST which is for a series of triangle primitives, so basically every set of 3 vertices that we send it connects as its own triangle. We also want a primitiveRestartEnable set to false because what it’ll do is discard primitive values that don’t form a full triangle.

### 7.2.5 | Viewport

That was meant to be short, anyway moving on there’s the viewport state which is far simpler, it comes under the structure VkPipelineViewportStateCreateInfo and requires the sType VK\_STRUCTURE\_TYPE\_PIPELINE\_VIEWPORT\_STATE\_CREATE\_INFO alongside a viewport and scissor count. The viewport count is likely just what it says on the tin, as for the scissor count it needs to be the same as the viewport count and essentially operates within the viewport and lets you do pixel shader operations on specific portions of the screen. The viewport is a separate transform and isn’t going to be modified by this, which is weird because it sort of looks like they are similar by this stage.

### 7.2.6 | Rasterizer

Next up is the rasterizer. This is going to take a few more values than the previous ones so let’s get ready to explain everything, it comes under the type VkPipelineRasterizationStateCreateInfo. The sType is VK\_STRUCTURE\_TYPE\_PIPELINE\_RASTERIZATION\_STATE\_CREATE\_INFO. Ok now that that’s out the way have this thing called a depth clamp, we’re going to set it to false, as with every value in this thing it’s probably super important for some niche use, it’s meant to clamp a fragments depth values, cool story bro.

We also have something called a rasterizerDiscardEnable, which can allow primitives to be discarded before rasterization, again probably useful for some reason, we’re going to set it to false though.

Next polygonMode is going to be set to VK\_POLYGON\_MODE\_FILL which will allow I assume fill all those triangles we sent in before. The line width can be whatever, it’s 1 for now. Cull mode is another simple one because I don’t want to focus on culling at the moment, optimization isn’t super important right now. Just set it to no culling by doing VK\_CULL\_MODE\_NONE.

Next up there’s something called frontFace. This is essentially how the rasterizer is going to determine whether we’re looking at the front of a face or the back, this is useful in regards to that culling value we set before, I don’t know how it’s calculated but the value is VK\_FRONT\_FACE\_COUNTER\_CLOCKWISE.

Finally we have to sort out the bias, and by sort it out, we’re going to totally ignore it. Set enable to vk\_false, constant factor, clamp, and slope factor to 0 and call it a day, essentially depth bias is manipulating how you may see the depth of an object being rasterized. Probably I haven’t actually looked it up.

### 7.2.7 | Sampling

Next up is multisampling, and this is when we uhh… sample more pixels. The structure is under the type VkPipelineMultisampleStateCreateInfo and has the sType VK\_STRUCTURE\_TYPE\_PIPELINE\_MULTISAMPLE\_STATE\_CREATE\_INFO.

The next value to set is sampleShadingEnable which we want as another vk\_false. Sample shading can be used to set a minimum for unique samples for a fragment, you also need a fragment shader enabled to use this. We’d be lucky to have a functioning pixel by the end of this so let’s not think about that.

Then there’s the rasterizationSamples which we set to VK\_SAMPLE\_COUNT\_1\_BIT. This property is how many times we sample a pixel, we’re sampling it once, so we set it to 1.

After that we have minSampleShading which we set to 1.0f, and this sets the minimum fraction of sample shading, lord only know what that means but I’m going to write it here anyway.

The pSampleMask is a pointer to a VkSampleMask, we’re not using this so just set it to nullptr. I don’t know what this means someone please help.

AlphaToCoverageEnable and alphaToOneEnable are also set to false, the former controls when a coverage value is generated based off of a pixels alpha, the coverage value detects whether or not one of the primitives for a triangle is covered or not and it’s using the alpha value for that. The latter value determines whether the alpha value of the coverage is replaced by 1.

### 7.2.8 | Depth Buffer

The depth stencil comes in with a type of VkPipelineDepthStencilStateCreateInfo and an sType of VK\_STRUCTURE\_TYPE\_PIPELINE\_DEPTH\_STENCIL\_STATE\_CREATE\_INFO. The depth stencil is essentially used to calculate how far away something is. Usually it’s called a Z buffer because Z usually indicates forwards in a game.

We’re going to set depthTestEnable and depthWriteEnable to true, this will as expected enable depth testing to allow for culling and also enables writing that depth to the values.

DepthCompareOp is set to VK\_COMPARE\_OP\_LESS\_OR\_EQUAL and this determines how the depth is compared by choosing the operators used. We used less or equal because we’re going to check if something is lesser in depth, or equal.

As for the depthBoundsTestEnable and depthStencilTestEnable, those are both false and they determine the min and max values to depth test at and we’re not using a stencil test, stencil tests will just discard extra fragments if they need to chop out parts of a frame.

### 7.2.9 | Colour Blending

Next is something called the colour blend attachment and it comes under the type VkPipelineColorBlendAttachmentState and this struct doesn’t have an sType. We do however need to write a few values to it.

To start with the colorWriteMask is going to use 4 values.

* VK\_COLOR\_COMPONENT\_R\_BIT
* VK\_COLOR\_COMPONENT\_G\_BIT
* VK\_COLOR\_COMPONENT\_B\_BIT
* VK\_COLOR\_COMPONENT\_A\_BIT

These 4 are bit masks and are all connected with bitwise OR operators to stack all the colour bits up because I don’t think we want to mask off any colours. We’re also going to let blendEnable be set to true because colour blending is kind of important.

Then there are 2 ColorBlendFactors, src and dst, src sets the blend factor for the source, and dst for the destination. We’re going to set src as VK\_BLEND\_FACTOR\_SRC\_ALPHA and dst as VK\_BLEND\_FACTOR\_ONE\_MINUS\_SRC\_ALPHA.

After that the colorBlendOp determines how the blending for the RGB is calculated, we’re going to use VK\_BLEND\_OP\_ADD and then we move on to the AlphaBlendFactor which is kinda similar to the colour blending part.

Once again there’s an src and a dst, the src this time however takes the VK\_BLEND\_FACTOR\_ONE\_MINUS\_SRC\_ALPHA value and the dst takes VK\_BLEND\_FACTOR\_ZERO. Now we just need to do the alphaBlendOp like we did to the colour blending operation and we’re using the additional one again.

Now the attachment didn’t have an sType, but that was because this next part is the actual component for it, so we’re now declaring a struct of type VkPipelineColorBlendStateCreateInfo with the sType VK\_STRUCTURE\_TYPE\_PIPELINE\_COLOR\_BLEND\_STATE\_CREATE\_INFO and we’re going to need to set some more values including using the attachment we made before.

logicOpEnable allows you to apply logical operations, and then logicOp decides which one to apply, we set it to false and then set the logic operation to VK\_LOGIC\_OP\_COPY. I don’t know why this is a thing but it’s a thing.

Next up is the attachmentCount and a pointer to the attachments, we have one of them that we just made, so just put it to 1 and then set a reference to said attachment.

As for the blend constants this is one of those rare instances where it’s literally just an array of 4, but we’re going to do that thing we always do where we set all of them to 0. These are just the 4 blend constants for the RGBA components.

### 7.2.10 | Dynamic States

Next up are dynamic states, also typed as VkDynamicState. There are 3 that we deal with so we’re making an array of them, the 3 types you want to inhabit the array are VK\_DYNAMIC\_STATE\_VIEWPORT, VK\_DYNAMIC\_STATE\_SCISSOR, and VK\_DYNAMIC\_STATE\_LINE\_WIDTH. Dynamic states allow you to change more about a state without recreating the pipeline, this is necessary because a lot of the pipeline is sort of baked in so changing things can be a bit of a pain.

As for the create info, it comes under a type named VkPipelineDynamicStateCreateInfo and it’s filled exactly how you expect a create info to be filled. sType this time is VK\_STRUCTURE\_TYPE\_PIPELINE\_DYNAMIC\_STATE\_CREATE\_INFO, pointer to the dynamic states would just be feeding it your struct array, and the state count would be the size of that array, leaving flags at 0.

### 7.2.11 | Pipeline create info setup

We now have every stage set up, so we can prepare a VkGraphicsPipelineCreateInfo as we get ready to fill out the struct that will be used to create the pipeline. The sType is going to be VK\_STRUCTURE\_TYPE\_GRAPHICS\_PIPELINE\_CREATE\_INFO and there’s going to be a lot to fill out.

Starting from the top, the shader stage count is going to be form the VkPipelineShaderStageCreateInfo and the pointer to the stages will be from the stages themselves.

Afterwards grab the part from the input attributes and make a reference to the vertex input info, and then the input assembly create info, then the viewport state create info, and the rasterizer, we’re basically just going from section 7.2.1 to here and running down the list until we hit the pipeline layout. That is actually the null handle we made in section 7.1.

## 7.2 | Render Pass

It’s time to take a little detour. We stopped at the render pass because we haven’t actually made one yet. We’re talking about it after we made the pipeline because ~~I forgot that this is made first so the pipeline can use it~~ while it is made first, it is still part of the pipeline process, and it kind of slots in after all the other parts are set up. The type we’re working with here is called VkRenderPass.

Before we make a render pass we have to set up the colour and depth buffer information, what the render pass then does is wrap them all up so that they can be rendered through a render pass object. The render pass is essentially the part that does the rendering to the buffer, we have to create this before we slot it into the pipeline right after where section 7.2.11 happened.

### 7.2.1 | Attachment Description

We have 3 attachments to set up here so this might take some time, what we want to do is make a few arrays for those 3 attachments under the type VkFormat and put in the following 3 types into the array

1. VK\_FORMAT\_D32\_SFLOAT - This gives 32 signed floating point bits to the depth
2. VK\_FORMAT\_D32\_SFLOAT\_S8\_UINT - Same as before + 8 unsigned int bits in the stencil
3. VK\_FORMAT\_D24\_UNORM\_S8\_UINT - Same stencil + 24 unsigned normalised bits for depth

Once we’ve defined the formats it’s worth checking what is supported by the GPU.

### 7.2.2 | Format Support

So we’ve defined a few formats we we’re going to loop through them and check their properties to see if our GPU supports it. So what we need to do is make a VkFormatProperties as a temporary variable, and put it into the vkGetPhysicalDeviceFormatProperties() function alongside the relevant parameters.

Once we have the properties filled out with that function, check the properties for optimalTilingFeatures to have the bit VK\_FORMAT\_FEATURE\_DEPTH\_STENCIL\_ATTACHMENT\_BIT and return the format that has it. Ideally this happens on the first loop where we set VK\_FORMAT\_D32\_SFLOAT because that’s the one we want to use. The rest are just backup. If that doesn’t work out, set it as VK\_FORMAT\_UNDEFINED since nothing seems to be supported which is probably a different can of worms.

### 7.2.3 | Attachment Description Setup

Back to the attachment descriptions, we’re going to need 2 of which you might recognise the names of from previous sections. Present, Colour, and Depth. They all set up in a similar way and by similar I mean this first stage is identical so take one and do it to all 3.

The format to begin with is just your surfaces format, it’s matching the swap chain images, our sample count is set to 1 bit which you might recall from the multisampling section that we’re deciding not to dig into.

loadOp and storeOp determine how we deal with the data before and after rendering. We want to clear the values to a constant at the beginning so we use VK\_ATTACHMENT\_LOAD\_OP\_CLEAR. As for storeOp which happens after we’ve rendered we set it to VK\_ATTACHMENT\_STORE\_OP\_STORE to let us read the memory later.

As for the stencil versions of these two, we’re not using it so just use VK\_ATTACHMENT\_LOAD\_OP\_DONT\_CARE, and VK\_ATTACHMENT\_STORE\_OP\_DONT\_CARE respectively. We’re going to work with the colour and depth values from the last two so that matters a bit more.

The initialLayout is going to be VK\_IMAGE\_LAYOUT\_UNDEFINED because we’re going to clear the initial layout anyway. The finalLayout will use VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR so we can put it on a buffer after rendering.

Since we have 3 of these it’s worth storing these descriptions in an array for later.

### 7.2.4 | Attachment References

We’re sticking with the rule of 3 here so we need 3 suitable attachment references for those 3 attachment descriptions (or if you really don’t care then make them unsuitable I won’t judge). They come under the typing VkAttachmentReference as expected and they only have 2 values to fill.

To roughly explain what we’re doing here, render passes have subpasses which bundles different operations together for possible performance improvements, I don’t know exactly why but it’s more of an optimization type thing.

First is just “attachment” which is probably just where in the array that attachment is, followed by the layout which is VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL for the presentation and colour and VK\_IMAGE\_LAYOUT\_DEPTH\_STENCIL\_ATTACHMENT\_OPTIMAL for the depth stencil. It’s called optimal because its performance optimised for reasons beyond my lazy comprehension.

When we get to the part about subpasses, we’re going to need to bundle our colour attachment references together, basically just make an attachment reference array for present and colour. Once that’s done you can make your first subpass. Exciting I know.

### 7.2.5 | Subpass Setup

Subpasses come under the typing VkSubpassDescription and require just a few properties for the struct. Firstly it needs a bind point which is more or less the pipeline type. We’re making a graphics pipeline so use VK\_PIPELINE\_BIND\_POINT\_GRAPHICS.

As for the colour attachment count, just refer back to the attachment array you made about 2 paragraphs ago, and that also applies to the pColorAttachmentCount. As for the depth stencil attachment it’s exactly what you expect, it’s the depth attachment that we made earlier.

Next we need to deal with a subpass dependency. Since subpasses don’t run in any specific order this makes it so that we can tell it what order to run in. We only have one subpass so we’re only going to set the dependencies for this subpass.

They come under the type VkSubpassDependency and you have to fill up a bunch of src’s and dst’s. To begin with we need the srcSubpass which we set to VK\_SUBPASS\_EXTERNAL which makes it depend on the previous subpass. As for the dst version that’s going to be 0 because we basically have one subpass anyway.

The next part involves a stage mask which is a bit mask for a given number of stages. In the src we’re setting VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT which is a mask for every rendering stage until now, so we’re waiting up to the colour attachment part last subsubsection. The dst will be the same because it’s the same subpass.

The access mask is a bitmask for the memory access types that will be used by the src and dts subpass that we set earlier, We’re leaving the src at 0 because the src was in the last render pass. The dst masks off the parts we’re going to use in dst subPass which is the following, VK\_ACCESS\_COLOR\_ATTACHMENT\_READ\_BIT | VK\_ACCESS\_COLOR\_ATTACHMENT\_WRITE\_BIT. I don’t know for sure but the reasoning would be we dealt with it in the dst, and we only have 1 subpass so we don’t need to worry about previous passes.

Finally we have dependency flags which we set to VK\_DEPENDENCY\_BY\_REGION\_BIT. I don’t know what a dependency flag does, but we need this flag in particular because we made the src and dst subpasses equal.

### 7.2.6 | Render Pass Creation

Finally we now have all the pieces in place and we can make the create info for the render pass, it comes under the type VkRenderPassCreateInfo and has an sType of VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_CREATE\_INFO.

The attachment count you should be able to recall from 7.2.3 as we’re attaching the descriptions and so the attachments would be the array of descriptions themselves.

Our subpass count will of course be 1 and we also need to pass in the subpass that we just made. Same goes for the dependency that we just made.

After that we can just VkResult it using vkCreateRenderPass() and get the function filled out the way we always do.

## 7.3 | Pipeline Completion

Now that we’ve gotten the render pass sorted out we need to go back to the pipeline in 7.1 and finish off the creation info for it, starting with adding the render pass and then setting the subpass to 0. The base pipeline handle and index are optional so we can null handle the first one and put -1 for the index. The final bit we need is the flag VK\_PIPELINE\_CREATE\_ALLOW\_DERIVATIVES\_BIT which allows this created pipeline to be the parent of the next one.

After that we can result it using vkCreateGraphicsPipelines() and fill it in using the correct parameters assuming they exist, you’re going to null handle the cache because we didn’t make one of those.

### 7.3.1 | Buffers

Finally we have to create the vertex and index buffers. We did previously make a function to create a buffer, it would be exactly like that but it’s going to require some changes so it needs to be less rigid, so we’re going to give it some parameters. The variables that need converting over are VkBuffer, VkDeviceMemory, VkDeviceSize, VkBufferUsageFlags, VkSharingMode, and finally VkMemoryPropertyFlags. Just go through the old version and find the spots where these variables are used.

That does indeed mean we need to declare a few variables before we get these buffers handed in, It’s mostly just declaring the variables and setting them as null handle for now, the only one that is a little different is the size variable. That’s instead going to be the size of the vertex data structure that we made at the start, but 3 of them. Do the same for the index buffer but make it for 3 32 bit integers instead.

As for the last 3 parameters you want, the buffer usage would be VK\_BUFFER\_USAGE\_UNIFORM\_BUFFER\_BIT | VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT and I don’t know why so I’ll come back to this.

As for the sharing mode that’s set to exclusive because we don’t need to worry about other queue families. And the property flags are set to VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT this allows for host access in the next paragraph.

Next we need to get a pointer to the vertex memory we just set up, so we need a result variable getting the value out of the vkMapMemory() function with pretty self-explanatory parameters. The last parameter is going to be the returned pointer for the mapped memory so we’re going to pass in a void pointer. This function will map the GPU memory to the to a CPU accessible pointer, you can kinda just ignore the offset and the flags and set them to 0.

Next we can memcpy that data to the GPU so the destination would be the mapped pointer you just made, and the source would be the vertex data structure with the size being the buffer size variable.

The index buffer is honestly exactly the same minus the variable name changes.

### 7.3.2 | Command Buffer

Finally we just need to set up the command buffer

# 9 | Vertex Shader

## 9.1 | What exactly are we doing here

I didn’t start with the input assembler because that’s mostly just stored information like model data and stuff, so we’re going to go over the processing of it instead.

From the input assembler we get things like world position and normals, colours and various other bits of detail that lets us convert the position of a vertex to some pixels on the screen.

Some of this data will also be used in the fragment shader, which long story short needs this data when interpolating things like colours between and textured based off of the angles and positions of its vertices.

The vertex shader after going through all that processing will spit out a clip co-ordinate, and data for the fragment shader.

## 9.2 | Explaining clip co-ordinates

This is basically the frame buffer coordinates, clip here I think refers to the clipping plane. As for what we do with it, we have to normalize them, that way the frame buffer is mapped between -1 and 1 on both axes.

## 9.3 | Setting it up