Vulkan

Emanuele Franchi

February 3, 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

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);
```
<div align="center">Listing 2.1: Create Vulkan instance</div>

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;
```
<div align="center">Listing 2.2: VkInstanceCreateInfo initialization</div>

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;
```
<div align="center">Listing 2.3: VkApplicationInfo initialization</div>

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 inter-
cept, 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 pro-
vide. 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.

```
1  #ifdef _DEBUG
2  VkDebugUtilsMessengerCreateInfoEXT dbgInfo = {};
3  dbgInfo.sType =
        VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
4  dbgInfo.messageSeverity = severity;
5  dbgInfo.messageType = type;
6  dbgInfo.pfnUserCallback = VulkanDebugCallback;
7  #endif
8
9  VkInstanceCreateInfo createInfo = {};
10 #ifdef _DEBUG
11 createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*)(dbgInfo);
12 #endif
13
14 // ... after instance creation
15
16 // Enabling debug callback for all other API functions
17 #ifdef _DEBUG
18 VkDebugUtilsMessengerEXT debugMessenger = VK_NULL_HANDLE;
19 CreateDebugUtilsMessengerEXT(instance, &dbgInfo, nullptr, &
        debugMessenger)
20 #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.

```
1  static VkResult CreateDebugUtilsMessengerEXT
2  (
3      VkInstance instance,
4      const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
5      const VkAllocationCallbacks* pAllocator,
6      VkDebugUtilsMessengerEXT* pDebugMessenger
7  )
8  {
9      PFN_vkCreateDebugUtilsMessengerEXT f = (
       PFN_vkCreateDebugUtilsMessengerEXT)(vkGetInstanceProcAddr(
       instance, "vkCreateDebugUtilsMessengerEXT"));
10     return f(instance, pCreateInfo, pAllocator, pDebugMessenger);
11 }
```

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 need to destroy the debug messenger and destroy our vulkan instance. DestroyDebugUtilsMessengerEXT is an

extension function proxy.

```
1  #ifdef _DEBUG
2  DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
3  #endif
4
5  vkDestroyInstance(instance, nullptr);
```

<div align="center">Listing 2.8: Vulkan Instance Cleanup</div>

2.2 Open A Window

After creating our Vulkan instance we need to 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.

```
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 );
```

<div align="center">Listing 2.9: Creating a window handle using Win32 API</div>

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
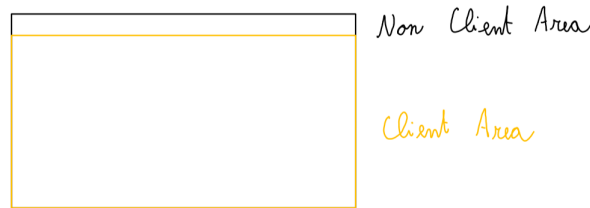takes the whole window width and the whole window height as arguments.

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 need to destroy our window and
unregister its class.

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

Listing 2.13: Window Cleanup

Chapter 3

Open The Window

Chapter 4

Clear The Window

Chapter 5

Our First Pipeline

Chapter 6

Vertex Buffer

Chapter 7

Staging Buffer

Chapter 8

Uniform Buffer

Chapter 9

Depth Buffer

Chapter 10

Setting Up A Simple Scene

Chapter 11

Blinn-Phong Lighting

Chapter 12

Multisample Anti Aliasing

Chapter 13

Conclusion

Appendix A

Appendix

Bibliography

[1] 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`".

[2] Instance - Vulkan Tutorial. "`https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance`".

[3] Introduction - Vulkan Tutorial. "`https://vulkan-tutorial.com/`".

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

[5] Validation layers - Vulkan Tutorial. "`https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Validation_layers`".

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

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