BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

EASY VULKAN

SNADNÝ VULKAN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TIMOTEJ HALÁS**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2021/2022

Master's Thesis Specification

Student: **Halás Timotej, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Computer Graphics and Interaction
Title: **Easy Vulkan**
Category: Computer Graphics
Assignment:

1. Familiarize yourself with the API Vulkan and other graphics APIs.
2. Design a C++ Vulkan wrapper library allowing fast and easy prototyping of graphics and computing applications.
3. Implement the designed C++ library.
4. Evaluate implemented library, create examples and suggest possibilities for the continuation of the project.
5. Create a demonstration video.

Recommended literature:

- Vulkan - A Specification. The Khronos Vulkan Working Group. https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf.
- V-EZ API Documentation. Advanced Micro Devices - Version 1.1.0, 2018-08-25. https://gpuopen-librariesandsdks.github.io/V-EZ/.
- Vulkan Memory Allocator. 2017-2021 Advanced Micro Devices. https://gpuopen.com/vulkan-memory-allocator/.

Requirements for the semestral defence:

- Items 1, 2 and the core of the library.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Milet Tomáš, Ing., Ph.D.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: November 1, 2021

Abstract

While older graphics APIs (Application Programming Interface) like OpenGL or DirectX of version 11 and lower are still commonly used nowadays, newer APIs especially DirectX 12 and Vulkan bring many enhancements like better performance, native Ray-tracing on supported hardware, and more efficient CPU and GPU usage. Performance and efficiency enhancements are the results of the nature of DirectX 12 and Vulkan APIs. Both are quite low-level APIs. That means that GPUs can be controlled on a much lower level which results in much more code that needs to be written to get similar results as when an older API is used. This thesis presents a new framework, vkEasy, that encapsulates Vulkan API in a way that most of its features stay usable, but makes it much easier to use Vulkan API for rendering or compute operations. Source code contains examples that were implemented using vkEasy to show simplicity of vkEasy and to compare it to raw Vulkan code. Average 94 % reduction in needed lines of code was observed.

Abstrakt

Zatiaľ čo staršie grafické API (Application Programming Interface) ako OpenGL alebo DirectX verzie 11 a nižšej sa v súčasnosti stále bežne používajú, novšie rozhrania API, najmä DirectX 12 a Vulkan, prinášajú mnohé vylepšenia, ako je lepší výkon, natívny Ray-tracing na podporovanom hardvéri a efektívnejšie využitie CPU a GPU. Vylepšenia výkonu a efektívnosti sú výsledkom povahy rozhraní DirectX 12 a Vulkan API. Obidve sú pomerne nízkoúrovňové API. To znamená, že GPU je možné ovládať na oveľa nižšej úrovni, čo má za následok oveľa viac kódu potrebného, aby boli dosiahlnuté podobné výsledky ako pri použití staršieho rozhrania API. Táto práca predstavuje nový framework, vkEasy, ktorý zapuzdruje Vulkan API takým spôsobom, že väčšina jeho funkcií zostáva použiteľná, ale výrazne uľahčuje používanie Vulkan API na vkresľovacie alebo výpočtové operácie. Zdrojový kód obsahuje príklady, ktoré boli implementované pomocou vkEasy, aby ukázali jednoduchosť vkEasy a porovnali ho s kódom napísaným v čistom Vulkane. Bolo pozorované priemerne 94% zníženie potrebných riadkov kódu.

Keywords

Vulkan, easy, SPIR-V, GLSL, graphics, framework, library, simplification, render graph, VMA, Shaderc

Kľúčové slová

Vulkan, jednoduchý, SPIR-V, GLSL, grafický, framework, knižnica, zjednodušenie, render graf, VMA, Shaderc

Reference

HALÁS, Timotej. *Easy Vulkan.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Rozšírený abstrakt

Hlavným cieľom tejto práce bolo vytvoriť framework s názvom vkEasy, ktorý zjednodušuje prácu s GPU za využitia Vulkan API. Framework je určený pre ľudí, ktorí majú záujem o rendering alebo využitie výpočtového výkonu GPU. Vulkan je moderné výpočtové a grafické API, ktoré umožňuje využívať výpočtový výkon GPU veľmi efektívne, avšak za cenu zložitosti kódu. Úplné pochopenie Vulkan API nie je ľahká úloha a vkEasy sa snaží uľahčiť prístup k funkciám Vulkan aj bez zložitých znalostí jazyka Vulkan.

Vulkan je veľmi komplexné API, pretože je potrebné napísať veľmi veľa kódu pre vytvorenie hocijakého Vulkan objektu. Týchto objektov Vulkan obsahuje veľmi veľa a pre napísanie funkčného kódu je potrebná znalosť týchto objektov. Napríklad jednoduché vykreslenie trojuholníka vyžaduje vytvorenie minimálne 18 Vulkan objektov. Nehovoriac o tom, že pre vytvorenie množstva z týchto objektov je potrebné vytvoriť a inicializovať veľmi veľa zložitých štruktúr. Stručne povedané, na vykreslenie jednoduchého trojuholníka pomocou čistého Vulkanu je potrebných približne 800 až 900 riadkov kódu v jazyku C. Veľkú časť z tohto kódu možno určiť z kontextu použitia a odložením vytvorenia objektu, kým nebude známy celý kontext. vkEasy tohto princípu využíva. Užívateľ musí najskôr zadefinovať všetko čo je potrebné pre jeho aplikáciu no v tom momente ešte žiadny Vulkan objekt neexistuje. Až keď užívateľ zavolá funkciu kompilácie programu sa na pozadí vytvoria všetky potrebné Vulkan objekty bez toho aby užívateľ musel písať obrovské množstvo kódu pre ich manuálne vytvorenie.

Ďalšia náročná tematika vo Vulkan API je synchronizácia prístupu k zdrojom (obrazové a dátové buffere). Keďže GPU dokážu vykonávať prácu paralelne vo viacerých vláknach, je potrebné synchronizáciu ju vykonať manuálne. vkEasy rieši tento problém automaticky a užívateľ synchronizáciu nemusí riešiť. vkEasy využíva snímkový graf, ktorý slúži ako abstrakcia vykresľovaného snímku alebo bežných výpočtových operácií na sériu úloh. Každej úlohe je potrebné priradiť zdroje, ktoré budú využité a z takéhoto grafu je potom možné určiť akým spôsobom bude riešená synchronizácia.

V neposlednom rade je správa pamäte vo Vulkan API tiež zložitou témou. Pre vytvorenie obrazového alebo dátového bufferu je potrebné najskôr alokovať pamäť vytvorením objektu vkDeviceMemory, ďalej vytvorenie objektu vkImage alebo vkBuffer a nakoniec ich napojenie. A na toto všetko je tak isto potrebné napísať veľa kódu. Časť tejto problematiky je vo frameworku vkEasy vyriešené pomocou knižnice Vulkan Memory Allocator. Pri vytváraní objektov je tiež potrebné poznať akým spôsobom budú objekty vkImage a vkBuffer využívané a toto sa dá zistiť z už vyššie spomenutého snímkového grafu.

Ďalšou výhodou frameworku vkEasy je podpora pre písanie shaderov v jazykoch HLSL a GLSL. Čistý Vulkan podporuje len binárny jazyk SPIR-V, ktorý nie je primárne určený ako jazyk pre užívateľov. vkEasy tiež dokáže vykresľovať do viacerých okien a používať viac grafických kariet paralelne.

Aktuálna implementácia vkEasy znižuje množstvo potrebného kódu a nutnosť porozumieť Vulkanu na hlbšej úrovni riešením týchto problémov. Sú dosiahnuté veľmi veľké redukcie potrebného kódu, až okolo 94 %. Framework vkEasy je napísaný v jazyku C++ a ako zostavovací systém je použitý CMake. vkEasy je podporovaný a otestovaný na operačných systémoch Windows a Linux.

Easy Vulkan

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Tomáš Milet. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Timotej Halás
May 25, 2022

Acknowledgements

I would like to thank my supervisor Tomáš Milet for his assistance, great and helpful consultations and ideas which made framework more user-friendly.

Contents

Chapter 1

Introduction

The main goal of this project is to create a framework named vkEasy simplifying work with GPUs (Graphics Processing Unit) for people who might be interested in rendering or using the computational power of GPUs. Vulkan is a modern compute and graphics API which allows using of most of the computational power GPUs offer very efficiently, but at the cost of code complexity for users. Full Vulkan understanding is not an easy task and vkEasy tries to make access to Vulkan features much easier even without complex Vulkan knowledge.

Why is Vulkan so complex? First of all, there is a lot of boilerplate code that needs to be written for creating any Vulkan objects. For example, simple triangle rendering requires a minimum of 18 Vulkan objects to be created. Not to mention that for the creation of a lot of those objects, a lot of complex structures need to be created and initialized. Summed up, to render a simple triangle using raw Vulkan, around 800 to 900 lines of code are needed. A lot of this code can be determined from the context of usage and by postponing object creation until the whole context is known.

Secondly, synchronization of access to resources (textures and buffers) is also not an easy task in Vulkan and needs to be done manually. This can be solved by using a frame graph. How the frame graph work is briefly described in Section 3.3.

Last but not least, memory management is also a difficult topic in Vulkan, luckily library VMA (more on that in Section 5.3) that does this automatically already exists, and is utilized in vkEasy.

The actual implementation of vkEasy reduces a lot of boilerplate code and the necessity to understand Vulkan on a deeper level by solving those problems. It will hopefully make users want to use Vulkan more and make it easy for them. There is work needed to be done, but actual results are promising as can be seen in Section 6.

The second chapter contains information about the actual state of the implemented framework, the most important features of vkEasy, and related work in this field. The third chapter describes what is Vulkan, what are problems with Vulkan are, and the description of what is frame graph. The fourth chapter contains the description of classes available in vkEasy and their usage. Information about used libraries, technologies, and some implementation details of vkEasy can be found in the fifth chapter. The sixth chapter shows some examples of usage of vkEasy and measurements of reduction of code using vkEasy compared to using raw Vulkan. The evaluation of the work can be found in the seventh chapter.

Chapter 2

Actual state of vkEasy and related work

As Vulkan is a low-level and high-performance API it requires a lot of boilerplate code. Therefore, to write even a really simple program that uses Vulkan needs too much code which is not too user-friendly. vkEasy solves this and makes usage of Vulkan much easier and reduces needed code by around 94 %. Multiple examples are implemented, buildable and available in framework source code. Results of comparisons of raw Vulkan code and code with the same functionality written in vkEasy are available in Section 6. vkEasy is open-source and available on GitHub [14]. It was also presented at the student conference Excel@FIT at the Faculty of Information Technology. This chapter contains available features of vkEasy and a comparison with other related frameworks.

2.1 Features of vkEasy

Framework vkEasy makes working with Vulkan API easier by removing the need for a lot of boilerplate code needed by Vulkan API. It also implements features that are not available in Vulkan by default and uses third-party libraries that helped to make the implementation of some of these features easier. Here are some of the main features offered by vkEasy:

Compute and Graphics pipelines support

vkEasy by default supports Compute and Graphics pipelines. It is also planned to extend support for the last remaining RayTracing pipeline. The actual framework design should make it quite easy to extend by the RayTracing pipeline.

Task graph-based work execution

Writing programs in vkEasy involves first creating a graph node, which can be imagined as one graphics or compute pipeline. It can also encapsulate more nodes into one more complex task. Then resources that will be used are assigned to the node. The last step is recording the order of node execution and compiling the graph. This graph can be then executed. To make this possible frame graph is utilized which is described more in Section 3.3.

Multiple GPUs usage in parallel

vkEasy supports creating more instances of one GPU or more GPUs and executing work on more devices. It is possible to use multiple devices at once in Vulkan but as of now, multi-device synchronization is not supported by Vulkan itself. But manual synchronization on the CPU is possible.

GLSL and HLSL support

Vulkan supports only SPIR-V shading language which is not user-friendly at all. vkEasy uses library Shaderc which supports the compilation of GLSL and HLSL into SPIR-V therefore both are supported in vkEasy by default. This is described in more detail in Section 5.2.

Automatic memory management

To create resources in Vulkan, a lot of boilerplate code is needed. vkEasy uses Vulkan Memory Allocator that reduces this boilerplate code. More about memory management can be found in Section 3.2.3 and about Vulkan Memory Allocator in Section 5.3.

Automatic memory access synchronization

Vulkan needs explicitly specified memory access barriers. vkEasy does this automatically by utilizing a frame graph (more in Section 3.3) to collect information about resource usage and then correctly placing memory barriers.

Automatic data transfer to and from GPU

In Vulkan to access GPU memory, it is mostly not simply mapping memory and copying data. Depending on memory type it is necessary to create a staging buffer and enqueue copy command from or to this staging buffer. vkEasy does this automatically and the user just needs to set the data that should be copied.

Easy rendering into multiple windows and dynamic windows resizing

vkEasy supports creating multiple windows and rendering into them easily. It also supports dynamic windows resizing by recreating the swap chain automatically without user involvement. The usefulness of this feature can be experienced for example in Microsoft Flight Simulator 2020. It has a feature that allows opening some of the in-game HUDs (Head-Up Display) in other windows, so it does not obstruct the in-game view. This can be helpful for multi-monitor systems and at least for me, being able to move in-game HUDs to a secondary monitor made the gaming experience much better. How this looks can be seen in Figure 2.1.

Support for Linux and Windows Operating Systems

Vulkan supports a lot of platforms by default. vkEasy contains examples that also serve as a testing platform for building and running on both Windows and Linux Operating Systems. Designed Window System Integration (WSI) also supports creating windows both on Windows (Desktop Window Manager) and Linux (both X and Wayland display servers) OSes.

Figure 2.1: This figure shows a screenshot from Microsoft Flight Simulator 2020 (flying over the Faculty of Information Technology) and shows its feature to open in-game HUD in the other than the game window. It is possible to open multiple windows but only one is shown because more windows on one monitor would be too much.

Easy integration of framework into projects

Examples available with source code serves also as base building blocks for potential developers so they can get to know the framework faster. The framework is developed in C++ language and built using CMake build tools so it can be easily compiled. Build was tested with Microsoft Visual C++ Compiler (MSVC) and GNU C++ Compiler (g++). It is also developed under an MIT license so everyone can use it for any purpose. GIT version control system is used for easier development. The framework is published on GitHub [14].

2.2 Related Work

Vulkan is still quite a young graphics API. The first version of the Vulkan specification was released on February 16th, 2016 [4]. There are already many big game companies using Vulkan for rendering their games and proving that Vulkan makes games run faster on the same hardware compared to DirectX 11 or OpenGL, but those are mostly closed-source. There are also a few open-source higher-level rendering frameworks built on Vulkan making work with it easier. But I found only two of them implement a frame graph (more in Section 3.3). And as this vkEasy also implements frame graph, only those two I considered as related to this project.

The first of these two frameworks is Granite [13] and the second one is Pumex [17]. I started studying code and examples and found that both have quite different approaches to simplifying Vulkan and there were reasons I did not like either of them. With Granite, I dislike the fact that while it uses a render graph implementation in the background it is not accessible by the user. It looks like the developers tried to implement a public API similar

to OpenGL. So it can be useful for users who are used to OpenGL. Pumex enables the users to use frame graph openly but it is sometimes really confusing how to use it because all of its classes can be instantiated without a parent object and users who do not know connections between objects can be confused same as I was when I started implementing vkEasy with zero knowledge about Vulkan. This is one of the things this vkEasy tries to solve. There is a hierarchy of classes starting with the Context object and each class has its parent class and can be instantiated only by its parent which makes it easier for the user to understand what can be done with each object.

2.2.1 Contributions of vkEasy compared to Pumex and Granite

These are some of the contributions vkEasy brings compared to related frameworks:

- Even simpler use of Vulkan API.
- Vulkan API still accessible.
- Frame graph with direct access.
- Object instantiation from the parent.

Chapter 3

Programming and working with GPUs

This chapter contains the description of Vulkan API. The chapter will also contain descriptions and details of some of the most important objects in Vulkan, how they are connected to each other and what they are used for in programming. There will be also a brief introduction to programming GPUs using Shading Languages and SPIR-V. Different methods of sending work to GPU will be described. Resource management and synchronization of tasks are also a very important part of this chapter and will be described in detail in this chapter. Lastly, a brief description and comparison of already existing Vulkan frameworks will be included.

3.1 Vulkan

Vulkan is a graphics and compute open standard API that provides high-efficiency, cross-platform access to modern GPUs. Created and evolved by the Khronos Group standards consortium, Vulkan satisfies the needs of software developers in fields as varied as game, mobile, and workstation application development. Vulkan's explicit API design enables efficient implementations on platforms that span a wide range of mobile, console, embedded and desktop hardware using the Windows, Linux, and Android operating systems. The API provides a multi-threading-centric design to leverage modern multi-core CPUs and provide access to GPUs via multiple parallel command queues. Some of the latest Vulkan features include ray tracing, bindless resources, and shader programming using GLSL or HLSL. Vulkan is not tied to a specific platform and enables developers to write GPU code that is portable to diverse devices and operating systems. Definition of Vulkan is taken from NVIDIA Developer [12].

While there is quite a big selection of graphics APIs Vulkan was chosen for this project because of its cross-platform availability because cross-platform support is planned for Windows and Linux at least. Also, there is the possibility to achieve some performance gains compared to other cross-platform APIs like OpenGL.

3.1.1 Vulkan Objects

An important part of learning the Vulkan API just like any other API is to understand what types of objects are defined in it. Every Vulkan object is a value of a certain type prefixed by `Vk`. These prefixes, like the `vk` prefix for function names, are eliminated from

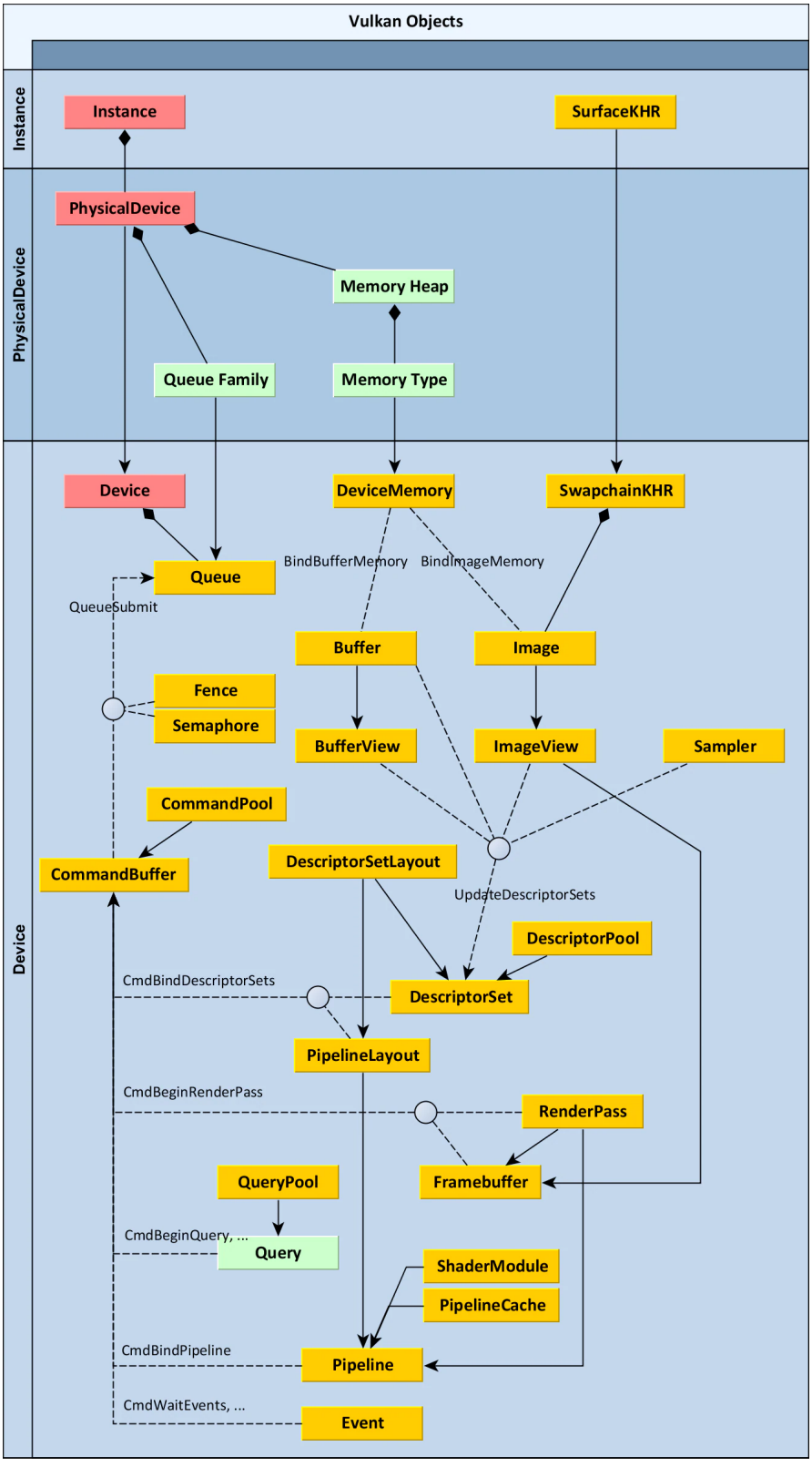

Figure 3.1: This figure shows a diagram containing all the Vulkan objects and some of their relationships. Those relationships shows mainly the order in which objects should be created one from another. Image is taken from AMD GPUOpen [18].

the schematic for clarity. Sampler in the diagram, for example, denotes the existence of a Vulkan object type called `VkSampler`. These types should not be considered as ordinal integers or pointers. Their values should not be interpreted in any way. They should be viewed as opaque handles that can be handed from one function to the next, and should, of course, be destroyed when no longer required. Green-background objects (Figure 3.1) do not have their own types and instead, they are represented by a `uint32_t` numeric index within their parent object, such as `Queries` within `QueryPool`.

The order of creation is represented in Figure 3.1 by solid lines with arrows. To create a `DescriptorSet`, for example, an existing `DescriptorPool` must be specified. Composition is represented by solid lines with a diamond. That means that this object does not need to be created because it already exists inside its parent object and can be retrieved from it. `PhysicalDevice` objects can be enumerated from an `Instance` object, for example. Other relationships, such as submitting various commands to a `CommandBuffer`, are represented by dashed lines.

There are three sections in the diagram in Figure 3.1. Each section has a central object, which is highlighted in red. All other objects in a section are created from that main object, either directly or indirectly. The function `vkCreateSampler`, for example, takes `VkDevice` as its first parameter when creating a `Sampler`. For clarity, relationships to the main objects are not drawn on this diagram. This entire section is inspired by AMD GPUOpen's Understanding Vulkan Objects [18] and Vulkan Specification [11].

Instance

The first object that must be created is `Instance`. It keeps track of all application-specific Vulkan states. It should only be used once in a program and also represents the connection between an application and the Vulkan runtime. When creating a `Instance`, all required instance layers (such as the Validation Layer) and instance extensions must be specified.

Physical Device

`PhysicalDevice` represents a specific Vulkan-compatible device, such as a graphics card available to host that implements complete Vulkan specification. From `Instance`, all compatible devices can be enumerated and their `vendorID`, `deviceID`, and supported features, as well as other properties and limits, can be queried. All available types of queue families can be enumerated by `PhysicalDevice`. Those queue families can support one or more queue types. Types contain graphics queue, compute queue, transfer queue or sparse binding queue.

A `Memory Heap` represents a particular RAM pool. It can abstract a portion of video RAM on a dedicated graphics card, a motherboard's system RAM for the integrated graphics card, or any other host or device-specific memory that the driver wants to expose. When allocating memory, the Memory Type must be specified. Memory blobs that are visible to the host have different Memory Type than those that are coherent (between CPU and GPU), and that are cached. Depending on the device driver, different combinations of these types can be used. Memory Heaps and Memory Types can be enumerated from `PhysicalDevice`.

Device

`Device` is an object that represents a logical or opened device. It is an instance of `PhysicalDevice`'s implementation with its own states and resources independent of other logical devices. This is one of the main objects that after its initialisation it is ready to create all other objects. The features that will be enabled must be specified during device creation. Some of them are essential, such as anisotropic texture filtering. All queues that will be used, their number, and their queue families must be specified.

Queue

`Queue` is an object that represents a command queue that will be executed on the device. Using the function `vkQueueSubmit`, all of the work to be done by the GPU is requested by filling `CommandBuffers` and submitting them to `Queue`. Different `CommandBuffers` can be sent to each of the queues, such as the main graphics queue and the compute queue. Asynchronous compute can be enabled in this way, which can result in a significant speedup if done correctly. Queue families also determine which commands are supported by `Queue`. Transfer queue supports only transfer commands, compute queue only compute commands, etc., but queue can support multiple queue families.

Command Pool

The `CommandPool` object is a simple object that can be used for allocating `CommandBuffers`. It belongs to a particular queue family and `CommandBuffers` which were allocated from specific `CommandPool` must be filled only with commands supported by a particular queue family.

Command Buffer

`CommandBuffer` is an object that is used to record commands which can be then submitted to the `Devices`'s `Queue` for execution. `CommandBuffers` can be allocated from a specific `CommandPool`. A command buffer can be used to call a variety of functions, all of which begin with `vkCmd`. They're used to specify the order, type, and parameters of tasks that should be performed after the `CommandBuffer` is sent to a `Queue` and then subsequently consumed by the `Device`.

Buffer, Image and Device Memory

Vulkan supports two primary types of resources. First is `Buffer` which is the simpler one. It is a linear array for any unformatted binary data that just has its length, expressed in bytes.

The second one is `Image`, which is a collection of pixels. It is a multidimensional array of data with a lot of parameters. It can store up to three dimensions and during the creation, various pixel formats (such as `R8G8B8A8_UNORM` or `R32_SFLOAT`). It can also have multiple array layers or MIP levels (or both), resulting in many discrete images. Because it does not always consist of a linear set of pixels that can be accessed directly, `Image` is a separate object type. The graphics driver can manage a different implementation-specific internal format (tiling and layout) for `Images`.

Creating a `Buffer` with a specific length or a `Image` with specific dimensions does not allocate memory for it automatically. It's a three-step process that the developer must

complete manually. The Vulkan Memory Allocator library, which handles the allocation, can be used. To create `Buffe` or `Image`, firstly a `DeviceMemory` must be allocated. Then `Buffer` or `Image` can be created and lastly they must be bound together using function `vkBindBufferMemory` or `vkBindImageMemory`.

As a result, a `DeviceMemory` object must be created as well. It specifies a block of memory with a given length in bytes allocated from a specific memory type which can be enumerated from `PhysicalDevice`. `DeviceMemory` should not be allocated separately for each `Buffer` or `Image`. Instead, larger memory chunks should be allocated and using parts of chunks as backing memory for `Buffers` and `Images`. Allocation is a time-consuming process, and the maximum number of allocations is also limited. All of this information can be requested from `PhysicalDevice`.

Buffer View and Image View

`Buffers` and `Images` are not always used directly in rendering. Another layer, called views, sits on top of them. Using the set of parameters during the creation of the view it is possible to use them to look at underlying data in a certain way. For example, `BufferView` enables shaders to interpret buffer data as formatted data. It can also be used to limit access to buffer to only a subset of buffer data. Similarly, `ImageView` can be used to limit the view to a defined range of MIP levels or array layers, and interpret data as other format or swizzle components.

Sampler

`Sampler` represents the state of an image sampler. It is a set of parameters, like filtering mode, MIP map mode, addressing mode, etc. It is not directly bound to any `Image`.

Surface

`SurfaceKHR` is an object which represents the presentable surface of the window or screen. It can be also thought of as the Vulkan equivalent of a window. Creating a window needs a different approach for each operating system and also different display servers in the same operating system (like Wayland and X11 on Linux). The same applies to the creation of `SurfaceKHR`. For the creation of `SurfaceKHR`, the `Instance` object is required, as well as some operating system specific arguments. These are, for example, instance handle (`HINSTANCE`) and window handle (`HWND`) on Windows.

Swapchain

`SwapchainKHR` represents a collection of images that can be displayed on the `SurfaceKHR` using double or triple buffering. `SurfaceKHR` is needed to create a`SwapchainKHR`. A `Device` is required for this object. is an exception to the requirement of allocating and binding `DeviceMemory` for every `Image`. The `SwapchainKHR` can be queried for `Images` contained in it. The system has already allocated backing memory for these images.

Descriptor Set Layout

`DescriptorSetLayout` acts as a `DescriptorSet` template and a layout must be specified and created to be able to create a `DescriptorSet`. Descriptors are used by shaders to

access resources (Buffers, Images and Samplers). In Vulkan, descriptors do not exist on their own, instead, they are always found in DescriptorSets.

Descriptor Pool

DescriptorPool same as CommandPool is a simple object used to allocate descriptor sets. When creating a descriptor pool, the maximum number of descriptor sets and descriptors of various types that will be allocated from it must be specified.

Descriptor Set

The DescriptorSet represents memory that stores actual descriptors, and it can be configured to point to a specific Buffer, BufferView, Image, or Sampler. A DescriptorSet can be allocated from DescriptorPool. To be able to do it, both DescriptorPool and DescriptorSetLayout are needed. The function vkUpdateDescriptorSets can be used to accomplish this.

Pipeline Layout

PipelineLayout is a rendering pipeline configuration that specifies which types of descriptor sets will be bound to the CommandBuffer. In a CommandBuffer, several DescriptorSets can be bound as active sets to be used by rendering commands. To accomplish this, the function vkCmdBindDescriptorSets can be used. This function also requires another object, PipelineLayout, because multiple DescriptorSets may be bound, and Vulkan needs to know how many and what types of them to expect ahead of time. To create PipelineLayout, an array of DescriptorSetLayouts can be used.

Render Pass

A RenderPass object contains a collection of attachments, subpasses, and dependencies between subpasses, as well as information about how the attachments are used throughout the subpasses. Draw commands must be recorded within a RenderPass instance. Each render pass instance specifies a set of image attachments that are used during rendering. The immediate mode approach can be used in other graphics APIs to render whatever comes next. In Vulkan, this is not possible. Instead, a frame's rendering must be planned ahead of time and divided into passes and subpasses. Subpasses are not separate objects, but they are an essential part of Vulkan's rendering system. When defining a RenderPass in Vulkan, the number and formats of attachments that will be used in that pass are extremely important.

Attachment is Vulkan's name for what is commonly referred to as a render target, an Image that is used as a rendering output. There is no need to point to a specific Image here. It is only necessary to describe their formats. A simple rendering pass, for example, might include a colour attachment with the format R8G8B8A8_UNORM and a depth-stencil attachment with the format D16_UNORM. It should also be specified whether the content of an attachment should be saved, discarded, or cleared at the start of the pass.

Framebuffer

Framebuffer (which is not the same as SwapchainKHR) represents a collection of actual memory attachments (Images) that are used in RenderPass. By specifying the RenderPass

and a set of `ImageViews`, a `Framebuffer` object can be created. Their number and formats must match the `RenderPass` specification. The function `vkCmdBeginRenderPass` must be called whenever rendering of a `RenderPass` begins, and the `Framebuffer` must also be passed to it.

Pipeline

`Pipeline` represents the configuration of the whole pipeline and it contains many parameters. It is one of the largest objects in Vulkan is `Pipeline`, which includes the majority of the previously mentioned objects. One of the parameters is `PipelineLayout`. It specifies the layout of descriptors and push constant layout. Depending on how the pipeline is created can use one of the GPU pipelines. These include compute pipeline, graphics pipeline and ray tracing pipeline. Because it only supports compute-only programs, compute pipeline is the simplest of the three (sometimes called compute shader).

The Graphics pipeline is far more complicated because it includes all of the shader stages such as vertex, fragment, geometry, compute, and tessellation. Its other parameters which can be modified are vertex attributes, primitive topology, backface culling, blending mode, etc. All those parameters that were previously separate settings in much older graphics APIs (DirectX 9, OpenGL), were later grouped into a smaller number of state objects as the APIs progressed (DirectX 10 and 11), and must now be baked into a single big, immutable object with today's modern APIs like Vulkan. A new `Pipeline` must be created for each different set of parameters required during the process. The function `vkCmdBindPipeline` can then be used to set it as the current active `Pipeline` in a `CommandBuffer`.

The last one is the ray tracing pipeline which is the newest one and is not supported on older hardware.

Shader Module

`ShaderModule` represents a piece of shader code, possibly partially compiled, but not yet capable of being executed by the GPU. Shader compilation in Vulkan is a multi-stage process. Vulkan does not support any high-level shading languages such as GLSL or HLSL. Instead, Vulkan accepts SPIR-V (section 3.1.2), an intermediate format that any higher-level language can be translated into. To create a `ShaderModule`, the buffer filled with SPIR-V data is needed.

Pipeline Cache

`PipelineCache` is a helper object that can be used to speed up pipeline creation. It's a simple object that can be passed in during `Pipeline` creation, but it significantly improves performance by reducing memory usage and pipeline compilation time. The driver can use it internally to store some intermediate data, potentially speeding up the creation of similar `Pipelines`. A `PipelineCache` object's state can be saved and loaded to a binary data buffer, which can then be saved on disk and used the next time the application is run. It is suggested to use them.

Fence

`Fence` is a synchronization object which can be used by the host to wait until a task has been successfully completed. On the host, it can be polled, waited for, and manually unsigned.

It doesn't have its own command function, but it is passed when calling `vkQueueSubmit`. The appropriate fence is signalled once the submitted queue is complete.

Semaphore

`Semaphore` is a synchronization object that can be used to manage access to resources across multiple queues. `Semaphore` can be created without any configuration parameters. It can be signalled or waited on as part of command buffer submission, as well as with a call to `vkQueueSubmit`, and it can be signalled on one queue (for example compute) and waited on another (for example graphics).

Event

`Event` is the last of Vulkan synchronization objects. Using the functions `vkCmdSetEvent`, `vkCmdResetEvent`, and `vkCmdWaitEvents`, it can be waited on or signaled on the GPU as a separate command submitted to `CommandBuffer`. It can also be set, reset, and waited on (through polling calls to `vkGetEventStatus`) from one or more CPU threads. `Event` can be created without parameters.

3.1.2 Shading Languages

Shading languages are the interface used to program key parts of the modern graphics pipeline which have previously been fixed-function state machines without programmability. With shading languages, the vertex transformation and lighting fixed function pipeline is replaced by vertex program instructions supplied by the application, and key parts of the rasterization pipeline, mainly texture environment and fog are replaced by fragment program instructions supplied by the application. The key to understanding shaders is that vertex shaders are fed by graphics primitives like triangles and lines with vertex attributes like colour, texture coordinates, position, and other generic attributes, for each vertex the program is executed, and the output is screen space primitives with similar types of per-vertex data to the input. The output of a vertex shader is then transformed to the viewport and clipped by the fixed function pipeline. The primitive is rasterized using prudicing per fragment interpolated values for the results of the vertex shader. The fragment shader program is then executed for each pixel produced by the aforementioned interpolation process using the interpolated output of the vertex shader as the input to the fragment shader. The fragment shader outputs colour attributes and possibly other outputs like zbuffer depth (outputs supported depend on specific shader language feature support). The output from the fragment shader is depth tested and stencil tested using fixed-function hardware and if passed the colour is blended with the destination pixel using the fixed-function hardware. This section was inspired by Khronos Wikipedia about shading languages [6].

SPIR and SPIR-V

SPIR (Standard Portable Intermediate Representation) was initially developed for use by OpenCL and SPIR versions 1.2 and 2.0 were based on LLVM. SPIR has now evolved into a cross-API intermediate language that is fully defined by Khronos with native support for shader and kernel features used by APIs such as Vulkan – called SPIR-V.

SPIR-V is catalyzing a revolution in the ecosystem for shader and kernel language compilers used for expressing parallel computation and GPU-based graphics. SPIR-V enables

high-level language front-ends to emit programs in a standardized intermediate form to be ingested by Vulkan, OpenGL, or OpenCL drivers. SPIR-V eliminates the need for high-level language front-end compilers in device drivers, significantly reducing driver complexity, enabling a broad range of language and framework front-ends to run on diverse hardware architectures, and encouraging a vibrant ecosystem of open-source analysis, porting, debug, and optimization tools.

For developers, using SPIR-V means that kernel source code no longer has to be directly exposed, kernel load times can be accelerated, and developers can choose the use of a common language front-end compiler, improving kernel reliability and portability across multiple hardware implementations. This section was inspired by Khronos's SPIR Overview [3].

3.2 Main problems of raw Vulkan API

Here are listed main problems in the raw Vulkan framework which do not make it easy to work with and possible solutions:

3.2.1 A lot of boilerplate code

Some features of Vulkan or GPU are disabled by default. During the initialization process, any of those features must be explicitly enabled. This can be annoying for users because during the implementation, they will probably many times come back to the initialization where some settings are missing or incorrect. However, the correct initialization can be determined from the context of the program and the functionality required by the user. For this to be possible, it is necessary to delay the initialization of the objects until sufficient information is available. Examples of such behaviour are layers, extensions, device features, and more.

When creating Vulkan Instance, layers and extensions that will be used are needed. The same applies to Vulkan Device which needs to know what extensions, features, and queues will be used. Also, an already initialized Vulkan Instance is needed to create a device. Vulkan Images and Buffers need allocated Device Memory which needs initialized Vulkan Device. The same applies to a lot of other Vulkan Objects (see Figure 3.1). And even to create any object a lot of information is required. But most of the time this information can be determined automatically by knowing the specifications of the program which will be executed on the device.

vkEasy offers a solution to this by collecting information about the context of usage by specifying the whole program, all resources, and work to be done without creating any Vulkan objects. Deferred initialization of all needed Vulkan objects is done when the whole context is known. Each program will have specified workflow and dependencies sooner than all Vulkan objects will be created therefore framework will know how to create a lot of objects with no user involvement.

3.2.2 Synchronization of access to buffers and images

There are more types of synchronization in Vulkan. Firstly, there is memory access synchronization. Barriers are used for this type of synchronization. The next type is synchronization between multiple queues. Semaphores are used for this type of synchronization. Those are quite hard to get right and are quite user-error-prone. Frame graph execution of work approach will be used which can quite easily track all resource usage dependencies and

according to those tracked data use barriers and semaphores automatically and correctly. More about frame graph can be found in Section 3.3.

3.2.3 Memory management

Compared to older graphics APIs like OpenGL memory allocation and resource (buffer and image) creation in Vulkan is much more complicated. Just like everything else in Vulkan, because it is a low-level and high-performance API, it needs a lot of boilerplate code. Also `VkDeviceMemory` is allocated individually from creating `VkBuffers` and `VkImages`. `VkBuffers` and `VkImages` must be bound to `VkDeviceMemory` what adds an extra level of indirection. Various hardware vendors provide different types of memory. Because of that driver must be queried for supported memory heaps and memory types. Also recommended practice is to allocate bigger chunks of memory and assign parts of chunks to particular resources. This can introduce fragmentation.

3.3 Frame graph

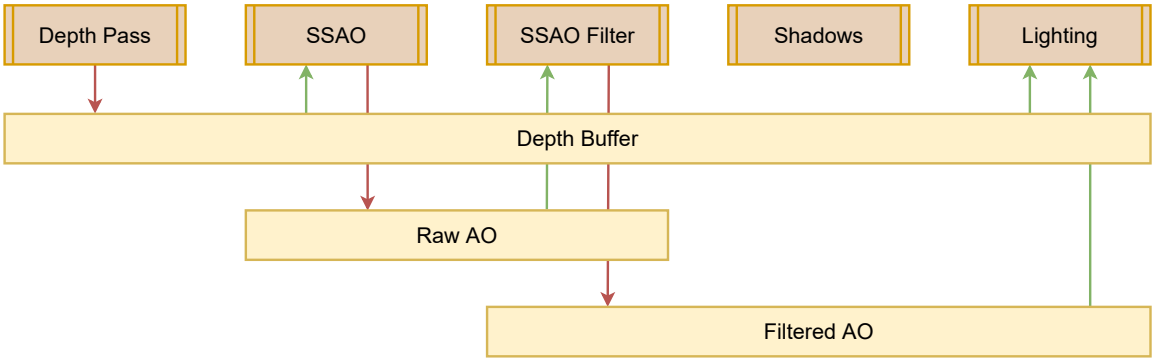

Figure 3.2: This graph consists of five render tasks (brown rectangles) and three resources (yellow rectangles). Red arrows represent writing to the resource and green arrows represents reading the resource. On this graph, it can be seen that placing memory barriers is pretty straightforward. For example `Depth pass` render task writes to `Depth Buffer` resource and `SSAO` render task reads from same resource and should be executed after `Depth pass` render task. That means that a memory barrier must be placed between the execution of those two. Same can be seen with `SSAO` and `SSAO Filter` render tasks and `Raw AO` resource. `Shadows` render task has no inputs and outputs therefore if it is not marked as having side effects it will be culled from executing. Image is taken from Yuriy O'Donnell's presentation at GDC Expo 2017 [15].

Information in this section is from Yuriy O'Donnell's presentation at GDC Expo 2017 [15]. A frame graph, also known as a Render graph is a rendering abstraction that describes a frame as a directed acyclic graph of render tasks and resources. A render task is any compute or graphics task to be performed as part of the rendering pipeline. The resource is a buffer or image created, read, or written by the render task. An example of a simple Frame graph can be seen in Figure 3.2.

Frame graph helps to build high-level knowledge of the entire frame. This knowledge then can be used to simplify resource management and rendering pipeline configuration. It also makes asynchronous compute tasks easier to implement. Placing resource barriers,

which can be quite hard to do right in the case of complex rendering pipelines, is also a lot easier. Frame graph also helps to create self-contained and efficient rendering modules for example node which implements a deferred shading pipeline that can be reused quite easily. Also, graphs can be visualized and the same applies to frame graphs. Visualization of the graph can help with debugging complex rendering pipelines.

Using frame graph consists of three phases namely the Setup phase, Compile phase and Execute phase.

Setup phase

In **Setup phase**, render tasks and resources are defined. These resources are then assigned as inputs and outputs to and from render tasks and the order of render tasks is specified. In this phase, no GPU commands are used and resources are virtual, which means, they do not have memory assigned on GPU yet and information about rendering operations for the frame is gathered. For example, when creating image resource, dimensions, format, initial data, etc.. is specified here.

Compile phase

Next phase is **Compile phase**. In this phase, the graph is being traversed and unreferenced render tasks and resources are culled. It is possible to mark render tasks as having some side effects, so they are not culled. During graph traversal, resource lifetimes are calculated and resource bind flags are derived based on usage.

Execute phase

Last phase is **Execute phase**. Here, all render tasks are iterated in the correct order and GPU commands of each render task are executed. Also, resources, which were not culled, are created whenever they are needed and destroyed when they are not needed anymore.

Chapter 4

Framework design

This chapter contains information about the design of vkEasy, a description of its classes, and how are these classes interconnected. It also contains some code snippets showing some of the use cases and how vkEasy can be used.

4.1 vkEasy's Classes

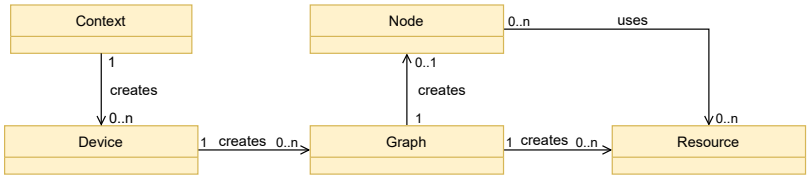

Figure 4.1: This diagram contains simplified class relationships of the vkEasy framework. Class `Context` is singleton class. Class `Device` encapsulates logical device. One hardware device can be used in multiple logical device instances. Class `Graph` encapsulates frame graph. `Node` class is an abstract class that represents one render task. Class `Resource` is also an abstract class and can represent different types of Buffers and Images.

All vkEasy classes are encapsulated in the C++ namespace `vk::easy`. Base classes of vkEasy are classes `Context`, `Device`, `Graph`, `Node` and `Resource`. To run the render or compute task on GPU it is needed to create `Node` which executes this task. If it is needed input and output resources for this node can be specified. But to execute the node it must be enqueued into a `Graph` in which it will be executed. The `Node` can be created from the `Graph` object and must be executed on the same `Graph`. The `Graph` must be created from the `Device` and it will also be executed on the same `Device`. And lastly, the `Device` must be created from `Context`. These relationships can be seen in the diagram in Figure 4.1 shows a simplified class relationship diagram. Ownership of objects is designed so destructors of all objects are called in correct order same as Vulkan needs. This section contains more details about these and other vkEasy classes. Each subsection of this section corresponds exactly by name to one of the vkEasy classes.

4.1.1 Class Context

vkEasy's main class is singleton class `Context`. This class serves for creating logical devices (vkEasy's class `Device`) and takes care of creating a Vulkan instance. When creating a

`Device`, the used GPU is selected automatically based on the support of features or can be explicitly selected by the user. It is also possible to manually add Instance extensions and layers if some of them are needed. It is possible to call method `setDebugOutput()` to enable or disable debug output. If debug output is enabled, the `Context` class will automatically add debugging layer and extension, which will write Vulkan debug messages into the console containing useful data if something is not working properly.

4.1.2 Class Device

The `Device` class represents the logical instance of a hardware device (GPU). It creates Vulkan Device, Queue and CommandPool objects when the whole context of the application is known. It also automatically selects the most powerful hardware device if the device is not explicitly selected by the developer. To instantiate class `Graph` from `Device`, method `createGraph()` can be called. During the execution of the program, underlaying Vulkan Device object is available with a call of method `getLogicalDevice()`. If some task is executing on GPU it is possible to call the blocking method `wait()`, which will block until work is done. This class also takes care of initialising the Vulkan Memory Allocator library. More about Vulkan Memory Allocator can be found in Section 5.3.

4.1.3 Class Graph

Class `Graph` implements frame graph principles described in Section 3.3. `Graph` can only be instantiated from class `Device`. It is possible to create `Nodes` (more in Section 4.1.5) and `Resources` (more in Section 4.1.7). There is templated method to instantiate any class which inherits `Node` or `Resource` classes. There are helper methods (for example `createGraphicsNode()`) to create all existing `Nodes` and `Resources`. With method `setNumberOfFramesInFlight())` it is possible to set how many frames can be prerecorded in advance while one of them is being rendered. Each graph can use one window and to get this window method `getGLFWWindow()` can be called to create and get this window. More about windows can be found in Section 4.1.8. Framebuffer (more details in Section 4.1.9) can be also created from `Graph` using method `createFramebuffer()`. Method `compile()` serves for compiling frame graph and should be called after whole context (`Nodes`, `Resources`, `Framebuffers` and other vkEasy objects) is created and initialized. Then after compilation, method `execute()` can be called and will execute this `Graph`. During execution `Nodes` can ask for Vulkan CommandBuffers so they can record commands into them. After execution of all nodes in `Graph` all recorded CommandBuffers are sent to `Device` and executed in Vulkan Queue object owned by `Device`.

4.1.4 Class MemoryAllocator

This class serves as a lightweight wrapper for the Vulkan Memory Allocator library (more about Vulkan Memory Allocator in Section 5.3). Vulkan Memory Allocator is a C library so its functions for creating and destroying objects must be called manually. This class uses the RAII principle, so when `MemoryAllocator` class is instantiated create function of Vulkan Memory Allocator is called and when it goes out of scope or is explicitly destroyed. This makes sure that cleaning up is done every time it is needed. It contains functions for creating both Images and Buffers easily which use Vulkan Memory Allocator in the background. This class is instantiated once for each `Device` and each `Resource` can query for Image or Buffer through `Device`.

4.1.5 Class Node

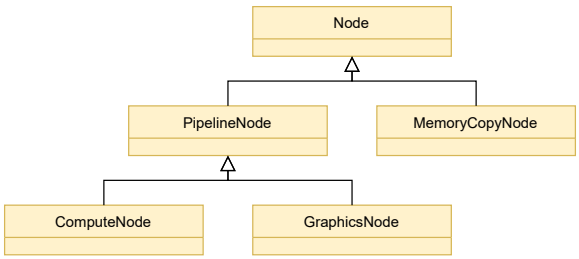

Figure 4.2: This diagram contains all classes existing in vkEasy and inheriting from abstract class Node. Nodes are executable classes and it is to execute them after enqueueing to `Graph`.

Abstract class `Node` serves as an interface for defining the render task. As of now, two classes implement class `Node`. Classes inheriting from class `Node` can be instantiated only from class `Graph`. As frame graph can cull nodes from execution it is possible to set the node as culling immune here. This can be helpful for example if the `Node` has some side effects and is being culled by the `Graph`. It is also possible to add required Vulkan Device extensions which will be then collected from all nodes to enable the required features. Figure 4.2 shows an inheritance diagram of all existing classes with `Node` as the base class.

Class MemoryCopyNode

`MemoryCopyNode` really simple class and serves for copying data from one resource to another. It is possible to use it to copy data from the buffer to buffer, image to image or buffer to image and vice versa. An example of usage can be copying data to the device's local memory. This type of memory cannot be directly accessed by mapping it on the CPU but first staging buffer must be created. Staging buffer has host visible memory type which can be mapped and read or written by CPU. Then this node can be used to copy data to or from GPU.

Class PipelineNode

Abstract class `PipelineNode` implements class `Node` and serves as base for all nodes that uses Vulkan Pipeline object. For now, only Compute and Graphics Pipelines are implemented in classes `ComputeNode` and `PipelineNode`. Implementing RayTracing Pipeline should be as easy as implementing a node for example `RayTracingNode`, initialising RayTracing Pipeline creation info and RayTracing should work. A lot of code which is the same for all Pipelines is already implemented in `PipelineNode`. It automatically takes care of all `Resources` used in `Node` by building Vulkan PipelineLayout and DescriptorSets objects and all objects needed to create them. What is only missing for classes inheriting `PipelineNode` is to create a Vulkan Pipeline object by implementing a pure virtual method `buildPipeline()` and Pipeline type-specific features.

Class ComputeNode

Compute pipeline is the simplest type of pipeline. It has only one `ShaderStage` (more on class `ShaderStage` in Section 4.1.6) and its only property is setting dispatch size. Whole

implementation of compute pipeline after inheriting `PipelineNode` is only around 40 lines of code. To get `ShaderStage` there is method `getComputeShaderStage()`.

Class GraphicsNode

A graphics pipeline is a much more complex pipeline than compute pipeline. It has much more properties which can be set. In vkEasy, a lot of them are still hidden but making them visible is only a matter of creating getters and setters and then calling protected method `needsRebuild()` which will make sure that before the next usage of the pipeline it will be rebuilt with this new property set accordingly. `GraphicsNode` contains getters for its shader stages (for now only Vertex and Fragment shader stages). Method `setFramebuffer()` serves for setting `Framebuffer` (more in Section 4.1.9) which will be rendered into. To define vertex attribute of some Vertex Buffer `defineAttribute()` can be called. Index buffer can be used by setting it with method `setIndexBuffer()`.

4.1.6 Class ShaderStage

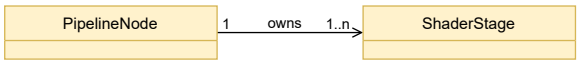

Figure 4.3: This diagram shows relationship between class `ShaderStage` and classes inheriting from `PipelineNode`. One `ShaderStage` must belong only to one `PipelineNode` but `PipelineNode` can contain multiple `ShaderStages`. That's because for example Graphics pipeline consists of Vertex `ShaderStage`, Fragment `ShaderStage`, Geometry `ShaderStage`, etc.

Class `ShaderStage` implements one programmable pipeline stage like vertex or fragment stage in the graphics pipeline. The object of this class can be acquired from nodes inheriting `PipelineNode`. In the background creates the Vulkan ShaderModule object and fills all info needed to create the Vulkan PipelineShaderStage object. It supports loading SPIR-V, GLSL or HLSL shading languages. To support GLSL and HLSL code it uses Shaderc (more in Section 5.2) library to compile it into SPIR-V. More about shading languages can be found in Section 3.1.2. It owns one or more objects of class `ShaderStage`, which uses Shaderc library for automatic compilation to SPIR-V. SPIR-V code can be set to `ShaderStage` with method `setShaderData()`. If method `setShaderFile()` is used it determines usage of Shaderc compiler based on extension of file. If the file extension is `spv` it loads the file as SPIR-V code and if not it uses Shaderc library to compile the file. Figure 4.3 shows `ShaderStage`'s relationship with `PipelineNode`.

4.1.7 Class Resource

Abstract class `Resource` is for implementation of different types of Buffers and Images like Uniform Buffers, Storage Buffers, Attachment Images, etc. Here Vulkan Image or Buffer object is stored after acquiring it from `MemoryAllocator` as written in Section 4.1.4. It automatically takes of copying data to GPU what is needed if `Resource` lives in a device's local memory and is not directly accessible from the CPU. It automatically creates `StagingBuffer` and `MemoryCopyNode` and uses them when needed. Persistence can be also set for each resource. It means that it will not be created and destroyed in each frame but it will persist until it is not destroyed manually. This can be useful for big read-only data like

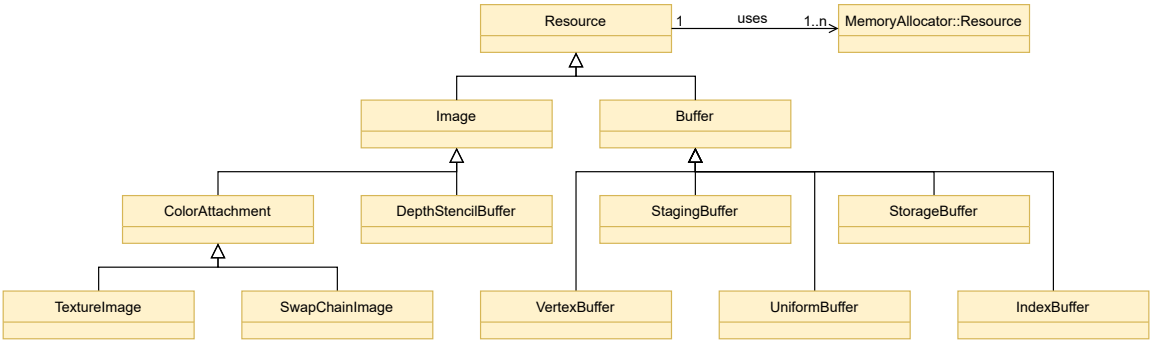

Figure 4.4: This diagram contains all classes existing in vkEasy and inheriting from abstract class `Resource`. `Resources` can be assigned to `Nodes` and then after enqueueing nodes to `Graph` it is possible to calculate resource lifetimes and places where pipeline barriers must be placed. Also as mentioned in Section 4.1.4, `Resources` can ask for Vulkan Image or Buffer which is wrapped in class `MemoryAllocator::Resource`.

textures or vertex buffers. If rendering into Window is used Vulkan SwapChain object will also exist. According to the count of images in SwapChain non-persistent, the same number of underlying Vulkan Image or Buffer objects must be created and it is done automatically in the background. And each frame's correct resource index is used depending on the actual frame index. It also takes care of recording usage by nodes and then it is possible to place buffer and image barriers in the correct places. Classes inheriting abstract class `Resource` can be instantiated only from class `Graph`. Figure 4.4 shows an inheritance diagram of all existing classes with `Resource` as the base class.

Class Buffer

Class `Buffer` serves as the base class for all resources which use Vulkan object Buffer. During the execution of `Graph`, it is possible to get the underlying Vulkan Buffer object by calling method `getVkBuffer()`. Most of the classes which inherit from `Buffer` only set the correct buffer usage flags needed by Vulkan to create the buffer. But their functionality can greatly differ. More about the available `Buffer` types in vkEasy is below.

Class StagingBuffer

`StagingBuffer` is a buffer which is always host visible and therefore mappable and writable by CPU. This buffer can be used as a destination or source buffer for `MemoryCopyNode` and used as a transfer medium between GPU and CPU.

Class UniformBuffer

`UniformBuffer` serves for creating buffer used as constant data readable by GPU in shaders. It is always available to read from GPU but sometimes it is possible to make it also host visible so the CPU can write or read it directly. It is mostly used for small data.

Class StorageBuffer

`StorageBuffer` is the type of buffer serving as storage for big data like holding data of an entire scene, geometry, etc. Usually, it is a little slower as `UniformBuffer` but can hold

much more data. It is always available to be used from GPU but sometimes can also be host visible.

Class VertexBuffer

`VertexBuffer` is a buffer which can be used for drawing in `GraphicsNode`'s Graphics Pipeline as the source for Vertex attributes. It is possible to set Vertex data with the method `setVertices()`.

Class IndexBuffer

`IndexBuffer` can be used for indexed drawing in `GraphicsNode`'s Graphics Pipeline. It is possible to set indices with the method `setIndices()`.

Class Image

Class `Image` serves as the base class for all resources which use Vulkan object Image. During the execution of `Graph`, it is possible to get the underlying Vulkan Image object by calling method `getVkImage()`. Most of the classes which inherit from `Image` only set the correct image usage flags needed by Vulkan to create the image. But their functionality can differ. All images have getters and setters for different properties like format, dimensions, number of MIP levels, etc. More about actually available Image types are written below.

Class DepthStencilBuffer

Even though class `DepthStencilBuffer` contains Buffer in its name its underlying Vulkan object Image and therefore it is inheriting vkEasy's `Image` class. The naming is the same in raw Vulkan so it was kept. `DepthStencilBuffer` serves for Z-buffering or Stencil testing or both. It can be used in `GraphicsNode`'s Graphics Pipeline. For now, only Z-buffering works but support for Stencil testing is also planned. It is possible to set a clear value using method `setClearValue()`;

Class ColorAttachment

`ColorAttachment` class serves as the base for all Image classes which can be used as color attachments in the graphics pipeline which means that the graphics pipeline can use them as render targets. By default `ColorAttachment` object can be instantiated from `Framebuffer` object (more about `Framebuffer` object in Section 4.1.9). It is possible to set a clear color using method `setClearColor()`;

Class SwapChainImage

`SwapChainImage` inherits class `ColorAttachment` and serves as render target for graphics pipeline which can be drawn into window. It is a special case of `Resource`. As mentioned in Section 4.1.7 each Resource can be chosen to be persistent or not. `SwapChainImage` is persistent by default but still can contain multiple underlying Vulkan Image objects. The number of images can be specified during the creation of the Vulkan SwapChain object and is chosen by vkEasy automatically. Other resources used in the same graph will have the same number of underlying Vulkan Image of Buffer objects depending on the number of Vulkan Images created with the Vulkan SwapChain object. Only one `SwapChainImage` can

exist for one `Graph` and can be acquired from classes inheriting class `WSI` (more about WSI in Section 4.1.8). These relationships can be seen in Figure 4.5.

Class TextureImage

`TextureImage` inherits class `ColorAttachment` and serves as image which can be sampled from in shaders in graphics pipeline.

4.1.8 Class WSI

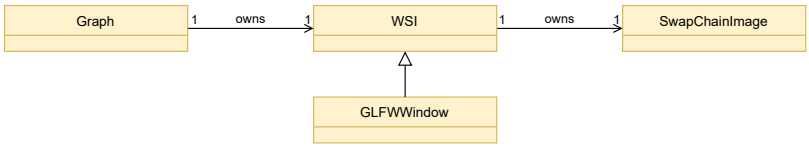

Figure 4.5: This diagram shows relationships of other classes to class `WSI`. Each `Graph` can own only one `WSI` and `WSI` owns one `SwapChainImage`. Also each `WSI` is unique to its creating `Graph` and `SwapChainImage` is unique to its creating `WSI`. `WSI` is for now implemented only by one class `GLFWWindow`.

Abstract class `WSI` serves as an abstraction over windows. It can be implemented using different libraries like Simple DirectMedia Layer (SDL), Graphics Library Framework (GLFW), etc. More about that in Section 5.4. GLFW is already implemented and available for testing. `WSI` should exist only once per `Graph`. Internally it creates `SwapChainImage` which can be used as render target in graphics pipeline in `GraphicsNode`. Implementing classes must provide the Vulkan SurfaceKHR object. Relations to other classes can be seen in Figure 4.5.

Class GLFWWindow

`GLFWWindow` is simple class implementing `WSI` using GLFW library (more in Section 5.4) to create windows and Vulkan SurfaceKHR object.

4.1.9 Class Framebuffer

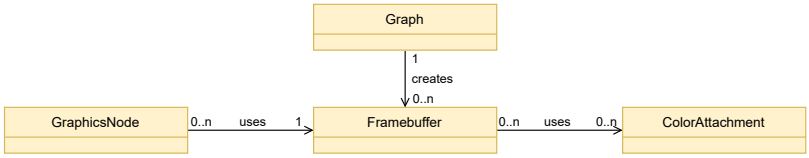

Figure 4.6: This diagram shows relationships of other classes to class `Framebuffer`. `Framebuffer` can be created from `Graph` and as many `ColorAttachments` can be assigned to it as needed. `Framebuffer` then can be used by `GraphicsNode`.

`Framebuffer` class has more use cases. Firstly it groups all render targets which are somehow related to each other, they must share same resolution, and can serve as render targets, depth or stencil buffers. One render pass contains one or more subpasses where subpass consists of one graphics pipeline (more in Section 3.1.1). All `GraphicsNodes` which

share same `Framebuffer` objects are grouped to one render pass where each `GraphicsNode` is taken as one subpass. Also it is possible to set classes implementing `WSI` to it as Window. This makes sure that all render targets are set to correct resolution depending on resolution of window which is rendered into. In Figure 4.6 relationships to other classes can be seen.

4.2 Interface design and usage

Now when classes and their purpose are known, this section shows an example of usage of vkEasy. vkEasy was developed from top to bottom. That means that firstly its expected interface and classes were designed and then the backend was implemented. This section shows some important parts of the vkEasy's interface needed for drawing rotating textured triangle using vkEasy. Parts of code are from example 5 which is described in Section 6.1.5 and is available in source code [14]. Figure 6.4 shows what the output of the code described looks like. Library OpenGL Mathematics (GLM) is used to represent vectors and matrices (types in namespace glm::) in this example.

Including vkEasy

Listing 4.1: Pretty straightforward code just showing how to include vkEasy into project.

```
#include <vkEasy/vkEasy.h>
```

Creating all of needed vkEasy objects

Listing 4.2: Code below shows how easy it is to create any of vkEasy objects. First `Device` object must be created, then `Graph` object can be created from it. All other objects are then created from `Graph`.

```
    auto& device = vk::easy::Context::get().createDevice();
    auto& graph = device.createGraph();
    auto& framebuffer = graph.createFramebuffer();
    auto& vertexBuffer = graph.createVertexBuffer();
    auto& indexBuffer = graph.createIndexBuffer();
    auto& uniformBuffer = graph.createUniformBuffer();
    auto& graphics = graph.createGraphicsNode();
    auto& textureImage = graph.createTextureImage();
    auto& window = graph.getGLFWWindow(800, 600, "Graphics Test");
```

Preparing `VertexBuffer`

Listing 4.3: Code bellow shows creates four vertices with position, color and uv attributes in vector `vertices`. Then previously created `vertexBuffer` is filled with `vertices`.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
    glm::vec2 uv;
};
const std::vector<Vertex> vertices =
  { { { -0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f }, { 1.0f, 0.0f } },
    { {  0.5f, -0.5f }, { 0.0f, 1.0f, 0.0f }, { 0.0f, 0.0f } },
    { {  0.5f,  0.5f }, { 0.0f, 0.0f, 1.0f }, { 0.0f, 1.0f } },
    { { -0.5f,  0.5f }, { 1.0f, 1.0f, 1.0f }, { 1.0f, 1.0f } } };
vertexBuffer.setVertices(vertices);
```

Preparing `IndexBuffer`

Listing 4.4: Code below shows filling of vector `indices`, and filling previously created `indexBuffer` object with vector `indices`.

```
const std::vector<uint16_t> indices = { 0, 1, 2, 2, 3, 0 };
indexBuffer.setIndices(indices);
```

Preparing `TextureImage`

Listing 4.5: Code below shows filling of previously created `textureImage` with pixel data loaded by some third-party image loader. `texWidth`, `texWidth` are dimensions of texture loaded from file and `pixels` is pointer to data with size of `imageSize`. Loading data part was skipped because any image loader can be used.

```
auto& textureImage = graph.createTextureImage();
textureImage.setDimensions(vk::Extent3D(texWidth, texHeight, 1));
textureImage.setDimensionality(vk::ImageType::e2D);
textureImage.setData(pixels, imageSize);
```

Preparing `Framebuffer` and `GLFWWindow`

Listing 4.6: Code below shows how previously created `window` can be set to `framebuffer` object. This will ensure that all `framebuffer`'s attachments will be resized according to window size. Then `framebuffer` is assigned to `GraphicsNode graphics`. `ColorAttachment` obtainable from `window` is then set to `graphics` and will be accessible from shaders at layout 0.

```
framebuffer.setWindow(window);
graphics.setFramebuffer(framebuffer);
graphics.setColorAttachment(window.getAttachment(), 0);
```

Preparing vertex and fragment shaders

Listing 4.7: Here is shown how easy it its to set shaders to `graphics` node. Corresponding `ShaderStage` is obtained and then shader file can be set. Shaders are written in GLSL and internally compiled into SPIR-V using Shaderc

```
graphics.getVertexShaderStage().setShaderFile("shader.vert");
graphics.getFragmentShaderStage().setShaderFile("shader.frag");
```

Using `indexBuffer`, `vertexBuffer` and defining its attributes

Listing 4.8: This code shows how attribute can be defined using `vertexBuffer` and `graphics` node. Method `defineAttribute` of `GraphicsNode` takes as first parameter location as accessible from shaders. Second parameter is offset in buffer, third parameter is stride in buffer and last parameter is buffer itself. There is also untemplated version where format can be set manually. Last line shows setting `indexBuffer` to `graphics` node.

```
graphics.defineAttribute<glm::vec2>(0, offsetof(Vertex, pos),
                                    sizeof(Vertex), &vertexBuffer);
graphics.defineAttribute<glm::vec3>(1, offsetof(Vertex, color),
                                    sizeof(Vertex), &vertexBuffer);
graphics.defineAttribute<glm::vec2>(2, offsetof(Vertex, uv),
                                    sizeof(Vertex), &vertexBuffer);
graphics.setIndexBuffer(&indexBuffer);
```

Using `uniformBuffer` and `textureImage`

Listing 4.9: Code below shows creating descriptors. Code is same for any node inheriting from class `PipelineNode`. First parameter is resource, second and third are binding and set under which resource is available in shader.

```
graphics.createDescriptor({ &uniformBuffer }, 0, 0);
graphics.createDescriptor({ &textureImage }, 1, 0);
```

Enqueueing `GraphicsNode` to `Graph`

Listing 4.10: Code below shows enqueueing `graphics` node to `graph` and than compiling `graph` before it can be executed. Here any number of nodes can be enqueued.

```
graph.enqueueNode(graphics);
graph.compile();
```

Graph execution

Listing 4.11: And the last part is execution itself. The first lines show creating structure, which holds rendering data which will fill `uniformBuffer` every frame. Model, view and projection matrices respectively. This structure is filled with calculated data for every frame and will rotate the rectangle in the scene. Calculations of these matrices are skipped in this code example. While cycle will end if the close button of the window is clicked.

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
std::vector<UniformBufferObject> ubo;
ubo.resize(1);
while (!window.shouldClose()) {
    // calculating model, view and projection matrices
    // and filling ubo vector with data
    uniformBuffer.setData(ubo);
    graph.execute();
}
```

Result

After compiling and running the program, a window should open and a rotating and textured rectangle should be seen. Same as can be seen in Figure 6.4.

Chapter 5

Implementation

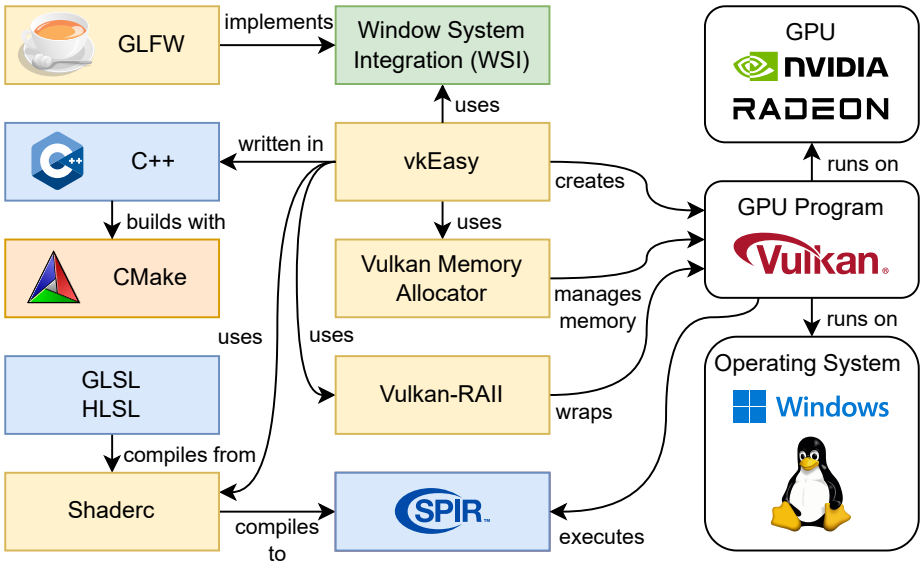

Figure 5.1: This figure shows the graph of used technologies and libraries and their relationship to vkEasy. Yellow rectangles are representing libraries, blue rectangles represent used and programmable languages, orange rectangles represent used tools, and lastly, green rectangles represent classes of vkEasy.

vkEasy is written in C++ language and it is required for the compiler to be able to build code written in C++17. CMake is used as the build system and at least version 3.16 is required. vkEasy is buildable on Windows and Linux operating systems and was tested with Microsoft Visual C++ (MSVC) compiler and GNU C++ (g++) compilers. This chapter contains information about libraries used by vkEasy. Graph with relationships of tools, languages, and libraries to vkEasy can be seen in Figure 5.1.

5.1 Vulkan C++ wrapper

Vulkan is a graphics API implemented in the C language. While it is possible to use raw Vulkan C API vkEasy uses C++ language there are Vulkan C++ wrappers that do work

with Vulkan just a little easier. There are two well-known C++ wrappers, namely Vulkan-Hpp and Vulkan-RAII (available at Khronos Vulkan-Hpp GitHub repository [7][10]).

Vulkan-Hpp

Vulkan-Hpp provides header-only C++ bindings for Vulkan C API. Its goal is to improve the developer's Vulkan experience without introducing CPU runtime costs. It adds new features like type safety for enums and bitfields, STL container support, exceptions, and simple enumerations. More information, examples and source code can be found at Khronos Vulkan-Hpp GitHub repository [7].

Vulkan-RAII

Vulkan-RAII adds additional C++ layer on the top Vulkan-Hpp. It uses all the enums and structure wrappers from Vulkan-Hpp. It also provides new wrapper classes for the Vulkan handle types but in a more refined way than Vulkan-Hpp. As its name already suggests it follows the RAII principle (RAII: Resource Acquisition Is Initialization). Instead of creating Vulkan handles with `vkAllocate` or `vkCreate` and destroying them with `vkFree` or `vkDestroy`, constructor and destructor of corresponding Vulkan handle wrapper is used called. More information, examples and source code can be found also at Khronos Vulkan-Hpp GitHub repository [7]. Programming guide for Vulkan-RAII can be found GitHub repository [10].

Why Vulkan-RAII

Vulkan-RAII is used in vkEasy because of the ease of use of the RAII principle. It also contains simple to use dynamic loader of Vulkan, which means that there is no need to use a dynamic loader library like Volk.

5.2 GLSL/HLSL to SPIR-V compiler

By default, Vulkan accepts only programs written using SPIR-V unlike OpenGL, which also accepts GLSL (OpenGL shading language). There are two probably best-known shading language compilers named Glslang and Shaderc, respectively.

`Glslang`

`Glslang` is the official reference compiler by Khronos Group for the ESSL (OpenGL ES shading language), GLSL (OpenGL shading language) and HLSL. It firstly translates those languages to Glslang's internal abstract syntax tree (ASL). Then ASL is translated to Khronos-specified SPIR-V intermediate language. It is open and free for anyone to use, either from a command line or programmatically. The OpenGL and OpenGL ES are maintaining consistency between the reference compiler and the corresponding GLSL and ESSL specifications. More information and source code can be found at Khronos's glslang GitHub repository [2].

Shaderc

Shaderc is composed of library libshaderc and command line tool glslc. glslc is a command line compiler used for compiling shader strings from GLSL and HLSL to SPIR-V. In the

background, it uses above mentioned Glslang and also SPIRV-Tool. Library libshaderc is an API for accessing glslc functionality. Compared to glslang it comes with a simpler API and increased functionality like support for `#include` directives. More information and source code can be found at Google's Shaderc GitHub repository [5].

Why Shaderc

SPIR-V is not a user-friendly language so library Shaderc [5] is used to make vkEasy compatible with GLSL and HLSL (High-level shader language). Shaderc supports both GLSL and HLSL and it also comes with support for `#include` directives which are very useful. GLSL in OpenGL does not support `#include` directives and if code needs to be reused it must be copied into every shader.

5.3 Memory Management

As mentioned in Section 3.2.3, memory management and resource allocation is quite a difficult topic. There is already a really good library, Vulkan Memory Allocator, created by AMD GPUOpen, which is utilized in vkEasy.

Vulkan Memory Allocator

The Vulkan Memory Allocator (VMA) [8] library is a simple and easy to integrate API, which helps with allocating memory and creation of Vulkan Buffer and Image objects. To make memory allocations and resource creation easier it offers some higher-level functions:

- functions that help to choose the correct and optimal memory type based on the intended usage of the memory.
  - required or preferred traits of the memory are expressed using higher-level description compared to Vulkan flags.
- functions that allocate memory blocks, reserve and return parts of them (`VkDeviceMemory` + offset + size) to the user.
  - library keeps track of allocated memory blocks, used and unused ranges inside them finds best matching unused ranges for new allocations, and respects all the rules of alignment and buffer/image granularity.
- functions that can create an image/buffer, allocate memory and bind it to the corresponding image/buffer – all in one call.
- functions that can defragment already allocated memory.

The library really helped to make memory management in vkEasy much easier and it also has high-quality documentation which helped to get to know it really quickly. This section was inspired by and more information about VMA can be found at AMD GPUOpen [8] and Vulkan Memory Allocator GitHub repository [9].

5.4 Rendering into window

For Vulkan to be able to render into a window it needs a Surface object. Creating the surface object needs system-dependent parameters (as written in Section 3.1.1). To make this easier libraries like Simple DirectMedia Layer (SDL) or Graphics Library Framework (GLFW).

Graphics Library Framework

Graphics Library Framework is an Open Source, multi-platform library for OpenGL, OpenGL ES, and Vulkan application development. It provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc. GLFW natively supports Windows, macOS, Linux and other Unix-like systems. On Linux, both X11 and Wayland are supported. More information about GLFW can be found at GLFW GitHub [1].

Why GLFW

SDL library is a quite complex and big library with a lot of functionality most of which is not needed for testing of vkEasy. GLFW is a lightweight framework and that is the reason why it is used. And as stated in Section 4.1.8 it is possible to use other frameworks like GLFW by inheriting and implementing the abstract class WSI.

Chapter 6

Experiments

This chapter shows how much code can be reduced by using the framework vkEasy. First, it shows implemented and working examples available in source code [14] and then compares lines of code needed to write these examples in raw Vulkan with lines of code needed to write the same example in vkEasy.

6.1 Examples

This section contains examples implemented using vkEasy and their description. They are all available in source code [14]. Compute example 1 was inspired by an example by Sascha Willems [19] and graphics examples by some examples from Vulkan Tutorial [16]. They are written in raw Vulkan and were used to compare usability and lines of code reduction of vkEasy. They also serve as tests if the framework works correctly.

6.1.1 Example 1 – Compute pipeline – Fibonacci sequence

This example serves as a test for the compute pipeline. A simple Fibonacci sequence shader is used to calculate the first 32 numbers of the sequence on GPU and write the contents of the output buffer to the console. Shader source code is taken from minimal headless compute example by Sascha Willems[1]. In vkEasy source code name of this example is vkEasyCompute.

6.1.2 Example 2 – Graphics pipeline – Triangle

This example serves as a basic test of the graphics pipeline. It draws a coloured triangle into the window as shown in Figure 6.1. Triangle is hardcoded in the shader so no vertex buffer is used. Shader source code and inspiration were taken from Vulkan Tutorial[2]. Name of this example in vkEasy source code is vkEasyGraphics.

6.1.3 Example 3 – Graphics pipeline – Vertex and index buffers

This example draws a coloured rectangle (two triangles) into the window as shown in Figure 6.2. The rectangle is now stored in the vertex buffer and the index buffer is also

[1] https://github.com/SaschaWillems/Vulkan/blob/master/examples/computeheadless/computeheadless.cpp

[2] https://github.com/Overv/VulkanTutorial/blob/master/code/15_hello_triangle.cpp

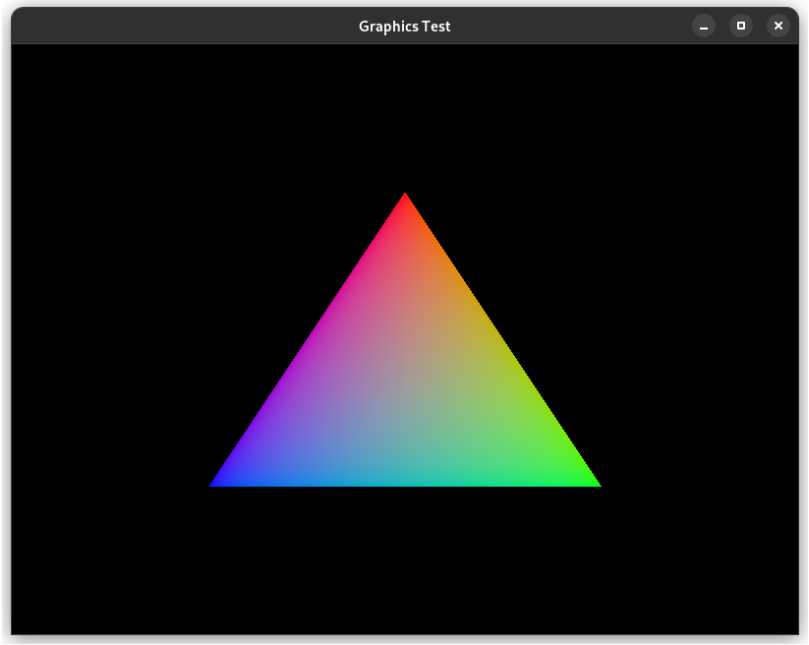

Figure 6.1: Basic test of graphics pipeline drawing shader hardcoded triangle to window.

used. Vertex buffer consists of two attributes one of which is position and the second is colour. Shader source code and inspiration were taken from Vulkan Tutorial[3]. Name of this example in vkEasy source code is vkEasyGraphicsVertexIndexBuffers.

6.1.4 Example 4 – Graphics pipeline – Uniform buffer

This example draws the same rectangle as in example 3 but now also uses a uniform buffer containing model, view, and projection matrices updated every frame causing rotation of triangle in 3D space. What this example looks like is shown in Figure 6.3. Shader source code and inspiration were taken from Vulkan Tutorial[4]. Name of this example in vkEasy source code is vkEasyGraphicsUniformBuffers.

6.1.5 Example 5 – Graphics pipeline – Texture

This example draws the same rotating rectangle as in example 4 but this rectangle is now textured instead of interpolated colour as shown in Figure 6.4. Vertex buffer now contains a new attribute that is texture coordinate. Shader source code, texture, and inspiration were taken from Vulkan Tutorial[5]. The name of this example in the vkEasy source code is vkEasyGraphicsTexture.

6.1.6 Example 6 – Graphics pipeline – Depth buffer

This example draws two rectangles with offset on the z-axis. Each rectangle is the same as in example 5 and this example shows how to use a depth buffer. What this example looks like

[3] https://github.com/Overv/VulkanTutorial/blob/master/code/21_index_buffer.cpp
[4] https://github.com/Overv/VulkanTutorial/blob/master/code/23_descriptor_sets.cpp
[5] https://github.com/Overv/VulkanTutorial/blob/master/code/25_sampler.cpp

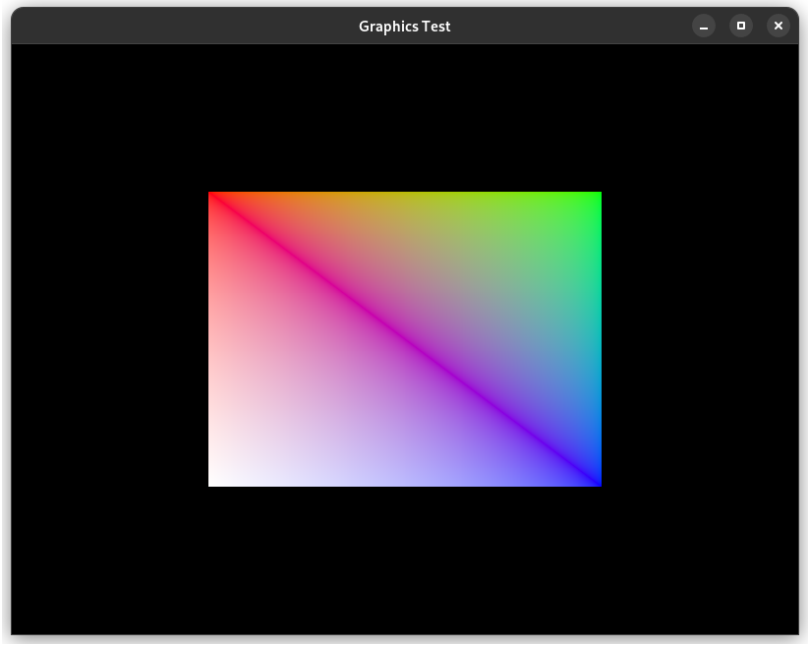

Figure 6.2: Example that draws colored rectangle into window using vertex and index buffer.

is shown in Figure 6.5. Shader source code, texture, and inspiration were taken from Vulkan Tutorial[6]. Name of this example in vkEasy source code is vkEasyGraphicsDepthBuffer.

6.1.7 Example 7 – Graphics pipeline – 3D Model

This example draws a rotating 3D model as shown in Figure 6.6. Shader source code, 3D model, texture, and inspiration were taken from Vulkan Tutorial[7]. Name of this example in vkEasy source code is vkEasyGraphicsModel.

6.2 Code Reductions

This section summarizes code reductions of examples presented in the previous section. Application CLOC was used to count an exact number of lines except for empty lines and comments. For results to be more precise all include directives were removed because they are different for every code. All sources were formatted using the same C++ language formatter so it corresponds to each other also with the format. Also in Sascha's example, there were code parts containing code intended to be used with Android OS which was also removed from counting. Results were as follows:
As seen in Table 6.1, using vkEasy reduces the code needed for using Vulkan by a lot. For examples implemented in this project, the average reduction of lines of code is 94 %. Also from the table, it can be seen that the lowest reduction of 90 % was achieved in the first example and the highest reduction of 97 % in the second example. The first example has the lowest reduction because the raw Vulkan part for creating all necessary objects for the compute pipeline does not create as many Vulkan objects as all other examples. The

[6] https://github.com/Overv/VulkanTutorial/blob/master/code/27_depth_buffering.cpp
[7] https://github.com/Overv/VulkanTutorial/blob/master/code/28_model_loading.cpp

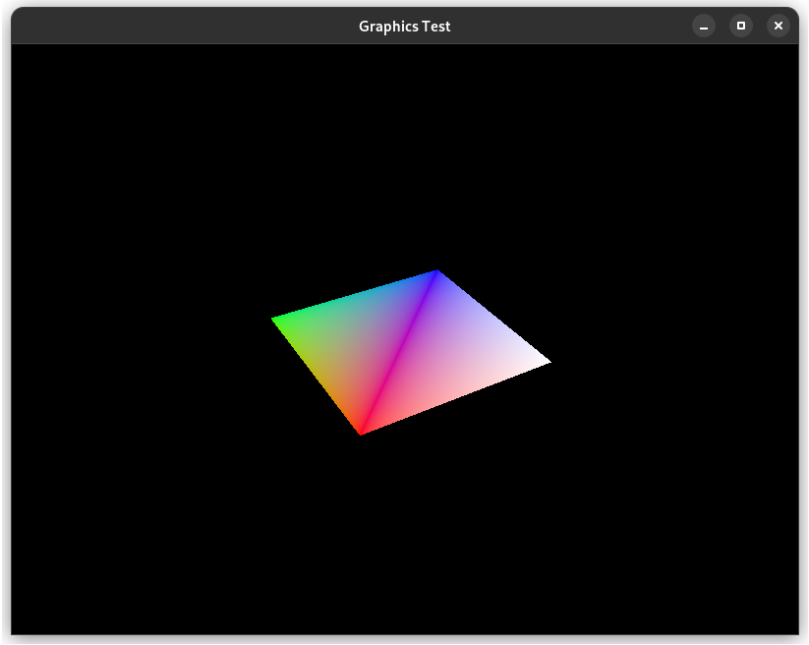

Figure 6.3: Example that draws same rectangle as in example 3 but it is rotating using model, view and projection matrices from uniform buffer.

Table 6.1: Table of lines of code reductions

Example	Raw Vulkan lines	vkEasy lines	Reduction
1	335	33	90 %
2	757	21	97 %
3	958	38	96 %
4	1074	64	94 %
5	1245	80	94 %
6	1359	86	94 %
7	1396	114	92 %

graphics pipeline is much more complex and much more boilerplate code is needed. And that's why the second example achieved the highest reduction. Triangle is hardcoded into shaders so code like loading filling vertex buffers, loading texture and data from disk or loading 3D model from disk is not needed. Therefore reduction is really high in this case.

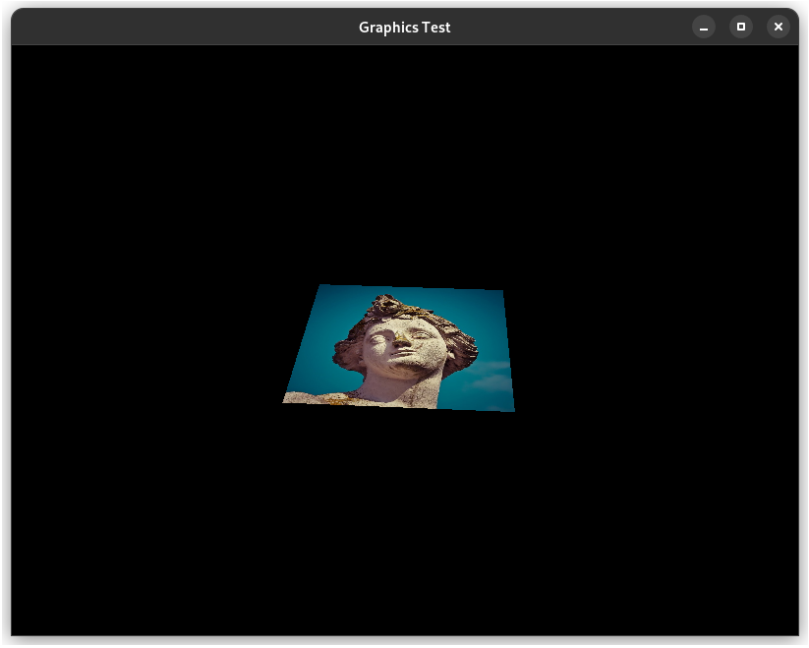

Figure 6.4: This example tests texturing rotating rectangle with loaded texture from file.

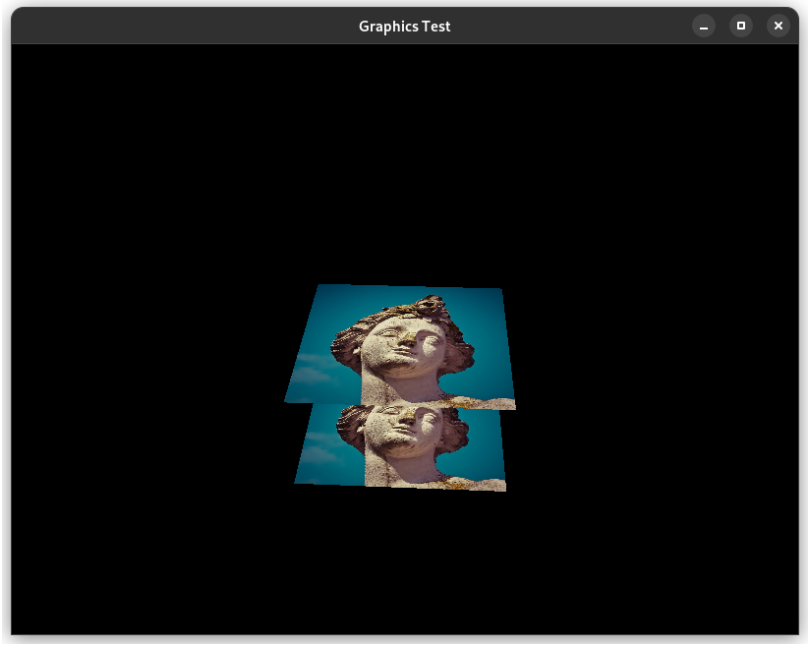

Figure 6.5: Example that shows usage of depth buffer. It draws two textured rectangles with offset in z axis.

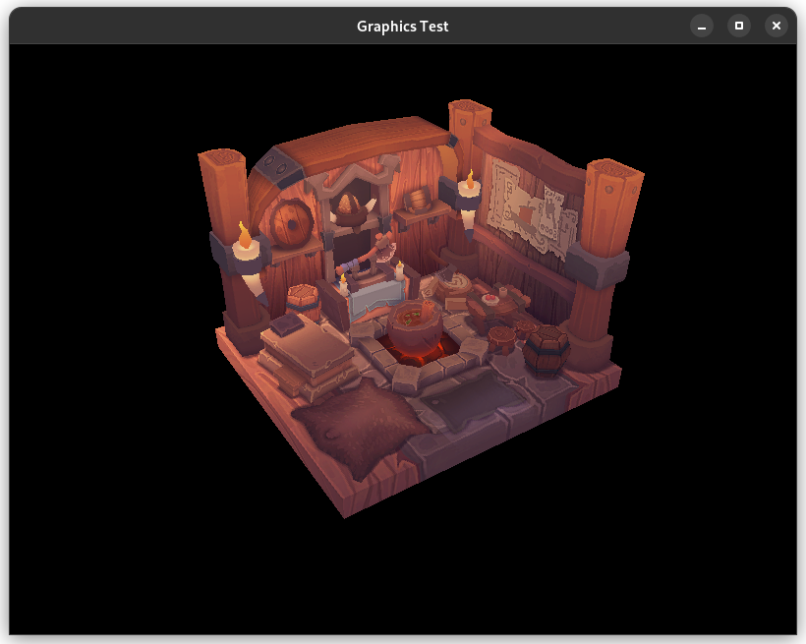

Figure 6.6: This example tests drawing of more complex textured 3D model.

Chapter 7

Conclusion

While Vulkan is a very complex and low-level API, there are ways to make work with it much easier. vkEasy implements deferred Vulkan object creation to hide a lot of boilerplate code. It also implements a frame graph, which makes it easier for the user to think about a frame as a series of graphics or compute tasks, which need to be done to get the final frame or desired compute results. Next, it uses the Vulkan Memory Allocator framework so the user doesn't need to think about complex memory allocation. It also uses the Shaderc library to compile much more user-friendly shading languages such as GLSL or HLSL to SPIR-V which is the language that Vulkan can understand. vkEasy makes it easy to use multiple GPUs for compute and render tasks. It also automatically manages memory and synchronizes access to it. vkEasy also makes it easy to send and read data from GPU. The framework was tested and supports Linux and Windows operating systems and makes it easy to create windows that can be rendered into.
The proposed architecture helps to increase the ease of use of Vulkan and reduces lines of code needed to use GPUs. Specifically as mentioned in Section 6, it reduced needed lines of code in examples on average by 94 %.
Compared to related framework Granite it does not go by way of trying to be similar API like OpenGL but opens possibilities of frame graph for the user. Compared to framework Pumex it has a strict class hierarchy that cannot be disobeyed and makes it easier for the user to understand which class is good for what.
There is still a lot of space for improvements. User testing and feedback on ease of use by users of vkEasy would be really helpful to make it even more user-friendly. Rethinking some parts of the class hierarchy could reduce the complexity of use even more. Bringing support for the ray-tracing pipeline would be also a nice addition. More complex features like MIP mapping, multi-sampling, and other things that are mostly related to Image Vulkan objects and are planned but not supported yet. Also, a lot of features are not yet visible in the graphics pipeline but it is only a matter of creating getters and setters for them. The goal of creating vkEasy was not to develop a good performance framework but to make work with Vulkan easier so there are a lot of things to increase the performance of vkEasy. While automatic memory access synchronization works on the inter-pipeline level, it can be done on the inter-pipeline stage level to increase performance. This can be achieved with shader reflection. For now, there is support only for one universal queue. Support for asynchronous compute queue, separate transfer queue and sparse binding queues would be a nice addition and could increase performance. Also, multi-threaded command buffer recording is planned and should increase performance.

Bibliography

[1] *GLFW* [online]. Community project [cit. 2022-05-21]. Available at: https://github.com/glfw/glfw.

[2] *Glslang* [online]. Khronos Group [cit. 2022-01-10]. Available at: https://github.com/KhronosGroup/glslang.

[3] *The Industry Open Standard Intermediate Language for Parallel Compute and Graphics* [online]. Khronos Group [cit. 2022-01-10]. Available at: https://www.khronos.org/spir/.

[4] *Khronos Releases Vulkan 1.0 Specification* [online]. Khronos Group [cit. 2022-04-22]. Available at: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification.

[5] *Shaderc* [online]. Google Inc. [cit. 2022-01-10]. Available at: https://github.com/google/shaderc.

[6] *Shading languages: General* [online]. Khronos Group [cit. 2022-01-10]. Available at: https://www.khronos.org/opengl/wiki/Shading_languages:_General.

[7] *Vulkan-Hpp: C++ Bindings for Vulkan* [online]. Khronos Group [cit. 2022-01-10]. Available at: https://github.com/KhronosGroup/Vulkan-Hpp.

[8] *Vulkan Memory Allocator* [online]. AMD GPUOpen [cit. 2022-01-10]. Available at: https://gpuopen.com/vulkan-memory-allocator.

[9] *Vulkan Memory Allocator* [online]. AMD GPUOpen [cit. 2022-01-10]. Available at: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator.

[10] *Vulkan_raii.hpp: a programming guide* [online]. Khronos Group [cit. 2022-01-10]. Available at: https://github.com/KhronosGroup/Vulkan-Hpp/blob/master/vk_raii_ProgrammingGuide.md.

[11] *Vulkan® 1.1.215 - A Specification* [online]. The Khronos® Vulkan Working Group [cit. 2022-01-10]. Available at: https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html.

[12] *What Is Vulkan?* [online]. NVIDIA Corporation [cit. 2022-01-10]. Available at: https://developer.nvidia.com/vulkan.

[13] Arntzen, H.-K. *Granite* [online]. [cit. 2022-04-22]. Available at: https://github.com/Themaister/Granite.

[14] Halás, T. *VkEasy* [online]. [cit. 2022-05-19]. Available at: https://github.com/timoti111/vkEasy.

[15] O'Donnell, Y. *FrameGraph: Extensible Rendering Architecture in Frostbite* [online]. [cit. 2022-04-22]. Available at: https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in.

[16] Overvoorde, A. *Vulkan tutorial* [online]. [cit. 2022-04-22]. Available at: https://github.com/Overv/VulkanTutorial.

[17] Paweł. *Pumex library* [online]. [cit. 2022-04-22]. Available at: https://github.com/pumexx/pumex.

[18] Sawicki, A. *Understanding Vulkan® Objects* [online]. [cit. 2022-01-10]. Available at: https://gpuopen.com/learn/understanding-vulkan-objects/.

[19] Willems, S. *Vulkan Example – Minimal headless compute example* [online]. [cit. 2022-04-22]. Available at: https://github.com/SaschaWillems/Vulkan/blob/master/examples/computeheadless/computeheadless.cpp.