Vulkan

Emanuele Franchi

February 8, 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?

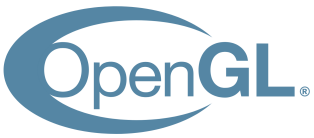

Figure 1.2: OpenGL logo

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 vkEnumerateInstance-ExtensionProperties. 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.

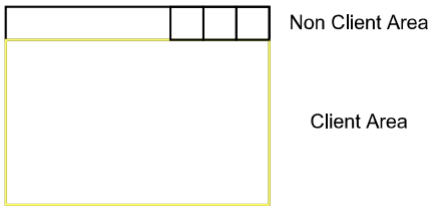

Figure 2.1: Anatomy of a Win32 Window

```
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

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

Listing 2.22: Create info struct when queue families are different

2.5.2 Retrieve Queue Handles

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

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

Listing 2.23: Retrieve queue handles

2.5.3 Cleanup

We use vkDestroyDevice to destroy our logical device.

2.6 Create A Swapchain

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

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

Listing 2.24: Create a swapchain

2.6.1 VkSwapchainCreateInfoKHR

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

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

Listing 2.25: Configure our swapchain

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

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

Listing 2.26: Configure queue ownership over swapchain images

2.6.2 Select The Minimum Swapchain Image Count

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

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

Listing 2.27: Select swapchain image count

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

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

2.6.3 Select The Swapchain Image Format

We must specify a proper format for our swapchain images. To do this, we first query for all image formats that are supported by our surface. We can do this using vkGetPhysicalDeviceSurfaceFormatsKHR.

Once we have a list of valid formats we could either pick one randomly or try to pick the one that we consider the best.

```
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 draw something on our window.

First, during our application startup phase, we need to 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 need to execute a set of steps 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.

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.

```
1  VkRenderPass renderPass = VK_NULL_HANDLE;
2  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.

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

Listing 3.8: Configure our render pass

3.4.2 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.

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

Listing 3.9: Render pass attachment descriptions

3.4.3 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.

Figure 3.1: Clear the window background blue

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.

```
1  u32 nextSwapchainImageIndex = 0;
2  vkAcquireNextImageKHR(
3      device,
4      swapchain,
5      UINT64_MAX,
6      imageAvailableSemaphore,
7      VK_NULL_HANDLE,
8      &nextSwapchainImageIndex
9  );
10
11 nextSwapchainImage = swapchainImages[nextSwapchainImageIndex];
12 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.

```
1  vkWaitForFences(device, 1, &commandBufferFence, VK_TRUE, UINT64_MAX
       );
2  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.

```
1  vkDestroyFramebuffer(device, framebuffer, nullptr);
2  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 VkFramebuffer-CreateInfo 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

Our First Pipeline

Chapter 5

Vertex Buffer

Chapter 6

Staging Buffer

Chapter 7

Uniform Buffer

Chapter 8

Depth Buffer

Chapter 9

Setting Up A Simple Scene

Chapter 10

Blinn-Phong Lighting

Chapter 11

Multisample Anti Aliasing

Chapter 12

Conclusion

Appendix A

Appendix

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

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

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

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

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

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

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

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

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

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

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

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

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