Vulkan

Emanuele Franchi

February 2, 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 Initialize VkInstanceCreateInfo struct

To access any of the functionalities offered by Vulkan we first have to create a Vulkan instance. To do this we call vkCreateInstance. When calling this function we need to pass a pointer to a VkInstanceCreateInfo struct. This struct collects all the information needed to configure our Vulkan instance.

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
createInfo.enabledLayerCount = layerCount;
createInfo.ppEnabledLayerNames = layers;
createInfo.enabledExtensionCount = extensionCount;
createInfo.ppEnabledExtensionNames = extentions;
```

Listing 2.1: VkInstanceCreateInfo initialization

2.1.1 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.

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

Listing 2.2: VkApplicationInfo initialization

2.1.2 Layers

While we initialize oru VkInstanceCreateInfo struct, we can specify the layers that we want to enable.

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.

```
const char* const layers[] =
{
    #ifdef _DEBUG
    "VK_LAYER_KHRONOS_validation",
    #endif
    // other layers ...
};
```

Listing 2.3: 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.

2.1.3 Extensions

While we initialize our VkInstanceCreateInfo struct, we can specify the instance extensions that we want to enable.

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 2.4: 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 2.5: 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 2.6: 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.

2.2 Creating the Vulkan instance

At lats, we can create our Vulkan instance with a simple function call. This call will load all the layers and the extensions specified in our VkInstanceCreateInfo struct.

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

Listing 2.7: Create Vulkan instance

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.