Vulkan

Emanuele Franchi

February 4, 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?

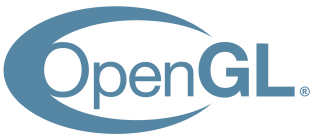

Figure 1.2: OpenGL logo

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

TODO: chapter introduction ...

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

To call vkCreateInstance we need to pass a pointer to a VkInstanceCreateInfo struct. This struct collects all the information needed to configure our Vulkan instance.

```
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. There are different validation layers. Here follows an example. 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 the Vulkan instance creation.

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.

```
1  const char* const* extensions[] =
2  {
3      #ifdef _DEBUG
4      VK_EXT_DEBUG_UTILS_EXTENSION_NAME,
5      #endif
6      // Other extensions ...
7  };
```

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 us 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

When our application is shutting down, we destroy the debug messenger and destroy our vulkan instance. DestroyDebugUtilsMessengerEXT is an extension

function proxy.

```
#ifdef _DEBUG
DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
#endif

vkDestroyInstance(instance, nullptr);
```

Listing 2.8: Vulkan Instance Cleanup

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 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 its 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 surface.

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 how we want our window.

```
DWORD windowStyle = (WS_OVERLAPPEDWINDOW | WS_VISIBLE | WS_CAPTION)
     & (~WS_THICKFRAME) & (~WS_MINIMIZEBOX) & (~WS_MAXIMIZEBOX);
DWORD windowExtendedStyle = 0;

HWND handle = CreateWindowEx(
    windowExtendedStyle,
    WINDOW_CLASS_NAME,
    name,
    windowStyle,
    CW_USEDEFAULT, CW_USEDEFAULT,
    windowWidth, windowHeight,
    0,
    0,
    GetModuleHandle(0),
    0
);
```

Listing 2.9: 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.

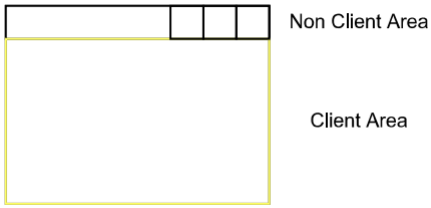

Figure 2.1: Anatomy of a Win32 Window

```
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 2.10: 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 2.11: Register Window Class

2.2.4 Window Procedure

While filling in our WNDCLASSEX struct, we also passed a WindowProcedure. This is a callback function that we have to define. 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.

```
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 2.12: Window Procedure

2.2.5 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, to create our presentation surface we need to use vkCreateWin32SurfaceKHR.

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

Listing 2.14: Create Presentation Surface

2.3.1 VkWin32SurfaceCreateInfoKHR

When we call vkCreateWin32SurfaceKHR, we need to pass a pointer to a VkWin32SurfaceCreateInfoKHR struct. Such struct lets us configure our presentation surface creation.

```
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

When our application is shutting down, we destroy our presentation surface.

```
1  vkDestroySurfaceKHR(instance, surface, nullptr);
```

Listing 2.17: Presentation Surface Cleanup

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?

Spport 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 2.18: Check for graphics operations support

Spport 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.

```
1  for (u32 i = 0; i < queueFamilyCount; i++)
2  {
3      VkBool32 presentSupport = false;
4      vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface
       , &presentSupport);
5      if (presentSupport)
6      {
7          // present operations are supported and i is the index
8          // of a queue family that supports such operations
9      }
10 }
```

Listing 2.19: 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 2.20: 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 2.21: Create a logical device

2.5.1 VkDeviceCreateInfo

When we call vkCreateDevice, we need to pass a pointer to a VkDeviceCreateInfo struct. Such struct lets us 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 ways we populate our VkDeviceCreateInfo struct is different based on whether these two indices are equal or not.

```
// Specify requested device features here
VkPhysicalDeviceFeatures deviceFeatures = {};
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 2.22: Create info struct when queue families are the same

```
// Specify requested device features here
VkPhysicalDeviceFeatures deviceFeatures = {};
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 2.23: Create info struct when queue families are different

2.5.2 Retrieve Queue Handles

After creating our logical device, it's useful to retrieve the handles to the queues we will use during our application. In our case we need to submit both graphics and present commands to our GPU. Hence, we need a handle to a graphics

queue and a handle to a present queue.

```
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 2.24: Retrieve queue handles

2.6 Create A Swapchain

Chapter 3

Clear The Window

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. Instance - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance".

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

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

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

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

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