Exploring The Vulkan Graphics API

Emanuele Franchi

February 15, 2022

Abstract

Thesis about Vulkan

Dedication

Bla Bla Bla

Acknowledgments

I want to thank...

Contents

List of Figures

Listings

Chapter 1

Vulkan

1.1 What is Vulkan?

Figure 1.1: Vulkan logo

Vulkan is a modern graphics API. It is maintained by the Khronos Group. Vulkan is meant to abstract how modern GPUs work. Using Vulkan, the programmer can write more performant code. The better performance comes at the cost of having a more verbose and low level API compared to other existing APIs such as OpenGL or Direct3D 11 and prior. Vulkan is not the only modern graphics API, other such APIs are Direct3D 12 and Metal. Nonetheless, Vulkan has the advantage of being fully cross platform.

1.2 What problems does Vulkan solve?

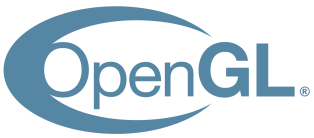

Figure 1.2: OpenGL logo

Common graphics APIs like OpenGL or Direct3D were developed during the 1990s. At that time, graphics card hardware was very limited not only in terms of computational power but also from a functionality standpoint. As time progressed, graphics card architectures continued to evolve, offering new functionalities. All these new functionalities had to be integrated with the old existing APIs. The more new functionalities were integrated, the more the GPU's driver complexity grew. Such complicated GPU drivers are inefficient and are also the cause of many inconsistencies between implementations of the same graphics API but on different GPUs.

1.3 How does Vulkan solve these problems?

Vulkan doesn't suffer from the problems we saw above because it has been designed from scratch and with modern GPU's architecture in mind. It reduces the driver overhead by being more verbose and low level. It is also designed to be multithreaded, allowing the programmer to submit GPU commands from different threads. This is very beneficial to performance, since modern CPUs usually have more than one core.

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 Vulkan 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.

We can 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 also 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.

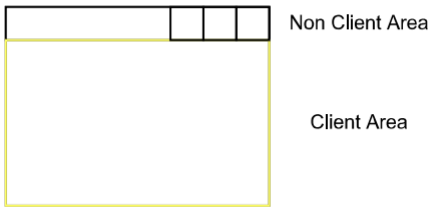

Figure 2.1: Anatomy of a Win32 Window

```
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 our 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 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

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

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

VkDeviceQueueCreateInfo presentQueueCreateInfo = {};
presentQueueCreateInfo.sType =
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
presentQueueCreateInfo.queueFamilyIndex = presentQueueFamilyIndex;
presentQueueCreateInfo.queueCount = 1;
presentQueueCreateInfo.pQueuePriorities = &queuePriority;

VkDeviceQueueCreateInfo queueCreateInfos[] =
{
    graphicsQueueCreateInfo,
    presentQueueCreateInfo,
};

VkDeviceCreateInfo createInfo = {};
deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.queueCreateInfoCount = arraysize(queueCreateInfos);
createInfo.pQueueCreateInfos = queueCreateInfos;
createInfo.enabledExtensionCount = deviceExtensionCount;
createInfo.ppEnabledExtensionNames = deviceExtensions;
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.

```
VkQueue graphicsQueue = VK_NULL_HANDLE;
vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &
    graphicsQueue);

VkQueue presentQueue = VK_NULL_HANDLE;
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.

```
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      // We search for a format that we like
10     for (u32 i = 0; i < formatCount; i++)
11     {
12         if (formats[i].format == VK_FORMAT_R8G8B8A8_UNORM)
13         {
14             return formats[i];
15         }
16     }
17
18     // We haven't found the format(s) that we were looking for
19     // Pick the first format
20     return formats[0];
21 }
```

Listing 2.28: Select swapchain image format

We would like to use SRGB color space, with 32 bit RGBA 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

A presentation mode tells the conditions for showing our swapchain images to the screen.

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 the 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 The Swapchain Images

Now that we have created a swapchain we can retrieve the handles to the images in it. We 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 have to destroy the swapchain image views using `vkDestroyImageView`. We then destroy the swapchain itself using `vkDestroySwapchainKHR`. The swapchain images will be automatically destroyed when we destroy our swapchain.

2.7 Our Application So Far

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

First, during our application startup phase, we create some resources required for rendering. We create two semaphores to synchronize the execution of graphics and present commands. We create a command pool and allocate a command buffer from it. Finally, we describe our rendering operations using a render pass.

Then, during our application main loop, we execute a set of steps that are necessary for our rendering and presenting to be correct. We first acquire a swapchain image that will be used as our render target. We wait for the previously submitted commands to finish their execution. We bundle our swapchain image into a framebuffer, so that we can use it during rendering. At last we record graphics commands into our command buffer. We submit the command buffer to our graphics queue. Finally, we submit a present command to our present queue.

Figure 3.1: Clear the window background blue

3.1 Create Commands Synchronization Resources

We have to manually synchronize the commands we use for rendering an image and the commands we use for presenting said image to the window. To do this we use two semaphores. One semaphore will be signaled when a swapchain image is available to be used as our render target. Another semaphore will be signaled when we finish rendering our image. Only after this semaphore has been signaled, we are allowed to present our image to the window.

```
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 Graphics Command Pool

Before submitting commands to a GPU queue, we need to create a command pool. We will explicitly submit commands only to our graphics queue. Hence, we create one graphics command pool.

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

Listing 3.2: Create graphics command pool

3.2.1 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 3.3: Configure our graphics command pool

3.2.2 Cleanup

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

3.3 Create Command Buffer

We need a command buffer to submit commands to our GPU. We allocate a command buffer from a command pool.

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

Listing 3.4: Allocate a command buffer from our graphics command pool

3.3.1 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 GPUs to be executed.

```
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 3.5: Configure command buffer creation

3.3.2 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

3.3.3 Cleanup

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

3.4 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.

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

Listing 3.7: Create a render pass

3.4.1 VkRenderPassCreateInfo

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

```
VkRenderPassCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
createInfo.attachmentCount = attachmentCount;
createInfo.pAttachments = attachments;
createInfo.subpassCount = subpassCount;
createInfo.pSubpasses = subpassess;
// If there is more than one subpass, we need to specify
// synchronization requirements through subpass dependencies
createInfo.dependencyCount = 0;
createInfo.pDependencies = nullptr;
```

Listing 3.8: Configure our render pass

3.4.2 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 our render pass.

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

```
VkAttachmentDescription colorAttachment = {};
colorAttachment.format         = swapchainImageFormat;
colorAttachment.samples        = VK_SAMPLE_COUNT_1_BIT;
colorAttachment.loadOp         = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp        = VK_ATTACHMENT_STORE_OP_STORE;
colorAttachment.stencilLoadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
colorAttachment.initialLayout  = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout    = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

VkAttachmentDescription attachments[] =
{
    colorAttachment,
};
```

Listing 3.9: Render pass attachment descriptions

3.4.3 Render Pass Subpasses

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

In our case we have only one subpass that uses our single attachment to write color data into it. In order to be able to write color data into our attachment we need it to have a proper image layout.

```
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

3.4.4 Cleanup

To destroy our render pass we use `vkDestroyRenderPass`.

3.5 Clear The Window

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

3.5.1 Acquire A Swapchain Image

The first step for drawing something to the screen is to get an image that serves as our render target. We also need to be able to present this image 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 presentable swapchain image. To determine such image we use `vkAcquireNextImageKHR`. The image is not guaranteed to be already available when the function returns. For this reason, we use our image available semaphore. 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

3.5.2 Wait For The Previous Commands To Finish

Before recording new commands into our command buffer, we have to wait for its execution to finish. To do this wait for our command buffer fence to be signaled. After the wait terminates, we have to manually reset our 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

3.5.3 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 our 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 presentable 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

3.5.4 Record Rendering Commands

Now we can start recording our 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 our 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.

3.5.5 Submit Rendering Commands

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

3.5.6 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

3.6 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`.

3.7 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. 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.

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

In our application main loop, during our 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.

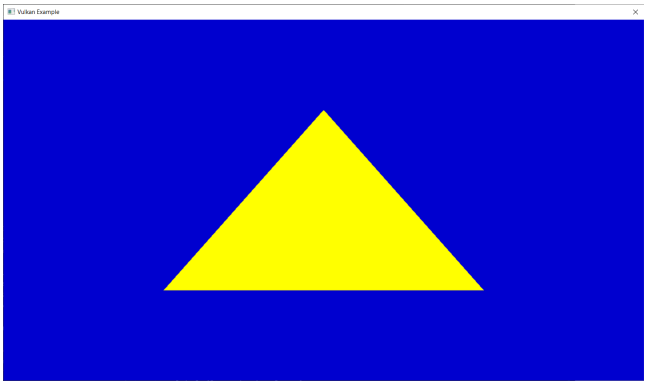

Figure 4.1: Rendering our triangle

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.

`layout` is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline. It doesn't matter in our case. We will see its use when we talk about uniforms.

`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.

```
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

4.1.2 Shader Stages

We have to 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.

`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.

```
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

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.

`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.

```
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

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.

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

```
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

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.

`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.

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

4.1.10 Pipeline Layout

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

```
1  VkPipelineLayout pipelineLayout = VK_NULL_HANDLE;
2  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 our pipeline layout. We will use it in later chapters.

```
1  VkPipelineLayoutCreateInfo createInfo = {};
2  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.

```
1  // begin render pass ...
2
3  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
       pipeline);
4  vkCmdDraw(
5      commandBuffer,
6      3, // number of vertices to draw
7      1, // number of instances to draw (we don't use instancing)
8      0, // index of the first vertex to draw
9      0  // instance ID of the first instance to draw
10 );
11
12 // 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 new, fancier, triangle.

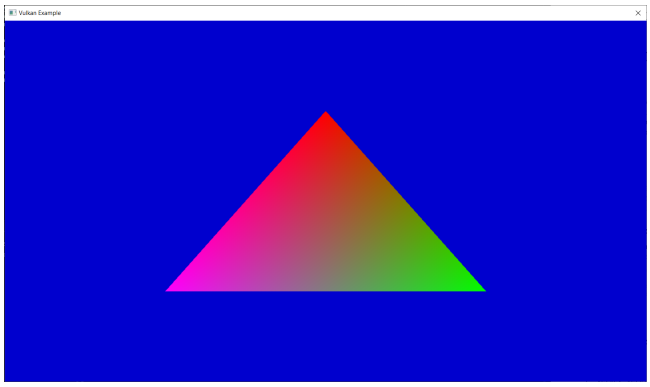

Figure 5.1: Rendering a triangle

5.1 Vertex Data

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

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

In order to draw a triangle, we specify three 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      { { +0.0f, -0.5f, }, { 1.0f, 0.0f, 0.0f, }, },
4      { { +0.5f, +0.5f, }, { 0.0f, 1.0f, 0.0f, }, },
5      { { -0.5f, +0.5f, }, { 1.0f, 0.0f, 1.0f, }, },
6  };
```

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. The new vertex shader will also have to handle the fact that our vertices now store a color value. The new fragment shader is due to the fact that 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 our vertex color to our 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 triangle, we upload our vertex data to the GPU. This vertex data will then be 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 side 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.

```
1  u32 VexFindMemoryType
2  (
3      VkPhysicalDevice physicalDevice,
4      u32 supportedMemoryTypes,
5      VkMemoryPropertyFlags requiredMemoryProperties
6  )
7  {
8      // Get the available types of memory
9      VkPhysicalDeviceMemoryProperties memoryProperties = {};
10     vkGetPhysicalDeviceMemoryProperties(physicalDevice, &
       memoryProperties);
11
12     // Find a memory type that is supported
13     for (u32 i = 0; i < memoryProperties.memoryTypeCount; i++)
14     {
15         bool isMemoryTypeSupported = (supportedMemoryTypes & (1 <<
       i));
16         VkMemoryPropertyFlags memoryTypeProperties =
       memoryProperties.memoryTypes[i].propertyFlags;
17         bool areRequiredMemoryPropertiesSupported = ((
       memoryTypeProperties & requiredMemoryProperties) ==
       requiredMemoryProperties);
18         if (isMemoryTypeSupported &&
       areRequiredMemoryPropertiesSupported)
19         {
20             return i;
21         }
22     }
23
24     assert(false, "Failed to find a suitable memory type");
25
26     return 0;
27 }
```

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.

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

```
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

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. Thus, in our case, 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. `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.

```
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

5.5 Draw Using Our Vertex Data

The only missing thing now is to tell Vulkan to render a triangle using our vertex data. We do this during our 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 triangle 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. To make uniforms accessible from our vertex shader, we use a uniform buffer. At the end, we use our uniforms to rotate and render our triangle from a perspective camera's point of view.

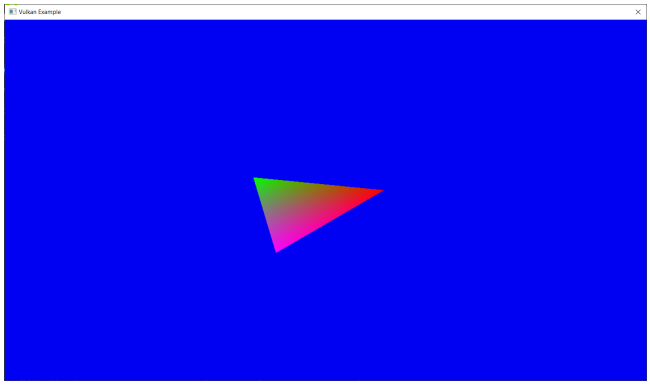

Figure 6.1: Rendering a triangle

6.1 Uniform Data

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

6.1.1 Uniforms

To draw our triangle we need three uniforms: a model matrix, a view matrix, and a projection matrix.

```
1  glm::mat4 model;
2  glm::mat4 view;
3  glm::mat4 projection;
```

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

6.1.2 Uniform Buffer Data

In order to draw our triangle, we have to specify the model, the view and the projection matrices. Unlike vertex data, which remains unchanged throughout the execution of our application, uniform data usually changes frame by frame. Hence, we need to upload such data during or main loop.

We use our model matrix to continuously rotate the triangle 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, a near and far planes of 0.1 and 10 respectively.

```
1  model = glm::rotate(glm::mat4(1.0f), timeSinceStart * glm::radians
       (90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
2  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));
3  f32 aspect = (f32)(swapchainImageExtent.width) / (f32)(
       swapchainImageExtent.height);
4  projection = glm::perspective(glm::radians(45.0f), aspect, 0.1f,
       10.0f);
5
6  // Perspective matrix correction (only for glm)
7  ubo.projection[1][1] *= -1;
```

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

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. We multiply the vertex position with our matrices. This transforms the vertex position. This transformation is defined by our matrices.

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

```
#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

6.2 Upload Uniform Data To The GPU

We use a buffer to upload the uniform data to the GPU. Such buffer is called a uniform buffer.

6.2.1 Uniform Buffer Layout

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

```
struct UBO
{
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 projection;
};
```

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.

```
VkBuffer uniformBufferObject = VK_NULL_HANDLE;
VkDeviceMemory uniformBufferMemory = VK_NULL_HANDLE;
u32 uniformBufferSizeInBytes = sizeof(UBO);
VexCreateBuffer
(
    physicalDevice,
    device,
    uniformBufferSizeInBytes,
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
    &uniformBufferObject,
    &uniformBufferMemory
);
```

Listing 6.5: Uniform buffer creation

Cleanup

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

6.2.3 Upload Uniform Data

To make the updated uniform values available to our shaders we must upload them to the GPU. This means uploading them to the uniform buffer.

```
UBO ubo = {};
ubo.model = model;
ubo.view = view;
ubo.projection = projection;

void* data = nullptr;
vkMapMemory(device, uniformBufferMemory, 0, sizeof(ubo), 0, &data);
memcpy(data, &ubo, sizeof(ubo));
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 `UBO` structure 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 (4 bytes for `f32`)
- A `vec2` must be aligned by $2N$
- A `vec3` or `vec4` must be aligned by $4N$
- A nested 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 Our Pipeline Layout

Now that we pass uniforms to our shaders, we modify our pipeline creation. In particular we update our pipeline layout.

6.3.1 VkPipelineLayout

A pipeline layout can be thought of 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 info.pushConstantRangeCount = 0;
11 info.pPushConstantRanges = nullptr;
12
13 vkCreatePipelineLayout(device, &info, nullptr, &pipelineLayout);
```

Listing 6.7: Update pipeline layout creation

6.3.2 VkDescriptorSetLayout

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

`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

```
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

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 first have to allocate a descriptor set from a descriptor pool. Then we can 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. In our case

`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.

```
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

6.4.3 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

6.4.4 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 only allocate one descriptor set with one uniform buffer resource.

`maxSets` is the maximum number of descriptor sets that can be allocated from the pool. `poolSizeCount` is the number of elements in `pPoolSizes`. `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.

```
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

6.4.5 Populate Descriptor Set

Once we have allocated a descriptor set, we have to actually populate it with resources. This means that we have to write into the descriptor set. We use

vkUpdateDescriptorSets for this task. Here, descriptorWrites is a pointer to an array of VkWriteDescriptorSet structures describing the descriptor sets to write to.

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

Listing 6.15: Populate descriptor set

6.4.6 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.

```
1  VkDescriptorBufferInfo info = {};
2  info.buffer = uniformBufferObject;
3  info.offset = 0;
4  info.range = uniformBufferSizeInBytes;
5
6  VkWriteDescriptorSet descriptorWrite = {};
7  descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
8  descriptorWrite.dstSet = pipelineDescriptorSet;
9  descriptorWrite.dstBinding = 0;
10 descriptorWrite.dstArrayElement = 0;
11 descriptorWrite.descriptorCount = 1;
12 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
13 descriptorWrite.pBufferInfo = &info;
```

Listing 6.16: Descriptor set write

6.5 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. 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 triangle using our uniform data

Chapter 7

Depth Testing

Chapter 8

Setting Up A Simple Scene

Chapter 9

Blinn-Phong Lighting

Chapter 10

Multisample Anti Aliasing

Chapter 11

Conclusion

Appendix A

Concepts

A.1 Graphics Pipeline

TODO: to be written ...

A.2 Shaders

TODO: to be written ...

Bibliography

[1] Creating a window - win32 apps — microsoft docs. "https://docs.microsoft.com/en-us/windows/win32/learnwin32/creating-a-window".

[2] Erika Johnson. Overview of vulkan loader and layers - LunarG. "https://www.lunarg.com/tutorial-overview-of-vulkan-loader-layers".

[3] 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".

[4] 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".

[5] 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".

[6] 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".

[7] 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".

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

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

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

[11] Alexander Overvoorde. Graphics pipeline basics - Conclusion - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Conclusion".

[12] Alexander Overvoorde. Graphics pipeline basics - Fixed Functions - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Fixed_functions".

[13] Alexander Overvoorde. Graphics pipeline basics - Introduction - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction".

[14] Alexander Overvoorde. Graphics pipeline basics - Render Passes - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Render_passes".

[15] Alexander Overvoorde. Graphics pipeline basics - Shader Modules - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules".

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

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

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

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

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

[21] Alexander Overvoorde. Rendering and presentation - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Rendering_and_presentation".

[22] Alexander Overvoorde. Swap chain - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Swap_chain".

[23] Alexander Overvoorde. Validation layers - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Validation_layers".

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

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

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

[27] Alexander Overvoorde. Window surface - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Window_surface".

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