Università degli Studi di Torino
Dipartimento di Informatica

Corso di Laurea in Informatica
Anno Accademico 2020/2021

Exploring The Vulkan Graphics API

Relatore
Prof. Maurizio Lucenteforte

Candidato
Emanuele Franchi

Dichiarazione Di Originalità

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Abstract

The aim of this work is to explore the Vulkan graphics API. We do this by studying how the API works and using it to produce a working application. Each chapter tries to focus on one single Vulkan concept producing a demo that shows how it works in practice.

In chapter 1 we introduce the history behind Vulkan. This lets us understand the reasons that guided the design of the API. In chapter 2 we explain all the work that is almost always necessary to get a Vulkan application up and running. In chapter 3 we finally do the simplest form of rendering: we clear the window with a flat color. This may seem trivial, but it's an important stepping stone. Doing this, we can see all the concepts that come into play when drawing something on the screen. In chapter 4 we finally write the computer graphics hello world program: rendering a triangle. In chapter 5 and in chapter 6 we see how to send data from the CPU to the GPU, making our application more flexible. In chapter 7 we see how to solve a common problem in computer graphics: how to determine the order in which to draw objects, so that closer objects cover far away ones. In chapter 8 we take a break from Vulkan and explain a very simple way to describe objects we want to draw, and how to place them inside a virtual world. In chapter 9 we use all the concepts we have learned so far to implement a lighting model. In chapter 10 we improve the visual quality of our application enabling a Vulkan feature called multisample anti aliasing.

Ringraziamenti

Innanzitutto, desidero ringraziare i miei genitori, che mi hanno dato la possibilità di studiare e di seguire la mia passione per l'informatica. Ringrazio la mia ragazza, che mi è sempre stata accanto, portandomi conforto nei momenti più amari, e condividendo la gioia dei momenti più dolci. Desidero ringraziare il mio relatore, il professore Maurizio Lucenteforte, che mi ha aiutato durante tutto il percorso, offrendomi il suo consiglio. Infine, ringrazio tutti i professori che ho incontrato durante il mio percorso accademico per avermi fatto interessare sempre di più all'informatica.

Contents

List of Figures

Listings

Chapter 1

Journey Towards Vulkan

1.1 Software Rendering

In the early days of computer graphics, if you wanted to draw an image on the screen, you had to directly instruct the CPU to do so. For example, drawing a line segment would require to run a loop and set the color of each pixel lying along the line. This is called software rendering. Because software rendering required a lot of CPU time, graphics performance was very slow.

1.2 GPUs

In 1981, Jim Clark, a professor at Stanford, had the idea to build ad hoc hardware for making graphics operations faster. This hardware is what we today call a graphics processing unit, or GPU, for short. The massive increase in graphics performance is given by two factors. GPUs have several specialized processors that can work in parallel. GPUs also have their own dedicated memory. GPU processors have very fast access to this memory, much faster than their access time on RAM.

1.3 Graphics APIs

Now that computers had specialized graphics hardware, programmers needed a way to interact and use said hardware effectively. To simplify this process, each graphics card manufacturer also developed a graphics API to directly interact with their custom hardware.

Drawing using a graphics API is much simpler than using software rendering. We simply need to instruct the CPU to send the appropriate commands and data to the GPU. The GPU will then be responsible for executing the commands. In this way, the CPU offloads most of the work to the GPU, which is optimized to perform graphics commands very quickly. For example, if we want to draw a line segment, we simply need to send to the GPU the points that define the line segment itself, and then send a command that tells the GPU to draw it.

1.4 OpenGL

Using a graphics API was very convenient for programmers. The problem was that each graphics card manufacturer had their own custom graphics API. Thus, if you wanted to port your software to other platforms, you had to rewrite it using another graphics API. This was obviously a nuisance to many people. Adding to this problem was the fact that different graphics APIs could have entirely different ideas on how to do graphics, making porting software even more difficult.

At that time, the leading graphics card manufacturer was SGI. The graphics API that was used to interact with SGI hardware was called GL. With time, other companies realized that GL was a very good graphics API, and wanted to let programmers work that way. So, in 1992, the most prominent graphics card manufacturers and other companies banded together to form a committee and created the OpenGL specification.

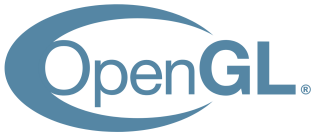

Figure 1.1: OpenGL logo

OpenGL is based on the fact that software vendors have to provide their own implementation that conforms to the OpenGL standard. On the other hand, graphics card manufacturers have to provide programs that allow OpenGL to talk to the underlying graphics hardware, what today we call device drivers.

1.5 OpenGL Issues

With time, graphics hardware continued to evolve, and graphics cards started to offer new functionalities, becoming more and more programmable. To access these new GPU features, OpenGL had to be extended, adding new concepts to the API, while still maintaining the older functionalities. This lead to a growth in the API's complexity, which had to be shouldered by the device drivers, making them more bloated. Because of this, device drivers became inefficient and also riddled with bugs caused by many inconsistencies between different OpenGL implementations.

GPUs were not the only thing to change. CPUs also continued to evolve. In particular, CPUs started to have more than one core, offering the possibility of multithreading. The problem is that OpenGL wasn't meant to be used in a multithreaded context, being a strictly synchronous API. This obviously can be a big bottleneck in terms of performance.

1.6 Vulkan

Figure 1.2: Vulkan logo

Vulkan is OpenGL's spiritual successor. It is a newer graphics API that is meant to abstract how modern GPUs work. Vulkan doesn't suffer from the problems that plague OpenGL because it has been designed from scratch and with modern GPU's architecture in mind. Vulkan reduces the driver overhead by being more low level, allowing us to write more performant code. It is also designed to be easy to use in a multithreaded environment, allowing us to submit GPU commands from multiple threads.

1.7 Vulkan, OpenGL And Alternatives

Now that we have both discussed OpenGL and Vulkan, we can see that both of them have their own pros and cons.

Vulkan is a lower level API compared to OpenGL. This means that Vulkan exposes more complexity to the programmer. Vulkan does this because it wants to allow programmers to write code that better suits their performance needs. This could also be considered a drawback, since Vulkan is more verbose and also requires a lot more code to get the same results compared to OpenGL.

Being low level doesn't always mean being faster. With OpenGL, a lot of thing were taken care by the device driver, written by people that know how GPUs work. With Vulkan, the programmer has to bear an even greater responsibility to sensibly use the API in order to not tank the application's performance.

Another thing to note is the fact that OpenGL is older than Vulkan. This means that not all hardware that is still around today supports Vulkan. If we want to run a graphics application on older hardware, we must fall back to OpenGL.

Both OpenGL and Vulkan can be considered quite low level APIs nowadays. In fact, many people don't directly use them. For example, it's common to use simpler libraries that abstract over these APIs to provide a more user friendly experience for programmers. In this way, people can leverage the performance improvements offered by Vulkan, or the ubiquitous portability provided by OpenGL, without having to directly deal with the complexity that comes using these APIs.

Chapter 2

Initializing Vulkan

In this chapter we go through all the necessary steps to initialize a Vulkan application. We first create a Vulkan instance. Then, we create a window and link it to our instance creating a presentation surface. We determine what GPU will be used by our application and create a logical device to interface with it. Finally, we create a swapchain in order to interface with the presentation engine of our operating system.

2.1 Create Vulkan Instance

To access any of the functionalities offered by Vulkan we first have to create a Vulkan instance. To do this we call `vkCreateInstance`.

```
1  VkInstance instance = VK_NULL_HANDLE;
2  vkCreateInstance(&createInfo, nullptr, instance);
```

Listing 2.1: Create Vulkan instance

2.1.1 VkInstanceCreateInfo

We use a `VkInstanceCreateInfo` struct to configure the Vulkan instance we are about to create.

```
1  VkInstanceCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
3  createInfo.pApplicationInfo = &appInfo;
4  createInfo.enabledLayerCount = layerCount;
5  createInfo.ppEnabledLayerNames = layers;
6  createInfo.enabledExtensionCount = extensionCount;
7  createInfo.ppEnabledExtensionNames = extentions;
```

Listing 2.2: VkInstanceCreateInfo initialization

2.1.2 VkApplicationInfo

We can see that the `VkInstanceCreateInfo` struct is not the only thing we need. We have to specify a pointer to a `VkApplicationInfo` struct. Such struct describes our Vulkan application.

```
1  VkApplicationInfo appInfo = {};
2  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
3  appInfo.pApplicationName = "Vulkan example";
4  appInfo.apiVersion = VK_API_VERSION_1_2;
```

Listing 2.3: VkApplicationInfo initialization

2.1.3 Layers

While we initialize our `VkInstanceCreateInfo` struct, we can specify the layers that we want to enable. The specified layers will be loaded after the Vulkan instance creation.

Layers are optional components that hook into Vulkan. Layers can intercept, evaluate and modify existing Vulkan functions. Layers are implemented as libraries and are loaded during instance creation.

If we want to enable error checking, we need to load a layer that provides such functionality. This kind of layer is know as validation layer. Since validation layers cause overhead, we can disable them when we build the application in release mode.

```
1  const char* const layers[] =
2  {
3      #ifdef _DEBUG
4      "VK_LAYER_KHRONOS_validation",
5      #endif
6      // other layers ...
7  };
```

Listing 2.4: Enabling the Khronos validation layer

Checking whether our layers are supported

Before creating our Vulkan instance, we should check if the layers we require are actually supported. To do this we use `vkEnumerateInstanceLayerProperties`. This function returns all the layers supported by our Vulkan installation. If all the layers we require are present, then we can proceed to create our Vulkan instance.

2.1.4 Extensions

While we initialize our `VkInstanceCreateInfo` struct, we can specify the instance extensions that we want to enable. The specified instance extensions will be loaded after creating our Vulkan instance.

Extensions are additional features that Vulkan implementations may provide. Extensions add new functions and structs to the API. Extensions may also change some of the behavior of existing functions. We can either enable extensions at an instance level or at a device level.

Here, we use an extension to provide a callback to handle the debug messages generated by the validation layers.

```
const char* const* extensions[] =
{
    #ifdef _DEBUG
    VK_EXT_DEBUG_UTILS_EXTENSION_NAME,
    #endif
    // Other extensions ...
};
```

Listing 2.5: Enabling an extention to handle validation layer debug messages

We specify one callback that handles messages generated by instance creation and destruction. We specify another callback that handles all other API debug messages.

```
#ifdef _DEBUG
VkDebugUtilsMessengerCreateInfoEXT dbgInfo = {};
dbgInfo.sType =
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
dbgInfo.messageSeverity = severity;
dbgInfo.messageType = type;
dbgInfo.pfnUserCallback = VulkanDebugCallback;
#endif

VkInstanceCreateInfo createInfo = {};
#ifdef _DEBUG
createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*)(dbgInfo);
#endif

// ... after instance creation

// Enabling debug callback for all other API functions
#ifdef _DEBUG
VkDebugUtilsMessengerEXT debugMessenger = VK_NULL_HANDLE;
CreateDebugUtilsMessengerEXT(instance, &dbgInfo, nullptr, &
    debugMessenger)
#endif
```

Listing 2.6: Setting up debug extension callbacks

The function that creates the `VkDebugUtilsMessengerEXT` object comes from the extension we have enabled. Because of this, we have to load it manually into our address space using `vkGetInstanceProcAddr`. An elegant way to solve this issue is to create a proxy function that handles this matter for us.

```
static VkResult CreateDebugUtilsMessengerEXT
(
    VkInstance instance,
    const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDebugUtilsMessengerEXT* pDebugMessenger
)
{
    PFN_vkCreateDebugUtilsMessengerEXT f = (
    PFN_vkCreateDebugUtilsMessengerEXT)(vkGetInstanceProcAddr(
    instance, "vkCreateDebugUtilsMessengerEXT"));
    return f(instance, pCreateInfo, pAllocator, pDebugMessenger);
}
```

Listing 2.7: Extension function proxy

Checking whether our extensions are supported

Before creating our Vulkan instance, we should check if the instance extensions we require are actually supported. To do this we use `vkEnumerateInstanceExtensionProperties`. This function returns all the instance extensions that are supported by our Vulkan installation. If all the instance extensions we require are present, then we can proceed to create our Vulkan instance.

2.1.5 Vulkan Instance Cleanup

To destroy our debug messenger we use `vkDestroyDebugUtilsMessengerEXT`. This function must be manually loaded using `vkGetInstanceProcAddr`. To destroy our Vulkan instance we use `vkDestroyInstance`.

2.2 Open A Window

After creating our Vulkan instance we open a window. To do this we have two options. We can use a cross platform library (SDL, GLFW) that will do all the heavy lifting for us, so that we don't have to worry about directly interacting with the OS, freeing us from the burden of knowing how the windowing API works. We can also decide to not use a library and opening the window ourselves. We will do the latter, since it's interesting to know how things work under the hood. Since I'm on Windows, I'll be dealing with the Win32 API. We won't go in depth about the specifics of this API since it's beyond our scope.

2.2.1 Create Window Handle

To create a handle to a window we use `CreateWindowEx`. We use `windowStyle` and `windowExtendedStyle` variables to configure the look of our window.

```
1  DWORD windowStyle = (WS_OVERLAPPEDWINDOW | WS_VISIBLE | WS_CAPTION)
       & (~WS_THICKFRAME) & (~WS_MINIMIZEBOX) & (~WS_MAXIMIZEBOX);
2  DWORD windowExtendedStyle = 0;
3
4  HWND handle = CreateWindowEx(
5      windowExtendedStyle,
6      WINDOW_CLASS_NAME,
7      name,
8      windowStyle,
9      CW_USEDEFAULT, CW_USEDEFAULT,
10     windowWidth, windowHeight,
11     0,
12     0,
13     GetModuleHandle(0),
14     0
15 );
```

Listing 2.8: Creating a window handle using Win32 API

2.2.2 Computing Window Dimensions

Before creating our window, we need to compute its width and height. This is due to the fact that a window comprises of a client area and a non client area.

We usually want our client area to be of a certain size, but `CreateWindowEx` takes the whole window width and the whole window height as arguments.

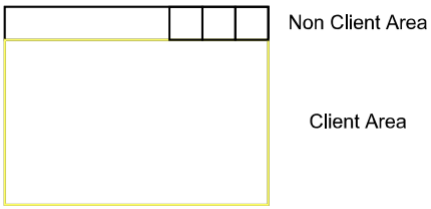

Figure 2.1: Anatomy of a Win32 Window

```
1  RECT windowDimensions = { 0, 0, clientWidth, clientHeight };
2  AdjustWindowRectEx(&windowDimensions, windowStyle, false,
       windowExtendedStyle);
3  i32 windowWidth = windowDimensions.right - windowDimensions.left;
4  i32 windowHeight = windowDimensions.bottom - windowDimensions.top;
```

Listing 2.9: Compute window width and height

2.2.3 Register Window Class

Before creating our window, we need to register its window class. To do this we use `RegisterClassEx`. This function takes a pointer to a `WNDCLASSEX` struct. This struct is used to configure our window class.

```
1  WNDCLASSEX windowClass = {};
2  windowClass.cbSize = sizeof(windowClass);
3  windowClass.style = CS_HREDRAW | CS_VREDRAW;
4  windowClass.lpfnWndProc = WindowProcedure;
5  windowClass.hInstance = GetModuleHandle(0);
6  windowClass.hIcon = LoadIcon(0, IDI_APPLICATION);
7  windowClass.hCursor = LoadCursor(0, IDC_ARROW);
8  windowClass.lpszClassName = WINDOW_CLASS_NAME;
9  windowClass.hIconSm = LoadIcon(0, IDI_APPLICATION);
10
11 RegisterClassEx(&windowClass);
```

Listing 2.10: Register Window Class

2.2.4 Window Procedure

While filling in our `WNDCLASSEX` struct, we have to pass a window procedure. This is a callback function used internally by the windowing API. We use this function to handle the events that our window will receive during the lifespan of our application. The Win32 API also provides a default window procedure. Our custom window procedure will call this default procedure when we don't want to handle particular events ourselves. When we receive a quit, close or destroy message we enqueue a quit message into our message queue.

```
1  static LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
       wparam, LPARAM lparam)
2  {
3      LRESULT result = 0;
4      switch (msg)
5      {
6      case WM_QUIT:
7      case WM_CLOSE:
8      case WM_DESTROY: { PostQuitMessage(0); } break;
9      default: { result = DefWindowProcA(hwnd, msg, wparam, lparam);
       } break;
10     };
11
12     return result;
13 }
```

Listing 2.11: Window Procedure

2.2.5 Process Window Messages

In order for the user to be able to interact with our window, we need to handle the window messages that are dispatched by the OS towards our window. All these messages come from the application's message queue.

```
1  MSG message = {};
2  while (PeekMessage(&message, 0, 0, 0, PM_REMOVE))
3  {
4      switch (message.message)
5      {
6      case WM_QUIT:
7      {
8          isApplicationRunning = false;
9      } break;
10
11     default:
12     {
13         TranslateMessage(&message);
14         DispatchMessageA(&message);
15     } break;
16     }
17 }
```

Listing 2.12: Process Window Messages

Here we iterate over all the window messages that we haven't handled. If we find a quit message, then we exit our application. All other messages will be dispatched to our window procedure.

2.2.6 Window Cleanup

When our application is shutting down, we destroy our window and unregister its class.

```
1  DestroyWindow(handle);
2  UnregisterClass(WINDOW_CLASS_NAME, GetModuleHandle(0));
```

Listing 2.13: Window Cleanup

2.3 Create A Presentation Surface

We must link the newly created window to our Vulkan instance. To do this we create a presentation (or window) surface. This operation is platform specific. Since we are using Windows, in order to create our presentation surface we use `vkCreateWin32SurfaceKHR`.

```
1  VkSurfaceKHR surface = VK_NULL_HANDLE;
2  vkCreateWin32SurfaceKHR(instance, &createInfo, nullptr, &surface);
```

Listing 2.14: Create Presentation Surface

2.3.1 VkWin32SurfaceCreateInfoKHR

We use a `VkWin32SurfaceCreateInfoKHR` struct to configure the presentation surface we are about to create.

```
1  VkWin32SurfaceCreateInfoKHR createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
3  createInfo.hinstance = GetModuleHandleA(0);
4  createInfo.hwnd = handle;
```

Listing 2.15: Filling in a VkWin32SurfaceCreateInfoKHR struct

2.3.2 Required Instance Extensions

Vulkan, being cross platform, cannot interact directly with the OS windowing system. To do this we use two extensions.

The first extension that we enable is the instance level KHR surface extension. This extension exposes a `VkSurfaceKHR` object that represents a surface to present rendered images to. This surface will be backed by the window we have created.

The second extension we enable is platform specific and is needed to create our `VkSurfaceKHR` object. In our case, since we are using Windows, we enable the instance level KHR win32 surface extension.

```
1  #define VK_USE_PLATFORM_WIN32_KHR
2  #include "Vulkan.h"
3
4  const char* const* extensions[] =
5  {
6      VK_KHR_SURFACE_EXTENSION_NAME,
7      VK_KHR_WIN32_SURFACE_EXTENSION_NAME,
8      // ... other extensions
9  }
```

Listing 2.16: Presentation Surface Extensions

Notice the `#define` preprocessor directive right before including our Vulkan header. We do this to access our native platform functions.

2.3.3 Presentation Surface Cleanup

To destroy our presentation surface we use `vkDestroySurfaceKHR`.

2.4 Pick A Physical Device

Now that we have a Vulkan instance and a presentation surface, we select a physical device (a GPU) that supports the features we need. The selected GPU will be the one that will be used by our application.

2.4.1 Listing Available Physical Devices

We first get a list of all the physical devices that are available on the system. To do this we use `vkEnumeratePhysicalDevices`. These physical devices can either be integrated or dedicated GPUs.

2.4.2 Finding A Suitable Physical Device

Now that we have a list of all the physical devices, we can select one of them. We could, for example, automatically pick the first one without doing any kind of checking. This approach is doable if we don't have any particular requirement for our physical devices.

Usually we have a set of specific physical device features that are mandatory for our application to run. Hence, in our list, some physical devices will be suitable for our application, while others won't.

The approach we take here is to iterate through the list of all physical devices and pick the first one that is suitable for our application. One question still remains: how can we tell whether a physical device is suitable or not?

Support Grpahics Operations

To check if our physical device supports graphics operations we list all the queue families of our physical device. To do this we use `vkGetPhysicalDeviceQueueFamilyProperties`. Then we check if at least one queue family supports graphics operations.

```
1  for (u32 i = 0; i < queueFamilyCount; i++)
2  {
3      VkQueueFamilyProperties queueFamily = queueFamilies[i];
4      if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT)
5      {
6          // graphics operations supported and i is the index
7          // of a queue family that supports such operations
8      }
9  }
```

Listing 2.17: Check for graphics operations support

Support Present Operations

To check if our physical device supports present operations we list all the queue families of our physical device. To do this we use `vkGetPhysicalDeviceQueueFamilyProperties`. Then we check if at least one queue family supports present operations.

```
for (u32 i = 0; i < queueFamilyCount; i++)
{
    VkBool32 presentSupport = false;
    vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface
    , &presentSupport);
    if (presentSupport)
    {
        // present operations are supported and i is the index
        // of a queue family that supports such operations
    }
}
```

Listing 2.18: Check for present operations support

Support Presentation To A Surface

Not only our physical device must support present operations. It must also be able to present images to the screen. Image presentation is tied to the window and consequently to the surface associated with it. For this reason, image presentation to the screen is not part of Vulkan. We have to enable the KHR swapchain device extension to support such operation. We need this particular extension because image presentation to a surface is achieved using a swapchain.

```
const char* const* deviceExtensions[] =
{
    VK_KHR_SWAPCHAIN_EXTENSION_NAME,
    // ... other device extensions
};
```

Listing 2.19: Device extension for image presentation to the screen

As we have seen earlier, before enabling an extension, we should check for its support. To check whether our physical device supports one or more device extensions we use `vkEnumerateDeviceExtensionProperties`. This function returns a list of all the extensions supported by our physical device. Then, we simply check whether all the extensions we require are present in the list.

Support A Present Mode

Checking if a swapchain is supported is not sufficient. Even if it's supported, it may not be compatible with our presentation surface. We need to check whether our physical device supports at least one present mode for our presentation surface. We can do this using `vkGetPhysicalDeviceSurfacePresentModesKHR`. This functions returns a list of present modes supported by our physical device that are compatible with our presentation surface. If there is at least one present mode in the list, then we are good to go.

2.5 Create A Logical Device

To interact with the physical device we have selected we need to create a logical device.

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
u32 graphicsQueueFamilyIndex;
u32 presentQueueFamilyIndex;

// ... selecting physical device

VkDevice device = VK_NULL_HANDLE;
vkCreateDevice(physicalDevice, &createInfo, nullptr, &device)
```

Listing 2.20: Create a logical device

2.5.1 VkDeviceCreateInfo

We use a `VkDeviceCreateInfo` struct to configure the device we are about to create.

During physical device picking, we saved two queue family indices. The first for a queue family that supports graphics operations. The second for a queue family that supports present operations. The way we populate our `VkDeviceCreateInfo` struct is different based on whether these two indices are equal or not. If our graphics and present queue families are the same, we tell our device that we want to create a single queue. Otherwise, we tell our device that we want to create two queues, one from our graphics queue family, and the other from our present queue family.

```
// Specify requested device features here
VkPhysicalDeviceFeatures deviceFeatures = {};
// We don't use priority queues
f32 queuePriority = 1.0f;

VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex;
queueCreateInfo.queueCount = 1;
queueCreateInfo.pQueuePriorities = &queuePriority;

VkDeviceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.queueCreateInfoCount = 1;
createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.enabledExtensionCount = deviceExtensionCount;
createInfo.ppEnabledExtensionNames = deviceExtensions;
createInfo.pEnabledFeatures = &deviceFeatures;
```

Listing 2.21: Create info struct when queue families are the same

```
1  // Specify requested device features here
2  VkPhysicalDeviceFeatures deviceFeatures = {};
3  // We don't use priority queues
4  f32 queuePriority = 1.0f;
5
6  VkDeviceQueueCreateInfo graphicsQueueCreateInfo = {};
7  graphicsQueueCreateInfo.sType =
       VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
8  graphicsQueueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex
       ;
9  graphicsQueueCreateInfo.queueCount = 1;
10 graphicsQueueCreateInfo.pQueuePriorities = &queuePriority;
11
12 VkDeviceQueueCreateInfo presentQueueCreateInfo = {};
13 presentQueueCreateInfo.sType =
       VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
14 presentQueueCreateInfo.queueFamilyIndex = presentQueueFamilyIndex;
15 presentQueueCreateInfo.queueCount = 1;
16 presentQueueCreateInfo.pQueuePriorities = &queuePriority;
17
18 VkDeviceQueueCreateInfo queueCreateInfos[] =
19 {
20     graphicsQueueCreateInfo,
21     presentQueueCreateInfo,
22 };
23
24 VkDeviceCreateInfo createInfo = {};
25 deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
26 createInfo.queueCreateInfoCount = arraysize(queueCreateInfos);
27 createInfo.pQueueCreateInfos = queueCreateInfos;
28 createInfo.enabledExtensionCount = deviceExtensionCount;
29 createInfo.ppEnabledExtensionNames = deviceExtensions;
30 createInfo.pEnabledFeatures = &deviceFeatures;
```

Listing 2.22: Create info struct when queue families are different

2.5.2 Retrieve Queue Handles

After creating our logical device, we retrieve the handles to the queues we created together with our device.

```
1  VkQueue graphicsQueue = VK_NULL_HANDLE;
2  vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &
       graphicsQueue);
3
4  VkQueue presentQueue = VK_NULL_HANDLE;
5  vkGetDeviceQueue(device, presentQueueFamilyIndex, 0, &presentQueue)
       ;
```

Listing 2.23: Retrieve queue handles

2.5.3 Cleanup

We use `vkDestroyDevice` to destroy our logical device.

2.6 Create A Swapchain

After having created our logical device, we can create a swapchain object. We need a swapchain to handle the logic for image presentation. A swapchain creates and manages a set of images that can be presented to the screen.

```
1  VkSwapchainKHR swapchain = VK_NULL_HANDLE;
2  vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapchain);
```

Listing 2.24: Create a swapchain

2.6.1 VkSwapchainCreateInfoKHR

We use a `VkSwapchainCreateInfoKHR` struct to configure the swapchain we are about to create.

```
1  VkSwapchainCreateInfoKHR createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
3  createInfo.surface = surface;
4  createInfo.minImageCount = swapchainMinImageCount;
5  createInfo.imageFormat = swapchainImageFormat.format;
6  createInfo.imageColorSpace = swapchainImageFormat.colorSpace;
7  createInfo.imageExtent = swapchainImageExtent;
8  createInfo.imageArrayLayers = 1;
9  createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
10 createInfo.preTransform = surfaceCapabilites.currentTransform;
11 createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
12 createInfo.presentMode = swapchainPresentMode;
13 createInfo.clipped = VK_TRUE;
14 createInfo.oldSwapchain = VK_NULL_HANDLE;
```

Listing 2.25: Configure our swapchain

We have to additionally provide other data that depends on whether or not we use the same queue for graphics and present operations.

```
1  u32 queueFamilyIndices[] =
2  {
3      graphicsQueueFamilyIndex,
4      presentQueueFamilyIndex,
5  };
6
7  if (graphicsQueueFamilyIndex != presentQueueFamilyIndex)
8  {
9      // Using the concurrent sharing mode we don't need to worry
10     // about resource queue ownership transitions
11     swapchainCreateInfo.imageSharingMode =
       VK_SHARING_MODE_CONCURRENT;
12     swapchainCreateInfo.queueFamilyIndexCount = arraysize(
       queueFamilyIndices);
13     swapchainCreateInfo.pQueueFamilyIndices = queueFamilyIndices;
14 }
15 else
16 {
17     // We use a single queue, thus it can exclusively own the
18     // swapchain images that will be created.
19     // This is more efficient
20     swapchainCreateInfo.imageSharingMode =
       VK_SHARING_MODE_EXCLUSIVE;
21 }
```

Listing 2.26: Configure queue ownership over swapchain images

2.6.2 Select The Minimum Swapchain Image Count

Here we want to determine the minimum number of swapchain images to create. We can do this by querying the surface capabilities with `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.

```
1  u32 swapchainMinImageCount = capabilities->minImageCount + 1;
2  // If maxImageCount is 0, there is no limit on the number of images
3  if ((capabilities->maxImageCount > 0) && (swapchainMinImageCount >
       capabilities->maxImageCount))
4  {
5      swapchainMinImageCount = capabilities->maxImageCount;
6  }
7
8  return swapchainMinImageCount;
```

Listing 2.27: Select swapchain image count

Here we would like to use one more image than the bare minimum. This is due to the fact that, if we use the bare minimum number of images, we may have to wait for the driver to complete internal operations before we can acquire another swapchain image to render to.

Here we also have to be aware of the fact that there can be a maximum number of swapchain images we can require. Thus, we must be careful to cap the number of images that we request to the nominal maximum.

2.6.3 Select The Swapchain Image Format

We must specify a proper format for our swapchain images. To do this, we first query for all image formats that are supported by our surface. We can do this using `vkGetPhysicalDeviceSurfaceFormatsKHR`. Once we have a list of valid formats we could either pick one randomly or try to pick the one that we consider the best. Here, we would like to use SRGB color space, with 32 bit RGBA format.

```
1  if ((formatCount == 1) && (formats[0].format == VK_FORMAT_UNDEFINED
       ))
2  {
3      // There is no preferred surface format
4      return { VK_FORMAT_R8G8B8A8_UNORM,
       VK_COLORSPACE_SRGB_NONLINEAR_KHR };
5  }
6  else
7  {
8      // We have to pick a format from the list
9      for (u32 i = 0; i < formatCount; i++)
10     {
11         if (formats[i].format == VK_FORMAT_R8G8B8A8_UNORM)
12         {
13             return formats[i];
14         }
15     }
16
17     // We haven't found the format(s) that we were looking for
18     // Pick the first format
19     return formats[0];
20 }
```

Listing 2.28: Select swapchain image format

2.6.4 Select The Swapchain Image Extent

We must specify the resolution for our swapchain images. This will almost always be equal to the resolution of our window. Some windowing systems allow us to differ, indicating that the current width and height are the maximum value of a an unsigned 32 bits integer. In this scenario, we have to pick the resolution that best matches the window within the bounds specified by our surface capabilities.

```
1  if (capabilities->currentExtent.width == 0xFFFFFFFF)
2  {
3      VkExtent2D extent = { windowWidth, windowHeight };
4      extent.width = clamp(extent.width, capabilities->minImageExtent
       .width, capabilities->maxImageExtent.width);
5      extent.height = clamp(extent.height, capabilities->
       minImageExtent.height, capabilities->maxImageExtent.height);
6      return extent;
7  }
8  else
9  {
10     // the current surface size is perfect for the job
11     return capabilities->currentExtent;
12 }
```

Listing 2.29: Select swapchain image extent

2.6.5 Select The Swapchain Presentation Mode

We start by listing all the presentation modes that our physical device supports for presenting images to our surface. We can do this using `vkGetPhysicalDeviceSurfacePresentModesKHR`. We already did this while selecting our physical device. After that, we check whether the mailbox presentation mode is supported. We would like to use this present mode because it doesn't suffer from tearing and it's not locked to the screen refresh rate. If it's present we are good to go. Otherwise we select a presentation mode that is guaranteed to be always supported: `VK_PRESENT_MODE_FIFO_KHR`.

```
1  for (u32 i = 0; i < modeCount; i++)
2  {
3      if (modes[i] == VK_PRESENT_MODE_MAILBOX_KHR)
4      {
5          return VK_PRESENT_MODE_MAILBOX_KHR;
6      }
7  }
8
9  // Use FIFO since it's always supported (spec)
10 return VK_PRESENT_MODE_FIFO_KHR;
```

Listing 2.30: Select swapchain present mode

2.6.6 Retrieve Swapchain Images

Now that we have created a swapchain we can retrieve the handles to the images that were created together with it. We will use these images during rendering. We can do this using `vkGetSwapchainImagesKHR`;

2.6.7 Create Swapchain Image Views

Vulkan doesn't allow us to use images directly. Before using an image, we first have to create a view on it. This also applies to our swapchain images. Thus, for every image in the swapchain, we must create a corresponding image view for it.

```
1  VkImageViewCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
3  createInfo.image = swapchainImages[i];
4  createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
5  createInfo.format = swapchainImageFormat.format;
6  createInfo.components =
7  {
8      VK_COMPONENT_SWIZZLE_IDENTITY,
9      VK_COMPONENT_SWIZZLE_IDENTITY,
10     VK_COMPONENT_SWIZZLE_IDENTITY,
11     VK_COMPONENT_SWIZZLE_IDENTITY,
12 };
13 createInfo.subresourceRange =
14 {
15     VK_IMAGE_ASPECT_COLOR_BIT,
16     0,
17     1,
18     0,
19     1,
20 };
21
22 VkImageView* swapchainImageViews = nullptr;
23 vkCreateImageView(device, &createInfo, nullptr, &
       swapchainImageViews[i]);
```

Listing 2.31: Create swapchain image views

2.6.8 Cleanup

We first destroy the swapchain image views using `vkDestroyImageView`. After that, we destroy the swapchain itself using `vkDestroySwapchainKHR`. The swapchain images will be automatically destroyed when `vkDestroySwapchainKHR` is called.

2.7 Our Application So Far

Here we can see how all the parts we presented in this chapter fit together to form a working application.

```
1  int main()
2  {
3      // Create Vulkan instance and debug messenger ...
4      // Create window ...
5      // Create presentation surface ...
6      // Pick physical device ...
7      // Create logical device ...
8      // Create swapchain ...
9
10     bool isApplicationRunning = true;
11     while (isApplicationRunning)
12     {
13         // Process window messages ...
14     }
15
16     // Cleanup ...
17
18     return 0;
19 }
```

Listing 2.32: Structure of our application

Chapter 3

Clearing The Window

In this chapter we see all the steps that are required to clear our window with a flat color.

During the application startup phase, we create some resources required for rendering: two semaphores, used to synchronize the execution of graphics and present commands, a command buffer, used to submit commands to the GPU, and a render pass, used to describe the rendering itself.

During the application main loop, we must follow these steps for our rendering to be correct. We acquire a swapchain image, this will be used as our render target. Before starting recording new commands into the command buffer, we wait for the previously submitted commands to finish their execution. We bundle the acquired swapchain image into a framebuffer, this will let us use said image during rendering. We record the appropriate graphics commands into the command buffer. We submit the command buffer to the graphics queue. Finally, we submit a present command to the present queue.

Figure 3.1: Clear the window background blue

3.1 Create Commands Synchronization Resources

When we use Vulkan, we have to take into account the fact that we must handle commands synchronizations ourselves. In particular, we must synchronize rendering and present commands. To accomplish this, we use two semaphores. One semaphore will be signaled when a swapchain image is available to be used as our render target. When this semaphore is signaled, we may start rendering. Another semaphore will be signaled when we finish rendering. When this semaphore is signaled, we may present the image.

```
1  VkSemaphore imageAvailableSemaphore = VK_NULL_HANDLE;
2  VkSemaphore renderFinishedSemaphore = VK_NULL_HANDLE;
3
4  VkSemaphoreCreateInfo createInfo = {};
5  createInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
6  vkCreateSemaphore(device, &createInfo, nullptr, &
       imageAvailableSemaphore);
7  vkCreateSemaphore(device, &createInfo, nullptr, &
       renderFinishedSemaphore);
```

Listing 3.1: Create semaphores

3.1.1 Cleanup

We destroy the previously allocated semaphores with `vkDestroySemaphore`.

3.2 Create Command Buffer

We use a command buffer to submit commands to the GPU. We use `vkAllocateCommandBuffers` to create a command buffer.

```
1  VkCommandBuffer commandBuffer = VK_NULL_HANDLE;
2  vkAllocateCommandBuffers(device, &allocInfo, commandBuffer);
```

Listing 3.2: Allocate a command buffer from our graphics command pool

3.2.1 VkCommandBufferAllocateInfo

We use a `VkCommandBufferAllocateInfo` struct to configure the command buffer we are about to create. In our case we allocate a primary command buffer. Such buffers can be directly submitted to the GPU; this is what we want.

```
1  VkCommandBufferAllocateInfo allocInfo = {};
2  allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
3  allocInfo.commandPool = graphicsCommandPool;
4  allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
5  allocInfo.commandBufferCount = 1;
```

Listing 3.3: Configure command buffer creation

3.2.2 Create Command Pool

We create a command buffer allocating it from a command pool. Thus, before creating a command buffer, we must create a command pool. In our case, we explicitly submit commands only to the graphics queue. Hence, we only need to create one graphics command pool.

```
1  VkCommandPool graphicsCommandPool = VK_NULL_HANDLE;
2  vkCreateCommandPool(device, &createInfo, nullptr, &
       graphicsCommandPool);
```

Listing 3.4: Create graphics command pool

VkCommandPoolCreateInfo

We use a `VkCommandPoolCreateInfo` struct to configure the command pool we are about to create. Here we use the reset command buffer flag because we want to be able to write commands multiple times into the command buffers created from this pool.

```
1  VkCommandPoolCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
3  createInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
4  createInfo.queueFamilyIndex = graphicsQueueFamilyIndex;
```

Listing 3.5: Configure our graphics command pool

Cleanup

When our application is shutting down, we have to destroy all the previously created command pools. To do this we use `vkDestroyCommandPool`.

3.2.3 Command Buffer Fence

Together with our command buffer, we also create a fence. We can use a fence to wait for our command buffer execution to finish. The fence that we create is already signaled from the start. This is due to how we will use it later.

```
1  VkFenceCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
3  createInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
4
5  VkFence commandBufferFence = VK_NULL_HANDLE;
6  vkCreateFence(device, &createInfo, nullptr, &commandBufferFence);
```

Listing 3.6: Create a fence for our command buffer

3.2.4 Cleanup

We use `vkFreeCommandBuffers` to free the previously allocated command buffers. We use `vkDestroyFence` to destroy our the previously created fences.

3.3 Create Render Pass

Before rendering, we need to describe what types of images will be used and the order of our draw calls. To do this we create a render pass.

```
1  VkRenderPass renderPass = VK_NULL_HANDLE;
2  vkCreateRenderPass(device, &createInfo, nullptr, &renderPass);
```

Listing 3.7: Create a render pass

3.3.1 VkRenderPassCreateInfo

We use a `VkRenderPassCreateInfo` struct to configure the render pass we are about to create.

```
1  VkRenderPassCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
3  createInfo.attachmentCount = attachmentCount;
4  createInfo.pAttachments = attachments;
5  createInfo.subpassCount = subpassCount;
6  createInfo.pSubpasses = subpassess;
7  // If there is more than one subpass, we need to specify
8  // synchronization requirements through subpass dependencies
9  createInfo.dependencyCount = 0;
10 createInfo.pDependencies = nullptr;
```

Listing 3.8: Configure our render pass

3.3.2 Render Pass Attachment Descriptions

During render pass creation, we specify an array of attachment descriptions. This array describes all the attachments that are going to be used by the render pass.

In our case we have only one attachment. This attachment will be one of the swapchain images. We want to clear our attachment before using it for the first time in the render pass. We want to preserve the attachment's contents after using it for the last time in the render pass. We don't care about the attachment's stencil components. We don't care about the attachment's image layout before starting the render pass. We want to transition the attachment to a layout compatible with image presentation at the end of the render pass.

```
1  VkAttachmentDescription colorAttachment = {};
2  colorAttachment.format         = swapchainImageFormat;
3  colorAttachment.samples        = VK_SAMPLE_COUNT_1_BIT;
4  colorAttachment.loadOp         = VK_ATTACHMENT_LOAD_OP_CLEAR;
5  colorAttachment.storeOp        = VK_ATTACHMENT_STORE_OP_STORE;
6  colorAttachment.stencilLoadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
7  colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
8  colorAttachment.initialLayout  = VK_IMAGE_LAYOUT_UNDEFINED;
9  colorAttachment.finalLayout    = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
10
11 VkAttachmentDescription attachments[] =
12 {
13     colorAttachment,
14 };
```

Listing 3.9: Render pass attachment descriptions

3.3.3 Render Pass Subpasses

During render pass creation, we specify an array of subpass descriptions. This array describes the subpasses that define the render pass.

In our case we have only one subpass that uses our single attachment to write color data into it. Note that we specify the image layout that the attachment must have during this subpass.

```
1  VkAttachmentReference colorAttachmentReference = {};
2  // Attachment's index in 'attachments' array
3  colorAttachmentReference.attachment = 0;
4  // Layout the attachment uses during the subpass
5  colorAttachmentReference.layout =
       VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
6
7  VkAttachmentReference colorAttachmentReferences[] =
8  {
9      colorAttachmentReference,
10 };
11
12 VkSubpassDescription colorSubpass = {};
13 colorSubpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
14 colorSubpass.colorAttachmentCount = arraysize(
       colorAttachmentReferences);
15 colorSubpass.pColorAttachments = colorAttachmentReferences;
16
17 VkSubpassDescription subpassess[] =
18 {
19     colorSubpass,
20 };
```

Listing 3.10: Render pass subpass descriptions

3.3.4 Cleanup

To destroy our render pass we use `vkDestroyRenderPass`.

3.4 Clear The Window

In our application, for every iteration of the main loop, we render an image. In this case, we simply clear the window background with a flat color.

3.4.1 Acquire A Swapchain Image

The first step for drawing something to the screen is to get an image that serves as our render target. This image must also be presentable to the presentation engine. Only swapchain images satisfy the latter requirement. Hence, we must use one of them as our render target. The problem is that we don't know the next available swapchain image. To determine such image we use `vkAcquireNextImageKHR`. Note that the image is not guaranteed to be already available when the function returns. For this reason, we use the image available semaphore we created earlier. It will be signaled when the image will actually be ready.

```
u32 nextSwapchainImageIndex = 0;
vkAcquireNextImageKHR(
    device,
    swapchain,
    UINT64_MAX,
    imageAvailableSemaphore,
    VK_NULL_HANDLE,
    &nextSwapchainImageIndex
);

nextSwapchainImage = swapchainImages[nextSwapchainImageIndex];
nextSwapchainImageView = swapchainImageViews[
    nextSwapchainImageIndex];
```

Listing 3.11: Acquire the next swapchain image that will be presented

3.4.2 Wait For The Previous Commands To Finish

Before recording new commands into our command buffer, we have to wait for the previously submitted commands to finish. To do this wait on the command buffer fence. After the wait terminates, we have to manually reset the fence state to unsignaled. We do this so that we can wait on the fence again, during the next frame.

```
vkWaitForFences(device, 1, &commandBufferFence, VK_TRUE, UINT64_MAX
    );
vkResetFences(device, 1, &commandBufferFence);
```

Listing 3.12: Wait for command buffer execution to finish

3.4.3 Create A Framebuffer

Before recording our rendering commands, we need to create a new framebuffer. A framebuffer is the set of attachments that a render pass uses during rendering. Before creating a new framebuffer, remember to destroy the framebuffer that was used during the previous frame. It's also important to remember to destroy the last created framebuffer during the application cleanup phase.

```
vkDestroyFramebuffer(device, framebuffer, nullptr);
vkCreateFramebuffer(device, &createInfo, nullptr, &framebuffer);
```

Listing 3.13: Create a new framebuffer

VkFramebufferCreateInfo

To configure the framebuffer we are about to create we use a `VkFramebufferCreateInfo` struct. In our case, our framebuffer will contain a single attachment: the next available swapchain image.

```
1  VkFramebufferCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
3  createInfo.renderPass = renderPass;
4  createInfo.attachmentCount = 1;
5  createInfo.pAttachments = &nextSwapchainImageView;
6  createInfo.width = swapchainImageExtent.width;
7  createInfo.height = swapchainImageExtent.height;
8  createInfo.layers = 1;
```

Listing 3.14: Configure our framebuffer

3.4.4 Record Rendering Commands

Now we can start recording the new rendering commands. All the functions that write a command into our command buffer must lay between `vkBeginCommandBuffer` and `vkEndCommandBuffer`.

```
1  VkCommandBufferBeginInfo beginInfo = {};
2  beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3  beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4  vkBeginCommandBuffer(commandBuffer, &beginInfo);
5
6  // Vulkan commands go here ...
7
8  vkEndCommandBuffer(commandBuffer);
```

Listing 3.15: Boilerplate code for recording a command buffer

Here we are recording a one time submit command buffer. It means that each recording will only be submitted once to the GPU. Indeed, for every frame, we record and then submit our command buffer. Hence, each recording will be submitted only once. We do this so that we can change our clear color over time.

```
1  VkClearValue clearValue = {};
2  {
3      f32 red = 0.0f;
4      f32 blue = std::abs(std::sin(time));
5      f32 green = 0.0f;
6      clearValue.color = { red, green, blue, 0.0f };
7  }
```

Listing 3.16: Change window clear color over time

Now we can actually write some commands into our command buffer. The idea is very simple. We record two commands: the first is for starting an instance of our render pass; the second is for ending the render pass instance.

```
1  vkCmdBeginRenderPass(commandBuffer, &beginInfo,
       VK_SUBPASS_CONTENTS_INLINE);
2  vkCmdEndRenderPass(commandBuffer);
```

Listing 3.17: Clear the window using a render pass

First we have to configure the render pass instance using a `VkRenderPassBeginInfo` struct. The field that requires an explanation is `pClearValues`. This is an array of clear values for each attachment. The array is indexed by attachment number. Consider the i-th attachment. If it has `VK_ATTACHMENT_LOAD_OP_CLEAR` as `loadOp` value, then `pClearValues[i]` will be used for the clear value. Otherwise, `pClearValues[i]` will be ignored.

We use the clear color we computed earlier as our clear value.

```
1  VkRenderPassBeginInfo beginInfo = {};
2  beginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
3  beginInfo.renderPass = renderPass;
4  beginInfo.framebuffer = framebuffer;
5  beginInfo.renderArea.offset = { 0, 0 };
6  beginInfo.renderArea.extent = context.swapchainImageExtent;
7  beginInfo.clearValueCount = 1;
8  beginInfo.pClearValues = &clearValue;
```

Listing 3.18: Configure our render pass instance

Now we can explain how we clear the image with our clear value. We start by beginning our render pass. This causes the first subpass to start. Right before the start of our subpass, an implicit image layout transition occurs. This causes the swapchain image to transition to `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. With this layout, we can write color data into the image. Since our subpass is the first to use our swapchain image color attachment, the image is cleared using the specified clear value. Right before ending the render pass, another implicit image layout transition occurs. This causes the swapchain image to transition to `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. With this layout, our image can be used by the presentation engine.

3.4.5 Submit Rendering Commands

Once we have recorded all the necessary rendering commands into a command buffer, we can submit it to the GPU for execution. In our case, we submit the command buffer to the graphics queue. When the execution of the command buffer is completed, our command buffer fence will be signaled.

```
1  vkQueueSubmit(graphicsQueue, 1, &submitInfo, commandBufferFence);
```

Listing 3.19: Submit command buffer to the GPU

We use a `VkSubmitInfo` struct to configure our command buffer submission.

`pWaitSemaphores` is an array of semaphores upon which to wait before the submitted command buffers begin execution. In our case we only use one semaphore: our image available semaphore. We do this because we have to wait for our swapchain image to be available before rendering into it.

`pWaitDstStageMask` is a bitmask of pipeline stages at which each corresponding semaphore wait will occur. In our case we are saying that we do our semaphore wait as soon as the graphics pipeline starts executing the commands recorded into our command buffer.

`pSignalSemaphores` is an array of semaphores to be signaled once the submitted command buffers have completed execution. In our case we signal only one semaphore: our render finished semaphore. When this semaphore is signaled, it means that we have finished rendering our image.

```
1  VkPipelineStageFlags waitDstStageMask =
       VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
2
3  VkSubmitInfo submitInfo = {};
4  submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
5  submitInfo.waitSemaphoreCount = 1;
6  submitInfo.pWaitSemaphores = &imageAvailableSemaphore;
7  submitInfo.pWaitDstStageMask = &waitDstStageMask;
8  submitInfo.commandBufferCount = 1;
9  submitInfo.pCommandBuffers = &commandBuffer;
10 submitInfo.signalSemaphoreCount = 1;
11 submitInfo.pSignalSemaphores = &renderFinishedSemaphore;
```

Listing 3.20: Configure command buffer submission

3.4.6 Present

The only thing missing is to actually present our rendered image to the window. Here we specify our present queue as the GPU queue that will execute our present command.

```
1  vkQueuePresentKHR(presentQueue, &presentInfo);
```

Listing 3.21: Issue a present command

We use a `VkPresentInfoKHR` struct to configure our present command submission.

`pWaitSemaphores` is an array of semaphores to wait for before issuing the present command. In our case we only use one semaphore: our render finished semaphore. Simply put, we have to wait for our rendering to finish before presenting the image to the window.

```
1  VkPresentInfoKHR presentInfo = {};
2  presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
3  presentInfo.waitSemaphoreCount = 1;
4  presentInfo.pWaitSemaphores = &renderFinishedSemaphore;
5  presentInfo.swapchainCount = 1;
6  presentInfo.pSwapchains = &swapchain;
7  presentInfo.pImageIndices = &nextSwapchainImageIndex;
8  presentInfo.pResults = nullptr;
```

Listing 3.22: Configure present command submission

3.5 Cleanup

Now that our application submits commands to the GPU, a problem may arise. Being commands executed asynchronously on the GPU, we could exit the application, and thus freeing all our resources, before all submitted commands finish their execution. This can lead to errors, because some commands may act upon one or more resources that were deleted. We can fix this issue by waiting for our device to be idle, meaning that all processing on all device's queues is finished, before cleaning up our resources. We can do this using `vkDeviceWaitIdle`.

3.6 Our Application So Far

Here we can see how all the concepts we have seen in this chapter come together to form our application

```
1  int main()
2  {
3      // Initialize Vulkan ...
4
5      // Create semaphores ...
6      // Crate graphics command pool ...
7      // Create command buffer and fence ...
8      // Create render pass ...
9
10     bool isApplicationRunning = true;
11     while (isApplicationRunning)
12     {
13         // Process window messages ...
14
15         // Acquire a swapchain image ...
16         // Wait for the previous commands to finish ...
17         // Create a framebuffer ...
18         // Record rendering commands ...
19         // Submit rendering commands ...
20         // Present ...
21     }
22
23     // Wait device idle ...
24
25     // Destroy last created framebuffer ....
26
27     // Cleanup ...
28
29     return 0;
30 }
```

Listing 3.23: Structure of our application

Chapter 4

Rendering Our First Triangle

In this chapter we see how to render a triangle on the screen. This is a very important step. If we are able to draw a single triangle, then we can draw almost any shape. We can do this by considering the shape we want to draw as if it were made up of one or more triangles and then drawing them. For example a square can be made using two triangles. A cube can be made using six squares. And the list goes on.

In order to render a triangle using Vulkan, we use a pipeline state object. This object describes the entire state of the graphics pipeline. Thus, it also describes how we want to draw something.

In the application main loop, during the command buffer recording, we simply need to use our pipeline state object and issue a draw call that activates our pipeline and draws what we want.

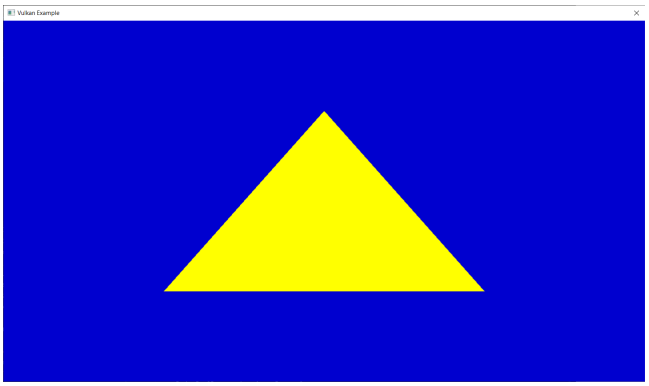

Figure 4.1: Rendering our triangle

4.1 Create A Pipeline State Object

To create a pipeline state object we use `vkCreateGraphicsPipelines`.

```
1  VkPipeline pipeline = VK_NULL_HANDLE;
2  vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &createInfo,
       nullptr, &pipeline);
```

Listing 4.1: Create a pipeline state object

4.1.1 VkGraphicsPipelineCreateInfo

We use a `VkGraphicsPipelineCreateInfo` struct to configure the pipeline state object we are about to create.

```
1  VkGraphicsPipelineCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
3  createInfo.stageCount          = arraysize(shaderStages);
4  createInfo.pStages             = shaderStages;
5  createInfo.pVertexInputState   = &vertexInputInfo;
6  createInfo.pInputAssemblyState = &inputAssemblyState;
7  createInfo.pTessellationState  = nullptr;
8  createInfo.pViewportState      = &viewportState;
9  createInfo.pRasterizationState = &rasterizationState;
10 createInfo.pMultisampleState   = &multisamplingState;
11 createInfo.pDepthStencilState  = &depthStencilState;
12 createInfo.pColorBlendState    = &colorBlendState;
13 createInfo.pDynamicState       = nullptr;
14 createInfo.layout              = pipelineLayout;
15 createInfo.renderPass          = renderPass;
16 createInfo.subpass             = 0;
```

Listing 4.2: Configure pipeline state object

`renderPass` is a handle to a render pass object describing the environment in which the pipeline will be used. `subpass` is the index of the subpass in the render pass where this pipeline will be used. We will explain the meaning of the remaining relevant struct fields in the next sections.

4.1.2 Shader Stages

We specify a collection of all shader stages and shader programs that will be used during rendering.

```
1  VkPipelineShaderStageCreateInfo shaderStages[] =
2  {
3      vertShaderStageInfo,
4      fragShaderStageInfo,
5  };
```

Listing 4.3: Shader stages

VkPipelineShaderStageCreateInfo

In our case we need two instances of a `VkPipelineShaderStageCreateInfo` struct that describe our vertex and our fragment shader stages.

```
1  VkPipelineShaderStageCreateInfo stageInfo = {};
2  stageInfo.sType =
       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
3  stageInfo.stage = stage;
4  stageInfo.module = shaderModule;
5  stageInfo.pName = "main";
```

Listing 4.4: Describe a shader stage

`stage` is a `VkShaderStageFlagBits` value specifying the pipeline stage. `module` is a shader module object containing the shader for this stage. `pName` is a string specifying the entry point name of the shader for this stage.

In order to create our vertex shader stage we need to pass the shader module that contains our vertex shader code and use the `VK_SHADER_STAGE_VERTEX_BIT` stage flag bit. In order to create our fragment shader stage we need to pass the shader module that contains our fragment shader code and use the `VK_SHADER_STAGE_FRAGMENT_BIT` stage flag bit.

VkShaderModule

To create a shader module we need to load our shader code written in SPIR-V binary format. Then, we simply use `vkCreateShaderModule`. After creating our shader module, we can discard our loaded shader code.

```
1  const char* shaderPath = "path/to/shader.spv";
2  u32 codeSize = GetFileSize(shaderPath);
3  u8* codeData = LoadFile(shaderPath, codeSize);
4
5  VkShaderModuleCreateInfo createInfo = {};
6  createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
7  createInfo.codeSize = codeSize;
8  createInfo.pCode = (u32*)(codeData);
9
10 VkShaderModule shaderModule = VK_NULL_HANDLE;
11 vkCreateShaderModule(device, &createInfo, nullptr, shaderModule);
12
13 free(codeData);
14 codeSize = 0;
```

Listing 4.5: Create a shader module

In our case, we create one shader module for our vertex shader and one shader module for our fragment shader.

Vertex Shader Code

We write our vertex shader in GLSL. We use a `.vert` file extension. 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 an hardcoded array of vertex data. We will see how to upload vertex data later. The built in variable `gl_Position` functions as the output.

```
#version 450
#extension GL_KHR_vulkan_glsl : enable

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);
}
```

Listing 4.6: Our first vertex shader

Fragment Shader Code

We write our fragment shader in GLSL. We use a `.frag` file extension. Unlike `gl_Position` in the vertex shader, there is no built in variable to output a color for the current fragment. We have to specify our own output variable. The color yellow is written to this `outColor` variable.

```
#version 450

layout(location = 0) out vec4 outColor;

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

Listing 4.7: Our first fragment shader

Compiling GLSL To SPIR-V

Since we write our shaders in GLSL, we need to compile them into SPIR-V binary format. The compiler that does this is shipped together with the Vulkan SDK.

Cleanup

After the pipeline state object is created, we can destroy the shader modules we created using `vkDestroyShaderModule`.

4.1.3 Vertex Input State

We use a `VkPipelineVertexInputStateCreateInfo` struct to configure the vertex input state of the pipeline object we are about to create.

This struct describes the format of the vertex data that will be passed to the vertex shader. Here we don't have any vertex data.

```
1  VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
2  vertexInputInfo.sType =
       VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
```

Listing 4.8: Configure vertex input state

4.1.4 Input Assembly State

We use a `VkPipelineInputAssemblyStateCreateInfo` struct to configure the input assembly state of the pipeline object we are about to create.

This struct describes how vertices are assembled into primitives. In our case, the vertex data we have hardcoded into our vertex shader specifies a list of triangles.

```
1  VkPipelineInputAssemblyStateCreateInfo inputAssemblyState = {};
2  inputAssemblyState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
3  inputAssemblyState.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
```

Listing 4.9: Configure input assembly state

4.1.5 Viewport State

We use a `VkPipelineViewportStateCreateInfo` struct to configure the viewport state of the pipeline object we are about to create.

```
1  VkPipelineViewportStateCreateInfo viewportState = {};
2  viewportState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
3  viewportState.viewportCount = 1;
4  viewportState.pViewports = &viewport;
5  viewportState.scissorCount = 1;
6  viewportState.pScissors = &scissor;
```

Listing 4.10: Configure viewport state

Viewports

The `pViewports` struct field is an array of viewports that will be used by our pipeline. A viewport describes what part of the image (or texture, or window) we want to draw. In our case we want to draw the entire image. The graphics pipeline also uses viewports to transform normalized device coordinates into screen coordinates.

```
1  VkViewport viewport = {};
2  // the viewport's upper left corner (x,y)
3  viewport.x = 0.0f;
4  viewport.y = 0.0f;
5  // viewport's width and height
6  viewport.width = (f32)(swapchainImageExtent.width);
7  viewport.height = (f32)(swapchainImageExtent.height);
8  // the depth range for the viewport
9  viewport.minDepth = 0.0f;
10 viewport.maxDepth = 1.0f;
```

Listing 4.11: Viewport

Scissors

The `pScissors` struct field is an array of scissor rectangles. The graphics pipeline uses these rectangles to decide which fragments to discard. Any pixels outside the scissor rectangles will be discarded by the rasterizer. In our case we don't discard any fragments.

```
1  VkRect2D scissor = {};
2  scissor.offset.x = 0;
3  scissor.offset.y = 0;
4  scissor.extent = swapchainImageExtent;
```

Listing 4.12: Scissor

4.1.6 Rasterization State

We use a `VkPipelineRasterizationStateCreateInfo` struct to configure the rasterization state of the pipeline object we are about to create.

This struct describes how polygons are going to be rasterized (changed into fragments). 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. The rasterizer also performs depth testing, face culling and the scissor test.

```
1  VkPipelineRasterizationStateCreateInfo rasterizationState = {};
2  rasterizationState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
3  rasterizationState.polygonMode = VK_POLYGON_MODE_FILL;
4  rasterizationState.lineWidth = 1.0f;
```

Listing 4.13: Rasterization state

`polygonMode` determines how fragments are generated for geometry. Using any mode other than fill requires enabling a GPU feature. `lineWidth` describes the width of rasterized line segments.

4.1.7 Multisample State

We use a `VkPipelineMultisampleStateCreateInfo` struct to configure the multisample state of the pipeline object we are about to create. We don't use multisampling here.

```
1  VkPipelineMultisampleStateCreateInfo multisamplingState = {};
2  multisamplingState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
3  multisamplingState.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
```

Listing 4.14: Multisample state

4.1.8 Depth Stencil State

We use a `VkPipelineDepthStencilStateCreateInfo` struct to configure the depth stencil state of the pipeline object we are about to create. We neither use depth testing nor stencil testing here.

```
1  VkPipelineDepthStencilStateCreateInfo depthStencilState = {};
2  depthStencilState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
3  depthStencilState.depthTestEnable = VK_FALSE;
4  depthStencilState.stencilTestEnable = VK_FALSE;
```

Listing 4.15: Depth stencil state

4.1.9 Color Blend State

We use a `VkPipelineColorBlendStateCreateInfo` struct to configure the color blend state of the pipeline object we are about to create.

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.

```
1  VkPipelineColorBlendStateCreateInfo colorBlendState = {};
2  colorBlendState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
3  colorBlendState.attachmentCount = 1;
4  colorBlendState.pAttachments = &colorBlendAttachmentState;
```

Listing 4.16: Color blend state

`pAttachments` is an array of of `VkPipelineColorBlendAttachmentState` structures defining blend state for each color attachment.

VkPipelineColorBlendAttachmentState

We have to configure how color blending works for every color attachment in our framebuffer. Since we have only one color attachment, we need only one description. In our case, we don't use color blending. We simply write all the color components to the framebuffer as they are.

```
1  VkPipelineColorBlendAttachmentState colorBlendAttachmentState = {};
2  colorBlendAttachmentState.blendEnable = VK_FALSE;
3  colorBlendAttachmentState.colorWriteMask = VK_COLOR_COMPONENT_R_BIT
       | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT |
      VK_COLOR_COMPONENT_A_BIT;
```

Listing 4.17: Color blend attachment state

`blendEnable` controls whether blending is enabled for the corresponding color attachment. If blending is not enabled, the source fragment's color for that attachment is passed through unmodified.

`colorWriteMask` is a bitmask specifying which of the R, G, B, and/or A components are enabled for writing. This bitmask determines whether the final color values R, G, B and A are written to the framebuffer attachment.

4.1.10 Pipeline Layout

Before creating our pipeline, we need to define its layout. We do this by creating a pipeline layout object.

```
VkPipelineLayout pipelineLayout = VK_NULL_HANDLE;
vkCreatePipelineLayout(device, &createInfo, nullptr, &
    pipelineLayout)
```

Listing 4.18: Create our pipeline layour

VkPipelineLayoutCreateInfo

We use a `VkPipelineLayoutCreateInfo` struct to configure the pipeline layout we are about to create. In this scenario we can ignore the pipeline layout. We will use it in later chapters.

```
VkPipelineLayoutCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
```

Listing 4.19: Configure our pipeline layout

4.1.11 Cleanup

We first destroy our pipeline state object using `vkDestroyPipeline`. Then, we destroy our pipeline layout using `vkDestroyPipelineLayout`.

4.2 Use Our Pipeline To Draw A Triangle

Now that we have created our pipeline state object, we can use it to set the graphics pipeline current state. Then we can tell our graphics pipeline to draw three vertices. This will lead to our triangle being rendered.

```
// begin render pass ...

vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipeline);
vkCmdDraw(
    commandBuffer,
    3, // number of vertices to draw
    1, // number of instances to draw (we don't use instancing)
    0, // index of the first vertex to draw
    0  // instance ID of the first instance to draw
);

// end render pass ...
```

Listing 4.20: Rendering our triangle

Chapter 5

Shader Local Data: Vertices

In this chapter we see how to pass per-vertex data to our vertex shader. To accomplish this task we introduce the concept of vertex buffer. We also have an in depth look at a technique that allows us to get the most performance out of a vertex buffer. At the end, we use the vertex data stored in our vertex buffer to render a quad.

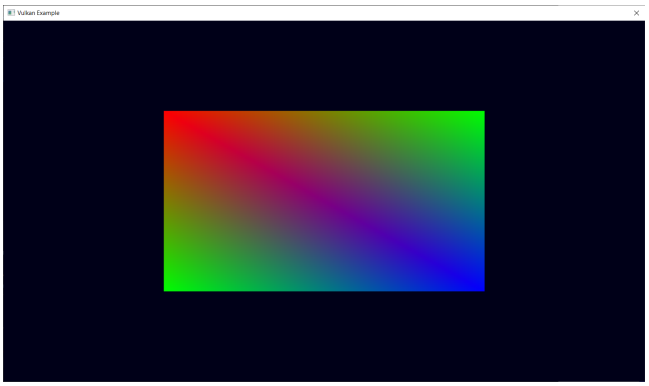

Figure 5.1: Rendering a quad

5.1 Vertex Data

In this section we define the vertex data that we will use to draw our quad. We will first define the structure of a single vertex. After that, we will lay out the quad's vertices.

5.1.1 Vertex

Here we define the structure of our vertices. In our case, a vertex stores its 2D position in normalized device coordinates and its color.

```
1  struct Vertex
2  {
3      glm::vec2 position;
4      glm::vec3 color;
5  };
```

Listing 5.1: What data we store per vertex

5.1.2 Vertex Data

Here we want to draw a quad. The problem is that we can only render triangles. We solve this issue by building our quad using two triangles. Hence, to define our quad, we need six vertices. These are the vertices that will be later uploaded to our GPU and be used by the graphics pipeline.

```
1  Vertex vertices[] =
2  {
3      { { -0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
4      { { +0.5f, -0.5f }, { 0.0f, 1.0f, 0.0f } },
5      { { +0.5f, +0.5f }, { 0.0f, 0.0f, 1.0f } },
6      { { +0.5f, +0.5f }, { 0.0f, 0.0f, 1.0f } },
7      { { -0.5f, +0.5f }, { 0.0f, 1.0f, 0.0f } },
8      { { -0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
9  };
```

Listing 5.2: The vertices that our application will use

5.2 Shaders

We need to write a new vertex and a new fragment shader. The new vertex shader is due to the fact that we don't use hardcoded vertex data anymore. We need a new vertex shader because we don't use hardcoded vertex data anymore. The new vertex shader will also have to handle the fact that our vertices now store a color value. We need a new fragment shader because we use the vertex color value for our fragment color.

5.2.1 Vertex Shader

Our vertex shader takes as input a position value and a color value. We also want to pass the vertex color to the fragment shader.

```
1  #version 450
2
3  layout(location = 0) in  vec2 inPosition;
4  layout(location = 1) in  vec3 inColor;
5
6  layout(location = 0) out vec3 outColor;
7
8  void main()
9  {
10     gl_Position = vec4(inPosition, 0.0, 1.0);
11     outColor = inColor;
12 }
```

Listing 5.3: Our new vertex shader

5.2.2 Fragment Shader

Our fragment shader now takes as input the fragment's color. We use this value to color our fragment.

```
1  #version 450
2
3  layout(location = 0) in  vec3 inColor;
4
5  layout(location = 0) out vec4 outColor;
6
7  void main()
8  {
9      outColor = vec4(inColor, 1.0);
10 }
```

Listing 5.4: Our new fragment shader

5.3 Upload Vertex Data To The GPU

Before rendering our quad, we must upload its vertex data to the GPU. This data will be later passed as input to the graphics pipeline.

5.3.1 Understanding The Problem

Uploading data to the GPU means copying bytes from RAM to GPU memory. The issue is that we don't know what kind of GPU memory we want to use.

Modern GPUs have different types of memory. Each GPU memory type has also different memory properties. There are two memory properties that interest us. Host visible memory and device local memory.

A host visible memory is a GPU memory that can be mapped to the application's address space. A device local memory is a GPU memory that cannot be mapped to the application's address space.

A host visible memory will always be orders of magnitude slower than a device local one. This is due to the fact that a host visible memory must be visible from both CPU and GPU side. This requires particular care from the driver or from the programmer to keep the data consistent.

Keeping in mind that we don't directly change vertex data at run time, and that we use said data every frame, we would love to use a memory type that is device local. This would improve performance. The problem is that we still need to upload the vertex data to our GPU, and to accomplish this task we can only use a memory that is host visible.

5.3.2 Our Solution: Idea

One solution to this problem is to use two buffers. One buffer, called a staging buffer, will be allocated on host visible GPU memory. We use this staging buffer to upload data to the GPU. The other buffer, called vertex buffer, will be allocated on device local GPU memory. After uploading our data to the staging buffer, we issue a memory transfer command to our GPU. This command, when executed, will copy the staging buffer's contents into the vertex buffer. Later, our vertex buffer will be used by the graphics pipeline for rendering.

5.3.3 How To Create A Buffer

Before implementing our solution we need to know how to cerate a buffer using Vulkan. Buffer creation is divided in three steps. We first create a buffer object. Then, we allocate our buffer memory. Finally, we bind our buffer memory to our buffer object.

Create Buffer Object

The only parameter that can puzzle people is `sharingMode`. This is the buffer's sharing mode when it will be accessed by multiple queue families. Our buffers are only used by the graphics queue. Thus, we use the more performant exclusive sharing mode.

```
1  VkBufferCreateInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
3  info.size = size; // buffer's size in bytes
4  info.usage = usage; // buffer's usage
5  info.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
6
7  VkBuffer buffer = VK_NULL_HANDLE;
8  vkCreateBuffer(device, &info, nullptr, &buffer);
```

Listing 5.5: Create a buffer object

Allocate Buffer Memory

Allocating GPU memory requires some work. This is due to the fact that modern GPUs have many different types of memories. An added complexity is also caused by the fact that we want our buffer memory to satisfy a given set of memory properties (host visible, device local, etc ...).

```
1  // Bitmask specifying the properties that we want
2  // our buffer memory to have
3  VkMemoryPropertyFlags properties = ...;
4
5  // Query the memory requirements for our buffer
6  VkMemoryRequirements memoryRequirements = {};
7  vkGetBufferMemoryRequirements(device, buffer, &memoryRequirements);
8
9  VkMemoryAllocateInfo info = {};
10 info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
11 info.allocationSize = memoryRequirements.size;
12 info.memoryTypeIndex = FindMemoryType(physicalDevice,
       memoryRequirements.memoryTypeBits, properties);
13
14 VkDeviceMemory memory = VK_NULL_HANDLE;
15 vkAllocateMemory(device, &info, nullptr, memory);
```

Listing 5.6: Allocate buffer memory

We first need to query our buffer memory requirements. The query's result contains the set of all memory types that are compatible with our buffer.

With this information at hand, we pick a memory type on which we will allocate our buffer memory. Remember that the memory type we pick must satisfy the memory properties that we require. In order to make this process simpler, we use an auxiliary function called `FindMemoryType`. We use `vkAllocateMemory` to allocate our buffer memory.

```
u32 VexFindMemoryType
(
    VkPhysicalDevice physicalDevice,
    u32 supportedMemoryTypes,
    VkMemoryPropertyFlags requiredMemoryProperties
)
{
    // Get the available types of memory
    VkPhysicalDeviceMemoryProperties memoryProperties = {};
    vkGetPhysicalDeviceMemoryProperties(physicalDevice, &
    memoryProperties);

    // Find a memory type that is supported
    for (u32 i = 0; i < memoryProperties.memoryTypeCount; i++)
    {
        bool isMemoryTypeSupported = (supportedMemoryTypes & (1 <<
    i));
        VkMemoryPropertyFlags memoryTypeProperties =
    memoryProperties.memoryTypes[i].propertyFlags;
        bool areRequiredMemoryPropertiesSupported = ((
    memoryTypeProperties & requiredMemoryProperties) ==
    requiredMemoryProperties);
        if (isMemoryTypeSupported &&
    areRequiredMemoryPropertiesSupported)
        {
            return i;
        }
    }

    assert(false, "Failed to find a suitable memory type");

    return 0;
}
```

Listing 5.7: Find suitable memory type index

Bind Buffer Object And Memory

We use `vkBindBufferMemory` to bind the buffer's object and buffer's memory together.

5.3.4 Our Solution: Implementation

Now that we know how to allocate a buffer, we can see how our solution is implemented.

Create A Staging Buffer

We use the `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` flag because our buffer will be used by the GPU as a source for transfer operations.

We use the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` flag because we want our buffer to be host visible.

We use the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` flag because we don't want to manually flush our buffer memory.

```
1  VkBuffer stagingBufferObject = VK_NULL_HANDLE;
2  VkDeviceMemory stagingBufferMemory = VK_NULL_HANDLE;
3  u32 stagingBufferSizeInBytes = vertexBufferSizeInBytes;
4  CreateBuffer
5  (
6      physicalDevice,
7      device,
8      stagingBufferSizeInBytes,
9      VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
10     VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
       VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
11     &stagingBufferObject,
12     &stagingBufferMemory
13 );
```

Listing 5.8: Crete staging buffer

Upload Vertex Data To The Staging Buffer

Uploading the vertex data to our staging buffer is very simple. We map the staging buffer memory into our application's address space. We copy the data. We unmap the previously mapped memory.

```
1  void* data = nullptr;
2  vkMapMemory(device, stagingBufferMemory, 0,
       stagingBufferSizeInBytes, 0, &data);
3  memcpy(data, vertices, stagingBufferSizeInBytes);
4  vkUnmapMemory(device, stagingBufferMemory);
```

Listing 5.9: Upload our vertex data to the staging buffer

Create The Vertex Buffer

We use the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` flag because our buffer will be used by the GPU as a destination for transfer operations.

We use the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` flag because our buffer will be used by the GPU as a vertex buffer.

We use the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` flag because we want our buffer to be device local.

```
1  VkBuffer vertexBufferObject = VK_NULL_HANDLE;
2  VkDeviceMemory vertexBufferMemory = VK_NULL_HANDLE;
3  u32 vertexBufferSizeInBytes = sizeof(*vertices) * arraysize(
       vertices);
4  VexCreateBuffer
5  (
6      physicalDevice,
7      device,
8      vertexBufferSizeInBytes,
9      VK_BUFFER_USAGE_TRANSFER_DST_BIT |
       VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
10     VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
11     &vertexBufferObject,
12     &vertexBufferMemory
13 );
```

Listing 5.10: Create vertex buffer

Allocate Command Buffer

We allocate a command buffer from our graphics command pool. Is it ok to use the graphics command pool to execute transfer commands? Yes, because all graphics queues always support transfer commands.

```
1  VkCommandBufferAllocateInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
3  info.commandPool = graphicsCommandPool;
4  info.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
5  info.commandBufferCount = 1;
6
7  VkCommandBuffer commandBuffer = VK_NULL_HANDLE;
8  vkAllocateCommandBuffers(device, &info, &commandBuffer);
```

Listing 5.11: Allocate our transfer command buffer

Record Copy Command

Now we record the memory copy command into our command buffer. We are telling our GPU to copy `copyRegion.size` bytes from our staging buffer to our vertex buffer.

```
1  VkCommandBufferBeginInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3  info.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4  vkBeginCommandBuffer(commandBuffer, &info);
5  {
6      VkBufferCopy copyRegion = {};
7      copyRegion.size = vertexBufferSizeInBytes;
8      vkCmdCopyBuffer(commandBuffer, stagingBufferObject,
       vertexBufferObject, 1, &copyRegion);
9  }
10 vkEndCommandBuffer(commandBuffer);
```

Listing 5.12: Record copy command into our command buffer

Submit Command Buffer

Now we have to submit our command buffer to the GPU graphics queue. Doing this will start the execution of our copy command.

```
1  VkSubmitInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
3  info.commandBufferCount = 1;
4  info.pCommandBuffers = &commandBuffer;
5  vkQueueSubmit(graphicsQueue, 1, &info, VK_NULL_HANDLE);
```

Listing 5.13: Submit transfer command buffer

Wait For The Command Buffer Execution To Finish

For simplicity's sake, we wait for our transfer command to finish before continuing the execution of our application. To do this we call `vkQueueWaitIdle` on our graphics queue. This is not a performance issue, since we should be doing this only during the application's setup phase.

Cleanup

We can now free our command buffer using `vkFreeCommandBuffers`. We do this since we won't be using it anymore. We can also destroy our staging buffer using `vkFreeMemory` and `vkDestroyBuffer`.

5.3.5 Review The Process

Here we can see some pseudocode that outlines all the steps necessary to create a vertex buffer.

```
1  void CreateVertexBuffer(...)
2  {
3      // Create staging buffer ...
4      // Upload vertex data to the staging buffer ...
5      // Create vertex buffer ...
6      // Issue a copy command ...
7      // Wait for the copy to finish ...
8      // Cleanup ...
9  }
```

Listing 5.14: Steps for creating our vertex buffer

5.4 Pipeline Vertex Input State

During the creation of our pipeline state object, we must specify the format of the pipeline vertex input data. To do this we use a `VkPipelineVertexInputStateCreateInfo` struct.

```
1  VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
2  vertexInputInfo.sType =
       VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
3  vertexInputInfo.vertexBindingDescriptionCount = arraysize(
       bindingDescriptions);
4  vertexInputInfo.pVertexBindingDescriptions = bindingDescriptions;
5  vertexInputInfo.vertexAttributeDescriptionCount = arraysize(
       attributeDescriptions);
6  vertexInputInfo.pVertexAttributeDescriptions =
       attributeDescriptions;
```

Listing 5.15: Describe the pipeline input data

`pVertexBindingDescriptions` is an array of vertex input binding descriptions. `pVertexAttributeDescriptions` is an array of vertex input attribute descriptions.

5.4.1 Binding Descriptions

A `VkVertexInputBindingDescription` struct has three fields. `binding` is the binding number that this structure describes. `stride` is the number of bytes between consecutive elements within the vertex buffer. `inputRate` is a value that specifies whether vertex attribute addressing is a function of the vertex index or of the instance index.

Some of you may be wondering: what is a vertex input binding? Before recording a draw command into a command buffer, we must bind a vertex buffer in order for our pipeline to use it. We use `vkCmdBindVertexBuffers` to

do this. This function takes an array of buffers. This is because our pipeline can get vertex data from multiple buffers at the same time. `binding` is an index into the `pBuffers` array bound by `vkCmdBindVertexBuffers`.

In our case, all the vertex data is packed together and comes from a single buffer. Hence, we only have one binding description.

```
1  VkVertexInputBindingDescription bindingDescription = {};
2  bindingDescription.binding = 0;
3  bindingDescription.stride = sizeof(Vertex);
4  bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
5
6  VkVertexInputBindingDescription bindingDescriptions[] =
7  {
8      bindingDescription,
9  };
```

Listing 5.16: Describe our vertex input bindings

5.4.2 Attribute Descriptions

A vertex attribute is an input variable that is supplied per-vertex to a shader. In our case, for each vertex, we pass into our shader a position and a color. Hence, for us, a vertex has two attributes: a position attribute and a color attribute.

Before creating our pipeline state object we must describe all our vertex attributes. We do this using an array of `VkVertexInputAttributeDescription` struct instances.

```
1  VkVertexInputAttributeDescription positionAttributeDescription =
       {};
2  positionAttributeDescription.location = 0;
3  positionAttributeDescription.binding = 0;
4  positionAttributeDescription.format = VK_FORMAT_R32G32_SFLOAT;
5  positionAttributeDescription.offset = offsetof(Vertex, position);
6
7  VkVertexInputAttributeDescription colorAttributeDescription = {};
8  colorAttributeDescription.location = 1;
9  colorAttributeDescription.binding = 0;
10 colorAttributeDescription.format = VK_FORMAT_R32G32B32_SFLOAT;
11 colorAttributeDescription.offset = offsetof(Vertex, color);
12
13 VkVertexInputAttributeDescription attributeDescriptions[] =
14 {
15     positionAttributeDescription,
16     colorAttributeDescription,
17 };
```

Listing 5.17: Describe our vertex input attributes

`location` is the shader input location number for this attribute. We have seen this value earlier when we wrote `layout(location = 0)` in our vertex shader. `binding` is the binding number which this attribute takes its data from. `format` is the size and type of the vertex attribute data. `offset` is the byte offset of this attribute relative to the start of an element in the vertex input binding.

5.5 Draw Using Our Vertex Data

The only missing thing now is to tell Vulkan to use our vertex data for rendering operations. We do this during the rendering command buffer recording.

```
1  // Begin render pass ...
2
3  // Bind our pipeline state object ...
4
5  VkBuffer buffers[] = { vertexBufferObject };
6  VkDeviceSize offsets[] = { 0 };
7  vkCmdBindVertexBuffers(commandBuffer, 0, 1, buffers, offsets);
8
9  vkCmdDraw(commandBuffer,arraysize(vertices), 1, 0, 0);
10
11 // End render pass ...
```

Listing 5.18: Draw quad using our vertex data

Chapter 6

Shader Global Data: Uniforms

In this chapter we see how to pass global data to our vertex shader. We can do this using one or more uniforms. We upload uniform data to a vertex shader using a uniform buffer. At the end, we use our uniforms to rotate the quad and render it using a perspective camera.

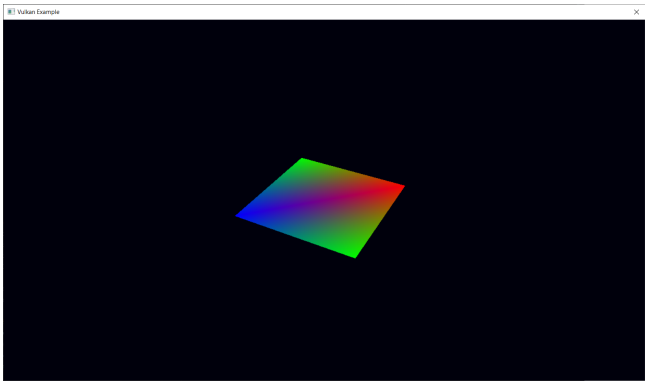

Figure 6.1: Rendering a quad using a perspective camera

6.1 Uniform Data

In this section we define the uniform data that we use to draw our quad.

6.1.1 Uniforms

In our case, we use three uniforms: a model matrix, a view matrix, and a projection matrix.

```
glm::mat4 model;
glm::mat4 view;
glm::mat4 projection;
```

Listing 6.1: Data that will be globally available to our shaders

6.1.2 Uniform Buffer Data

Every frame we update our uniforms as follows.

```
model = glm::rotate(glm::mat4(1.0f), timeSinceStart * glm::radians
    (90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0
    f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f));
f32 aspect = (f32)(swapchainImageExtent.width) / (f32)(
    swapchainImageExtent.height);
projection = glm::perspective(glm::radians(45.0f), aspect, 0.1f,
    10.0f);

// Perspective matrix correction (only for glm)
ubo.projection[1][1] *= -1;
```

Listing 6.2: Updating uniforms during the application's main loop

We use our model matrix to continuously rotate the quad based on the amount of time since the application has started. We use our view matrix to represent a camera with position at coordinates $(2, 2, 2)$, looking at $(0, 0, 0)$ and with $(0, 0, 1)$ as up vector. We use our projection matrix to define the frustum of our camera. Here we use a perspective projection with a field of view of 45 degrees, an aspect ratio based on the swapchain image, and a near and far planes of 0.1 and 10 respectively.

6.1.3 Vertex Shader

We must update the vertex shader for us to use our uniform data. In our case, we use a single uniform buffer containing our uniforms.

```
#version 450

layout(location = 0) in  vec2 inPosition;
layout(location = 1) in  vec3 inColor;

layout(location = 0) out vec3 outColor;

layout(set = 0, binding = 0) uniform UBO
{
    mat4 model;
    mat4 view;
    mat4 projection;
} ubo;

void main()
{
    gl_Position = ubo.projection * ubo.view * ubo.model * vec4(
    inPosition, 0.0, 1.0);
    outColor = inColor;
}
```

Listing 6.3: Vertex shader that uses our uniforms

In the vertex shader, we transform the vertex position using our matrices. This is the usual way in which we change vertex data. Instead of directly modifying vertices, we use one or more matrices that define the transformation we want. This computation is very fast because it's executed concurrently on the GPU.

6.2 Upload Uniform Data To The GPU

Uniform data usually changes very frequently, even on a frame by frame basis. This being the case, every time our uniforms change, we upload them to the GPU to make the change visible to the shaders that use them. We use a buffer to upload uniform data to the GPU. Such buffer is called a uniform buffer.

6.2.1 Uniform Buffer Layout

Before creating a uniform buffer, we declare its layout. In our case, since we have three uniforms, we pack them together inside a uniform buffer.

```
1  struct UBO
2  {
3      glm::mat4 model;
4      glm::mat4 view;
5      glm::mat4 projection;
6  };
```

Listing 6.4: Uniform buffer definition

6.2.2 Uniform Buffer Creation

We use the `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` flag because our buffer will be used by the GPU as a uniform buffer.

We use the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` flag because we want our buffer to be host visible. This is due to the fact that we upload our uniforms to the uniform buffer every frame.

We use the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` flag because we don't want to manually flush our buffer memory.

```
1  VkBuffer uniformBufferObject = VK_NULL_HANDLE;
2  VkDeviceMemory uniformBufferMemory = VK_NULL_HANDLE;
3  u32 uniformBufferSizeInBytes = sizeof(UBO);
4  VexCreateBuffer
5  (
6      physicalDevice,
7      device,
8      uniformBufferSizeInBytes,
9      VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
10     VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
       VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
11     &uniformBufferObject,
12     &uniformBufferMemory
13 );
```

Listing 6.5: Uniform buffer creation

Cleanup

We use `vkFreeMemory` and `vkDestroyBuffer` to clean up a uniform buffer.

6.2.3 Upload Uniform Data

We upload our uniform data simply by writing it into our uniform buffer.

```
1  UBO ubo = {};
2  ubo.model = model;
3  ubo.view = view;
4  ubo.projection = projection;
5
6  void* data = nullptr;
7  vkMapMemory(device, uniformBufferMemory, 0, sizeof(ubo), 0, &data);
8  memcpy(data, &ubo, sizeof(ubo));
9  vkUnmapMemory(device, uniformBufferMemory);
```

Listing 6.6: Upload uniforms to the uniform buffer

6.2.4 Uniform Buffer Data Alignment

Vulkan expects the data in our uniform buffer to be aligned in memory in a specific way. You can find the full list of alignment requirements in the Vulkan specification. Here follows a brief list of the most important requirements.

- Scalars have to be aligned by N (for example, 4 bytes for `f32` values)
- A `vec2` must be aligned by $2N$
- A `vec3` or `vec4` must be aligned by $4N$
- A structure must be aligned by the base alignment of its members rounded up to a multiple of 16
- A `mat4` matrix must have the same alignment as a `vec4`

6.3 Update Pipeline Layout

We must inform our pipeline that we are using one or more uniforms. To do this we update our pipeline layout.

6.3.1 VkPipelineLayout

A pipeline layout can be seen as an interface between shader stages and shader resources as it takes these groups of resources, describes how they are gathered, and provides them to the pipeline.

```
1  VkDescriptorSetLayout descriptorSetLayouts[] =
2  {
3      pipelineDescriptorSetLayout,
4  };
5
6  VkPipelineLayoutCreateInfo info = {};
7  info.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
8  info.setLayoutCount = arraysize(descriptorSetLayouts);
9  info.pSetLayouts = descriptorSetLayouts;
10
11 VkPipelineLayout pipelineLayout = VK_NULL_HANDLE;
12 vkCreatePipelineLayout(device, &info, nullptr, &pipelineLayout);
```

Listing 6.7: Update pipeline layout creation

6.3.2 Descriptor Set Layout

We use a `VkDescriptorSetLayout` object to tell the number and the types of global resources that are available to our pipeline shaders.

```
1  VkDescriptorSetLayout pipelineDescriptorSetLayout = VK_NULL_HANDLE;
2  vkCreateDescriptorSetLayout(device, &info, nullptr, &
       pipelineDescriptorSetLayout));
```

Listing 6.8: Describe pipeline global resources

VkDescriptorSetLayoutCreateInfo

We use a `VkDescriptorSetLayoutCreateInfo` struct to configure the descriptor set layout we are about to create.

```
1  VkDescriptorSetLayoutCreateInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
3  info.bindingCount = arraysize(bindings);
4  info.pBindings = bindings;
```

Listing 6.9: Descriptor set layout configuration

Descriptor Set Layout Bindings

We use a `VkDescriptorSetLayoutBinding` to describe, for a given type, how many resources are globally available to our pipeline. In our application we use only one uniform buffer accessed from the vertex shader.

```
1  VkDescriptorSetLayoutBinding uniformBufferBinding = {};
2  uniformBufferBinding.binding = 0;
3  uniformBufferBinding.descriptorType =
       VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
4  uniformBufferBinding.descriptorCount = 1;
5  uniformBufferBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
6
7  VkDescriptorSetLayoutBinding bindings[] =
8  {
9      uniformBufferBinding
10 };
```

Listing 6.10: Descriptor set layout bindings

`binding` is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages. We have seen this binding number in our vertex shader when we wrote `layout(set = 0, binding = 0) uniform UBO` to access the uniform buffer. `descriptorType` is a `VkDescriptorType` specifying which type of resource descriptors are used for this binding. `descriptorCount` is the number of resources contained in the binding. `stageFlags` is a bitmask of `VkShaderStageFlagBits` specifying which pipeline shader stages can access a resource for this binding

Cleanup

We use `vkDestroyDescriptorSetLayout` do destroy a descriptor set layout.

6.4 Descriptor Set

A descriptor set is an object that contains all the physical resources that are globally available to a set of pipeline shader stages. We allocate a descriptor set from a descriptor pool. Then we populate the descriptor set with one or more resources.

6.4.1 Descriptor Set Allocation

We allocate a descriptor set with `vkAllocateDescriptorSets`.

```
1  VkDescriptorSet pipelineDescriptorSet = VK_NULL_HANDLE;
2  vkAllocateDescriptorSets(device, &info, &pipelineDescriptorSet);
```

Listing 6.11: Allocate a descriptor set

6.4.2 VkDescriptorSetAllocateInfo

We use a `VkDescriptorSetAllocateInfo` struct to configure the descriptor set we are about to create.

```
1  VkDescriptorSetLayout setLayouts[] =
2  {
3      pipelineDescriptorSetLayout,
4  };
5
6  VkDescriptorSetAllocateInfo info = {};
7  info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
8  info.descriptorPool = pipelineDescriptorPool;
9  info.descriptorSetCount = arraysize(setLayouts);
10 info.pSetLayouts = setLayouts;
```

Listing 6.12: Configure descriptor set

`descriptorPool` is the pool which the sets will be allocated from. `descriptorSetCount` determines the number of descriptor sets to be allocated from the pool. `pSetLayouts` is a pointer to an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

6.5 Descriptor Pool

A descriptor set must be allocated from a descriptor pool. Thus, we must create a descriptor pool before allocating a descriptor set.

```
1  VkDescriptorPool pipelineDescriptorPool = VK_NULL_HANDLE;
2  vkCreateDescriptorPool(device, &info, nullptr, &
       pipelineDescriptorPool);
```

Listing 6.13: Create descriptor pool

6.5.1 VkDescriptorPoolCreateInfo

We use a `VkDescriptorPoolCreateInfo` struct to configure the descriptor pool we are about to create. In our case, since we use only one uniform buffer, we create a descriptor pool from which we can allocate only one descriptor set with one uniform buffer resource.

```
1  VkDescriptorPoolSize uniformBufferPoolSize = {};
2  uniformBufferPoolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
3  uniformBufferPoolSize.descriptorCount = 1;
4
5  VkDescriptorPoolSize poolSizes[] =
6  {
7      uniformBufferPoolSize,
8  };
9
10 VkDescriptorPoolCreateInfo info = {};
11 info.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
12 info.maxSets = 1;
13 info.poolSizeCount = arraysize(poolSizes);
14 info.pPoolSizes = poolSizes;
```

Listing 6.14: Configure descriptor poll creation

`maxSets` is the maximum number of descriptor sets that can be allocated from the pool. `pPoolSizes` is a pointer to an array of `VkDescriptorPoolSize` structures, each containing a descriptor type and the number of resources of that type that will be allocated in total from the pool.

6.5.2 Populate Descriptor Set

Once we have allocated a descriptor set, we have to populate it with resources. This means writing into the descriptor set. We use `vkUpdateDescriptorSets` for this task.

```
1  VkWriteDescriptorSet descriptorWrites[] =
2  {
3      descriptorWrite,
4  };
5
6  vkUpdateDescriptorSets(device, arraysize(descriptorWrites),
       descriptorWrites, 0, nullptr);
```

Listing 6.15: Populate descriptor set

6.5.3 VkWriteDescriptorSet

We use a `VkWriteDescriptorSet` struct to configure the descriptor write operations that we are about to execute. Here we are telling our descriptor set to use our uniform buffer as a uniform buffer resource.

```
VkDescriptorBufferInfo info = {};
info.buffer = uniformBufferObject;
info.offset = 0;
info.range = uniformBufferSizeInBytes;

VkWriteDescriptorSet descriptorWrite = {};
descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrite.dstSet = pipelineDescriptorSet;
descriptorWrite.dstBinding = 0;
descriptorWrite.dstArrayElement = 0;
descriptorWrite.descriptorCount = 1;
descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrite.pBufferInfo = &info;
```

Listing 6.16: Descriptor set write

6.5.4 Cleanup

We use `vkDestroyDescriptorPool` to destroy a descriptor pool. All descriptor sets allocated from the pool will be automatically freed when the pool is destroyed.

6.6 Draw Using Our Uniform Data

The only thing missing now is to tell Vulkan to use our uniforms during rendering. To accomplish this, we tell Vulkan to use the resources that are inside our descriptor set for rendering. We do this during our command buffer recording.

```
// Begin render pass ...

// Bind pipeline state object ...
// Bind vertex buffer ...

vkCmdBindDescriptorSets
(
    commandBuffer,
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipelineLayout,
    0,
    1,
    &pipelineDescriptorSet,
    0,
    nullptr
);

// Draw ...

// End render pass ...
```

Listing 6.17: Draw quad using our uniform data

Chapter 7

Depth Testing

Suppose we are drawing multiple objects. To correctly render them on the screen we have to worry about the order in which we execute our draw operations. In particular, to get a realistic image, we must draw them in descending order based on their distance from the camera. We do this to give to objects near the camera the possibility of hiding objects further away.

Although this solution works, it would require us to sort every object that is going to be rendered using its depth. If there are moving objects, we may have to sort them every frame. If there are a lot of them, the sorting may require a considerable amount of time, decreasing the application's framerate. This solution also causes problems when two objects overlap. In this case, it's not possible to determine which one to render first without having graphical artifacts.

An alternative solution that doesn't suffer from the drawbacks illustrated earlier, is using a depth buffer. A depth buffer is an image that stores depth data for every pixel. 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.

Mind that this technique works only when rendering opaque objects. When rendering semitransparent or transparent objects we would need to resort to depth sorting to produce the correct image.

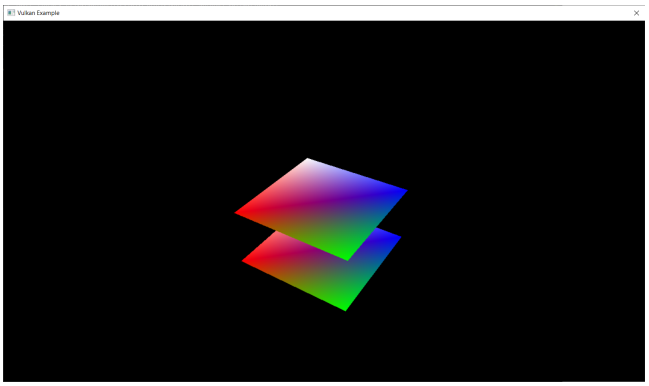

Figure 7.1: Rendering two quads using a depth buffer

7.1 Creating A Depth Buffer

Creating a depth buffer means creating an image that will be used to store depth values during rendering operations.

7.1.1 Creating An Image

Crating an image in Vulkan consists of four steps. We first create a `VkImage` object. Then, we allocate some memory for our `VkImage`. We do this creating a `VkDeviceMemory` object. Once we have both the image object and memory, we bind them together using `vkBindImageMemory`. Finally, in order to use the image we have created, we create a `VkImageView` on it.

VkImage

To create a `VkImage` we use `vkCreateImage`. Here we are creating a 2D image of a given `width` and `height`. We also specify the image `format` and `tiling`. These two values define the data that the image stores per pixel and how it's layed out in memory. Finally, we also specify the image's `usage`. The images we create will always be used only by our graphics queue. Thus, we use the more performant exclusive sharing mode.

```
1  VkImageCreateInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
3  info.imageType = VK_IMAGE_TYPE_2D;
4  info.format = format;
5  info.extent = { width, height, 1 };
6  info.mipLevels = 1;
7  info.arrayLayers = 1;
8  info.samples = VK_SAMPLE_COUNT_1_BIT;
9  info.tiling = tiling;
10 info.usage = usage;
11 info.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
12 info.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
13
14 VkImage imageObject = VK_NULL_HANDLE;
15 vkCreateImage(device, &info, nullptr, &imageObject);
```

Listing 7.1: Create image object

`imageType` tells Vulkan with what kind of coordinate system the pixels in the image are going to be addressed. `tiling` can have one of two values: linear or optimal. With linear tiling, pixels are laid out in row-major order. With optimal tiling, pixels are laid out in an implementation defined order for optimal access. `usage` is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the image.

VkDeviceMemory

We allocate some memory for a image object using `vkAllocateMemory`. Allocating image memory is the same as allocating buffer memory.

```
1  VkMemoryRequirements memoryRequirements = {};
2  vkGetImageMemoryRequirements(device, imageObject, &
       memoryRequirements);
3
4  VkMemoryAllocateInfo info = {};
5  info.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
6  info.allocationSize = memoryRequirements.size;
7  info.memoryTypeIndex = FindMemoryType(physicalDevice,
       memoryRequirements.memoryTypeBits, properties);
8
9  VkDeviceMemory imageMemory = VK_NULL_HANDLE;
10 vkAllocateMemory(device, &info, nullptr, &imageMemory);
```

Listing 7.2: Allocate image memory

VkImageView

We create an image view using `vkCreateImageView`. Here, we create a 2D view on an image object, specifying how to read its pixel data and which image aspects are included in the view.

```
1  VkImageViewCreateInfo info = {};
2  info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
3  info.image = imageObject;
4  info.viewType = VK_IMAGE_VIEW_TYPE_2D;
5  info.format = format;
6  info.subresourceRange.aspectMask = aspectMask;
7  info.subresourceRange.baseMipLevel = 0;
8  info.subresourceRange.levelCount = 1;
9  info.subresourceRange.baseArrayLayer = 0;
10 info.subresourceRange.layerCount = 1;
11
12 VkImageView imageView = VK_NULL_HANDLE;
13 vkCreateImageView(device, &info, nullptr, &imageView);
```

Listing 7.3: Create image View

`format` is the format and type used to interpret pixels in the image. `aspectMask` is a bitmask of `VkImageAspectFlagBits` specifying which aspect(s) of the image are included in the view.

7.1.2 Cleanup

We destroy a image view with `vkDestroyImageView`. Then we call `vkDestroyImage` and `vkFreeMemory` to clean up the resources we acquired to create an image.

7.1.3 Depth Image Creation

Now that we know how to create an image, we can create our depth image. A depth image should have the same resolution as our swapchain images. It must have an appropriate image usage to be used as a depth buffer. It should use optimal tiling and have device local memory, supporting fast read and write operations. The image should have one of the following formats:

- `VK_FORMAT_D32_SFLOAT` using 32 bits for depth data
- `VK_FORMAT_D32_SFLOAT_S8_UINT` using 32 bits for depth data and 8 for stencil data
- `VK_FORMAT_D24_UNORM_S8_UINT` using 24 bits for depth data and 8 for stencil data

The most commonly used depth format is `VK_FORMAT_D32_SFLOAT`.

```
VexCreateImage
(
    physicalDevice,
    device,
    width,
    height,
    VK_FORMAT_D32_SFLOAT,
    VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
    &depthBufferImage,
    &depthBufferMemory
);

VexCreateImageView(
    device,
    depthBufferImage,
    depthBufferFormat,
    VK_IMAGE_ASPECT_DEPTH_BIT,
    &depthBufferView
);
```

Listing 7.4: Create Depth Image

7.2 Depth Image Render Pass Attachment

To use our depth buffer we must add it to our render pass as a depth attachment.

7.2.1 VkAttachmentDescription

Here we describe our depth buffer attachment. The format should be the same as the depth image itself. We want to clear it before using it for the first time. We don't care about storing the depth data, because it will not be used after drawing has finished. This may allow the driver to perform additional optimizations. When our render pass instance begins, we don't care about its previous contents. Right before finishing our render pass instance, we want its layout to be optimal for depth stencil operations.

```
VkAttachmentDescription depthAttachment = {};
depthAttachment.format = VK_FORMAT_D32_SFLOAT;
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;
```

Listing 7.5: Depth buffer render pass attachment description

7.2.2 Attachment descriptions

Now that we have a new attachment description, we must add it to our render pass create info attachments array.

```
1  VkAttachmentDescription attachments[] =
2  {
3      colorAttachment,
4      depthAttachment,
5  };
```

Listing 7.6: New render pass attachments array

7.2.3 Render Pass Subpasses

Inside our render pass we still have a single subpass. But, unlike before, not only we have a color attachment, but we also have a depth stencil attachment.

Here we are saing that our depth image will be used as the subpass depth stencil attachment. We are also telling Vulkan to transition our depth buffer image layout to `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` during this subpass. We do this because depth operations will be executed on the depth image.

```
1  VkAttachmentReference depthAttachmentReference = {};
2  depthAttachmentReference.attachment = 1;
3  depthAttachmentReference.layout =
       VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
4
5  VkSubpassDescription colorSubpass = {};
6  colorSubpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
7  colorSubpass.colorAttachmentCount = arraysize(
       colorAttachmentReferences);
8  colorSubpass.pColorAttachments = colorAttachmentReferences;
9  colorSubpass.pDepthStencilAttachment = &depthAttachmentReference;
```

Listing 7.7: New render pass subpass

7.3 Pipeline Depth Stencil State

We need to enable depth testing in our pipeline state object. This is configured through a `VkPipelineDepthStencilStateCreateInfo` struct.

Here, we are enabling depth testing. We are also telling Vulkan to write the values that pass the depth test into the depth buffer. We also specify the comparison operation that decides which fragments to keep and which to discard. We are sticking to the convention that lower depth means closer to the camera. Thus, in order for a fragment to pass the depth test, its depth value should be less than the depth value stored into the respective depth image pixel.

```
1  VkPipelineDepthStencilStateCreateInfo depthStencilState = {};
2  depthStencilState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
3  depthStencilState.depthTestEnable = VK_TRUE;
4  depthStencilState.depthWriteEnable = VK_TRUE;
5  depthStencilState.depthCompareOp = VK_COMPARE_OP_LESS;
```

Listing 7.8: Configure pipeline state depth testing

7.4 Depth Image Framebuffer Attachment

Before creating a framebuffer, we add our depth image to the framebuffer create info attachments array.

```
1  VkImageView attachments[] =
2  {
3      nextSwapchainImageView,
4      depthBufferView,
5  };
6
7  VkFramebufferCreateInfo createInfo = {};
8  createInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
9  createInfo.renderPass = renderPass;
10 createInfo.attachmentCount = arraysize(attachments);
11 createInfo.pAttachments = attachments;
12 createInfo.width = swapchainImageExtent.width;
13 createInfo.height = swapchainImageExtent.height;
14 createInfo.layers = 1;
```

Listing 7.9: Modify framebuffer creation

7.5 Render Pass Clear Values

Because we now have multiple attachments with `VK_ATTACHMENT_LOAD_OP_CLEAR`, we also need to specify multiple clear values when we begin our render pass instance. We clear our color attachment with an opaque black color. We clear our depth attachment with a depth value of 1.0.

```
1  VkClearValue clearValues[2] = {};
2  clearValues[0].color = { 0.0f, 0.0f, 0.0f, 1.0f };
3  clearValues[1].depthStencil = { 1.0f, 0 };
4
5  VkRenderPassBeginInfo renderPassBeginInfo = {};
6  renderPassBeginInfo.sType =
       VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
7  renderPassBeginInfo.renderPass = renderPass;
8  renderPassBeginInfo.framebuffer = framebuffer;
9  renderPassBeginInfo.renderArea.offset = { 0, 0 };
10 renderPassBeginInfo.renderArea.extent = context.
       swapchainImageExtent;
11 renderPassBeginInfo.clearValueCount = arraysize(clearValues);
12 renderPassBeginInfo.pClearValues = clearValues;
```

Listing 7.10: Render pass clear values

Depth values range between 0.0 and 1.0. A depth value of 1.0 means that the fragment is the furthest away from the camera. A depth value of 0.0 means that the fragment is the nearest to the camera.

Chapter 8

Setting Up A Scene

In this chapter we see how to lay out different objects into a virtual world, i.e. a scene. To do this, we introduce the concept of entity. At the end of the chapter, we see how to set up and render a simple scene.

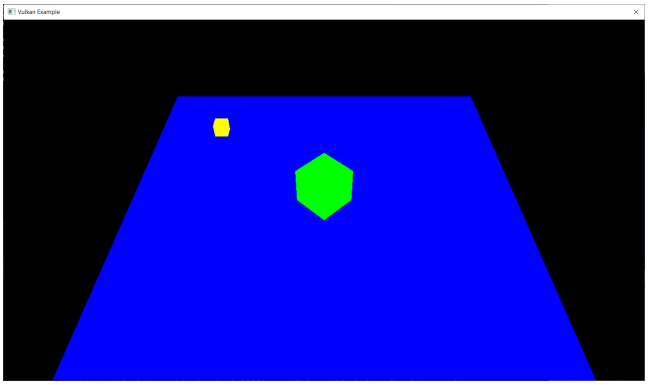

Figure 8.1: Define and render a simple scene

8.1 Why Do We Need Entities?

Suppose we want to render two squares like we did here 7.1. We could, for example, use the following vertex data.

```
1  // First quad
2  { { -0.5f, -0.5f, +0.0f }, },
3  { { +0.5f, -0.5f, +0.0f }, },
4  { { +0.5f, +0.5f, +0.0f }, },
5  { { +0.5f, +0.5f, +0.0f }, },
6  { { -0.5f, +0.5f, +0.0f }, },
7  { { -0.5f, -0.5f, +0.0f }, },
8
9  // Second quad
10 { { -0.5f, -0.5f, -0.5f }, },
11 { { +0.5f, -0.5f, -0.5f }, },
12 { { +0.5f, +0.5f, -0.5f }, },
13 { { +0.5f, +0.5f, -0.5f }, },
14 { { -0.5f, +0.5f, -0.5f }, },
15 { { -0.5f, -0.5f, -0.5f }, },
```

Listing 8.1: Vertices for drawing two squares

This is what we have done in all previous chapters. Although this works, it's obviously not a flexible solution. Those of you with a keen eye have surely noticed that our squares almost have the same vertices. The only difference being in the z coordinates. Drawing two squares means drawing two instances of the same geometric data. Why would we need to repeat the same data twice, just with some slight variations? There is no reason to do it. We can use matrices to define the transformations we want to apply to each object. For one square we could use a matrix that doesn't apply any transformation. For the other square we could use a matrix that translates the vertices downwards.

There is a catch. Now, our objects are not only defined by their vertex data. They also have a position, a rotation, etc. We also observe that, if possible, our objects can share the same vertex data. We should do this to avoid redundancy and to make our program more efficient. We introduce the concept of entity to deal with this situation.

8.2 Entity

An entity is simply a collection of all the data that is necessary to place an object in a scene and to draw it accordingly.

8.2.1 Entity Positional Data

We now have an idea of the nature of entities. We can start by defining all the data necessary to place an entity into a scene. Let's start with an example. We can have a cube placed at $(0, 0, 0)$, the origin of our scene. We could place another cube at $(5, 3, 0)$ and rotate it by 30 degrees around the x axis and by 60 degrees around the z axis. We could also place another cube around the scene and scale it by a factor of 10 to make it bigger.

From this example, we can see that our entities have three main properties that define how the entity is contextualized inside a scene:

- a 3d vector that represents its position inside the scene
- a 3d vector that represents its rotations around the x, y and z axis
- a scalar that represents its scale

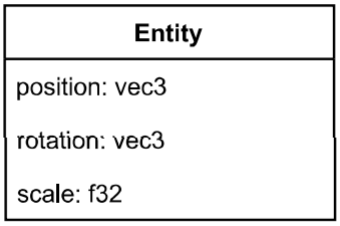

Figure 8.2: Entity positional data

8.2.2 Entity Rendering Data

In our previous example, we have talked about placing cubes around a scene. Earlier we have discussed that entities should share their vertex data if possible. If two or more entities are cubes, they should share the same vertex data. In our case, sharing vertex data means using the same vertex buffer. We know that a vertex buffer is used in tandem with a pipeline state object for drawing operations. Because of this, sharing a vertex buffer also means sharing a pipeline state object. Together with the pipeline state object we should also store the pipeline's layout.

To transform our vertices from local space to world space, we use a model matrix. We also have to use a both a view and projection matrix to transform our vertices from world space to clip space. We have seen how these three matrices are passed to the vertex shader as uniforms. Hence, our entity also needs to use a uniform buffer. Contrary to the other rendering resources, we must create a uniform for every entity. This is because the entity's uniform data refers to the entity itself and is not shared with other entities. Together with the uniform buffer we should store a pipeline descriptor set that contains our uniform buffer resource.

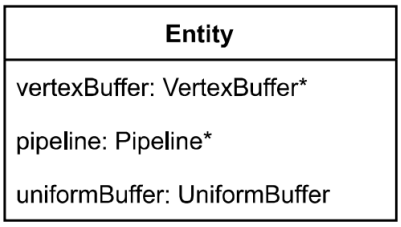

Figure 8.3: Entity rendering data

8.2.3 Other Entity Data

We have discussed all the data that it's always necessary for our entities. Obviously, we can later add more data to entities depending on our needs. We will

see multiple instances of this in the future.

8.2.4 Updating Entities

Some of our entities are not static. This means that their properties change over time. They could rotate, or move along one axis, for example. The important thing to note here is that, if we want to use our updated values in our rendering operations, we must remember to upload the relevant data into the entity's uniform buffer.

```
1  Entity cube = ...;
2
3  // Update cube properties ...
4  cube.position = newPosition;
5  cube.rotation = newRotation;
6  cube.scale = newScale;
7
8  // Update uniform buffer data ...
9  cube.ubo.model = ComputeModelMatrix(&cube);
10 cube.ubo.view = ComputeViewMatrix(&camera);
11 cube.ubo.projection = ComputeProjectionMatrix(&camera);
12
13 // Update uniform buffer ...
14 // Use vkMapMemory, memcpy and vkUnmapMemory ...
```

Listing 8.2: Update entity data and uniform buffer

8.2.5 Rendering Entities

Here we can see how to use the rendering data stored in our entities to actually draw them on the screen.

```
1  Entity cube = ...;
2
3  // Bind cube pipeline object ...
4  // Bind cube vertex data ...
5  // Bind uniform buffer descriptor set ...
6  // Draw ...
```

Listing 8.3: Render entity

8.3 Camera

We have seen that our entities's view and a projection matrices are computed from a `camera`. This is a bundle of all the necessary data that is used to represent a virtual camera inside our scene. We need a camera in order to define the point of view from which we observe the scene. We could consider our camera as the eyes through which we can glimpse at our virtual world.

8.3.1 Camera Data

In our case, we use a perspective camera. This adds perspective to the scene, making the rendered image more realistic. In order to compute a view matrix, our camera must have:

- a position, like our entities

- a target, i.e. a 3d vector identifying a point that the camera is looking at
- an up vector, i.e. a 3d vector that defines the camera's up direction

In order to compute a projection matrix, we must define our camera's frustum. We do this using the following values:

- a scalar value representing the camera's field of view
- a scalar value representing the camera's aspect ratio
- a scalar value representing a near plane
- a scalar value representing a far plane

All the entities that fall within the frustum will be rendered. All the other entities will be clipped, i.e. discarded and won't be renderer.

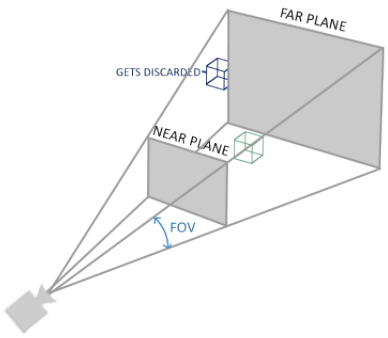

Figure 8.4: Perspective camera frustum

Camera
position: vec3
target: vec3
up: vec3
fov: f32
aspect: f32
nearPlane: f32
farPlane: f32

Figure 8.5: Perspective camera data

8.4 Setting Up A Simple Scene

In this section we set up a simple scene. We do this to see in action the concepts we discussed so far.

8.4.1 Scene Entities

Our scene is made up by three entities. A floor, a cube at the center of the floor, and another cube floating in the air. The floating cube is supposed to represent a light, but here obviously we haven't implemented lighting yet. Don't worry about it for now, we will see how to improve our scene adding lighting to it later.

```
1  Entity floor = {};
2  floor.position = glm::vec3(0.0f, 0.0f, 0.0f);
3  floor.rotation = glm::vec3(0.0f, 0.0f, 0.0f);
4  floor.scale = 10.0f;
5  floor.color = glm::vec3(0.0f, 0.0f, 1.0f);
6  floor.vertexBuffer = &quadVertexBuffer;
7  floor.pipeline = &defaultPipeline;
8  floor.uniformBuffer = CreateUniformBuffer(floor.pipeline.
       descriptorSetLayout);
```

Listing 8.4: FloorEntity

```
1  VexEntity cube = {};
2  cube.position = glm::vec3(0.0f, 0.0f, 0.5f);
3  cube.rotation = glm::vec3(0.0f, 0.0f, VEX_PI / 4.0f);
4  cube.scale = 1.0f;
5  cube.color = glm::vec3(0.0f, 1.0f, 0.0f);
6  cube.vertexBuffer = &cubeVertexBuffer;
7  cube.pipeline = &defaultPipeline;
8  cube.uniformBuffer = CreateUniformBuffer(cube.pipeline.
       descriptorSetLayout);
```

Listing 8.5: Cube entity

```
1  VexEntity light = {};
2  light.position = glm::vec3(2.0f, 1.0f, 3.0f);
3  light.rotation = glm::vec3(0.0f, 0.0f, 0.0f);
4  light.scale = 0.25f;
5  light.color = glm::vec3(1.0f, 1.0f, 0.0f);
6  light.vertexBuffer = &cubeVertexBuffer;
7  light.pipeline = &defaultPipeline;
8  light.uniformBuffer = CreateUniformBuffer(light.pipeline.
       descriptorSetLayout)
```

Listing 8.6: Light entity

8.4.2 Rendering Data

As you can see, the cube and the light entity share the same vertex data. Obviously, our floor takes its vertex data from a different vertex buffer. You can also observe that all three entities use the same pipeline state object. This is because we use the same pipeline configuration to render all of them. This pipeline state object is a slight variation of the one we have seen in previous chapters.

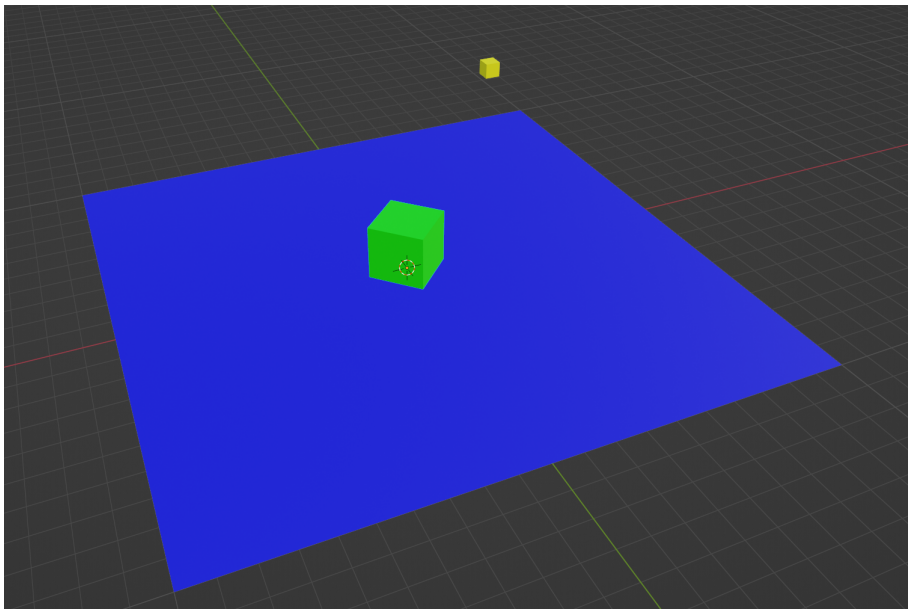

Figure 8.6: Our scene seen from a modeling software

```
#version 450

layout(location = 0) in vec3 inPosition;

layout(location = 0) out vec3 outColor;

layout(set = 0, binding = 0) uniform UBO
{
    mat4 model;
    mat4 view;
    mat4 projection;
    vec3 color;
} ubo;

void main()
{
    gl_Position = ubo.projection * ubo.view * ubo.model * vec4(
    inPosition, 1.0);
    outColor = ubo.color;
}
```

Listing 8.7: Default pipeline vertex shader

```
#version 450

layout(location = 0) in vec3 inColor;

layout(location = 0) out vec4 outColor;

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

Listing 8.8: Default pipeline fragment shader

Our vertex data has only one attribute, a 3d position. Inside our vertex shader we take as uniforms the model, view and projection matrices. Nothing strange here. Note that we also pass a color value as a uniform. This represents the entity's color. We transform our vertex data using our matrices. We also pass to the fragment shader the entity's color. This will be the color used by the produced fragments.

8.4.3 Camera

We have defined our entities. Now we define our camera. Our camera is located at $(0, 8, 8)$. It's looking at our cube: the center of the scene. We use the positive z axis as up vector. We use a field of view of 45 degrees because it's a common value for 3d games. We also use the window's aspect ratio.

```
Camera camera = {};
camera.position = glm::vec3(0.0f, 8.0f, 8.0f);
camera.target = cube.position;
camera.up = glm::vec3(0.0f, 0.0f, 1.0f);
camera.fov = glm::radians(45.0f);
camera.aspect = (f32)(WINDOW_WIDTH) / (f32)(WINDOW_HEIGHT);
camera.nearPlane = 0.01f;
camera.farPlane = 100.0f;
```

Listing 8.9: Scene camera setup

8.5 Our Application So Far

We have set up our simple scene. During our application's startup phase, we initialize all the scene data. In our case we set up the floor, cube and light entities. We then set up the camera. Then, during the application's main loop, we update all entities and render them. In our case all the entities are static, hence we don't need to actually update them. We can render the entities in any order we want because we are using a depth buffer.

```
1  int main(void)
2  {
3      // Setup floor entity ...
4      // Setup cube entity ...
5      // Setup light entity ...
6      // Setup camera ...
7
8      while (isApplicationRunning)
9      {
10         // Update entities ...
11         // Render entities ...
12     }
13 }
```

Listing 8.10: Scene application

Chapter 9

Blinn-Phong Lighting

In this chapter we add lighting to the scene we previously set up. We first implement the Blinn-Phong lighting model. After that, we refine its behavior adding materials to our objects and lights.

9.1 Vulkan Related Details

Before starting to implement the lighting model, we must clarify some technical things related to Vulkan.

9.1.1 Pipeline State Objects

The scene's entities can be divided into two groups: the entities to which we apply lighting, i.e. the floor and the cube; and the entities to which we don't apply lighting, i.e. the light source itself. This means that we must use two sets of shaders: one that implements the Blinn-Phong lighting model, and the other that simply draws a flat color. For this reason, we need to create two different pipeline state objects.

9.1.2 Updating Our Vertex Data

For each fragment, the lighting model needs to know the surface normal. To solve this problem, we store, for each vertex, a normal vector. Here we can see a visualization of the quad and cube vertex normals.

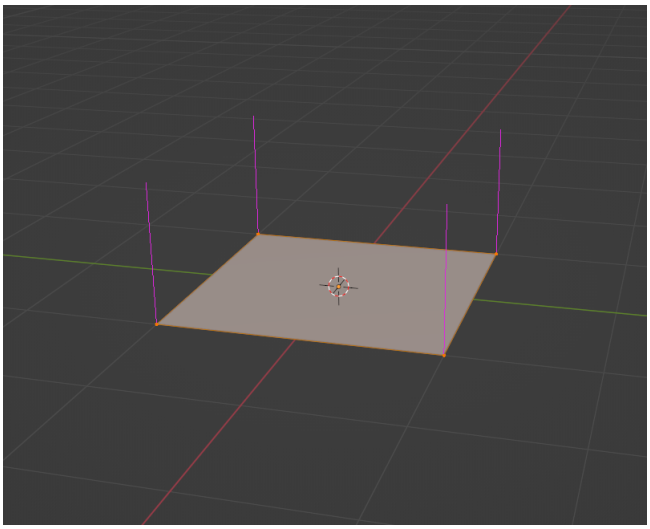

Figure 9.1: Quad vertex normals visualization

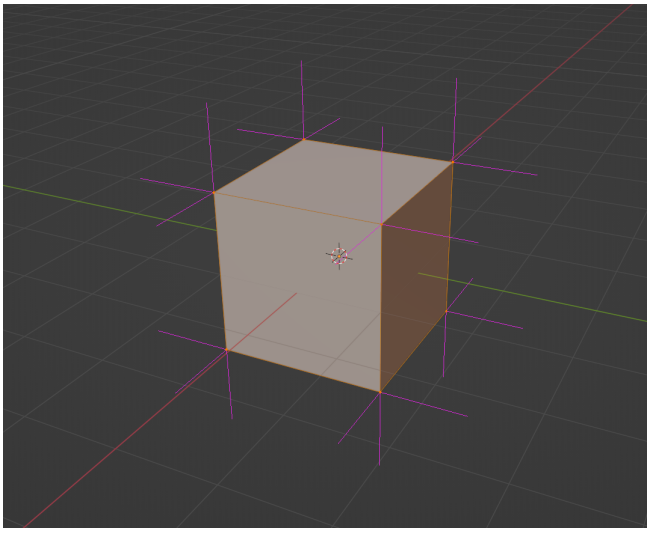

Figure 9.2: Cube vertex normals visualization

9.2 Blinn-Phong Lighting Model

In computer graphics, we approximate lighting using simplified models. The lighting model we implement here is called Blinn-Phong. This model divides light into three components: ambient light, diffuse light and specular light.

9.2.1 Ambient Lighting

In the real world, even when there is no apparent light source, objects aren't completely dark. This is because light can come from different sources around

us, even if they are not directly visible. Indeed, light scatters and bounces in different directions. Thus, some light sources can indirectly light our objects. To simulate this light property, we use a small constant light value that we add to our objects' lighting.

```
1  // Compute diffuse component
2  float ambientStrength = 0.1;
3  vec3 ambient = ambientStrength * inLightColor;
4
5  vec3 result = ambient * inObjectColor;
6  outFragmentColor = vec4(result, 1.0);
```

Listing 9.1: Computing ambient component

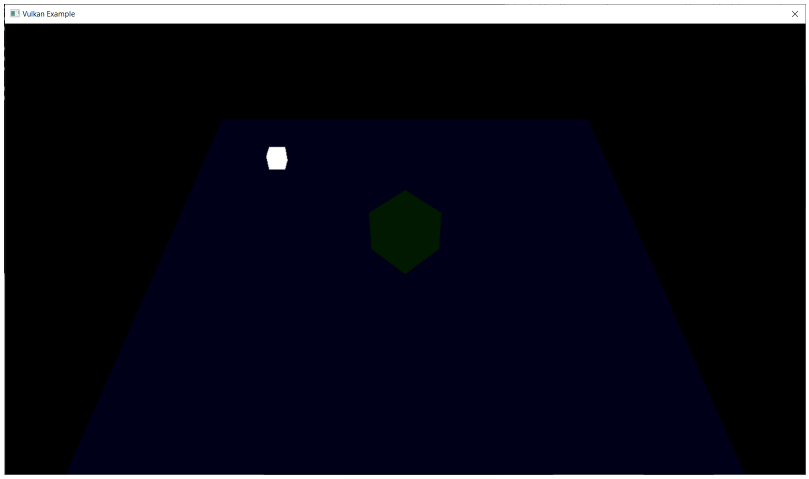

Figure 9.3: Scene with ambient lighting

9.2.2 Diffuse Lighting

Diffuse lighting simulates the impact a light has on an opaque object. In simple terms, the more a part of the object faces the light, the brighter it becomes. The diffuse impact is the strongest when the angle between the surface's normal and the light ray is zero. The diffuse impact will be zero when the angle is greater than or equal to ninety degrees.

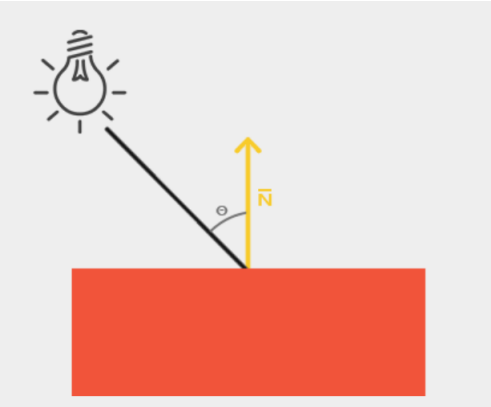

Figure 9.4: Diffuse lighting

```
1  // Compute diffuse component
2  vec3 normal = normalize(inNormal);
3  vec3 lightDirection = normalize(inLightPosition -
       inFragmentPosition);
4  float diffuseImpact = max(dot(normal, lightDirection), 0.0);
5  vec3 diffuse = diffuseImpact * inLightColor;
6
7  vec3 result = diffuse * inObjectColor;
8  outFragmentColor = vec4(result, 1.0);
```

Listing 9.2: Computing diffuse component

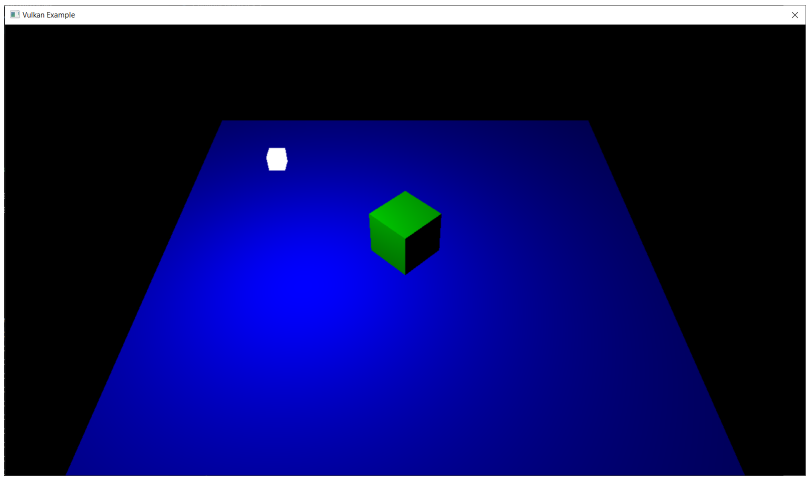

Figure 9.5: Scene with diffuse lighting

9.2.3 Specular Lighting

Specular lighting simulates the bright spot that lights cause on shiny objects. This spot is called specular highlight. The specular impact is the strongest when our view direction is perfectly aligned with the light ray that is reflected off the

object's surface. The more our view deviates form the reflected vector, the less the specular impact will be.

To compute the specular impact we first compute a unit vector exactly halfway between the view direction and the light direction. The closer this halfway vector aligns with the surface's normal, the higher the specular impact.

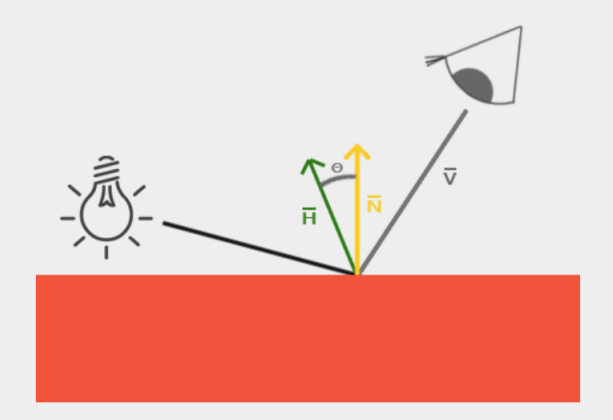

Figure 9.6: Specular lighting

```
1  float specularStrength = 0.5;
2  float shininess = 64;
3  vec3 cameraDirection = normalize(inCameraPosition -
       inFragmentPosition);
4  vec3 halfwayDirection = normalize(lightDirection + cameraDirection)
       ;
5  float specularImpact = pow(max(dot(normal, halfwayDirection), 0.0),
        shininess);
6  vec3 specular = specularImpact * inLightColor;
7
8  vec3 result = specular * inObjectColor;
9  outFragmentColor = vec4(result, 1.0);
```

Listing 9.3: Computing specular component

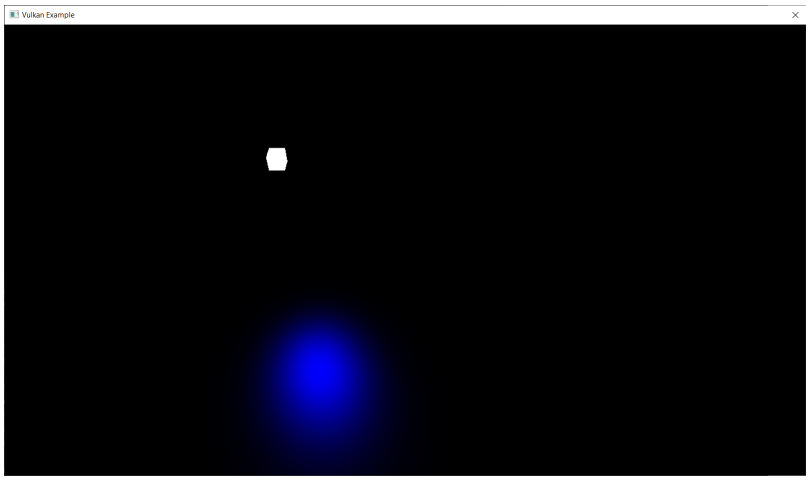

Figure 9.7: Scene with specular lighting

9.2.4 Putting It All Together

Finally we can merge ambient, diffuse and specular components together. We do this by simply adding them together.

```
1  vec3 result = (ambient + diffuse + specular) * inObjectColor;
2  outFragmentColor = vec4(result , 1.0);
```

Listing 9.4: Computing final light value

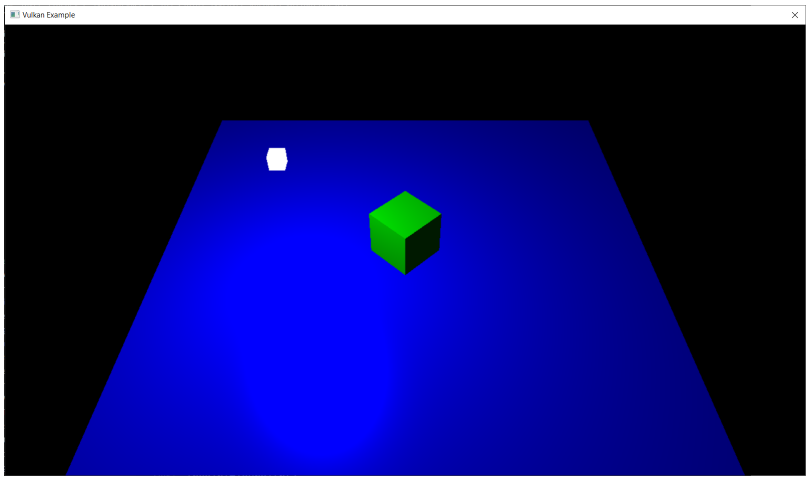

Figure 9.8: Scene with ambient, diffuse and specular lighting

9.3 Materials

Real world objects have different reactions to light. We simulate this property using materials. We can describe a material by specifying four different proper-

ties: one for each lighting component plus a shininess value. We can simulate a lot of different real world materials simply using different combinations of these four values.

9.3.1 Scene Materials

In our scene, we use two materials. The `turquoise` material is used by the floor entity. The `emerald` material is used by the cube entity.

```
Material turquoise = {};
turquoise.ambient   = { 0.1f,      0.18725f, 0.1745f   };
turquoise.diffuse   = { 0.396f,    0.74151f, 0.69102f  };
turquoise.specular  = { 0.297254f, 0.30829f, 0.306678f };
turquoise.shininess = 128.0f * 0.1f;

Material emerald = {};
emerald.ambient   = { 0.0215f, 0.1745f,   0.0215f };
emerald.diffuse   = { 0.0215f, 0.1745f,   0.0215f };
emerald.specular  = { 0.633f,  0.727811f, 0.633f  };
emerald.shininess = 128.0f * 0.6f;
```

Listing 9.5: Materials used in our scene

9.3.2 Blinn-Phong With Object Materials

The ambient material property defines what color the surface reflects under ambient lighting. This is usually the same as the surface's color. The diffuse material property defines the color of the surface under diffuse lighting. The specular material property defines the color of the specular highlight. The shininess material property affects the specular highlight's radius.

```
// Compute ambient lighting
vec3 ambient = inLightColor * inObjectMaterialAmbient;

// Compute diffuse lighting
vec3 normal = normalize(inNormal);
vec3 lightDirection = normalize(inLightPosition -
    inFragmentPosition);
float diffuseImpact = max(dot(normal, lightDirection), 0.0);
vec3 diffuse = inLightColor * (diffuseImpact *
    inObjectMaterialDiffuse);

// Compute specular lighting
bool blinn = true;
float blinnShininessScale = 3;
vec3 cameraDirection = normalize(inCameraPosition -
    inFragmentPosition);
vec3 halfwayDirection = normalize(lightDirection + cameraDirection)
    ;
float specularImpact = pow(max(dot(normal, halfwayDirection), 0.0),
     inObjectMaterialShininess.x * blinnShininessScale);
vec3 specular = inLightColor * (specularImpact *
    inObjectMaterialSpecular);

// Combine ambient, diffuse and specular lighting
vec3 result = ambient + diffuse + specular;
outFragmentColor = vec4(result, 1.0);
```

Listing 9.6: Blinn-Phong lighting using object materials

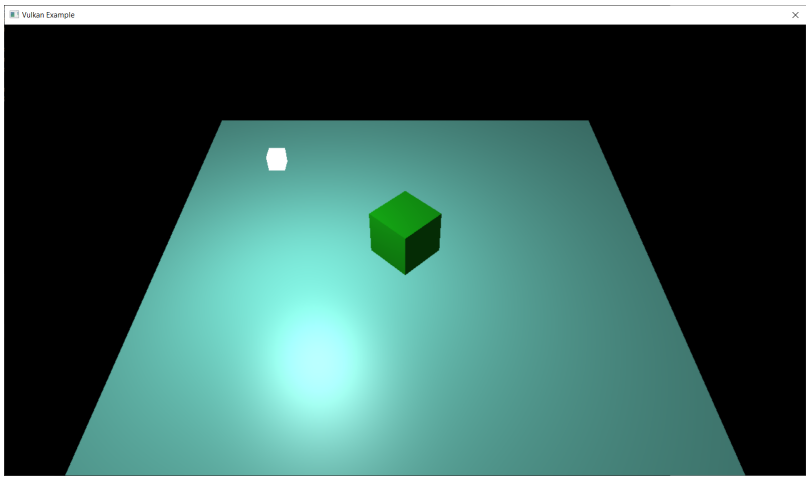

Figure 9.9: Scene lighting using material properties

9.3.3 Blinn-Phong With Object And Light Materials

In the previous image, we see that the objects are too bright. This is due to the fact that the ambient, diffuse and specular colors are computed simply using the light's color. Lights also have different intensities for their ambient, diffuse and specular components.

We set the ambient component to a low intensity because we don't want the ambient color to be too dominant. We set the diffuse component to a slightly darkened version of the light's color. We set the specular component to the light's color, shining at full intensity.

```
1  light.color             = glm::vec3(1.0f, 1.0f, 1.0f);
2  light.material.ambient  = light.color * 0.2f;
3  light.material.diffuse  = light.color * 0.5f;
4  light.material.specular = light.color * 1.0f;
```

Listing 9.7: Our scene light's material

```
// Compute ambient lighting
vec3 ambient = inLightMaterialAmbient * inObjectMaterialAmbient;

// Compute diffuse lighting
vec3 normal = normalize(inNormal);
vec3 lightDirection = normalize(inLightPosition -
    inFragmentPosition);
float diffuseImpact = max(dot(normal, lightDirection), 0.0);
vec3 diffuse = inLightMaterialDiffuse * (diffuseImpact *
    inObjectMaterialDiffuse);

// Compute specular lighting
bool blinn = true;
float blinnShininessScale = 3;
vec3 cameraDirection = normalize(inCameraPosition -
    inFragmentPosition);
vec3 halfwayDirection = normalize(lightDirection + cameraDirection)
    ;
float specularImpact = pow(max(dot(normal, halfwayDirection), 0.0),
     inObjectMaterialShininess.x * blinnShininessScale);
vec3 specular = inLightMaterialSpecular * (specularImpact *
    inObjectMaterialSpecular);

// Combine ambient, diffuse and specular lighting
vec3 result = ambient + diffuse + specular;
outFragmentColor = vec4(result, 1.0);
```

Listing 9.8: Blinn-Phong lighting using object and light materials

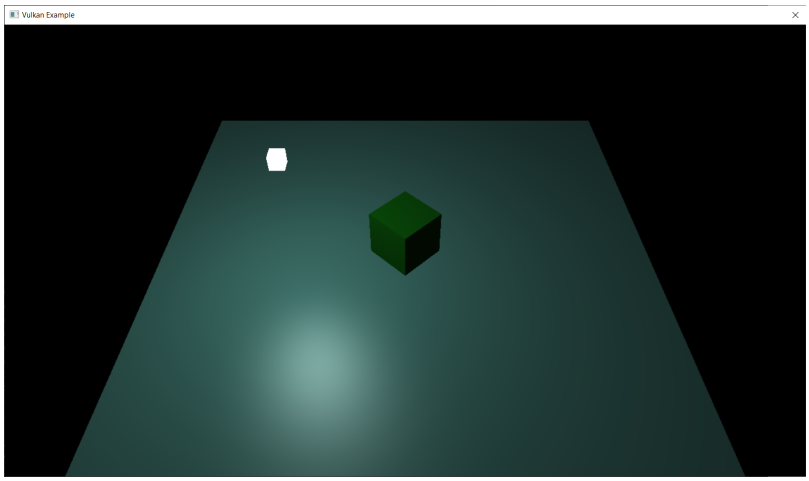

Figure 9.10: Scene lighting using object and light materials

Chapter 10

Multisample Anti Aliasing

If we examine up close the images we have rendered up to this point, we can notice that some of our edges have a jagged saw like pattern. This effect is called aliasing and it's the result of having to render our images on a grid with a finite number of pixels. We can't completely avoid this effect since screens will always have a finite resolution. We can, however, attenuate the issue using a technique known as multisample anti aliasing, MSAA for short.

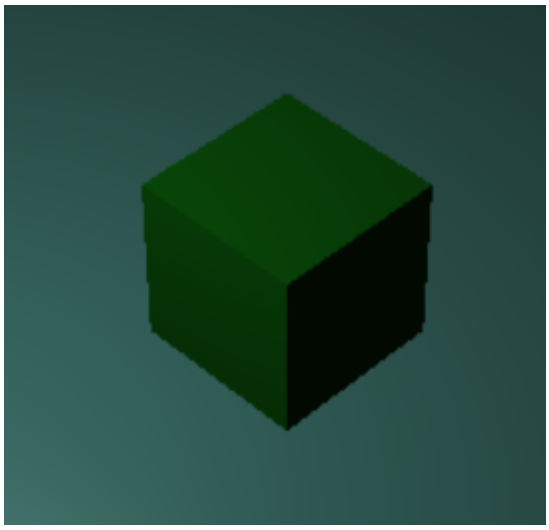

Figure 10.1: An example of aliasing

10.1 MSAA

So far, we have always used only one sample per pixel. This means that we determine its color using a single sample point, placed at its center. If the rendered geometry doesn't cover the sample point, the entire pixel is left blank. Else, if the rendered geometry covers the sample point, we color the entirety of the pixel. This is what causes aliasing.

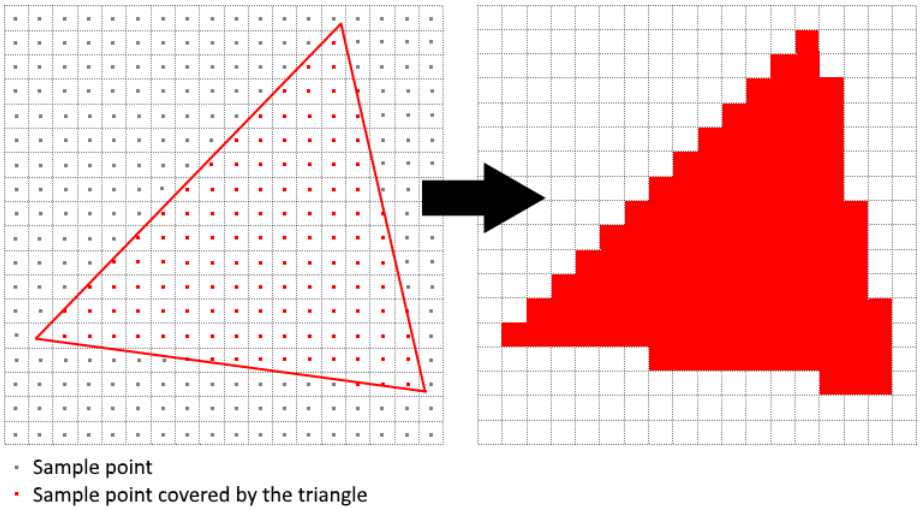

Figure 10.2: Rendering using one sample per pixel

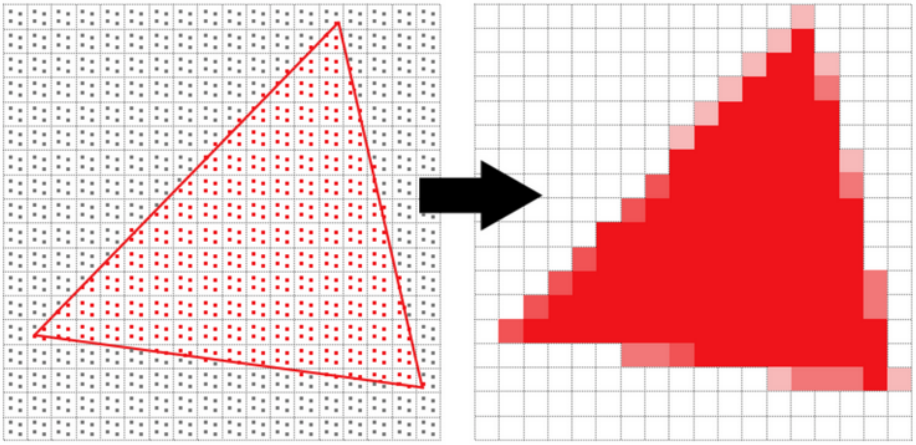

Figure 10.3: Rendering using four samples per pixel

MSAA uses multiple sample points per pixel to determine its final color. More samples lead to better results but also mean more overhead. For each pixel, the less the samples are covered by the geometry, the less the geometry color contributes to the pixel final color. Thanks to this, edges are surrounded by colors slightly lighter than the edge's color itself. This causes the edge to appear smooth when viewed from a distance.

10.2 Adding MSAA In Vulkan

In this section we discuss all the steps that are necessary to add MSAA to our Vulkan application.

10.2.1 Get Available Sample Count

We must determine how many samples our hardware can use. To do this we query the maximum number of samples for both color and depth values. After that, we pick the highest sample count that they both support.

```
1  VkSampleCountFlagBits GetMSAASamples(VkPhysicalDevice
       physicalDevice)
2  {
3      VkPhysicalDeviceProperties props = {};
4      vkGetPhysicalDeviceProperties(physicalDevice, &props);
5
6      VkSampleCountFlags flags =
7          props.limits.framebufferColorSampleCounts &
8          props.limits.framebufferDepthSampleCounts;
9
10     if (flags & VK_SAMPLE_COUNT_64_BIT) return
       VK_SAMPLE_COUNT_64_BIT;
11     if (flags & VK_SAMPLE_COUNT_32_BIT) return
       VK_SAMPLE_COUNT_32_BIT;
12     if (flags & VK_SAMPLE_COUNT_16_BIT) return
       VK_SAMPLE_COUNT_16_BIT;
13     if (flags & VK_SAMPLE_COUNT_8_BIT)  return
       VK_SAMPLE_COUNT_8_BIT;
14     if (flags & VK_SAMPLE_COUNT_4_BIT)  return
       VK_SAMPLE_COUNT_4_BIT;
15     if (flags & VK_SAMPLE_COUNT_2_BIT)  return
       VK_SAMPLE_COUNT_2_BIT;
16
17     return VK_SAMPLE_COUNT_1_BIT;
18 }
```

Listing 10.1: Determine the maximum supported sample count

10.2.2 Create New Render Target

Up to this point, we have always used the next swapchain image as our render target. The problem is that our swapchain images store only one sample per pixel. They can't store more than one sample per pixel, because that, would make them not presentable. Thus, we have to create a new multisampled image that will be used as our application's render target.

Creating The Multisample Render Target

To create our new render target, we reuse the image creation functions we saw in earlier chapters. The only thing different from before, is that we now pass the number of samples that our image will store per pixel.

```
VexCreateImage
(
    physicalDevice,
    device,
    swapchainImageExtent.width,
    swapchainImageExtent.height,
    msaaSamples,
    swapchainImageFormat.format,
    VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
    &renderTargetImage, &renderTargetMemory
);

VexCreateImageView
(
    device,
    renderTargetImage,
    swapchainImageFormat.format,
    VK_IMAGE_ASPECT_COLOR_BIT,
    &renderTargetView
);
```

Listing 10.2: Create the multisample render target

Update Depth Buffer Creation

We must also remember to update the depth buffer creation. This is because the depth buffer will also store more samples per pixel.

10.2.3 Update Render Pass

We set the color attachment and the depth attachment samples count to `msaaSamples`. We also change the color attachment's final layout to `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. We do this because multisampled images can't be directly presented.

Add A Resolve Color Attachment

Right now, we only have a multisampled render target and a multisampled depth buffer. Neither is suitable to be presented to the screen. Because of this, we add a new color attachment to the render pass. After finishing our rendering operations, we will resolve our render target to this new color attachment. This resolve color attachment is the one that will be presented to the screen.

```
1  VkAttachmentDescription colorResolveAttachment = {};
2  colorResolveAttachment.format = swapchainImageFormat.format;
3  colorResolveAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
4  colorResolveAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
5  colorResolveAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
6  colorResolveAttachment.stencilLoadOp =
       VK_ATTACHMENT_LOAD_OP_DONT_CARE;
7  colorResolveAttachment.stencilStoreOp =
       VK_ATTACHMENT_STORE_OP_DONT_CARE;
8  colorResolveAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
9  colorResolveAttachment.finalLayout =
       VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Listing 10.3: Color resolve attachment

Render Pass Attchments

Now our render pass has three attachments.

```
1  VkAttachmentDescription attachments[] =
2  {
3      colorAttachment,
4      depthAttachment,
5      colorResolveAttachment,
6  };
```

Listing 10.4: MSAA render pass attachments

Update Color Subpass Description

We must tell to our color subpass to actually use the color resolve attachment. To do this we update its description.

```
1  VkAttachmentReference colorAttachmentResolveReference = {};
2  colorAttachmentResolveReference.attachment = 2;
3  colorAttachmentResolveReference.layout =
       VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
4
5  // ...
6
7  colorSubpass.pResolveAttachments = &colorAttachmentResolveReference
       ;
```

Listing 10.5: Color resolve attachment reference

10.2.4 Update Multisampling Pipeline State

Not only we need to create resources that are compatible with MSAA. We also need to directly enable it while creating the pipeline state object.

```
1  VkPipelineMultisampleStateCreateInfo multisamplingState = {};
2  multisamplingState.sType =
       VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
3  multisamplingState.rasterizationSamples = msaaSamples;
```

Listing 10.6: Enable multisampling

10.2.5 Update Framebuffer

Now that we have updated our render pass attachments, we also must update the attachments that are used by our framebuffers. We use the multisample render target as the first attachment. We use the depth buffer as the second attachment. And we use the next swapchain image as the color resolve attachment.

10.3 Side By Side Comparison

Here we can see a side by side comparison of how MSAA influences the rendered image's quality. On the left we see the cube rendered without using MSAA. On the right we see the cube rendered using MSAA. As explained earlier, the jagged pattern gets blurred making it look smoother. This leads to more pleasant visuals.

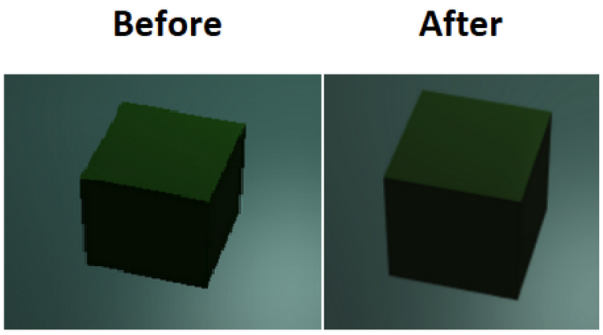

Figure 10.4: Before and after MSAA

Conclusion

The goal of this work was to explore the Vulkan graphics API, studying its ideas and seeing how to put them into practice. During this work, we have dealt with many concepts that, not only, are core to Vulkan, but also to other modern graphics APIs, such as Direct3D and Metal. Having a good grasp on these concepts allows us to be more at ease transitioning to these other APIs. Having a good understanding of Vulkan, we can also implement many real time rendering ideas: we have seen an example using Blinn-Phong lighting.

We have met a lot of Vulkan concepts so far, but there are many more that we haven't faced, being more advanced and specific. We haven't discussed on how to use multiple render pass subpasses and how to describe the dependencies between them. We haven't talked about how we could render multiple frames concurrently, using a pool of command buffers. We haven't faced the problem of GPU memory allocation and how to implement an appropriate memory allocator. We haven't seen how to deal with textures, generating mipmaps and mapping them to 3D objects. These are some ideas that I would like to suggest to people that want to keep delving deeper into Vulkan.

Appendix A

Vulkan Concepts

A.1 Queues, Queue Families And Command Buffers

A.1.1 Command Buffer

In older graphics APIs, like OpenGL, we simply need to call a function from our code to execute some operations on the GPU. This function call causes our application to send a request to the GPU driver. The GPU driver will then execute the operations we want. The problem is that it's inefficient to send each request separately. Vulkan requires us to batch our requests together into a buffer. This batch is called a command buffer.

A.1.2 Queues and Queue Families

A command buffer is executed by a GPU queue. A command buffer can contain different kinds of operations, such as graphics commands, compute commands, transfer commands and so on. Specific types of commands can only be executed by specific types of queues. These queue types are called queue families.

There is no connection between a command buffer and a queue or queue family. There is, although, a connection between the command pool form which a command buffer is allocated and a queue family. Each command buffer that takes memory from a given pool can only be submitted to a queue from the queue family used to create the command pool itself.

A.1.3 Command Buffer Execution

If we want our GPU to execute any kind of commands we need to record a command buffer and submit it to a proper queue. When a command buffer is submitted to a queue, all the recorded commands start being processed by the GPU. The Vulkan specification guarantees that each command will start execution in order, but complete their execution out of order. This is caused by the fact that commands are executed concurrently. This means that unless we add synchronization ourselves, all commands in a queue execute out of order. Commands my get reordered multiple command buffers and even across different command buffer submissions. The GPU only sees a linear stream of commands.

It's a common pitfall to assume that splitting command buffers or submits adds some sort of automatic synchronization between them.

A.2 Image Layouts And Layout Transitions

Images are used for different purposes. They can be used as render targets, as textures or even for data transfers. When we create an image we also specify its usage flags. These flags indicate the different types of operations we want to perform on the image. Image usage is not the only thing we need to specify. Each type of operation is linked to a specific image layout. We could also use a general layout that supports all operations, but this would be inefficient. We usually change the image's layout before we perform a given type of operation. We explicitly transition an image's layout using an image memory barrier. We implicitly transition an image's layout using a render pass' subpass.

A.3 Swapchain

A swapchain is nothing but a set of images that can be presented on the screen. Presenting an image means displaying it. In practice, we present an image by giving it back to the OS's presentation engine. We use a present mode to configure how images are internally processed by the presentation engine, and how they are displayed to the screen.

A.3.1 Immediate

This is the simplest present mode. Using this present mode, the presentation engine reserves only one swapchain image at a time. This image is the one that is currently displayed on the screen. When we execute a present request, the presentation engine releases the current displayed image and acquires the one we want to present. The problem with this present mode is that the presentation engine doesn't wait for our monitor to finish reading the image. This can cause a graphical artifact known as tearing.

A.3.2 FIFO

Using this present mode, the presentation engine reserves at least two swapchain images at a time. One image that is being presented and one or more images that we can use for rendering. When we execute a present request, the presentation engine acquires the image we want to present and puts it into a queue. When the monitor finishes displaying the current image, the presentation engine removes it from the head of the queue, releasing it, and then starts presenting the next image in the queue. Doing this we avoid tearing, but another problem arises. When the queue is full, our application doesn't have other images that can be used for rendering, hence it has to wait for the presentation engine to release the currently presented image.

A.3.3 Mailbox

Using this present mode, the presentation engine reserves at least three swapchain images. One image that is being presented, one image that waits for the current image to be presented and one or more images that can bes used for rendering. When we issue a present request, the presentation engine replaces the image that is waiting to be presented, releasing it, with the one we want to present. This technique solves tearing, and also doesn't require our application to block execution, since there is always at least an image we can use.

A.4 Render Pass And Framebuffer

A.4.1 Render Pass

A render pass describes a set of images, also called attachments, required for drawing operations. A render pass also describes a series of subpasses that drawing operations are ordered into.

A subpass collects drawing operations that use the same attachments. Each of these drawing operations may use some attachments as inputs, reding data from them, and other attachments as outputs, writing data to them.

A.4.2 Framebuffer

A framebuffer is the set of images a render pass instance operates on. Hence, a framebuffer collects the actual attachments used by a render pass.

A.5 Pipeline State Object

The graphics pipeline is composed by a series of stages. Each performing a given operation. Some stages are programmable by us, while other stages can only be partially configured. OpenGL allows us to change the stages' configuration whenever we want. In Vulkan we can't do this. We must bundle our pipeline configuration into one monolithic object that represent the pipeline's state in its entirety. This comes from performance considerations. Changing the configuration of just one single stage may require a lot of operations executed by the driver in the background at runtime. This may cause noticeable slowdowns in our application. In Vulkan we avoid this by creating, ahead of time, multiple pipeline state objects, each representing a given set of stages' configurations.

A.6 Descriptors And Descriptor Sets

Resources used inside shaders are called descriptors. Vulkan doesn't allow us to directly provide descriptors to shaders. We must aggregate descriptors in an object called descriptor set. We can place whatever resources we want inside a descriptor set, but we have to respect its specific structure. The structure of a descriptor set describes what types of resources can be contained inside the set, the number of each of these resources and their order. This description is provided by a descriptor set layout. In summary, a descriptor set is an object

in which we store resources' handles, a descriptor set layout defines how to interpret the data stored inside a descriptor set.

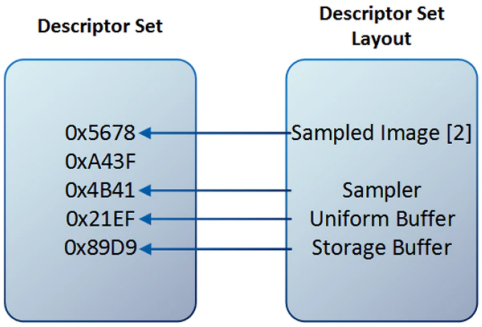

Figure A.1: Descriptor set and descriptor set layout

Bibliography

[1] Pawel Lapinski. *Vulkan Cookbook: work through recipes to unlock the full potential of the next generation graphics API-Vulkan.* eng. Birmingham: Packt, 2017. ISBN: 9781786468154.

[2] Graham Sellers. *Vulkan programming guide: the official guide to learning Vulkan.* OpenGL series. Boston: Addison-Wesley, 2017. ISBN: 9780134464541.

Sitography

[3] Wayne Brown. *Learn WebGL - Computer Graphics - A Brief History.* "http://learnwebgl.brown37.net/the_big_picture/webgl_history.html".

[4] *Creating a Window - Win32 apps — Microsoft Docs.* "https://docs.microsoft.com/en-us/windows/win32/learnwin32/creating-a-window".

[5] David J. Eck. *Graphics Book - Hardware And Software.* "https://math.hws.edu/graphicsbook/c1/s3.html".

[6] Mike Gleicher. *Graphics History and the Red Book.* "https://pages.graphics.cs.wisc.edu/559-f14/2014/09/02/graphics-history-and-the-red-book/".

[7] Erika Johnson. *Overview of Vulkan Loader and Layers - LunarG.* "https://www.lunarg.com/tutorial-overview-of-vulkan-loader-layers".

[8] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 1: The Beginning.* "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-1.html".

[9] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 2: Swap Chain.* "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-2.html".

[10] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 3: First Triangle.* "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-3.html".

[11] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 4: Vertex Attributes.* "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-4.html".

[12] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 5: Staging Resources.* "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-5.html".

[13] Pawel Lapinski. *API without Secrets: Introduction to Vulkan Part 7: Uniform Buffers*. "https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-7.html".

[14] Eddy Luten. *OpenGL Book - Chapter 0 - Preface - What is OpenGL*. "https://openglbook.com/chapter-0-preface-what-is-opengl.html".

[15] Alexander Overvoorde. *Command buffers - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Command_buffers".

[16] Alexander Overvoorde. *Depth Buffering - Vulkan Tutorial*. "https://vulkan-tutorial.com/Depth_buffering".

[17] Alexander Overvoorde. *Framebuffers - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Framebuffers".

[18] Alexander Overvoorde. *Graphics Pipeline Basics - Conclusion - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Conclusion".

[19] Alexander Overvoorde. *Graphics Pipeline Basics - Fixed Functions - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Fixed_functions".

[20] Alexander Overvoorde. *Graphics Pipeline Basics - Introduction - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction".

[21] Alexander Overvoorde. *Graphics Pipeline Basics - Render Passes - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Render_passes".

[22] Alexander Overvoorde. *Graphics Pipeline Basics - Shader Modules - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules".

[23] Alexander Overvoorde. *Image views - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Image_views".

[24] Alexander Overvoorde. *Instance - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance".

[25] Alexander Overvoorde. *Introduction - Vulkan Tutorial*. "https://vulkan-tutorial.com/".

[26] Alexander Overvoorde. *Logical devices and queues - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Logical_device_and_queues".

[27] Alexander Overvoorde. *Multisampling - Vulkan Tutorial*. "https://vulkan-tutorial.com/Multisampling".

[28] Alexander Overvoorde. *Physical devices and queue families - Vulkan Tutorial*. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Physical_devices_and_queue_families".

[29] Alexander Overvoorde. *Rendering And Presentation - Vulkan Tutorial.* "https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Rendering_and_presentation".

[30] Alexander Overvoorde. *Swap Chain - Vulkan Tutorial.* "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Swap_chain".

[31] Alexander Overvoorde. *Texture Mapping - Image View And Sampler - Vulkan Tutorial.* "https://vulkan-tutorial.com/Texture_mapping/Image_view_and_sampler".

[32] Alexander Overvoorde. *Texture Mapping - Images - Vulkan Tutorial.* "https://vulkan-tutorial.com/Texture_mapping/Images".

[33] Alexander Overvoorde. *Uniform Buffers - Descriptor Layout And Buffer - Vulkan Tutorial.* "https://vulkan-tutorial.com/Uniform_buffers/Descriptor_layout_and_buffer".

[34] Alexander Overvoorde. *Uniform Buffers - Descriptor Pool And Sets - Vulkan Tutorial.* "https://vulkan-tutorial.com/Uniform_buffers/Descriptor_pool_and_sets".

[35] Alexander Overvoorde. *Validation Layers - Vulkan Tutorial.* "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Validation_layers".

[36] Alexander Overvoorde. *Vertex Buffers - Staging Buffer - Vulkan Tutorial.* "https://vulkan-tutorial.com/Vertex_buffers/Staging_buffer".

[37] Alexander Overvoorde. *Vertex Buffers - Vertex Buffer Creation - Vulkan Tutorial.* "https://vulkan-tutorial.com/Vertex_buffers/Vertex_buffer_creation".

[38] Alexander Overvoorde. *Vertex Buffers - Vertex Input Description - Vulkan Tutorial.* "https://vulkan-tutorial.com/Vertex_buffers/Vertex_input_description".

[39] Alexander Overvoorde. *Window Surface - Vulkan Tutorial.* "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Window_surface".

[40] Joey de Vries. *Learn OpenGL - Advanced Lighting - Advanced Lighting.* "https://learnopengl.com/Advanced-Lighting/Advanced-Lighting".

[41] Joey de Vries. *Learn OpenGL - Advanced Lighting - Anti-Aliasing.* "https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing".

[42] Joey de Vries. *Learn OpenGL - Getting Started - Coordinate Systems.* "https://learnopengl.com/Getting-started/Coordinate-Systems".

[43] Joey de Vries. *Learn OpenGL - Lighting - Basic Lighting.* "https://learnopengl.com/Lighting/Basic-Lighting".

[44] Joey de Vries. *Learn OpenGL - Lighting - Colors.* "https://learnopengl.com/Lighting/Colors".

[45] Joey de Vries. *Learn OpenGL - Lighting - Materials.* "https://learnopengl.com/Lighting/Materials".