Excel@FIT 2022
http://excel.fit.vutbr.cz

Easy Vulkan

Timotej Halás*

Abstract
While older graphics APIs 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, 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 paper presents a new framework, vkEasy, that encapsulates Vulkan API in a way that all of its features stay usable, but makes it much easier to use Vulkan API for rendering or compute operations. Four examples were implemented using vkEasy and compared to raw Vulkan code with an average 94 % reduction in needed lines of code.

Keywords: Vulkan — Frame Graph — Deferred Evaluation

Supplementary Material: Downloadable Code

*xhalas10@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The main goal of this project is to create a framework named vkEasy simplifying work with GPUs 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 3.2.

Last but not least, memory management is also a difficult topic in Vulkan, luckily library VMA (more on that in Section 3.2) 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 4.

2. Related Work

Vulkan is still quite a young graphics API. The first version of the Vulkan specification was released on February 16th, 2016 [1]. 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 3.2). 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 [2] and the second one is Pumex [3]. I started studying code and examples and found that both have quite different approaches to simplifying Vulkan and there were reasons I didn't 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.

2.1 Contributions

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

- Simpler use of Vulkan API
- Vulkan API still accessible
- Frame graph with direct access
- Object instantiation from parent

3. The Proposed Solutions

As Vulkan is a low-level and high-performance API it requires a lot of boilerplate code. Writing even a really simple program that uses Vulkan needs quite much code and it is not too user-friendly. That's why making work with Vulkan easier is the main goal for vkEasy.

3.1 Deferred object creation

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 behavior 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. 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.

3.2 Frame graph

Information in this section is from Yuriy O'Donnell's presentation at GDC Expo 2017 [4]. 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 1.

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.

In **Setup phase**, render tasks and resources are

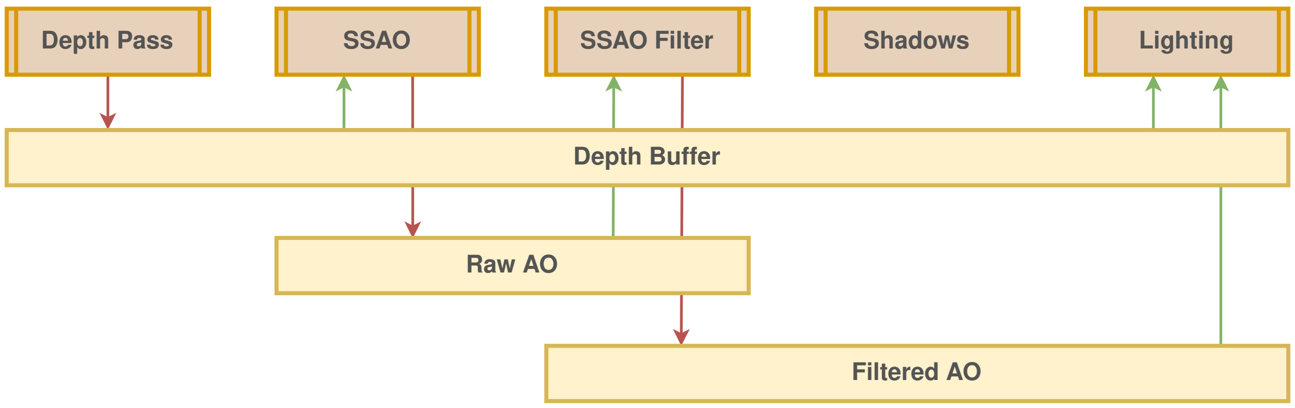

Figure 1. This graph consists of five render tasks (brown rectangles) and three resources (yellow rectangles). Red arrows represent writing to the resource and green arrows represents reading the resource. On this graph, it can be seen that placing memory barriers is pretty straightforward. For example `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 [4].

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.

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.

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.

3.3 Shader compilation

By default, Vulkan accepts only programs written using SPIR-V unlike OpenGL, which accepts also GLSL (OpenGL shading language). 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.

3.4 Vulkan C++ wrapper

Vulkan is a graphics API implemented in C language and as vkEasy uses C++ language, so I decided to use Vulkan C++ wrapper. There are two well known C++ wrappers, namely Vulkan-Hpp [6] and Vulkan-RAII [7].

The goal of the Vulkan-Hpp is to provide header-only C++ bindings for the Vulkan C API to improve the developer's Vulkan experience without introducing CPU runtime cost. It adds features like type safety for enums and bitfields, STL container support, exceptions, and simple enumerations.

Vulkan-RAII is a C++ layer on top of Vulkan-Hpp that follows the RAII principle (RAII: Resource Acquisition Is Initialization). This header-only library uses all the enums and structure wrappers from Vulkan-Hpp and provides a new set of wrapper classes for the Vulkan handle types. Instead of creating Vulkan handles with `vkAllocate` or `vkCreate` functions a constructor of the corresponding Vulkan handle wrapper class is used. And instead of destroying Vulkan handles with `vkFree` or `vkDestroy` functions, the destructor of that handle class is called.

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.

3.5 Memory allocation

Memory allocation and resource (buffer and image) creation in Vulkan is difficult (compared to older graphics APIs like OpenGL) for several reasons:

- it requires a lot of boilerplate code, just like everything else in Vulkan, because it is a low-level and high-performance API
- there is an additional level of indirection: Device Memory is allocated separately from creating Buffer and Image which must be bound to Memory
- driver must be queried for supported memory heaps and memory types and different Hardware Vendors provide different types of memories
- it is recommended practice to allocate bigger chunks of memory and assign parts of chunks to particular resources, but this can introduce fragmentation

There is already a really good library Vulkan Memory Allocator created by AMD GPUOpen, which is utilized in vkEasy.

The Vulkan Memory Allocator (VMA) [8] library provides a simple and easy to integrate API to help with allocating memory and creation of Vulkan Buffers and Images. This library can help game developers to manage memory allocations and resource creation by offering some higher-level functions:

- functions that help to choose the correct and optimal memory type based on the intended usage of the memory
- functions that allocate memory blocks, reserve and return parts of them (Device Memory + offset + size) to the user
- 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

Information about VMA can be found at [8] and GitHub repository [9].

3.6 Framework Design

vkEasy's main class is singleton class `Context`. This class is for creating logical devices (class `Device`) and Vulkan instance. Used GPU is selected automatically based on the support of features or can be explicitly selected by the user. Class `Graph`, that implements frame graph principles, can be instantiated from class `Device`. Automatic synchronization barrier insertion and node culling are working as it is defined in Section 3.2.

Abstract class `Resource` is for implementation of different types of Buffers and Images like Uniform Buffers, Storage Buffers, Attachment Images, Samplers, etc. Here Vulkan Memory Allocator library is used for allocating and creating memory and resource objects.

Abstract class `Node` serves as an interface for defining the render task. As of now, two classes implement class `Node`.

`MemoryCopyNode` is for copying data from one resource to another (for example for copying from CPU accessible buffer to GPU local buffer). Abstract class `PipelineNode` also implements class `Node` and is for all nodes that uses Vulkan Pipeline object. It owns one or more objects of class `ShaderStage`, which uses Shaderc library for automatic compilation to SPIR-V. `ComputeNode` contains and implements compute pipeline and `GraphicsNode` contains and implements graphics pipeline. Both classes implements abstract class `PipelineNode`.

Classes inheriting abstract class `Node` and abstract class `Resource` can be instantiated from class `Graph`. Diagram in Figure 2 shows simplified class relationship diagram.

4. Comparison with raw Vulkan

Compute example by Sascha Willems [10] and some examples from Vulkan Tutorial [11] were used to compare usability and lines of code reduction of vkEasy. Application CLOC was used to count an exact number of lines of code. All include directives were removed because they are different for every code. 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:

Table 1. Table of lines of code reductions

Example	Raw Vulkan (lines)	vkEasy	Reduction
1	335 lines	34 lines	89 %
2	913 lines	22 lines	97 %
3	954 lines	37 lines	96 %
4	1070 lines	63 lines	94 %

As seen in Table 1, using vkEasy implemented in this project, the average reduction of lines of code of tested examples is 94 %. Examples can be found in the source code.

5. Conclusions

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

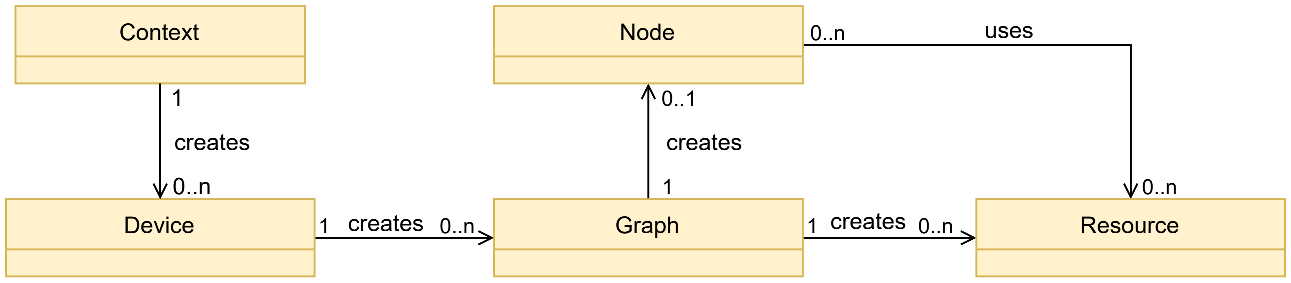

Figure 2. This diagram contains simplified class relationships of the vkEasy framework. Class `Context` is singleton class. Class `Device` encapsulates logical device. One hardware device can be used in multiple logical device instances. Class `Graph` encapsulates frame graph. `Node` class is an abstract class that represents one render task. Class `Resource` is also an abstract class and can represent different types of Buffers and Images.

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.

The proposed architecture helps to increase the ease of use of Vulkan and reduces lines of code needed to use GPUs. Specifically, 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. Bringing support for the ray-tracing pipeline would be also a nice addition. Even rethinking some parts of the class hierarchy could even more reduce the complexity of use.

Acknowledgements

I would like to thank my supervisor Tomáš Milet for his help and ideas which helped to make vkEasy more user-friendly.

References

[1] Khronos Group. Khronos releases vulkan 1.0 specification. online. https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification.

[2] Hans-Kristian Arntzen. Granite. online. https://github.com/Themaister/Granite.

[3] Paweł. Pumex library. online. https://github.com/pumexx/pumex.

[4] Yuriy O'Donnell. Framegraph: Extensible rendering architecture in frostbite. online. https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in.

[5] Google Inc. Shaderc. online. https://github.com/google/shaderc.

[6] Khronos Group. Vulkan-hpp: C++ bindings for vulkan. online. https://github.com/KhronosGroup/Vulkan-Hpp.

[7] Khronos Group. vulkan_raii.hpp: a programming guide. online. https://github.com/KhronosGroup/Vulkan-Hpp/blob/master/vk_raii_ProgrammingGuide.md.

[8] AMD GPUOpen. Vulkan memory allocator. online. https://gpuopen.com/vulkan-memory-allocator/.

[9] AMD GPUOpen. Vulkan memory allocator. online. https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator.

[10] Sascha Willems. Vulkan example – minimal headless compute example. online. https://github.com/SaschaWillems/Vulkan/blob/master/examples/computeheadless/computeheadless.cpp.

[11] Alexander Overvoorde. Vulkan tutorial. online. https://github.com/Overv/VulkanTutorial.