Bournemouth University

National Centre for Computer Animation

MSc in Computer Animation and Visual Effects

evulkan

A Vulkan Library

Eimear Crotty

August 2020

Abstract

Vulkan is a low-level graphics API which aims to provide users with faster draw speeds by removing overhead from the driver. The user is expected to explicitly provide the details previously generated by the driver. The resulting extra code can be difficult to understand and taxing to write for beginners, leading to the need for a helper library.

Acknowledgements

Jon Macey
Mum, Dad, Rory, Aisling, Aoife, Ellie.
Neil.

Dedication

Contents

List of Figures

Chapter 1

Introduction

Vulkan (Khronos Group 2016b) is a cross-platform graphics and compute API. It aims to provide higher efficiency than other current cross-platform APIs, by using the full performance available in today's largely-multithreaded machines. Vulkan achieves this by allowing tasks to be generated and submitted to the GPU in parallel (multithreaded programming). In addition, the API itself is written at a lower-level than other graphics APIs, meaning that the developer is required to provide many of the details previously generated by the driver at run-time.

This project aims to alleviate this cost by providing a wrapper library for Vulkan, which allows a developer to use some of the more common features of Vulkan with much less effort than writing an application from scratch. This library is written in C++, using modern C++features, adheres to both the official C++Core Guidelines and Google C++Style Guide and is fully unit tested. The library is available for download from GitHub and can be built using CMake.

The library is specifically written with beginners and casual users of Vulkan in mind. The examples included in the repository provide a demonstration of how to use the library for different purposes, including drawing a triangle, loading an OBJ with a texture and using multiple passes to render simple objects with deferred shading. A non-goal is to create a library which is as fast as writing pure Vulkan, however the library must be reasonably fast.

Chapter 2

Previous Work

While Vulkan is a relatively new API for graphics and compute, many engines now support Vulkan, including CryEngine, Valve's Source, Unity and Unreal Engine. As a result, there are many libraries and utilities available online for Vulkan, each of which serves a different purpose.

2.1 V-EZ

AMD created the open-source V-EZ library (AMD 2018). Its main goal is to increase the adoption of Vulkan in the games industry by reducing the complexity of Vulkan. It is a lightweight C API wrapped around the basic Vulkan API. It is part of the GPU-Open initiative.

It still requires the user to have a good knowledge of Vulkan, making it difficult for beginners to adopt. For example, some rather complex components include semaphores, swapchain creation and lengthy enumerations such as

```
VK_BUFFER_USAGE_TRANSFER_DST_BIT
```

While it does remove some of the boilerplate, it is still relatively low level and, as a result, is not perfectly suited to beginners.

2.2 Anvil

The goal of Anvil is to reduce the amount of time taken to write Vulkan applications. It is ideal for rapidly prototyping Vulkan applications, but it still requires a large amount of writing. It is stated in the documentation itself that Anvil is not suitable for beginners.

> *Anvil is not the right choice for developers who do not have a reasonable understanding of how Vulkan works.* (AMD 2016)

2.3 GLOVE

GLOVE (Think Silicon 2016) provides an intermediate layer between an OpenGL ES application and Vulkan. It is easy to build and integrate new features and has a GL interface for developing applications.

GLOVE is useful for developing Vulkan applications for embedded devices, especially for developers who already have an understanding of GL applications. However, GLOVE is not useful for learning Vulkan as it only provides a GL interface.

2.4 MoltenVK

As Apple hardware lacks native Vulkan driver support, MoltenVK (Khronos Group 2016a) provides an interface over Apple's Metal graphics framework. This provides no speedup in terms of development time, it simply allows Vulkan to be developed and run on macOS. As a result, it does not provide any extra help for beginners to Vulkan.

2.5 Personal Inquiry

This library was developed using a previous project as a starting point (Crotty 2020). The base project can be found at http://github.com/eimearc/vulkan. It provided the boilerplate to run an instance of Vulkan and it saved days of typing 1000 lines of code to simply have a starting point. All class construction, library design and testing was implemented in this masters project.

Chapter 3

Technical Background

3.1 Useful Resources

As Vulkan is a relatively complex topic, many resources, both online and in-print, came in useful during this project and may help the reader with their Vulkan understanding.

- Vulkan Programming Guide (Sellers 2016)
- Sascha Willem's Vulkan examples (Willems 2015)
- Vulkan Tutorial (Overvoorde 2020)
- ARM Vulkan tutorial (ARM 2020)

3.2 Comparison with OpenGL

Vulkan is a low-overhead, cross-platform graphics and computing API. It was developed to allow higher performance and more balanced CPU/GPU usage in comparison to older APIs such as OpenGL.

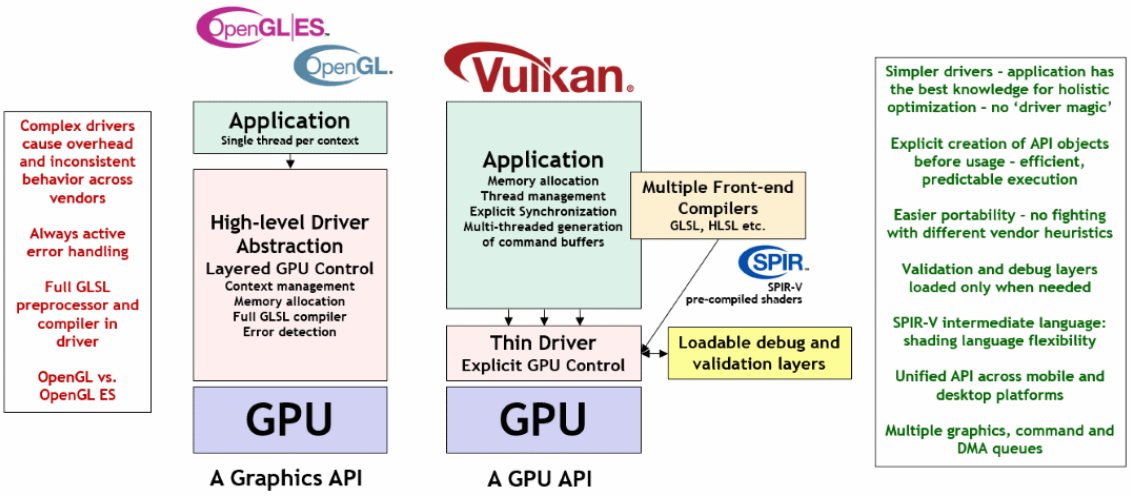

Figure 3.1: OpenGL compared to Vulkan (Khronos Group 2016c, p.1).

While OpenGL acts as a state machine, keeping track of application state, Vulkan requires the developer to keep track of such state. OpenGL requires operations to be submitted in sequence, while Vulkan takes full advantage of modern multicore machines and allows operations to be recorded and submitted in parallel.

OpenGL handles host-device synchronization and memory management in the driver, while Vulkan requires the developer to deal with this. The idea behind this is that the developer knows best how their data will be accessed and, as a result, the developer knows the optimal way to lay out data in memory. While this does result in a more explicit, low level API and longer development times, the advantage becomes apparent in the runtime speedup. There is much less overhead in the Vulkan driver, as the developer provides most of the required detail. Less driver work generally results in faster run times.

OpenGL provides a constant level of error checking. While this is useful during the development phase, once an application is rolled out to production, error checking slows down the application. Vulkan provides a way around this with validation layers that can be registered during development and removed afterwards, further speeding up an application.

OpenGL reads shader code in GLSL and compiles it at run time. This leads to a slower run time in the best case, or run time errors in the worst case when the GLSL is not properly formed. Vulkan requires the developer to compile the shader code to byte code (SPIR-V) ahead of time and to provide as such. This has the dual advantage of ensuring the shader code is correct and speeding up the run time.

The pattern is apparent; Vulkan requires more setup, state tracking and memory management from the developer. This removes much of the required word from the driver, resulting in faster draw speeds in comparison with older APIs such as OpenGL.

3.3 Vulkan Layers

More traditional APIs have a flat structure. Any calls made to the API are forwarded to the driver for more work. If a developer wants to extend the structure and capabilities of the API, they are required to either "hack" together a platform-specific implementation, or have their extension built directly by the API developers into the API and driver. This increases the bulkiness of the API, requiring all users to have this large API when they may only use the minimal number of features. This "all-or-nothing" approach decreases the speed of the application, which is quite important for smaller applications running on embedded systems.

Vulkan, in contrast, is a layered API, using a loader to create this layered architecture. This layered approach results in faster applications, as certain features which are needed in development, such as validation, can be unloaded when releasing an application.

3.3.1 Loader

The Vulkan loader is "the central arbiter in the Vulkan runtime" (TODO: quote). The application interfaces with the loader and it is the task of the loader to dispatch incoming

requests to the correct subsystem. The loader exposes all of the core Vulkan functions. When an application calls such functions, they are routed through the loader, instead of directly to the driver.

When creating an instance, certain extensions are required. Extensions are grouped into layers. These layers are specific to a system and platform and are registered in a well-known location on that machine in JSON files. These files contain the names of the extensions provided by the layer and where to find the actual library is on the system This means that whenever the Vulkan loader queries for a specific layer, the JSON file is read - the layer module itself does not need to be loaded.

For example, a layer JSON file may be found at

```
/usr/local/share/vulkan/explicit_layer.d/VkLayer_khronos_validation.json
```

Included in the file may be the following (edited for brevity):

```
"instance_extensions": [
...
  {
    "spec_version": "1",
    "name": "VK_EXT_debug_utils"
  },
...
],
...
"library_path": "../../../lib/libVkLayer_khronos_validation.dylib"
```

3.3.2 Dispatch Chains

A dispatch chain (figure 3.2) is the path along which execution flows. The application calls a function, for example `vkCreateInstance`. In the loader code, the layers and extensions are validated. The loader then passes execution along to the first layer, which also calls `vkCreateInstance`, then passing execution along to the following layer. The loader terminates with its own code, before passing off the execution to the ICD (installable client driver). All available drivers are now combined into one unified front.

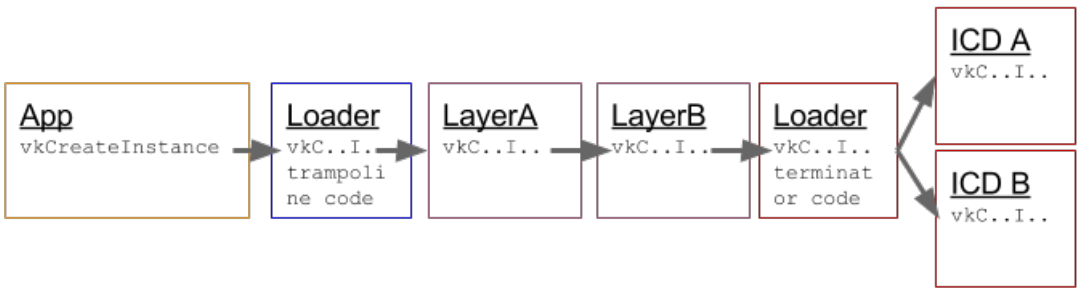

Figure 3.2: Vulkan dispatch chain (Karlsson 2018, p.1).

This execution style also creates a dispatch table (figure 3.3), where each layer in the queue calls `vkGetInstanceProcAddr` on the next layer. This long chain of function pointers means that each layer knows how to pass on control to the next layer in the chain.

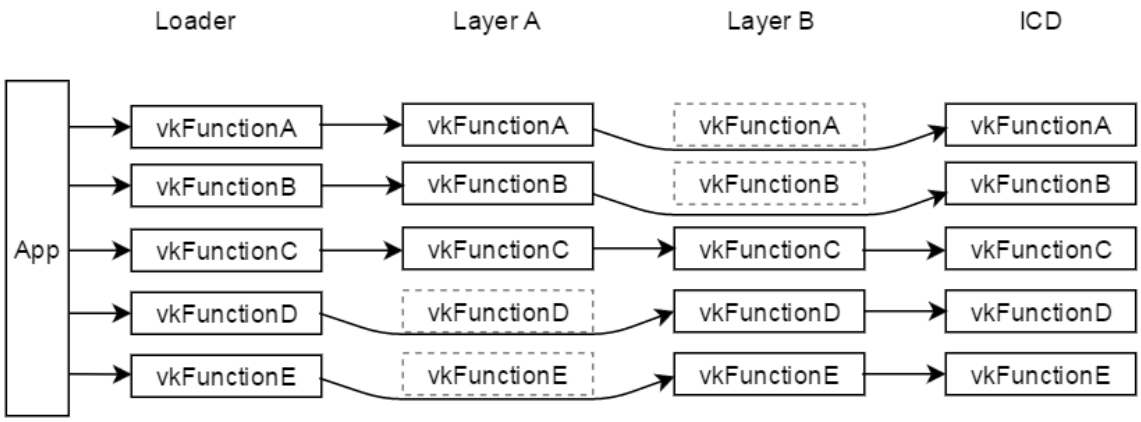

Figure 3.3: Vulkan dispatch table (Karlsson 2018, p.1).

3.4 Vulkan Components

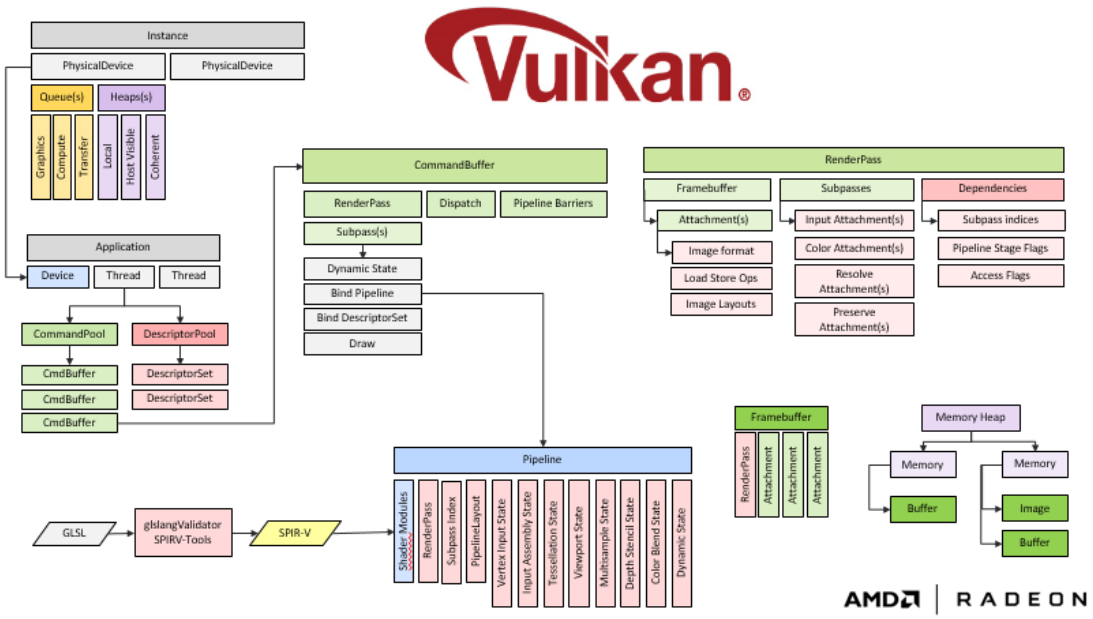

Figure 3.4: Vulkan API objects and their interactions (AMD 2018, p.1).

3.4.1 VkInstance

A Vulkan instance is the first Vulkan component a developer creates in their application. As Vulkan has no global state, all per-application state is contained within a Vulkan instance. By creating a VkInstance, the application loads the Vulkan commands and

initializes Vulkan. Within each instance are multiple physical devices.

After the Vulkan instance is created, devices and queues are the main way the application interacts with the Vulkan implementation.

3.4.2 VkPhysicalDevice

A physical device represents a single hardware device on the machine which has Vulkan capabilities, such as a GPU.

3.4.3 VkDevice

A logical device (simply a "device") is a software abstraction around a physical device. A physical device is queried for its capabilities and, based on required application criteria, a device is created from the suitable physical device. A device represents an instance of a physical device and contains its own state and resources. This is the Vulkan component that is most commonly interacted with and is used in constructing all subsequent components. An application is required to create a different device for each physical device it uses. Each device exposes a number of queues.

3.4.4 VkQueue

A queue is where a piece of work is submitted for completion by the GPU, for example a draw command. A queue is created in conjunction with a device and the application queries the device for a suitable queue. Queues are partitioned into a set of families, where each family supports one or more types of functionality. Examples of such functionality include graphics, presentation and compute. For most applications, graphics functionality is required to modify the incoming vertices and presentation support is required to display the resulting images on the screen.

Queue submission occurs when work is submitted to a queue using commands such as vkQueueSubmit. Such commands specify a set of underlying operations which are to be executed by the associated physical device.

Each queue works asynchronously to other queues, making it suitable for multi-threaded use.

3.4.5 VkDeviceMemory

Memory is explicitly managed by the application. There are two types of memory in Vulkan, host memory and device memory. Device local memory is physically connected to the device, while host visible memory is visible to the host. Each device exposes the types of memory available to the application.

When creating a buffer, the user must specify both how the buffer will be used and where this buffer will reside. Host-visible memory can be accessible by the CPU through the use of the vkMapMemory command, while the device-local is the most efficient for GPU access.

3.4.6 VkCommandBuffer

The application can control the device through the submission of command buffers. Prior to submission, the application records units of work into these command buffers. These command buffers may be constructed over multiple threads and may be reused multiple times. The command buffers are submitted to queues. Command buffers in separate queues may execute in parallel, while command buffers in a single queue execute in respect to queue submission order. Upon command buffer queue submission, control is returned to the application immediately.

There are two different types of command buffers, primary command buffers and secondary command buffers.

- A primary command buffer is submitted to a queue for execution. It holds references to an array of secondary command buffers.
- A secondary command buffer is not submitted to a queue for execution. Instead, work is recorded into it and a reference to the command buffer is attached to a primary command buffer, along with other secondary command buffers. This allows for multiple threads to construct multiple secondary command buffers in parallel, attach them to a primary command buffer and submit for execution.

All of this work can be recorded into the buffers ahead of draw time, resulting in faster draw speeds.

3.4.7 VkSwapchainKHR

3.5 Vulkan Object Model

Vulkan objects (VkInstance, VkDevice and so on) are represented by handles - an abstract reference to a piece of memory that is managed by Vulkan. Handles come in two types; dispatchable and non-dispatchable.

Dispatchable objects consist of a pointer to an opaque type. These objects internally hold a dispatch table. This table is used by other components of the system to determine what code to execute when the application makes calls to Vulkan. Examples of dispatchable objects include the VkInstance, VkPhysicalDevice, VkDevice, VkCommandBuffer and VkQueue. The first argument to any Vulkan function is a dispatchable object. This excludes VkInstance creation, as this is the first dispatchable Vulkan object created.

Non-dispatchable objects are 64-bit integer types which are implementation dependent. They either contain a reference to another object, or encode information about the object directly. Objects created on a specific device are private to that device and cannot be used on another device.

Chapter 4

The evulkan Library

4.1 How does it work?

The library exposes a number of components, each of which is a wrapper around one or more Vulkan objects.

4.1.1 evk::Device

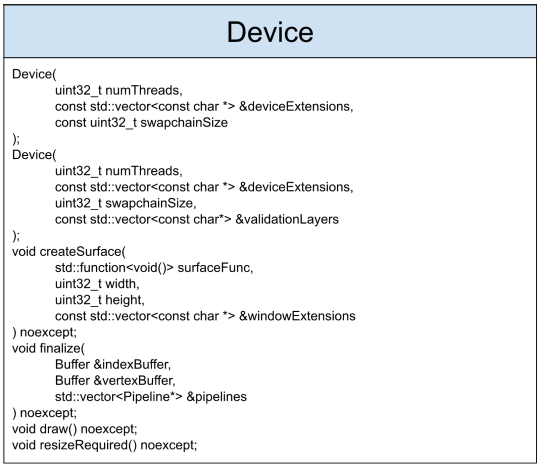

Figure 4.1: Device class diagram.

A device is the basic component of the library. It is the first Vulkan component that is constructed in the application. It encapsulates, among other things, a VkInstance, VkPhysicalDevice, VkDevice, VkQueues. A user can set up the device with or without validation layers. Leaving validation layers turned off results in a faster application suitable for production.

The Device object tracks state across the program. It is used in the creation of other evk Vulkan objects. The Device is responsible for creating a VkInstance, VkPhysicalDevice, VkDevice, VkSwapchainKHR, VkFramebuffer and all the required command buffers.

The Device is multithreaded, making use of Vulkan's multithreading capabilities. For some more intensive operations, such as recording draw operations into the command buffer, it splits the operation across multiple threads, each of which records its portion of the operation into a separate secondary command buffer.

```
vkCmdDrawIndexed(
  secondaryCommandBuffer, numIndices,
  1, indexOffset, 0, 0
);
```

These secondary command buffers are then executed from the primary command buffer.

```
vkCmdExecuteCommands(
  primaryCommandBuffer,
  secondaryCommandBuffers.size(),
  secondaryCommandBuffers.data()
);
```

4.1.2 evk::Texture

A Texture allows a user to load in a texture from a file. It transitions the image to a transfer-optimal format before copying the pixels from the image to device-local memory, allowing for faster GPU access. It then transitions the image to a layout readable by the shader, frees unneeded resources and finally creates a VkImageView and VkSampler.

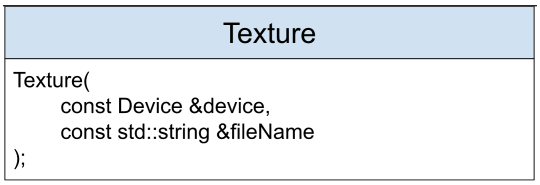

Figure 4.2: Texture class diagram.

A VkImageView is required by the Shaders to access images. They represent contiguous ranges of subresources and any metadata required to operate on them. A VkSampler is a handle to an image sampler and is used by the Shader to read image data from Textures and apply filters.

4.1.3 evk::Attachment

An Attachment represents a resource that can be read from, or written to by a Shader. Each Attachment has a binding location and a Type, one of FRAMEBUFFER, COLOR or DEPTH. Both a FRAMEBUFFER and DEPTH Attachment are required for any program as the shaders must write to the depth buffer and to the screen. For more advanced usages (see the multipass example) a COLOR Attachment is useful.

An Attachment consists of a VkAttachmentDescription, which describes the properties of the Attachment, and multiple VkAttachmentReferences, which allow other stages to refer to these Attachments. For DEPTH and COLOR Attachments, a VkImage is created, while this is already created in the Device creation stage for the FRAMEBUFFER Attachment.

An Attachment also contains a VkClearValue. This value specifies how an Attachment should be cleared when it is reused.

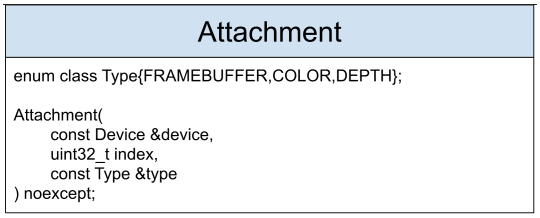

Figure 4.3: Attachment class diagram.

Attachments are used in Subpasses as either an input attachment, a colour attachment or a depth attachment. A colour or depth attachment is written by the shader, while an input attachment is read into a shader, making it useful for multipass rendering.

4.1.4 evk::Buffer

A Buffer encapsulates VkBuffer-related structs and methods. It handles the creation and copying of VkBuffers, which are linear arrays of data, along with copying user data into them and updating them after creation. There are two types of Buffers; StaticBuffer and DynamicBuffer.

A StaticBuffer is used when the contents of the Buffer will not be updated after they have been set. This allows for some optimization to take place. The Buffer contents are copied to device-local memory to allow faster GPU-access at draw time.

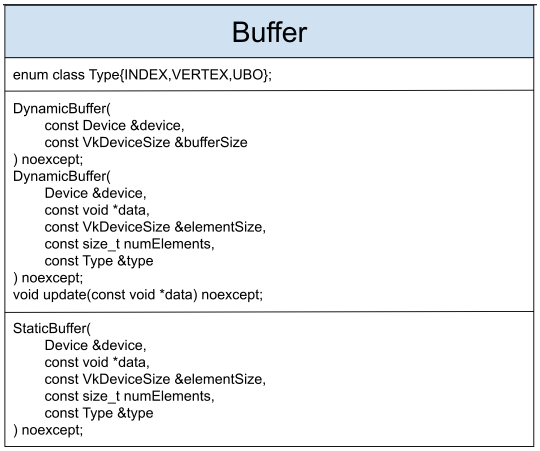

Figure 4.4: Buffer class diagram.

A DynamicBuffer is used when the Buffer contents will be updated at draw time. This skips the relatively expensive step of copying the Buffer data to device-local memory, and leaves them in host-visible memory for faster update speeds. A DynamicBuffer can be updated using the update() command.

4.1.5 evk::Descriptor

A Descriptor is used to describe any resources that will be accessed by the Shader, such as a texture sampler, a uniform buffer or an input attachment. A user constructs a Descriptor and then adds the necessary resources using class methods. Descriptors are bound in advance of draw commands.

When a user calls one of these functions, for example addTextureSampler(), the following happens:

- A VkDescriptorSetLayoutBinding is constructed. This describes the shader stage (e.g. a fragment shader) at which the resource will be accessed, in addition to the type of the resource.

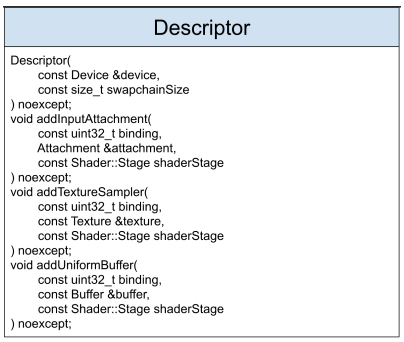

Figure 4.5: Descriptor class diagram.

- A VkWriteDescriptorSet is constructed, specifying the binding of the resource, along with the type and any other information required for a descriptor set write operation. In this example, extra information includes the texture sampler handle along with the image view handle.
- Later on in the program, during pipeline creation, the finalize() method is called on the Descriptor. This creates the VkDescriptorPool, from which the VkDescriptorSets are allocated. VkDescriptorSets are sets of resources which are bound into the VkPipeline as a group. Finally, the VkWriteDescriptorSets are updated using vkUpdateDescriptorSets() command, binding the resources into the descriptor sets.

4.1.6 evk::Subpass

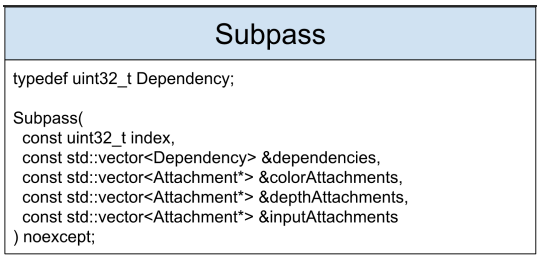

Figure 4.6: Subpass class diagram.

A Subpass describes a pass, or a phase of operation, within a Renderpass. It can depend on the completion of other Subpasses. It has a set of Attachments, which can be either colour, depth or input Attachments (as described above) along with their references. Contained within the Subpass is a VkSubpassDescription, a struct describing the Subpass. The index of the Subpass indicates when the Subpass will be executed in order.

4.1.7 evk::Renderpass

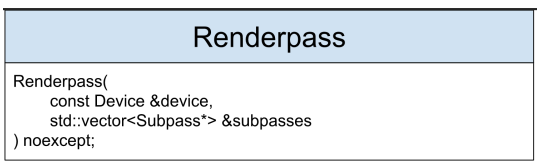

Figure 4.7: Renderpass class diagram.

A Renderpass object is a collection of Subpasses, along with their dependencies and attachments. A VkRenderPass is created within the Renderpass by passing in the set of VkSubpassDescriptions, VkAttachmentDescriptions and the VkSubpassDependencies between subpasses.

4.1.8 evk::Pipeline

A Pipeline takes all the previous information and generates a VkPipeline - the sequence of operations that take the input vertices and produces the output, to the screen or otherwise. We will only cover graphics pipeline creation here. A Pipeline is associated with exactly one Subpass.

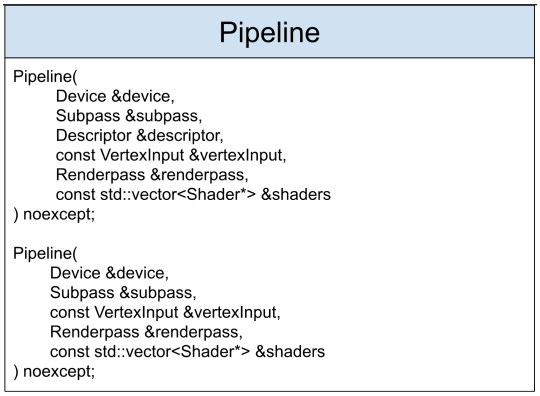

Figure 4.8: Pipeline class diagram.

A Pipeline setup call takes the user-provided Shaders, Renderpass and Subpass index, and combines it with automatically generated fixed-function information (input assembly, viewport state, rasterizer, multisampling, colour blending) to create the VkPipeline.

4.1.9 evk::Shader

A Shader represents a shader program, which contains operations that execute for each vertex or fragment. A shader contains a VkShaderModule, which contains the shader code, and a VkPipelineShaderStageCreateInfo struct, which is used during Pipeline construction.

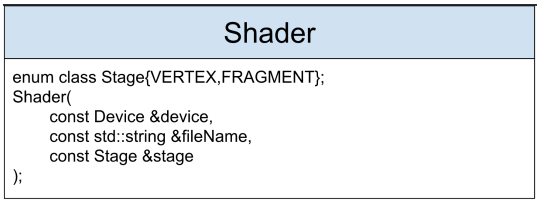

Figure 4.9: Shader class diagram.

There are two supported Shader stages, vertex and fragment. A vertex Shader operates on each vertex and any associated vertex input attributes, specified using the VertexInput struct. A fragment Shader operates on every fragment. Both a vertex Shader and fragment Shader are required for the evulkan library.

A user can load in SPIR-V code from a file of their choice. SPIR-V is easily generated from GLSL using the glslc command, which is available for download from https://github.com/google/shaderc, or with the Vulkan SDK from LunarG.

4.1.10 evk::VertexInput

A VertexInput object is a wrapper around a VkVertexInputAttributeDescription and a VkVertexInputBindingDescription. It is used for describing vertex attributes that will be bound to the shader, such as colour and normal.

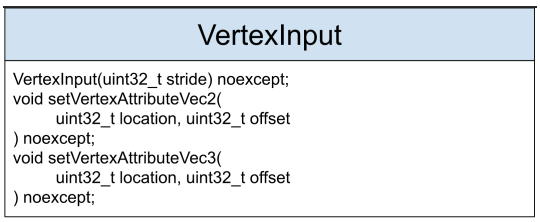

Figure 4.10: VertexInput class diagram.

4.1.11 Window System Integration

Vulkan is a platform-agnostic API. It is up to the application to specify the extensions required to interface with the window system. The way in which the VkSurfaceKHR object is created also needs to be handled by the application.

To keep the evk library platform-agnostic, the user must register two things: a function which will be called to create the surface and the set of instance extensions required for Vulkan to correctly interface with the surface object. For GLFW, this may result in something like the following

```
auto glfwExtensions = glfwGetRequiredInstanceExtensions(
                        &glfwExtensionCount
                      );

std::vector<const char*> surfaceExtensions(
  glfwExtensions, glfwExtensions + glfwExtensionCount
);

auto surfaceFunc = [&](){
  glfwCreateWindowSurface(
    device.instance(), window, nullptr, &device.surface()
  );
};

device.createSurface(surfaceFunc,800,600,surfaceExtensions);
```

For SDL, the following would suffice

```
SDL_Vulkan_GetInstanceExtensions(
  window, &sdlExtensionCount, surfaceExtensions
);

auto surfaceFunc = [&](){
  SDL_Vulkan_CreateSurface(
    window, device.instance(), &device.surface()
  );
};

device.createSurface(
  surfaceFunc,800,600,surfaceExtensions
);
```

When a window is resized, the evk library automatically recreates the required objects (swapchain, framebuffer, input attachments). The window resize triggers a `VK_ERROR_OUT_OF_DATE_KHR` or `VK_SUBOPTIMAL_KHR` result from either the vkAcquireNextImageKHR() function or the vkQueuePresentKHR() function within the draw() method. However, there are times when the platform or driver does not correctly trigger a resize event. As such, it is recommended that the user register a callback function as follows

```
glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
```

4.1.12 Error Handling

Error handling is a contentious topic. Different people have different (strong) opinions on which approach to take. Unlike other programming languages, C++ does not have a unified approach to error handling, leading to diverging dialects of the language.

There are two main types of errors: user errors and programming errors. The former are the fault of the person using the library, while the latter are the fault of the library developer. User errors should be reported to the user and the program should continue execution. Programming errors, on the other hand, should halt the program and provide as low level information to help the programmer debug the issue. Such errors would then be fixed before release (ideally).

Error Return Codes

A common way to handle errors is to return a simple type (bool or int) which the user can then test to determine if a method succeeded.

```
bool create_file(const std::string &name);
```

There are many problems with this approach.

- The return channel is blocked with the error code, meaning the function can not return anything else. A solution would be to use out parameters.
- A bool has low information bandwidth - it tells us if an operation failed, but it does not tell us how or why it failed. Of course, the bool type could be extended to use an unsigned int, but then it is up to the user to compare the int against enums to determine the correct path to take. This can be messy. Messy code is brittle code.
- The user can choose to ignore the value and continue on, leading to reduced code safety. A nodiscard keyword could be added here, but only if the error is being returned, not used as an out parameter as specified above.
- The code that is written to handle the error can be totally unrelated to the code that detects the error. For example, a function 30 levels deep into the call stack might generate an error that can only be handled at the top level in main.

Exceptions

Exceptions are the official way to handle errors. It involves throwing an error in the body of code and catching it and dealing with it somewhere else.

```
void create_file(const std::string &name);

try
{
  create_file("my_file.txt");
} catch(std::exception &e)
{
  // Handle error.
}
```

There are some problems with using exceptions:

- Exceptions augment the function stack frame to include information about how to handle an exception by unwinding the stack.
- Like error codes, there is a disconnect between the code that throws the exception and the code that catches the exception. There may be functions embedded within the two that become involved in the stack unwinding process. Such disconnect makes writing robust programs difficult. It means that every part of the code needs to be able to handle a less-than-graceful exit.
- Again like error codes, the user might have to figure out how to handle the plethora of possible exceptions generated by the code.

> *By using exceptions instead of just crashing we are creating a more complicated API (the API now includes all the different exceptions that the different functions might call) and significantly increasing the mental burden on the caller for very little gain.* (Frykholm 2012)

C++20 has a proposal in the works for contract-based programming which eliminates many of these problems, but as this library is explicitly being developed for C++14, it's out of scope.

Some developers advocate against the use of exceptions completely, disabling their use at the compiler level. Instead, they use assertions.

Assertions

An assertion checks an expression passed to it. If it evaluates to true, nothing happens - if it evaluates to false, the program is halted and some information is displayed to the developer. Assertions can be extended to pass helpful messages to the developer.

Assertions ensure that invariants hold. By placing assertions throughout a code base, potentially difficult to find bugs manifest themselves before they become a problem. The

error cannot be ignored as the program has crashed.

Assertions are defined using macros and are used during development. Prior to release, they can be removed to gain some performance. By toggling a variable, assertions can be easily switched on and off.

Assertions are the industry standard for video games, and, as such, they are used in the evulkan library. The Game Engine Architecture Book (Gregory 2014) has an excellent section discussing the advantages and disadvantages of different kinds of error handling.

4.1.13 Architecture

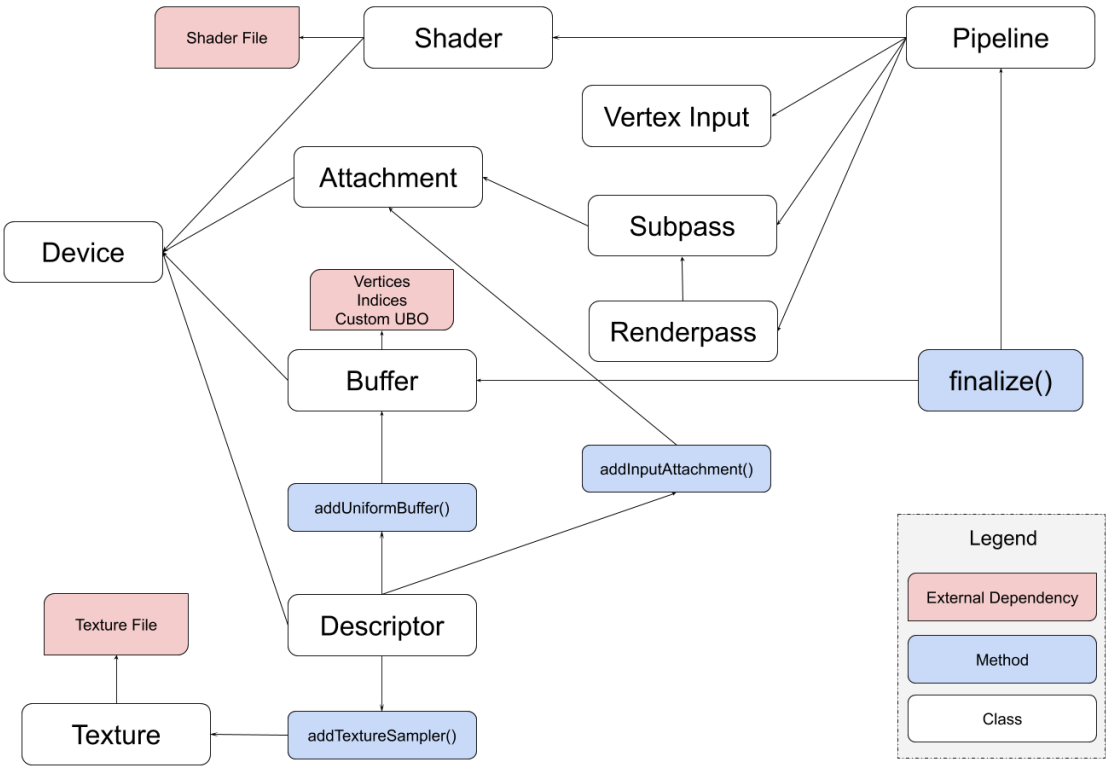

Figure 4.11: evulkan architecture.

4.2 Installation and Use

As the library uses CMake, it is relatively easy to install. Follow these instruction in figure 4.12 from the root of the directory.

```
$ mkdir build
$ cd build
$ cmake ..
$ make install
```

Figure 4.12: instructions for installing evulkan.

4.3 Known Problems

4.3.1 Colo(u)r

Spelling is always an issue in writing code, given the wide-reaching nature of its collaboration. In order to match the spelling of the (mostly American led) Vulkan, American spelling of some words was used, including "color". However, given the audience, within this paper the British spelling is used.

4.3.2 Pointers

In order to allow this library to handle the binding of any type of vertex data, `void *` pointers had to be used. Combined with the need to have multithreaded naked pointer arithmetic (`void *dst`).

4.4 Why Should You Use This Library?

The idea surfaced when a lecturer was questioning whether or not to teach an introduction to Vulkan next year. A major problem with starting Vulkan is the sheer amount of code you need to write to get it up and running. This library removes that step, changing the process of learning Vulkan from a typing exercise into a graphics lesson. It allows the user to begin to understand basic Vulkan concepts without having to wrangle more complex topics.

This library contains well-documented C++ code that adheres to best principles (C++ Core Guidelines) and style (Google Style Guide). It uses namespacing to prevent name clashes and uses standard header guards instead of the non-standard `#pragma once` directive.The library is unit tested using Google GTest. It uses C++14, adhering to the C++ requirements set by the VFX Reference Platform for CY2020.

Chapter 5

Conclusion

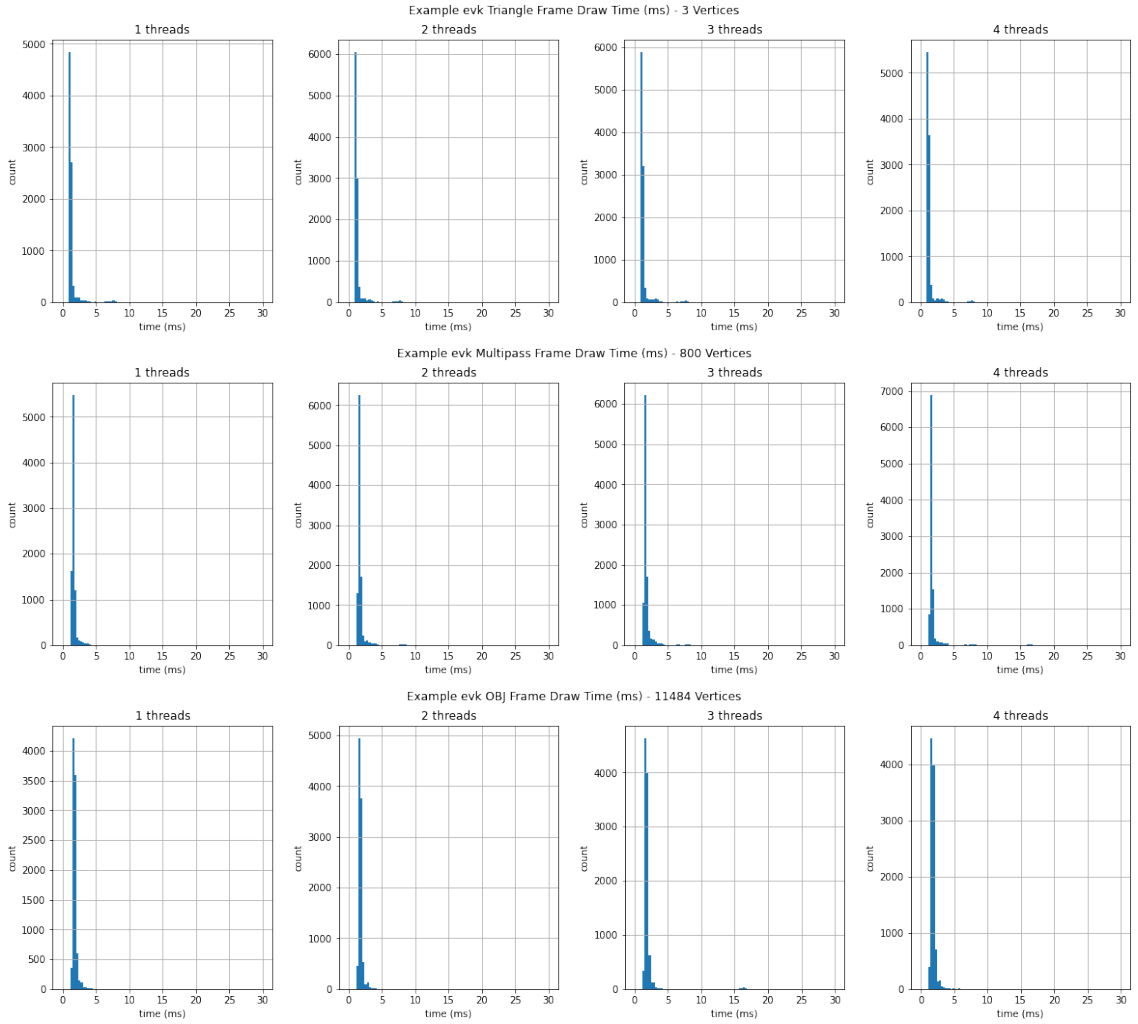

Figure 5.1: Draw time for different examples over multiple threads.

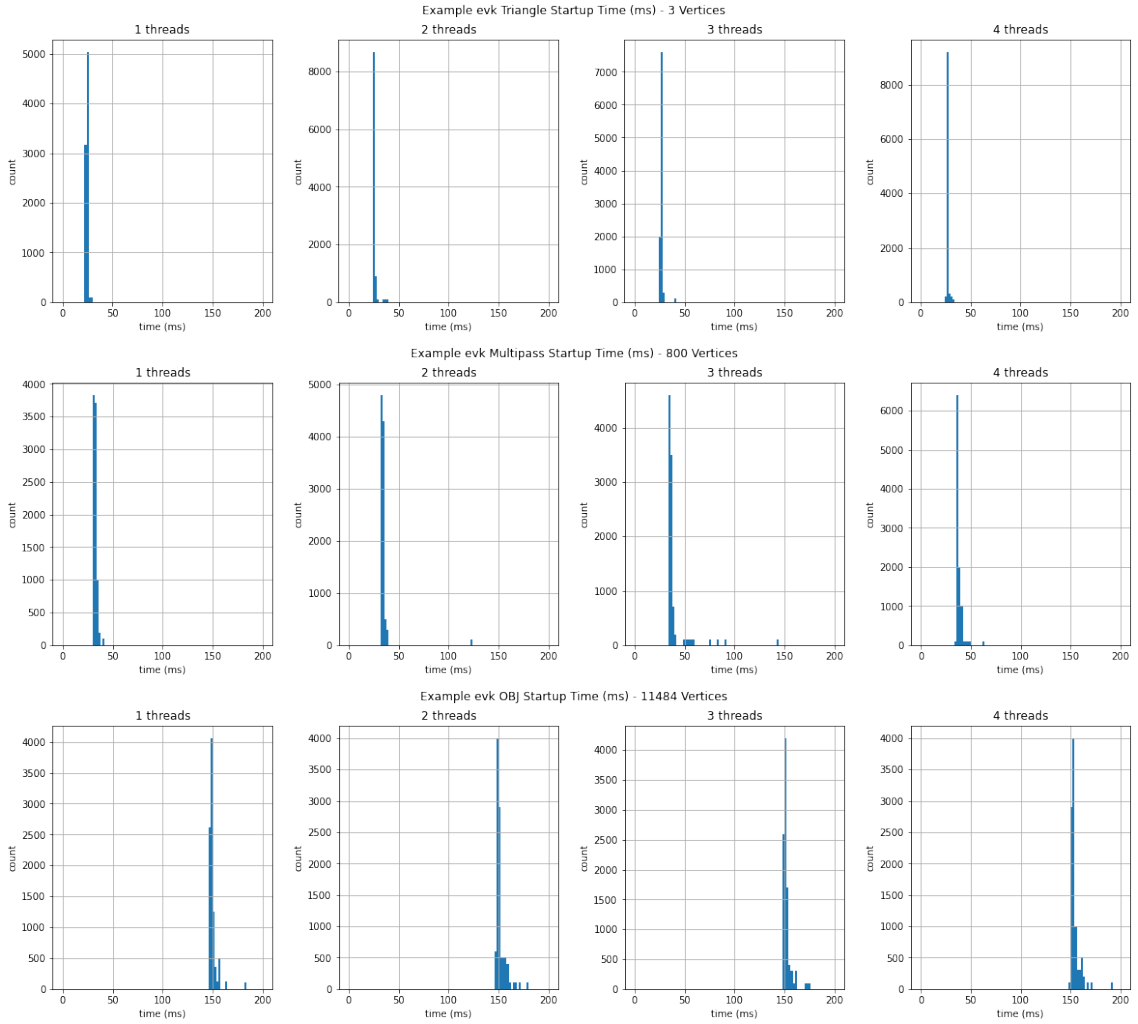

Figure 5.2: Setup time for different examples over multiple threads.

Bibliography

AMD, 2016. *Anvil*. 1.0 [computer program]. USA: AMD.

AMD, 2018. *V-EZ*. 1.1 [computer program]. USA: AMD.

ARM, , 2020. *Vulkan Tutorial* [online]. USA. Available from: `https://arm-software.github.io/vulkan-sdk/vulkan_intro.html`. [Accessed 3 July 2020].

Crotty, E., 2020. *Programs for benchmarking multithreaded Vulkan and OpenGL applications* [online]. Bournemouth: GitHub. Available from: `https://github.com/eimearc/vulkan`. [Accessed 10 August 2020].

Frykholm, N., 2012. *In-depth: Sensible error handling, part 1* [online]. USA: Gamasutra. Available from: `https://www.gamasutra.com/view/news/129438/Indepth_Sensible_error_handling_part_1.php`. [Accessed 14 August 2020].

Gregory, J., 2014. *Game Engine Architecture, Second Edition*. 2nd edition. USA: A. K. Peters, Ltd.

Karlsson, B., 2018. *Brief guide to Vulkan layers* [online]. USA: Karlsson,Baldur. Available from: `https://renderdoc.org/vulkan-layer-guide.html`. [Accessed 9 August 2020].

Khronos Group, 2016a. *MoltenVK*. 1.0.38 [computer program]. USA: Khronos Group.

Khronos Group, 2016b. *Vulkan*. 1.2 [computer program]. USA: Khronos Group.

Khronos Group, , 2016c. *Vulkan Guide* [online]. USA. Available from: `https://github.com/KhronosGroup/Vulkan-Guide/blob/master/chapters/what_is_vulkan.md`. [Accessed 20 July 2020].

Overvoorde, A., 2020. *Vulkan Tutorial* [online]. USA. Available from: `https://vulkan-tutorial.com/`. [Accessed 1 August 2020].

Sellers, G., 2016. *Vulkan Programming Guide*. 1st edition. USA: Addison-Wesley Professional.

Think Silicon, 2016. *GLOVE*. 1.0 [computer program]. USA: Think Silicon.

Willems, S., 2015. *Examples and demos for the new Vulkan API* [online]. USA: GitHub. Available from: `https://github.com/SaschaWillems/Vulkan`. [Accessed 13 August 2020].

Appendices