Vulkan Tutorial

Alexander Overvoorde

May 2017

Contents

Introduction

About

This tutorial will teach you the basics of using the Vulkan graphics and compute API. Vulkan is a new API by the Khronos group (known for OpenGL) that provides a much better abstraction of modern graphics cards. This new interface allows you to better describe what your application intends to do, which can lead to better performance and less surprising driver behavior compared to existing APIs like OpenGL and Direct3D. The ideas behind Vulkan are similar to those of Direct3D 12 and Metal, but Vulkan has the advantage of being fully cross-platform and allows you to develop for Windows, Linux and Android at the same time.

However, the price you pay for these benefits is that you have to work with a significantly more verbose API. Every detail related to the graphics API needs to be set up from scratch by your application, including initial frame buffer creation and memory management for objects like buffers and texture images. The graphics driver will do a lot less hand holding, which means that you will have to do more work in your application to ensure correct behavior.

The takeaway message here is that Vulkan is not for everyone. It is targeted at programmers who are enthusiastic about high performance computer graphics, and are willing to put some work in. If you are more interested in game development, rather than computer graphics, then you may wish to stick to OpenGL or Direct3D, which will not be deprecated in favor of Vulkan anytime soon. Another alternative is to use an engine like Unreal Engine or Unity, which will be able to use Vulkan while exposing a much higher level API to you.

With that out of the way, let's cover some prerequisites for following this tutorial:

- A graphics card and driver compatible with Vulkan (NVIDIA, AMD, Intel)
- Experience with C++ (familiarity with RAII, initializer lists)
- A compiler compatible with C++11 (Visual Studio 2013+, GCC 4.8+)
- Some existing experience with 3D computer graphics

This tutorial will not assume knowledge of OpenGL or Direct3D concepts, but it does require you to know the basics of 3D computer graphics. It will not explain the math behind perspective projection, for example. See this online book for a great introduction of computer graphics concepts.

You can use C instead of C++ if you want, but you will have to use a different linear algebra library and you will be on your own in terms of code structuring. We will use C++ features like classes and RAII to organize logic and resource lifetimes.

Tutorial structure

We'll start with an overview of how Vulkan works and the work we'll have to do to get the first triangle on the screen. The purpose of all the smaller steps will make more sense after you've understood their basic role in the whole picture. Next, we'll set up the development environment with the Vulkan SDK, the GLM library for linear algebra operations and GLFW for window creation. The tutorial will cover how to set these up on Windows with Visual Studio, and on Ubuntu Linux with GCC.

After that we'll implement all of the basic components of a Vulkan program that are necessary to render your first triangle. Each chapter will follow roughly the following structure:

- Introduce a new concept and its purpose
- Use all of the relevant API calls to integrate it into your program
- Abstract parts of it into helper functions

Although each chapter is written as a follow-up on the previous one, it is also possible to read the chapters as standalone articles introducing a certain Vulkan feature. That means that the site is also useful as a reference. All of the Vulkan functions and types are linked to the specification, so you can click them to learn more. Vulkan is a very new API, so there may be some shortcomings in the specification itself. You are encouraged to submit feedback to this Khronos repository.

As mentioned before, the Vulkan API has a rather verbose API with many parameters to give you maximum control over the graphics hardware. This causes basic operations like creating a texture to take a lot of steps that have to be repeated every time. Therefore we'll be creating our own collection of helper functions throughout the tutorial.

Every chapter will also conclude with a link to the full code listing up to that point. You can refer to it if you have any doubts about the structure of the code, or if you're dealing with a bug and want to compare. All of the code files have been tested on graphics cards from multiple vendors to verify correctness. Each chapter also has a comment section at the end where you can ask any questions that are relevant to the specific subject matter. Please specify your platform, driver version, source code, expected behavior and actual behavior to help us help you.

This tutorial is intended to be a community effort. Vulkan is still a very new API and best practices have not really been established yet. If you have any type of feedback on the tutorial and site itself, then please don't hesitate to submit an issue or pull request to the GitHub repository.

After you've gone through the ritual of drawing your very first Vulkan powered triangle onscreen, we'll start expanding the program to include linear transformations, textures and 3D models.

If you've played with graphics APIs before, then you'll know that there can be a lot of steps until the first geometry shows up on the screen. There are many of these initial steps in Vulkan, but you'll see that each of the individual steps is easy to understand and does not feel redundant. It's also important to keep in mind that once you have that boring looking triangle, drawing fully textured 3D models does not take that much extra work, and each step beyond that point is much more rewarding.

Ready to dive into the future of high performance graphics APIs? Let's go!

Overview

This chapter will start off with an introduction of Vulkan and the problems it addresses. After that we're going to look at the ingredients that are required for the first triangle. This will give you a big picture to place each of the subsequent chapters in. We will conclude by covering the structure of the Vulkan API and the general usage patterns.

Origin of Vulkan

Just like the previous graphics APIs, Vulkan is designed as a cross-platform abstraction over GPUs. The problem with most of these APIs is that the era in which they were designed featured graphics hardware that was mostly limited to configurable fixed functionality. Programmers had to provide the vertex data in a standard format and were at the mercy of the GPU manufacturers with regards to lighting and shading options.

As graphics card architectures matured, they started offering more and more programmable functionality. All this new functionality had to be integrated with the existing APIs somehow. This resulted in less than ideal abstractions and a lot of guesswork on the graphics driver side to map the programmer's intent to the modern graphics architectures. That's why there are so many driver updates for improving the performance in games, sometimes by significant margins. Because of the complexity of these drivers, application developers also need to deal with inconsistencies between vendors, like the syntax that is accepted for shaders. Aside from these new features, the past decade also saw an influx of mobile devices with powerful graphics hardware. These mobile GPUs have different architectures based on their energy and space requirements. One such example is tiled rendering, which would benefit from improved performance by offering the programmer more control over this functionality. Another limitation originating from the age of these APIs is limited multi-threading support, which can result in a bottleneck on the CPU side.

Vulkan solves these problems by being designed from scratch for modern graphics architectures. It reduces driver overhead by allowing programmers to clearly specify their intent using a more verbose API, and allows multiple threads to create and submit commands in parallel. It reduces inconsistencies in shader compilation by switching to a standardized byte code format with a single compiler. Lastly, it acknowledges the general purpose processing capabilities of modern graphics cards by unifying the graphics and compute functionality into a single API.

What it takes to draw a triangle

We'll now look at an overview of all the steps it takes to render a triangle in a well-behaved Vulkan program. All of the concepts introduced here will be elaborated on in the next chapters. This is just to give you a big picture to relate all of the individual components to.

Step 1 - Instance and physical device selection

A Vulkan application starts by setting up the Vulkan API through a `VkInstance`. An instance is created by describing your application and any API extensions you will be using. After creating the instance, you can query for Vulkan supported hardware and select one or more `VkPhysicalDevice`s to use for operations. You can query for properties like VRAM size and device capabilities to select desired devices, for example to prefer using dedicated graphics cards.

Step 2 - Logical device and queue families

After selecting the right hardware device to use, you need to create a VkDevice (logical device), where you describe more specifically which VkPhysicalDeviceFeatures you will be using, like multi viewport rendering and 64 bit floats. You also need to specify which queue families you would like to use. Most operations performed with Vulkan, like draw commands and memory operations, are asynchronously executed by submitting them to a VkQueue. Queues are allocated from queue families, where each queue family supports a specific set of operations in its queues. For example, there could be separate queue families for graphics, compute and memory transfer operations. The availability of queue families could also be used as a distinguishing factor in physical device selection. It is possible for a device with Vulkan support to not offer any graphics functionality, however all graphics cards with Vulkan support today will generally support all queue operations that we're interested in.

Step 3 - Window surface and swap chain

Unless you're only interested in offscreen rendering, you will need to create a window to present rendered images to. Windows can be created with the native platform APIs or libraries like GLFW and SDL. We will be using GLFW in this tutorial, but more about that in the next chapter.

We need two more components to actually render to a window: a window surface (VkSurfaceKHR) and a swap chain (VkSwapChainKHR). Note the `KHR` postfix, which means that these objects are part of a Vulkan extension. The Vulkan API itself is completely platform agnostic, which is why we need to use the standardized WSI (Window System Interface) extension to interact with the window manager. The surface is a cross-platform abstraction over windows to render to and is generally instantiated by providing a reference to the native window handle, for example `HWND` on Windows. Luckily, the GLFW library has a built-in function to deal with the platform specific details of this.

The swap chain is a collection of render targets. Its basic purpose is to ensure that the image that we're currently rendering to is different from the one that is currently on the screen. This is important to make sure that only complete images are shown. Every time we want to draw a frame we have to ask the swap chain to provide us with an image to render to. When we've finished drawing a frame, the image is returned to the swap chain for it to be presented to the screen at some point. The number of render targets and conditions for presenting finished images to the screen depends on the present mode. Common present modes are double buffering (vsync) and triple buffering. We'll look into these in the swap chain creation chapter.

Step 4 - Image views and framebuffers

To draw to an image acquired from the swap chain, we have to wrap it into a VkImageView and VkFramebuffer. An image view references a specific part of an image to be used, and a framebuffer references image views that are to be used for color, depth and stencil targets. Because there could be many different images in the swap chain, we'll preemptively create an image view and framebuffer for each of them and select the right one at draw time.

Step 5 - Render passes

Render passes in Vulkan describe the type of images that are used during rendering operations, how they will be used, and how their contents should be treated. In our initial triangle rendering application, we'll tell Vulkan that we will use a single image as color target and that we want it to be cleared to a solid color right before the drawing operation. Whereas a render pass only describes the type of images, a VkFramebuffer actually binds specific images to these slots.

Step 6 - Graphics pipeline

The graphics pipeline in Vulkan is set up by creating a VkPipeline object. It describes the configurable state of the graphics card, like the viewport size and depth buffer operation and the programmable state using VkShaderModule objects. The VkShaderModule objects are created from shader byte code. The driver also needs to know which render targets will be used in the pipeline, which we specify by referencing the render pass.

One of the most distinctive features of Vulkan compared to existing APIs, is that almost all configuration of the graphics pipeline needs to be in advance. That means that if you want to switch to a different shader or slightly change your vertex layout, then you need to entirely recreate the graphics pipeline. That means that you will have to create many VkPipeline objects in advance for all the different combinations you need for your rendering operations. Only some basic configuration, like viewport size and clear color, can be changed dynamically. All of the state also needs to be described explicitly, there is no default color blend state, for example.

The good news is that because you're doing the equivalent of ahead-of-time compilation versus just-in-time compilation, there are more optimization opportunities for the driver and runtime performance is more predictable, because large state changes like switching to a different graphics pipeline are made very explicit.

Step 7 - Command pools and command buffers

As mentioned earlier, many of the operations in Vulkan that we want to execute, like drawing operations, need to be submitted to a queue. These operations first need to be recorded into a VkCommandBuffer before they can be submitted. These command buffers are allocated from a `VkCommandPool` that is associated with a specific queue family. To draw a simple triangle, we need to record a command buffer with the following operations:

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

Because the image in the framebuffer depends on which specific image the swap chain will give us, we need to record a command buffer for each possible image and select the right one at draw time. The alternative would be to record the command buffer again every frame, which is not as efficient.

Step 8 - Main loop

Now that the drawing commands have been wrapped into a command buffer, the main loop is quite straightforward. We first acquire an image from the swap chain with vkAcquireNextImageKHR. We can then select the appropriate command buffer for that image and execute it with vkQueueSubmit. Finally, we return the image to the swap chain for presentation to the screen with vkQueuePresentKHR.

Operations that are submitted to queues are executed asynchronously. Therefore we have to use synchronization objects like semaphores to ensure a correct order of execution. Execution of the draw command buffer must be set up to wait on image acquisition to finish, otherwise it may occur that we start rendering to an image that is still being read for presentation on the screen. The vkQueuePresentKHR call in turn needs to wait for rendering to be finished, for which we'll use a second semaphore that is signaled after rendering completes.

Summary

This whirlwind tour should give you a basic understanding of the work ahead for drawing the first triangle. A real-world program contains more steps, like allocating vertex buffers, creating uniform buffers and uploading texture images that will be covered in subsequent chapters, but we'll start simple because Vulkan has enough of a steep learning curve as it is. Note that we'll cheat a bit by initially embedding the vertex coordinates in the vertex shader instead of using a vertex buffer. That's because managing vertex buffers requires some familiarity with command buffers first.

So in short, to draw the first triangle we need to:

- Create a VkInstance
- Select a supported graphics card (VkPhysicalDevice)
- Create a VkDevice and VkQueue for drawing and presentation
- Create a window, window surface and swap chain
- Wrap the swap chain images into VkImageView
- Create a render pass that specifies the 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

It's a lot of steps, but the purpose of each individual step will be made very simple and clear in the upcoming chapters. If you're confused about the relation of a single step compared to the whole program, you should refer back to this chapter.

API concepts

This chapter will conclude with a short overview of how the Vulkan API is structured at a lower level.

Coding conventions

All of the Vulkan functions, enumerations and structs are defined in the `vulkan.h` header, which is included in the Vulkan SDK developed by LunarG. We'll look into installing this SDK in the next chapter.

Functions have a lower case `vk` prefix, types like enumerations and structs have a `Vk` prefix and enumeration values have a `VK_` prefix. The API heavily uses structs to provide parameters to functions. For example, object creation generally follows this pattern:

```cpp
VkXXXCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr;
createInfo.foo = ...;
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

Many structures in Vulkan require you to explicitly specify the type of structure in the `sType` member. The `pNext` member can point to an extension structure and will always be `nullptr` in this tutorial. Functions that create or destroy an object will have a VkAllocationCallbacks parameter that allows you to use a custom allocator for driver memory, which will also be left `nullptr` in this tutorial.

Almost all functions return a VkResult that is either `VK_SUCCESS` or an error code. The specification describes which error codes each function can return and what they mean.

Validation layers

As mentioned earlier, Vulkan is designed for high performance and low driver overhead. Therefore it will include very limited error checking and debugging capabilities by default. The driver will often crash instead of returning an error code if you do something wrong, or worse, it will appear to work on your graphics card and completely fail on others.

Vulkan allows you to enable extensive checks through a feature known as *validation layers*. Validation layers are pieces of code that can be inserted between the API and the graphics driver to do things like running extra checks on function parameters and tracking memory management problems. The nice thing is that you can enable them during development and then completely disable them when releasing your application for zero overhead. Anyone can write their own validation layers, but the Vulkan SDK by LunarG provides a standard set of validation layers that we'll be using in this tutorial. You also need to register a callback function to receive debug messages from the layers.

Because Vulkan is so explicit about every operation and the validation layers are so extensive, it can actually be a lot easier to find out why your screen is black compared to OpenGL and Direct3D!

There's only one more step before we'll start writing code and that's setting up the development environment. # Development environment

In this chapter we'll set up your environment for developing Vulkan applications and install some useful libraries. All of the tools we'll use, with the exception of the compiler, are compatible with both Windows and Linux, but the steps for installing them differ a bit, which is why they're described separately here.

Windows

If you're developing for Windows, then I will assume that you are using Visual Studio 2013 or 2015 to compile your code. The steps are the same for both versions, but the Vulkan SDK currently only includes debug symbols that are compatible with Visual Studio 2013. That isn't really a problem in practice, but it's something that you may wish to take into account.

Vulkan SDK

The most important component you'll need for developing Vulkan applications is the SDK. It includes the headers, standard validation layers, debugging tools and a loader for the Vulkan functions. The loader looks up the functions in the driver at runtime, similarly to GLEW for OpenGL - if you're familiar with that.

The SDK can be downloaded from the LunarG website using the buttons at the bottom of the page. You don't have to create an account, but it will give you access to some additional documentation that may be useful to you.

Figure 1:

Proceed through the installation and pay attention to the install location of the SDK. The first thing we'll do is verify that your graphics card and driver properly support Vulkan. Go to the directory where you installed the SDK, open the `Bin32` directory and run the `cube.exe` demo. You should see the following:

If you receive an error message then ensure that your drivers are up-to-date, include the Vulkan runtime and that your graphics card is supported. See the introduction chapter for links to drivers from the major vendors.

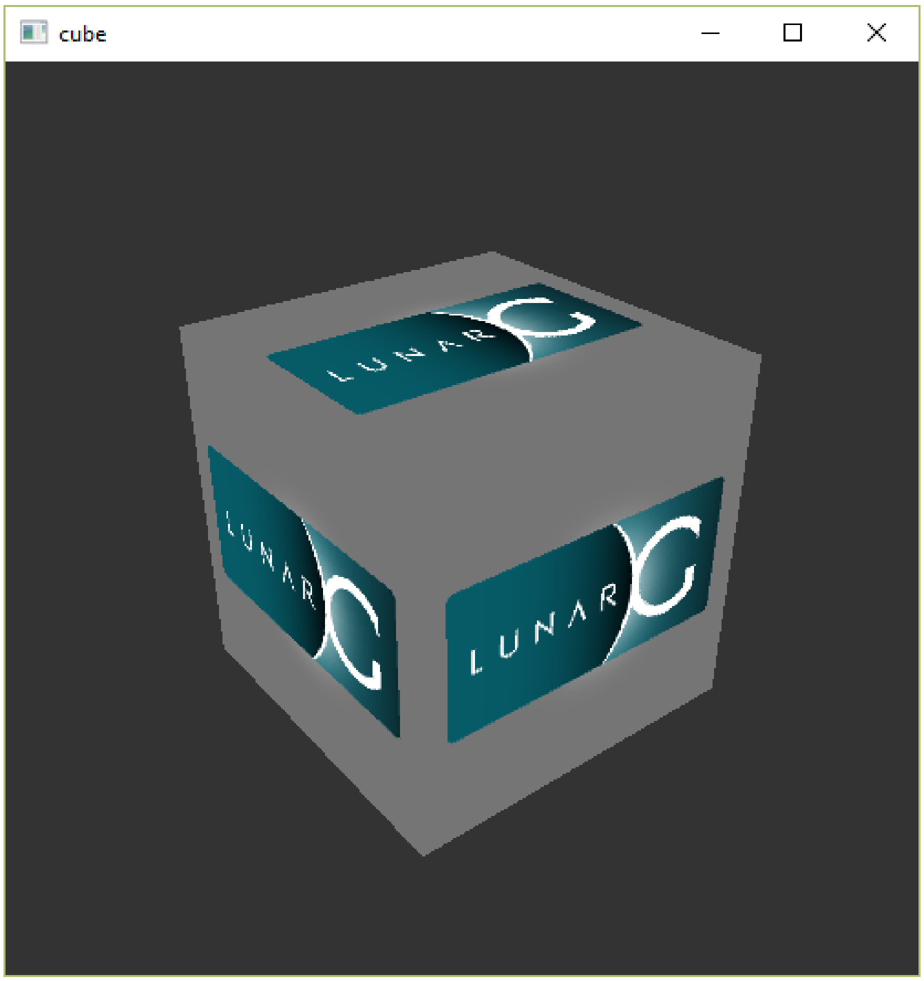

Figure 2:

There are two other programs in this directory that will be useful for development. The `vkjson_info.exe` program generates a JSON file with a detailed description of the capabilities of your hardware when using Vulkan. If you are wondering what support is like for extensions and other optional features among the graphics cards of your end users, then you can use this website to view the results of a wide range of GPUs.

The `glslangValidator.exe` program will be used to compile shaders from the human-readable GLSL to bytecode. We'll cover this in depth in the shader modules chapter. The `Bin32` directory also contains the binaries of the Vulkan loader and the validation layers, while the `Lib32` directory contains the libraries.

The `Doc` directory contains useful information about the Vulkan SDK and an offline version of the entire Vulkan specification. Lastly, there's the `Include` directory that contains the Vulkan headers. Feel free to explore the other files, but we won't need them for this tutorial.

GLFW

As mentioned before, Vulkan by itself is a platform agnostic API and does not include tools for creating a window to display the rendered results. To benefit from the cross-platform advantages of Vulkan and to avoid the horrors of Win32, we'll use the GLFW library to create a window, which supports both Windows and Linux. There are other libraries available for this purpose, like SDL, but the advantage of GLFW is that it also abstracts away some of the other platform-specific things in Vulkan besides just window creation.

You can find the latest release of GLFW on the official website. In this tutorial we'll be using the 32-bit binaries, but you can of course also choose to build in 64 bit mode. In that case make sure to link with the Vulkan SDK binaries in the `Bin` directory. After downloading it, extract the archive to a convenient location. I've chosen to create a `Libraries` directory in the Visual Studio directory under documents.

GLM

Unlike DirectX 12, Vulkan does not include a library for linear algebra operations, so we'll have to download one. GLM is a nice library that is designed for use with graphics APIs and is also commonly used with OpenGL.

GLM is a header-only library, so just download the latest version and store it in a convenient location. You should have a directory structure similar to the following now:

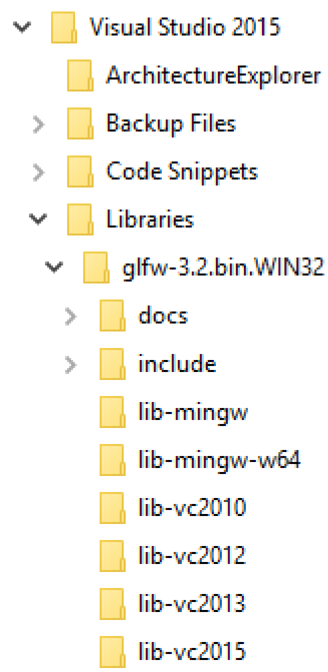
Visual Studio 2015
ArchitectureExplorer
Backup Files
Code Snippets
Libraries
glfw-3.2.bin.WIN32
docs
include
lib-mingw
lib-mingw-w64
lib-vc2010
lib-vc2012
lib-vc2013
lib-vc2015

Figure 3:

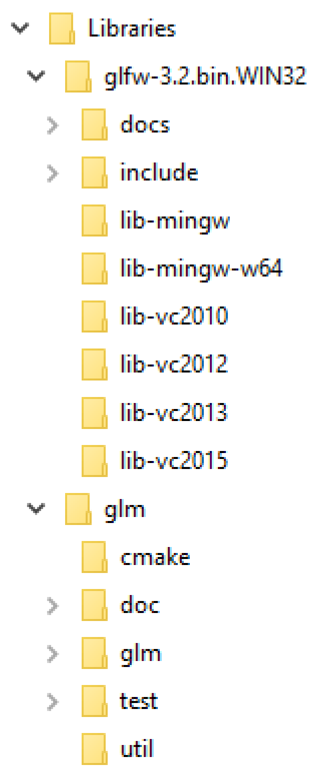
Libraries
glfw-3.2.bin.WIN32
docs
include
lib-mingw
lib-mingw-w64
lib-vc2010
lib-vc2012
lib-vc2013
lib-vc2015
glm
cmake
doc
glm
test
util

Figure 4:

Setting up Visual Studio

Now that you've installed all of the dependencies we can set up a basic Visual Studio project for Vulkan and write a little bit of code to make sure that everything works.

Start Visual Studio and create a new C++ Win32 project.

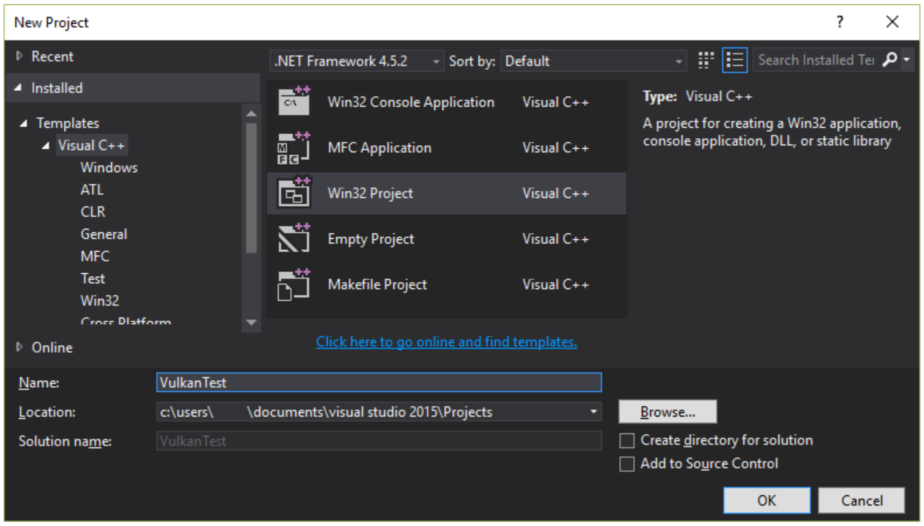

Figure 5:

Click `Next`, select `Console application` as application type and make sure that `Empty project` is checked.

Press `Finish` to create the project and add a C++ source file. You should already know how to do that, but the steps are included here for completeness.

Now add the following code to the file. Don't worry about trying to understand it right now; we're just making sure that you can compile and run Vulkan applications. We'll start from scratch in the next chapter.

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

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>
```

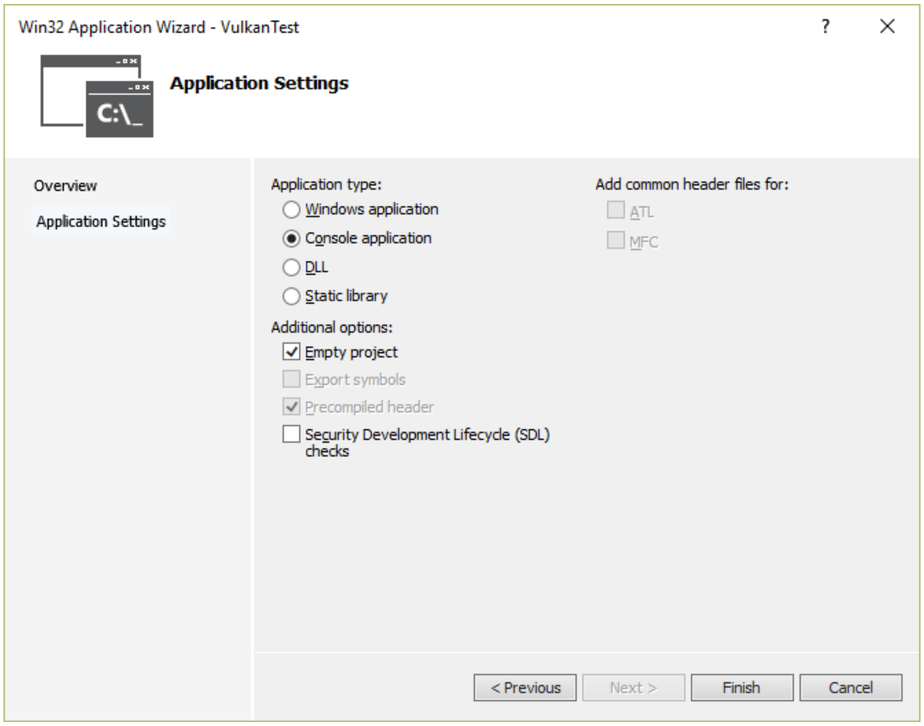

Figure 6:

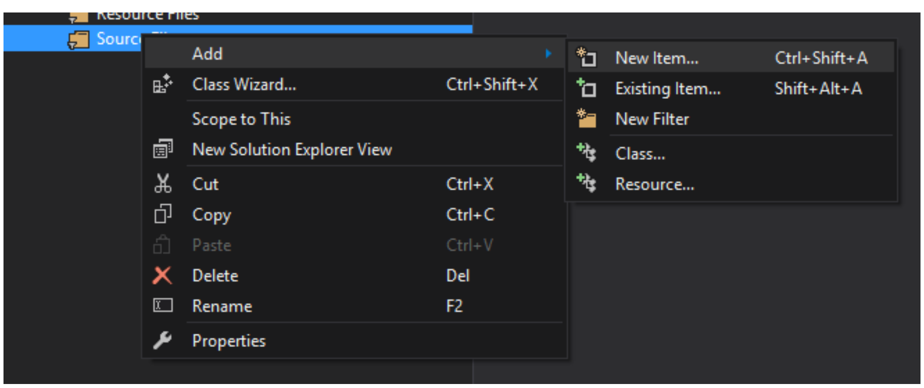

Figure 7:

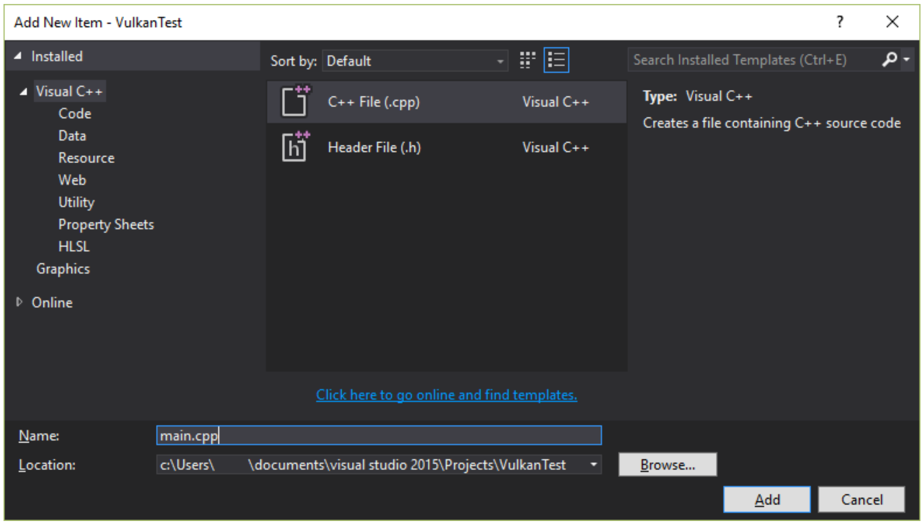

Figure 8:

```
int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

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

    std::cout << extensionCount << " extensions supported" << std::endl;

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
```

Let's now configure the project to get rid of the errors. Open the project properties dialog and ensure that `All Configurations` is selected, because most of the settings apply to both `Debug` and `Release` mode.

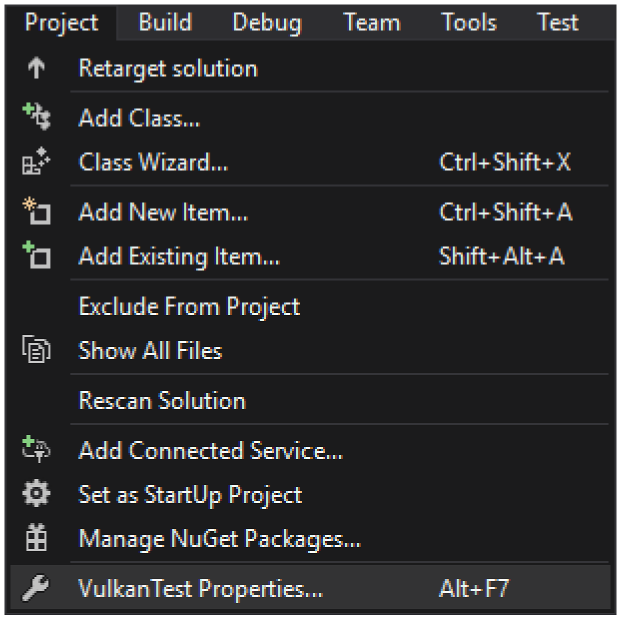

Figure 9:

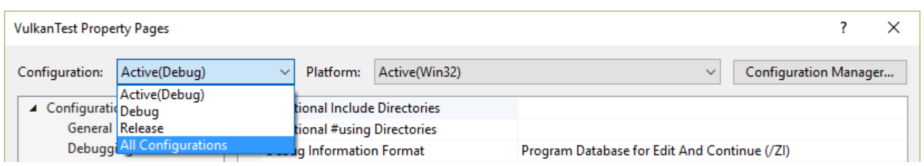

Figure 10:

Go to `C++ -> General -> Additional Include Directories` and press `<Edit...>` in the dropdown box.

Add the header directories for Vulkan, GLFW and GLM:

Next, open the editor for library directories:

And add the locations of the object files for Vulkan and GLFW:

Go to `Linker -> Input` and press `<Edit...>` in the `Additional Dependencies` dropdown box.

Enter the names of the Vulkan and GLFW object files:

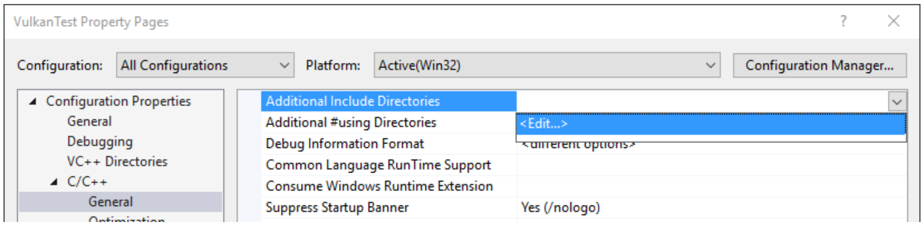

Figure 11:

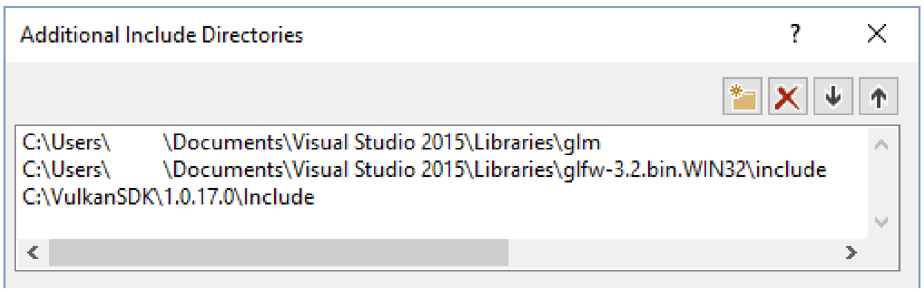

Figure 12:

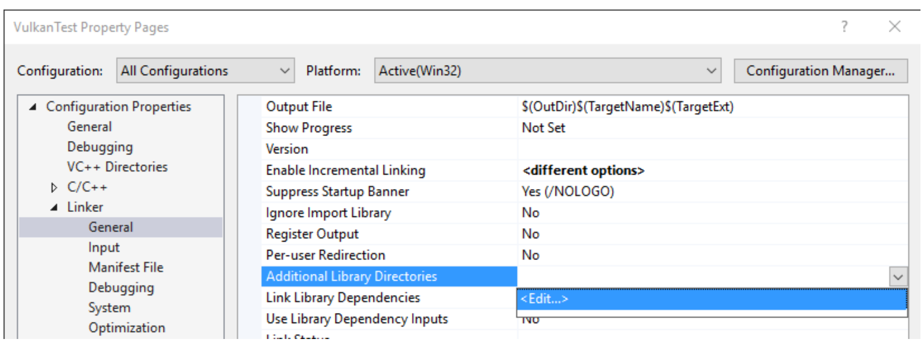

Figure 13:

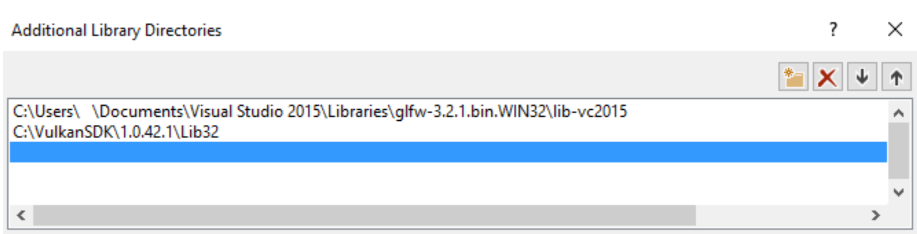

Figure 14:

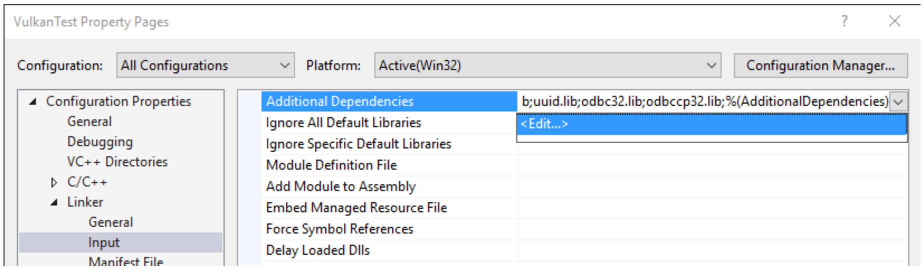

Figure 15:

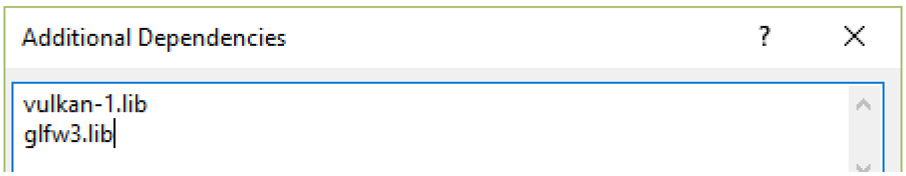

Figure 16:

You can now close the project properties dialog. If you did everything right then you should no longer see any more errors being highlighted in the code.

Press `F5` to compile and run the project and you should see a command prompt and a window pop up like this:

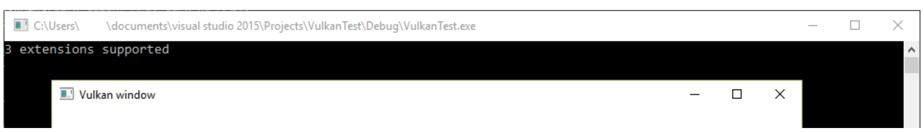

Figure 17:

The number of extensions should be non-zero. Congratulations, you're all set for playing with Vulkan!

To avoid having to repeat this work all over again every time, you can create a template from it. Select `File -> Export Template...`. Select `Project template` and fill in a nice name and description for the template.

Press `Finish` and you should now have a handy template in the `New Project` dialog! Use it to create a `Hello Triangle` project as preparation for the next chapter.

You are now all set for the real adventure.

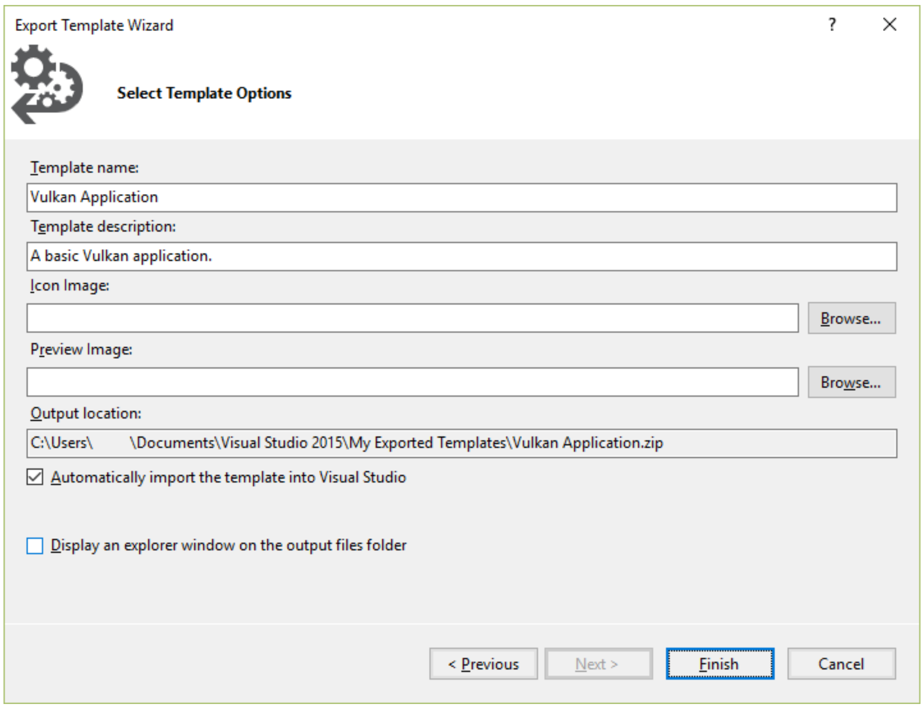
Export Template Wizard
Select Template Options
Template name:
Vulkan Application
Template description:
A basic Vulkan application.
Icon Image:
Browse...
Preview Image:
Browse...
Output location:
C:\Users\ \Documents\Visual Studio 2015\My Exported Templates\Vulkan Application.zip
Automatically import the template into Visual Studio
Display an explorer window on the output files folder
< Previous
Next >
Finish
Cancel

Figure 18:

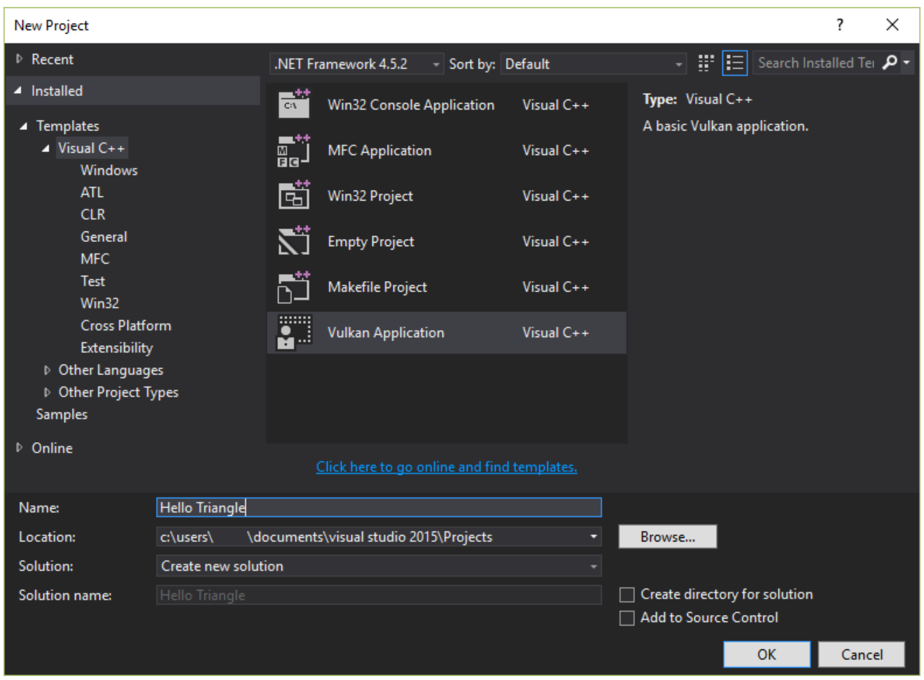
New Project
Recent
Installed
Templates
Visual C++
Windows
ATL
CLR
General
MFC
Test
Win32
Cross Platform
Extensibility
Other Languages
Other Project Types
Samples
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Te
Win32 Console Application
Visual C++
MFC Application
Visual C++
Win32 Project
Visual C++
Empty Project
Visual C++
Makefile Project
Visual C++
Vulkan Application
Visual C++
Type: Visual C++
A basic Vulkan application.
Click here to go online and find templates.
Name:
Hello Triangle
Location:
c:\users\ \documents\visual studio 2015\Projects
Browse...
Solution:
Create new solution
Solution name:
Hello Triangle
Create directory for solution
Add to Source Control
OK
Cancel

Figure 19:

Linux

These instructions will be aimed at Ubuntu users, but you may be able to follow along by compiling the LunarG SDK yourself and changing the `apt` commands to the package manager commands that are appropriate for you. You should already have a version of GCC installed that supports modern C++ (4.8 or later). You also need both CMake and make.

Vulkan SDK

The most important component you'll need for developing Vulkan applications is the SDK. It includes the headers, standard validation layers, debugging tools and a loader for the Vulkan functions. The loader looks up the functions in the driver at runtime, similarly to GLEW for OpenGL - if you're familiar with that.

The SDK can be downloaded from the LunarG website using the buttons at the bottom of the page. You don't have to create an account, but it will give you access to some additional documentation that may be useful to you.

Figure 20:

Open a terminal in the directory where you've downloaded the `.run` script, make it executable and run it:

```
chmod +x vulkansdk-linux-x86_64-xxx.run
./vulkansdk-linux-x86_64-xxx.run
```

It will extract all of the files in the SDK to a `VulkanSDK` subdirectory in the working directory. Move the `VulkanSDK` directory to a convenient place and take note of its path. Open a terminal in the root directory of the SDK, which will contain files like `build_examples.sh`.

The samples in the SDK and one of the libraries that you will later use for your program depend on the XCB library. This is a C library that is used to interface with the X Window System. It can be installed in Ubuntu from the libxcb1-dev package. You also need the generic X development files that come with the xorg-dev package.

```
sudo apt install libxcb1-dev xorg-dev
```

You can now build the Vulkan examples in the SDK by running:

```
./build_examples.sh
```

If compilation was successful, then you should now have a ./examples/build/cube executable. Run it from the examples/build directory with ./cube and ensure that you see the following pop up in a window:

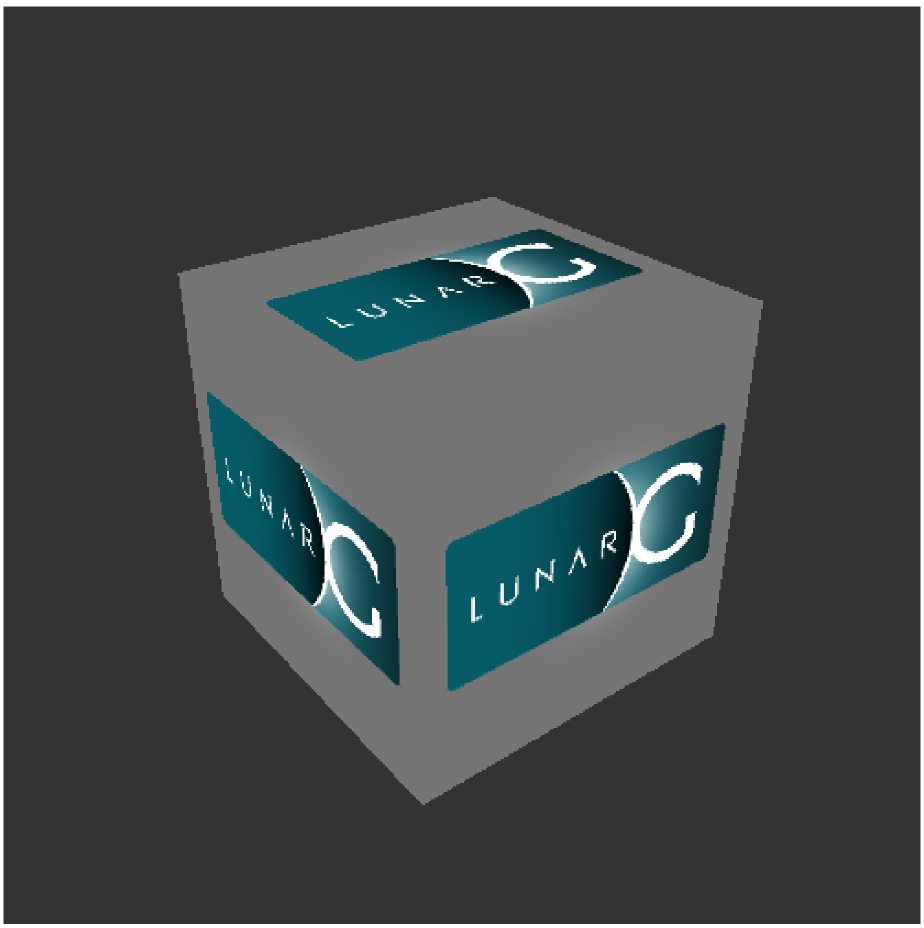

Figure 21:

If you receive an error message then ensure that your drivers are up-to-date, include the Vulkan runtime and that your graphics card is supported. See the introduction chapter for links to drivers from the major vendors.

GLFW

As mentioned before, Vulkan by itself is a platform agnostic API and does not include tools for creation a window to display the rendered results. To benefit from the cross-platform advantages of Vulkan and to avoid the horrors of X11, we'll use the GLFW library to create a window, which supports both Windows and Linux. There are other libraries available for this purpose, like SDL, but the advantage of GLFW is that it also abstracts away some of the other platform-specific things in Vulkan besides just window creation.

We'll be installing GLFW from source instead of using a package, because the Vulkan support requires a recent version. You can find the sources on the official website. Extract the source code to a convenient directory and open a terminal in the directory with files like `CMakeLists.txt`.

Run the following commands to generate a makefile and compile GLFW:

```
cmake .
make
```

You may see a warning stating `Could NOT find Vulkan`, but you can safely ignore this message. If compilation was successful, then you can install GLFW into the system libraries by running:

```
sudo make install
```

GLM

Unlike DirectX 12, Vulkan does not include a library for linear algebra operations, so we'll have to download one. GLM is a nice library that is designed for use with graphics APIs and is also commonly used with OpenGL.

It is a header-only library that can be installed from the `libglm-dev` package:

```
sudo apt install libglm-dev
```

Setting up a makefile project

Now that you have installed all of the dependencies, we can set up a basic makefile project for Vulkan and write a little bit of code to make sure that everything works.

Create a new directory at a convenient location with a name like `VulkanTest`. Create a source file called `main.cpp` and insert the following code. Don't worry about trying to understand it right now; we're just making sure that you can compile and run Vulkan applications. We'll start from scratch in the next chapter.

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

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

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

    std::cout << extensionCount << " extensions supported" << std::endl;

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
```

Next, we'll write a makefile to compile and run this basic Vulkan code. Create a new empty file called `Makefile`. I will assume that you already have some basic experience with makefiles, like how variables and rules work. If not, you can get up to speed very quickly with this tutorial.

We'll first define a couple of variables to simplify the remainder of the file. Define a `VULKAN_SDK_PATH` variable that refers to the location of the `x86_64` directory in the LunarG SDK, for example:

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
```

Next, define a `CFLAGS` variable that will specify the basic compiler flags:

```
CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include
```

We're going to use modern C++ (`-std=c++11` or `std=c++14`), and we need to be able to locate `vulkan.h` in the LunarG SDK.

Similarly, define the linker flags in a `LDFLAGS` variable:

```
LDFLAGS = -L$(VULKAN_SDK_PATH)/lib `pkg-config --static --libs glfw3` -lvulkan
```

The first flag specifies that we want to be able to find libraries like `libvulkan.so` in the LunarG SDK's `x86_64/lib` directory. The second component invokes `pkg-config` to automatically retrieve all of the linker flags necessary to build an application with GLFW. Finally, `-lvulkan` links with the Vulkan function loader that comes with the LunarG SDK.

Specifying the rule to compile `VulkanTest` is straightforward now. Make sure to use tabs for indentation instead of spaces.

```
VulkanTest: main.cpp
    g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)
```

Verify that this rule works by saving the makefile and running `make` in the directory with `main.cpp` and `Makefile`. This should result in a `VulkanTest` executable.

We'll now define two more rules, `test` and `clean`, where the former will run the executable and the latter will remove a built executable:

```
.PHONY: test clean

test: VulkanTest
    ./VulkanTest

clean:
    rm -f VulkanTest
```

You will find that `make clean` works perfectly fine, but `make test` will most likely fail with the following error message:

```
./VulkanTest: error while loading shared libraries: libvulkan.so.1: cannot open shared objec
```

That's because `libvulkan.so` is not installed as system library. To alleviate this problem, explicitly specify the library loading path using the `LD_LIBRARY_PATH` environment variable:

```
test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib ./VulkanTest
```

The program should now run successfully, and display the number of Vulkan extensions. The application should exit with the success return code (`0`) when you close the empty window. However, there is one more variable that you need to set. We will start using validation layers in Vulkan and you need to tell the Vulkan library where to load these from using the `VK_LAYER_PATH` variable:

```
test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib VK_LAYER_PATH=$(VULKAN_SDK_PATH)/etc/explicit_lay
```

You should now have a complete makefile that resembles the following:

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include
LDFLAGS = -L$(VULKAN_SDK_PATH)/lib `pkg-config --static --libs glfw3` -lvulkan

VulkanTest: main.cpp
    g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)

.PHONY: test clean

test: VulkanTest
    LD_LIBRARY_PATH=$(VULKAN_SDK_PATH)/lib VK_LAYER_PATH=$(VULKAN_SDK_PATH)/etc/explicit_lay

clean:
    rm -f VulkanTest
```

You can now use this directory as a template for your Vulkan projects. Make a copy, rename it to something like `HelloTriangle` and remove all of the code in `main.cpp`.

Before we move on, let's explore the Vulkan SDK a bit more. There are two programs in it that will be very useful for development. The `x86_64/bin/vkjson_info` program generates a JSON file with a detailed description of the capabilities of your hardware when using Vulkan. If you are wondering what support is like for extensions and other optional features among the graphics cards of your end users, then you can use this website to view the results of a wide range of GPUs. This program needs to be run with the same `LD_LIBRARY_PATH` variable as your own programs:

```
LD_LIBRARY_PATH=../lib ./vkjson_info
```

The `x86_64/bin/glslangValidator` program will be used to compile shaders from the human-readable GLSL to bytecode. We'll cover this in depth in the shader modules chapter. It does not depend on the Vulkan library.

The `Doc` directory contains useful information about the Vulkan SDK and an offline version of the entire Vulkan specification. Feel free to explore the other files, but we won't need them for this tutorial.

You are now all set for the real adventure.

Base code

General structure

In the previous chapter you've created a Vulkan project with all of the proper configuration and tested it with the sample code. In this chapter we're starting from scratch with the following code:

```
#include <vulkan/vulkan.h>

#include <iostream>
#include <stdexcept>
#include <functional>

class HelloTriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
    }

private:
    void initVulkan() {

    }

    void mainLoop() {

    }
};

int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::runtime_error& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

We first include the Vulkan header from the LunarG SDK, which provides the functions, structures and enumerations. The `stdexcept` and `iostream` headers are included for reporting and propagating errors. The `functional` headers will be used for a lambda functions in the resource management section.

The program itself is wrapped into a class where we'll store the Vulkan objects as private class members and add functions to initiate each of them, which will be called from the `initVulkan` function. Once everything has been prepared, we enter the main loop to start rendering frames. We'll fill in the `mainLoop` function to include a loop that iterates until the window is closed in a moment.

If any kind of fatal error occurs during execution then we'll throw a `std::runtime_error` exception with a descriptive message, which will propagate back to the `main` function and be printed to the command prompt. One example of an error that we will deal with soon is finding out that a certain required extension is not supported.

Roughly every chapter that follows after this one will add one new function that will be called from `initVulkan` and one or more new Vulkan objects to the private class members.

Resource management

You may have noticed that there's no cleanup function anywhere to be seen and that is intentional. Every Vulkan object needs to be destroyed with a function call when it's no longer needed, just like each chunk of memory allocated with `malloc` requires a call to `free`. Doing that manually is a lot of work and is very error-prone, but we can completely avoid that by taking advantage of the C++ RAII principle. To do that, we're going to create a class that wraps Vulkan objects and automatically cleans them up when it goes out of scope, for example because the application was closed.

First consider the interface we want from this `VDeleter` wrapper class. Let's say we want to store a `VkInstance` object that should be destroyed with `vkDestroyInstance` at some point. Then we would add the following class member:

```
VDeleter<VkInstance> instance{vkDestroyInstance};
```

The template argument specifies the type of Vulkan object we want to wrap and the constructor argument specifies the function to use to clean up the object when it goes out of scope.

To assign an object to the wrapper, we would simply want to pass its pointer to the creation function as if it was a normal `VkInstance` variable:

```
vkCreateInstance(&instanceCreateInfo, nullptr, &instance);
```

Unfortunately, taking the address of the handle in the wrapper doesn't necessarily mean that we want to overwrite its existing value. A common pattern is to simply use `&instance` as short-hand for an array of instances with 1 item. If we intend to write a new handle, then the wrapper should clean up any previous object to not leak memory. Therefore it would be better to have the `&` operator return a constant pointer and have an explicit function to state that we wish to replace the handle. The `replace` function calls clean up for any existing handle and then gives you a non-const pointer to overwrite the handle:

```
vkCreateInstance(&instanceCreateInfo, nullptr, instance.replace());
```

Just like that we can now use the `instance` variable wherever a `VkInstance` would normally be accepted. We no longer have to worry about cleaning up anymore, because that will automatically happen once the `instance` variable becomes unreachable! That's pretty easy, right?

The implementation of such a wrapper class is fairly straightforward. It just requires a bit of lambda magic to shorten the syntax for specifying the cleanup functions.

```
template <typename T>
class VDeleter {
public:
    VDeleter() : VDeleter([](T, VkAllocationCallbacks*) {}) {}

    VDeleter(std::function<void(T, VkAllocationCallbacks*)> deletef) {
        this->deleter = [=](T obj) { deletef(obj, nullptr); };
    }

    VDeleter(const VDeleter<VkInstance>& instance, std::function<void(VkInstance, T, VkAlloc
        this->deleter = [&instance, deletef](T obj) { deletef(instance, obj, nullptr); };
    }

    VDeleter(const VDeleter<VkDevice>& device, std::function<void(VkDevice, T, VkAllocation(
        this->deleter = [&device, deletef](T obj) { deletef(device, obj, nullptr); };
    }

    ~VDeleter() {
        cleanup();
    }

    const T* operator &() const {
        return &object;
    }

    T* replace() {
        cleanup();
        return &object;
    }

    operator T() const {
```

```
        return object;
    }

    void operator=(T rhs) {
        if (rhs != object) {
            cleanup();
            object = rhs;
        }
    }

    template<typename V>
    bool operator==(V rhs) {
        return object == T(rhs);
    }

private:
    T object{VK_NULL_HANDLE};
    std::function<void(T)> deleter;

    void cleanup() {
        if (object != VK_NULL_HANDLE) {
            deleter(object);
        }
        object = VK_NULL_HANDLE;
    }
};
```

The three non-default constructors allow you to specify all three types of deletion functions used in Vulkan:

- `vkDestroyXXX(object, callbacks)`: Only the object itself needs to be passed to the cleanup function, so we can simply construct a `VDeleter` with just the function as argument.
- `vkDestroyXXX(instance, object, callbacks)`: A `VkInstance` also needs to be passed to the cleanup function, so we use the `VDeleter` constructor that takes the `VkInstance` reference and cleanup function as parameters.
- `vkDestroyXXX(device, object, callbacks)`: Similar to the previous case, but a `VkDevice` must be passed instead of a `VkInstance`.

The `callbacks` parameter is optional and we always pass `nullptr` to it, as you can see in the `VDeleter` definition.

All of the constructors initialize the object handle with the equivalent of `nullptr` in Vulkan: `VK_NULL_HANDLE`. Any extra arguments that are needed for the deleter functions must also be passed, usually the parent object. It overloads the address-of, assignment, comparison and casting operators to make the wrapper as transparent as possible. When the wrapped object goes out of scope, the destructor is invoked, which in turn calls the cleanup function we specified.

The address-of operator returns a constant pointer to make sure that the object within the wrapper is not unexpectedly changed. If you want to replace the handle within the wrapper through a pointer, then you should use the `replace()` function instead. It will invoke the cleanup function for the existing handle so that you can safely overwrite it afterwards.

There is also a default constructor with a dummy deleter function that can be used to initialize it later, which will be useful for lists of deleters.

I've added the class code between the headers and the `HelloTriangleApplication` class definition. You can also choose to put it in a separate header file. We'll use it for the first time in the next chapter where we'll create the very first Vulkan object!

Integrating GLFW

Vulkan works perfectly fine without a creating a window if you want to use it off-screen rendering, but it's a lot more exciting to actually show something! First replace the `#include <vulkan/vulkan.h>` line with

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

That way GLFW will include its own definitions and automatically load the Vulkan header with it. Add a `initWindow` function and add a call to it from the `run` function before the other calls. We'll use that function to initialize GLFW and create a window.

```
void run() {
    initWindow();
    initVulkan();
    mainLoop();
}

private:
    void initWindow() {

    }
```

The very first call in `initWindow` should be `glfwInit()`, which initializes the GLFW library. Because GLFW was originally designed to create an OpenGL context, we need to tell it to not create an OpenGL context with a subsequent call:

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

Because handling resized windows takes special care that we'll look into later, disable it for now with another window hint call:

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

All that's left now is creating the actual window. Add a `GLFWwindow* window;` private class member to store a reference to it and initialize the window with:

```
window = glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

The first three parameters specify the width, height and title of the window. The fourth parameter allows you to optionally specify a monitor to open the window on and the last parameter is only relevant to OpenGL.

It's a good idea to use constants instead of hardcoded width and height numbers because we'll be referring to these values a couple of times in the future. I've added the following lines above the `HelloTriangleApplication` class definition:

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

and replaced the window creation call with

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
```

You should now have a `initWindow` function that looks like this:

```
void initWindow() {
    glfwInit();

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

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
}
```

To keep the application running until either an error occurs or the window is closed, we need to add an event loop to the `mainLoop` function as follows:

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

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

This code should be fairly self-explanatory. It loops and checks for events like pressing the X button until the window has been closed by the user. This is also the loop where we'll later call a function to render a single frame. Once the window is closed, we need to clean up resources by destroying it and GLFW] itself.

When you run the program now you should see a window titled `Vulkan` show up until the application is terminated by closing the window. Now that we have the skeleton for the Vulkan application, let's create the first Vulkan object!

C++ code

Instance

Creating an instance

The very first thing you need to do is initialize the Vulkan library by creating an *instance*. The instance is the connection between your application and the Vulkan library and creating it involves specifying some details about your application to the driver.

Start by adding a `createInstance` function and add a call to it in the `initVulkan` function.

```
void initVulkan() {
    createInstance();
}
```

Additionally add a class member to hold the handle to the instance, like we saw in the resource management section of the previous chapter.

```
private:
VDeleter<VkInstance> instance {vkDestroyInstance};
```

The `vkDestroyInstance` function, as you might imagine, will clean up the instance that we'll create in a moment. The second parameter is optional and allows you to specify callbacks for a custom allocator. You'll see that most of the creation and destroy functions have such a callback parameter and we'll always pass a `nullptr` as argument, as seen in the `VDeleter` definition.

Now, to create an instance we'll first have to fill in a struct with some information about our application. This data is technically optional, but it may provide some useful information to the driver to optimize for our specific application, for example because it uses a well-known graphics engine with certain special behavior. This struct is called `VkApplicationInfo`:

```
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "Hello Triangle";
appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.pEngineName = "No Engine";
```

```
appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.apiVersion = VK_API_VERSION_1_0;
```

As mentioned before, many structs in Vulkan require you to explicitly specify the type in the `sType` member. This is also one of the many structs with a `pNext` member that can point to extension information in the future. We're using default initialization here to leave it as `nullptr`.

A lot of information in Vulkan is passed through structs instead of function parameters and we'll have to fill in one more struct to provide sufficient information for creating an instance. This next struct is not optional and tells the Vulkan driver which global extensions and validation layers we want to use. Global here means that they apply to the entire program and not a specific device, which will become clear in the next few chapters.

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
```

The first two parameters are straightforward. The next two layers specify the desired global extensions. As mentioned in the overview chapter, Vulkan is a platform agnostic API, which means that you need an extension to interface with the window system. GLFW has a handy built-in function that returns the extension(s) it needs to do that which we can pass to the struct:

```
unsigned int glfwExtensionCount = 0;
const char** glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

The last two members of the struct determine the global validation layers to enable. We'll talk about these more in-depth in the next chapter, so just leave these empty for now.

```
createInfo.enabledLayerCount = 0;
```

We've now specified everything Vulkan needs to create an instance and we can finally issue the `vkCreateInstance` call:

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

As you'll see, the general pattern that object creation function parameters in Vulkan follow is:

- Pointer to struct with creation info
- Pointer to custom allocator callbacks, always `nullptr` in this tutorial
- Pointer to the variable that stores the handle to the new object

If everything went well then the handle to the instance was stored in the wrapped `VkInstance` class member. Nearly all Vulkan functions return a value of type `VkResult` that is either `VK_SUCCESS` or an error code. To check if the instance was created successfully, simply add a check for the success value:

```
if (vkCreateInstance(&createInfo, nullptr, instance.replace()) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```

Now run the program to make sure that the instance is created successfully.

Checking for extension support

If you look at the `vkCreateInstance` documentation then you'll see that one of the possible error codes is `VK_ERROR_EXTENSION_NOT_PRESENT`. We could simply specify the extensions we require and terminate if that error code comes back. That makes sense for essential extensions like the window system interface, but what if we want to check for optional functionality?

To retrieve a list of supported extensions before creating an instance, there's the `vkEnumerateInstanceExtensionProperties` function. It takes a pointer to a variable that stores the number of extensions and an array of `VkExtensionProperties` to store details of the extensions. It also takes an optional first parameter that allows us to filter extensions by a specific validation layer, which we'll ignore for now.

To allocate an array to hold the extension details we first need to know how many there are. You can request just the number of extensions by leaving the latter parameter empty:

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

Now allocate an array to hold the extension details (`include <vector>`):

```
std::vector<VkExtensionProperties> extensions(extensionCount);
```

Finally we can query the extension details:

```
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());
```

Each `VkExtensionProperties` struct contains the name and version of an extension. We can list them with a simple for loop (`\t` is a tab for indentation):

```
std::cout << "available extensions:" << std::endl;

for (const auto& extension : extensions) {
    std::cout << "\t" << extension.extensionName << std::endl;
}
```

You can add this code to the `createInstance` function if you'd like to provide some details about the Vulkan support. As a challenge, try to create a function that checks if all of the extensions returned by `glfwGetRequiredInstanceExtensions` are included in the supported extensions list.

Before continuing with the more complex steps after instance creation, it's time to evaluate our debugging options by checking out validation layers.

C++ code

Validation layers

What are validation layers?

The Vulkan API is designed around the idea of minimal driver overhead and one of the manifestations of that goal is that there is very limited error checking in the API by default. Even mistakes as simple as setting enumerations to incorrect values or passing null pointers to required parameters are generally not explicitly handled and will simply result in crashes or undefined behavior. Because Vulkan requires you to be very explicit about everything you're doing, it's easy to make many small mistakes like using a new GPU feature and forgetting to request it at logical device creation time.

However, that doesn't mean that these checks can't be added to the API. Vulkan introduces an elegant system for this known as *validation layers*. Validation layers are optional components that hook into Vulkan function calls to apply additional operations. Common operations in validation layers 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's an example of what the implementation of a function in a diagnostics validation layer could look like:

```
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkInstance* instance) {

    if (pCreateInfo == nullptr || instance == nullptr) {
        log("Null pointer passed to required parameter!");
        return VK_ERROR_INITIALIZATION_FAILED;
    }
```

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

These validation layers can be freely stacked to include all the debugging functionality that you're interested in. You can simply enable validation layers for debug builds and completely disable them for release builds, which gives you the best of both worlds!

Vulkan does not come with any validation layers built-in, but the LunarG Vulkan SDK provides a nice set of layers that check for common errors. They're also completely open source, so you can check which kind of mistakes they check for and contribute. Using the validation layers is the best way to avoid your application breaking on different drivers by accidentally relying on undefined behavior.

Validation layers can only be used if they have been installed onto the system. For example, the LunarG validation layers are only available on PCs with the Vulkan SDK installed.

There were formerly two different types of validation layers in Vulkan. Instance and device specific layers. The idea was that instance layers would only check calls related to global Vulkan objects like instances and device specific layers only calls related to a specific GPU. Device specific layers have now been deprecated, which means that instance validation layers apply to all Vulkan calls. The specification document still recommends that you enable validation layers at device level as well for compatibility, which is required by some implementations. We'll simply specify the same layers as the instance at logical device level, which we'll see later on.

Using validation layers

In this section we'll see how to enable the standard diagnostics layers provided by the Vulkan SDK. Just like extensions, validation layers need to be enabled by specifying their name. Instead of having to explicitly specify all of the useful layers, the SDK allows you to request the `VK_LAYER_LUNARG_standard_validation` layer that implicitly enables a whole range of useful diagnostics layers.

Let's first add two configuration variables to the program to specify the layers to enable and whether to enable them or not. I've chosen to base that value on whether the program is being compiled in debug mode or not. The `NDEBUG` macro is part of the C++ standard and means "not debug".

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

const std::vector<const char*> validationLayers = {
    "VK_LAYER_LUNARG_standard_validation"
```

```
};

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

We'll add a new function `checkValidationLayerSupport` that checks if all of the requested layers are available. First list all of the available extensions using the `vkEnumerateInstanceLayerProperties` function. Its usage is identical to that of `vkEnumerateInstanceExtensionProperties` which was discussed in the instance creation chapter.

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

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

    return false;
}
```

Next, check if all of the layers in `validationLayers` exist in the `availableLayers` list. You may need to include `<cstring>` for `strcmp`.

```
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 now use this function in `createInstance`:

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

```
    }

    ...
}
```

Now run the program in debug mode and ensure that the error does not occur. If it does, then make sure you have properly installed the Vulkan SDK. If none or very few layers are being reported, then you may be dealing with this issue (requires a LunarG account to view). See that page for help with fixing it.

Finally, modify the `VkInstanceCreateInfo` struct instantiation to include the validation layer names if they are enabled:

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

If the check was successful then `vkCreateInstance` should not ever return a `VK_ERROR_LAYER_NOT_PRESENT` error, but you should run the program to make sure.

Message callback

Unfortunately just enabling the layers doesn't help much, because they currently have no way to relay the debug messages back to our program. To receive those messages we have to set up a callback, which requires the `VK_EXT_debug_report` extension.

We'll first create a `getRequiredExtensions` function that will return the required list of extensions based on whether validation layers are enabled or not:

```
std::vector<const char*> getRequiredExtensions() {
    std::vector<const char*> extensions;

    unsigned int glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    for (unsigned int i = 0; i < glfwExtensionCount; i++) {
        extensions.push_back(glfwExtensions[i]);
    }

    if (enableValidationLayers) {
        extensions.push_back(VK_EXT_DEBUG_REPORT_EXTENSION_NAME);
```

```
    }

    return extensions;
}
```

The extensions specified by GLFW are always required, but the debug report extension is conditionally added. Note that I've used the `VK_EXT_DEBUG_REPORT_EXTENSION_NAME` macro here which is equal to the literal string "VK_EXT_debug_report". Using this macro lets you avoid typos.

We can now use this function in `createInstance`:

```
auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = extensions.size();
createInfo.ppEnabledExtensionNames = extensions.data();
```

Run the program to make sure you don't receive a `VK_ERROR_EXTENSION_NOT_PRESENT` error. We don't really need to check for the existence of this extension, because it should be implied by the availability of the validation layers.

Now let's see what a callback function looks like. Add a new static member function called `debugCallback` with the `PFN_vkDebugReportCallbackEXT` prototype. The `VKAPI_ATTR` and `VKAPI_CALL` ensure that the function has the right signature for Vulkan to call it.

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugReportFlagsEXT flags,
    VkDebugReportObjectTypeEXT objType,
    uint64_t obj,
    size_t location,
    int32_t code,
    const char* layerPrefix,
    const char* msg,
    void* userData) {

    std::cerr << "validation layer: " << msg << std::endl;

    return VK_FALSE;
}
```

The first parameter specifies the type of message, which can be a combination of any of the following bit flags:

- `VK_DEBUG_REPORT_INFORMATION_BIT_EXT`
- `VK_DEBUG_REPORT_WARNING_BIT_EXT`
- `VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT`
- `VK_DEBUG_REPORT_ERROR_BIT_EXT`
- `VK_DEBUG_REPORT_DEBUG_BIT_EXT`

The `objType` parameter specifies the type of object that is the subject of the message. For example if `obj` is a `VkPhysicalDevice` then `objType` would be `VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT`. This works because internally all Vulkan handles are typedef'd as `uint64_t`.

The `msg` parameter contains the pointer to the message itself. Finally, there's a `userData` parameter to pass your own data to the callback.

All that remains now is telling Vulkan about the callback function. Perhaps somewhat surprisingly, even the debug callback in Vulkan is managed with a handle that needs to be explicitly created and destroyed. Add a class member for this handle right under `instance`:

```
VkDebugReportCallbackEXT callback;
```

Now add a function `setupDebugCallback` to be called from `initVulkan` right after `createInstance`:

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
}

void setupDebugCallback() {
    if (!enableValidationLayers) return;

}
```

We'll need to fill in a structure with details about the callback:

```
VkDebugReportCallbackCreateInfoEXT createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT;
createInfo.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT;
createInfo.pfnCallback = debugCallback;
```

The `flags` field allows you to filter which types of messages you would like to receive. The `pfnCallback` field specifies the pointer to the callback function. You can optionally pass a pointer to the `pUserData` field which will be passed along to the callback function via the `userData` parameter. You could use this to pass a pointer to the `HelloTriangleApplication` class, for example.

This struct should be passed to the `vkCreateDebugReportCallbackEXT` function to create the `VkDebugReportCallbackEXT` object. Unfortunately, because this function is an extension function, it is not automatically loaded. We have to look up its address ourselves using `vkGetInstanceProcAddr`. We're going to create our own proxy function that handles this in the background. I've added it right above the `VDeleter` definition.

```
VkResult CreateDebugReportCallbackEXT(VkInstance instance, const VkDebugReportCallbackCreate
    auto func = (PFN_vkCreateDebugReportCallbackEXT) vkGetInstanceProcAddr(instance, "vkCrea
    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pCallback);
```

```
    } else {
        return VK_ERROR_EXTENSION_NOT_PRESENT;
    }
}
```

The `vkGetInstanceProcAddr` function will return `nullptr` if the function couldn't be loaded. We can now call this function to create the extension object if it's available:

```
if (CreateDebugReportCallbackEXT(instance, &createInfo, nullptr, &callback) != VK_SUCCESS) {
    throw std::runtime_error("failed to set up debug callback!");
}
```

Let's see if it works... Run the program and close the window once you're fed up with staring at the blank window. You'll see that the following message is printed to the command prompt:

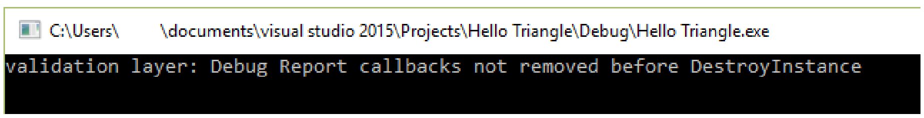

Figure 22:

Oops, it has already spotted a bug in our program! The `VkDebugReportCallbackEXT` object needs to be cleaned up with a call to `vkDestroyDebugReportCallbackEXT`. Change the `callback` variable to use our deleter wrapper. Similarly to `vkCreateDebugReportCallbackEXT` the function needs to be explicitly loaded. Create another proxy function right below `CreateDebugReportCallbackEXT`:

```
void DestroyDebugReportCallbackEXT(VkInstance instance, VkDebugReportCallbackEXT callback, c
    auto func = (PFN_vkDestroyDebugReportCallbackEXT) vkGetInstanceProcAddr(instance, "vkDes
    if (func != nullptr) {
        func(instance, callback, pAllocator);
    }
}
```

Make sure that this function is either a static class function or a function outside the class. We can then specify it as cleanup function:

```
VDeleter<VkDebugReportCallbackEXT> callback{instance, DestroyDebugReportCallbackEXT};
```

Make sure to change the line that creates the debug report callback to use the `replace()` method of the wrapper:

```
if (CreateDebugReportCallbackEXT(instance, &createInfo, nullptr, callback.replace()) != VK_S
```

When you run the program again you'll see that the error message has disappeared. If you want to see which call triggered a message, you can add a breakpoint to the message callback and look at the stack trace.

Configuration

There are a lot more settings for the behavior of validation layers than just the flags specified in the `VkDebugReportCallbackCreateInfoEXT` struct. Browse to the Vulkan SDK and go to the `Config` directory. There you will find a `vk_layer_settings.txt` file that explains how to configure the layers.

To configure the layer settings for your own application, copy the file to the `Debug` and `Release` directories of your project and follow the instructions to set the desired behavior. However, for the remainder of this tutorial I'll assume that you're using the default settings.

Throughout this tutorial I'll be making a couple of intentional mistakes to show you how helpful the validation layers are with catching them and to teach you how important it is to know exactly what you're doing with Vulkan. Now it's time to look at Vulkan devices in the system.

C++ code

Physical devices and queue families

Selecting a physical device

After initializing the Vulkan library through a VkInstance we need to look for and select a graphics card in the system that supports the features we need. In fact we can select any number of graphics cards and use them simultaneously, but in this tutorial we'll stick to the first graphics card that suits our needs.

We'll add a function `pickPhysicalDevice` and add a call to it in the `initVulkan` function.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    pickPhysicalDevice();
}

void pickPhysicalDevice() {

}
```

The graphics card that we'll end up selecting will be stored in a VkPhysicalDevice handle that is added as a new class member. This object will be implicitly destroyed when the VkInstance is destroyed, so we don't need to add a delete wrapper.

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

Listing the graphics cards is very similar to listing extensions and starts with querying just the number.

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

If there are 0 devices with Vulkan support then there is no point going further.

```
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
```

Otherwise we can now allocate an array to hold all of the VkPhysicalDevice handles.

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

Now we need to evaluate each of them and check if they are suitable for the operations we want to perform, because not all graphics cards are created equal. For that we'll introduce a new function:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

And we'll check if any of the physical devices meet the requirements that we'll add to that function.

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

if (physicalDevice == VK_NULL_HANDLE) {
    throw std::runtime_error("failed to find a suitable GPU!");
}
```

The next section will introduce the first requirements that we'll check for in the `isDeviceSuitable` function. As we'll start using more Vulkan features in the later chapters we will also extend this function to include more checks.

Base device suitability checks

To evaluate the suitability of a device we can start by querying for some details. Basic device properties like the name, type and supported Vulkan version can be queried using vkGetPhysicalDeviceProperties.

```
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(device, &deviceProperties);
```

The support for optional features like texture compression, 64 bit floats and multi viewport rendering (useful for VR) can be queried using vkGetPhysicalDeviceFeatures:

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
```

There are more details that can be queried from devices that we'll discuss later concerning device memory and queue families (see the next section).

As an example, let's say we consider our application only usable for dedicated graphics cards that support geometry shaders. Then the `isDeviceSuitable` function would look like this:

```
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;
}
```

Instead of just checking if a device is suitable or not and going with the first one, you could also give each device a score and pick the highest one. That way you could favor a dedicated graphics card by giving it a higher score, but fall back to an integrated GPU if that's the only available one. You could implement something like that as follows:

```
#include <map>

...

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;
}
```

You don't need to implement all that for this tutorial, but it's to give you an idea of how you could design your device selection process. Of course you can also just display the names of the choices and allow the user to select.

Because we're just starting out, Vulkan support is the only thing we need and therefore we'll settle for just any GPU:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

In the next section we'll discuss the first real required feature to check for.

Queue families

It has been briefly touched upon before that almost every operation in Vulkan, anything from drawing to uploading textures, requires commands to be submitted to a queue. There are different types of queues that originate from different *queue families* and each family of queues allows only a subset of commands. For example, there could be a queue family that only allows processing of compute commands or one that only allows memory transfer related commands.

We need to check which queue families are supported by the device and which one of these supports the commands that we want to use. For that purpose we'll add a new function `findQueueFamilies` that looks for all the queue families we need. Right now we'll only look for a queue that supports graphics commands, but we may extend this function to look for more at a later point in time.

This function will return the indices of the queue families that satisfy certain desired properties. The best way to do that is using a structure, where an index of `-1` will denote "not found":

```
struct QueueFamilyIndices {
    int graphicsFamily = -1;

    bool isComplete() {
        return graphicsFamily >= 0;
    }
};
```

We can now begin implementing `findQueueFamilies`:

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    ...

    return indices;
}
```

The process of retrieving the list of queue families is exactly what you expect and uses `vkGetPhysicalDeviceQueueFamilyProperties`:

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

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());
```

The VkQueueFamilyProperties struct contains some details about the queue family, including the type of operations that are supported and the number of queues that can be created based on that family. We need to find at least one queue family that supports `VK_QUEUE_GRAPHICS_BIT`.

```
int i = 0;
for (const auto& queueFamily : queueFamilies) {
    if (queueFamily.queueCount > 0 && queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        indices.graphicsFamily = i;
    }
```

```
    if (indices.isComplete()) {
        break;
    }

    i++;
}
```

Now that we have this fancy queue family lookup function, we can use it as a check in the `isDeviceSuitable` function to ensure that the device can process the commands we want to use:

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

    return indices.isComplete();
}
```

Great, that's all we need for now to find the right physical device! The next step is to create a logical device to interface with it.

C++ code

Logical device and queues

Introduction

After selecting a physical device to use we need to set up a *logical device* to interface with it. The logical device creation process is similar to the instance creation process and describes the features we want to use. We also need to specify which queues to create now that we've queried which queue families are available. You can even create multiple logical devices from the same physical device if you have varying requirements.

Start by adding a new class member to store the logical device handle in. Make sure to place the declaration below the `VkInstance` member, because it needs to be cleaned up before the instance is cleaned up. See C++ destruction order. Logical devices are cleaned up with the `vkDestroyDevice` function.

```
VDeleter<VkDevice> device{vkDestroyDevice};
```

Next, add a `createLogicalDevice` function that is called from `initVulkan`.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    pickPhysicalDevice();
    createLogicalDevice();
```

```
}

void createLogicalDevice() {

}
```

Specifying the queues to be created

The creation of a logical device involves specifying a bunch of details in structs again, of which the first one will be `VkDeviceQueueCreateInfo`. This structure describes the number of queues we want for a single queue family. Right now we're only interested in a queue with graphics capabilities.

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily;
queueCreateInfo.queueCount = 1;
```

The currently available drivers will only allow you to create a low number of queues for each family queue and you don't really need more than one. That's because you can create all of the command buffers on multiple threads and then submit them all at once on the main thread with a single low-overhead call.

Vulkan lets you assign priorities to queues to influence the scheduling of command buffer execution using floating point numbers between `0.0` and `1.0`. This is required even if there is only a single queue:

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

Specifying used device features

The next information to specify is the set of device features that we'll be using. These are the features that we queried support for with `vkGetPhysicalDeviceFeatures` in the previous chapter, like geometry shaders. Right now we don't need anything special, so we can simply define it and leave everything to `VK_FALSE`. We'll come back to this structure once we're about to start doing more interesting things with Vulkan.

```
VkPhysicalDeviceFeatures deviceFeatures = {};
```

Creating the logical device

With the previous two structures in place, we can start filling in the main VkDeviceCreateInfo structure.

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

First add pointers to the queue creation info and device features structs:

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

createInfo.pEnabledFeatures = &deviceFeatures;
```

The remainder of the information bears a resemblance to the VkInstanceCreateInfo struct and requires you to specify extensions and validation layers. The difference is that these are device specific this time.

An example of a device specific extension is VK_KHR_swapchain, which allows you to present rendered images from that device to windows. It is possible that there are Vulkan devices in the system that lack this ability, for example because they only support compute operations. We will come back to this extension in the swap chain chapter.

As mentioned in the validation layers chapter, we will enable the same validation layers for devices as we did for the instance. We won't need any device specific extensions for now.

```
createInfo.enabledExtensionCount = 0;

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

That's it, we're now ready to instantiate the logical device with a call to the appropriately named vkCreateDevice function.

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

The parameters are the physical device to interface with, 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. Similarly to the instance creation function, this call can return errors based on enabling non-existent extensions or specifying the desired usage of unsupported features.

Retrieving queue handles

The queues are automatically created along with the logical device, but we don't have a handle to interface with them yet. First add a class member to store a handle to the graphics queue:

```
VkQueue graphicsQueue;
```

Device queues are implicitly cleaned up when the device is destroyed, so we don't need to wrap it in a deleter object.

We can use the `vkGetDeviceQueue` function to retrieve queue handles for each queue family. The parameters are the logical device, queue family, queue index and a pointer to the variable to store the queue handle in. Because we're only creating a single queue from this family, we'll simply use index `0`.

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

With the logical device and queue handles we can now actually start using the graphics card to do things! In the next few chapters we'll set up the resources to present results to the window system.

C++ code

Window surface

Since Vulkan is a platform agnostic API, it can not interface directly with the window system on its own. To establish the connection between Vulkan and the window system to present results to the screen, we need to use the WSI (Window System Integration) extensions. In this chapter we'll discuss the first one, which is `VK_KHR_surface`. It exposes a `VkSurfaceKHR` object that represents an abstract type of surface to present rendered images to. The surface in our program will be backed by the window that we've already opened with GLFW.

The `VK_KHR_surface` extension is an instance level extension and we've actually already enabled it, because it's included in the list returned by `glfwGetRequiredInstanceExtensions`. The list also includes some other WSI extensions that we'll use in the next couple of chapters.

The window surface needs to be created right after the instance creation, because it can actually influence the physical device selection. The reason we postponed this is because window surfaces are part of the larger topic of render targets and presentation for which the explanation would have cluttered the basic setup. It should also be noted that window surfaces are an entirely optional component in Vulkan, if you just need off-screen rendering. Vulkan allows you to do that without hacks like creating an invisible window (necessary for OpenGL).

Window surface creation

Start by adding a `surface` class member right below the debug callback. Surfaces are destroyed using the `vkDestroySurfaceKHR` call.

```
VDeleter<VkSurfaceKHR> surface{instance, vkDestroySurfaceKHR};
```

Although the `VkSurfaceKHR` object and its usage is platform agnostic, its creation isn't because it depends on window system details. For example, it needs the `HWND` and `HMODULE` handles on Windows. Therefore there is a platform-specific addition to the extension, which on Windows is called `VK_KHR_win32_surface` and is also automatically included in the list from `glfwGetRequiredInstanceExtensions`.

I will demonstrate how this platform specific extension can be used to create a surface on Windows, but we won't actually use it in this tutorial. It doesn't make any sense to use a library like GLFW and then proceed to use platform-specific code anyway. GLFW actually has `glfwCreateWindowSurface` that handles the platform differences for us. Still, it's good to see what it does behind the scenes before we start relying on it.

Because a window surface is a Vulkan object, it comes with a `VkWin32SurfaceCreateInfoKHR` struct that needs to be filled in. It has two important parameters: `hwnd` and `hinstance`. These are the handles to the window and the process.

```
VkWin32SurfaceCreateInfoKHR createInfo;
createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
createInfo.hwnd = glfwGetWin32Window(window);
createInfo.hinstance = GetModuleHandle(nullptr);
```

The `glfwGetWin32Window` function is used to get the raw `HWND` from the GLFW window object. The `GetModuleHandle` call returns the `HINSTANCE` handle of the current process.

After that the surface can be created with `vkCreateWin32SurfaceKHR`, which needs to be explicitly loaded again. Other than that the call is trivial and includes a parameter for the instance, surface creation details, custom allocators and the variable for the surface handle to be stored in.

```
auto CreateWin32SurfaceKHR = (PFN_vkCreateWin32SurfaceKHR) vkGetInstanceProcAddr(instance, '

if (!CreateWin32SurfaceKHR || CreateWin32SurfaceKHR(instance, &createInfo,
   nullptr, surface.replace()) != VK_SUCCESS) {
    throw std::runtime_error("failed to create window surface!");
}
```

The process is similar for other platforms like Linux, where `vkCreateXcbSurfaceKHR` takes an XCB connection and window as creation details with X11.

The `glfwCreateWindowSurface` function performs exactly this operation with a different implementation for each platform. We'll now integrate it into our program. Add a function `createSurface` to be called from `initVulkan` right after instance creation and `setupDebugCallback`.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createSurface() {

}
```

The GLFW call takes simple parameters instead of a struct which makes the implementation of the function very straightforward:

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, surface.replace()) != VK_SUCCESS)
        throw std::runtime_error("failed to create window surface!");
    }
}
```

The parameters are the `VkInstance`, GLFW window pointer, custom allocators and pointer to `VkSurfaceKHR` variable. It simply passes through the `VkResult` from the relevant platform call.

Querying for presentation support

Although the Vulkan implementation may support window system integration, that does not mean that every device in the system supports it. Therefore we need to extend `isDeviceSuitable` to ensure that a device can present images to the surface we created. Since the presentation is a queue-specific feature, the problem is actually about finding a queue family that supports presenting to the surface we created.

It's actually possible that the queue families supporting drawing commands and the ones supporting presentation do not overlap. Therefore we have to take into account that there could be a distinct presentation queue by modifying the `QueueFamilyIndices` structure:

```
struct QueueFamilyIndices {
    int graphicsFamily = -1;
    int presentFamily = -1;

    bool isComplete() {
        return graphicsFamily >= 0 && presentFamily >= 0;
```

```
    }
};
```

Next, we'll modify the `findQueueFamilies` function to look for a queue family that has the capability of presenting to our window surface. The function to check for that is `vkGetPhysicalDeviceSurfaceSupportKHR`, which takes the physical device, queue family index and surface as parameters. Add a call to it in the same loop as the `VK_QUEUE_GRAPHICS_BIT`:

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

Then simply check the value of the boolean and store the presentation family queue index:

```
if (queueFamily.queueCount > 0 && presentSupport) {
    indices.presentFamily = i;
}
```

Note that it's very likely that these end up being the same queue family after all, but throughout the program we will treat them as if they were separate queues for a uniform approach. Nevertheless, you could add logic to explicitly prefer a physical device that supports drawing and presentation in the same queue for improved performance.

Creating the presentation queue

The one thing that remains is modifying the logical device creation procedure to create the presentation queue and retrieve the `VkQueue` handle. Add a member variable for the handle:

```
VkQueue presentQueue;
```

Next, we need to have multiple `VkDeviceQueueCreateInfo` structs to create a queue from both families. An elegant way to do that is to create a set of all unique queue families that are necessary for the required queues:

```
#include <set>

...

QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<int> uniqueQueueFamilies = {indices.graphicsFamily, indices.presentFamily};

float queuePriority = 1.0f;
for (int 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);
}
```

And modify `VkDeviceCreateInfo` to point to the vector:

```
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.queueCreateInfoCount = (uint32_t) queueCreateInfos.size();
```

If the queue families are the same, then we only need to pass its index once. Finally, add a call to retrieve the queue handle:

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

In case the queue families are the same, the two handles will most likely have the same value now. In the next chapter we're going to look at swap chains and how they give us the ability to present images to the surface.

C++ code

Swap chain

In this chapter we will look at the infrastructure that gives you images to render to that can be presented to the screen afterwards. This infrastructure is known as the *swap chain* and must be created explicitly in Vulkan. The swap chain is essentially a queue of images that are waiting to be presented to the screen. Our application will acquire such an image to draw to it, and then return it to the queue. How exactly the queue works and the conditions for presenting an image from the queue depend on how the swap chain is set up, but the general purpose of the swap chain is to synchronize the presentation of images with the refresh rate of the screen.

Checking for swap chain support

Not all graphics cards are capable of presenting images directly to a screen for various reasons, for example because they are designed for servers and don't have any display outputs. Secondly, since image presentation is heavily tied into the window system and the surfaces associated with windows, it is not actually part of the Vulkan core. You have to enable the `VK_KHR_swapchain` device extension after querying for its support.

For that purpose we'll first extend the `isDeviceSuitable` function to check if this extension is supported. We've previously seen how to list the extensions that are supported by a `VkPhysicalDevice`, so doing that should be fairly straightforward. Note that the Vulkan header file provides a nice macro `VK_KHR_SWAPCHAIN_EXTENSION_NAME` that is defined as `VK_KHR_swapchain`. The advantage of using this macro is that the compiler will catch misspellings.

First declare a list of required device extensions, similar to the list of validation layers to enable.

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

Next, create a new function `checkDeviceExtensionSupport` that is 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;
}
```

Modify the body of the function to enumerate the extensions and check if all of the required extensions are amongst them.

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

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

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

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

    return requiredExtensions.empty();
}
```

I've chosen to use a set of strings here to represent the unconfirmed required extensions. That way we can easily tick them off while enumerating the sequence of available extensions. Of course you can also use a nested loop like in `checkValidationLayerSupport`. The performance difference is irrelevant. Now run the code and verify that your graphics card is indeed capable of creating a swap chain. It should be noted that the availability of a presentation queue, as we checked in the previous chapter, implies that the swap chain extension must be supported. However, it's still good to be explicit about things, and the extension does have to be explicitly enabled.

Enabling the extension just requires a small change to the logical device creation structure:

```
createInfo.enabledExtensionCount = deviceExtensions.size();
createInfo.ppEnabledExtensionNames = deviceExtensions.data();
```

Querying details of swap chain support

Just checking if a swap chain is available is not sufficient, because it may not actually be compatible with our window surface. Creating a swap chain also involves a lot more settings than instance and device creation, so we need to query for some more details before we're able to proceed.

There are basically three kinds of properties we need to check:

- Basic surface capabilities (min/max number of images in swap chain, min/max width and height of images)
- Surface formats (pixel format, color space)
- Available presentation modes

Similar to `findQueueFamilies`, we'll use a struct to pass these details around once they've been queried. The three aforementioned types of properties come in the form of the following structs and lists of structs:

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

We'll now create a new function `querySwapChainSupport` that will populate this struct.

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

    return details;
}
```

This section covers how to query the structs that include this information. The meaning of these structs and exactly which data they contain is discussed in the next section.

Let's start with the basic surface capabilities. These properties 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 the supported capabilities. All of the support querying functions have these two as first parameters because they are the core components of the swap chain.

The next step is about 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 that the vector is resized to hold all the available formats. And 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.pr
}
```

All of the details are in the struct now, so let's extend `isDeviceSuitable` once more to utilize this function to verify that swap chain support is adequate. Swap chain support is sufficient for this tutorial if there is at least one supported image format and one supported presentation mode given the window surface we have.

```
bool swapChainAdequate = false;
if (extensionsSupported) {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(device);
    swapChainAdequate = !swapChainSupport.formats.empty() && !swapChainSupport.presentModes.
}
```

It is important that we only try to query for swap chain support after verifying that the extension is available. The last line of the function changes to:

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

Choosing the right settings for the swap chain

If the swapChainAdequate conditions were met then the support is definitely sufficient, but there may still be many different modes of varying optimality. We'll now write a couple of functions to find the right settings for the best possible swap chain. There are three types of settings to determine:

- Surface format (color depth)
- Presentation mode (conditions for "swapping" images to the screen)
- Swap extent (resolution of images in swap chain)

For each of these settings we'll have an ideal value in mind that we'll go with if it's available and otherwise we'll create some logic to find the next best thing.

Surface format

The function for this setting starts out like this. We'll later pass the formats member of the SwapChainSupportDetails struct as argument.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>& availableF

}
```

Each VkSurfaceFormatKHR entry contains format and colorSpace member. The format member specifies the color channels and types. For example, VK_FORMAT_B8G8R8A8_UNORM means that we store the B, G, R and alpha channels in that order with an 8 bit unsigned integer for a total of 32 bits per pixel. The colorSpace member indicates if the SRGB color space is supported or not using the VK_COLOR_SPACE_SRGB_NONLINEAR_KHR flag. Note that this flag used to be called VK_COLORSPACE_SRGB_NONLINEAR_KHR in old versions of the specification.

For the color space we'll use SRGB if it is available, because it results in more accurate perceived colors. Working directly with SRGB colors is a little bit challenging, so we'll use standard RGB for the color format, of which one of the most common ones is VK_FORMAT_B8G8R8A8_UNORM.

The best case scenario is that the surface has no preferred format, which Vulkan indicates by only returning one VkSurfaceFormatKHR entry which has its format member set to VK_FORMAT_UNDEFINED.

```
if (availableFormats.size() == 1 && availableFormats[0].format == VK_FORMAT_UNDEFINED) {
    return {VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
}
```

If we're not free to choose any format, then we'll go through the list and see if the preferred combination is available:

```
for (const auto& availableFormat : availableFormats) {
    if (availableFormat.format == VK_FORMAT_B8G8R8A8_UNORM && availableFormat.colorSpace ==
        return availableFormat;
    }
}
```

If that also fails then we could start ranking the available formats based on how "good" they are, but in most cases it's okay to just settle with the first format that is specified.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>& availableF
    if (availableFormats.size() == 1 && availableFormats[0].format == VK_FORMAT_UNDEFINED) {
        return {VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR};
    }

    for (const auto& availableFormat : availableFormats) {
        if (availableFormat.format == VK_FORMAT_B8G8R8A8_UNORM && availableFormat.colorSpace
            return availableFormat;
        }
    }

    return availableFormats[0];
}
```

Presentation mode

The presentation mode is arguably the most important setting for the swap chain, because it represents the actual conditions for showing images to the screen. There are four possible modes available in Vulkan:

- `VK_PRESENT_MODE_IMMEDIATE_KHR`: Images submitted by your application are transferred to the screen right away, which may result in tearing.
- `VK_PRESENT_MODE_FIFO_KHR`: The swap chain is a queue where the display takes an image from the front of the queue on a vertical blank and the program inserts rendered images at the back of the queue. If the queue is full then the program has to wait. This is most similar to vertical sync as found in modern games.
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: This mode only differs from the first one if the application is late and the queue was empty at the last vertical blank. Instead of waiting for the next vertical blank, the image is transferred right away when it finally arrives. This may result in visible tearing.
- `VK_PRESENT_MODE_MAILBOX_KHR`: This is another variation of the first mode. Instead of blocking the application when the queue is full, the images that are already queued are simply replaced with the newer ones. This mode can be used to implement triple buffering, which allows you to

avoid tearing with significantly less latency issues than standard vertical sync that uses double buffering.

Only the VK_PRESENT_MODE_FIFO_KHR mode is guaranteed to be available, so we'll again have to write a function that looks for the best mode that is available:

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR> availablePresentM
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

I personally think that triple buffering is a very nice trade-off. It allows us to avoid tearing while still maintaining a fairly low latency by rendering new images that are as up-to-date as possible right until the vertical blank. So, let's look through the list to see if it's available:

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

    return VK_PRESENT_MODE_FIFO_KHR;
}
```

Unfortunately some drivers currently don't properly support VK_PRESENT_MODE_FIFO_KHR, so we should prefer VK_PRESENT_MODE_IMMEDIATE_KHR if VK_PRESENT_MODE_MAILBOX_KHR is not available:

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR> availablePresentM
    VkPresentModeKHR bestMode = VK_PRESENT_MODE_FIFO_KHR;

    for (const auto& availablePresentMode : availablePresentModes) {
        if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR) {
            return availablePresentMode;
        } else if (availablePresentMode == VK_PRESENT_MODE_IMMEDIATE_KHR) {
            bestMode = availablePresentMode;
        }
    }

    return bestMode;
}
```

Swap extent

That leaves only one major property, for which we'll add one last function:

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

}
```

The swap extent is the resolution of the swap chain images and it's almost always exactly equal to the resolution of the window that we're drawing to. The range of the possible resolutions is defined in the `VkSurfaceCapabilitiesKHR` structure. Vulkan tells us to match the resolution of the window by setting the width and height in the `currentExtent` member. However, some window managers do allow us to differ here and this is indicated by setting the width and height in `currentExtent` to a special value: the maximum value of `uint32_t`. In that case we'll pick the resolution that best matches the window within the `minImageExtent` and `maxImageExtent` bounds.

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    } else {
        VkExtent2D actualExtent = {WIDTH, HEIGHT};

        actualExtent.width = std::max(capabilities.minImageExtent.width, std::min(capabilit
        actualExtent.height = std::max(capabilities.minImageExtent.height, std::min(capabili

        return actualExtent;
    }
}
```

The `max` and `min` functions are used here to clamp the value of `WIDTH` and `HEIGHT` between the allowed minimum and maximum extents that are supported by the implementation. Make sure to include the `<algorithm>` header to use them.

Creating the swap chain

Now that we have all of these helper functions assisting us with the choices we have to make at runtime, we finally have all the information that is needed to create a working swap chain.

Create a `createSwapChain` function that starts out with the results of these calls and make sure to call it from `initVulkan` after logical device creation.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
```

```
}

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

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

There is actually one more small things that need to be decided upon, but it's so simple that it's not really worth creating separate functions for them. The first one is the number of images in the swap chain, essentially the queue length. The implementation specifies the minimum amount of images to function properly and we'll try to have one more than that to properly implement triple buffering.

```
uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount > swapChainSupport.capabil
    imageCount = swapChainSupport.capabilities.maxImageCount;
}
```

A value of `0` for `maxImageCount` means that there is no limit besides memory requirements, which is why we need to check for that.

As is tradition with Vulkan objects, creating the swap chain object requires filling in a large structure. It starts out very familiarly:

```
VkSwapchainCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = surface;
```

After specifying which surface the swap chain should be tied to, the details of the swap chain images are specified:

```
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 is always `1` unless you are developing a stereoscopic 3D application. The `imageUsage` bit field specifies what kind of operations we'll use the images in the swap chain for. In this tutorial we're going to render directly to them, which means that they're used as color attachment. It is also possible that you'll render images to a separate image first to perform operations like post-processing. In that case you may use a value like `VK_IMAGE_USAGE_TRANSFER_DST_BIT` instead and use a memory operation to transfer the rendered image to a swap chain image.

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
uint32_t queueFamilyIndices[] = {(uint32_t) indices.graphicsFamily, (uint32_t) indices.prese

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
}
```

Next, we need to specify how to handle swap chain images that will be used across multiple queue families. That will be the case in our application if the graphics queue family is different from the presentation queue. We'll be drawing on the images in the swap chain from the graphics queue and then submitting them on the presentation queue. There are two ways to handle images that are accessed from multiple queues:

- `VK_SHARING_MODE_EXCLUSIVE`: An image is owned by one queue family at a time and ownership must be explicitly transfered before using it in another queue family. This option offers the best performance.
- `VK_SHARING_MODE_CONCURRENT`: Images can be used across multiple queue families without explicit ownership transfers.

If the queue families differ, then we'll be using the concurrent mode in this tutorial to avoid having to do the ownership chapters, because these involve some concepts that are better explained at a later time. Concurrent mode requires you to specify in advance between which queue families ownership will be shared using the `queueFamilyIndexCount` and `pQueueFamilyIndices` parameters. If the graphics queue family and presentation queue family are the same, which will be the case on most hardware, then we should stick to exclusive mode, because concurrent mode requires you to specify at least two distinct queue families.

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

We can specify that a certain transform should be applied to images in the swap chain if it is supported (`supportedTransforms` in `capabilities`), like a 90 degree clockwise rotation or horizontal flip. To specify that you do not 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. You'll almost always want to simply ignore the alpha channel, hence `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 that means that we don't care about the color of pixels that are obscured, for example because another window is in front of them. Unless you really need to be able to read these pixels back and get predictable results, you'll get the best performance by enabling clipping.

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

That leaves one last field, `oldSwapChain`. With Vulkan it's possible that in your swap chain becomes invalid or unoptimized while your application is running, for example because the window was resized. In that case the swap chain actually needs to be recreated from scratch and a reference to the old one must be specified in this field. This is a complex topic that we'll learn more about in a future chapter. For now we'll assume that we'll only ever create one swap chain.

Now add a class member to store the `VkSwapchainKHR` object with a proper deleter. Make sure to add it after `device` so that it gets cleaned up before the logical device is.

```
VDeleter<VkSwapchainKHR> swapChain{device, vkDestroySwapchainKHR};
```

Now creating the swap chain is as simple as calling `vkCreateSwapchainKHR`:

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

The parameters are the logical device, swap chain creation info, optional custom allocators and a pointer to the variable to store the handle in. No surprises there. Now run the application to ensure that the swap chain is created successfully!

Try removing the `createInfo.imageExtent = extent;` line with validation layers enabled. You'll see that one of the validation layers immediately catches the mistake and a helpful message is printed:

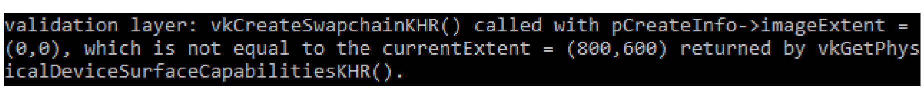

Figure 23:

Retrieving the swap chain images

The swap chain has been created now, so all that remains is retrieving the handles of the `VkImage`s in it. We'll reference these during rendering operations in later chapters. Add a class member to store the handles:

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

The images were created by the implementation for the swap chain and they will be automatically cleaned up once the swap chain has been destroyed, therefore we don't need a deleter here.

I'm adding the code to retrieve the handles to the end of the `createSwapChain` function, right after the `vkCreateSwapchainKHR` call. Retrieving them is very similar to the other times where we retrieved an array of objects from Vulkan. First query the number of images in the swap chain with a call to `vkGetSwapchainImagesKHR`, then resize the container and finally call it again to retrieve the handles.

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

Note that when we created the swap chain, we passed the number of desired images to a field called `minImageCount`. The implementation is allowed to create more images, which is why we need to explicitly query the amount again.

One last thing, store the format and extent we've chosen for the swap chain images in member variables. We'll need them in future chapters.

```
VDeleter<VkSwapchainKHR> swapChain{device, vkDestroySwapchainKHR};
std::vector<VkImage> swapChainImages;
VkFormat swapChainImageFormat;
VkExtent2D swapChainExtent;

...

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

We now have a set of images that can be drawn onto and can be presented to the window. The next two chapters will cover how we can set up the images as render targets and then we start looking into the actual drawing commands!

C++ code

Image views

To use any `VkImage`, including those in the swap chain, in the render pipeline we have to create a `VkImageView` object. An image view is quite literally a view into an image. It describes how to access the image and which part of the image to access, for example if it should be treated as a 2D texture depth texture without any mipmapping levels.

In this chapter we'll write a `createImageViews` function that creates a basic image view for every image in the swap chain so that we can use them as color targets later on.

First add a class member to store the image views in. Unlike the `VkImage`s, the `VkImageView` objects are created by us so we need to clean them up ourselves later.

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

Create the `createImageViews` function and call it right after swap chain creation.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
}

void createImageViews() {

}
```

The first thing we need to do is resize the list to fit all of the image views we'll be creating. This is also the place where we'll actually define the deleter function.

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size(), VDeleter<VkImageView>{device, vkDestr

}
```

The `resize` function initializes all of the list items with the right deleter. Next, set up the loop that iterates over all of the swap chain images.

```
for (uint32_t i = 0; i < swapChainImages.size(); i++) {

}
```

The parameters for image view creation are specified in a `VkImageViewCreateInfo` structure. The first few parameters are straightforward.

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

The `viewType` and `format` fields specify how the image data should be interpreted. The `viewType` parameter allows you to treat images as 1D textures, 2D textures, 3D textures and cube maps.

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

The `components` field allows you to swizzle the color channels around. For example, you can map all of the channels to the red channel for a monochrome texture. You can also map constant values of `0` and `1` to a channel. In our case we'll stick to the default mapping.

```
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;
```

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

```
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 you were working on a stereographic 3D application, then you would create a swap chain with multiple layers. You could then create multiple image views for each image representing the views for the left and right eyes by accessing different layers.

Creating the image view is now a matter of calling `vkCreateImageView`:

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

That's it, now run the program to verify that the image views are created properly and destroyed properly. Checking the latter requires enabling the validation layers, or putting a print statement in the deleter function.

An image view is sufficient to start using an image as a texture, but it's not quite ready to be used as a render target just yet. That requires one more step of indirection, known as a framebuffer. But first we'll have to set up the graphics pipeline.

C++ code

Graphics pipeline basics

Over the course of the next few chapters we'll be setting up a graphics pipeline that is configured to draw our first triangle. 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 simplified overview is displayed below:

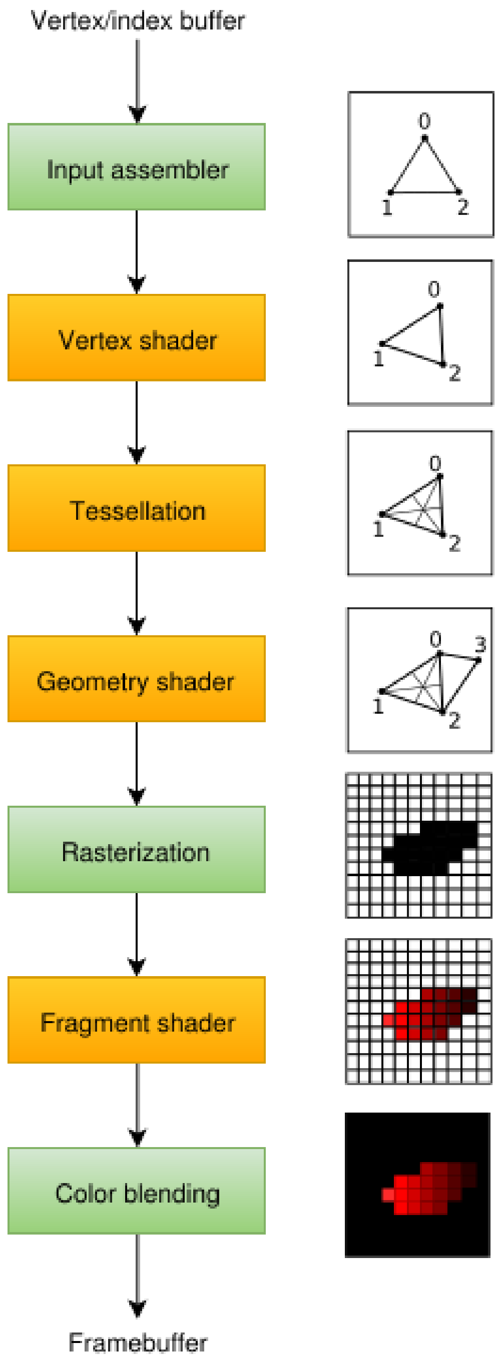
Vertex/index buffer
Input assembler
Vertex shader
Tessellation
Geometry shader
Rasterization
Fragment shader
Color blending
Framebuffer
0
1
2
0
1
2
0
1
2
0
3
1
2

Figure 24:

The *input assembler* collects the raw vertex data from the buffers you specify and may also use an index buffer to repeat certain elements without having to duplicate the vertex data itself.

The *vertex shader* is run for every vertex and generally applies transformations to turn vertex positions from model space to screen space. It also passes per-vertex data down the pipeline.

The *tessellation shaders* allow you to subdivide geometry based on certain rules to increase the mesh quality. This is often used to make surfaces like brick walls and staircases look less flat when they are nearby.

The *geometry shader* is run on every primitive (triangle, line, point) and can discard it or output more primitives than came in. This is similar to the tessellation shader, but much more flexible. However, it is not used much in today's applications because the performance is not that good on most graphics cards except for Intel's integrated GPUs.

The *rasterization* stage discretizes the primitives into *fragments*. These are the pixel elements that they fill on the framebuffer. Any fragments that fall outside the screen are discarded and the attributes outputted by the vertex shader are interpolated across the fragments, as shown in the figure. Usually the fragments that are behind other primitive fragments are also discarded here because of depth testing.

The *fragment shader* is invoked for every fragment that survives and determines which framebuffer(s) the fragments are written to and with which color and depth values. It can do this using the interpolated data from the vertex shader, which can include 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 simply overwrite each other, add up or be mixed based upon transparency.

Stages with a green color are known as *fixed-function* stages. These stages allow you to tweak their operations using parameters, but the way they work is predefined.

Stages with an orange color on the other hand are `programmable`, which means that you can upload your own code to the graphics card to apply exactly the operations you want. This allows you to use fragment shaders, for example, to implement anything from texturing and lighting to ray tracers. These programs run on many GPU cores simultaneously to process many objects, like vertices and fragments in parallel.

If you've used older APIs like OpenGL and Direct3D before, then you'll be used to being able to change any pipeline settings at will with calls like `glBlendFunc` and `OMSetBlendState`. The graphics pipeline in Vulkan is almost completely immutable, so you must recreate the pipeline from scratch if you want to change shaders, bind different framebuffers or change the blend function. The disadvantage is that you'll have to create a number of pipelines that represent all of the different combinations of states you want to use in your rendering operations. However, because all of the operations you'll be doing in the pipeline are known in advance, the driver can optimize for it much better.

Some of the programmable stages are optional based on what you intend to do. For example, the tessellation and geometry stages can be disabled if you are just drawing simple geometry. If you are only interested in depth values then you can disable the fragment shader stage, which is useful for shadow map generation.

In the next chapter we'll first create the two programmable stages required to put a triangle onto the screen: the vertex shader and fragment shader. The fixed-function configuration like blending mode, viewport, rasterization will be set up in the chapter after that. The final part of setting up the graphics pipeline in Vulkan involves the specification of input and output framebuffers.

Create a `createGraphicsPipeline` function that is called right after `createImageViews` in `initVulkan`. We'll work on this function throughout the following chapters.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createGraphicsPipeline();
}

...

void createGraphicsPipeline() {

}
```

C++ code

Shader modules

Unlike earlier APIs, shader code in Vulkan has to be specified in a bytecode format as opposed to human-readable syntax like GLSL and HLSL. This bytecode format is called SPIR-V and is designed to be used with both Vulkan and OpenCL (both Khronos APIs). It is a format that can be used to write graphics and compute shaders, but we will focus on shaders used in Vulkan's graphics pipelines in this tutorial.

The advantage of using a bytecode format is that the compilers written by GPU vendors to turn shader code into native code are significantly less complex. The past has shown that with human-readable syntax like GLSL, some GPU vendors were rather flexible with their interpretation of the standard. If you happen to write non-trivial shaders with a GPU from one of these vendors, then you'd risk other vendor's drivers rejecting your code due to syntax errors, or worse, your shader running differently because of compiler bugs. With a straightforward bytecode format like SPIR-V that will hopefully be avoided.

However, that does not mean that we need to write this bytecode by hand. Khronos has released their own vendor-independent compiler that compiles GLSL to SPIR-V. This compiler is designed to verify that your shader code is fully standards compliant and produces one SPIR-V binary that you can ship with your program. You can also include this compiler as a library to produce SPIR-V at runtime, but we won't be doing that in this tutorial. The compiler is already included with the LunarG SDK as `glslangValidator.exe`, so you don't need to download anything extra.

GLSL is a shading language with a C-style syntax. Programs written in it have a `main` function that is invoked for every object. Instead of using parameters for input and a return value as output, GLSL uses global variables to handle input and output. The language includes many features to aid in graphics programming, like built-in vector and matrix primitives. Functions for operations like cross products, matrix-vector products and reflections around a vector are included. The vector type is called `vec` with a number indicating the amount of elements. For example, a 3D position would be stored in a `vec3`. It is possible to access single components through members like `.x`, but it's also possible to create a new vector from multiple components at the same time. For example, the expression `vec3(1.0, 2.0, 3.0).xy` would result in `vec2`. The constructors of vectors can also take combinations of vector objects and scalar values. For example, a `vec3` can be constructed with `vec3(vec2(1.0, 2.0), 3.0)`.

As the previous chapter mentioned, we need to write a vertex shader and a fragment shader to get a triangle on the screen. The next two sections will cover the GLSL code of each of those and after that I'll show you how to produce two SPIR-V binaries and load them into the program.

Vertex shader

The vertex shader processes each incoming vertex. It takes its attributes, like world position, color, normal and texture coordinates as input. The output is the final position in clip coordinates and the attributes that need to be passed on to the fragment shader, like color and texture coordinates. These values will then be interpolated over the fragments by the rasterizer to produce a smooth gradient.

Clip coordinates are homogeneous coordinates that map the framebuffer to a [-1, 1] by [-1, 1] coordinate system that looks like the following:

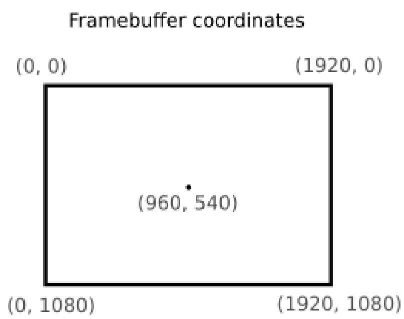

Figure 25:

You should already be familiar with these if you have dabbed in computer graphics before. If you have used OpenGL before, then you'll notice that the sign of the Y coordinates is now flipped. The Z coordinate now uses the same range as it does in Direct3D, from 0 to 1.

For our first triangle we won't be applying any transformations, we'll just specify the positions of the three vertices directly in clip coordinates to create the following shape:

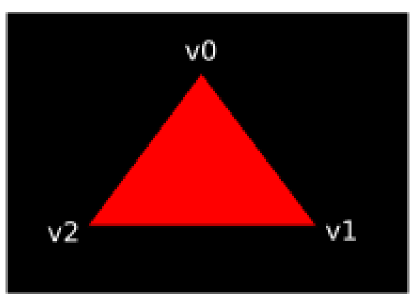

Figure 26:

Normally these coordinates would be stored in a vertex buffer, but creating a vertex buffer in Vulkan and filling it with data is not trivial. Therefore I've decided to postpone that until after we've had the satisfaction of seeing a triangle pop up on the screen. We're going to do something a little unorthodox in the meanwhile: include the coordinates directly inside the vertex shader. The code looks like this:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4 gl_Position;
};

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);
}
```

The `main` function is invoked for every vertex. The built-in `gl_VertexIndex` variable contains the index of the current vertex. This is usually an index into the vertex buffer, but in our case it will be an index into a hardcoded array of vertex data. The position of each vertex is accessed from the constant array in the shader and combined with dummy `z` and `w` components to produce a position in clip coordinates. The built-in variable `gl_Position` functions as the output. The `GL_ARB_separate_shader_objects` extension is required for Vulkan shaders to work.

Fragment shader

The triangle that is formed by the positions from the vertex shader fills an area on the screen with fragments. The fragment shader is invoked on these fragments to produce a color and depth for the framebuffer (or framebuffers). A simple fragment shader that outputs the color red for the entire triangle looks like this:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

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 `main` function is called for every vertex. 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. You have to specify your own output variable for each framebuffer where the `layout(location = 0)` modifier specifies the index of the framebuffer. The color red is written to this `outColor` variable that is linked to the first (and only) framebuffer at index `0`.

Per-vertex colors

Making the entire triangle red is not very interesting, wouldn't something like the following look a lot nicer?

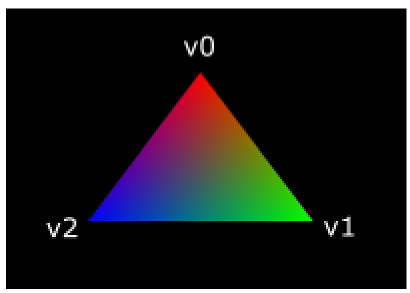

Figure 27:

We have to make a couple of changes to both shaders to accomplish this. First off, we need to specify a distinct color for each of the three vertices. The vertex shader should now include an array with colors just like it does for positions:

```
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)
);
```

Now we just need to pass these per-vertex colors to the fragment shader so it can output their interpolated values to the framebuffer. Add an output for color to the vertex shader and write to it in the `main` function:

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

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

Next, we need to add a matching input in the fragment shader:

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

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

The input variable does not necessarily have to use the same name, they will be linked together using the indexes specified by the `location` directives. The `main` function has been modified to output the color along with an alpha value. As shown in the image above, the values for `fragColor` will be automatically interpolated for the fragments between the three vertices, resulting in a smooth gradient.

Compiling the shaders

Create a directory called `shaders` in the root directory of your project and store the vertex shader in a file called `shader.vert` and the fragment shader in a file called `shader.frag` in that directory. GLSL shaders don't have an official extension, but these two are commonly used to distinguish them.

The contents of `shader.vert` should be:

```glsl
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4 gl_Position;
};

layout(location = 0) out vec3 fragColor;

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

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)
);

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

And the contents of `shader.frag` should be:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

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

We're now going to compile these into SPIR-V bytecode using the `glslangValidator` program.

Windows

Create a `compile.bat` file with the following contents:

```
C:/VulkanSDK/1.0.17.0/Bin32/glslangValidator.exe -V shader.vert
C:/VulkanSDK/1.0.17.0/Bin32/glslangValidator.exe -V shader.frag
pause
```

Replace the path to `glslangValidator.exe` with the path to where you installed the Vulkan SDK. Double click the file to run it.

Linux

Create a `compile.sh` file with the following contents:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslangValidator -V shader.vert
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslangValidator -V shader.frag
```

Replace the path to `glslangValidator` with the path to where you installed the Vulkan SDK. Make the script executable with `chmod +x compile.sh` and run it.

End of platform-specific instructions

These two commands invoke the compiler with the `-V` flag, which tells it to compile the GLSL source files to SPIR-V bytecode. When you run the compile script, you'll see that two SPIR-V binaries are created: `vert.spv` and `frag.spv`. The names are automatically derived from the type of shader, but you can rename them to anything you like. You may get a warning about some missing features when compiling your shaders, but you can safely ignore that.

If your shader contains a syntax error then the compiler will tell you the line number and problem, as you would expect. Try leaving out a semicolon for example and run the compile script again. Also try running the compiler without any arguments to see what kinds of flags it supports. It can, for example, also output the bytecode into a human-readable format so you can see exactly what your shader is doing and any optimizations that have been applied at this stage.

Loading a shader

Now that we have a way of producing SPIR-V shaders, it's time to load them into our program to plug them into the graphics pipeline at some point. We'll first write a simple helper function to load the binary data from the files.

```
#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!");
    }
}
```

The `readFile` function will read all of the bytes from the specified file and return them in a byte array managed by `std::vector`. We start by opening the file with two flags:

- `ate`: Start reading at the end of the file
- `binary`: Read the file as binary file (avoid text transformations)

The advantage of starting to read at the end of the file is that we can use the read position to determine the size of the file and allocate a buffer:

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

After that, we can seek back to the beginning of the file and read all of the bytes at once:

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

And finally close the file and return the bytes:

```
file.close();

return buffer;
```

We'll now call this function from `createGraphicsPipeline` to load the bytecode of the two shaders:

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

Make sure that the shaders are loaded correctly by printing the size of the buffers and checking if they match the actual file size in bytes.

Creating shader modules

Before we can pass the code to the pipeline, we have to wrap it in a `VkShaderModule` object. Let's create a helper function `createShaderModule` to do that.

```
void createShaderModule(const std::vector<char>& code, VDeleter<VkShaderModule>& shaderModul

}
```

The function will take a buffer with the bytecode as parameter and create a `VkShaderModule` from it. Instead of returning this handle directly, it's written to the variable specified for the second parameter, which makes it easier to wrap it in a deleter variable when calling `createShaderModule`.

Creating a shader module is simple, we only need to specify a pointer to the buffer with the bytecode and the length of it. This information is specified in a `VkShaderModuleCreateInfo` structure. The one catch is that the size of the bytecode is specified in bytes, but the bytecode pointer is a `uint32_t` pointer rather than a `char` pointer. Therefore we need to temporarily copy the bytecode to a container that has the right alignment for `uint32_t`:

```
VkShaderModuleCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
createInfo.codeSize = code.size();

std::vector<uint32_t> codeAligned(code.size() / sizeof(uint32_t) + 1);
memcpy(codeAligned.data(), code.data(), code.size());
createInfo.pCode = codeAligned.data();
```

The `VkShaderModule` can then be created with a call to `vkCreateShaderModule`:

```
if (vkCreateShaderModule(device, &createInfo, nullptr, shaderModule.replace()) != VK_SUCCESS
    throw std::runtime_error("failed to create shader module!");
```

```
}
```

The parameters are the same as those in previous object creation functions: the logical device, pointer to create info structure, optional pointer to custom allocators and handle output variable. The buffer with the code can be freed immediately after creating the shader module.

The shader module objects are only required during the pipeline creation process, so instead of declaring them as class members, we'll make them local variables in the `createGraphicsPipeline` function:

```
VDeleter<VkShaderModule> vertShaderModule{device, vkDestroyShaderModule};
VDeleter<VkShaderModule> fragShaderModule{device, vkDestroyShaderModule};
```

They will be automatically cleaned up when the graphics pipeline has been created and `createGraphicsPipeline` returns. Now just call the helper function we created and we're done:

```
createShaderModule(vertShaderCode, vertShaderModule);
createShaderModule(fragShaderCode, fragShaderModule);
```

Shader stage creation

The `VkShaderModule` object is just a dumb wrapper around the bytecode buffer. The shaders aren't linked to each other yet and they haven't even been given a purpose yet. Assigning a shader module to either the vertex or fragment shader stage in the pipeline happens through a `VkPipelineShaderStageCreateInfo` structure, which is part of the actual pipeline creation process.

We'll start by filling in the structure 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 step, besides the obligatory `sType` member, is telling Vulkan in which pipeline stage the shader is going to be used. There is an enum value for each of the programmable stages described in the previous chapter.

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

The next two members specify the shader module containing the code, and the function to invoke. That means that it's possible to combine multiple fragment shaders into a single shader module and use different entry points to differentiate between their behaviors. In this case we'll stick to the standard `main`, however.

There is one more (optional) member, `pSpecializationInfo`, which we won't be using here, but is worth discussing. It allows you to specify values for shader constants. You can use a single shader module where its behavior can be configured at pipeline creation by specifying different values for the constants used in it. This is more efficient than configuring the shader using variables at render time, because the compiler can do optimizations like eliminating `if` statements that depend on these values. If you don't have any constants like that, then you can set the member to `nullptr`, which our struct initialization does automatically.

Modifying the structure to suit the fragment shader is easy:

```
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";
```

Finish by defining an array that contains these two structs, which we'll later use to reference them in the actual pipeline creation step.

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

That's all there is to describing the programmable stages of the pipeline. In the next chapter we'll look at the fixed-function stages.

C++ code / Vertex shader / Fragment shader

Fixed functions

The older graphics APIs provided default state for most of the stages of the graphics pipeline. In Vulkan you have to be explicit about everything, from viewport size to color blending function. In this chapter we'll fill in all of the structures to configure these fixed-function operations.

Vertex input

The `VkPipelineVertexInputStateCreateInfo` structure describes the format of the vertex data that will be passed to the vertex shader. It describes this in roughly two ways:

- Bindings: spacing between data and whether the data is per-vertex or per-instance (see instancing)
- Attribute descriptions: type of the attributes passed to the vertex shader, which binding to load them from and at which offset

Because we're hard coding the vertex data directly in the vertex shader, we'll fill in this structure to specify that there is no vertex data to load for now. We'll get back to it in the vertex buffer chapter.

```
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
```

The `pVertexBindingDescriptions` and `pVertexAttributeDescriptions` members point to an array of structs that describe the aforementioned details for loading vertex data. Add this structure to the `createGraphicsPipeline` function right after the `shaderStages` array.

Input assembly

The `VkPipelineInputAssemblyStateCreateInfo` struct describes two things: what kind of geometry will be drawn from the vertices and if primitive restart should be enabled. The former is specified in 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`: every second vertex is used as start vertex for the next line
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`: triangle from every 3 vertices without reuse
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`: every third vertex is used as first vertex for 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 you to perform optimizations like reusing vertices. If you set the `primitiveRestartEnable` member to `VK_TRUE`, then it's possible to break up lines and triangles in the `_STRIP` topology modes by using a special index of `0xFFFF` or `0xFFFFFFFF`.

We intend to draw triangles throughout this tutorial, so we'll stick to the following data for the structure:

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

Viewports and scissors

A viewport basically describes the region of the framebuffer that the output will be rendered to. This will almost always be `(0, 0)` to `(width, height)` and in this tutorial that will also be the case.

```
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;
```

Remember that the size of the swap chain and its images may differ from the `WIDTH` and `HEIGHT` of the window. The swap chain images will be used as framebuffers later on, so we should stick to their size.

The `minDepth` and `maxDepth` values specify the range of depth values to use for the framebuffer. These values must be within the `[0.0f, 1.0f]` range, but `minDepth` may be higher than `maxDepth`. If you aren't doing anything special, then you should stick to the standard values of `0.0f` and `1.0f`.

While viewports define the transformation from the image to the framebuffer, scissor rectangles define in which regions pixels will actually be stored. Any pixels outside the scissor rectangles will be discarded by the rasterizer. They function like a filter rather than a transformation. The difference is illustrated below. Note that the left scissor rectangle is just one of the many possibilities that would result in that image, as long as it's larger than the viewport.

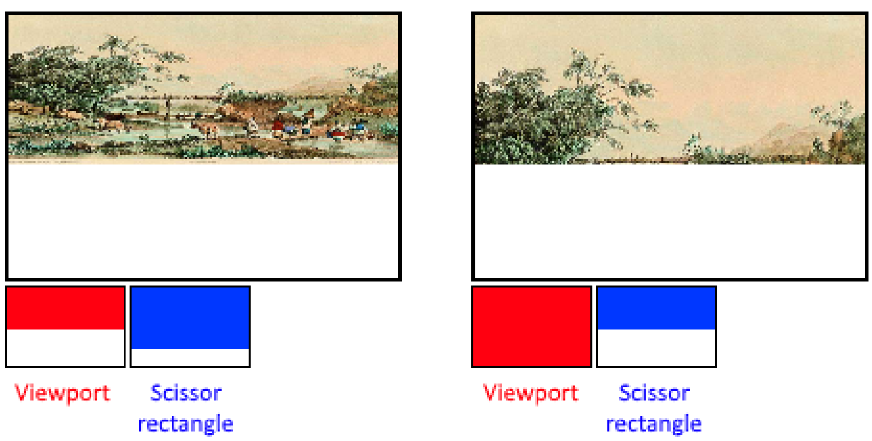

Figure 28:

In this tutorial we simply want to draw to the entire framebuffer, so we'll specify a scissor rectangle that covers it entirely:

```
VkRect2D scissor = {};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
```

Now this viewport and scissor rectangle need to be combined into a viewport state using the `VkPipelineViewportStateCreateInfo` struct. It is possible to use multiple viewports and scissor rectangles on some graphics cards, so its members reference an array of them. Using multiple requires enabling a GPU feature (see logical device creation).

```
VkPipelineViewportStateCreateInfo viewportState = {};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportState.viewportCount = 1;
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

Rasterizer

The rasterizer takes the geometry that is shaped by the vertices from the vertex shader and turns it into fragments to be colored by the fragment shader. It also performs depth testing, face culling and the scissor test, and it can be configured to output fragments that fill entire polygons or just the edges (wireframe rendering). All this is configured using the `VkPipelineRasterizationStateCreateInfo` structure.

```
VkPipelineRasterizationStateCreateInfo rasterizer = {};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
```

If `depthClampEnable` is set to `VK_TRUE`, then fragments that are beyond the near and far planes are clamped to them as opposed to discarding them. This is useful in some special cases like shadow maps. Using this requires enabling a GPU feature.

```
rasterizer.rasterizerDiscardEnable = VK_FALSE;
```

If `rasterizerDiscardEnable` is set to `VK_TRUE`, then geometry never passes through the rasterizer stage. This basically disables any output to the framebuffer.

```
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
```

The `polygonMode` determines how fragments are generated for geometry. The following modes are available:

- `VK_POLYGON_MODE_FILL`: fill the area of the polygon with fragments

- `VK_POLYGON_MODE_LINE`: polygon edges are drawn as lines
- `VK_POLYGON_MODE_POINT`: polygon vertices are drawn as points

Using any mode other than fill requires enabling a GPU feature.

```
rasterizer.lineWidth = 1.0f;
```

The `lineWidth` member is straightforward, it describes the thickness of lines in terms of number of fragments. The maximum line width that is supported depends on the hardware and any line thicker than `1.0f` requires you to enable the `wideLines` GPU feature.

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
```

The `cullMode` variable determines the type of face culling to use. You can disable culling, cull the front faces, cull the back faces or both. The `frontFace` variable specifies the vertex order for faces to be considered front-facing and can be clockwise or counterclockwise.

```
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

The rasterizer can alter the depth values by adding a constant value or biasing them based on a fragment's slope. This is sometimes used for shadow mapping, but we won't be using it. Just set `depthBiasEnable` to `VK_FALSE`.

Multisampling

The `VkPipelineMultisampleStateCreateInfo` struct configures multisampling, which is one of the ways to perform anti-aliasing. It works by combining the fragment shader results of multiple polygons that rasterize to the same pixel. This mainly occurs along edges, which is also where the most noticeable aliasing artifacts occur. Because it doesn't need to run the fragment shader multiple times if only one polygon maps to a pixel, it is significantly less expensive than simply rendering to a higher resolution and then downscaling. Enabling it requires enabling a GPU feature.

```
VkPipelineMultisampleStateCreateInfo multisampling = {};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampling.sampleShadingEnable = VK_FALSE;
multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optional
multisampling.pSampleMask = nullptr; /// Optional
multisampling.alphaToCoverageEnable = VK_FALSE; // Optional
multisampling.alphaToOneEnable = VK_FALSE; // Optional
```

In this tutorial we'll not be using multisampling, but feel free to experiment with it. See the specification for the meaning of each parameter.

Depth and stencil testing

If you are using a depth and/or stencil buffer, then you also need to configure the depth and stencil tests using `VkPipelineDepthStencilStateCreateInfo`. We don't have one right now, so we can simply pass a `nullptr` instead of a pointer to such a struct. We'll get back to it in the depth buffering chapter.

Color blending

After a fragment shader has returned a color, it needs to be combined with the color that is already in the framebuffer. This transformation is known as color blending and there are two ways to do it:

- Mix the old and new value to produce a final color
- Combine the old and new value using a bitwise operation

There are two types of structs to configure color blending. The first struct, `VkPipelineColorBlendAttachmentState` contains the configuration per attached framebuffer and the second struct, `VkPipelineColorBlendStateCreateInfo` contains the *global* color blending settings. In our case we only have one framebuffer:

```
VkPipelineColorBlendAttachmentState colorBlendAttachment = {};
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
colorBlendAttachment.blendEnable = VK_FALSE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```

This per-framebuffer struct allows you to configure the first way of color blending. The operations that will be performed are best demonstrated using the following pseudocode:

```
if (blendEnable) {
    finalColor.rgb = (srcColorBlendFactor * newColor.rgb) <colorBlendOp> (dstColorBlendFacto
    finalColor.a = (srcAlphaBlendFactor * newColor.a) <alphaBlendOp> (dstAlphaBlendFactor *
} else {
    finalColor = newColor;
}
```

```
finalColor = finalColor & colorWriteMask;
```

If `blendEnable` is set to `VK_FALSE`, then the new color from the fragment shader is passed through unmodified. Otherwise, the two mixing operations are performed to compute a new color. The resulting color is AND'd with the `colorWriteMask` to determine which channels are actually passed through.

The most common way to use color blending is to implement alpha blending, where we want the new color to be blended with the old color based on its opacity. The `finalColor` should then be computed as follows:

```
finalColor.rgb = newAlpha * newColor + (1 - newAlpha) * oldColor;
finalColor.a = newAlpha.a;
```

This can be accomplished with the following parameters:

```
colorBlendAttachment.blendEnable = VK_TRUE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;
```

You can find all of the possible operations in the `VkBlendFactor` and `VkBlendOp` enumerations in the specification.

The second structure references the array of structures for all of the framebuffers and allows you to set blend constants that you can use as blend factors in the aforementioned calculations.

```
VkPipelineColorBlendStateCreateInfo colorBlending = {};
colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlending.logicOpEnable = VK_FALSE;
colorBlending.logicOp = VK_LOGIC_OP_COPY; // Optional
colorBlending.attachmentCount = 1;
colorBlending.pAttachments = &colorBlendAttachment;
colorBlending.blendConstants[0] = 0.0f; // Optional
colorBlending.blendConstants[1] = 0.0f; // Optional
colorBlending.blendConstants[2] = 0.0f; // Optional
colorBlending.blendConstants[3] = 0.0f; // Optional
```

If you want to use the second method of blending (bitwise combination), then you should set `logicOpEnable` to `VK_TRUE`. The bitwise operation can then be specified in the `logicOp` field. Note that this will automatically disable the first method, as if you had set `blendEnable` to `VK_FALSE` for every attached framebuffer! The `colorWriteMask` will also be used in this mode to determine which channels in the framebuffer will actually be affected. It is also possible to disable both modes, as we've done here, in which case the fragment colors will be written to the framebuffer unmodified.

Dynamic state

A limited amount of the state that we've specified in the previous structs *can* actually be changed without recreating the pipeline. Examples are the size of the viewport, line width and blend constants. If you want to do that, then you'll have to fill in a `VkPipelineDynamicStateCreateInfo` structure like this:

```
VkDynamicState dynamicStates[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo dynamicState = {};
dynamicState.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.dynamicStateCount = 2;
dynamicState.pDynamicStates = dynamicStates;
```

This will cause the configuration of these values to be ignored and you will be required to specify the data at drawing time. We'll get back to this in a future chapter. This struct can be substituted by a `nullptr` later on if you don't have any dynamic state.

Pipeline layout

You can use `uniform` values in shaders, which are globals similar to dynamic state variables that can be changed at drawing time to alter the behavior of your shaders without having to recreate them. They are commonly used to pass the transformation matrix to the vertex shader, or to create texture samplers in the fragment shader.

These uniform values need to be specified during pipeline creation by creating a `VkPipelineLayout` object. Even though we won't be using them until a future chapter, we are still required to create an empty pipeline layout.

Create a class member to hold this object, because we'll refer to it from other functions at a later point in time:

```
VDeleter<VkPipelineLayout> pipelineLayout{device, vkDestroyPipelineLayout};
```

And then create the object in the `createGraphicsPipeline` function:

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optional
```

```
pipelineLayoutInfo.pSetLayouts = nullptr; // Optional
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = 0; // Optional

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr,
    pipelineLayout.replace()) != VK_SUCCESS) {
    throw std::runtime_error("failed to create pipeline layout!");
}
```

The structure also specifies *push constants*, which are another way of passing dynamic values to shaders that we'll get into later.

Conclusion

That's it for all of the fixed-function state! It's a lot of work to set all of this up from scratch, but the advantage is that we're now nearly fully aware of everything that is going on in the graphics pipeline! This reduces the chance of running into unexpected behavior because the default state of certain components is not what you expect.

There is however one more object to create before we can finally create the graphics pipeline and that is a render pass.

C++ code / Vertex shader / Fragment shader

Render passes

Setup

Before we can finish creating the pipeline, we need to tell Vulkan about the framebuffer attachments that will be used while rendering. We need to specify how many color and depth buffers there will be, how many samples to use for each of them and how their contents should be handled throughout the rendering operations. All of this information is wrapped in a *render pass* object, for which we'll create a new `createRenderPass` function. Call this function from `initVulkan` before `createGraphicsPipeline`.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
```

```
    createRenderPass();
    createGraphicsPipeline();
}

...

void createRenderPass() {

}
```

Attachment description

In our case we'll have just a single color buffer attachment represented by one of the images from the swap chain.

```
void createRenderPass() {
    VkAttachmentDescription colorAttachment = {};
    colorAttachment.format = swapChainImageFormat;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
}
```

The `format` of the color attachment should match the one of the swap chain images and we're not doing anything with multisampling, so we stick to 1 sample.

```
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
```

The `loadOp` and `storeOp` determine what to do with the data in the attachment before rendering and after rendering. We have the following choices for `loadOp`:

- `VK_ATTACHMENT_LOAD_OP_LOAD`: Preserve the existing contents of the attachment
- `VK_ATTACHMENT_LOAD_OP_CLEAR`: Clear the values to a constant at the start
- `VK_ATTACHMENT_LOAD_OP_DONT_CARE`: Existing contents are undefined; we don't care about them

In our case we're going to use the clear operation to clear the framebuffer to black before drawing a new frame. There are only two possibilities for the `storeOp`:

- `VK_ATTACHMENT_STORE_OP_STORE`: Rendered contents will be stored in memory and can be read later
- `VK_ATTACHMENT_STORE_OP_DONT_CARE`: Contents of the framebuffer will be undefined after the rendering operation

We're interested in seeing the rendered triangle on the screen, so we're going with the store operation here.

```
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

The `loadOp` and `storeOp` apply to color and depth data, and `stencilLoadOp` / `stencilStoreOp` apply to stencil data. Our application won't do anything with the stencil buffer, so the results of loading and storing are irrelevant.

```
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Textures and framebuffers in Vulkan are represented by `VkImage` objects with a certain pixel format, however the layout of the pixels in memory can change based on what you're trying to do with an image.

Some of the most common layouts are:

- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`: Images used as color attachment
- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`: Images to be presented in the swap chain
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`: Images to be used as destination for a memory copy operation

We'll discuss this topic in more depth in the texturing chapter, but what's important to know right now is that images need to be transitioned to specific layouts that are suitable for the operation that they're going to be involved in next.

The `initialLayout` specifies which layout the image will have before the render pass begins. The `finalLayout` specifies the layout to automatically transition to when the render pass finishes. Using `VK_IMAGE_LAYOUT_UNDEFINED` for `initialLayout` means that we don't care what previous layout the image was in. The caveat of this special value is that the contents of the image are not guaranteed to be preserved, but that doesn't matter since we're going to clear it anyway. We want the image to be ready for presentation using the swap chain after rendering, which is why we use `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` as `finalLayout`.

Subpasses and attachment references

A single render pass can consist of multiple subpasses. Subpasses are subsequent rendering operations that depend on the contents of framebuffers in previous passes, for example a sequence of post-processing effects that are applied one after another. If you group these rendering operations into one render pass, then Vulkan is able to reorder the operations and conserve memory bandwidth for possibly better performance. For our very first triangle, however, we'll stick to a single subpass.

Every subpass references one or more of the attachments that we've described using the structure in the previous sections. These references are themselves `VkAttachmentReference` structs that look like this:

```
VkAttachmentReference colorAttachmentRef = {};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

The `attachment` parameter specifies which attachment to reference by its index in the attachment descriptions array. Our array consists of a single `VkAttachmentDescription`, so its index is `0`. The `layout` specifies which layout we would like the attachment to have during a subpass that uses this reference. Vulkan will automatically transition the attachment to this layout when the subpass is started. We intend to use the attachment to function as a color buffer and the `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` layout will give us the best performance, as its name implies.

The subpass is described using a `VkSubpassDescription` structure:

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

Vulkan may also support compute subpasses in the future, so we have to be explicit about this being a graphics subpass. Next, we specify the reference to the color attachment:

```
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
```

The index of the attachment in this array is directly referenced from the fragment shader with the `layout(location = 0) out vec4 outColor` directive!

The following other types of attachments can be referenced by a subpass:

- `pInputAttachments`: Attachments that are read from a shader
- `pResolveAttachments`: Attachments used for multisampling color attachments
- `pDepthStencilAttachment`: Attachments for depth and stencil data
- `pPreserveAttachments`: Attachments that are not used by this subpass, but for which the data must be preserved

Render pass

Now that the attachment and a basic subpass referencing it have been described, we can create the render pass itself. Create a new class member variable to hold the `VkRenderPass` object right above the `pipelineLayout` variable:

```
VDeleter<VkRenderPass> renderPass{device, vkDestroyRenderPass};
```

The render pass object can then be created by filling in the `VkRenderPassCreateInfo` structure with an array of attachments and subpasses. The `VkAttachmentReference` objects reference attachments using the indices of this array.

```
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = &colorAttachment;
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;

if (vkCreateRenderPass(device, &renderPassInfo, nullptr, renderPass.replace()) != VK_SUCCESS
    throw std::runtime_error("failed to create render pass!");
}
```

That was a lot of work, but in the next chapter it all comes together to finally create the graphics pipeline object!

C++ code / Vertex shader / Fragment shader

Graphics pipeline construction

We can now combine all of the structures and objects from the previous chapters to create the graphics pipeline! Here's the types of objects we have now, as a quick recap:

- Shader stages: the shader modules that define the functionality of the programmable stages of the graphics pipeline
- Fixed-function state: all of the structures that define the fixed-function stages of the pipeline, like input assembly, rasterizer, viewport and color blending
- Pipeline layout: the uniform and push values referenced by the shader that can be updated at draw time
- Render pass: the attachments referenced by the pipeline stages and their usage

All of these combined fully define the functionality of the graphics pipeline, so we can now begin filling in the `VkGraphicsPipelineCreateInfo` structure at the end of the `createGraphicsPipeline` function.

```
VkGraphicsPipelineCreateInfo pipelineInfo = {};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
```

We start by referencing the array of `VkPipelineShaderStageCreateInfo` structs.

```
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optional
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.pDynamicState = nullptr; // Optional
```

Then we reference all of the structures describing the fixed-function stage.

```
pipelineInfo.layout = pipelineLayout;
```

After that comes the pipeline layout, which is a Vulkan handle rather than a struct pointer.

```
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
```

And finally we have the reference to the render pass and the index of the sub pass where this graphics pipeline will be used.

```
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE; // Optional
pipelineInfo.basePipelineIndex = -1; // Optional
```

There are actually two more parameters: `basePipelineHandle` and `basePipelineIndex`. Vulkan allows you to create a new graphics pipeline by deriving from an existing pipeline. The idea of pipeline derivatives is that it is less expensive to set up pipelines when they have much functionality in common with an existing pipeline and switching between pipelines from the same parent can also be done quicker. You can either specify the handle of an existing pipeline with `basePipelineHandle` or reference another pipeline that is about to be created by index with `basePipelineIndex`. Right now there is only a single pipeline, so we'll simply specify a null handle and an invalid index. These values are only used if the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag is also specified in the `flags` field of `VkGraphicsPipelineCreateInfo`.

Now prepare for the final step by creating a class member to hold the `VkPipeline` object:

```
VDeleter<VkPipeline> graphicsPipeline{device, vkDestroyPipeline};
```

And finally create the graphics pipeline:

```
if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, graphicsPip
    throw std::runtime_error("failed to create graphics pipeline!");
}
```

The `vkCreateGraphicsPipelines` function actually has more parameters than the usual object creation functions in Vulkan. It is designed to take multiple `VkGraphicsPipelineCreateInfo` objects and create multiple `VkPipeline` objects in a single call.

The second parameter, for which we've passed the `VK_NULL_HANDLE` argument, references an optional `VkPipelineCache` object. A pipeline cache can be used to store and reuse data relevant to pipeline creation across multiple calls to `vkCreateGraphicsPipelines` and even across program executions if the cache is stored to a file. This makes it possible to significantly speed up pipeline creation at a later time. We'll get into this in the pipeline cache chapter.

Now run your program to confirm that all this hard work has resulted in a successful pipeline creation! We are already getting quite close to seeing something pop up on the screen. In the next couple of chapters we'll set up the actual framebuffers from the swap chain images and prepare the drawing commands.

C++ code / Vertex shader / Fragment shader

Framebuffers

We've talked a lot about framebuffers in the past few chapters and we've set up the render pass to expect a single framebuffer with the same format as the swap chain images, but we haven't actually created any yet.

The attachments specified during render pass creation are bound by wrapping them into a `VkFramebuffer` object. A framebuffer object references all of the `VkImageView` objects that represent the attachments. In our case that will be only a single one: the color attachment. However, the image that we have to use as attachment depends on which image the swap chain returns when we retrieve one for presentation. That means that we have to create a framebuffer for all of the images in the swap chain and use the one that corresponds to the retrieved image at drawing time.

To that end, create another `std::vector` class member to hold the framebuffers:

```
std::vector<VDeleter<VkFramebuffer>> swapChainFramebuffers;
```

We'll create the objects for this array in a new function `createFramebuffers` that is called from `initVulkan` right after creating the graphics pipeline:

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
```

```
    createFramebuffers();
}

...

void createFramebuffers() {

}
```

Start by resizing the container to hold all of the framebuffers:

```
void createFramebuffers() {
    swapChainFramebuffers.resize(swapChainImageViews.size(), VDeleter<VkFramebuffer>{device,
}
```

We'll then iterate through the image views and create framebuffers from them:

```
for (size_t i = 0; i < swapChainImageViews.size(); i++) {
    VkImageView attachments[] = {
        swapChainImageViews[i]
    };

    VkFramebufferCreateInfo framebufferInfo = {};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = 1;
    framebufferInfo.pAttachments = attachments;
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, swapChainFramebuffers[i].repl
        throw std::runtime_error("failed to create framebuffer!");
    }
}
```

As you can see, creation of framebuffers is quite straightforward. We first need to specify with which `renderPass` the framebuffer needs to be compatible. You can only use a framebuffer with the render passes that it is compatible with, which roughly means that they use the same number and type of attachments.

The `attachmentCount` and `pAttachments` parameters specify the `VkImageView` objects that should be bound to the respective attachment descriptions in the render pass `pAttachment` array.

The `width` and `height` parameters are self-explanatory and `layers` refers to the number of layers in image arrays. Our swap chain images are single images, so the number of layers is `1`.

We've now reached the milestone where we have all of the objects that are required for rendering. In the next chapter we're going to write the first actual drawing commands.

C++ code / Vertex shader / Fragment shader

Command buffers

Commands in Vulkan, like drawing operations and memory transfers, are not executed directly using function calls. You have to record all of the operations you want to perform in command buffer objects. The advantage of this is that all of the hard work of setting up the drawing commands can be done in advance and in multiple threads. After that, you just have to tell Vulkan to execute the commands in the main loop.

Command pools

We have to create a command pool before we can create command buffers. Command pools manage the memory that is used to store the buffers and command buffers are allocated from them. Add a new class member to store a `VkCommandPool`:

```
VDeleter<VkCommandPool> commandPool{device, vkDestroyCommandPool};
```

Then create a new function `createCommandPool` and call it from `initVulkan` after the framebuffers were created.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
}

...

void createCommandPool() {

}
```

Command pool creation only takes two parameters:

```
QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

VkCommandPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily;
poolInfo.flags = 0; // Optional
```

Command buffers are executed by submitting them on one of the device queues, like the graphics and presentation queues we retrieved. Each command pool can only allocate command buffers that are submitted on a single type of queue. We're going to record commands for drawing, which is why we've chosen the graphics queue family.

There are two possible flags for command pools:

- `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT`: Hint that command buffers are rerecorded with new commands very often (may change memory allocation behavior)
- `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`: Allow command buffers to be rerecorded individually, without this flag they all have to be reset together

We will only record the command buffers at the beginning of the program and then execute them many times in the main loop, so we're not going to use either of these flags.

```
if (vkCreateCommandPool(device, &poolInfo, nullptr, commandPool.replace()) != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
}
```

Finish creating the command pool using the `vkCreateCommandPool` function. It doesn't have any special parameters.

Command buffer allocation

We can now start allocating command buffers and recording drawing commands in them. Because one of the drawing commands involves binding the right `VkFramebuffer`, we'll actually have to record a command buffer for every image in the swap chain once again. To that end, create a list of `VkCommandBuffer` objects as class member. Command buffers will be automatically freed when their command pool is destroyed, so we don't need a `VDeleter`.

```
std::vector<VkCommandBuffer> commandBuffers;
```

We'll now start working on a `createCommandBuffers` function that allocates and records the commands for each swap chain image.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
}

...

void createCommandBuffers() {
    commandBuffers.resize(swapChainFramebuffers.size());
}
```

Cleaning up command buffers involves a slightly different function than other objects. The `vkFreeCommandBuffers` function takes the command pool and an array of command buffers as parameters.

Command buffers are allocated with the `vkAllocateCommandBuffers` function, which takes a `VkCommandBufferAllocateInfo` struct as parameter that specifies the command pool and number of buffers to allocate:

```
VkCommandBufferAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = (uint32_t) commandBuffers.size();

if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate command buffers!");
}
```

The `level` parameter specifies if the allocated command buffers are primary or secondary command buffers.

- `VK_COMMAND_BUFFER_LEVEL_PRIMARY`: Can be submitted to a queue for execution, but cannot be called from other command buffers.
- `VK_COMMAND_BUFFER_LEVEL_SECONDARY`: Cannot be submitted directly, but can be called from primary command buffers.

We won't make use of the secondary command buffer functionality here, but you can imagine that it's helpful to reuse common operations from primary command buffers.

Starting command buffer recording

We begin recording a command buffer by calling `vkBeginCommandBuffer` with a small `VkCommandBufferBeginInfo` structure as argument that specifies some details about the usage of this specific command buffer.

```
for (size_t i = 0; i < commandBuffers.size(); i++) {
    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;
    beginInfo.pInheritanceInfo = nullptr; // Optional

    vkBeginCommandBuffer(commandBuffers[i], &beginInfo);
}
```

The `flags` parameter specifies how we're going to use the command buffer. The following values are available:

- `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`: The command buffer will be rerecorded right after executing it once.
- `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`: This is a secondary command buffer that will be entirely within a single render pass.
- `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`: The command buffer can be resubmitted while it is also already pending execution.

We have used the last flag because we may already be scheduling the drawing commands for the next frame while the last frame is not finished yet. The `pInheritanceInfo` parameter is only relevant for secondary command buffers. It specifies which state to inherit from the calling primary command buffers.

If the command buffer was already recorded once, then a call to `vkBeginCommandBuffer` will implicitly reset it. It's not possible to append commands to a buffer at a later time.

Starting a render pass

Drawing starts by beginning the render pass with `vkCmdBeginRenderPass`. The render pass is configured using some parameters in a `VkRenderPassBeginInfo` struct.

```
VkRenderPassBeginInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
```

The first parameters are the render pass itself and the attachments to bind. We created a framebuffer for each swap chain image that specifies it as color attachment.

```
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;
```

The next two parameters define the size of the render area. The render area defines where shader loads and stores will take place. The pixels outside this region will have undefined values. It should match the size of the attachments for best performance.

```
VkClearValue clearColor = {0.0f, 0.0f, 0.0f, 1.0f};
renderPassInfo.clearValueCount = 1;
renderPassInfo.pClearValues = &clearColor;
```

The last two parameters define the clear values to use for `VK_ATTACHMENT_LOAD_OP_CLEAR`, which we used as load operation for the color attachment. I've defined the clear color to simply be black with 100% opacity.

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

The render pass can now begin. All of the functions that record commands can be recognized by their `vkCmd` prefix. They all return `void`, so there will be no error handling until we've finished recording.

The first parameter for every command is always the command buffer to record the command to. The second parameter specifies the details of the render pass we've just provided. The final parameter controls how the drawing commands within the render pass will be provided. It can have one of two values:

- `VK_SUBPASS_CONTENTS_INLINE`: The render pass commands will be embedded in the primary command buffer itself and no secondary command buffers will be executed.
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS`: The render pass commands will be executed from secondary command buffers.

We will not be using secondary command buffers, so we'll go with the first option.

Basic drawing commands

We can now bind the graphics pipeline:

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
```

The second parameter specifies if the pipeline object is a graphics or compute pipeline. We've now told Vulkan which operations to execute in the graphics pipeline and which attachment to use in the fragment shader, so all that remains is telling it to draw the triangle:

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

The actual vkCmdDraw function is a bit anticlimactic, but it's so simple because of all the information we specified in advance. It has the following parameters, aside from the command buffer:

- vertexCount: Even though we don't have a vertex buffer, we technically still have 3 vertices to draw.
- instanceCount: Used for instanced rendering, use 1 if you're not doing that.
- firstVertex: Used as an offset into the vertex buffer, defines the lowest value of gl_VertexIndex.
- firstInstance: Used as an offset for instanced rendering, defines the lowest value of gl_InstanceIndex.

Finishing up

The render pass can now be ended:

```
vkCmdEndRenderPass(commandBuffers[i]);
```

And we've finished recording the command buffer:

```
if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {
    throw std::runtime_error("failed to record command buffer!");
}
```

In the next chapter we'll write the code for the main loop, which will acquire an image from the swap chain, execute the right command buffer and return the finished image to the swap chain.

C++ code / Vertex shader / Fragment shader

Rendering and presentation

Setup

This is the chapter where everything is going to come together. We're going to write the drawFrame function that will be called from the main loop to put the triangle on the screen. Create the function and call it from mainLoop:

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

    glfwDestroyWindow(window);
```

```
}

...

void drawFrame() {

}
```

Synchronization

The `drawFrame` function will perform the following operations:

- Acquire an image from the swap chain
- Execute the command buffer with that image as attachment in the framebuffer
- Return the image to the swap chain for presentation

Each of these events is set in motion using a single function call, but they are executed asynchronously. The function calls will return before the operations are actually finished and the order of execution is also undefined. That is unfortunate, because each of the operations depends on the previous one finishing.

There are two ways of synchronizing swap chain events: fences and semaphores. They're both objects that can be used for coordinating operations by having one operation signal and another operation wait for a fence or semaphore to go from the unsignaled to signaled state.

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

Semaphores

We'll need one semaphore to signal that an image has been acquired and is ready for rendering, and another one to signal that rendering has finished and presentation can happen. Create two class members to store these semaphore objects:

```
VDeleter<VkSemaphore> imageAvailableSemaphore{device, vkDestroySemaphore};
VDeleter<VkSemaphore> renderFinishedSemaphore{device, vkDestroySemaphore};
```

To create the semaphores, we'll add the last `create` function for this part of the tutorial: `createSemaphores`:

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
    createSemaphores();
}

...

void createSemaphores() {

}
```

Creating semaphores requires filling in the `VkSemaphoreCreateInfo`, but in the current version of the API it doesn't actually have any required fields besides `sType`:

```
void createSemaphores() {
    VkSemaphoreCreateInfo semaphoreInfo = {};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
}
```

Future versions of the Vulkan API or extensions may add functionality for the `flags` and `pNext` parameters like it does for the other structures. Creating the semaphores follows the familiar pattern with `vkCreateSemaphore`:

```
if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, imageAvailableSemaphore.replace()) !=
    vkCreateSemaphore(device, &semaphoreInfo, nullptr, renderFinishedSemaphore.replace()) !=

    throw std::runtime_error("failed to create semaphores!");
}
```

Acquiring an image from the swap chain

As mentioned before, the first thing we need to do in the `drawFrame` function is acquiring an image from the swap chain. Recall that the swap chain is an extension feature, so we must use a function with the `vk*KHR` naming convention:

```
void drawFrame() {
    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::max(), imageAva
}
```

The first two parameters of `vkAcquireNextImageKHR` are the logical device and the swap chain from which we wish to acquire an image. The third parameter specifies a timeout in nanoseconds for an image to become available. Using the maximum value of a 64 bit unsigned integer disables the timeout.

The next two parameters specify synchronization objects that are to be signaled when the presentation engine is finished using the image. That's the point in time where we can start drawing to it. It is possible to specify a semaphore, fence or both. We're going to use our `imageAvailableSemaphore` for that purpose here.

The last parameter specifies a variable to output the index of the swap chain image that has become available. The index refers to the `VkImage` in our `swapChainImages` array. We're going to use that index to pick the right command buffer.

Submitting the command buffer

Queue submission and synchronization is configured through parameters in the `VkSubmitInfo` structure.

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
```

The first three parameters specify which semaphores to wait on before execution begins and in which stage(s) of the pipeline to wait. We want to wait with writing colors to the image until it's available, so we're specifying the stage of the graphics pipeline that writes to the color attachment. That means that theoretically the implementation can already start executing our vertex shader and such while the image is not available yet. Each entry in the `waitStages` array corresponds to the semaphore with the same index in `pWaitSemaphores`.

```
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
```

The next two parameters specify which command buffers to actually submit for execution. As mentioned earlier, we should submit the command buffer that binds the swap chain image we just acquired as color attachment.

```
VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

The `signalSemaphoreCount` and `pSignalSemaphores` parameters specify which semaphores to signal once the command buffer(s) have finished execution. In our case we're using the `renderFinishedSemaphore` for that purpose.

```
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("failed to submit draw command buffer!");
}
```

We can now submit the command buffer to the graphics queue using `vkQueueSubmit`. The function takes an array of `VkSubmitInfo` structures as argument for efficiency when the workload is much larger. The last parameter references an optional fence that will be signaled when the command buffers finish execution. We're using semaphores for synchronization, so we'll just pass a `VK_NULL_HANDLE`.

Subpass dependencies

Remember that the subpasses in a render pass automatically take care of image layout transitions. These transitions are controlled by *subpass dependencies*, which specify memory and execution dependencies between subpasses. We have only a single subpass right now, but the operations right before and right after this subpass also count as implicit "subpasses".

There are two built-in dependencies that take care of the transition at the start of the render pass and at the end of the render pass, but the former does not occur at the right time. It assumes that the transition occurs at the start of the pipeline, but we haven't acquired the image yet at that point! There are two ways to deal with this problem. We could change the `waitStages` for the `imageAvailableSemaphore` to `VK_PIPELINE_STAGE_TOP_OF_PIPELINE_BIT` to ensure that the render passes don't begin until the image is available, or we can make the render pass wait for the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage. I've decided to go with the second option here, because it's a good excuse to have a look at subpass dependencies and how they work.

Subpass dependencies are specified in `VkSubpassDependency` structs. Go to the `createRenderPass` function and add one:

```
VkSubpassDependency dependency = {};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
```

The first two fields specify the indices of the dependency and the dependent subpass. The special value `VK_SUBPASS_EXTERNAL` refers to the implicit subpass before or after the render pass depending on whether it is specified in `srcSubpass` or `dstSubpass`. The index `0` refers to our subpass, which is the first and only one. The `dstSubpass` must always be higher than `srcSubpass` to prevent cycles in the dependency graph.

```
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
```

The next two fields specify the operations to wait on and the stages in which these operations occur. We need to wait for the swap chain to finish reading from the image before we can access it. This can be accomplished by waiting on the color attachment output stage itself.

```
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_
```

The operations that should wait on this are in the color attachment stage and involve the reading and writing of the color attachment. These settings will prevent the transition from happening until it's actually necessary (and allowed): when we want to start writing colors to it.

```
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

The `VkRenderPassCreateInfo` struct has two fields to specify an array of dependencies.

Presentation

The last step of drawing a frame is submitting the result back to the swap chain to have it eventually show up on the screen. Presentation is configured through a `VkPresentInfoKHR` structure at the end of the `drawFrame` function.

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = signalSemaphores;
```

The first two parameters specify which semaphores to wait on before presentation can happen, just like `VkSubmitInfo`.

```
VkSwapchainKHR swapChains[] = {swapChain};
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;
```

The next two parameters specify the swap chains to present images to and the index of the image for each swap chain. This will almost always be a single one.

```
presentInfo.pResults = nullptr; // Optional
```

There is one last optional parameter called `pResults`. It allows you to specify an array of `VkResult` values to check for every individual swap chain if presentation was successful. It's not necessary if you're only using a single swap chain, because you can simply use the return value of the present function.

```
vkQueuePresentKHR(presentQueue, &presentInfo);
```

The `vkQueuePresentKHR` function submits the request to present an image to the swap chain. We'll add error handling for both `vkAcquireNextImageKHR` and `vkQueuePresentKHR` in the next chapter, because their failure does not necessarily mean that the program should terminate, unlike the functions we've seen so far.

If you did everything correctly up to this point, then you should now see something resembling the following when you run your program:

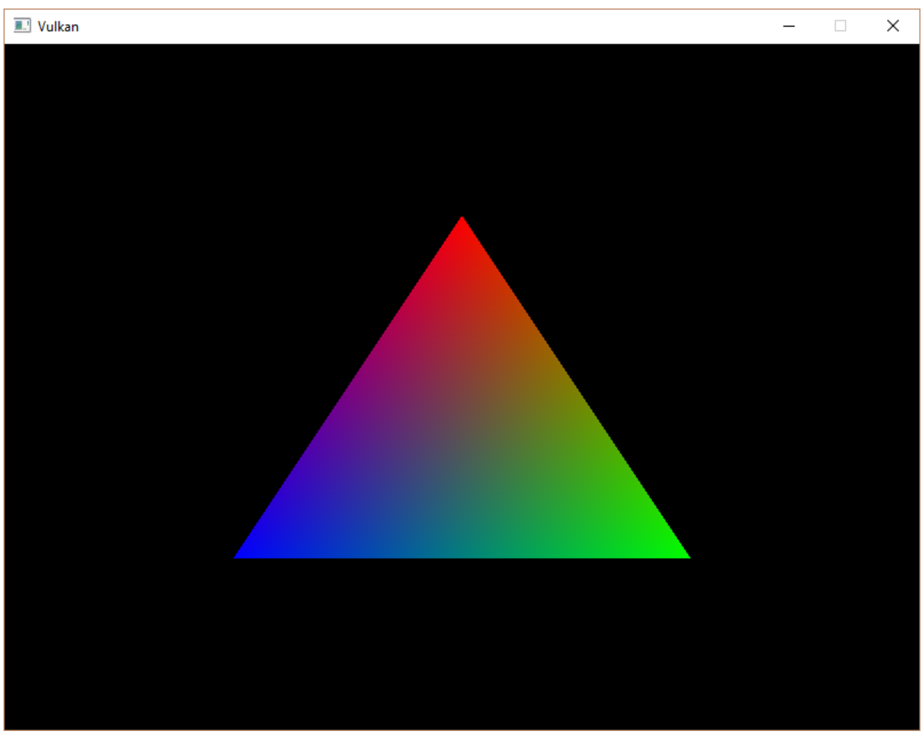

Figure 29:

Yay! Unfortunately, you'll see that when validation layers are enabled, the program crashes as soon as you close it. The message printed to the terminal from `debugCallback` tells us why:

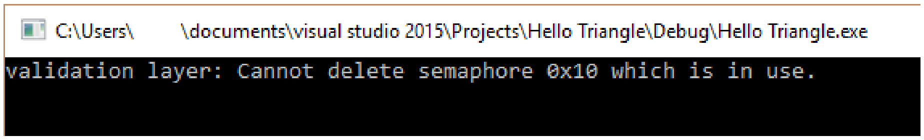

Figure 30:

Remember that all of the operations in `drawFrame` are asynchronous. That means that when we exit the loop in `mainLoop`, drawing and presentation operations may still be going on. Cleaning up resources while that is happening is a bad idea.

To fix that problem, we should wait for the logical device to finish operations before exiting `mainLoop` and destroying the window:

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

    vkDeviceWaitIdle(device);

    glfwDestroyWindow(window);
}
```

You can also wait for operations in a specific command queue to be finished with `vkQueueWaitIdle`. These functions can be used as a very rudimentary way to perform synchronization. You'll see that the program now exits without problems when closing the window.

Conclusion

About 800 lines of code later, we've finally gotten to the stage of seeing something pop up on the screen! Bootstrapping a Vulkan program is definitely a lot of work, but the take-away message is that Vulkan gives you an immense amount of control through its explicitness. I recommend you to take some time now to reread the code and build a mental model of the purpose of all of the Vulkan objects in the program and how they relate to each other. We'll be building on top of that knowledge to extend the functionality of the program from this point on.

In the next chapter we'll deal with one more small thing that is required for a well-behaved Vulkan program.

C++ code / Vertex shader / Fragment shader

Swap chain recreation

Introduction

The application we have now successfully draws a triangle, but there are some circumstances that it isn't handling properly yet. It is possible for the window surface to change such that the swap chain is no longer compatible with it. One of the reasons that could cause this to happen is the size of the window changing. We have to catch these events and recreate the swap chain.

Recreating the swap chain

Create a new `recreateSwapChain` function that calls `createSwapChain` and all of the creation functions for the objects that depend on the swap chain or the window size.

```
void recreateSwapChain() {
    vkDeviceWaitIdle(device);

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandBuffers();
}
```

We first call `vkDeviceWaitIdle`, because just like in the last chapter, we shouldn't touch resources that may still be in use. Obviously, the first thing we'll have to do is recreate the swap chain itself. The image views need to be recreated because they are based directly on the swap chain images. The render pass needs to be recreated because it depends on the format of the swap chain images. Viewport and scissor rectangle size is specified during graphics pipeline creation, so the pipeline also needs to be rebuilt. It is possible to avoid this by using dynamic state for the viewports and scissor rectangles. Finally, the framebuffers and command buffers also directly depend on the swap chain images.

Because of our handy `VDeleter` construct, most of the functions will work fine for recreation and will automatically clean up the old objects. However, the `createSwapChain` and `createCommandBuffers` functions still need some adjustments.

```
VkSwapchainKHR oldSwapChain = swapChain;
createInfo.oldSwapchain = oldSwapChain;

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

swapChain = newSwapChain;
```

We need to pass the previous swap chain object in the `oldSwapchain` parameter of `VkSwapchainCreateInfoKHR` to indicate that we intend to replace it. The old swap chain needs to stick around until after the new swap chain has been created, which means that we can't directly write the new handle to `swapChain`. The `VDeleter` would clear the old object before `vkCreateSwapchainKHR` has a chance to execute. That's why we use the temporary `newSwapChain` variable.

```
swapChain = newSwapChain;
```

This line will actually destroy the old swap chain and replace the handle with the handle of the new swap chain.

The problem with `createCommandBuffers` is that it doesn't free the old command buffers. There are two ways to solve this:

- Call `createCommandPool` as well, which will automatically free the old command buffers
- Extend `createCommandBuffers` to free any previous command buffers

As there isn't really a need to recreate the command pool itself, I've chosen to go for the second solution in this tutorial.

```
if (commandBuffers.size() > 0) {
    vkFreeCommandBuffers(device, commandPool, commandBuffers.size(), commandBuffers.data());
}

commandBuffers.resize(swapChainFramebuffers.size());
```

The `createCommandBuffers` function now first checks if the `commandBuffers` vector already contains previous command buffers, and if so, frees them. That's all it takes to recreate the swap chain!

Window resizing

Now we just need to figure out when swap chain recreation is necessary and call our new `recreateSwapChain` function. One of the most common conditions is resizing of the window. Let's make the window resizable and catch that event. Change the `initWindow` function to no longer include the `GLFW_RESIZABLE` line or change its argument from `GLFW_FALSE` to `GLFW_TRUE`.

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);

    glfwSetWindowUserPointer(window, this);
    glfwSetWindowSizeCallback(window, HelloTriangleApplication::onWindowResized);
}

...

static void onWindowResized(GLFWwindow* window, int width, int height) {
    if (width == 0 || height == 0) return;

    HelloTriangleApplication* app = reinterpret_cast<HelloTriangleApplication*>(glfwGetWindo
    app->recreateSwapChain();
}
```

The `glfwSetWindowSizeCallback` function can be used to specify a callback for the window resize event. Unfortunately it only accepts a function pointer as argument, so we can't directly use a member function. Luckily GLFW allows us to store an arbitrary pointer in the window object with `glfwSetWindowUserPointer`, so we can specify a static class member and get the original class instance back with `glfwGetWindowUserPointer`. We can then proceed to call `recreateSwapChain`, but only if the size of the window is non-zero. This case occurs when the window is minimized and it will cause swap chain creation to fail.

The `chooseSwapExtent` function should also be updated to take the current width and height of the window into account instead of the initial `WIDTH` and `HEIGHT`:

```
int width, height;
glfwGetWindowSize(window, &width, &height);

VkExtent2D actualExtent = {width, height};
```

Suboptimal or out-of-date swap chain

It is also possible for Vulkan to tell us that the swap chain is no longer compatible during presentation. The `vkAcquireNextImageKHR` and `vkQueuePresentKHR` functions can return the following special values to indicate this.

- `VK_ERROR_OUT_OF_DATE_KHR`: The swap chain has become incompatible with the surface and can no longer be used for rendering.

- `VK_SUBOPTIMAL_KHR`: The swap chain can still be used to successfully present to the surface, but the surface properties are no longer matched exactly. For example, the platform may be simply resizing the image to fit the window now.

```
VkResult result = vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::ma

if (result == VK_ERROR_OUT_OF_DATE_KHR) {
    recreateSwapChain();
    return;
} else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
    throw std::runtime_error("failed to acquire swap chain image!");
}
```

If the swap chain turns out to be out of date when attempting to acquire an image, then it is no longer possible to present to it. Therefore we should immediately recreate the swap chain and try again in the next `drawFrame` call.

You could also decide to do that if the swap chain is suboptimal, but I've chosen to proceed anyway in that case because we've already acquired an image. Both `VK_SUCCESS` and `VK_SUBOPTIMAL_KHR` are considered "success" return codes.

```
result = vkQueuePresentKHR(presentQueue, &presentInfo);

if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR) {
    recreateSwapChain();
} else if (result != VK_SUCCESS) {
    throw std::runtime_error("failed to present swap chain image!");
}
```

The `vkQueuePresentKHR` function returns the same values with the same meaning. In this case we will also recreate the swap chain if it is suboptimal, because we want the best possible result. Try to run it and resize the window to see if the framebuffer is indeed resized properly with the window.

Congratulations, you've now finished your very first well-behaved Vulkan program! In the next chapter we're going to get rid of the hardcoded vertices in the vertex shader and actually use a vertex buffer.

C++ code / Vertex shader / Fragment shader # Vertex input description

Introduction

In the next few chapters, we're going to replace the hardcoded vertex data in the vertex shader with a vertex buffer in memory. We'll start with the easiest approach of creating a CPU visible buffer and using `memcpy` to copy the vertex data into it directly, and after that we'll see how to use a staging buffer to copy the vertex data to high performance memory.

Vertex shader

First change the vertex shader to no longer include the vertex data in the shader code itself. The vertex shader takes input from a vertex buffer using the `in` keyword.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

The `inPosition` and `inColor` variables are *vertex attributes*. They're properties that are specified per-vertex in the vertex buffer, just like we manually specified a position and color per vertex using the two arrays. Make sure to recompile the vertex shader!

Vertex data

We're moving the vertex data from the shader code to an array in the code of our program. Start by including the GLM library, which provides us with linear algebra related types like vectors and matrices. We're going to use these types to specify the position and color vectors.

```
#include <glm/glm.hpp>
```

Create a new structure called `Vertex` with the two attributes that we're going to use in the vertex shader inside it:

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
};
```

GLM conveniently provides us with C++ types that exactly match the vector types used in the shader language.

```
const std::vector<Vertex> vertices = {
    {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

Now use the `Vertex` structure to specify an array of vertex data. We're using exactly the same position and color values as before, but now they're combined into one array of vertices. This is known as *interleaving* vertex attributes.

Binding descriptions

The next step is to tell Vulkan how to pass this data format to the vertex shader once it's been uploaded into GPU memory. There are two types of structures needed to convey this information.

The first structure is `VkVertexInputBindingDescription` and we'll add a member function to the `Vertex` struct to populate it with the right data.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription = {};

        return bindingDescription;
    }
};
```

A vertex binding describes at which rate to load data from memory throughout the vertices. It specifies the number of bytes between data entries and whether to move to the next data entry after each vertex or after each instance.

```
VkVertexInputBindingDescription bindingDescription = {};
bindingDescription.binding = 0;
bindingDescription.stride = sizeof(Vertex);
bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

All of our per-vertex data is packed together in one array, so we're only going to have one binding. The `binding` parameter specifies the index of the binding in the array of bindings. The `stride` parameter specifies the number of bytes from one entry to the next, and the `inputRate` parameter can have one of the following values:

- `VK_VERTEX_INPUT_RATE_VERTEX`: Move to the next data entry after each vertex

- `VK_VERTEX_INPUT_RATE_INSTANCE`: Move to the next data entry after each instance

We're not going to use instanced rendering, so we'll stick to per-vertex data.

Attribute descriptions

The second structure that describes how to handle vertex input is `VkVertexInputAttributeDescription`. We're going to add another helper function to `Vertex` to fill in these structs.

```
#include <array>

...

static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {
    std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions = {};

    return attributeDescriptions;
}
```

As the function prototype indicates, there are going to be two of these structures. An attribute description struct describes how to extract a vertex attribute from a chunk of vertex data originating from a binding description. We have two attributes, position and color, so we need two attribute description structs.

```
attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);
```

The `binding` parameter tells Vulkan from which binding the per-vertex data comes. The `location` parameter references the `location` directive of the input in the vertex shader. The input in the vertex shader with location `0` is the position, which has two 32-bit float components.

The `format` parameter describes the type of data for the attribute. A bit confusingly, the formats are specified using the same enumeration as color formats. The following shader types and formats are commonly used together:

- `float`: `VK_FORMAT_R32_SFLOAT`
- `vec2`: `VK_FORMAT_R32G32_SFLOAT`
- `vec3`: `VK_FORMAT_R32G32B32_SFLOAT`
- `vec4`: `VK_FORMAT_R32G32B32A32_SFLOAT`

As you can see, you should use the format where the amount of color channels matches the number of components in the shader data type. It is allowed to use more channels than the number of components in the shader, but they will be silently discarded. If the number of channels is lower than the number of components, then the BGA components will use default values of `(0, 0, 1)`. The color type (`SFLOAT`, `UINT`, `SINT`) and bit width should also match the type of the shader input. See the following examples:

- `ivec2`: `VK_FORMAT_R32G32_SINT`, a 2-component vector of 32-bit signed integers
- `uvec4`: `VK_FORMAT_R32G32B32A32_UINT`, a 4-component vector of 32-bit unsigned integers
- `double`: `VK_FORMAT_R64_SFLOAT`, a double-precision (64-bit) float

The `format` parameter implicitly defines the byte size of attribute data and the `offset` parameter specifies the number of bytes since the start of the per-vertex data to read from. The binding is loading one `Vertex` at a time and the position attribute (`pos`) is at an offset of `0` bytes from the beginning of this struct. This is automatically calculated using the `offsetof` macro.

```
attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);
```

The color attribute is described in much the same way.

Pipeline vertex input

We now need to set up the graphics pipeline to accept vertex data in this format by referencing the structures in `createGraphicsPipeline`. Find the `vertexInputInfo` struct and modify it to reference the two descriptions:

```
auto bindingDescription = Vertex::getBindingDescription();
auto attributeDescriptions = Vertex::getAttributeDescriptions();

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = attributeDescriptions.size();
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();
```

The pipeline is now ready to accept vertex data in the format of the `vertices` container and pass it on to our vertex shader. If you run the program now with validation layers enabled, you'll see that it complains that there is no vertex buffer bound to the binding. The next step is to create a vertex buffer and move the vertex data to it so the GPU is able to access it.

C++ code / Vertex shader / Fragment shader # Vertex buffer creation

Introduction

Buffers in Vulkan are regions of memory used for storing arbitrary data that can be read by the graphics card. They can be used to store vertex data, which we'll do in this chapter, but they can also be used for many other purposes that we'll explore in future chapters. Unlike the Vulkan objects we've been dealing with so far, buffers do not automatically allocate memory for themselves. The work from the previous chapters has shown that the Vulkan API puts the programmer in control of almost everything and memory management is one of those things.

Buffer creation

Create a new function `createVertexBuffer` and call it from `initVulkan` right before `createCommandBuffers`.

```
void initVulkan() {
    createInstance();
    setupDebugCallback();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createVertexBuffer();
    createCommandBuffers();
    createSemaphores();
}

...

void createVertexBuffer() {

}
```

Creating a buffer requires us to fill a `VkBufferCreateInfo` structure.

```
VkBufferCreateInfo bufferInfo = {};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = sizeof(vertices[0]) * vertices.size();
```

The first field of the struct is `size`, which specifies the size of the buffer in bytes. Calculating the byte size of the vertex data is straightforward with `sizeof`.

```
bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
```

The second field is `usage`, which indicates for which purposes the data in the buffer is going to be used. It is possible to specify multiple purposes using a bitwise or. Our use case will be a vertex buffer, we'll look at other types of usage in future chapters.

```
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

Just like the images in the swap chain, buffers can also be owned by a specific queue family or be shared between multiple at the same time. The buffer will only be used from the graphics queue, so we can stick to exclusive access.

The `flags` parameter is used to configure sparse buffer memory, which is not relevant right now. We'll leave it at the default value of `0`.

We can now create the buffer with `vkCreateBuffer`. Define a class member to hold the buffer handle and call it `vertexBuffer`.

```
VDeleter<VkBuffer> vertexBuffer{device, vkDestroyBuffer};

...

void createVertexBuffer() {
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = sizeof(vertices[0]) * vertices.size();
    bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, vertexBuffer.replace()) != VK_SUCCESS)
        throw std::runtime_error("failed to create vertex buffer!");
    }
}
```

Memory requirements

The buffer has been created, but it doesn't actually have any memory assigned to it yet. The first step of allocating memory for the buffer is to query its memory requirements using the aptly named `vkGetBufferMemoryRequirements` function.

```
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
```

The `VkMemoryRequirements` struct has three fields:

- `size`: The size of the required amount of memory in bytes, may differ from `bufferInfo.size`.

- `alignment`: The offset in bytes where the buffer begins in the allocated region of memory, depends on `bufferInfo.usage` and `bufferInfo.flags`.
- `memoryTypeBits`: Bit field of the memory types that are suitable for the buffer.

Graphics cards can offer different types of memory to allocate from. Each type of memory varies in terms of allowed operations and performance characteristics. We need to combine the requirements of the buffer and our own application requirements to find the right type of memory to use. Let's create a new function `findMemoryType` for this purpose.

```
uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {

}
```

First we need to query info about the available types of memory using `vkGetPhysicalDeviceMemoryProperties`.

```
VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
```

The `VkPhysicalDeviceMemoryProperties` structure has two arrays `memoryTypes` and `memoryHeaps`. Memory heaps are distinct memory resources like dedicated VRAM and swap space in RAM for when VRAM runs out. The different types of memory exist within these heaps. Right now we'll only concern ourselves with the type of memory and not the heap it comes from, but you can imagine that this can affect performance.

Let's first find a memory type that is suitable for the buffer itself:

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if (typeFilter & (1 << i)) {
        return i;
    }
}

throw std::runtime_error("failed to find suitable memory type!");
```

The `typeFilter` parameter will be used to specify the bit field of memory types that are suitable. That means that we can find the index of a suitable memory type by simply iterating over them and checking if the corresponding bit is set to `1`.

However, we're not just interested in a memory type that is suitable for the vertex buffer. We also need to be able to write our vertex data to that memory. The `memoryTypes` array consists of `VkMemoryType` structs that specify the heap and properties of each type of memory. The properties define special features of the memory, like being able to map it so we can write to it from the CPU. This property is indicated with `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`, but we also need to use the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` property. We'll see why when we map the memory.

We can now modify the loop to also check for the support of this property:

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].propertyFlags & properties)
        return i;
    }
}
```

In the future we may have more than one desirable property, so we should check if the result of the bitwise AND is not just non-zero, but equal to the desired properties bit field. If there is a memory type suitable for the buffer that also has all of the properties we need, then we return its index, otherwise we throw an exception.

Memory allocation

We now have a way to determine the right memory type, so we can actually allocate the memory by filling in the `VkMemoryAllocateInfo` structure.

```
VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, VK_MEMORY_PROPERT
```

Memory allocation is now as simple as specifying the size and type, both of which are derived from the memory requirements of the vertex buffer and the desired property. Create a class member to store the handle to the memory and allocate it with `vkAllocateMemory`.

```
VDeleter<VkBuffer> vertexBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> vertexBufferMemory{device, vkFreeMemory};

...

if (vkAllocateMemory(device, &allocInfo, nullptr, vertexBufferMemory.replace()) != VK_SUCCES
    throw std::runtime_error("failed to allocate vertex buffer memory!");
}
```

Note that specifying the `vertexBuffer` and `vertexBufferMemory` members in this order will cause the memory to be freed before the buffer is destroyed, but that's allowed as long as the buffer is no longer used.

If memory allocation was successful, then we can now associate this memory with the buffer using `vkBindBufferMemory`:

```
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
```

The first two parameters are self-explanatory and the third parameter is the offset within the region of memory. Since this memory is allocated specifically for this the vertex buffer, the offset is simply `0`. If the offset is non-zero, then it is required to be divisible by `memRequirements.alignment`.

Filling the vertex buffer

It is now time to copy the vertex data to the buffer. This is done by mapping the buffer memory into CPU accessible memory with `vkMapMemory`.

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
```

This function allows us to access a region of the specified memory resource defined by an offset and size. The offset and size here are `0` and `bufferInfo.size`, respectively. It is also possible to specify the special value `VK_WHOLE_SIZE` to map all of the memory. The second to last parameter can be used to specify flags, but there aren't any available yet in the current API. It must be set to the value `0`. The last parameter specifies the output for the pointer to the mapped memory.

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
    memcpy(data, vertices.data(), (size_t) bufferInfo.size);
vkUnmapMemory(device, vertexBufferMemory);
```

You can now simply `memcpy` the vertex data to the mapped memory and unmap it again using `vkUnmapMemory`. Unfortunately the driver may not immediately copy the data into the buffer memory, for example because of caching. It is also possible that writes to the buffer are not visible in the mapped memory yet. There are two ways to deal with that problem:

- Use a memory heap that is host coherent, indicated with `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- Call `vkFlushMappedMemoryRanges` to after writing to the mapped memory, and call `vkInvalidateMappedMemoryRanges` before reading from the mapped memory

We went for the first approach, which ensures that the mapped memory always matches the contents of the allocated memory. Do keep in mind that this may lead to slightly worse performance than explicit flushing, but we'll see why that doesn't matter in the next chapter.

Binding the vertex buffer

All that remains now is binding the vertex buffer during rendering operations. We're going to extend the `createCommandBuffers` function to do that.

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdDraw(commandBuffers[i], vertices.size(), 1, 0, 0);
```

The `vkCmdBindVertexBuffers` function is used to bind vertex buffers to bindings, like the one we set up in the previous chapter. The first two parameters, besides the command buffer, specify the offset and number of bindings we're going to specify vertex buffers for. The last two parameters specify the array of vertex buffers to bind and the byte offsets to start reading vertex data from. You should also change the call to `vkCmdDraw` to pass the number of vertices in the buffer as opposed to the hardcoded number `3`.

Now run the program and you should see the familiar triangle again:

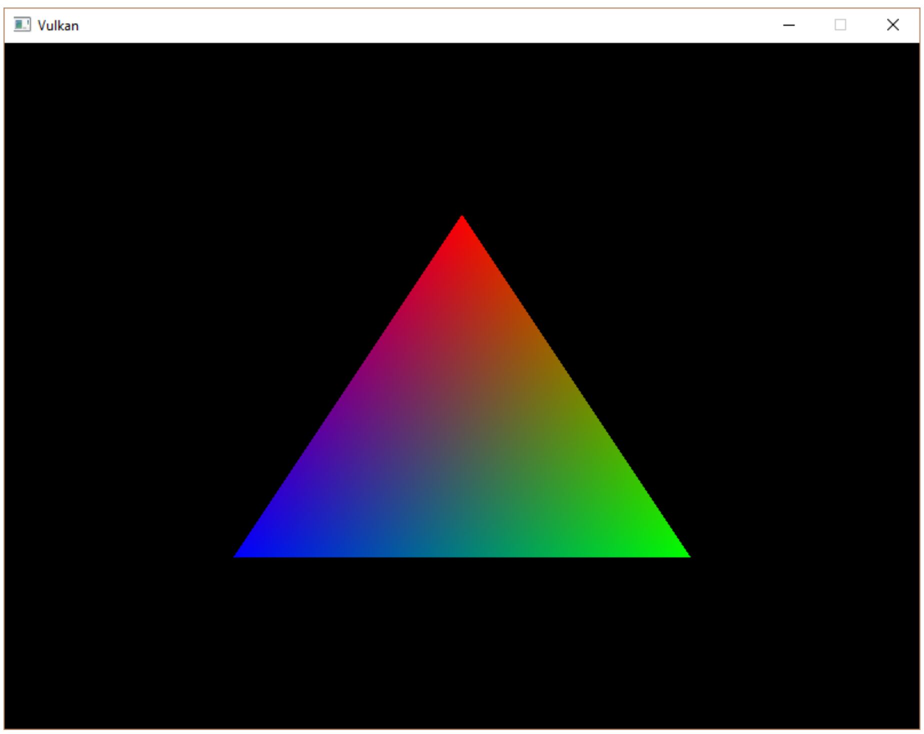

Figure 31:

Try changing the color of the top vertex to white by modifying the `vertices` array:

```
const std::vector<Vertex> vertices = {
```

```
    {{0.0f, -0.5f}, {1.0f, 1.0f, 1.0f}},
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

Run the program again and you should see the following:

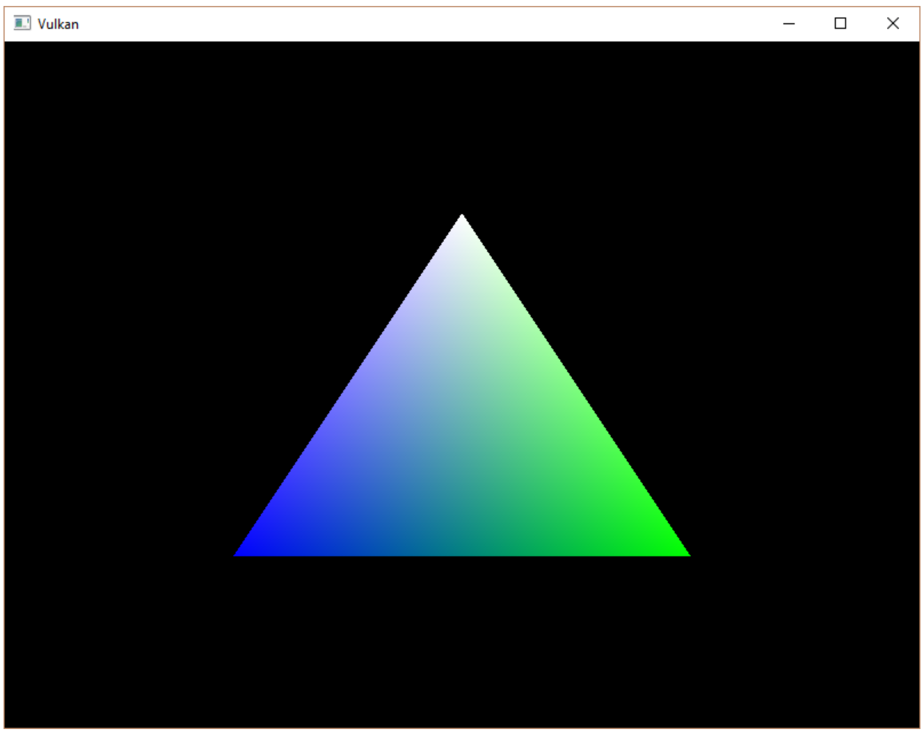

Figure 32:

In the next chapter we'll look at a different way to copy vertex data to a vertex buffer that results in better performance, but takes some more work.

C++ code / Vertex shader / Fragment shader # Staging buffer

Introduction

The vertex buffer we have right now works correctly, but the memory type that allows us to access it from the CPU may not be the most optimal memory type for the graphics card itself to read from. The most optimal memory has the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` flag and is usually not accessible by the CPU on dedicated graphics cards. In this chapter we're going to create two vertex buffers. One *staging buffer* in CPU accessible memory to upload the data from the vertex array to, and the final vertex buffer in device local memory. We'll then use a buffer copy command to move the data from the staging buffer to the actual vertex buffer.

Transfer queue

The buffer copy command requires a queue family that supports transfer operations, which is indicated using `VK_QUEUE_TRANSFER_BIT`. The good news is that any queue family with `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT` capabilities already implicitly support `VK_QUEUE_TRANSFER_BIT` operations. The implementation is not required to explicitly list it in `queueFlags` in those cases.

If you like a challenge, then you can still try to use a different queue family specifically for transfer operations. It will require you to make the following modifications to your program:

- Modify `QueueFamilyIndices` and `findQueueFamilies` to explicitly look for a queue family with the `VK_QUEUE_TRANSFER` bit, but not the `VK_QUEUE_GRAPHICS_BIT`.
- Modify `createLogicalDevice` to request a handle to the transfer queue
- Create a second command pool for command buffers that are submitted on the transfer queue family
- Change the `sharingMode` of resources to be `VK_SHARING_MODE_CONCURRENT` and specify both the graphics and transfer queue families
- Submit any transfer commands like `vkCmdCopyBuffer` (which we'll be using in this chapter) to the transfer queue instead of the graphics queue

It's a bit of work, but it'll teach you a lot about how resources are shared between queue families.

Abstracting buffer creation

Because we're going to create multiple buffers in this chapter, it's a good idea to move buffer creation to a helper function. Create a new function `createBuffer` and move the code in `createVertexBuffer` (except mapping) to it.

```
void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags propert
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, buffer.replace()) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }
```

```
    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, bufferMemory.replace()) != VK_SUCCESS
        throw std::runtime_error("failed to allocate buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Make sure to add parameters for the buffer size, memory properties and usage so that we can use this function to create many different types of buffers. The last two parameters are output variables to write the handles to.

You can now remove the buffer creation and memory allocation code from `createVertexBuffer` and just call `createBuffer` instead:

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
    createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_HOST_VIS

    void* data;
    vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
        memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, vertexBufferMemory);
}
```

Run your program to make sure that the vertex buffer still works properly.

Using a staging buffer

We're now going to change `createVertexBuffer` to only use a host visible buffer as temporary buffer and use a device local one as actual vertex buffer.

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();

    VDeleter<VkBuffer> stagingBuffer{device, vkDestroyBuffer};
    VDeleter<VkDeviceMemory> stagingBufferMemory{device, vkFreeMemory};
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISI
```

```
    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
        memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFE
}
```

We're now using a new `stagingBuffer` with `stagingBufferMemory` for mapping and copying the vertex data. In this chapter we're going to use two new buffer usage flags:

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT`: Buffer can be used as source in a memory transfer operation.
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT`: Buffer can be used as destination in a memory transfer operation.

The `vertexBuffer` is now allocated from a memory type that is device local, which generally means that we're not able to use `vkMapMemory`. However, we can copy data from the `stagingBuffer` to the `vertexBuffer`. We have to indicate that we intend to do that by specifying the transfer source flag for the `stagingBuffer` and the transfer destination flag for the `vertexBuffer`, along with the vertex buffer usage flag.

We're now going to write a function to copy the contents from one buffer to another, called `copyBuffer`.

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {

}
```

Memory transfer operations are executed using command buffers, just like drawing commands. Therefore we must first allocate a temporary command buffer. You may wish to create a separate command pool for these kinds of short-lived buffers, because the implementation may be able to apply memory allocation optimizations. You should use the `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` flag during command pool generation in that case.

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
}
```

And immediately start recording the command buffer:

```
VkCommandBufferBeginInfo beginInfo = {};
beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

vkBeginCommandBuffer(commandBuffer, &beginInfo);
```

The `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag that we used for the drawing command buffers is not necessary here, because we're only going to use the command buffer once and wait with returning from the function until the copy operation has finished executing. It's good practice to tell the driver about our intent using `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`.

```
VkBufferCopy copyRegion = {};
copyRegion.srcOffset = 0; // Optional
copyRegion.dstOffset = 0; // Optional
copyRegion.size = size;
vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);
```

Contents of buffers are transferred using the `vkCmdCopyBuffer` command. It takes the source and destination buffers as arguments, and an array of regions to copy. The regions are defined in `VkBufferCopy` structs and consist of a source buffer offset, destination buffer offset and size. It is not possible to specify `VK_WHOLE_SIZE` here, unlike the `vkMapMemory` command.

```
vkEndCommandBuffer(commandBuffer);
```

This command buffer only contains the copy command, so we can stop recording right after that. Now execute the command buffer to complete the transfer:

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
vkQueueWaitIdle(graphicsQueue);
```

Unlike the draw commands, there are no events we need to wait on this time. We just want to execute the transfer on the buffers immediately. There are again two possible ways to wait on this transfer to complete. We could use a fence and wait with `vkWaitForFences`, or simply wait for the transfer queue to become idle with `vkQueueWaitIdle`. A fence would allow you to schedule multiple transfers simultaneously and wait for all of them complete, instead of executing one at a time. That may give the driver more opportunities to optimize.

```
vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
```

Don't forget to clean up the command buffer used for the transfer operation.

We can now call `copyBuffer` from the `createVertexBuffer` function to move the vertex data to the device local buffer:

```
createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BI

copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
```

Run your program to verify that you're seeing the familiar triangle again. It may not be visible, but its vertex data is now being loaded from high performance memory. This will matter when we're going to start rendering more complex geometry.

Conclusion

It should be noted that in a real world application, you're not supposed to actually call `vkAllocateMemory` for every individual buffer. The maximum number of simultaneous memory allocations is limited by the `maxMemoryAllocationCount` physical device limit, which may be as low as `4096` even on high end hardware like an NVIDIA GTX 1080. The right way to allocate memory for a large number of objects at the same time is to create a custom allocator that splits up a single allocation among many different objects by using the `offset` parameters that we've seen in many functions.

You will currently have to write such an allocator yourself, but the author expects that there will be a library at some point that can be integrated into any Vulkan program to properly handle allocations. It's okay to use a separate allocation for every resource for this tutorial, because we won't come close to hitting any of these limits for now.

C++ code / Vertex shader / Fragment shader

Index buffer

Introduction

The 3D meshes you'll be rendering in a real world application will often share vertices between multiple triangles. This already happens even with something simple like drawing a rectangle:

Drawing a rectangle takes two triangles, which means that we need a vertex buffer with 6 vertices. The problem is that the data of two vertices needs to be duplicated resulting in 50% redundancy. It only gets worse with more complex meshes, where vertices are reused in an average number of 3 triangles. The solution to this problem is to use an *index buffer*.

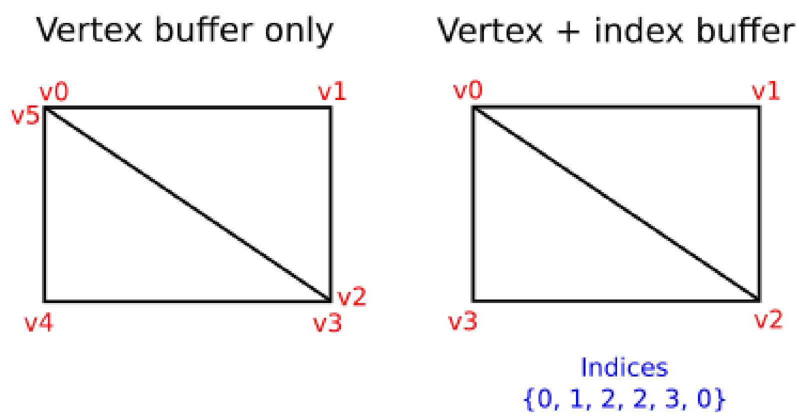

Figure 33:

An index buffer is essentially an array of pointers into the vertex buffer. It allows you to reorder the vertex data, and reuse existing data for multiple vertices. The illustration above demonstrates what the index buffer would look like for the rectangle if we have a vertex buffer containing each of the four unique vertices. The first three indices define the upper-right triangle and the last three indices define the vertices for the bottom-left triangle.

Index buffer creation

In this chapter we're going to modify the vertex data and add index data to draw a rectangle like the one in the illustration. Modify the vertex data to represent the four corners:

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}
};
```

The top-left corner is red, top-right is green, bottom-right is blue and the bottom-left is white. We'll add a new array `indices` to represent the contents of the index buffer. It should match the indices in the illustration to draw the upper-right triangle and bottom-left triangle.

```
const std::vector<uint16_t> indices = {
```

```
    0, 1, 2, 2, 3, 0
};
```

It is possible to use either `uint16_t` or `uint32_t` for your index buffer depending on the number of entries in `vertices`. We can stick to `uint16_t` for now because we're using less than 65535 unique vertices.

Just like the vertex data, the indices need to be uploaded into a `VkBuffer` for the GPU to be able to access them. Define two new class members to hold the resources for the index buffer:

```
VDeleter<VkBuffer> vertexBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> vertexBufferMemory{device, vkFreeMemory};
VDeleter<VkBuffer> indexBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> indexBufferMemory{device, vkFreeMemory};
```

The `createIndexBuffer` function that we'll add now is almost identical to `createVertexBuffer`:

```
void initVulkan() {
    ...
    createVertexBuffer();
    createIndexBuffer();
    ...
}

void createIndexBuffer() {
    VkDeviceSize bufferSize = sizeof(indices[0]) * indices.size();

    VDeleter<VkBuffer> stagingBuffer{device, vkDestroyBuffer};
    VDeleter<VkDeviceMemory> stagingBufferMemory{device, vkFreeMemory};
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIB

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, indices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER

    copyBuffer(stagingBuffer, indexBuffer, bufferSize);
}
```

There are only two notable differences. The `bufferSize` is now equal to the number of indices times the size of the index type, either `uint16_t` or `uint32_t`. The usage of the `indexBuffer` should be `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` instead of `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`, which makes sense. Other than that, the process is exactly the same. We create a staging buffer to copy the contents of `indices` to and then copy it to the final device local index buffer.

Using an index buffer

Using an index buffer for drawing involves two changes to `createCommandBuffers`. We first need to bind the index buffer, just like we did for the vertex buffer. The difference is that you can only have a single index buffer. It's unfortunately not possible to use different indices for each vertex attribute, so we do still have to completely duplicate vertex data even if just one attribute varies.

```
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT16);
```

An index buffer is bound with `vkCmdBindIndexBuffer` which has the index buffer, a byte offset into it, and the type of index data as parameters. As mentioned before, the possible types are `VK_INDEX_TYPE_UINT16` and `VK_INDEX_TYPE_UINT32`.

Just binding an index buffer doesn't change anything yet, we also need to change the drawing command to tell Vulkan to use the index buffer. Remove the `vkCmdDraw` line and replace it with `vkCmdDrawIndexed`:

```
vkCmdDrawIndexed(commandBuffers[i], indices.size(), 1, 0, 0, 0);
```

A call to this function is very similar to `vkCmdDraw`. The first two parameters specify the number of indices and the number of instances. We're not using instancing, so just specify `1` instance. The number of indices represents the number of vertices that will be passed to the vertex buffer. The next parameter specifies an offset into the index buffer, using a value of `1` would cause the graphics card to start reading at the second index. The second to last parameter specifies an offset to add to the indices in the index buffer. The final parameter specifies an offset for instancing, which we're not using.

Now run your program and you should see the following:

You now know how to save memory by reusing vertices with index buffers. This will become especially important in a future chapter where we're going to load complex 3D models.

The previous chapter already mentioned that you should allocate multiple resources like buffers from a single memory allocation, but in fact you should go a step further. Driver developers recommend that you also store multiple buffers, like the vertex and index buffer, into a single `VkBuffer` and use offsets in commands like `vkCmdBindVertexBuffers`. The advantage is that your data is more cache friendly in that case, because it's closer together. It is even possible to reuse the same chunk of memory for multiple resources if they are not used during the same render operations, provided that their data is refreshed, of course. This is known as *aliasing* and some Vulkan functions have explicit flags to specify that you want to do this.

C++ code / Vertex shader / Fragment shader

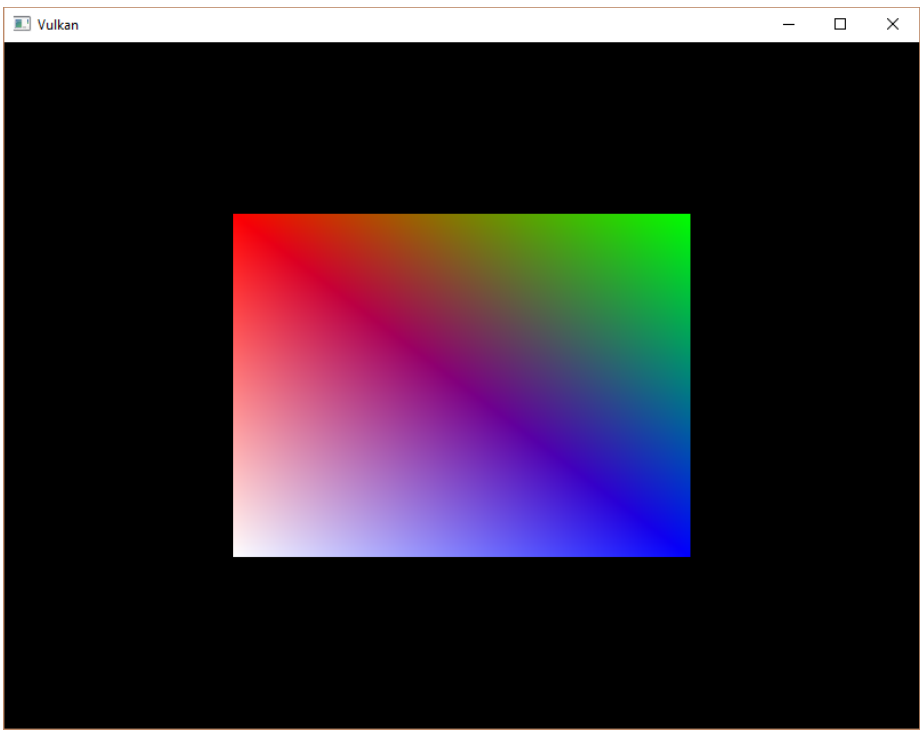

Figure 34:

Descriptor layout and buffer

Introduction

We're now able to pass arbitrary attributes to the vertex shader for each vertex, but what about global variables? We're going to move on to 3D graphics from this chapter on and that requires a model-view-projection matrix. We could include it as vertex data, but that's a waste of memory and it would require us to update the vertex buffer whenever the transformation changes. The transformation could easily change every single frame.

The right way to tackle this in Vulkan is to use *resource descriptors*. A descriptor is a way for shaders to freely access resources like buffers and images. We're going to set up a buffer that contains the transformation matrices and have the vertex shader access them through a descriptor. Usage of descriptors consists of three parts:

- Specify a descriptor layout during pipeline creation
- Allocate a descriptor set from a descriptor pool
- Bind the descriptor set during rendering

The *descriptor layout* specifies the types of resources that are going to be accessed by the pipeline, just like a render pass specifies the types of attachments that will be accessed. A *descriptor set* specifies the actual buffer or image resources that will be bound to the descriptors, just like a framebuffer specifies the actual image views to bind to render pass attachments. The descriptor set is then bound for the drawing commands just like the vertex buffers and framebuffer.

There are many types of descriptors, but in this chapter we'll work with uniform buffer objects (UBO). We'll look at other types of descriptors in future chapters, but the basic process is the same. Let's say we have the data we want the vertex shader to have in a C struct like this:

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Then we can copy the data to a `VkBuffer` and access it through a uniform buffer object descriptor from the vertex shader like this:

```
layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
```

```
void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

We're going to update the model, view and projection matrices every frame to make the rectangle from the previous chapter spin around in 3D.

Vertex shader

Modify the vertex shader to include the uniform buffer object like it was specified above. I will assume that you are familiar with MVP transformations. If you're not, see the resource mentioned in the first chapter.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Note that the order of the `uniform`, `in` and `out` declarations doesn't matter. The `binding` directive is similar to the `location` directive for attributes. We're going to reference this binding in the descriptor layout. The line with `gl_Position` is changed to use the transformations to compute the final position in clip coordinates.

Descriptor set layout

The next step is to define the UBO on the C++ side and to tell Vulkan about this descriptor in the vertex shader.

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

We can exactly match the definition in the shader using data types in GLM. The data in the matrices is binary compatible with the way the shader expects it, so we can later just `memcpy` a `UniformBufferObject` to a `VkBuffer`.

We need to provide details about every descriptor binding used in the shaders for pipeline creation, just like we had to do for every vertex attribute and its `location` index. We'll set up a new function to define all of this information called `createDescriptorSetLayout`. It should be called right before pipeline creation, because we're going to need it there.

```
void initVulkan() {
    ...
    createDescriptorSetLayout();
    createGraphicsPipeline();
    ...
}

...

void createDescriptorSetLayout() {

}
```

Every binding needs to be described through a `VkDescriptorSetLayoutBinding` struct.

```
void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding = {};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.descriptorCount = 1;
}
```

The first two fields specify the `binding` used in the shader and the type of descriptor, which is a uniform buffer object. It is possible for the shader variable to represent an array of uniform buffer objects, and `descriptorCount` specifies the number of values in the array. This could be used to specify a transformation for each of the bones in a skeleton for skeletal animation, for example. Our

MVP transformation is in a single uniform buffer object, so we're using a `descriptorCount` of `1`.

```
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

We also need to specify in which shader stages the descriptor is going to be referenced. The `stageFlags` field can be a combination of `VkShaderStage` flags or the value `VK_SHADER_STAGE_ALL_GRAPHICS`. In our case, we're only referencing the descriptor from the vertex shader.

```
uboLayoutBinding.pImmutableSamplers = nullptr; // Optional
```

The `pImmutableSamplers` field is only relevant for image sampling related descriptors, which we'll look at later. You can leave this to its default value.

All of the descriptor bindings are combined into a single `VkDescriptorSetLayout` object. Define a new class member above `pipelineLayout`:

```
VDeleter<VkDescriptorSetLayout> descriptorSetLayout{device, vkDestroyDescriptorSetLayout};
VDeleter<VkPipelineLayout> pipelineLayout{device, vkDestroyPipelineLayout};
```

We can then create it using `vkCreateDescriptorSetLayout`. This function accepts a simple `VkDescriptorSetLayoutCreateInfo` with the array of bindings:

```
VkDescriptorSetLayoutCreateInfo layoutInfo = {};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = 1;
layoutInfo.pBindings = &uboLayoutBinding;

if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, descriptorSetLayout.replace())
    throw std::runtime_error("failed to create descriptor set layout!");
}
```

We need to specify the descriptor set layout during pipeline creation to tell Vulkan which descriptors the shaders will be using. Descriptor set layouts are specified in the pipeline layout object. Modify the `VkPipelineLayoutCreateInfo` to reference the layout object:

```
VkDescriptorSetLayout setLayouts[] = {descriptorSetLayout};
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = setLayouts;
```

You may be wondering why it's possible to specify multiple descriptor set layouts here, because a single one already includes all of the bindings. We'll get back to that in the next chapter, where we'll look into descriptor pools and descriptor sets.

Uniform buffer

In the next chapter we'll specify the buffer that contains the UBO data for the shader, but we need to create this buffer first. We're going to copy new data to the uniform buffer every frame, so this time the staging buffer actually needs to stick around.

Add new class members for `uniformStagingBuffer`, `uniformStagingBufferMemory`, `uniformBuffer`, and `uniformBufferMemory`:

```
VDeleter<VkBuffer> indexBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> indexBufferMemory{device, vkFreeMemory};

VDeleter<VkBuffer> uniformStagingBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> uniformStagingBufferMemory{device, vkFreeMemory};
VDeleter<VkBuffer> uniformBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> uniformBufferMemory{device, vkFreeMemory};
```

Similarly, create a new function `createUniformBuffer` that is called after `createIndexBuffer` and allocates the buffers:

```
void initVulkan() {
    ...
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffer();
    ...
}

...

void createUniformBuffer() {
    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIB
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_UNIFORM_BUFF
}
```

We're going to write a separate function that updates the uniform buffer with a new transformation every frame, so there will be no `vkMapMemory` and `copyBuffer` operations here.

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();

        updateUniformBuffer();
        drawFrame();
```

```
    }

    vkDeviceWaitIdle(device);

    glfwDestroyWindow(window);
}

...

void updateUniformBuffer() {

}
```

Updating uniform data

Create a new function `updateUniformBuffer` and add a call to it from the main loop. This function will generate a new transformation every frame to make the geometry spin around. We need to include two new headers to implement this functionality:

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <chrono>
```

The `glm/gtc/matrix_transform.hpp` header exposes functions that can be used to generate model transformations like `glm::rotate`, view transformations like `glm::lookAt` and projection transformations like `glm::perspective`. The `GLM_FORCE_RADIANS` definition is necessary to make sure that functions like `glm::rotate` use radians as arguments, to avoid any possible confusion.

The `chrono` standard library header exposes functions to do precise timekeeping. We'll use this to make sure that the geometry rotates 90 degrees per second regardless of frame rate.

```
void updateUniformBuffer() {
    static auto startTime = std::chrono::high_resolution_clock::now();

    auto currentTime = std::chrono::high_resolution_clock::now();
    float time = std::chrono::duration_cast<std::chrono::milliseconds>(currentTime - startTi
}
```

The `updateUniformBuffer` function will start out with some logic to calculate the time in seconds since rendering has started with millisecond accuracy. If you need timing to be more precise, then you can use `std::chrono::microseconds` and divide by `1e6f`, which is short for `1000000.0f`.

We will now define the model, view and projection transformations in the uniform buffer object. The model rotation will be a simple rotation around the Z-axis using the `time` variable:

```
UniformBufferObject ubo = {};
ubo.model = glm::rotate(glm::mat4(), time * glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f)
```

The `glm::rotate` function takes an existing transformation, rotation angle and rotation axis as parameters. The `glm::mat4()` default constructor returns an identity matrix. Using a rotation angle of `time * glm::radians(90.0f)` accomplishes the purpose of rotation 90 degrees per second.

```
ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(
```

For the view transformation I've decided to look at the geometry from above at a 45 degree angle. The `glm::lookAt` function takes the eye position, center position and up axis as parameters.

```
ubo.proj = glm::perspective(glm::radians(45.0f), swapChainExtent.width / (float) swapChainEx
```

I've chosen to use a perspective projection with a 45 degree vertical field-of-view. The other parameters are the aspect ratio, near and far view planes. It is important to use the current swap chain extent to calculate the aspect ratio to take into account the new width and height of the window after a resize.

```
ubo.proj[1][1] *= -1;
```

GLM was originally designed for OpenGL, where the Y coordinate of the clip coordinates is inverted. The easiest way to compensate for that is to flip the sign on the scaling factor of the Y axis in the projection matrix. If you don't do this, then the image will be rendered upside down.

All of the transformations are defined now, so we can copy the data in the uniform buffer object to the uniform buffer. This happens in exactly the same way as we did for vertex buffers with a staging buffer:

```
void* data;
vkMapMemory(device, uniformStagingBufferMemory, 0, sizeof(ubo), 0, &data);
    memcpy(data, &ubo, sizeof(ubo));
vkUnmapMemory(device, uniformStagingBufferMemory);

copyBuffer(uniformStagingBuffer, uniformBuffer, sizeof(ubo));
```

Using a staging buffer and final buffer this way is not the most efficient way to pass frequently changing values to the shader. A more efficient way to pass a small buffer of data to shaders are *push constants*. We may look at these in a future chapter.

In the next chapter we'll look at descriptor sets, which will actually bind the `VkBuffer` to the uniform buffer descriptor so that the shader can access this transformation data.

C++ code / Vertex shader / Fragment shader

Descriptor pools and sets

Introduction

The descriptor layout from the previous chapter describes the type of descriptors that can be bound. In this chapter we're going to create a descriptor set, which will actually specify a `VkBuffer` resource to bind to the uniform buffer descriptor.

Descriptor pool

Descriptor sets can't be created directly, they must be allocated from a pool like command buffers. The equivalent for descriptor sets is unsurprisingly called a *descriptor pool*. We'll write a new function `createDescriptorPool` to set it up.

```
void initVulkan() {
    ...
    createUniformBuffer();
    createDescriptorPool();
    ...
}

...

void createDescriptorPool() {

}
```

We first need to describe which descriptor types our descriptor sets are going to contain and how many of them, using `VkDescriptorPoolSize` structures.

```
VkDescriptorPoolSize poolSize = {};
poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSize.descriptorCount = 1;
```

We only have a single descriptor right now with the uniform buffer type. This pool size structure is referenced by the main `VkDescriptorPoolCreateInfo`:

```
VkDescriptorPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = 1;
poolInfo.pPoolSizes = &poolSize;
```

We also need to specify the maximum number of descriptor sets that will be allocated:

```
poolInfo.maxSets = 1;
```

The structure has an optional flag similar to command pools that determines if individual descriptor sets can be freed or not: `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT`. We're not going to touch the descriptor set after creating it, so we don't need this flag. You can leave `flags` to its default value of `0`.

```
VDeleter<VkDescriptorPool> descriptorPool{device, vkDestroyDescriptorPool};

...

if (vkCreateDescriptorPool(device, &poolInfo, nullptr, descriptorPool.replace()) != VK_SUCC
    throw std::runtime_error("failed to create descriptor pool!");
}
```

Add a new class member to store the handle of the descriptor pool and call `vkCreateDescriptorPool` to create it.

Descriptor set

We can now allocate the descriptor set itself. Add a `createDescriptorSet` function for that purpose:

```
void initVulkan() {
    ...
    createDescriptorPool();
    createDescriptorSet();
    ...
}

...

void createDescriptorSet() {

}
```

A descriptor set allocation is described with a `VkDescriptorSetAllocateInfo` struct. You need to specify the descriptor pool to allocate from, the number of descriptor sets to allocate, and the descriptor layout to base them on:

```
VkDescriptorSetLayout layouts[] = {descriptorSetLayout};
VkDescriptorSetAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = 1;
allocInfo.pSetLayouts = layouts;
```

Add a class member to hold the descriptor set handle and allocate it with `vkAllocateDescriptorSets`:

```
VDeleter<VkDescriptorPool> descriptorPool{device, vkDestroyDescriptorPool};
VkDescriptorSet descriptorSet;

...

if (vkAllocateDescriptorSets(device, &allocInfo, &descriptorSet) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate descriptor set!");
}
```

You don't need to use a deleter for descriptor sets, because they will be automatically freed when the descriptor pool is destroyed. The call to `vkAllocateDescriptorSets` will allocate one descriptor set with one uniform buffer descriptor.

The descriptor set has been allocated now, but the descriptors within still need to be configured. Descriptors that refer to buffers, like our uniform buffer descriptor, are configured with a `VkDescriptorBufferInfo` struct. This structure specifies the buffer and the region within it that contains the data for the descriptor:

```
VkDescriptorBufferInfo bufferInfo = {};
bufferInfo.buffer = uniformBuffer;
bufferInfo.offset = 0;
bufferInfo.range = sizeof(UniformBufferObject);
```

The configuration of descriptors is updated using the `vkUpdateDescriptorSets` function, which takes an array of `VkWriteDescriptorSet` structs as parameter.

```
VkWriteDescriptorSet descriptorWrite = {};
descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrite.dstSet = descriptorSet;
descriptorWrite.dstBinding = 0;
descriptorWrite.dstArrayElement = 0;
```

The first two fields specify the descriptor set to update and the binding. We gave our uniform buffer binding index `0`. Remember that descriptors can be arrays, so we also need to specify the first index in the array that we want to update. We're not using an array, so the index is simply `0`.

```
descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrite.descriptorCount = 1;
```

We need to specify the type of descriptor again. It's possible to update multiple descriptors at once in an array, starting at index `dstArrayElement`. The `descriptorCount` field specifies how many array elements you want to update.

```
descriptorWrite.pBufferInfo = &bufferInfo;
descriptorWrite.pImageInfo = nullptr; // Optional
descriptorWrite.pTexelBufferView = nullptr; // Optional
```

The last field references an array with `descriptorCount` structs that actually configure the descriptors. It depends on the type of descriptor which one of the three you actually need to use. The `pBufferInfo` field is used for descriptors that refer to buffer data, `pImageInfo` is used for descriptors that refer to image data, and `pTexelBufferView` is used for descriptors that refer to buffer views. Our descriptor is based on buffers, so we're using `pBufferInfo`.

```
vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

The updates are applied using `vkUpdateDescriptorSets`. It accepts two kinds of arrays as parameters: an array of `VkWriteDescriptorSet` and an array of `VkCopyDescriptorSet`. The latter can be used to copy the configuration of descriptors, as its name implies.

Using a descriptor set

We now need to update the `createCommandBuffers` function to actually bind the descriptor set to the descriptors in the shader with `cmdBindDescriptorSets`:

```
vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout,
```

Unlike vertex and index buffers, descriptor sets are not unique to graphics pipelines. Therefore we need to specify if we want to bind descriptor sets to the graphics or compute pipeline. The next parameter is the layout that the descriptors are based on. The next three parameters specify the index of the first descriptor set, the number of sets to bind, and the array of sets to bind. We'll get back to this in a moment. The last two parameters specify an array of offsets that are used for dynamic descriptors. We'll look at these in a future chapter.

If you run your program now, then you'll notice that unfortunately nothing is visible. The problem is that because of the Y-flip we did in the projection matrix, the vertices are now being drawn in clockwise order instead of counter-clockwise order. This causes backface culling to kick in and prevents any geometry from being drawn. Go to the `createGraphicsPipeline` function and modify the `cullFace` in `VkPipelineRasterizationStateCreateInfo` to correct this:

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

Run your program again and you should now see the following:

The rectangle has changed into a square because the projection matrix now corrects for aspect ratio. The `updateUniformData` takes care of screen resizing, so we don't need to recreate the descriptor set in `recreateSwapChain`.

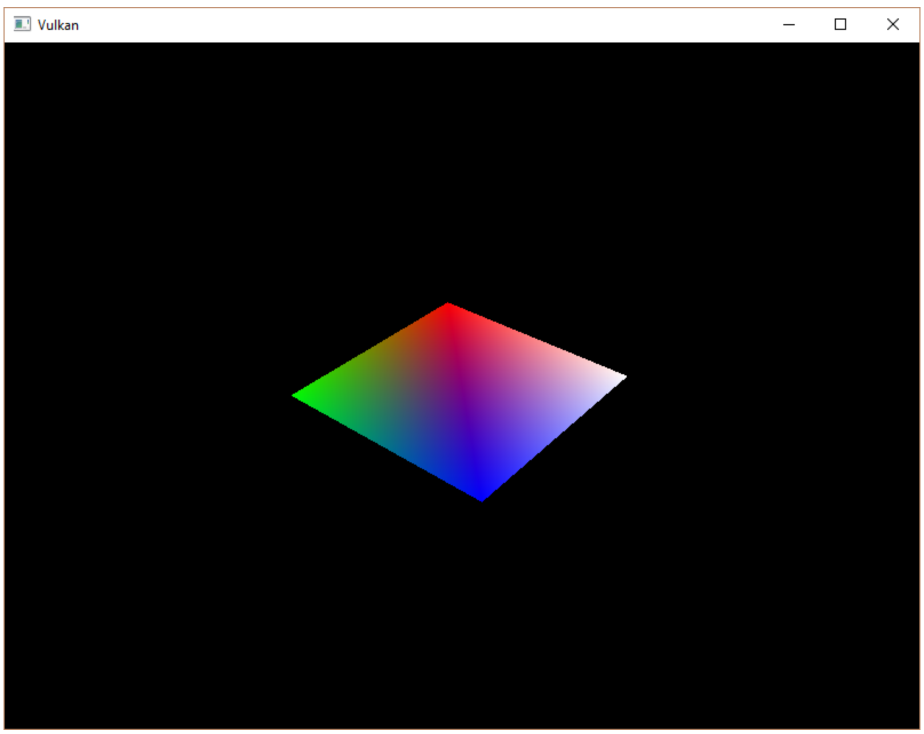

Figure 35:

Multiple descriptor sets

As some of the structures and function calls hinted at, it is actually possible to bind multiple descriptor sets. You need to specify a descriptor layout for each descriptor set when creating the pipeline layout. Shaders can then reference specific descriptor sets like this:

```
layout(set = 0, binding = 0) uniform UniformBufferObject { ... }
```

You can use this feature to put descriptors that vary per-object and descriptors that are shared into separate descriptor sets. In that case you avoid rebinding most of the descriptors across draw calls which is potentially more efficient.

C++ code / Vertex shader / Fragment shader

Images

Introduction

The geometry has been colored using per-vertex colors so far, which is a rather limited approach. In this part of the tutorial we're going to implement texture mapping to make the geometry look more interesting. This will also allow us to load and draw basic 3D models in a future chapter.

Adding a texture to our application will involve the following steps:

- Create an image object backed by device memory
- Fill it with pixels from an image file
- Create an image sampler
- Add a combined image sampler descriptor to sample colors from the texture

We've already worked with image objects before, but those were automatically created by the swap chain extension. This time we'll have to create one by ourselves. Creating an image and filling it with data is very similar to vertex buffer creation. You create a `VkImage`, query its memory requirements, allocate device memory, bind the memory to the image, and finally map the memory to upload the pixel data. We'll use a staging and final image again, to make sure that the texture image itself ends up in fast device local memory. There is a command to copy the contents of images similar to `vkCmdCopyBuffer`.

However, there is something extra that we'll have to take care of when working with images. Images can have different *layouts* that affect how the pixels are organized in memory. Due to the way graphics hardware works, simply storing the pixels row by row may not lead to the best performance, for example. When performing any operation on images, you must make sure that they have the layout that is optimal for use in that operation. We've actually already seen some of these layouts when we specified the render pass:

- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`: Optimal for presentation
- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`: Optimal as attachment for writing colors from the fragment shader
- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`: Optimal as source in a transfer operation, like `vkCmdCopyImage`
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`: Optimal as destination in a transfer operation, like `vkCmdCopyImage`
- `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`: Optimal for sampling from a shader

One of the most common ways to transition the layout of an image is a *pipeline barrier*. Pipeline barriers are primarily used for synchronizing access to resources, like making sure that an image was written to before it is read, but they can also be used to transition layouts. In this chapter we'll see how pipeline barriers are used for this purpose. Barriers can additionally be used to transfer queue family ownership when using `VK_SHARING_MODE_EXCLUSIVE`.

Image library

There are many libraries available for loading images, and you can even write your own code to load simple formats like BMP and PPM. In this tutorial we'll be using the stb_image library from the stb collection. The advantage of it is that all of the code is in a single file, so it doesn't require any tricky build configuration. Download `stb_image.h` and store it in a convenient location, like the directory where you saved GLFW and GLM. Add the location to your include path.

Visual Studio

Add the directory with `stb_image.h` in it to the `Additional Include Directories` paths.

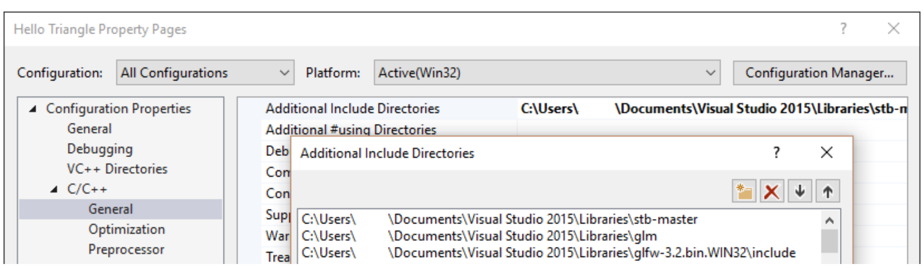

Figure 36:

Makefile

Add the directory with `stb_image.h` to the include directories for GCC:

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb

...

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH)
```

Loading an image

Include the image library like this:

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

The header only defines the prototypes of the functions by default. One code file needs to include the header with the `STB_IMAGE_IMPLEMENTATION` definition to include the function bodies, otherwise we'll get linking errors.

```
void initVulkan() {
    ...
    createCommandPool();
    createTextureImage();
    createVertexBuffer();
    ...
}

...

void createTextureImage() {

}
```

Create a new function `createTextureImage` where we'll load an image and upload it into a Vulkan image object. We're going to use command buffers, so it should be called after `createCommandPool`.

Create a new directory `textures` next to the `shaders` directory to store texture images in. We're going to load an image called `texture.jpg` from that directory. I've chosen to use the following CC0 licensed image resized to 512 x 512 pixels, but feel free to pick any image you want. The library supports most common image file formats, like JPEG, PNG, BMP and GIF.

Loading an image with this library is really easy:

```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels
    VkDeviceSize imageSize = texWidth * texHeight * 4;
```

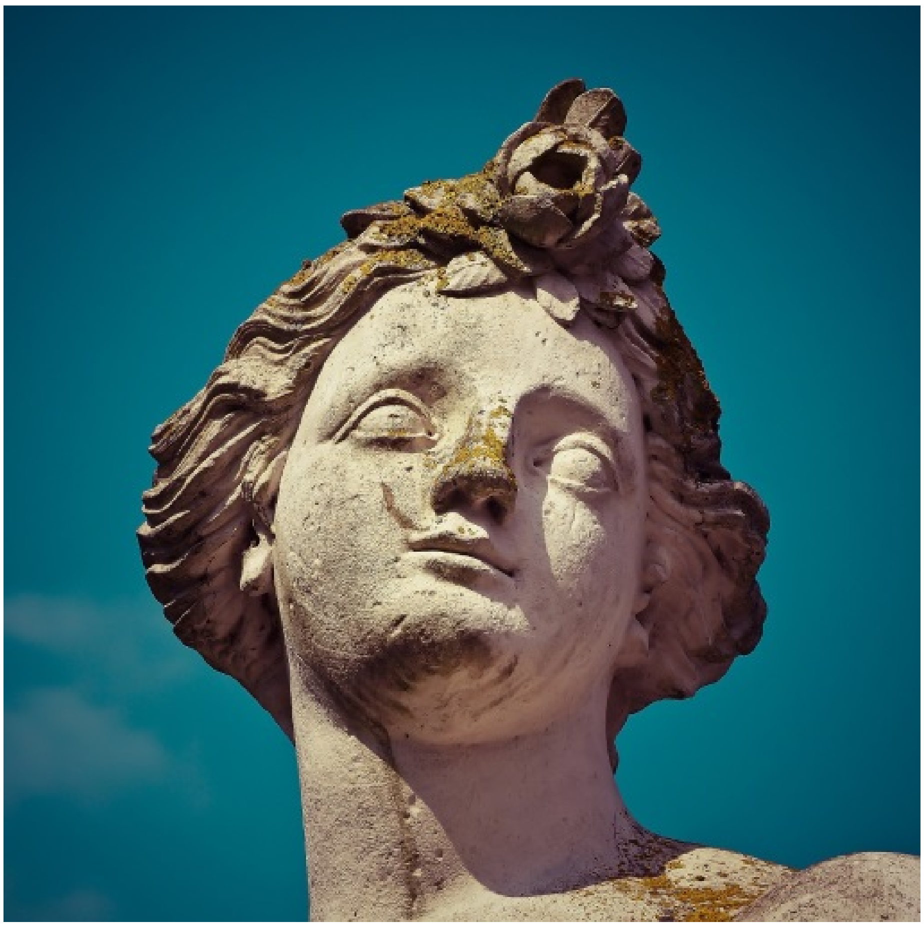

Figure 37:

```
    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }
}
```

The stbi_load function takes the file path and number of channels to load as arguments. The STBI_rgb_alpha value forces the image to be loaded with an alpha channel, even if it doesn't have one, which is nice for consistency with other textures in the future. The middle three parameters are outputs for the width, height and actual number of channels in the image. The pointer that is returned is the first element in an array of pixel values. The pixels are laid out row by row with 4 bytes per pixel in the case of STBI_rgba_alpha for a total of texWidth * texHeight * 4 values.

Staging image

We're now going to create an image in host visible memory so that we can use vkMapMemory and copy the pixels to it. Pixels within an image object are known as texels and we'll use that name from this point on. Add the following two variables in the createTextureImage function:

```
VDeleter<VkImage> stagingImage{device, vkDestroyImage};
VDeleter<VkDeviceMemory> stagingImageMemory{device, vkFreeMemory};
```

The parameters for an image are specified in a VkImageCreateInfo struct:

```
VkImageCreateInfo imageInfo = {};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType = VK_IMAGE_TYPE_2D;
imageInfo.extent.width = texWidth;
imageInfo.extent.height = texHeight;
imageInfo.extent.depth = 1;
imageInfo.mipLevels = 1;
imageInfo.arrayLayers = 1;
```

The image type, specified in the imageType field, tells Vulkan with that kind of coordinate system the texels in the image are going to be addressed. It is possible to create 1D, 2D and 3D images. One dimensional images can be used to store an array of data or gradient, two dimensional images are mainly used for textures, and three dimensional images can be used to store voxel volumes, for example. The extent field specifies the dimensions of the image, basically how many texels there are on each axis. That's why depth must be 1 instead of 0. Our texture will not be an array and we won't be using mipmapping for now.

```
imageInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
```

Vulkan supports many possible image formats, but it makes the most sense to use exactly the same format for the texels as the pixels loaded with the library.

```
imageInfo.tiling = VK_IMAGE_TILING_LINEAR;
```

The `tiling` field can have one of two values:

- `VK_IMAGE_TILING_LINEAR`: Texels are laid out in row-major order like our `pixels` array
- `VK_IMAGE_TILING_OPTIMAL`: Texels are laid out in an implementation defined order for optimal access

If you want to be able to directly access texels in the memory of the image, then you must use `VK_IMAGE_TILING_LINEAR`. We want to be able to directly copy the data in `pixels` to the staging image memory, so we should use it. Unlike the layout of an image, the tiling mode cannot be changed at a later time. We're going to use `VK_IMAGE_TILING_OPTIMAL` for the final image.

```
imageInfo.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
```

There are only two possible values for the `initialLayout` of an image:

- `VK_IMAGE_LAYOUT_UNDEFINED`: Not usable by the GPU and the very first transition will discard the texels.
- `VK_IMAGE_LAYOUT_PREINITIALIZED`: Not usable by the GPU, but the first transition will preserve the texels.

An initially undefined layout is suitable for images that will be used as attachments, like color and depth buffers. In that case we don't care about any initial data, because it'll probably be cleared by a render pass before use. If you want to fill it with data, like a texture, then you should use the preinitialized layout.

```
imageInfo.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
```

The `usage` field has the same semantics as the one during buffer creation. The staging image is going to be copied to the final texture image, so it should be set up as a transfer source.

```
imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

The staging image will only be used by one queue family: the one that supports transfer operations.

```
imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageInfo.flags = 0; // Optional
```

The `samples` flag is related to multisampling. This is only relevant for images that will be used as attachments, so stick to one sample. There are some optional flags for images that are related to sparse images. Sparse images are images where only certain regions are actually backed by memory. If you were using a 3D texture for a voxel terrain, for example, then you could use this to avoid allocating memory to store large volumes of "air" values. We won't be using it in this tutorial, so leave it to its default value of `0`.

```
if (vkCreateImage(device, &imageInfo, nullptr, stagingImage.replace()) != VK_SUCCESS) {
    throw std::runtime_error("failed to create image!");
}
```

The image is created using `vkCreateImage`, which doesn't have any particularly noteworthy parameters. It is possible that the `VK_FORMAT_R8G8B8A8_UNORM` format is not supported by the graphics hardware. You should have a list of acceptable alternatives and go with the best one that is supported. However, support for this particular format is so widespread that we'll skip this step. Using different formats would also require annoying conversions. We will get back to this in the depth buffer chapter, where we'll implement such a system.

```
VkMemoryRequirements memRequirements;
vkGetImageMemoryRequirements(device, stagingImage, &memRequirements);

VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, VK_MEMORY_PROPERT

if (vkAllocateMemory(device, &allocInfo, nullptr, stagingImageMemory.replace()) != VK_SUCCES
    throw std::runtime_error("failed to allocate image memory!");
}

vkBindImageMemory(device, stagingImage, stagingImageMemory, 0);
```

Allocating memory for an image works in exactly the same way as allocating memory for a buffer. Use `vkGetImageMemoryRequirements` instead of `vkGetBufferMemoryRequirements`, and use `vkBindImageMemory` instead of `vkBindBufferMemory`. Remember that we need the memory to be host visible to be able to use `vkMapMemory`, so you should specify that property when looking for the right memory type.

We can now use the `vkMapMemory` function to (temporarily) access the memory of the staging image directly from our application. It returns a pointer to the first byte in the memory buffer:

```
void* data;
vkMapMemory(device, stagingImageMemory, 0, imageSize, 0, &data);
```

Unfortunately we can't just copy the pixel bytes directly into the image memory with `memcpy` and assume that this works correctly. The problem is that there may be padding bytes between rows of pixels. In other words, the graphics card may assume that one row of pixels is not `texWidth * 4` bytes wide, but rather `texWidth * 4 + paddingBytes`. To handle this correctly, we need to query how bytes are arranged in our staging image using `vkGetImageSubresourceLayout`:

```
VkImageSubresource subresource = {};
subresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
```

```
subresource.mipLevel = 0;
subresource.arrayLayer = 0;

VkSubresourceLayout stagingImageLayout;
vkGetImageSubresourceLayout(device, stagingImage, &subresource, &stagingImageLayout);
```

Images contain one or more *subresources*, which are specific images within an image. For example, there is one subresource for every entry in an array image. In this case we don't have an array image, so there is simply one subresource at entry 0 and the base mipmapping level.

The `rowPitch` member of the `VkSubresourceLayout` struct specifies the total number of bytes of each row of pixels in the image. If this value is equal to `texWidth * 4`, then we're lucky and we *can* use `memcpy`, because there are no padding bytes in that case.

```
if (stagingImageLayout.rowPitch == texWidth * 4) {
    memcpy(data, pixels, (size_t) imageSize);
} else {

}
```

This is usually the case when your images have a power-of-2 size (e.g. 512 or 1024). Otherwise, we'll have to copy the pixels row-by-row using the right offset:

```
uint8_t* dataBytes = reinterpret_cast<uint8_t*>(data);

for (int y = 0; y < texHeight; y++) {
    memcpy(
        &dataBytes[y * stagingImageLayout.rowPitch],
        &pixels[y * texWidth * 4],
        texWidth * 4
    );
}
```

Each subsequent row in the image memory is offset by `rowPitch` and the original pixels are offset by `texWidth * 4` without padding bytes.

If you're done accessing the memory buffer, then you should unmap it with `vkUnmapMemory`. It is not necessary to call `vkUnmapMemory` now if you want to access the staging image memory again later on. The writes to the buffer will already be visible without calling this function.

```
void* data;
vkMapMemory(device, stagingImageMemory, 0, imageSize, 0, &data);

    if (stagingImageLayout.rowPitch == texWidth * 4) {
        memcpy(data, pixels, (size_t) imageSize);
    } else {
        uint8_t* dataBytes = reinterpret_cast<uint8_t*>(data);
```

```
        for (int y = 0; y < texHeight; y++) {
            memcpy(&dataBytes[y * stagingImageLayout.rowPitch], &pixels[y * texWidth * 4], t
        }
    }

vkUnmapMemory(device, stagingImageMemory);
```

Don't forget to clean up the original pixel array now:

```
stbi_image_free(pixels);
```

Texture image

We will now abstract image creation into a `createImage` function, like we did for buffers. Create the function and move the image object creation and memory allocation to it:

```
void createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling, Vk
    VkImageCreateInfo imageInfo = {};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    imageInfo.usage = usage;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateImage(device, &imageInfo, nullptr, image.replace()) != VK_SUCCESS) {
        throw std::runtime_error("failed to create image!");
    }

    VkMemoryRequirements memRequirements;
    vkGetImageMemoryRequirements(device, image, &memRequirements);

    VkMemoryAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);
```

```
    if (vkAllocateMemory(device, &allocInfo, nullptr, imageMemory.replace()) != VK_SUCCESS)
        throw std::runtime_error("failed to allocate image memory!");
    }

    vkBindImageMemory(device, image, imageMemory, 0);
}
```

I've made the width, height, format, tiling mode, usage, and memory properties parameters, because these will all vary between the images we'll be creating throughout this tutorial.

The `createTextureImage` function can now be simplified to:

```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }

    VDeleter<VkImage> stagingImage{device, vkDestroyImage};
    VDeleter<VkDeviceMemory> stagingImageMemory{device, vkFreeMemory};
    createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_TILING_LINEAR, VK_IM

    VkImageSubresource subresource = {};
    subresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    subresource.mipLevel = 0;
    subresource.arrayLayer = 0;

    VkSubresourceLayout stagingImageLayout;
    vkGetImageSubresourceLayout(device, stagingImage, &subresource, &stagingImageLayout);

    void* data;
    vkMapMemory(device, stagingImageMemory, 0, imageSize, 0, &data);

    if (stagingImageLayout.rowPitch == texWidth * 4) {
        memcpy(data, pixels, (size_t) imageSize);
    } else {
        uint8_t* dataBytes = reinterpret_cast<uint8_t*>(data);

        for (int y = 0; y < texHeight; y++) {
            memcpy(&dataBytes[y * stagingImageLayout.rowPitch], &pixels[y * texWidth * 4], t
        }
    }
```

```
    vkUnmapMemory(device, stagingImageMemory);

    stbi_image_free(pixels);
}
```

The next step is to create the actual texture image. Define two new class members to hold the handle to the image and its memory:

```
VDeleter<VkCommandPool> commandPool{device, vkDestroyCommandPool};
VDeleter<VkImage> textureImage{device, vkDestroyImage};
VDeleter<VkDeviceMemory> textureImageMemory{device, vkFreeMemory};
VDeleter<VkBuffer> vertexBuffer{device, vkDestroyBuffer};
```

The final texture image can now be created using the same function:

```
createImage(
    texWidth, texHeight,
    VK_FORMAT_R8G8B8A8_UNORM,
    VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
    textureImage,
    textureImageMemory
);
```

The dimensions of the image should be the same as the staging image. The formats should also be *compatible*, because the command simply copies the raw image data. Two color formats are compatible if they have the same number of bytes per pixel. Depth/stencil formats, which we'll see in one of the next chapters, need to be exactly equal. The tiling mode on the other hand does not need to be the same. The texture image will be used as the destination in the transfer, and we want to be able to sample texels from it in the shader. The `VK_IMAGE_USAGE_SAMPLED_BIT` flag is necessary to allow that. The memory of the image should be device local for best performance, just like the vertex buffer.

Layout transitions

The function we're going to write now involves recording and executing a command buffer again, so now's a good time to move that logic into a helper function or two:

```
VkCommandBuffer beginSingleTimeCommands() {
    VkCommandBufferAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;
```

```
    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);

    VkCommandBufferBeginInfo beginInfo = {};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    vkBeginCommandBuffer(commandBuffer, &beginInfo);

    return commandBuffer;
}

void endSingleTimeCommands(VkCommandBuffer commandBuffer) {
    vkEndCommandBuffer(commandBuffer);

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(graphicsQueue);

    vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
}
```

The code for these functions is based on the existing code in `copyBuffer`. You can now simplify that function to:

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    VkBufferCopy copyRegion = {};
    copyRegion.size = size;
    vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

    endSingleTimeCommands(commandBuffer);
}
```

If we were still using buffers, then we could now write a function to record and execute `vkCmdCopyImage` to finish the job, but this command requires the images to be in the right layout first. Create a new function to handle layout transitions:

```
void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageL
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    endSingleTimeCommands(commandBuffer);
```

```
}
```

One of the most common ways to perform layout transitions is using an *image memory barrier*. A pipeline barrier like that is generally used to synchronize access to resources, like ensuring that a write to a buffer completes before reading from it, but it can also be used to transition image layouts and transfer queue family ownership when `VK_SHARING_MODE_EXCLUSIVE` is used. There is an equivalent *buffer memory barrier* to do this for buffers.

```
VkImageMemoryBarrier barrier = {};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.oldLayout = oldLayout;
barrier.newLayout = newLayout;
```

The first two fields specify layout transition. It is possible to use `VK_IMAGE_LAYOUT_UNDEFINED` as `oldLayout` if you don't care about the existing contents of the image.

```
barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
```

If you are using the barrier to transfer queue family ownership, then these two fields should be the indices of the queue families. They must be set to `VK_QUEUE_FAMILY_IGNORED` if you don't want to do this (not the default value!).

```
barrier.image = image;
barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
barrier.subresourceRange.baseMipLevel = 0;
barrier.subresourceRange.levelCount = 1;
barrier.subresourceRange.baseArrayLayer = 0;
barrier.subresourceRange.layerCount = 1;
```

The `image` and `subresourceRange` specify the image that is affected and the specific part of the image. Our image is not an array and does not mipmapping levels, so only one level and layer are specified.

```
barrier.srcAccessMask = 0; // TODO
barrier.dstAccessMask = 0; // TODO
```

Barriers are primarily used for synchronization purposes, so you must specify which types of operations that involve the resource must happen before the barrier, and which operations that involve the resource must wait on the barrier. We need to do that despite already using `vkQueueWaitIdle` to manually synchronize. The right values depend on the old and new layout, so we'll get back to this once we've figured out which transitions we're going to use.

```
vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
    0,
    0, nullptr,
```

```
    0, nullptr,
    1, &barrier
);
```

All types of pipeline barriers are submitted using the same function. The first parameter specifies in which pipeline stage the operations occur that should happen before the barrier. The second parameter specifies the pipeline stage in which operations will wait on the barrier. We want it to happen immediately, so we're going with the top of the pipeline.

The third parameter is either `0` or `VK_DEPENDENCY_BY_REGION_BIT`. The latter turns the barrier into a per-region condition. That means that the implementation is allowed to already begin reading from the parts of a resource that were written so far, for example.

The last three pairs of parameters reference arrays of pipeline barriers of the three available types: memory barriers, buffer memory barriers, and image memory barriers like the one we're using here. Note that we're not using the `VkFormat` parameter yet, but we'll be using that one for special transitions in the depth buffer chapter.

Copying images

Before we get back to `createTextureImage`, we're going to write one more helper function: `copyImage`:

```
void copyImage(VkImage srcImage, VkImage dstImage, uint32_t width, uint32_t height) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    endSingleTimeCommands(commandBuffer);
}
```

Just like with buffers, you need to specify which part of the image needs to be copied to which part of the other image. This happens through `VkImageCopy` structs:

```
VkImageSubresourceLayers subResource = {};
subResource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
subResource.baseArrayLayer = 0;
subResource.mipLevel = 0;
subResource.layerCount = 1;

VkImageCopy region = {};
region.srcSubresource = subResource;
region.dstSubresource = subResource;
region.srcOffset = {0, 0, 0};
region.dstOffset = {0, 0, 0};
region.extent.width = width;
```

```
region.extent.height = height;
region.extent.depth = 1;
```

All of these fields are fairly self-explanatory. Image copy operations are enqueued using the `vkCmdCopyImage` function:

```
vkCmdCopyImage(
    commandBuffer,
    srcImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
    dstImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    1, &region
);
```

The first two pairs of parameters specify the source image/layout and destination image/layout. I'm assuming here that they've been previously transitioned to the optimal transfer layouts.

Preparing the texture image

We now have all of the tools we need to finish setting up the texture image, so we're going back to the `createTextureImage` function. The last thing we did there was creating the texture image. The next step is to copy the staging image to the texture image. This involves three operations:

- Transition the staging image to `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`
- Transition the texture image to `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`
- Execute the image copy operation

This is easy to do with the functions we just created:

```
transitionImageLayout(stagingImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_PREINITIALIZED
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_PREINITIALIZED
copyImage(stagingImage, textureImage, texWidth, texHeight);
```

Both `VK_IMAGE_LAYOUT_PREINITIALIZED` and `VK_IMAGE_LAYOUT_UNDEFINED` are valid values for old layout when transitioning `textureImage`, because we don't care about its contents before the copy operation.

To be able to start sampling from the texture image in the shader, we need one last transition:

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_LAYOUT_TRANSFER_DST_(
```

Transition barrier masks

If run your application with validation layers enabled now, then you'll see that it complains about the access masks in `transitionImageLayout` being invalid. We still need to set those based on the layouts in the transition.

There are three transitions we need to handle:

- Preinitialized → transfer source: transfer reads should wait on host writes
- Preinitialized → transfer destination: transfer writes should wait on host writes
- Transfer destination → shader reading: shader reads should wait on transfer writes

These rules are specified using the following access masks:

```
if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_SRC
    barrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANS
    barrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
} else {
    throw std::invalid_argument("unsupported layout transition!");
}
```

If we need to do more transitions in the future, then we'll extend the function. The application should now run successfully, although there are of course no visual changes yet. One thing to note is that command buffer submission results in implicit `VK_ACCESS_HOST_WRITE_BIT` synchronization at the beginning. Since the `transitionImageLayout` function executes a command buffer with only a single command, we can use this implicit synchronization and set `srcAccessMask` to `0` for the first two types of transitions. It's up to you if you want to be explicit about it or not, but I'm personally not a fan of relying on these OpenGL-like "hidden" operations.

There is actually a special type of image layout that supports all operations, `VK_IMAGE_LAYOUT_GENERAL`. The problem with it, of course, is that it doesn't necessarily offer the best performance for any operation. It is required for some special cases, like using an image as both input and output, or for reading an image after it has left the preinitialized layout.

All of the helper functions that submit commands so far have been set up to execute synchronously by waiting for the queue to become idle. For practical applications it is recommended to combine these operations in a single command buffer and execute them asynchronously for higher throughput, especially the transitions and copy in the `createTextureImage` function. Try to experiment with this by creating a `setupCommandBuffer` that the helper functions record commands into, and add a `flushSetupCommands` to execute the commands that have been recorded so far. It's best to do this after the texture mapping works to check if the texture resources are still set up correctly.

In this tutorial we used another image as staging resource for the texture, but it's also possible to use a buffer and copy pixels from it using `vkCmdCopyBufferToImage`. It is recommended to use this approach for improved performance on some hardware if you need to update the data in an image often.

The image now contains the texture, but we still need a way to access it from the graphics pipeline. We'll work on that in the next chapter.

C++ code / Vertex shader / Fragment shader # Image view and sampler

In this chapter we're going to create two more resources that are needed for the graphics pipeline to sample an image. The first resource is one that we've already seen before while working with the swap chain images, but the second one is new - it relates to how the shader will read texels from the image.

Texture image view

We've seen before, with the swap chain images and the framebuffer, that images are accessed through image views rather than directly. We will also need to create such an image view for the texture image.

Add a class member to hold a `VkImageView` for the texture image and create a new function `createTextureImageView` where we'll create it:

```
VDeleter<VkImageView> textureImageView{device, vkDestroyImageView};

...

void initVulkan() {
    ...
    createTextureImage();
    createTextureImageView();
    createVertexBuffer();
    ...
}

...

void createTextureImageView() {

}
```

The code for this function can be based directly on `createImageViews`. The only two changes you have to make are the `format` and the `image`:

```
VkImageViewCreateInfo viewInfo = {};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
```

```
viewInfo.image = textureImage;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;
```

I've left out the explicit `viewInfo.components` initialization, because `VK_COMPONENT_SWIZZLE_IDENTITY` is defined as `0` anyway. Finish creating the image view by calling `vkCreateImageView`:

```
if (vkCreateImageView(device, &viewInfo, nullptr, textureImageView.replace()) != VK_SUCCESS)
    throw std::runtime_error("failed to create texture image view!");
}
```

Because so much of the logic is duplicated from `createImageViews`, you may wish to abstract it into a new `createImageView` function:

```
void createImageView(VkImage image, VkFormat format, VDeleter<VkImageView>& imageView) {
    VkImageViewCreateInfo viewInfo = {};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    if (vkCreateImageView(device, &viewInfo, nullptr, imageView.replace()) != VK_SUCCESS) {
        throw std::runtime_error("failed to create texture image view!");
    }
}
```

The `createTextureImageView` function can now be simplified to:

```
void createTextureImageView() {
    createImageView(textureImage, VK_FORMAT_R8G8B8A8_UNORM, textureImageView);
}
```

And `createImageViews` can be simplified to:

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size(), VDeleter<VkImageView>{device, vkDestr

    for (uint32_t i = 0; i < swapChainImages.size(); i++) {
        createImageView(swapChainImages[i], swapChainImageFormat, swapChainImageViews[i]);
```

```
    }
}
```

Samplers

It is possible for shaders to read texels directly from images, but that is not very common when they are used as textures. Textures are usually accessed through samplers, which will apply filtering and transformations to compute the final color that is retrieved.

These filters are helpful to deal with problems like oversampling. Consider a texture that is mapped to geometry with more fragments than texels. If you simply took the closest texel for the texture coordinate in each fragment, then you would get a result like the first image:

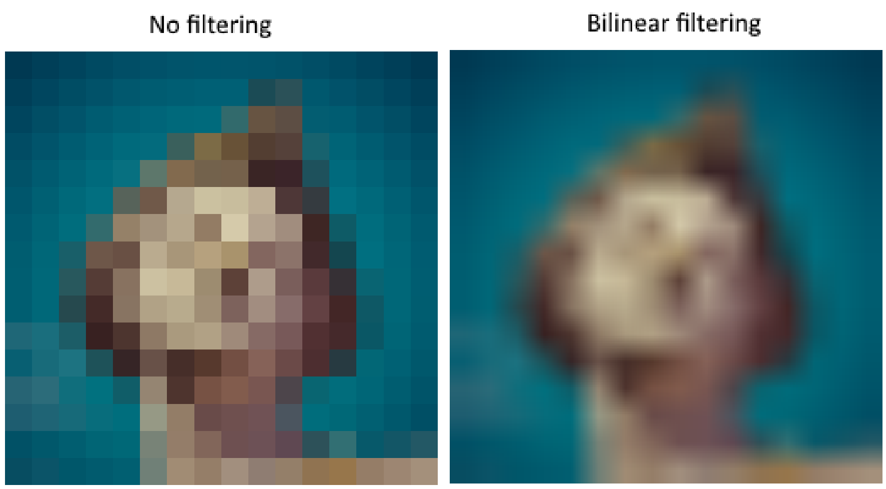

Figure 38:

If you combined the 4 closest texels through linear interpolation, then you would get a smoother result like the one on the right. Of course your application may have art style requirements that fit the left style more (think Minecraft), but the right is preferred in conventional graphics applications. A sampler object automatically applies this filtering for you when reading a color from the texture.

Undersampling is the opposite problem, where you have more texels than fragments. This will lead to artifacts when sampling high frequency patterns like a checkerboard texture at a sharp angle:

As shown in the left image, the texture turns into a blurry mess in the distance. The solution to this is anisotropic filtering, which can also be applied

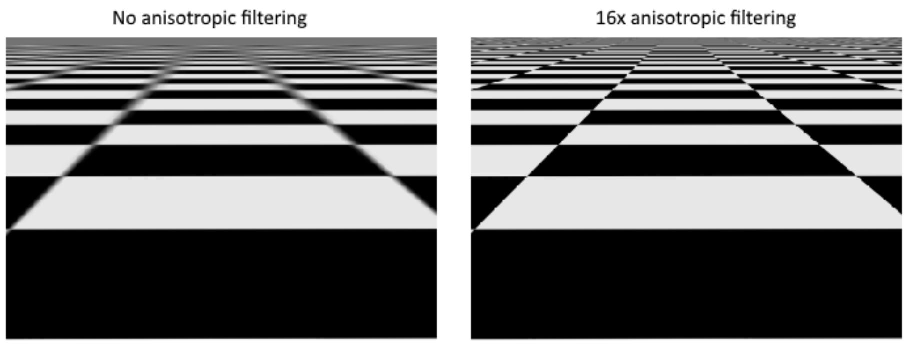

Figure 39:

automatically by a sampler.

Aside from these filters, a sampler can also take care of transformations. It determines what happens when you try to read texels outside the image through its *addressing mode*. The image below displays some of the possibilities:

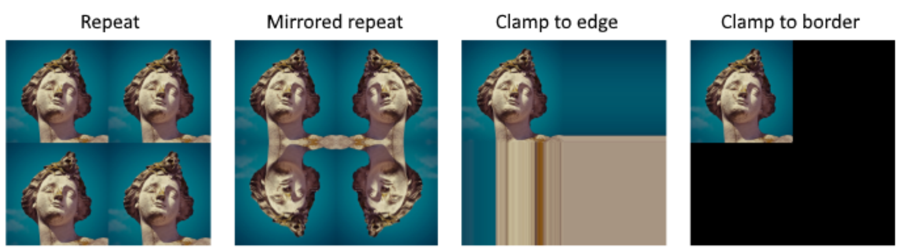

Figure 40:

We will now create a function `createTextureSampler` to set up such a sampler object. We'll be using that sampler to read colors from the texture in the shader later on.

```
void initVulkan() {
    ...
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    ...
}

...

void createTextureSampler() {
```

```
}
```

Samplers are configured through a `VkSamplerCreateInfo` structure, which specifies all filters and transformations that it should apply.

```
VkSamplerCreateInfo samplerInfo = {};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
```

The `magFilter` and `minFilter` fields specify how to interpolate texels that are magnified or minified. Magnification concerns the oversampling problem describes above, and minification concerns undersampling. The choices are `VK_FILTER_NEAREST` and `VK_FILTER_LINEAR`, corresponding to the modes demonstrated in the images above.

```
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

The addressing mode can be specified per axis using the `addressMode` fields. The available values are listed below. Most of these are demonstrated in the image above. Note that the axes are called U, V and W instead of X, Y and Z. This is a convention for texture space coordinates.

- `VK_SAMPLER_ADDRESS_MODE_REPEAT`: Repeat the texture when going beyond the image dimensions.
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT`: Like repeat, but inverts the coordinates to mirror the image when going beyond the dimensions.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`: Take the color of the edge closest to the coordinate beyond the image dimensions.
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE`: Like clamp to edge, but instead uses the edge opposite to the closest edge.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`: Return a solid color when sampling beyond the dimensions of the image.

It doesn't really matter which addressing mode we use here, because we're not going to sample outside of the image in this tutorial. However, the repeat mode is probably the most common mode, because it can be used to tile textures like floors and walls.

```
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
```

These two fields specify if anisotropic filtering should be used. There is no reason not to use this unless performance is a concern. The `maxAnisotropy` field limits the amount of texel samples that can be used to calculate the final color. A lower value results in better performance, but lower quality results. There is no graphics hardware available today that will use more than 16 samples, because the difference is negligible beyond that point.

```
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK ;
```

The `borderColor` field specifies which color is returned when sampling beyond the image with clamp to border addressing mode. It is possible to return black, white or transparent in either float or int formats. You cannot specify an arbitrary color.

```
samplerInfo.unnormalizedCoordinates = VK_FALSE;
```

The `unnormalizedCoordinates` field specifies which coordinate system you want to use to address texels in an image. If this field is `VK_TRUE`, then you can simply use coordinates within the `[0, texWidth)` and `[0, texHeight)` range. If it is `VK_FALSE`, then the texels are addressed using the `[0, 1)` range on all axes. Real-world applications almost always use normalized coordinates, because then it's possible to use textures of varying resolutions with the exact same coordinates.

```
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
```

If a comparison function is enabled, then texels will first be compared to a value, and the result of that comparison is used in filtering operations. This is mainly used for percentage-closer filtering on shadow maps. We'll look at this in a future chapter.

```
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = 0.0f;
```

All of these fields apply to mipmapping. We will look at mipmapping in a future chapter, but basically it's another type of filter that can be applied.

The functioning of the sampler is now fully defined. Add a class member to hold the handle of the sampler object and create the sampler with `vkCreateSampler`:

```
VDeleter<VkImageView> textureImageView{device, vkDestroyImageView};
VDeleter<VkSampler> textureSampler{device, vkDestroySampler};

...

void createTextureSampler() {
    ...

    if (vkCreateSampler(device, &samplerInfo, nullptr, textureSampler.replace()) != VK_SUCC
        throw std::runtime_error("failed to create texture sampler!");
    }
}
```

Note the sampler does not reference a `VkImage` anywhere. The sampler is a distinct object that provides an interface to extract colors from a texture. It can be applied to any image you want, whether it is 1D, 2D or 3D. This is different from many older APIs, which combined texture images and filtering into a single state.

In the next chapter we will expose the image and sampler objects to the shaders to draw the texture onto the square.

C++ code / Vertex shader / Fragment shader # Combined image sampler

Introduction

We looked at descriptors for the first time in the uniform buffers part of the tutorial. In this chapter we will look at a new type of descriptor: *combined image sampler*. This descriptor makes it possible for shaders to access an image resource through a sampler object like the one we created in the previous chapter.

We'll start by modifying the descriptor layout, descriptor pool and descriptor set to include such a combined image sampler descriptor. After that, we're going to add texture coordinates to `Vertex` and modify the fragment shader to read colors from the texture instead of just interpolating the vertex colors.

Updating the descriptors

Browse to the `createDescriptorSetLayout` function and add a `VkDescriptorSetLayoutBinding` for a combined image sampler descriptor. We'll simply put it in the binding after the uniform buffer:

```
VkDescriptorSetLayoutBinding samplerLayoutBinding = {};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.pImmutableSamplers = nullptr;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

std::array<VkDescriptorSetLayoutBinding, 2> bindings = {uboLayoutBinding, samplerLayoutBindi
VkDescriptorSetLayoutCreateInfo layoutInfo = {};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = bindings.size();
layoutInfo.pBindings = bindings.data();
```

Make sure to set the `stageFlags` to indicate that we intend to use the combined image sampler descriptor in the fragment shader. That's where the color of the fragment is going to be determined. It is possible to use texture sampling in the vertex shader, for example to dynamically deform a grid of vertices by a heightmap.

If you would run the application with validation layers now, then you'll see that it complains that the descriptor pool cannot allocate a descriptor set with this layout, because it doesn't have any combined image sampler descriptors. Go to the `createDescriptorPool` function and modify it to include a `VkDescriptorPoolSize` for this descriptor:

```
std::array<VkDescriptorPoolSize, 2> poolSizes = {};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = 1;
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = 1;

VkDescriptorPoolCreateInfo poolInfo = {};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = poolSizes.size();
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = 1;
```

The final step is to bind the actual image and sampler resources to the descriptor in the descriptor set. Go to the `createDescriptorSet` function.

```
VkDescriptorImageInfo imageInfo = {};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;
```

The resources for a combined image sampler structure must be specified in a `VkDescriptorImageInfo` struct, just like the buffer resource for a uniform buffer descriptor is specified in a `VkDescriptorBufferInfo` struct. This is where the objects from the previous chapter come together.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites = {};

descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSet;
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount = 1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSet;
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
```

```
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device, descriptorWrites.size(), descriptorWrites.data(), 0, nullptr
```

The descriptor must be updated with this image info, just like the buffer. This time we're using the `pImageInfo` array instead of `pBufferInfo`. The descriptor is now ready to be used by the shaders!

Texture coordinates

There is one important ingredient for texture mapping that is still missing, and that's the actual coordinates for each vertex. The coordinates determine how the image is actually mapped to the geometry.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription = {};
        bindingDescription.binding = 0;
        bindingDescription.stride = sizeof(Vertex);
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions = {};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        attributeDescriptions[1].binding = 0;
        attributeDescriptions[1].location = 1;
        attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[1].offset = offsetof(Vertex, color);

        attributeDescriptions[2].binding = 0;
        attributeDescriptions[2].location = 2;
        attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[2].offset = offsetof(Vertex, texCoord);
```

```
        return attributeDescriptions;
    }
};
```

Modify the `Vertex` struct to include a `vec2` for texture coordinates. Make sure to also add a `VkVertexInputAttributeDescription` so that we can use access texture coordinates as input in the vertex shader. That is necessary to be able to pass them to the fragment shader for interpolation across the surface of the square.

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};
```

In this tutorial, I will simply fill the square with the texture by using coordinates from `0, 0` in the top-left corner to `1, 1` in the bottom-right corner. Feel free to experiment with different coordinates. Try using coordinates below `0` or above `1` to see the addressing modes in action!

Shaders

The final step is modifying the shaders to sample colors from the texture. We first need to modify the vertex shader to pass through the texture coordinates to the fragment shader:

```
layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 2) in vec2 inTexCoord;

layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Just like the per vertex colors, the `fragTexCoord` values will be smoothly interpolated across the area of the square by the rasterizer. We can visualize this by having the fragment shader output the texture coordinates as colors:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable
```

```
layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;

layout(location = 0) out vec4 outColor;

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

You should see something like the image below. Don't forget to recompile the shaders!

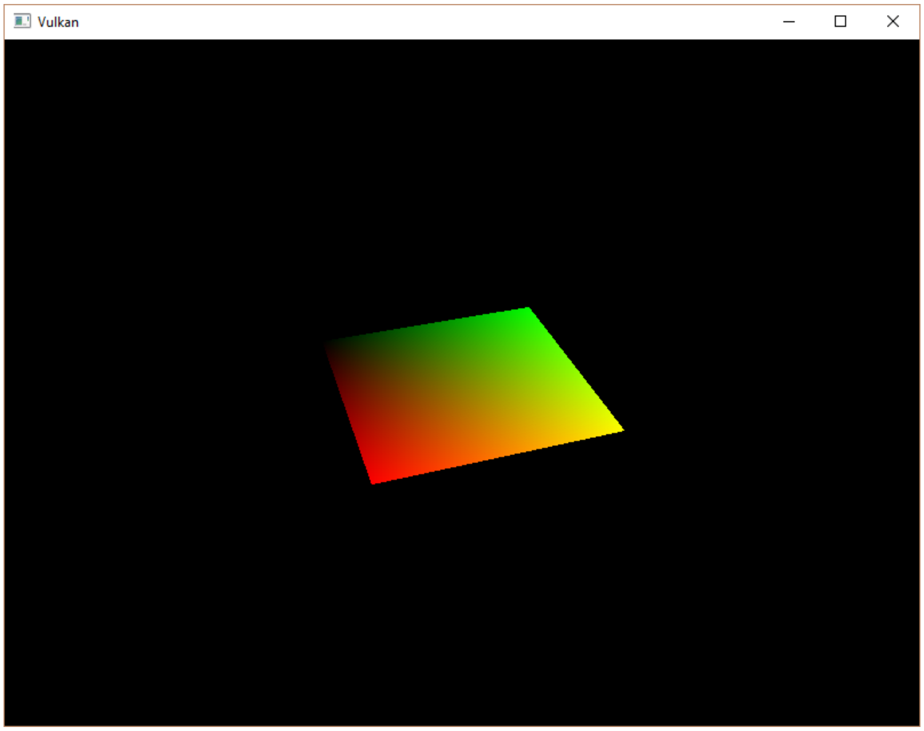

Figure 41:

The green channel represents the horizontal coordinates and the red channel the vertical coordinates. The black and yellow corners confirm that the texture coordinates are correctly interpolated from `0, 0` to `1, 1` across the square. Visualizing data using colors is the shader programming equivalent of `printf` debugging, for lack of a better option!

A combined image sampler descriptor is represented in GLSL by a sampler uniform. Add a reference to it in the fragment shader:

```
layout(binding = 1) uniform sampler2D texSampler;
```

There are equivalent `sampler1D` and `sampler3D` types for other types of images. Make sure to use the correct binding here.

```
void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

Textures are sampled using the built-in `texture` function. It takes a `sampler` and coordinate as arguments. The sampler automatically takes care of the filtering and transformations in the background. You should now see the texture on the square when you run the application:

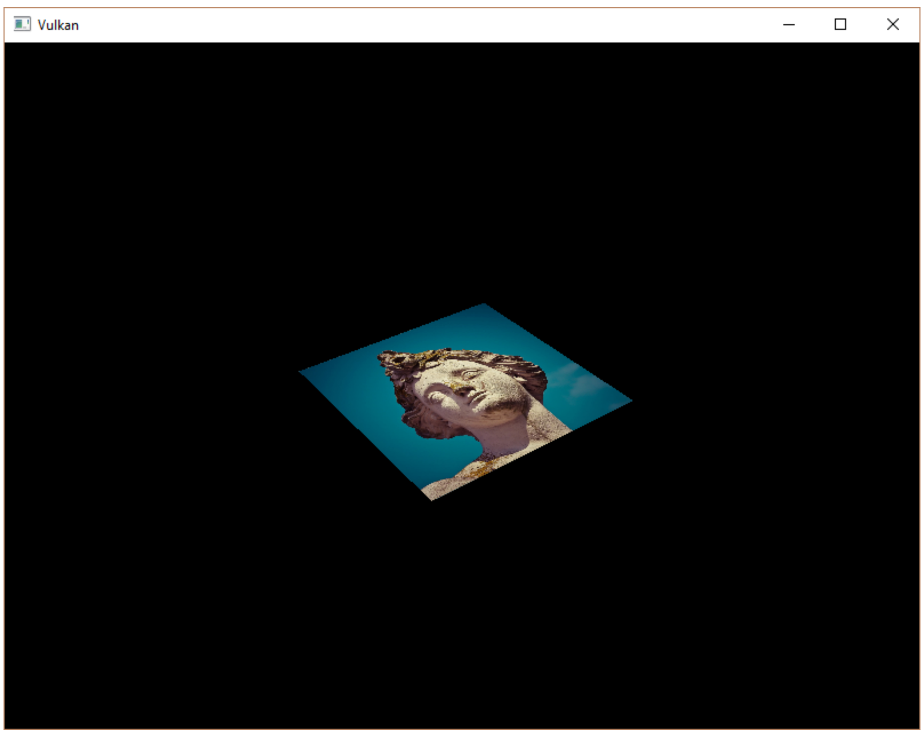

Figure 42:

Try experimenting with the addressing modes by scaling the texture coordinates to values higher than `1`. For example, the following fragment shader produces the result in the image below when using `VK_SAMPLER_ADDRESS_MODE_REPEAT`:

```
void main() {
    outColor = texture(texSampler, fragTexCoord * 2.0);
}
```

You can also manipulate the texture colors using the vertex colors:

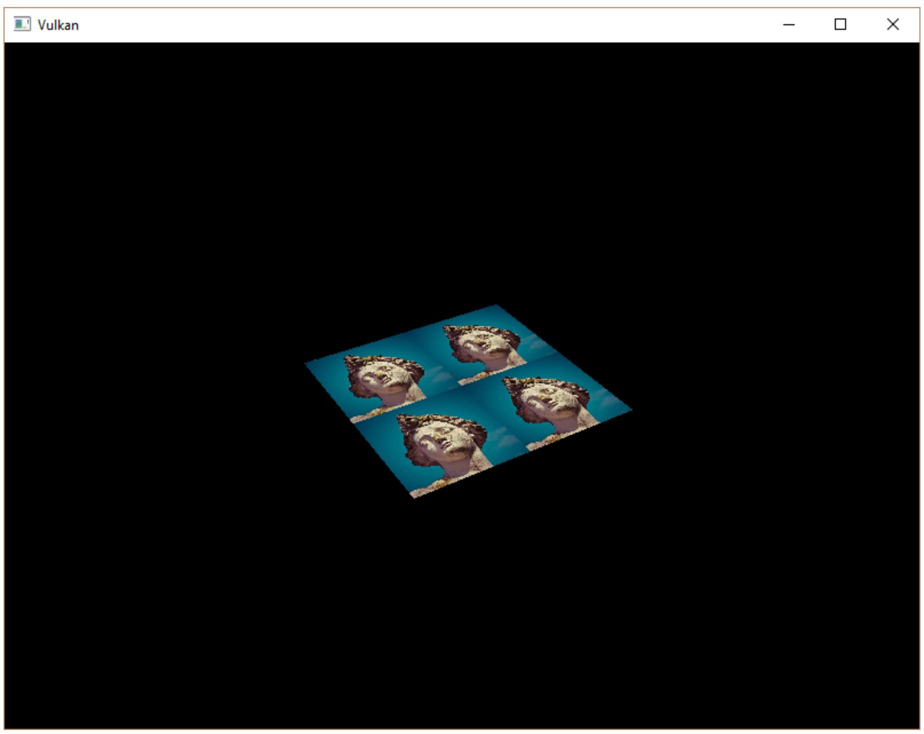

Figure 43:

```
void main() {
    outColor = vec4(fragColor * texture(texSampler, fragTexCoord).rgb, 1.0);
}
```

I've separated the RGB and alpha channels here to not scale the alpha channel.

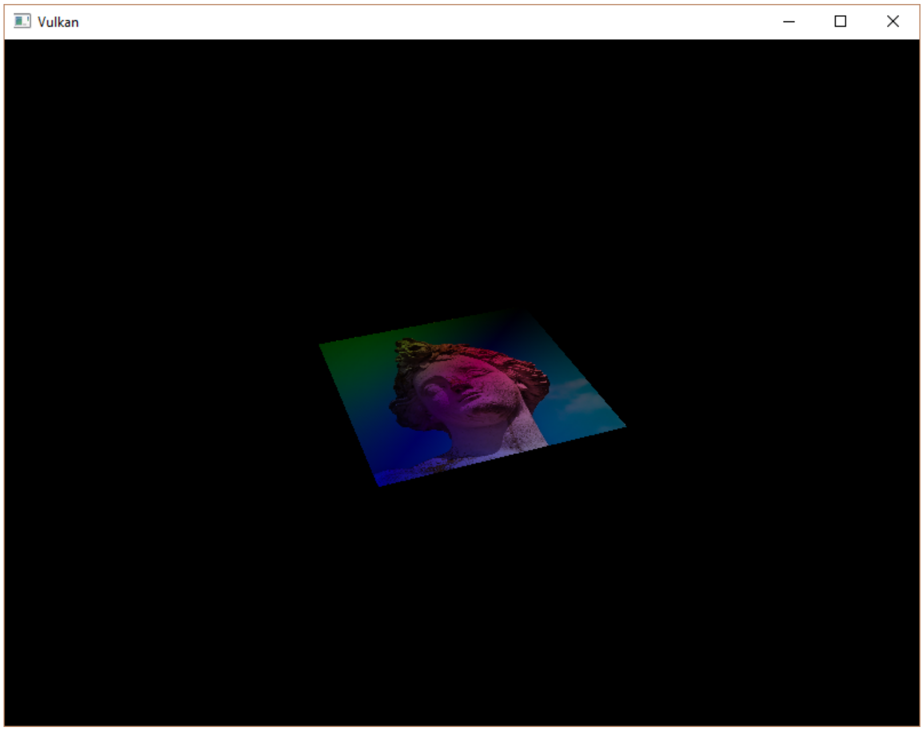

Figure 44:

You now know how to access images in shaders! This is a very powerful technique when combined with images that are also written to in framebuffers. You can use these images as inputs to implement cool effects like post-processing and camera displays within the 3D world.

C++ code / Vertex shader / Fragment shader # Depth buffering

Introduction

The geometry we've worked with so far is projected into 3D, but it's still completely flat. In this chapter we're going to add a Z coordinate to the position to prepare for 3D meshes. We'll use this third coordinate to place a square over the current square to see a problem that arises when geometry is not sorted by depth.

3D geometry

Change the `Vertex` struct to use a 3D vector for the position, and update the `format` in the corresponding `VkVertexInputAttributeDescription`:

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    ...

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions = {};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        ...
    }
};
```

Next, update the vertex shader to accept and transform 3D coordinates as input. Don't forget to recompile it afterwards!

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

...

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Lastly, update the `vertices` container to include Z coordinates:

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};
```

If you run your application now, then you should see exactly the same result as before. It's time to add some extra geometry to make the scene more interesting, and to demonstrate the problem that we're going to tackle in this chapter. Duplicate the vertices to define positions for a square right under the current one like this:

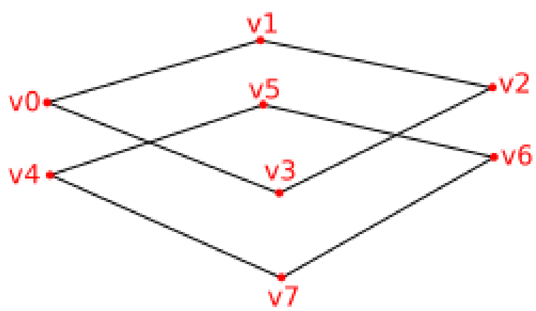

Figure 45:

Use Z coordinates of `-0.5f` and add the appropriate indices for the extra square:

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}},

    {{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4
};
```

Run your program now and you'll see something resembling an Escher illustration:

The problem is that the fragments of the lower square are drawn over the fragments of the upper square, simply because it comes later in the index array. There are two ways to solve this:

- Sort all of the draw calls by depth from back to front
- Use depth testing with a depth buffer

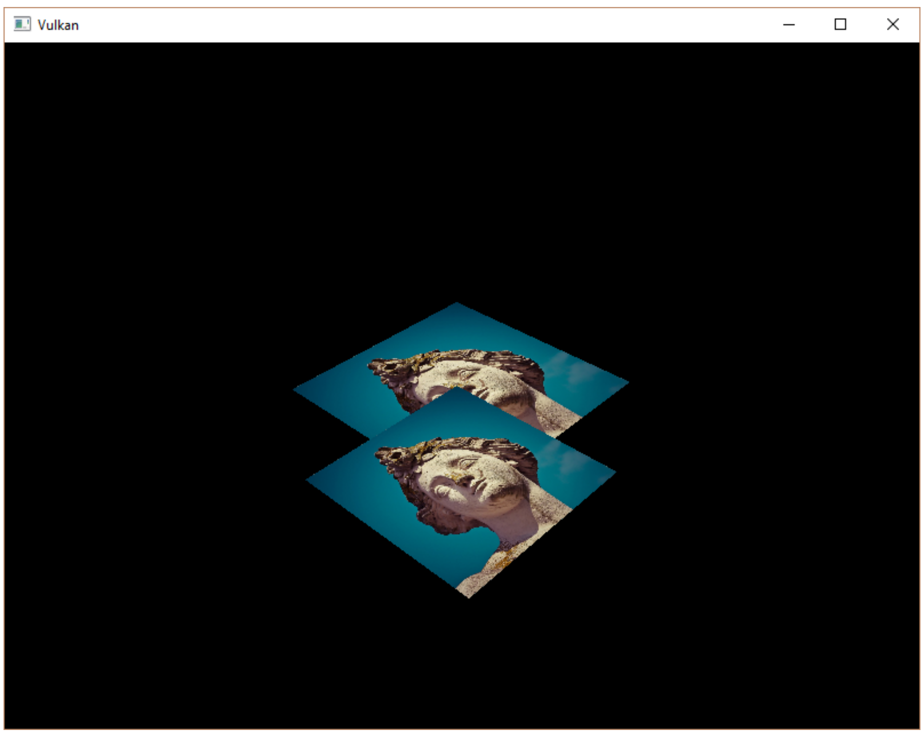

Figure 46:

The first approach is commonly used for drawing transparent objects, because order-independent transparency is a difficult challenge to solve. However, the problem of ordering fragments by depth is much more commonly solved using a *depth buffer*. A depth buffer is an additional attachment that stores the depth for every position, just like the color attachment stores the color of every position. Every time the rasterizer produces a fragment, the depth test will check if the new fragment is closer than the previous one. If it isn't, then the new fragment is discarded. A fragment that passes the depth test writes its own depth to the depth buffer. It is possible to manipulate this value from the fragment shader, just like you can manipulate the color output.

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

The perspective projection matrix generated by GLM will use the OpenGL depth range of `-1.0` to `1.0` by default. We need to configure it to use the Vulkan range of `0.0` to `1.0` using the `GLM_FORCE_DEPTH_ZERO_TO_ONE` definition.

Depth image and view

A depth attachment is based on an image, just like the color attachment. The difference is that the swap chain will not automatically create depth images for us. We only need a single depth image, because only one draw operation is running at once. The depth image will again require the trifecta of resources: image, memory and image view.

```
VDeleter<VkImage> depthImage{device, vkDestroyImage};
VDeleter<VkDeviceMemory> depthImageMemory{device, vkFreeMemory};
VDeleter<VkImageView> depthImageView{device, vkDestroyImageView};
```

Create a new function `createDepthResources` to set up these resources:

```
void initVulkan() {
    ...
    createCommandPool();
    createDepthResources();
    createTextureImage();
    ...
}

...

void createDepthResources() {

}
```

Creating a depth image is fairly straightforward. It should have the same resolution as the color attachment, defined by the swap chain extent, an image usage appropriate for a depth attachment, optimal tiling and device local memory. The only question is: what is the right format for a depth image? The format must contain a depth component, indicated by `_D??_` in the `VK_FORMAT_`.

Unlike the texture image, we don't necessarily need a specific format, because we won't be directly accessing the texels from the program. It just needs to have a reasonable accuracy, at least 24 bits is common in real-world applications. There are several formats that fit this requirement:

- `VK_FORMAT_D32_SFLOAT`: 32-bit float for depth
- `VK_FORMAT_D32_SFLOAT_S8_UINT`: 32-bit signed float for depth and 8 bit stencil component
- `VK_FORMAT_D24_UNORM_S8_UINT`: 24-bit float for depth and 8 bit stencil component

The stencil component is used for stencil tests, which is an additional test that can be combined with depth testing. We'll look at this in a future chapter.

We could simply go for the `VK_FORMAT_D32_SFLOAT` format, because support for it is extremely common (see the hardware database), but it's nice to add some extra flexibility to our application where possible. We're going to write a function `findSupportedFormat` that takes a list of candidate formats in order from most desirable to least desirable, and checks which is the first one that is supported:

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,

}
```

The support of a format depends on the tiling mode and usage, so we must also include these as parameters. The support of a format can be queried using the `vkGetPhysicalDeviceFormatProperties` function:

```
for (VkFormat format : candidates) {
    VkFormatProperties props;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
}
```

The `VkFormatProperties` struct contains three fields:

- `linearTilingFeatures`: Use cases that are supported with linear tiling
- `optimalTilingFeatures`: Use cases that are supported with optimal tiling
- `bufferFeatures`: Use cases that are supported for buffers

Only the first two are relevant here, and the one we check depends on the `tiling` parameter of the function:

```
if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features)
    return format;
```

```
} else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) ==
    return format;
}
```

If none of the candidate formats support the desired usage, then we can either return a special value or simply throw an exception:

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == f
            return format;
        } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & featu
            return format;
        }
    }

    throw std::runtime_error("failed to find supported format!");
}
```

We'll use this function now to create a `findDepthFormat` helper function to select a format with a depth component that supports usage as depth attachment:

```
VkFormat findDepthFormat() {
    return findSupportedFormat(
        {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
        VK_IMAGE_TILING_OPTIMAL,
        VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
    );
}
```

Make sure to use the `VK_FORMAT_FEATURE_` flag instead of `VK_IMAGE_USAGE_` in this case. All of these candidate formats contain a depth component, but the latter two also contain a stencil component. We won't be using that yet, but we do need to take that into account when performing layout transitions on images with these formats. Add a simple helper function that tells us if the chosen depth format contains a stencil component:

```
bool hasStencilComponent(VkFormat format) {
    return format == VK_FORMAT_D32_SFLOAT_S8_UINT || format == VK_FORMAT_D24_UNORM_S8_UINT;
}
```

Call the function to find a depth format from `createDepthResources`:

```
VkFormat depthFormat = findDepthFormat();
```

We now have all the required information to invoke our `createImage` and `createImageView` helper functions:

```
createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTI
createImageView(depthImage, depthFormat, depthImageView);
```

However, the `createImageView` function currently assumes that the subresource is always the `VK_IMAGE_ASPECT_COLOR_BIT`, so we will need to turn that field into a parameter:

```
void createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags, VDelete
    ...
    viewInfo.subresourceRange.aspectMask = aspectFlags;
    ...
}
```

Update all calls to this function to use the right aspect:

```
createImageView(swapChainImages[i], swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT, swapCha
...
createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT, depthImageView);
...
createImageView(textureImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT, textureIm
```

That's it for creating the depth image. We don't need to map it or copy another image to it, because we're going to clear it at the start of the render pass like the color attachment. However, it still needs to be transitioned to a layout that is suitable for depth attachment usage. We could do this in the render pass like the color attachment, but here I've chosen to use a pipeline barrier because the transition only needs to happen once:

```
transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_DE
```

The undefined layout can be used as initial layout, because there are no existing depth image contents that matter. We need to update some of the logic in `transitionImageLayout` to use the right subresource aspect:

```
if (newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;

    if (hasStencilComponent(format)) {
        barrier.subresourceRange.aspectMask |= VK_IMAGE_ASPECT_STENCIL_BIT;
    }
} else {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
}
```

Although we're not using the stencil component, we do need to include it in the layout transitions of the depth image.

Finally, add the correct access masks:

```
if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_SRC
    barrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
```

```
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_PREINITIALIZED && newLayout == VK_IMAGE_LAYOUT_TRANS
    barrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_DEPTH_STEN
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_ACCESS_DEPTH_ST
} else {
    throw std::invalid_argument("unsupported layout transition!");
}
```

The image is now completely ready for usage as depth attachment.

Render pass

We're now going to modify `createRenderPass` to include a depth attachment. First specify the `VkAttachementDescription`:

```
VkAttachmentDescription depthAttachment = {};
depthAttachment.format = findDepthFormat();
depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

The `format` should be the same as the depth image itself. This time we don't care about storing the depth data (`storeOp`), because it will not be used after drawing has finished. This may allow the hardware to perform additional optimizations. Just like the color buffer, we don't care about the previous depth contents, so we can use `VK_IMAGE_LAYOUT_UNDEFINED` as `initialLayout`.

```
VkAttachmentReference depthAttachmentRef = {};
depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Add a reference to the attachment for the first (and only) subpass:

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;
```

Unlike color attachments, a subpass can only use a single depth (+stencil) attachment. It wouldn't really make any sense to do depth tests on multiple buffers.

```
std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = attachments.size();
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

Finally, update the `VkRenderPassCreateInfo` struct to refer to both attachments.

Framebuffer

The next step is to modify the framebuffer creation to bind the depth image to the depth attachment. Go to `createFramebuffers` and specify the depth image view as second attachment:

```
std::array<VkImageView, 2> attachments = {
    swapChainImageViews[i],
    depthImageView
};

VkFramebufferCreateInfo framebufferInfo = {};
framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
framebufferInfo.renderPass = renderPass;
framebufferInfo.attachmentCount = attachments.size();
framebufferInfo.pAttachments = attachments.data();
framebufferInfo.width = swapChainExtent.width;
framebufferInfo.height = swapChainExtent.height;
framebufferInfo.layers = 1;
```

The color attachment differs for every swap chain image, but the same depth image can be used by all of them because only a single subpass is running at the same time due to our semaphores.

You'll also need to move the call to `createFramebuffers` to make sure that it is called after the depth image view has actually been created:

```
void initVulkan() {
    ...
    createDepthResources();
    createFramebuffers();
```

```
    ...
}
```

Clear values

Because we now have multiple attachments with `VK_ATTACHMENT_LOAD_OP_CLEAR`, we also need to specify multiple clear values. Go to `createCommandBuffers` and create an array of `VkClearValue` structs:

```
std::array<VkClearValue, 2> clearValues = {};
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};
clearValues[1].depthStencil = {1.0f, 0};

renderPassInfo.clearValueCount = clearValues.size();
renderPassInfo.pClearValues = clearValues.data();
```

The range of depths in the depth buffer is `0.0` to `1.0` in Vulkan, where `1.0` lies at the far view plane and `0.0` at the near view plane. The initial value at each point in the depth buffer should be the furthest possible depth, which is `1.0`.

Depth and stencil state

The depth attachment is ready to be used now, but depth testing still needs to be enabled in the graphics pipeline. It is configured through the `VkPipelineDepthStencilStateCreateInfo` struct:

```
VkPipelineDepthStencilStateCreateInfo depthStencil = {};
depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE;
```

The `depthTestEnable` field specifies if the depth of new fragments should be compared to the depth buffer to see if they should be discarded. The `depthWriteEnable` field specifies if the new depth of fragments that pass the depth test should actually be written to the depth buffer. This is useful for drawing transparent objects. They should be compared to the previously rendered opaque objects, but not cause further away transparent objects to not be drawn.

```
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

The `depthCompareOp` field specifies the comparison that is performed to keep or discard fragments. We're sticking to the convention of lower depth = closer, so the depth of new fragments should be *less*.

```
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.minDepthBounds = 0.0f; // Optional
depthStencil.maxDepthBounds = 1.0f; // Optional
```

The `depthBoundsTestEnable`, `minDepthBounds` and `maxDepthBounds` fields are used for the optional depth bound test. Basically, this allows you to only keep fragments that fall within the specified depth range. We won't be using this functionality.

```
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optional
depthStencil.back = {}; // Optional
```

The last three fields configure stencil buffer operations, which we also won't be using in this tutorial. If you want to use these operations, then you will have to make sure that the format of the depth/stencil image contains a stencil component.

```
pipelineInfo.pDepthStencilState = &depthStencil;
```

Update the `VkGraphicsPipelineCreateInfo` struct to reference the depth stencil state we just filled in. A depth stencil state must always be specified if the render pass contains a depth stencil attachment.

If you run your program now, then you should see that the fragments of the geometry are now correctly ordered:

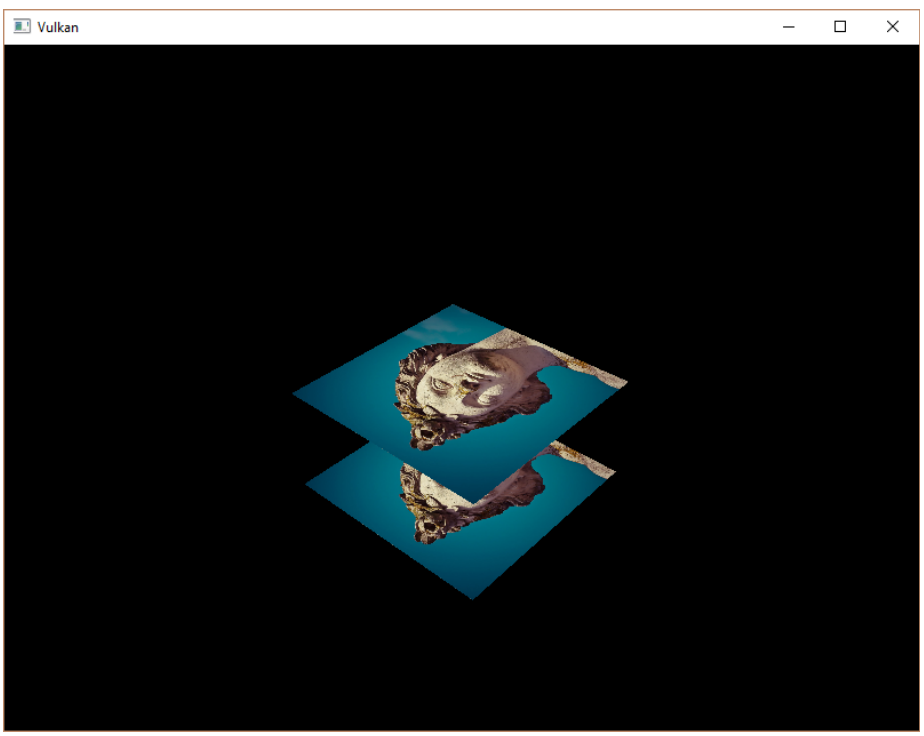

Figure 47:

Handling window resize

The resolution of the depth buffer should change when the window is resized to match the new color attachment resolution. Extend the `recreateSwapChain` function to recreate the depth resources in that case:

```
void recreateSwapChain() {
    vkDeviceWaitIdle(device);

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createDepthResources();
    createFramebuffers();
    createCommandBuffers();
}
```

Congratulations, your application is now finally ready to render arbitrary 3D geometry and have it look right. We're going to try this out in the next chapter by drawing a textured model!

C++ code / Vertex shader / Fragment shader

Loading models

Introduction

Your program is now ready to render textured 3D meshes, but the current geometry in the `vertices` and `indices` arrays is not very interesting yet. In this chapter we're going to extend the program to load the vertices and indices from an actual model file to make the graphics card actually do some work.

Many graphics API tutorials have the reader write their own OBJ loader in a chapter like this. The problem with this is that any remotely interesting 3D application will soon require features that are not supported by this file format, like skeletal animation. We *will* load mesh data from an OBJ model in this chapter, but we'll focus more on integrating the mesh data with the program itself rather than the details of loading it from a file.

Library

We will use the tinyobjloader library to load vertices and faces from an OBJ file. It's fast and it's easy to integrate because it's a single file library like stb_image.

Go to the repository linked above and download the `tiny_obj_loader.h` file to a folder in your library directory.

Visual Studio

Add the directory with `tiny_obj_loader.h` in it to the `Additional Include Directories` paths.

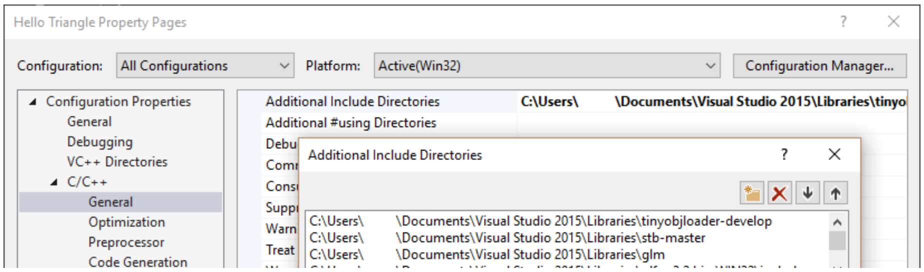

Figure 48:

Makefile

Add the directory with `tiny_obj_loader.h` to the include directories for GCC:

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb
TINYOBJ_INCLUDE_PATH = /home/user/libraries/tinyobjloader

...

CFLAGS = -std=c++11 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH) -I$(TINYOBJ_INCLUDE_
```

Sample mesh

In this chapter we won't be enabling lighting yet, so it helps to use a sample model that has lighting baked into the texture. An easy way to find such models is to look for 3D scans on Sketchfab. Many of the models on that site are available in OBJ format with a permissive license.

For this tutorial I've decided to go with the Chalet Hippolyte Chassande Baroz model by Escadrone. I tweaked the size and orientation of the model to use it as a drop in replacement for the current geometry:

- chalet.obj
- chalet.jpg

It has half a million triangles, so it's a nice benchmark for our application. Feel free to use your own model, but make sure that it only consists of one material and that is has dimensions of about 1.5 x 1.5 x 1.5 units. If it is larger than that, then you'll have to change the view matrix. Put the model file in a new `models` directory next to `shaders` and `textures`, and put the texture image in the `textures` directory.

Put two new configuration variables in your program to define the model and texture paths:

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

const std::string MODEL_PATH = "models/chalet.obj";
const std::string TEXTURE_PATH = "textures/chalet.jpg";
```

And update `createTextureImage` to use this path variable:

```
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels, STBI_
```

Loading vertices and indices

We're going to load the vertices and indices from the model file now, so you should remove the global `vertices` and `indices` arrays now. Replace them with non-const containers as class members:

```
std::vector<Vertex> vertices;
std::vector<uint32_t> indices;
VDeleter<VkBuffer> vertexBuffer{device, vkDestroyBuffer};
VDeleter<VkDeviceMemory> vertexBufferMemory{device, vkFreeMemory};
```

You should change the type of the indices from `uint16_t` to `uint32_t`, because there are going to be a lot more vertices than 65535. Remember to also change the `vkCmdBindIndexBuffer` parameter:

```
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
```

The tinyobjloader library is included in the same way as STB libraries. Include the `tiny_obj_loader.h` file and make sure to define `TINYOBJLOADER_IMPLEMENTATION` in one source file to include the function bodies and avoid linker errors:

```
#define TINYOBJLOADER_IMPLEMENTATION
#include <tiny_obj_loader.h>
```

We're now going to write a `loadModel` function that uses this library to populate the `vertices` and `indices` containers with the vertex data from the mesh. It should be called somewhere before the vertex and index buffers are created:

```
void initVulkan() {
    ...
    loadModel();
    createVertexBuffer();
```

```
    createIndexBuffer();
    ...
}

...

void loadModel() {

}
```

A model is loaded into the library's data structures by calling the `tinyobj::LoadObj` function:

```
void loadModel() {
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    std::string err;

    if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &err, MODEL_PATH.c_str())) {
        throw std::runtime_error(err);
    }
}
```

An OBJ file consists of positions, normals, texture coordinates and faces. Faces consist of an arbitrary amount of vertices, where each vertex refers to a position, normal and/or texture coordinate by index. This makes it possible to not just reuse entire vertices, but also individual attributes.

The `attrib` container holds all of the positions, normals and texture coordinates in its `attrib.vertices`, `attrib.normals` and `attrib.texcoords` vectors. The `shapes` container contains all of the separate objects and their faces. Each face consists of an array of vertices, and each vertex contains the indices of the position, normal and texture coordinate attributes. OBJ models can also define a material and texture per face, but we will be ignoring those.

The `err` string contains errors and warnings that occurred while loading the file, like a missing material definition. Loading only really failed if the `LoadObj` function returns `false`. As mentioned above, faces in OBJ files can actually contain an arbitrary number of vertices, whereas our application can only render triangles. Luckily the `LoadObj` has an optional parameter to automatically triangulate such faces, which is enabled by default.

We're going to combine all of the faces in the file into a single model, so just iterate over all of the shapes:

```
for (const auto& shape : shapes) {

}
```

The triangulation feature has already made sure that there are three vertices per face, so we can now directly iterate over the vertices and dump them straight into our `vertices` vector:

```
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex = {};

        vertices.push_back(vertex);
        indices.push_back(indices.size());
    }
}
```

For simplicity, we will assume that every vertex is unique for now, hence the simple auto-increment indices. The `index` variable is of type `tinyobj::index_t`, which contains the `vertex_index`, `normal_index` and `texcoord_index` members. We need to use these indices to look up the actual vertex attributes in the `attrib` arrays:

```
vertex.pos = {
    attrib.vertices[3 * index.vertex_index + 0],
    attrib.vertices[3 * index.vertex_index + 1],
    attrib.vertices[3 * index.vertex_index + 2]
};

vertex.texCoord = {
    attrib.texcoords[2 * index.texcoord_index + 0],
    attrib.texcoords[2 * index.texcoord_index + 1]
};

vertex.color = {1.0f, 1.0f, 1.0f};
```

Unfortunately the `attrib.vertices` array is an array of `float` values instead of something like `glm::vec3`, so you need to multiply the index by `3`. Similarly, there are two texture coordinate components per entry. The offsets of `0`, `1` and `2` are used to access the X, Y and Z components, or the U and V components in the case of texture coordinates.

Run your program now with optimization enabled (e.g. `Release` mode in Visual Studio and with the `-O3` compiler flag for GCC'). This is necessary, because otherwise loading the model will be very slow. You should see something like the following:

Great, the geometry looks correct, but what's going on with the texture? The problem is that the origin of texture coordinates in Vulkan is the top-left corner, whereas the OBJ format assumes the bottom-left corner. Solve this by flipping the vertical component of the texture coordinates:

```
vertex.texCoord = {
```

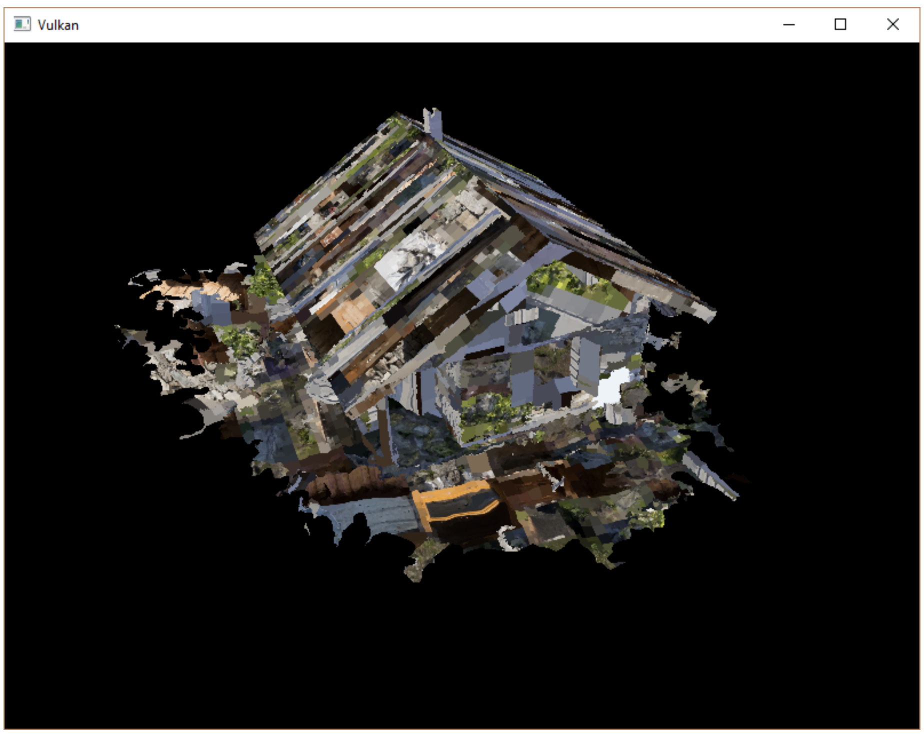

Figure 49:

```
    attrib.texcoords[2 * index.texcoord_index + 0],
    1.0f - attrib.texcoords[2 * index.texcoord_index + 1]
};
```

When you run your program again, you should now see the correct result:

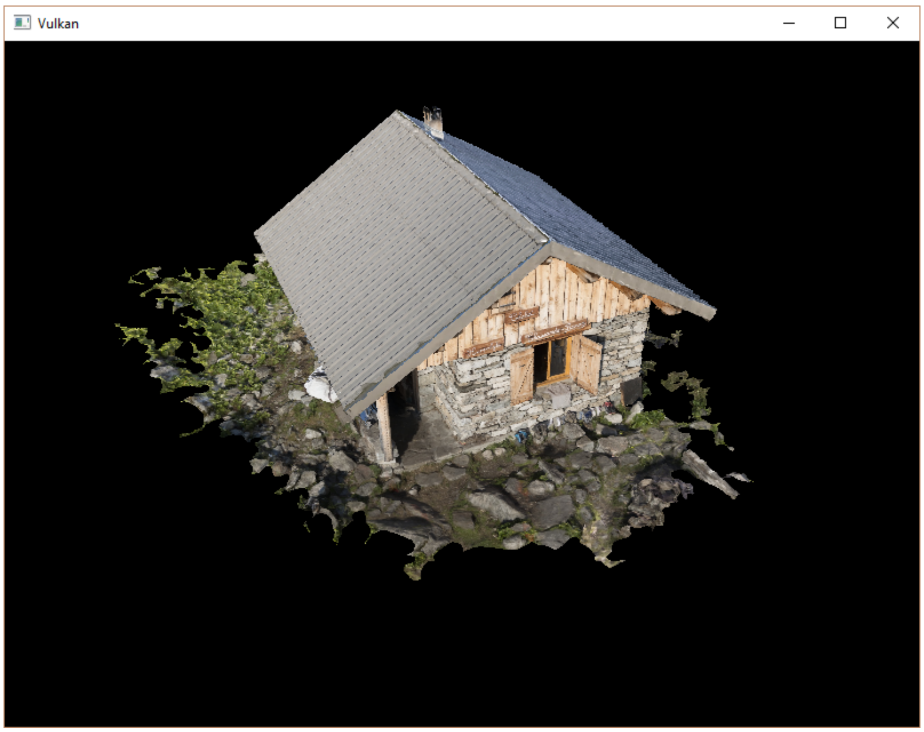

Figure 50:

All that hard work is finally beginning to pay off with a demo like this!

Vertex deduplication

Unfortunately we're not really taking advantage of the index buffer yet. The `vertices` vector contains a lot of duplicated vertex data, because many vertices are included in multiple triangles. We should keep only the unique vertices and use the index buffer to reuse them whenever they come up. A straightforward way to implement this is to use a `map` or `unordered_map` to keep track of the unique vertices and their index:

```
#include <unordered_map>

...
```

```
std::unordered_map<Vertex, int> uniqueVertices = {};

for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex = {};

        ...

        if (uniqueVertices.count(vertex) == 0) {
            uniqueVertices[vertex] = vertices.size();
            vertices.push_back(vertex);
        }

        indices.push_back(uniqueVertices[vertex]);
    }
}
```

Every time we read a vertex from the OBJ file, we check if we've already seen a vertex with the exact same position and texture coordinates before. If not, we add it to `vertices` and store its index in the `uniqueVertices` container. After that we add the index of the new vertex to `indices`. If we've seen the exact same vertex before, then we look up its index in `uniqueVertices` and store that index in `indices`.

The program will fail to compile right now, because using a user-defined type like our `Vertex` struct as key in a hash table requires us to implement two functions: equality test and hash calculation. The former is easy to implement by overriding the `==` operator in the `Vertex` struct:

```
bool operator==(const Vertex& other) const {
    return pos == other.pos && color == other.color && texCoord == other.texCoord;
}
```

A hash function for `Vertex` is implemented by specifying a template specialization for `std::hash<T>`. Hash functions are a complex topic, but cppreference.com recommends the following approach combining the fields of a struct to create a decent quality hash function:

```
namespace std {
    template<> struct hash<Vertex> {
        size_t operator()(Vertex const& vertex) const {
            return ((hash<glm::vec3>()(vertex.pos) ^
                   (hash<glm::vec3>()(vertex.color) << 1)) >> 1) ^
                   (hash<glm::vec2>()(vertex.texCoord) << 1);
        }
    };
}
```

This code should be placed outside the `Vertex` struct. The hash functions for the GLM types need to be included using the following header:

```
#include <glm/gtx/hash.hpp>
```

You should now be able to successfully compile and run your program. If you check the size of `vertices`, then you'll see that it has shrunk down from 1,500,000 to 265,645! That means that each vertex is reused in an average number of ~6 triangles. This definitely saves us a lot of GPU memory.

Conclusion

It has taken a lot of work to get to this point, but now you finally have a good base for a Vulkan program. The knowledge of the basic principles of Vulkan that you now possess should be sufficient to start exploring more of the features, like:

- Push constants
- Instanced rendering
- Dynamic uniforms
- Separate images and sampler descriptors
- Pipeline cache
- Multi-threaded command buffer generation
- Multiple subpasses

The current program can be extended in many ways, like adding Blinn-Phong lighting, post-processing effects and shadow mapping. You should be able to learn how these effects work from tutorials for other APIs, because despite Vulkan's explicitness, many concepts still work the same.

C++ code / Vertex shader / Fragment shader