Vulkan API Notes

Jaden Meyer

Chapter 1

How to Draw a Triangle

Where this is from

Step 1 of Triangle: Instance

Vulkan applications start by setting up the API through a `VkInstance`. This is created by describing your application and API extensions in use. You can query for Vulkan Supported hardware to select `VkPhysicalDevice`'s. This gets information like VRAM, and whatnot or to prefer a graphics card.

Step 2 of Triangle: Logical Device

After a device is chosen, a logical device ie a `VkDevice` needs to be created. Then you specifically describe which `VkPhysicalDeviceFeatures` you'll use (multi view port etc). Also need to specify which queue families to use. Most operations are asynchronously executed by sending them to a `VkQueue`. Queues taken from queue families in which each familiy supports specific sets of operations in the queues (graphics->compute/memory->etc).

Step 3 of Triangle: Window Surface, Swap Chain

Use GLFW (graphics library framework) to create a window. Note there are other APIs or libraries for this.

We also need a window surface `VkSurfaceKHR` and a swap chain `VkSwapchainKHR`. KHR means its part of the Vulkan API

The window surface is a cross-platform abstraction over the windows to render to. Instaniated by providing a reference to the native window handle. Most of this is done through a de-abstraction layer through GLFW.

The swap chain is a collection of render targets. Simply, ensures that the image we're rendering at the moment is different from the displayed one. In order to draw a frame, we aask the swap chain to provide us the next image. After we are done with the frame, we return it to the swap chain to be presented to the screen. To note, number of render targets and conditions for presenting images to screen depends on the present mode. These are like vsync (double buffering) and triple buffering.

A minor aside. Some platforms allow direct renders without any window manager through `VK_KHR_display` and `VK_KHR_display_swapchain` extensions. This is what would be used for your own window manager.

Step 4: Image Views and framebuffers

To draw something given to us from the swap chain, it must be wrapped into a `VkImageView` and `VkFramebuffer`.

A imageview references a specific part of an image to be used.

A framebuffer references imageviews that are to be used for color, depth and stencil targets.

Since the swap chain can contain a bunch of images, we make an imageview and a framebuffer for each and then select which one to use when we get to drawing.

Step 5: Render Pass

A render pass describes the type of images used during rendering operations. This includes information on how they are used, how their contents should be treated, etc. Since this chapter is about making a triangle, we will use a single image as color target and we want it to be set to a solid color before the drawing operation.

A big note here. A render pass only describes the images. A framebuffer actually is a binding contract for images to a slot.

Step 6: Graphics Pipeline

We set up a graphics pipeline by creating a `VkPipeline` object. This will describe the configurable state of the graphics card.

`VkShaderModule` objects are created to describe these configurable states such as viewport size, depth buffer operations and programmable states. These objects are created from Shader byte code.

In Vulkan, almost all of the configuration of the graphics pipeline is done in advance. Ie. if you want to change shaders of something, you will need to recreate the pipeline. This also means that we will need to create a bunch of `VkPipeline` objects in advance. Only some basic ones can be changed dynamically (viewport size for example).

Step 7: Command Pools with Command Buffers

Since most objects need to be submitted to a queue in order to be executed by Vulkan, we first need to record them in a `VkCommandBuffer` before it can be sent to the queue. We allocate the command buffers through a `VkCommandPool` associated wihta specific queue family. In order for the triangle, we record a command buffer with these operations:

- Begin render pass
- Bind the graphics pipeline
- Draw 3 vertices
- End the render pass

Since the image in the framebuffer is dependent on which image the swap chain will give us, we need to record a command buffer for each possible image to select from at drawing time.

Step 8: Creating the Main Loop

The drawing commands have now been wrapped into a command buffer. This simplifies the main loop by a lot. Start by getting an image from the swap chain using `vkAcquireNextImageKHR`. We can now pick the corresponding command buffer for that image. Then we can execute it with `vkQueueSubmit`. Now we can send the image to the swap chain for presentation to the screen itself using `vkQueuePresentKHR`.

Operations that are submitted to the queue are done so asynchronously. Therefore synchronization objects like semaphores are needed to ensure a correct order of execution. The draw command buffer must wait for the image acquisition otherwise we will run into trouble trying to render to an image still being read to the screen. Thus meaning vkQueuePresentKHR must also wait for any rendering to be finished.

Summary and Outline

This is merely an overview of all the needed steps to make *just* a triangle. A full outline of everything:

- Create a `VkInstance`
- select a valid graphics card `VkPhysicalDevice`
- Create `VkDevice` and `VkQueue` for drawing and presentation
- Create a window, window surface, and swap chain
- Create a render pass specifying render targets and usage
- Create framebuffers for the render pass
- Set up the graphics pipeline
- Allocate and record a command buffer with the draw commands for every possible swap chain image
- Draw frames by acquiring images, submitting the right draw command buffer and returning the images back to the swap chain

Some Vulkan Overview

All of the Vulkan functions, enumerations, and structs are defined in `vulkan.h`. This of course is part of the Vulkan SDK.

Functions have a lowercase `vk` prefix. Enumerations and structs have `Vk` ↪ prefixes. Enumeration values have a `Vk_` prefix. Structs are often used to provide parameters to functions.

Many Vulkan Structures require an explicit specification of type in the `sType` ↪ member. The `pNext` member can point to an extension structure. For now, this will be left `nullptr`. Functions that create or destroy and object will have `VkAllocationCallbacks` parameter that allows you to use a custom allocator for driver memory, which again is going to be left `nullptr` for now.

Almost all functions will return a `VkResult`that either has:

1. `VK_SUCCESS`

2. error code

This will be looked at later.

Vulkan is a very high performance low driver overhead option. So its default error checks and capabilities are limited. This means a lot of crashes are inevitable.

This is what a validation layer is for. These inject special pieces of code between the API and the graphics driver to do things like running checks on function parameters and tracking memory management problems.

We will be skipping how to setup the SDK for the desktop environment.

Finally Some Programming

Let's start from scratch. This is a template to use:

```
//Vulkan Starter Code
#include <vulkan/vulkan.h>
#include <iostream>
#include <stdexcept>
#include <cstdlib>

class TriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
        cleanup();
    }
private:
    void initVulkan() {};
    void mainLoop() {};
    void cleanup() {};

};

int main() {
    TriangleApplication tri;

    try {
        tri.run();
    }
    catch(const std::exception& e) {
        std::err << e.what() << std::endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

We import the vulkan headers to get everything we need. The rest of the headers are mostly there for error detection and whatnot. In a simpleform, `initVulkan` is in charge of initiating the Vulkan objects. The `mainLoop` will do all the rendering and will iterate until we close the window. At this time, `cleanup` is called to deallocate any resources.

It is more standard to implement RAII for more industry level code. However, in order to get a better feel, we are going to be explicit about allocation and deallocation.

In Vulkan, an object is either create through a `vkCreateXXX` or is allocated through another object with `vkAllocateXXX`. Since this is C++ we need to destory these using `vkDestroyXXX` and `vkFreeXXX`. For different types of objects the parameters tend to vary, but they all share a `pAllocator` which is used to specify callbacks for a custom memory allocator. For basic purposes, just pass a nullptr.

Integrating the Window

Add the following lines of code:

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>


void run() {
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
}
private:
    void initWindow() {};
```

We first need to define the window in `initWindow()` by using `glfwInit()` ↪ . This will initialize the GLFW library. However, GLFW was created with OpenGL in mind so we must first tell it we are not using OpenGl through a hint. Another hint we will need is to tell it not to rescale on its own (this takes special care).

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

Now we can create actual window by adding a private member of `GLFWwindow* window`. If we put this all together the new `initWindow()` function and private members should look like:

```
private:
    GLFWwindow* window;
    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
        //now create the window
        window = glfwCreateWindow(WIDTH, HEIGHT,
                    "Window Name", nullptr, nullptr);
    }
```

One thing to note is that we are defining Height and Width as global constant variables. Now that everything here is setup, we have to add to the main loop so that it knows to keep the window up and running. Then we have to deallocate from memory and cleanup the rest. These are pretty self explanatory, so here is the code:

```
void mainLoop() {
    while(!glfwWindowShouldClose(window)) {
        gflwPollEvents();
    }
}

void cleanup() {
    glfwDestroyWindow(window);
    glfwTerminate();
}
```

This is the sort of bare minimum required to be able to render a window to the screen. There is one downside to this however. If you are on Wayland, the protocol requires a buffer to be instantiated before the window will popup. In my current setup I am using Arch Linux on Hyprland. So this code will work in the sense that it will run in the background, but will not render a window to the screen.

Using a Vulkan Instance

First let's think about what an instance is actually doing. This is the connection between your code and the actual Vulkan Library. Start by creating a `createInstance()` function inside the `initVulkan()` function. Then in the private members invoke the object with `VkInstance instance`. Now, the driver actually needs to know some information so we invoke one of the provided structs given to us by the Vulkan header: `VkApplicationInfo`. We then fill in the information as such:

```
VkApplicationInfo info{};
info.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
//tell it your application name here
info.pApplicationName = "Application Name";
info.applicationVersion = VK_MAKE_VERSION(1,0,0);
info.pEngineName = "NoEngine";
info.engineVersion = VK_MAKE_VERSION(1,0,0);
info.apiVersion = VK_API_VERSION_1_0;
```

A lot of this is pretty self-explanatory. Its basically just making a long list of variables. One important one to point out though is the `sType`. This is the enum tag to tell the Vulkan loader and driver that we are pointing out the specific struct. This is often paired with a `pNext` (or other name) pointer. This will tell the Vulkan loader that there is more information to link ie to chain multiple structs together.

We now have to fill in one more struct. This is not optional like the information struct was.

```
VkInstanceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_CREATE_INFO;
createInfo.pApplicationInfo = &info;
```

We now need to know what extensions to use to interface with the window system. GLFW has some of these functions built in. We can use these and pass them into the struct.

```
uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&
    ↪ glfwExtensionCount);

//define what global validation layers to enable
createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
createInfo.enabledLayerCount = 0;
```

This is everything needed to instantiated a Vulkan instance. We can finally now do a `vkCreateInstance` function call as such:

```
VkResult result = vkCreateInstance(&createInfo, nullptr,
    ↪ &instance);
```

This illuminates the general structure to follow in order to create objects in Vulkan:

- pointer to struct with creation information
- pointer to custom allocator callbacks
- pointer to variable storing the handle to the new object.

Most Vulkan functions return a `VkResult` so we either get an error code or a `VK_SUCCESS`. So rather than wasting memory storing the object, we can do a simple check in a if statement as such:

```
if(vkCreateInstance(&createInfo, nullptr, &instance) !=
    ↪ VK_SUCCESS) {
    throw std::runtime_error("failure making instance");
}
```

We need to now check for extension support. Since one of the possible error codes from the
`VkCreateInstance` is

`VK_ERROR_EXTENSION_NOT_PRESENT`, we need to do some thinking about what extensions we want to add. We can get a list of supported extensions before we create the instance with the `vkEnumerateInstanceExtensionProperties` function. It takes in a pointer correspondant to a variable that stores the number of extensions and an array of `VkExtensionProperties` which store the details of the extensions. This means we have a setup that follows these steps:

- get number of extensions
- allocate an array/vector to hold extension details
- query the extension details

So the code looks like:

```
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &
    ↪ extensionCount, nullptr);

std::vector<VkExtensionProperties> extensions(
    ↪ extensionCount);
vkEnumerateInstanceExtensionProperties(nullptr, &
    ↪ extensionCount, extensions.data());
```

Note that each `VkExtensionProperties` struct contains the same name and version of any extension. So one could loop over it if they wanted to. Otherwise, add this code to the `createInstance()` function.

Lastly, we just need to cleanup everything thus far. This means using the `cleanup()` function. Note we need to destroy the Instance before the program (window) terminates. Therefore we have code that looks like:

```
void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Validation Layers

The Vulkan API has very little error checking by default. Since you are required to be so explicit about every step in Vulkan, some things will fall through the cracks. This is where validation layers are helpful. They are optional systems to hook into a Vulkan function call to apply additional operations. some common ones are:

- Checking the values of parameters against the specification to detect misuse
- Tracking creation and destruction of objects to find resource leaks
- Checking thread safety by tracking the threads that calls originate from
- Logging every call and its parameters to the standard output
- Tracing Vulkan calls for profiling and replaying

Here is a sample validation layer:

```
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkInstance* instance) {
//notice pAllocator over the nullptr!
if (pCreateInfo == nullptr || instance == nullptr) {
    log("Null pointer passed to required parameter!");
    return VK_ERROR_INITIALIZATION_FAILED;
}

return real_vkCreateInstance(pCreateInfo, pAllocator,
   ↪ instance);
}
```

These can be freely stacked to include all debugging functionality you're interested in. These can be turned off for release builds as well. One thing to note is these validation layers can only be used if you have them installed on your system. There are two typesof validation layers: **instance** and **device specific**. Instance layers would only check calls related to global Vulkan objects like instances. The device specific would only check calls related to a specific GPU. Now, device specific is depreciated so they are no longer useful.

So how do we use these layers? We need to enable them by specifying their name– much like extensions. All useful standard validation is in a bundle we can call from the SDK known by: `VK_LAYER_KHRONOS_validation`. Lets add two variables to specify layers. We can turn them on or off using a C++ macro for debug mode or not.

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
    const bool enableValidationLayers = false;
#else
    const bool enableValidationLayers = true;
#endif
```

We now add a new function `checkValidationLayerSupport` checking if all the requested layers are available. We can list all available layers using the `vkEnumerateInstanceLayerProperties` function. This usage is identical to that of `vkEnumerateInstanceExtensionProperties` which was talked about in the previous section.

```
bool checkValidationLayerSupport() {
    uint32_t layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount,
        ↪ nullptr);

    std::vector<VkLayerProperties> availableLayers(
        ↪ layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount,
        ↪ availableLayers.data());

    return false;
}
```

Next check if all the layers in the validationLayers exist in the availableLayers list. (note might need to include a string library here).

```
for (const char* layerName : validationLayers) {
    bool layerFound = false;

    for (const auto& layerProperties : availableLayers) {
        if (strcmp(layerName, layerProperties.layerName)
            ↪ == 0) {
            layerFound = true;
            break;
        }
    }

    if (!layerFound) {
        return false;
    }
}

return true;
```

We can then use this function in the `createInstance`. This looks like:

```
void createInstance() {
    if (enableValidationLayers && !
        ↪ checkValidationLayerSupport()) {
        throw std::runtime_error("validation layers
            ↪ requested, but not available!");
    }

    ...
}
```

At this point, we should have some working code in debug mode. If there is an error occuring at this point, it would be a good idea to read up on the Vulkan FAQ page.

Since we changed some data, we should make sure our `createInfo` struct takes note of this if they are enabled.

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

Assuming the check was successful, `vkCreateInstance` should never return a `VK_ERROR_LAYER_NOT_PRESENT` error.

This is where we start to take it up a notch. We can provide an explicit callback in our program so that instead of sending debug messages to the standard output we can handle them personally. This allows more error filtering since not all errors are destructive.

WE ARE GOING TO SKIP THIS AND COME BACK TO IT LATER
ITS VERY LONG AND HARD, BUT I WANT TO
MOVE ON A LITTLE BIT
TO GETTING TO THE
ACTUAL DATA STREAM

Physical Devices and Queue Families

After initialization of the Vulkan library through a VkInstance, we actually need to choose which graphics card we want to use in the system. We obviously want to choose the one that is most compatiable with all the features. Start by adding a function called: `pickPhysicalDevice()` and call it from the `initVulkan` ↪ `()` function. The card we select get stored in a `VkPhysicalDevice` handle that we are going to add as a class member. This object gets destroyed through the `VkInstance` when it is destroyed thus no new cleanup. For now declare it as such:

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

We can also list the graphics cards similar to extensions. We also should throw an error if no device is found:

```
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount,
    ↪ nullptr);

if(deviceCount ==  0) {
    throw std::runtime_error("No GPU's found with Vulkan
        ↪ Support")
}
```

Assuming we found a device, we should allocate an array/vector to hold the information found in the handles.

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount,
    ↪ devices.data());
```

Now we need to check one thing. First check if the devices are suitable for the operations. We can do this in a simple boolean function and call it as we iterate over each device in the vector.

```
    bool isDeviceSuitable(VkPhyiscalDevice device) {
        return true;
    }
for (const auto& device : devices) {
    if(isDeviceSuitable(device)) {
        physicalDevice = device
        break;
    }
}
```

Unfortunately since Vulkan requires everything to be complicated, this means back device suitability checking needs to be more in depth (can't have isdevice-suitable return true). Start by querying some details of the device itself. These can be found using
`vkGetPhysicalDeviceProperties`.

If you wanted extra support for specifics like multi-viewport rendering 64 bit floats or texture compression, add the commented lines of code as well:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    VkPhysicalDeviceProperties deviceProperties;
    VkPhysicalDeviceFeatures deviceFeatures;
    /*
    vkGetPhysicalDeviceProperties(device, &
       ↪ deviceProperties);
    vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
    */

    return deviceProperties.deviceType ==
       ↪ VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU &&
       ↪ deviceFeatures.geometryShader
}
```

There is one drawback to this though. We are just picking the first available device. What if instead we scored each device on a scale so we pick the best performer like the dedicaed graphics (rtx gtx radeon etc) over the integrated, then fall back on integrated when needed.This can be done through:

```
    void pickPhysicalDevice() {
    ...

    // Use an ordered map to automatically sort
        ↪ candidates by increasing score
    std::multimap<int, VkPhysicalDevice> candidates;

    for (const auto& device : devices) {
        int score = rateDeviceSuitability(device);
        candidates.insert(std::make_pair(score, device));
    }

    // Check if the best candidate is suitable at all
    if (candidates.rbegin()->first > 0) {
        physicalDevice = candidates.rbegin()->second;
    } else {
        throw std::runtime_error("failed to find a
            ↪ suitable GPU!");
    }
}

int rateDeviceSuitability(VkPhysicalDevice device) {
    ...

    int score = 0;

    // Discrete GPUs have a significant performance
        ↪ advantage
    if (deviceProperties.deviceType ==
        ↪ VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
        score += 1000;
    }

    // Maximum possible size of textures affects graphics
        ↪  quality
    score += deviceProperties.limits.maxImageDimension2D;

    // Application can't function without geometry
        ↪ shaders
    if (!deviceFeatures.geometryShader) {
        return 0;
    }

    return score;
}
```

For the sake of drawing a triangle just keep the return true picking the first device.

We are now ready to talk about Queue families. If you remember from the start of the chapter, anything that requires drawing, uploading textures, or what have you requires commands to be submitted to a queue. There are of course a bunch of different types of queues and queue families each of which has a different set of commands. In order to find the ones that correspond to our device and the commands we want to use we are going to add a `findQueueFamilies` function that does this search for us. Since this is something that repeats a little bit, we can use a struct to store these .

```
struct QueueFamilyIndices {
    uint32_t graphicsFamily
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice
    ↪ device) {
    QueueFamilyIndices indices;
    //other logic here to populate struct
    return indices
}
```

As we look at this one question you might have is what if there is no queue family available. This implies we need a way to check if a particular queue family was found. This is where `#include<optional>` comes into play. It holds no value until you assign something to it. Then, you can query if it contains a value or not using the `has_value()` function. This means we can change the struct to look like:

```
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
};
```

We can now work more intensly on the `findQueueFamilies` function. We use another Vulkan built in function: `vkGetPhysicalDeviceQueueFamilyProperties` ↪ . The corresponding `VkQueueFamilyProperties` contains details about hte queue family and the types of operations that are supported and the number of queusthat can be created based on that family. In such we need to find at least one queue family supporting `VK_QUEUE_GRAPHICS_BIT`.

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &
    ↪ queueFamilyCount, nullptr);

int i = 0;

for(const auto& queueFamily : queueFamilies) {
    if(queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        indices.graphicsFamily = i;
    }

    i++;
}
```

We can now use this fancy queue lookup function inside of the `isDeviceSuitable` function to ensure the device can process the commands.

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device
        ↪ );

    return indices.graphicsFamily.has_value();
}
```

We should also add a generic check into the `QueueFamilyIndices` struct as such:

```
bool isComplete() {
    return graphicsFamily.has_value();
}
```

This also means you can exit early from the for loop adding an `if(indices.` `↪ isComplete()){break;}` check.

Logical Devices and Queues

After setting up a physical device, we need to set up a *logical device* to interface with it. The logical device is very similar to the instance creation but also describes what features we want to use with it. You can even create multiple logical devices fomr the same physical device if you have varying requirements.

We start by adding a new class member storing the logical device handle using `VkDevice device`. Then, we add a `createLogicalDevice` function that is valled inside `initVulkan`.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createLogicalDevice() {

}
```

Now we have to specify a bunch of details in structs again. The first of these will be `VkDeviceQueueCreateInfo`. This describes the number of queues we want for a single queue family. Again right now we only care about those with graphics capabilites.

```
QueueFamilyIndices indices = findQueueFamilies(
    ↪ physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo{};
queueCreateInfo.sType =
    ↪ VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily
    ↪ .value();
queueCreateInfo.queuecount = 1;
```

The drivers only allow you to create a small number of queues for each queue family. You won't really need more than one because you can create all the commad buffers on multiple threads and then submit them all at once on the main thread with a low-overhead call.

Vulkan allows you to assign priorites to queues in order to influence scheduling of command buffer execution with floating point numbers between 0.0 and 1.0. This is required even if there is only one queue!!!

```
float queuePriority = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriority;
```

We must now setup the device features we plan on using. We got these from `vkGetPhysicalDeviceFeatures` earlier (like geometry shaders). Right now we are going to just define it as `VK_FALSE` for the definition. So we have:

```
VkPhysicalDeviceFeatures deviceFeatures{};
```

We can now starting filling in `VkDeviceCreateInfo` to finish creating the logical device. We start by adding pointers to the queue creation info and device features structs.

```
VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.queueCreateInfoCount = 1;

createInfo.pEnabledFeatures = &deviceFeatures;
```

The rest of the information is similar to the `VkInstanceCreateInfo` struct requiring one to specify extensions and validation layers; However, these are device specific this time around.

One specifc example is the device specifc extension of `VK_KHR_swapchain` which allows presentation of rendered images from the device to the window. It is possible for a Vulkan device to only be able to do compute operations and thus lack the ability.

In the past, Vulkan had a difference between instance and device specific validation layers, now no more. So this means we have to have some way to compare to old versions. Ie `enabledLayerCount` and `ppEnabledLayerNames` insde of `VkDeviceCreateInfo` are ignored by modern Vulkan but should be set up for older versions.

```
createInfo.enabledExtensionCount = 0;

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

For now, we don't need any device specific extensions. So this means we are now ready to instantiate the logical device with a call to the `vkCreateDevice` function.

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr,
    ↪ &device) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical
        ↪ device!");
}
```

The parameters are the physical device to interface with. These includes the queue and usage info we just specified, the optional allocation callbacks pointer, and a pointer to a variable to store the logical device handle in. This call can return errors based on enabling non-existent extensions or specifying the desired usage of unsupported features.

Furthermore, we should call this in the cleanup function. Use the code: `vkDestroyDevice(device,nullptr);`

As a final note for this part, logical devices don't interact directly with any instances, which is why its not a part of the parameter of the destroy device.

We can now retrieve the queue handles. The queues are created automatically along with the logical device. However, we don't have the handle to interface with them yet. First add a class member to store a handle to the graphics queue: `VkQueue graphicsQueue;`.

These device queues are implicity cleaned up when the device is destroyed so we don't need anything to destroy in cleanup.

We can use the `vkGetDeviceQueue` function to retrieve queue handles for each queue family. The parameters for this are: the logical device, queue family, queue index, and the pointer to the variable to store the queue handle in. Since we are only creating a single queue in this family, we just use index 0.

```
vkGetDeviceQueue(device, indices.graphicsFamily.value(),
    ↪ 0, &graphicsQueue);
```

Window Surface

Since Vulkan cannot interface directly with the window system, we need to create an intermeditate layer between the API and the window systems. This means we need to use the WSI (window system integration) extensions. We start with `VK_KHR_surface`. This is what exposes a `VkSurfaceKHR` object representing an abtract type of surface to present rendered images to.

`VK_KHR_surface` extension is an instance level extension that is already included in `glfwGetRequiredInstanceExtensions`. The surface needs to be created AFTER the instance creation since it can actually influence physical device selection. These are really important for render targets and presentation which will be talked about in depth later.

Start by create a surface class member below the debug callback `VkSurfaceKHR surface;`. Though the object and its usage is platform agnostic, its creation isn't because it depends on window system details. Since we use glfw we don't need to worry about specific implementation on like windows or something.

make a `createSurface()` funciton under `setupDebugMessenger()` function in the class. We now use some glfw magic and call a `vkDestroySurfaceKHR()` inside the cleanup.

```
void createSurface() {
    if(glfwCreateWindowSurface(instance, window, nullptr,
        ↪  &surface) != VK_SUCCESS) {
        throw std::runtime_error("failed creating window
            ↪ surface");
    }
}

void cleanup() {
    //make sure to destroy surface before instance
    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestoryInstance(instance, nullptr);
}
```

Even though Vulkan implementation may support window integration, it doesn't mean every device supports it. This means going back to our `isDeviceSuitable` and making sure the device can actually present images to the created surface.

One thing to note is that it is actually possible that queue families supporting drawing commands and the ones supporting presentation might not overlap. This means editing `QueueFamilyIndices` to make sure we take into account that there could be a distinct presentation queue.

```
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    bool isComplete() {
        return graphicsFamily.has_value() &&
            ↪ presentFamily.has_value();
    }
};
```

We will now modify the `findQueueFamilies` function to look for a queue family that has the capability of presenting to our window surface. The function that checks this is `vkGetPhysicalDeviceSurfaceSupportKHR` which takes a physical device, queue family index and a surface as parameters. We now add a call to it in the same loop where `VK_QUEUE_GRAPHICS_BIT` is. Then add a check for the value of the boolean and store the presentation family queue index.

```
VkBool32 presentSupport = false;
vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface,
    ↪ &presentSupport);

if(presentSupport) {
    indices.presentFamily = i;
}
```

It is now time to actually create the presentation queue. The thing we need to do is modify the logical device creation to create the presentation queue and retrieve the `VkQueue` handle. Thus add a `VkQueue presentQueue` private member object. Now we have to have some `VkDeviceQueueCreateInfo` structs to create a queue from both families. One way to do this is by creating a set of all the unique queue families that are necessary for required queues.

```
#include <set>

...

QueueFamilyIndices indices = findQueueFamilies(
    ↪ physicalDevice);

std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies = {indices.
    ↪ graphicsFamily.value(), indices.presentFamily.value
    ↪ ()};

float queuePriority = 1.0f;
for (uint32_t queueFamily : uniqueQueueFamilies) {
    VkDeviceQueueCreateInfo queueCreateInfo{};
    queueCreateInfo.sType =
        ↪ VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = queueFamily;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &queuePriority;
    queueCreateInfos.push_back(queueCreateInfo);
}
```

Now modify the `VkDeviceCreateInfo` to point to the vector:

```
createInfo.queueCreateInfoCount = static_cast<uint32_t>(
    ↪ queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
```

Now if all the queue families are the same, then we only need to pass the index once. so now the last step we need to do is:

```
vkGetDeviceQueue(device, indices.presentFamily.value(),
    ↪ 0, &presentQueue);
```

Swap Chains

Vulkan does not have the concept of a "default framebuffer". This menas it requires an infrastructure that will own the buffers that we will render to before we visualize them on the screen. This is known as the swap chain and it MUST be create EXPLICITLY through Vulkan. This is basically a queue of images waiting to be presented to the screen. The general idea is to sync refresh rate with the swap chain presentation images.

Let's first check for swap chain support. Since not every GPU was made for graphics, and image presentation is tied to the window system, its not really part of core Vulkan and thus we have to enable `VK_KHR_swapchain` device extension after querying for the support.

We again extend `isDeviceSuitable` function to check if the extention is supported. We sort of already saw how to do this. To note, the Vulkan header actually provides a `VK_KHR_SWAPCHAIN_EXTENSION_NAME` macro defined as `VK_KHR_swapchain`.

```
const std::vector<const char*> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};
```

NOTE: this is a global level thing. Now create a new function
`checkDeviceExtensionSupport` called from
`isDeviceSuitable` as an additional check:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device
        ↪ );

    bool extensionsSupported =
        ↪ checkDeviceExtensionSupport(device);

    return indices.isComplete() && extensionsSupported;
}

bool checkDeviceExtensionSupport(VkPhysicalDevice device)
    ↪  {
    return true;
}
```

Obviously returning true is not always idea (I mean what if it is actually false??). Ok let's change the function:

```
bool checkDeviceExtensionSupport(VkPhysicalDevice device)
    ↪ {
    uint32_t extensionCount;
    vkEnumerateDeviceExtensionProperties(device, nullptr,
        ↪ &extensionCount, nullptr);

    std::vector<VkExtensionProperties>
        ↪ availableExtensions(extensionCount);
    vkEnumerateDeviceExtensionProperties(device, nullptr,
        ↪ &extensionCount, availableExtensions.data());

    std::set<std::string> requiredExtensions(
        ↪ deviceExtensions.begin(), deviceExtensions.end
        ↪ ());

    for (const auto& extension : availableExtensions) {
        requiredExtensions.erase(extension.extensionName)
            ↪ ;
    }

    return requiredExtensions.empty();
}
```

In this case we chose a set of strings for representing the unconfirmed required extensions. This way it is easy to tick them off while enumerating the sequence of available extensions.You should run your code **here** and note if your GPU is capable of creating a swap chain.

Of course using a swapchain requires enabling the `VK_KHR_swapchain` extension first. This just requires changing a little bit of the logical device creation structure. Don't forget to replace `createInfo.enabledExtensionCount = 0;`.

```
createInfo.enabledExtensionCount = static_cast<uint32_t>(
    ↪ deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.
    ↪ data();

\\replace the createInfo.enabledExtensionCount = 0; line
    ↪ with this
```

We are now ready to query details of swap chain support. Obviously just checking support isn't enough because the window surface also has to work! Thus creating a swap chain involves a couple (lot) more settings inside the instance and device creation. There are three kinds of properties to check:

- Basic surface capatibilites (min/max height width, min/max #of images in swap chain)
- Surface formats (pixel format, color space)

- Available presentation modes

We will use a struct to pass these details around on query:

```
struct SwapChainSupportDetails {
    VkSurfaceCapabilitiesKHR capabilities;
    std::vector<VkSurfaceFormatKHR> formats;
    std::vector<VkPresentModeKHR> presentModes;
};
```

We should now make a new function and call it `querySwapChainSupport` to populate the struct:

```
SwapChainSupportDetails querySwapChainSupport(
    ↪ VkPhysicalDevice device) {
    SwapChainSupportDetails details;

    return details;
}
```

For now we just want to get the Triangle up and running and therefore we won't go into too much detail of what is happening here. So lets move onto some basic surface capabilites. These are simple to query and are returned into a single `VkSurfaceCapabilitiesKHR` struct.

```
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface
    ↪ , &details.capabilities);
```

This function takes the specified VkPhysicalDevice and VkSurfaceKHR window surface into account when determining supported capabilities. All the support querying functions have these two as first parameters because they are the core components of the swap chain.

We now are querying the supported surface formats. Because this is a list of structs, it follows the familiar ritual of 2 function calls:

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &
    ↪ formatCount, nullptr);

if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface,
        ↪ &formatCount, details.formats.data());
}
```

Make sure the vector is resized to hold all the available formats. Finally, querying the supported presentation modes works exactly the same way with `vkGetPhysicalDeviceSurfacePresentModesKHR`

```
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface
    ↪ , &presentModeCount, nullptr);

if (presentModeCount != 0) {
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device,
        ↪ surface, &presentModeCount, details.
        ↪ presentModes.data());
}
```

We are now done with filling in the details of the struct now. This means we can now go to extending `isDeviceSuitable` once again. This allows us to utilize the function the verify that the swap chain support is adequate. Just the support alone is good enough for drawing a triangle if there is at least one supported image format and one supported presentation mode.

```
bool swapChainAdequate = false;
if (extensionsSupported) {
    SwapChainSupportDetails swapChainSupport =
        ↪ querySwapChainSupport(device);
    swapChainAdequate = !swapChainSupport.formats.empty()
        ↪  && !swapChainSupport.presentModes.empty();
    //need to make sure to only try to query for swap
        ↪ chain support
    //after verifying extension is available so we change
        ↪  the last line (return) to:

    return indices.isComplete() && extensionsSupported &&
        ↪  swapChainAdequate;
}
```

It is now time to make sure we choose the right settings for the swap chain. If the `swapChainAdequate` conditions were met then the support is definitely sufficent, but there's always the possibility different modes of varying optimality exist. We write some functions to handle three types of settings:

- Surface format
- Presentation Mode
- Swap extent

Each of these we set so that we have an ideal value in mind we will go with if it is available otherwise we create some logic for the next best thing.

Starting with *Surface Format*. The function for this setting starts out simple. We will later pass the `formats` member of the `swapChainSupportDetails` struct as an argument.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::
    ↪ vector<VkSurfaceFormatKHR>& availableFormats) {


}
```

every `VkSurfaceFormatKHR` entry contains a `format` and a `colorSpace` member. The `format` member specifies the color channels and type.

Let's break down the colors a bit. `VK_FORMAT_B8G8R8A8_SRGB`. This means that we stor the B, G, R and alpha channels in that specific order with an 8 bit unsigned int for a total of 32 bits per pixel.

The `colorSpace` member indicates if the SRGB color space is support or not. This is checked by using this flag: `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`. For the color space we will use SRGB if it is available because it gives good results. It is also pretty much the standard for textures. Now lets check if the preferred combination is available. In most cases it is just ok to settle for the first format that is specified.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::
    ↪ vector<VkSurfaceFormatKHR>& availableFormats) {
    for (const auto& availableFormat : availableFormats)
        ↪ {
        if (availableFormat.format ==
            ↪ VK_FORMAT_B8G8R8A8_SRGB && availableFormat.
            ↪ colorSpace ==
            ↪ VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
            return availableFormat;
        }
    }

    return availableFormats[0];
}
```

Onto *Presentation Mode*. This is one of the most important parts of setting up the swap chain. There are 4 possible modes available in Vulkan:

1 `VK_PRESENT_MODE_IMMEDIATE_KHR` images submittd by the application are transferred to the screen asap (possible tearing)

2 `VK_PRESENT_MODE_FIFO_KHR` The swap chain is a queue in which the display takes an image from the front when the display is refreshed and then the

program inserts rendered images at the back. If the queue is full then the program will wait (vertical sync). When the display is refreshed this is the "vertical blank"

3 `VK_PRESENT_MODE_FIFO_RELAXED_KHR` This mode only differs from the previous if the application is late and the queue was empty at the last vertical blank. The image is transferred immediatley when it finally arrives. This may show tearing.

4 `VK_PRESENT_MODE_MAILBOX_KHR` Antoher variation of (2). Instead of blocking the application when the queue is full, images in the queue are simply replaced with the newer ones. This can be used to render frames as fast as possible while still avoiding tearing this is known as "triple buffering", but does not necessarily mean the framerate is unlocked. This results in fewer latency issues than vsync.

UNFORTUNATELY, only
`VK_PRESENT_MODE_FIFO_KHR` is guranteed to be availableso we have to write yet another function to get the best mode available.

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<
    ↪ VkPresentModeKHR>& availablePresentModes) {
    for (const auto& availablePresentMode :
        ↪ availablePresentModes) {
        if (availablePresentMode ==
            ↪ VK_PRESENT_MODE_MAILBOX_KHR) {
            return availablePresentMode;
        }
    }

    return VK_PRESENT_MODE_FIFO_KHR;
}
```

Note if the mailbox mode is chosen it can take more energy/battery to run so change this if it is a concern!

We have finally gotten to *Swap Extent* In here we will implement:

```
VkExtent2D chooseSwapExtent(const
    ↪ VkSurfaceCapabilitiesKHR& capabilities) {

}
```

The swap extent is basically the resolution of the swap chain images and is almost always equal ot the resolution of the window we're drawing to in pixels. The range of the possible resolutions is defined in the
`VkSurfaceCapabilitiesKHR` structure. Vulkan wants us to match the resolution of the window by setting width and height in the `currentExtent` member. Some

window managers allow us to differ by setting the height and width to the `uint32_t` max. If so just pick the best value between the lowest and highest bounds.

GLFW uses two units when looking at sizes: pixels and screen coords. Vulkan works in pixels which means the swap extent also has to be defined in pixels. However if the display is high enough DPI, screen coordinates don't correspond to pixels because of this it shows that the resolution of the window in pixel will be larger than the resolution in screen coordinates. So that means if Vulkan doesn't recognize this, we can't use the orignal HEIGHT and WIDTH variables we chose. Instead, we must use glfwGetFramebufferSize to query the resolution of the window in pixel before matching it against the min and max image extent.

```
#include <cstdint> // Necessary for uint32_t
#include <limits> // Necessary for std::numeric_limits
#include <algorithm> // Necessary for std::clamp

...

VkExtent2D chooseSwapExtent(const
    ↪ VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != std::
        ↪ numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    } else {
        int width, height;
        glfwGetFramebufferSize(window, &width, &height);

        VkExtent2D actualExtent = {
            static_cast<uint32_t>(width),
            static_cast<uint32_t>(height)
        };

        actualExtent.width = std::clamp(actualExtent.
            ↪ width, capabilities.minImageExtent.width,
            ↪ capabilities.maxImageExtent.width);
        actualExtent.height = std::clamp(actualExtent.
            ↪ height, capabilities.minImageExtent.height,
            ↪  capabilities.maxImageExtent.height);

        return actualExtent;
    }
}
```

The `clamp` function is used here to bound values of width and height between the allowed minimum and maximum extents supported by the implementation.

Back to creating the Swap chain now. Now that we have the helper functions we just wrote that help at runtime, we have all the information needed now to actually create a working swap chain. Create a `createSwapChain` function that starts out with the results of these calls and make srure to call it from the `initVulkan` after logical device creation.The function should look a little like:

```
void createSwapChain() {
    SwapChainSupportDetails swapChainSupport =
        ↪ querySwapChainSupport(physicalDevice);

    VkSurfaceFormatKHR surfaceFormat =
        ↪ chooseSwapSurfaceFormat(swapChainSupport.
        ↪ formats);
    VkPresentModeKHR presentMode = chooseSwapPresentMode(
        ↪ swapChainSupport.presentModes);
    VkExtent2D extent = chooseSwapExtent(swapChainSupport
        ↪ .capabilities);
}
```

We have to now define how many images we would like to have in the swap chain. Create a `uint32_t imageCount`. We could set it to the minimum swapChain capability with `swapChainSupport.capabilities.minImageCount ↪ ;`. But sticking to the mimimum means we might have to wait on the driver to complete internal operations before we can acquire another image to render to. Therefore its recommended to add at least one more image (+1). We also need to add a check to make sure we don't go over the maximum. Note that 0 is a special value meaning there is no maximum:

```
if (swapChainSupport.capabilities.maxImageCount > 0 &&
    ↪ imageCount > swapChainSupport.capabilities.
    ↪ maxImageCount) {
    imageCount = swapChainSupport.capabilities.
        ↪ maxImageCount;
}
```

As with Vulkan Objects, creating the swap chain object requires filling in a large structure but it is very familiar. But also after specifying which surface the swap chain should be tied to, details of the swap chain image are specified:

```
VkSwapchainCreateInfoKHR createInfo{};
createInfo.sType =
    ↪ VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = surface;
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1;
createInfo.imageUsage =
    ↪ VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

The `imageArrayLayers` specifies the amount of layers each image consists of. This should always be 1 unless you are developing a stereoscopic 3D application. The `imageUsage` bit field specifies what kind of operations we will use the images in the swap chain for. Here we are just going to render them directly meaning they are used as color attachment. It is also possible to render images to a sepeate image first to perform operations like post-processing. In which case, we may use a value like: `VK_IMAGE_USAGE_TRANSFER_DST_BIT` instead and then use a memory operation to transfer the rendered image to a swap chain image.

```
QueueFamilyIndices indices = findQueueFamilies(
    ↪ physicalDevice);
uint32_t queueFamilyIndices[] = {indices.graphicsFamily.
    ↪ value(), indices.presentFamily.value()};

if (indices.graphicsFamily != indices.presentFamily) {
     createInfo.imageSharingMode =
         ↪ VK_SHARING_MODE_CONCURRENT;
     createInfo.queueFamilyIndexCount = 2;
     createInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
     createInfo.imageSharingMode =
         ↪ VK_SHARING_MODE_EXCLUSIVE;
     createInfo.queueFamilyIndexCount = 0; // Optional
     createInfo.pQueueFamilyIndices = nullptr; // Optional
}
```

We need to specify how to handle swap chain images that will be used across multiple queue families. That will be the case here if the graphics queue family is differing from the presentation queue. We will be drawing on the images in the swap chain from the graphics queue then submitting them to the presentation queue. There are two ways of handling images accessed from multiple queues.

- `VK_SHARING_MODE_EXCLUSIVE` Image is owned by one queue family at a time and ownership must be explicitly transferred before using it in another queue family. This offers best performance.

- `VK_SHARING_MODE_CONCURRENT` Images can be created across multiple queue families without explicit ownership transfers.

If the queue families differ, we will be using the concurrent mode to avoid having to do the ownership chapters (mentioned in the tutorial) just don't have the background yet. Concurrent mode requires one to specify in advance which queue families are shared using `queueFamilyIndexCount` and `pQueueFamilyIndices` parameters. If the graphics queue family and presentation queue family are the same we should stick to exclusive mode since concurrent requires specification of at least two distinct queue families.

```
createInfo.preTransform = swapChainSupport.capabilities.
    ↪ currentTransform;
```

We can also specify that a certain transform should be applied to images in the swap chainif it is supported (`supportedTransforms` in `capabilities`), like a rotation or horizontal flip. To specify you don't want any transformation simply specify the current transformation:

```
createInfo.compositeAlpha =
    ↪ VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

The `compositeAlpha` field specifies if the alpha channel should be used for blending with other windows in the window system. We almost always want to ignore this because wehave: `VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR`.

```
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;
```

The `presentMode` member speaks for itself. If the `clipped` member is set to `VK_TRUE` then it means we do not care about the color of the pixels that are obscured. We will get the best performance by enabling clipping:

```
createInfo.oldSwapchain = VK_NULL_HANDLE;
```

This leaves us with one last field `oldSwapChain`. With Vulkan, it is possible that your swap chain be invalid or unoptimized while the application is running. This is a very complex thing to deal with and will be talked about much much later. Add in a class member to store the `VkSwapchainKHR` object. Now creating the swap chain is as simple as calling
`VkSwapchainKHR`

```
if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &
    ↪ swapChain) != VK_SUCCESS) {
    throw std::runtime_error("failed to create swap chain
        ↪ !");
}
```

the parameters are logical device, swap chain creating info, optional custom allocators, and a pointer to the variable to store the handle in. Then just add:

```
void cleanup() {
    vkDestroySwapchainKHR(device, swapChain, nullptr);
    ...
}
```

At this point it is a good idea to run the application to ensure that the swap chain is created properly. If something goes wrong it might be time to go to the FAQ. One thing to try is to remove `createInfo.imageExtent = extent;` with validation layers enabled so you can see it in action.

How do we now retrieve the swap chain images? We want these for when we do rendering oeprations. Add this vector to your private class members.

```
std::vector<VkImage> swapChainImages;
```

The images were created by the implemenation for the swap chain and they will be automatically cleaned up once the swap chain has been destroyed. In other words no need for any cleanup code! We are going to add the code at the end of `createSwapChain` function, right after the `vkCreateSwapchainKHR` call. Recall we only described a minimum number of images so we will first query the final number of images with `vkGetSwapchainImagesKHR` then resize the container then call it again to get the handles.

```
vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
    ↪ nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
    ↪ swapChainImages.data());
```

We should also store the format and extent we chose for the swap chain images in member variables as we will need them later.

```
VkSwapchainKHR swapChain;
std::vector<VkImage> swapChainImages;
VkFormat swapChainImageFormat;
VkExtent2D swapChainExtent;

...

swapChainImageFormat = surfaceFormat.format;
swapChainExtent = extent;
```

We know have a set of images that can be drawn onto and be presented to the window. So we are getting there! We will now start to set up the images as render targets and then we start lookin into the graphics pipeline and drawing commands.

Image Views

To use any `VkImage`, including any inside the swap chain, in the render pipeline we have to create a `VkImageView` object. An image view is literally a view into an image. It describes how to access the image and which part of the image to access.

Here, we are going to write a `createImageViews` function that creates a basic image view for every image in the swap chain so we can use them as color targets later on. first add a vector to store the image views in:

```
std::vector<VkImageView> swapChainImageViews;
```

Create the `createImageViews` function and call it right after swap chain createion. (inside initVulkan())

The first thing we need to do is resize the list to fit all the image views we will be creating. Next, we set up the loop to go over all the swap chain images.

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());
    for (size_t i = 0; i < swapChainImages.size(); i++) {

    }
}
```

The parameters for image view creation are going to be specified in `VkImageViewCreateInfo`. The first parameters are easy. The `viewType` and `format` fields specify hwo the image data should be interpreted. The `viewType` parameter allows one to treat images as 1,2,3D textures and cube maps.

We will show all the code in a second, but lets keep talking about the variables. The `components` field allows one to swizzle the color channels around (google this one day).

The `subresourceRange` field describes what the image's purpose is and which part of the image should be accessed. Our images will be used a color tagers without mipmapping levels or multiple layers.

```
VkImageViewCreateInfo createInfo{};
createInfo.sType =
    ↪ VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
createInfo.image = swapChainImages[i];

createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
createInfo.format = swapChainImageFormat;

createInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;

createInfo.subresourceRange.aspectMask =
    ↪ VK_IMAGE_ASPECT_COLOR_BIT;
createInfo.subresourceRange.baseMipLevel = 0;
createInfo.subresourceRange.levelCount = 1;
createInfo.subresourceRange.baseArrayLayer = 0;
createInfo.subresourceRange.layerCount = 1;
```

If working on stereoscopic 3D applications you would create a swap chain with multiple layers. This means you could create multiple image views for each image representing the views for the left and right eyes by accessing the different layers.

We are now able to create the image view by calling `vkCreateImageView`:

```
if (vkCreateImageView(device, &createInfo, nullptr, &
    ↪ swapChainImageViews[i]) != VK_SUCCESS) {
    throw std::runtime_error("failed to create image
        ↪ views!");
}
```

This is now the first time we have to iterate over something in the cleanup function:

```
for (auto imageView : swapChainImageViews) {
    vkDestroyImageView(device, imageView, nullptr);
}
```

This is sufficient to start using an image as a texture, but it not enough to be used a s render target yet; this requires a framebuffer which requires setting up the graphics pipeline.

The Graphics Pipline

The graphics pipeline is the sequence of operations that take the vertices and textures of your meshes all the way to the pixels in the render targets. A small overview looks sort of like:

1. Vertex/index buffer
2. Input Assembler
3. Vertex Shader
4. Tessellation
5. Geometry Shader
6. Rasterization
7. Fragment Shader
8. Color Blending
9. Framebuffer

The Input Assembler is the collector of raw vertex data from the buffers we specify. It may use an index buffer to repeat elements without duplicating the vertex data.

The Vertex Shader runs for ever certex and applies transformations to turn vertex positions from model space to the screen space. Also passes per-vertex data down the pipeline.

The Tessellation Shaders allow one to subdivide geometry based on rules to increase the mesh quality. Often used to make surfaces likes brick walls and staircases look less flat.

The Geometry Shader is run on every primitive (triangle, lines, points, rays?) and discard it or output more primitives than came in. Similar, but more flexible, to tessellation. The performace is really not that great unless Intel integrated GPUs.

The Rasterization stage discretizes the primitives to fragments. These are the pixel fragments that fill on the framebuffer. Any fragments falling outside the screen are discarded and its attributes ouputted by the vertex shader are interpolated across the fragments.

The Fragment Shader is invoked for every fragment that survives and determines which framebuffers the fragments ar written to and which color and depth values. Does this through interpolated data from the vertex shader including things like texture coordinates and normals for lighting.

The Color Blending stage applies operations to mix different fragments that map to the same pixel in the framebuffer. Fragments can overwrite each other, add up or be mixed based on transparency.

Stages with green color are known as *fixed-function* stages. These allowd one to tweak their operations using parameters. Stages with orange color are `programmable` meaning we can upload your own code to the GPU to apply the exact operations we want. So we could use fragment shaders to implement things from texturing to ray tracers.

In Vulkan you can only create the pipeline from scratch especially if you want to change shaders or bind different framebuffers or change blend functions. So obviously the downside is having to make multiple pipelines.

Some of the programmable stages are optional based on what you intend to do. The tessellation and geometry stages can be disabled if you are only doing simple geometry. If you are only interested in depth values, you can disable the fragment shader (useful for shadow map generation).

We are going to implement the vertex and fragment shaders for the triangle. Start by making a `createGraphicsPipeline` function called after the `createImageViews` in `initVulkan`.

Shader Modules

Unlike other graphics APIs, shader code in Vulkan has to be specified in bytecode format. The format is called: SPIR-V. This itself is a format used to write graphics and compute shaders.

There exists some hope as Khronos group has a vendor-independent compiler that compiles GLSL to SPIR-V. This makes it way easier and it also allows us to include the compiler as a library to produce SPIR-V at runtime. One could use `glslangValidator.exe` to use the compiler directly, or we could use Google's `glslc.exe` which uses the same parameter format as GCC and Clang. Both are included in the Vulkan SDK.

GLSL is a shading language with a C-Style syntax. Each program as a main function to invoke for every object. GLSL uses global variables to handle input and output. The language includes features to aid in graphics programming like vector and matrix primitives. The vector type is called `vec` with a number indicating the size ie ($\texttt{vec3} = (0, 0, 0)$). It is possible to create a new vector from existing vectors like:

$$\texttt{vec3}(1.0, 2.0, 3.0).xy.$$

This would result in a `vec2`. Also to note the vector contructors can take in combinations of vector objects and scalar values (ie one element in the parameters can be a vector).

Vertex Shader

The vertex shader processes each incoming vertex. It takes the attributes as input. This includes position, color, normal and texture coordinates. The output is the final position in the clip coordinates and the attributes that need to be passed on to the fragment shader (like color and texture coords). These values are later interpolated over the fragments by the rasterizer producing a smooth gradient.

A clip coordinate is a four dimensional vector from the vertex shader subsequently turned into a normalized device coordinate by dividing the whole vector by the last component. These normalized coordinates are homogenous coordinates mapping a framebuffer to a $[-1, 1]$ by $[-1, 1]$ coordinage system.

In our example, we are just going to specify the positions of the three vertices. We can now directly output normalized device coordinates by outputting them as clip coordinates from the vertex shader with the last component set to 1. That way, the division to transform clip coordinates to normalized won't change anything.

Nomally, we would store these coordinates in the vertex buffer; However, creating a vertex buffer in Vulkan and filling it with data is not trivial so we are going to save it for later.

We are going to include the coordinates of everything directly in the shader (not recommended):

```
vec2 positions[3] = vec2[](
    vec2(0.0, -0.5),
    vec2(0.5, 0.5),
    vec2(-0.5, 0.5)
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0,
        ↪ 1.0);
}
```

Note that the main function is invoked for ever vertex. The built in `gl_VertexIndex` ↪ variable contains the index of the current vertex. This is usually an index into the vertex buffer, but since we hard-programmed this is what we got.

Fragment Shader

The triangle that is formed by the positions from the vertex shader fills an area on the screen with fragments. The shader is invoked on these fragments to produce a color and depth for the framebuffer. A simple red color for the entire triangle is:

```
layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

the `main` function is called for every fragment just like the vertex shader. Colors in GLSL are 4-component vectors with the R,G,B and alpha channels within the $[0, 1]$ range. Unlike `gl_Position` in the vertex shader, there is no built in variable to output a color for the current fragment. Thus we have to specify your own output variable for each framebuffer where the `layout(location = 0)` modifier specifies the index of the framebuffer.

Obviously make the whole triangle red isn't that cool. Let's do it per vertex. We have edit the shaders just a bit. We need to specify the color for the position.

```
vec3 colors[3] = vec3[](
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
);
```

We need to pass these per-vertex colors to the fragment shader so it can output their interpolated values to the framebuffer. We first add a output for color to the vertex shader and write to it in the `main` function.

```
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0,
        ↪ 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

We now need to add a matching input in the fragment shader.

```
layout(location = 0) in vec3 fragColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

It is now time to compile the shaders. Create a directory called `shaders` in the root directory. Store the vertex shader in `shader.vert` and the fragment shader in a file called `frag`. The contents of each are the respective code we already wrote.

To compile these, create a `compile.sh` (for linux) file with:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.vert
    ↪ -o vert.spv
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.frag
    ↪ -o frag.spv
```

NOTE you want the path to be where you installed the Vulkan SDK. Make the script executable with: `chmod +x compile.sh`. The SDK has libshaderc to compile within the program.

We now have the ability to load shaders (yay!!). We will do this through `fstream`. Start with a helper function to load the binary data from the files. The `readFile` will read all the bytes from the specified file and return them in a byte array managed by the vector. We want to start reading at the end of the file using `std::ios::ate` and read as binary with `std::ios::binary`.

```
#include <fstream>
...

static std::vector<char> readFile(const std::string&
    ↪ filename) {
    std::ifstream file(filename, std::ios::ate | std::ios
        ↪ ::binary);

    if (!file.is_open()) {
        throw std::runtime_error("failed to open file!");
    }
}
```

This gives us the advantage of being able to use the read position to determine the size of the file in order to allocate a buffer. We can then seek back to the beginning of the file and read all bytes at once then we just need to close the file and return the bytes.

```
size_t fileSize = (size_t) file.tellg();
std::vector<char> buffer(fileSize);

file.seekg(0);
file.read(buffer.data(), fileSize);

file.close();

return buffer;
```

Then call this from the `createGraphicsPipeline` function to load the bytecode.Make sure the shaders are loaded correctly by printing the size of the buffers and checking if they match the actual file size in bytes.

```
void createGraphicsPipeline() {
    auto vertShaderCode = readFile("shaders/vert.spv");
    auto fragShaderCode = readFile("shaders/frag.spv");
}
```

Before we can pass the code to the pipeline, we have to wrap it in a `VkShaderModule`
↪ object. We can do this with a helper function called `createShaderModule`
↪ . This will take in a buffer with bytecode as a parameter and thus create a `VkShaderModule` from it.

```
VkShaderModule createShaderModule(const std::vector<char
    ↪ >& code) {

}
```

Creating a shader module is simple since we only need to specify a pointer to the buffer with the bytecode nad length of it. This is specified in the `VkShaderModuleCreateInfo` structure. One catch though is that the size of the bytecode is in bytes, but the bytecode pointer is a `uint32_t` pointer instead of a char. Since this is stored in a vector we don't need to do as much worrying.

```
VkShaderModuleCreateInfo createInfo{};
createInfo.sType =
    ↪ VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
createInfo.codeSize = code.size();
createInfo.pCode = reinterpret_cast<const uint32_t*>(code
    ↪ .data());

VkShaderModule shaderModule;
if (vkCreateShaderModule(device, &createInfo, nullptr, &
    ↪ shaderModule) != VK_SUCCESS) {
    throw std::runtime_error("failed to create shader
        ↪ module!");
}

return shaderModule;
```

the parameters are the same as those in previous object creation functions: logical device, pointer to create info structure, optional pointer for custom allocators, and handle output variable.

The shader modules are simply a thin wrapper around the shader bytecode that we have already loaded form a file and the functions defined in it. The linking and compilation of the SPIR-V bytecode to machine code by the GPU doesn't happen until the pipeline is created. This means we can destroy the shader modules again as soon as the pipeline creation is finished so this is why we make them local variables in the `createGraphicsPipeline` instead of class members.

The cleanup should then happen at the end of the function by adding two calls to `vkDestroyShaderModule`. All the remaining code will be inserted before:

```
void createGraphicsPipeline() {
    auto vertShaderCode = readFile("shaders/vert.spv");
    auto fragShaderCode = readFile("shaders/frag.spv");

    VkShaderModule vertShaderModule = createShaderModule(
       ↪ vertShaderCode);
    VkShaderModule fragShaderModule = createShaderModule(
       ↪ fragShaderCode);

    vkDestroyShaderModule(device, fragShaderModule,
       ↪ nullptr);
    vkDestroyShaderModule(device, vertShaderModule,
       ↪ nullptr);
}
```

To actually use the shaders, we will need to assign thme to a specific pipeline stage through `VkPipelineShaderStageCreateInfo` structures as part of the actual pipeline creation process.

We start by filling in the strucutre for the vertex shader, again in the `createGraphicsPipeline` function:

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
vertShaderStageInfo.sType =
    ↪ VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
    ↪ ;
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
```

The first on sType step is to tell Vulkan in which pipeline stage the shader is going to be used. There is an enum value for each of the programmable stages:

```
vertShaderStageInfo.module = vertShaderModule;
vertShaderStageInfo.pName = "main";
```

The next two members specify the shader module with the code and then entrypoint is the fucntion to invoke. Ie its possible to combine multiple fragment shadres into a single shader module and use different entry points to differentiate between their behaviors. In our case `main`.

We could also use `pSpecializationInfo` which we won't use. We modify the strucutre to suit the fragment shader:

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
fragShaderStageInfo.sType =
    ↪ VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
    ↪ ;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";
```

Then define and array that contains these structs which we will use later.

```
VkPipelineShaderStageCreateInfo shaderStages[] = {
    ↪ vertShaderStageInfo, fragShaderStageInfo};
```

That pretty much sums up the programmable stages of the pipeline now onto the fixed-function stages.

Fixed Functions

In most graphics APIs they provide the default state for most of the stages of the graphics pipeline. In Vulkan you have to be explicit about most pipeline states as it will be baked into an inmutable pipeline state object.

Dynamic State

While most of the pipeline state needs to be baked into the pipeline state, a limited amount of the state can actually be changed without recreating the pipeline at draw time. If you want a dynamic state you have to fill in a `kPipelineDynamicStateCreateInfo` structure like:

```
std::vector<VkDynamicState> dynamicStates = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_SCISSOR
};

VkPipelineDynamicStateCreateInfo dynamicState{};
dynamicState.sType =
    ↪ VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO
    ↪ ;
dynamicState.dynamicStateCount = static_cast<uint32_t>(
    ↪ dynamicStates.size());
dynamicState.pDynamicStates = dynamicStates.data();
```

You would now ignore a lot of values and you are now required to specify the data at drawing time. This gives a more flexible setup and is very common for things like viewport and scissor state, which would result in a more complex setup for the pipeline state.

The `VkPipelineVertexInputStateCreateInfo` structure decribes the format of the vertex data that will be passed to the vertex shader. This is done in two ways.

- Bindings (spacings between data and wheter data is per-vertex or per-instance)

- Attribute Descriptions (type of attributes passed to vertex shader, which binding to load them from and at which offset)

Since we hard coded the vertex data directly, we will fill in the struct to specify that there is no vertex data to load for now.

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType =
    ↪ VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO
    ↪ ;
vertexInputInfo.vertexBindingDescriptionCount = 0;
vertexInputInfo.pVertexBindingDescriptions = nullptr; //
    ↪ Optional
vertexInputInfo.vertexAttributeDescriptionCount = 0;
vertexInputInfo.pVertexAttributeDescriptions = nullptr;
    ↪ // Optional
```

For the input assembly, use the `VkPipelineInputAssemblyStateCreateInfo` struct which describes two things. What kind of geometry will be drawn from the vertices and if primitve restart should be enabled. The former is part of the `topology` member and can have values like:

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` points from vertices
- `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` line from every 2 vertices without reuse
- `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` the end vertex of every line is used as a start vertex for the next line
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST` triangle from every 3 vertives without reuse
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP` the second and third vertex of every triangle are used as first two vertices of the next triangle

Normally, the vertices are loaded from the vertex buffer by index in sequential order, but with an *element buffer* you can specify the indices to use yourself. This allows optimizations like reusing vertices. If one uses `primitiveRestartEnable` ↪ and set it to `VK_TRUE`, then it is possible to break up lines and triangles in the `_STRIP` topology modules by using a special index. For now we will do:

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType =
    ↪ VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO
    ↪ ;
inputAssembly.topology =
    ↪ VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

A viewport is basically the region of the framebuffer that the output will be rendered to.This is almost always $(0, 0) \to (\text{width}, \text{height})$. Also the case for our purposes.

```
VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float) swapChainExtent.width;
viewport.height = (float) swapChainExtent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
```