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.

Some platforms allow you to render directly to a display without interacting with any window manager through the `VK_KHR_display` and `VK_KHR_display_swapchain` extensions. These allow you to create a surface that represents the entire screen and could be used to implement your own window manager, for example.

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 set 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:

```
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 al-

lows 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.
