Bournemouth University

National Centre for Computer Animation

MSc in Computer Animation and Visual Effects

evulkan

A Vulkan Graphics Library

Eimear Crotty

github.com/eimearc/masters

August 2020

Abstract

Vulkan is a low-level graphics and compute 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

Thanks to Jon Macey for his help on the C++ side of things and for loaning me his Vulkan book.

Many thanks to my uncle Neil for giving me a second home during the year. Those curries and trips to the Glasshouse kept me going through multiple challenging assignments. I'm sorry they were cut short.

Finally, thanks to my mum Kerry, dad Owen, siblings Rory, Aisling, Aoife and Ellie for their support during the last year.

Contents

List of Figures

Listings

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 (Stroustrup and Sutter 2020) and Google C++ Style Guide (Google 2020) 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 (Crytek 2002), Source (Valve 2004), Unity (Unity Technologies 2005) and Unreal Engine (Epic Games 1998). 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 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 (AMD 2016) 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.

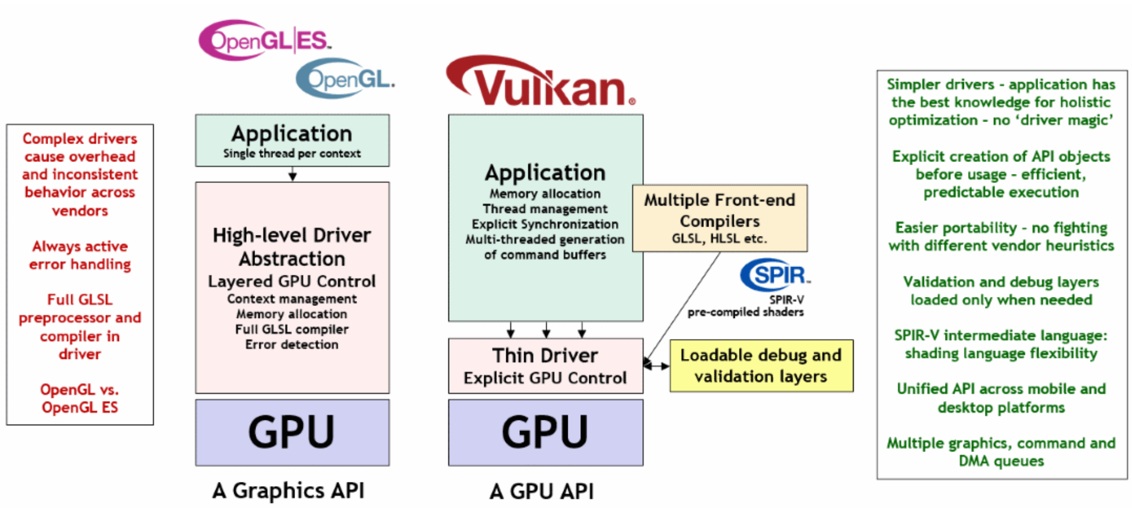

Figure 3.1: OpenGL compared to Vulkan (Khronos Group 2016c, p.1).

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. 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 work 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" (Karlsson 2018). 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 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 layers are now combined into one unified front.

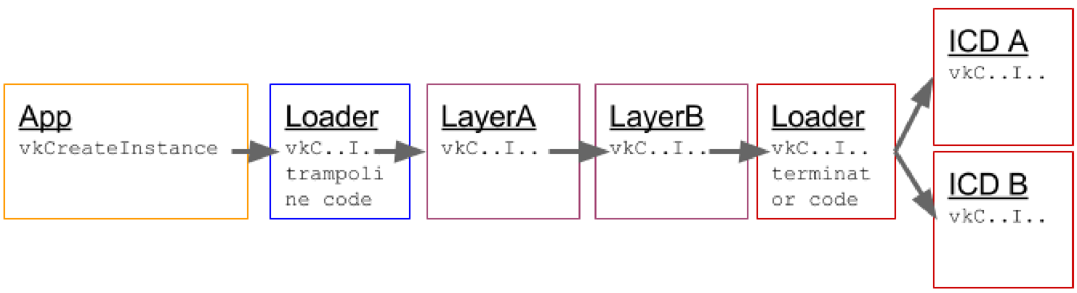

Figure 3.2: Vulkan dispatch chain (Karlsson 2018, p.1).

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.

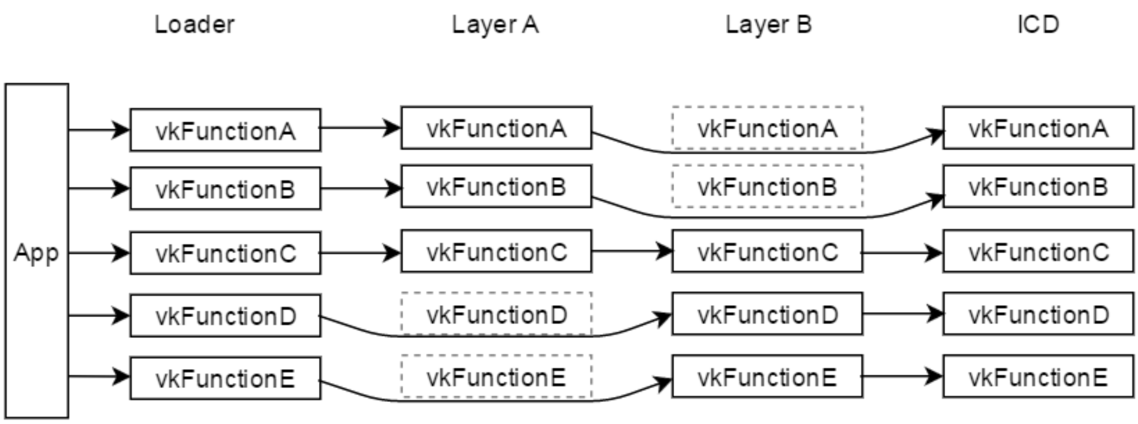

Figure 3.3: Vulkan dispatch table (Karlsson 2018, p.1).

3.4 Vulkan Components

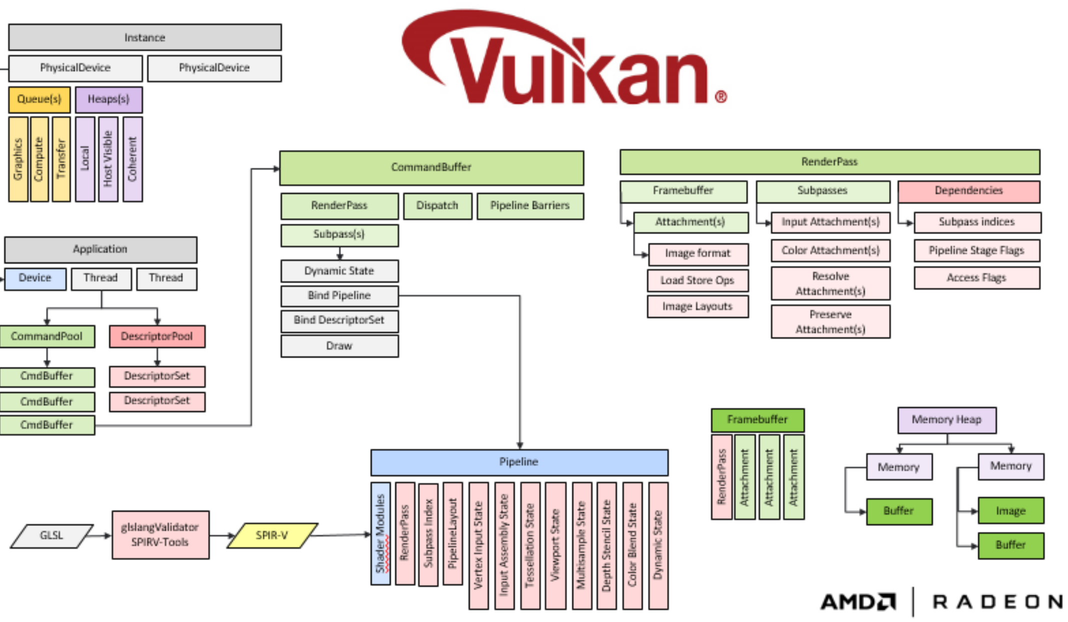

Figure 3.4: Vulkan API objects and their interactions (AMD 2018, p.1).

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 (or 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.

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 is accessible by the CPU through the use of the `vkMapMemory` command, while device-local memory is the most efficient for GPU access.

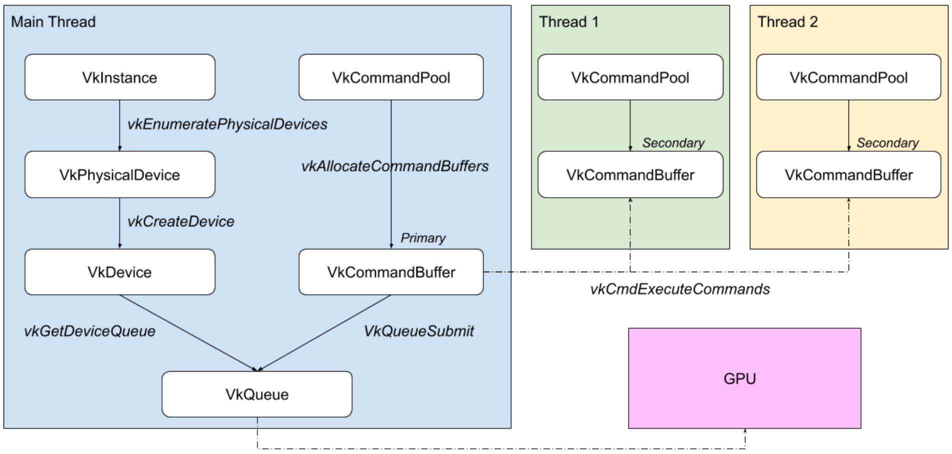

Figure 3.5: Multithreaded command buffer usage (Crotty 2020, p.6).

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 (figure 3.5), primary command buffers and secondary command buffers.

- A primary command buffer is submitted to a queue for execution. It may hold 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

The swapchain is an abstraction around a series of images that are presented to the screen. One image is presented at a time, but at the same time the application may be writing to another image (double buffering). The minimum and maximum number of images in a swapchain is implementation-dependent. It is possible to query Vulkan (using the `vulkaninfo` command) to determine the correct number of images to request in a swapchain. For many graphics cards, this number is between 2 and 3 inclusive.

The `KHR` suffix indicates that this is not a core Vulkan object, but is provided as part of an extension. This is because Vulkan is platform-agnostic and so cannot tie its presentation support to a single window system or presentation engine.

The application acquires an image from the swapchain using the `vkAcquireNextImageKHR` command and releases it for presentation using the `vkQueuePresentKHR` command.

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

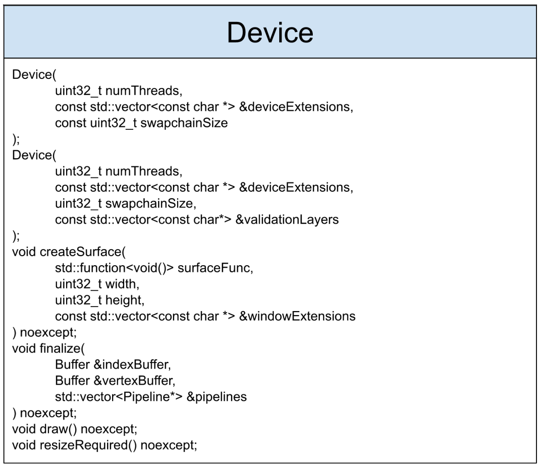

Figure 4.1: Device class diagram.

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 and VkQueues. A user can set up the device with or without validation layers. Leaving validation layers turned off results in a faster application ideal for production.

Listing 4.1: Device usage.

```
Device device = Device(
    numThreads, deviceExtensions, swapchainSize,
    validationLayers
);
```

The Device object tracks state across the program. It is used in the creation of other

Vulkan objects. The Device is responsible for creating a VkInstance, VkPhysicalDevice, VkDevice, VkSwapchainKHR, VkFramebuffer and all the required command buffers and synchronization objects.

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.

Listing 4.2: Secondary command buffer usage.

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

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

Listing 4.3: Primary command buffer usage.

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

4.1.2 evk::Texture

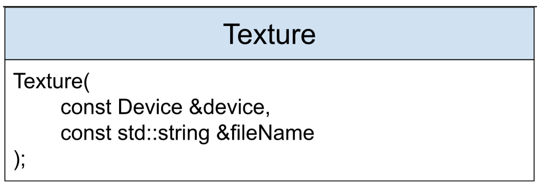

Figure 4.2: Texture class diagram.

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.

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.

Listing 4.4: Texture usage.

```
Texture texture(device, "viking_room.png");
```

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.

Listing 4.5: Attachment usage.

```
Attachment framebufferAttachment(
  device, 0, Attachment::Type::FRAMEBUFFER
);
Attachment depthAttachment(
  device, 1, Attachment::Type::DEPTH
);
```

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 before it is used.

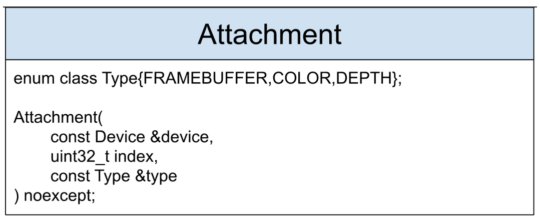

Figure 4.3: Attachment class diagram.

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 suitable for a Buffer that will not be updated after it has been created. 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. A StaticBuffer uses multiple threads to copy the buffer data across to the device-local memory. The user

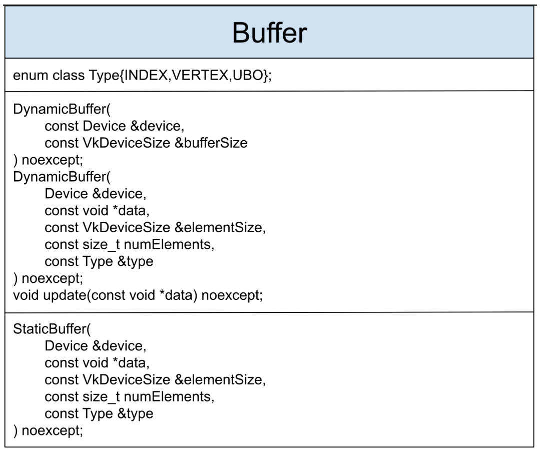

Figure 4.4: Buffer class diagram.

is able to specify the desired number of threads to use when copying across the data, allowing for speed ups in the application set up time. The number of threads is specified at Device creation time.

Listing 4.6: StaticBuffer usage.

```
StaticBuffer indexBuffer(
  device, in.data(), sizeof(in[0]),
  in.size(), Buffer::Type::INDEX
);
```

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

Listing 4.7: DynamicBuffer usage.

```
DynamicBuffer ubo(
    device, &uboUpdate, sizeof(uboUpdate),
    1, Buffer::Type::UBO
);
...
ubo.update(&uboUpdate);
```

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.

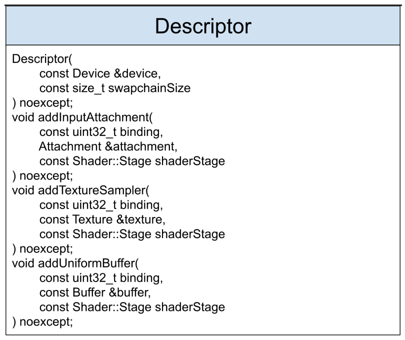

Figure 4.5: Descriptor class diagram.

Listing 4.8: Descriptor usage.

```
Descriptor descriptor(device, swapchainSize);
descriptor.addTextureSampler(
  1, texture, Shader::Stage::FRAGMENT
);
```

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.
- 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

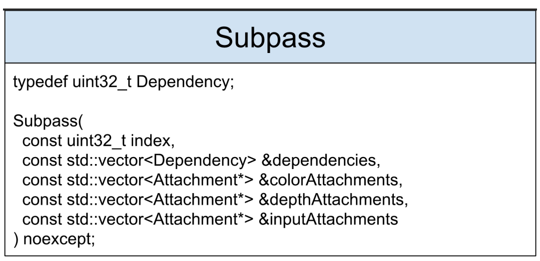

Figure 4.6: Subpass class diagram.

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 relation to the other Subpasses.

Listing 4.9: Subpass usage.

```
Subpass subpass(
    0, dependencies, colorAttachments,
    depthAttachments, inputAttachments
);
```

4.1.7 evk::Renderpass

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.

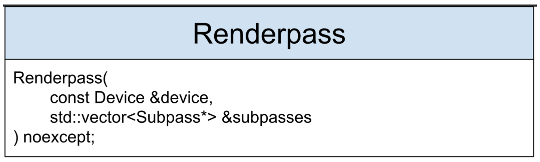

Figure 4.7: Renderpass class diagram.

Listing 4.10: Renderpass usage.

```
Renderpass renderpass(device,subpasses);
```

4.1.8 evk::Pipeline

A Pipeline takes all the previous information and generates a VkPipeline - the sequence of operations that takes 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.

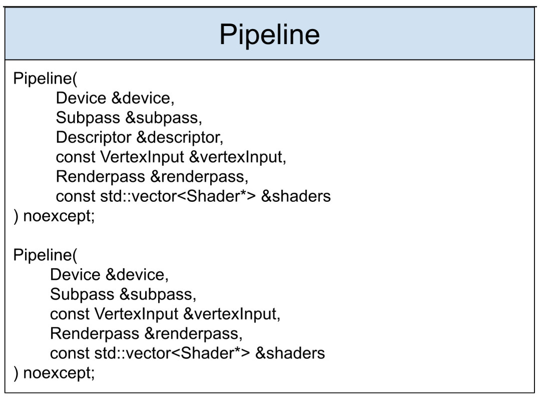

Figure 4.8: Pipeline class diagram.

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

Listing 4.11: Pipeline usage.

```
Pipeline pipeline(
  device, subpass, descriptor, vertexInput,
  renderpass, shaders
);
```

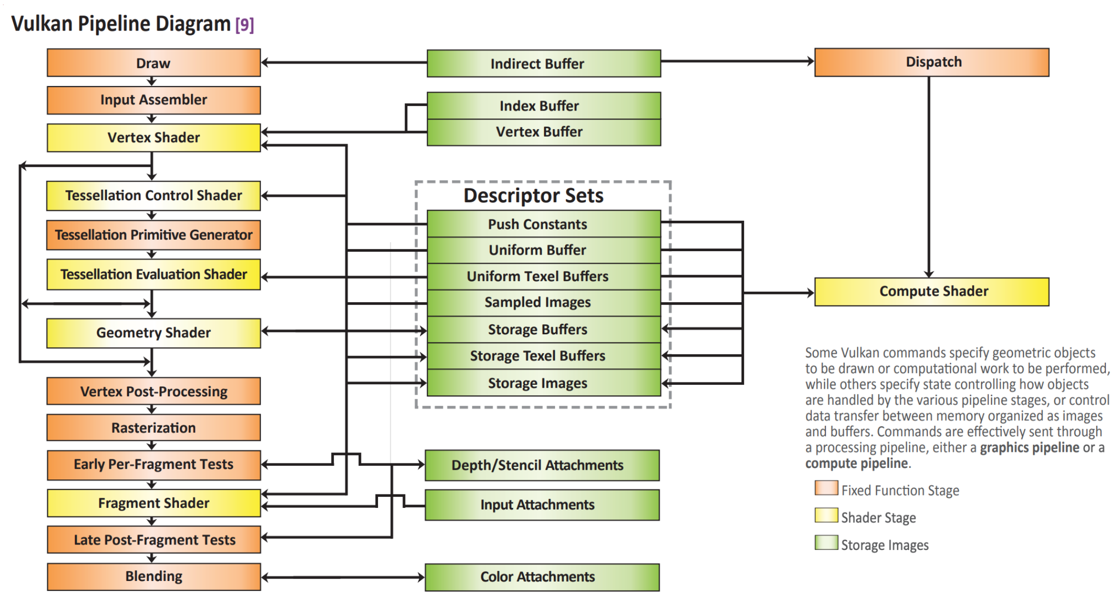

Figure 4.9: Vulkan pipeline (Khronos Group 2016d, p.5).

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.

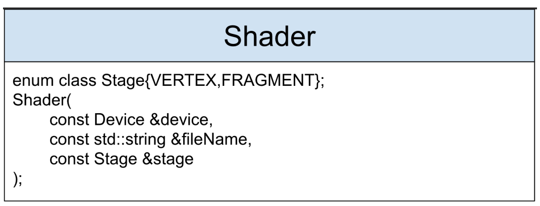

Figure 4.10: Shader class diagram.

There are two supported Shader stages in the evulkan library, 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.

Listing 4.12: Shader usage.

```
Shader vertexShader(
  device, "shader_vert.spv", Shader::Stage::VERTEX
);
Shader fragmentShader(
  device, "shader_frag.spv", Shader::Stage::FRAGMENT
);
```

The specified shader code must be in the SPIR-V format. 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

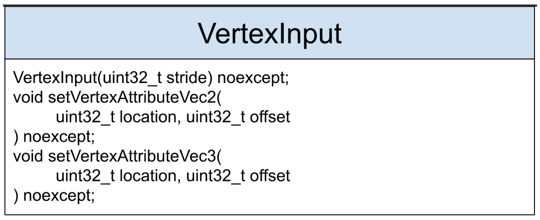

Figure 4.11: VertexInput class diagram.

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.

Listing 4.13: VertexInput usage.

```
VertexInput vertexInput(sizeof(Vertex));
vertexInput.setVertexAttributeVec3(
  0,offsetof(Vertex,pos)
);
vertexInput.setVertexAttributeVec3(
  1,offsetof(Vertex,color)
);
```

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 evulkan 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

Listing 4.14: GLFW window surface usage.

```
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

Listing 4.15: SDL window surface usage.

```
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

Listing 4.16: Window resize callback usage.

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

4.1.12 Error Handling

Error handling is a contentious topic. Different people have varying (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 low level information to help the programmer debug the issue. Such errors would then be fixed before release (ideally). The remainder of this section will discuss programming errors.

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, and could not used if an out parameter is used 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 (Dos Reis et al. 2018) 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.

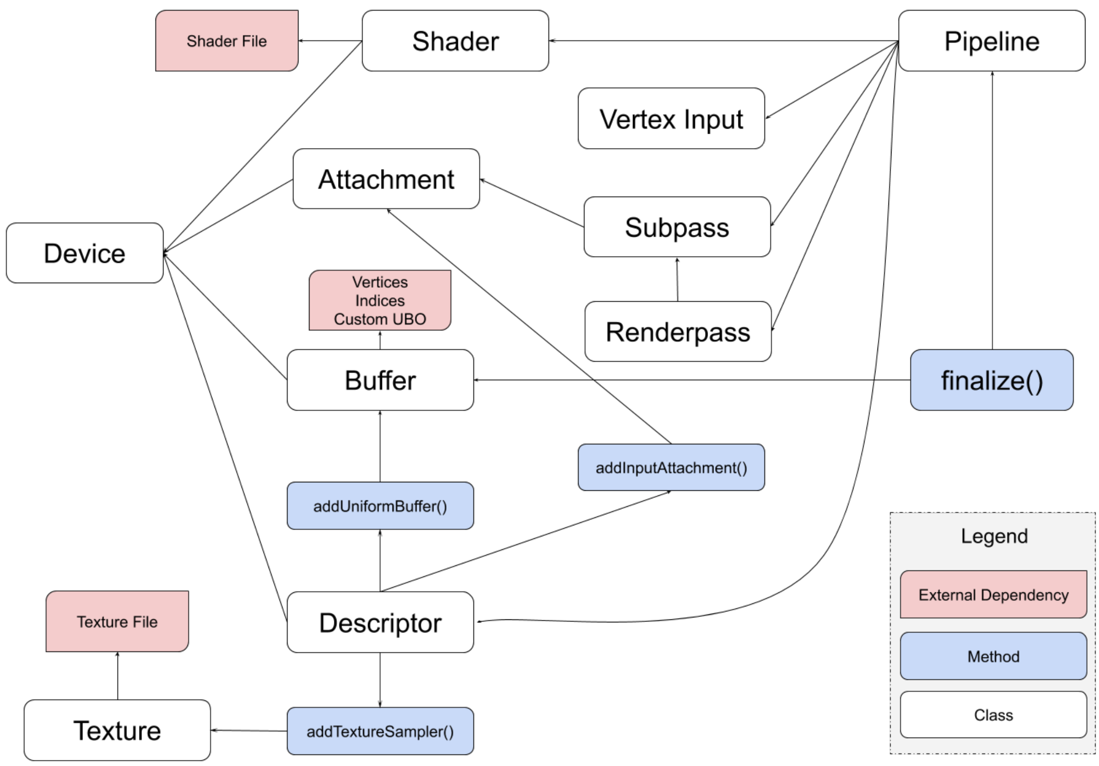

Figure 4.12: evulkan architecture.

4.2 Installation and Use

As the library uses CMake, it is relatively easy to install. Follow the instructions in figure 4.13 from the root of the directory.

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

Figure 4.13: Instructions for installing evulkan.

The library and header file will be installed in `/usr/local/` by default. Example files can be run from the generated `build/examples/` directory.

Listing 4.17: Example program usage.

```
 triangle$ ./triangle
multipass$ ./multipass --num_threads 3
      obj$ ./obj
```

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. In addition, there is also the need to have naked pointer arithmetic (`void *dst`) for offsetting indices when multithreading. This is not a robust solution and should be improved in future revisions.

4.4 Why Should You Use This Library?

The idea for the evulkan library 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. Many of the functions are properly marked as `const noexcept`. 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.

The source code is available on GitHub (https://github.com/eimearc/masters) and is easily accessible. The CMake file allows for multiple configurations. For example a user can simply build and install the library, or they can build and run the examples and the tests. The library has been statically analyzed using Cppcheck to flag any undefined behaviour and dangerous coding constructs. No issues were found.

Note not all possible Vulkan features are available with this library. As with any library, the goal is to make the most common solutions available to the user. If a developer wants all possible features, they should simply use the Vulkan API.

4.4.1 Things You Don't Need to Learn About

While this library is intended to help beginners learn some key Vulkan concepts, and how they interact, many more complex implementation details are purposely hidden from the user, including command buffers creation and use, synchronization of device and host, and swapchain creation.

While this library is not the way to learn everything about Vulkan, it is a good first step.

Chapter 5

Results

5.1 Efficiency

5.1.1 Time Profile

Using Apple's Instruments application, the library was profiled using the multipass example to check for any bottlenecks. The results are shown below.

%Time	Method	Library
95%	evk::Device::draw()	libevulkan.dylib
77.8%	vkQueueSubmit	libvulkan.1.2.131.dylib
5.8%	vkQueueWaitIdle	libvulkan.1.2.131.dylib
5.5%	vkQueuePresentKHR	libvulkan.1.2.131.dylib
3.1%	vkAcquireNextImageKHR	libvulkan.1.2.131.dylib
1.9%	vkWaitForFences	libvulkan.1.2.131.dylib
0.5%	vkResetFences	libvulkan.1.2.131.dylib
0.1%	evk::Device::primaryCommandBuffers() const	libevulkan.dylib
2.3%	_glfwPlatformPollEvents	libglfw.3.dylib
1.6%	evk::DynamicBuffer::update(void const*)	libevulkan.dylib

The column **% Time** refers to what percentage of the overall time was spent executing this command. Grey boxes indicate methods nested within other methods and their respective times are calculated in relation to the overall time, not the time of the parent method. For example, 77.8% of the overall program time was spent executing vkQueueSubmit. The **Library** column indicates the library from which the method originates, either the evulkan library, the official Vulkan library, or the GLFW library. The results are shown in decreasing amount of time spent - if a function is not in the table, it has a negligible cost.

The results indicate the library adds very little overhead to Vulkan. Considering that execution speed was not a goal for the project, this is an attractive bonus. Speed was not greatly affected for ease of use.

5.1.2 Benchmark

In addition to Instruments, a benchmarking program was written. It is located at examples/bench/main.cpp. It runs the setup and draw phases of the other example programs multiple times to gather data on timings. These timings were passed into Python to generate graphs. The generated graphs for draw and setup times are available in figures 6.4 and 6.5, on pages 43 and 44 respectively.

A test to compare the set up and draw time of a triangle with and without the evulkan library was also completed. The results are almost identical, with the evulkan triangle taking a few milliseconds longer than a pure Vulkan triangle. See figure 5.1 for more details.

	numThreads	numVerts	frame	startup
count	8290.0	8290.0	8290.000000	8290.000000
mean	1.0	3.0	0.928940	23.834535
std	0.0	0.0	0.859297	5.288742
min	1.0	3.0	0.316405	20.973300
25%	1.0	3.0	0.603134	21.430400
50%	1.0	3.0	0.713346	21.901700
75%	1.0	3.0	0.923519	22.927500
max	1.0	3.0	14.069900	52.624100

(a) Simple triangle.

	numThreads	numVerts	frame	startup
count	8380.0	8380.0	8380.000000	8380.000000
mean	1.0	3.0	1.330817	24.285771
std	0.0	0.0	0.725773	0.824338
min	1.0	3.0	0.868055	22.868000
25%	1.0	3.0	1.110620	23.849800
50%	1.0	3.0	1.175590	24.137200
75%	1.0	3.0	1.279552	24.598200
max	1.0	3.0	12.015900	29.349700

(b) evulkan triangle.

Figure 5.1: Test results for simple triangle and evulkan triangle.

The median (50%) frame draw time for the evulkan triangle is 0.4ms longer than that for the simple triangle, however the evulkan frame draw time has a shorter tail for the maximum value. The same is true for the startup time. While the draw and startup times might be slightly longer, they are much more consistent - the standard deviation of the simple triangle is 5.289 while that of the evulkan triangle is 0.824.

Note that the data has been stripped of outlier values (z value larger than 3). The experiment itself was run over 100 times for startup and within a startup another 100 times for drawing, resulting in 100,000 times overall. The Python code used to generate these results is in examples/bench/Bench.ipynb.

5.2 Usability

This is the main goal of the project. Usability refers to how well a person can use the library effectively. Usability here is measured in documentation, consistency and lines of code.

The code is fully documented with class and method comments. The instructions for how to download, build, install and use the library are available in this document. The code itself uses consistent naming schemes for methods and variables and is consistent with other open source projects, as it uses CMake for distribution and building. The code adheres to the Google Style Guide, has maximum 80 character length lines and it has `const` and `noexcept` methods where possible. Using the triangle example for comparison, the pure Vulkan implementation uses approximately 914 lines of code, while the evulkan implementation uses 94, almost a 10x decrease.

5.3 Availability

Availability refers to how easily the code is retrieved. The code is available on GitHub, and is structured in the standard open-source manner. It has examples, tests and the full source code available for distributing and building the library using CMake.

Chapter 6

Conclusion

This project is a success, resulting in an easy-to-use Vulkan library, removing much of the redundant and repetitive coding from the user while still allowing them access to many common Vulkan features.

6.1 What Could Have Been Better

6.1.1 COVID-19

Had COVID-19 not been an issue, given access to the lab machines with Linux and Windows, the library could have been tested across different configurations. In addition, it would have been useful to have classmates test the program and see if they could write a Vulkan program in a limited time frame.

6.1.2 Data-Driven Design

While care was taken to ensure this library was designed in a data-friendly manner, much of the time spent was ensuring that the library was easy to use and understand. While these two approaches often are in conflict, ease of use was paramount, leaving some cases where the solution is less-than optimal in terms of data access and overall speed. Starting off the project with a better understanding of Vulkan and library development would have made this step much easier and more fluid.

6.1.3 Mesh Shader

There are some new features that would be interesting to integrate with the library, specifically mesh shaders. However, the laptop on which the library was developed is relatively old (2015 MacBook Pro) and, as a result, has a relatively old integrated GPU (Intel Iris Graphics 6100). Without a machine on which these new features can be tested, it is impossible to ensure a feature is correctly integrated.

6.2 Future Developments

Possible future developments include exposing more Vulkan features (e.g. compute pipeline), adding more tests and integrating the library with NGL.

6.3 Alternatives Considered

6.3.1 Bazel

There were two options for the build system, Bazel (Google 2015) and CMake. Tests undertaken at the beginning of the project found that, while Bazel is good for large codebases and modeling interdependencies within the codebase, CMake is the standard used for distributing, building and installing open-source code across multiple systems.

6.3.2 Node-Based Graph

Instead of having a C++ library, an option was to create a node-based visual library, similar to how Houdini works. The advantages of this include an easier interface for users with limited programming experience.

However, this requires using a library to generate the graphs themselves as this is not a trivial exercise. Two open-source solutions were found online, but one of them was failing its build tests. The other (passing) library was an option. However, the author decided to focus on the full testing, documentation and completion of a stable and usable C++ library. Graphics programmers are expected to have a solid understanding of C++ so it is safe to assume that they would be able to set up and use a library such as this one.

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].

Crytek, 2002. *CryEngine*. 5.6.6. [computer program]. Frankfurt: Crytek.

Dos Reis, G., Garcia, J., Lakos, J., Meredith, A., Myers, N. and Stroustrup, B., 2018. *Support for contract based programming in C++* [online]. USA. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html. [Accessed 17 August 2020].

Epic Games, 1998. *Unreal Engine*. 4.25. [computer program]. Cary,NC: Epic Games.

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].

Google, 2015. *Bazel*. 3.1.0. [computer program]. Mountain View: Google.

Google, , 2020. *Google C++ Style Guide* [online]. USA: Google. Available from: https://google.github.io/styleguide/cppguide.html. [Accessed 17 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].

Khronos Group, , 2016d. *Vulkan Reference Guide* [online]. USA. Available from: https://www.khronos.org/files/vulkan11-reference-guide.pdf. [Accessed 17 August 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.

Stroustrup, B. and Sutter, H., 2020. *C++ Core Guidelines* [online]. USA: isocpp. Available from: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. [Accessed 17 August 2020].

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

Unity Technologies, 2005. *Unity*. 2020.1.2. [computer program]. San Francisco: Unity Technologies.

Valve, 2004. *Source*. 1.0. [computer program]. Bellevue: Valve.

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

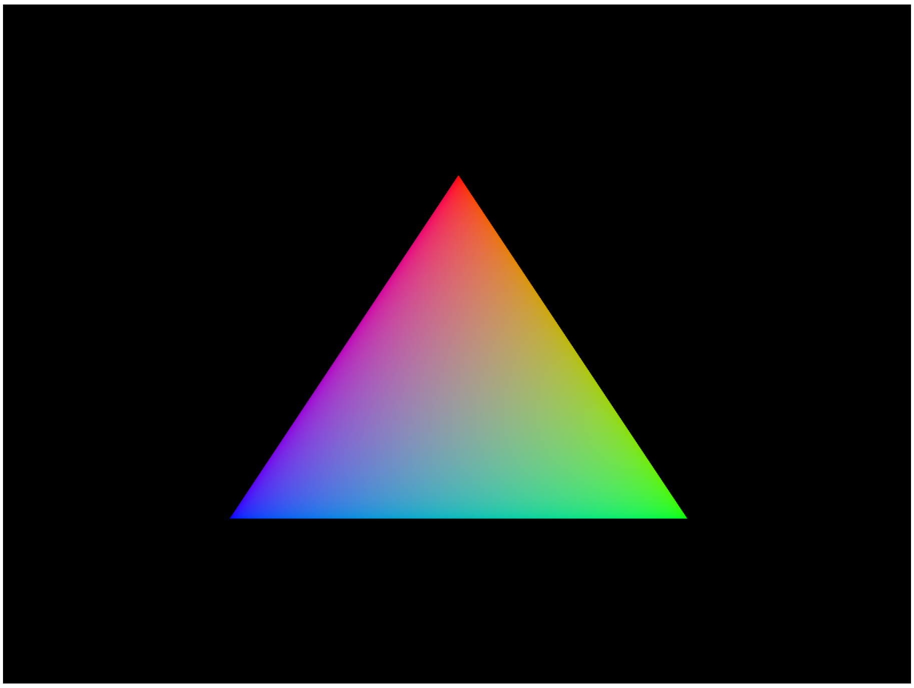

Figure 6.1: Triangle example in action.

Listing 6.1: Triangle drawing with evulkan library.

```
#include "evulkan/evulkan.h"

#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

using namespace evk;

int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    GLFWwindow *window=glfwCreateWindow(
      800, 600, "Vulkan", nullptr, nullptr
    );

    const uint32_t numThreads = 1;
    const uint32_t swapchainSize = 2;
```

```
Device device = Device(
    numThreads, deviceExtensions, swapchainSize,
    validationLayers
);

WindowResize r;
createSurfaceGLFW(device,window,r);

std::vector<Vertex> vertices=setupVerts();
std::vector<uint32_t> indices={0,1,2};

Attachment framebufferAttachment(
  device, 0, Attachment::Type::FRAMEBUFFER
);
Attachment depthAttachment(
  device, 1, Attachment::Type::DEPTH
);

std::vector<Attachment*> colorAttachments
  = {&framebufferAttachment};
std::vector<Attachment*> depthAttachments
  = {&depthAttachment};
std::vector<Attachment*> inputAttachments;
std::vector<Subpass::Dependency> dependencies;

Subpass subpass(
    0, dependencies, colorAttachments,
    depthAttachments, inputAttachments
);

std::vector<Subpass*> subpasses = {&subpass};
Renderpass renderpass(device, subpasses);

VertexInput vertexInput(sizeof(Vertex));
vertexInput.setVertexAttributeVec3(0,offsetof(Vertex,pos));
vertexInput.setVertexAttributeVec3(1,offsetof(Vertex,color));

StaticBuffer indexBuffer(
    device, indices.data(), sizeof(indices[0]),
    indices.size(), Buffer::Type::INDEX
);
StaticBuffer vertexBuffer(
    device, vertices.data(), sizeof(vertices[0]),
    vertices.size(), Buffer::Type::VERTEX
);

Shader vertexShader(
  device, "shader_vert.spv", Shader::Stage::VERTEX
);
Shader fragmentShader(
  device, "shader_frag.spv", Shader::Stage::FRAGMENT
);
std::vector<Shader*> shaders = {&vertexShader,&fragmentShader};

Pipeline pipeline(
  device, subpass, vertexInput, renderpass, shaders
);
std::vector<Pipeline*> pipelines = {&pipeline};
```

```
    device.finalize(indexBuffer,vertexBuffer,pipelines);

    // Main loop.
    while(!glfwWindowShouldClose(window))
    {
        glfwPollEvents();
        device.draw();
    }
}
```

Figure 6.2: OBJ example in action.

Listing 6.2: OBJ drawing with evulkan library.

```
#include "evulkan/evulkan.h"

using namespace evk;

struct UniformBufferObject
{
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};

int main(int argc, char **argv)
{
    gflags::SetUsageMessage(
        "A program demonstrating how to use OBJs and
        textures in Vulkan."
    );
    gflags::ParseCommandLineFlags(&argc, &argv, true);

    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    GLFWwindow *window=glfwCreateWindow(
      800, 600, "Vulkan", nullptr, nullptr
    );

    const uint32_t numThreads = FLAGS_num_threads;
    const uint32_t swapchainSize = 2;

    Device device(
        numThreads, deviceExtensions, swapchainSize,
        validationLayers
```

```
);

WindowResize r;
createSurfaceGLFW(device, window, r);

std::vector<Vertex> v;
std::vector<uint32_t> in;
evk::loadOBJ("viking_room.obj", v, in);

Texture texture(device, "viking_room.png");

Descriptor descriptor(device, swapchainSize);
descriptor.addTextureSampler(
  1, texture, Shader::Stage::FRAGMENT
);

Attachment framebufferAttachment(
  device, 0, Attachment::Type::FRAMEBUFFER
);
Attachment depthAttachment(
  device, 1, Attachment::Type::DEPTH
);

std::vector<Attachment*> colorAttachments
  = {&framebufferAttachment};
std::vector<Attachment*> depthAttachments
  = {&depthAttachment};
std::vector<Attachment*> inputAttachments;
std::vector<Subpass::Dependency> dependencies;

Subpass subpass(
    0, dependencies, colorAttachments,
    depthAttachments, inputAttachments
);

std::vector<Subpass*> subpasses = {&subpass};
Renderpass renderpass(device,subpasses);

UniformBufferObject uboUpdate = {};
uboUpdate.model=glm::mat4(1.0f);
uboUpdate.model=glm::rotate(
    glm::mat4(1.0f), 0.001f * glm::radians(90.0f)*0,
    glm::vec3(0.0f,0.0f,1.0f)
);
uboUpdate.view = glm::lookAt(
    glm::vec3(2.0f, 2.0f, 2.0f),
    glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f)
);
uboUpdate.proj = glm::perspective(
    glm::radians(45.0f), 800 / (float) 600 , 0.1f, 10.0f
);
uboUpdate.proj[1][1] *= -1;
DynamicBuffer ubo(
    device, &uboUpdate, sizeof(uboUpdate),
    1, Buffer::Type::UBO
);
descriptor.addUniformBuffer(0, ubo, Shader::Stage::VERTEX);
```

```
VertexInput vertexInput(sizeof(Vertex));
vertexInput.setVertexAttributeVec3(0,offsetof(Vertex,pos));
vertexInput.setVertexAttributeVec3(1,offsetof(Vertex,color));
vertexInput.setVertexAttributeVec2(2,offsetof(Vertex,texCoord));

StaticBuffer indexBuffer(
    device, in.data(), sizeof(in[0]),
    in.size(), Buffer::Type::INDEX
);
StaticBuffer vertexBuffer(
    device, v.data(), sizeof(v[0]),
    v.size(), Buffer::Type::VERTEX
);

Shader vertexShader(
  device, "shader_vert.spv", Shader::Stage::VERTEX
);
Shader fragmentShader(
  device, "shader_frag.spv", Shader::Stage::FRAGMENT
);
std::vector<Shader*> shaders = {&vertexShader,&fragmentShader};

Pipeline pipeline(
    device, subpass, descriptor, vertexInput,
    renderpass, shaders
);

std::vector<Pipeline*> pipelines = {&pipeline};

device.finalize(indexBuffer,vertexBuffer,pipelines);

// Main loop.
size_t counter=0;
while(!glfwWindowShouldClose(window))
{
    glfwPollEvents();

    UniformBufferObject uboUpdate = {};
    uboUpdate.model=glm::mat4(1.0f);
    uboUpdate.model=glm::rotate(
        glm::mat4(1.0f), 0.001f * glm::radians(90.0f)*counter,
        glm::vec3(0.0f,0.0f,1.0f)
    );
    uboUpdate.view = glm::lookAt(
        glm::vec3(2.0f, 2.0f, 2.0f),
        glm::vec3(0.0f, 0.0f, 0.0f),
        glm::vec3(0.0f, 0.0f, 1.0f)
    );
    uboUpdate.proj = glm::perspective(
        glm::radians(45.0f), 800 / (float) 600 , 0.1f, 10.0f
    );
    uboUpdate.proj[1][1] *= -1;
    ubo.update(&uboUpdate);

    device.draw();

    counter++;
```

```
      }
}
```

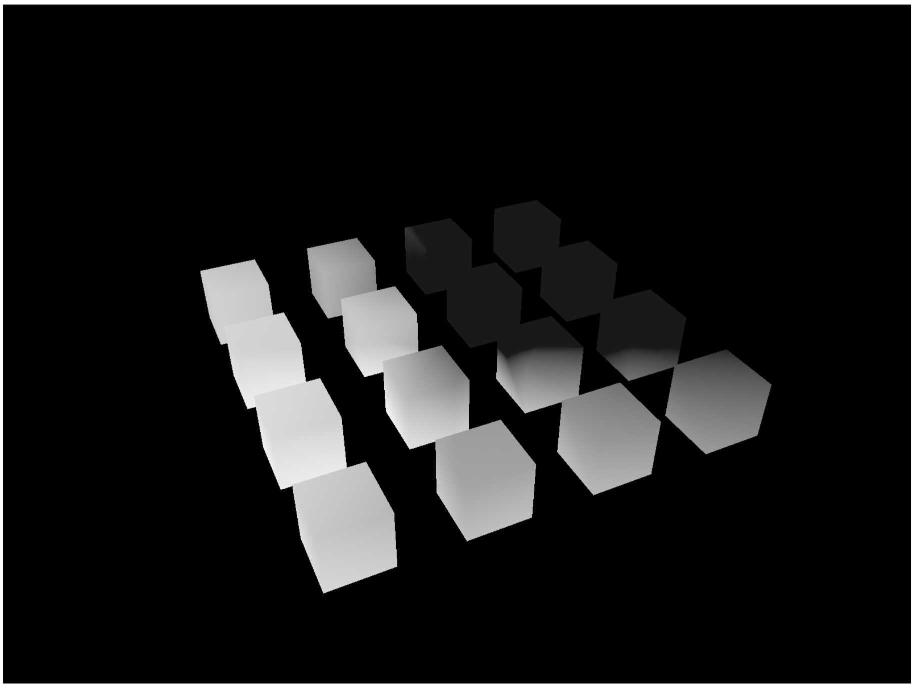

Figure 6.3: Multipass example in action.

Listing 6.3: Multipass drawing with evulkan library.

```
#include "evulkan/evulkan.h"

using namespace evk;

struct UniformBufferObject
{
    glm::mat4 MVP_model;
    glm::mat4 MVP_light;
    glm::mat4 MV;
};

int main(int argc, char **argv)
{
    gflags::SetUsageMessage(
        "A program for using multipass
        Vulkan over multiple threads."
    );
    gflags::ParseCommandLineFlags(&argc, &argv, true);

    const uint32_t numThreads = FLAGS_num_threads;
    const uint32_t swapchainSize = 2;

    glfwInit();
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    GLFWwindow *window=glfwCreateWindow(
      800, 600, "Vulkan", nullptr, nullptr
    );

    std::vector<Vertex> vertices;
    std::vector<uint32_t> indices;
    createGrid(FLAGS_num_cubes, vertices, indices);
```

```cpp
Device device(
    numThreads, deviceExtensions, swapchainSize,
    validationLayers
);

WindowResize r;
createSurfaceGLFW(device, window, r);

Attachment framebufferAttachment(
  device, 0, Attachment::Type::FRAMEBUFFER
);
Attachment colorAttachment(device, 1, Attachment::Type::COLOR);
Attachment depthAttachment(device, 2, Attachment::Type::DEPTH);

std::vector<Attachment*> colorAttachments = {&colorAttachment};
std::vector<Attachment*> depthAttachments = {&depthAttachment};
std::vector<Attachment*> inputAttachments;
std::vector<Subpass::Dependency> dependencies;

Subpass subpass0(
    0, dependencies, colorAttachments, depthAttachments,
    inputAttachments
);

colorAttachments = {&framebufferAttachment};
depthAttachments.resize(0);
inputAttachments = {&colorAttachment, &depthAttachment};
dependencies = {0};
  // Require previous subpass to complete before this one.

Subpass subpass1(
    1, dependencies, colorAttachments, depthAttachments,
    inputAttachments
);

std::vector<Subpass*> subpasses = {&subpass0, &subpass1};
Renderpass renderpass(device,subpasses);

DynamicBuffer ubo(
  device, sizeof(UniformBufferObject), Buffer::Type::UBO
);

Descriptor descriptor0(device, swapchainSize);
descriptor0.addUniformBuffer(0, ubo, Shader::Stage::VERTEX);

Descriptor descriptor1(device, swapchainSize);
descriptor1.addUniformBuffer(0, ubo, Shader::Stage::VERTEX);
descriptor1.addInputAttachment(
  0, colorAttachment, Shader::Stage::FRAGMENT
);
descriptor1.addInputAttachment(
  1, depthAttachment, Shader::Stage::FRAGMENT
);

VertexInput vertexInput0(sizeof(Vertex));
vertexInput0.setVertexAttributeVec3(0,offsetof(Vertex,pos));
vertexInput0.setVertexAttributeVec3(1,offsetof(Vertex,normal));
```

```
VertexInput vertexInput1(sizeof(Vertex));
vertexInput1.setVertexAttributeVec3(0,offsetof(Vertex,pos));

StaticBuffer indexBuffer(
    device, indices.data(), sizeof(indices[0]),
    indices.size(), Buffer::Type::INDEX
);
StaticBuffer vertexBuffer(
    device, vertices.data(), sizeof(vertices[0]),
    vertices.size(), Buffer::Type::VERTEX
);

Shader vertexShader0(
  device, "pass_0_vert.spv", Shader::Stage::VERTEX
);
Shader fragmentShader0(
  device, "pass_0_frag.spv", Shader::Stage::FRAGMENT
);
std::vector<Shader*> shaders0 = {
  &vertexShader0, &fragmentShader0
};

Pipeline pipeline0(
    device, subpass0, descriptor0,
    vertexInput0, renderpass, shaders0
);

Shader vertexShader1(
  device, "pass_1_vert.spv", Shader::Stage::VERTEX
);
Shader fragmentShader1(
  device, "pass_1_frag.spv", Shader::Stage::FRAGMENT
);
std::vector<Shader*> shaders1 = {
  &vertexShader1, &fragmentShader1
};

Pipeline pipeline1(
    device, subpass1, descriptor1, vertexInput1,
    renderpass, shaders1
);
std::vector<Pipeline*> pipelines = {&pipeline0, &pipeline1};

device.finalize(indexBuffer,vertexBuffer,pipelines);

// Main loop.
size_t counter=0;
glm::mat4 model(1.0f);
glm::mat4 view(1.0f);
glm::mat4 proj(1.0f);
while(!glfwWindowShouldClose(window))
{
    glfwPollEvents();

    UniformBufferObject uboUpdate = {};
    model = glm::rotate(
        glm::mat4(1.0f), 0.005f * glm::radians(90.0f)*counter,
```

```
            glm::vec3(0.0f,0.0f,1.0f)
        );
        view = glm::lookAt(
            glm::vec3(2.0f, 2.0f, 2.0f),
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::vec3(0.0f, 0.0f, 1.0f)
        );
        proj = glm::perspective(
            glm::radians(45.0f), 800 / (float) 600 , 0.1f, 10.0f
        );
        proj[1][1] *= -1;
        uboUpdate.MV = view * model;
        uboUpdate.MVP_model = proj * view * model;
        uboUpdate.MVP_light = proj * view;

        ubo.update(&uboUpdate);

        device.draw();

        counter++;
    }

    // Tidy up.
    glfwDestroyWindow(window);
    glfwTerminate();
}
```

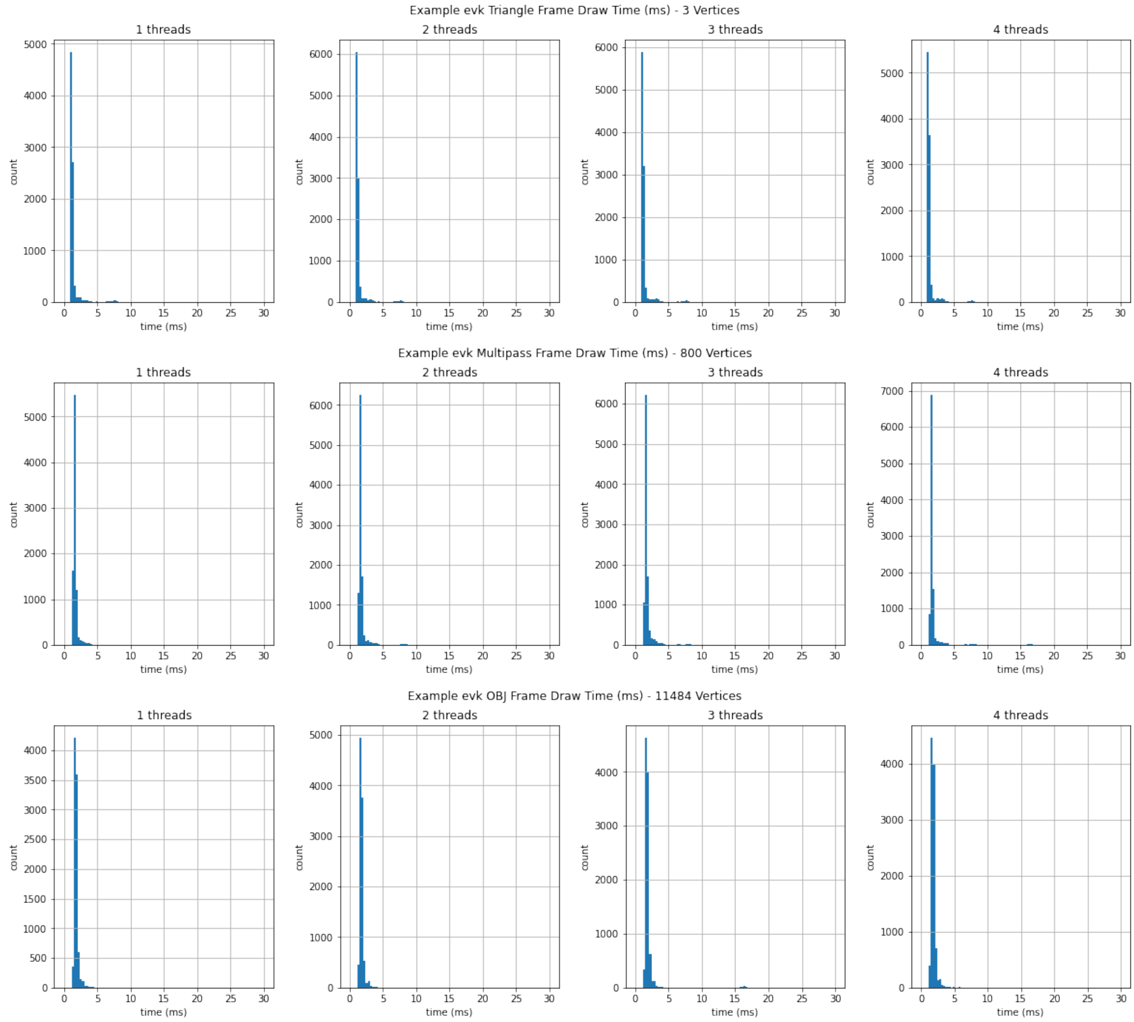

Figure 6.4: Draw time for different examples over multiple threads.

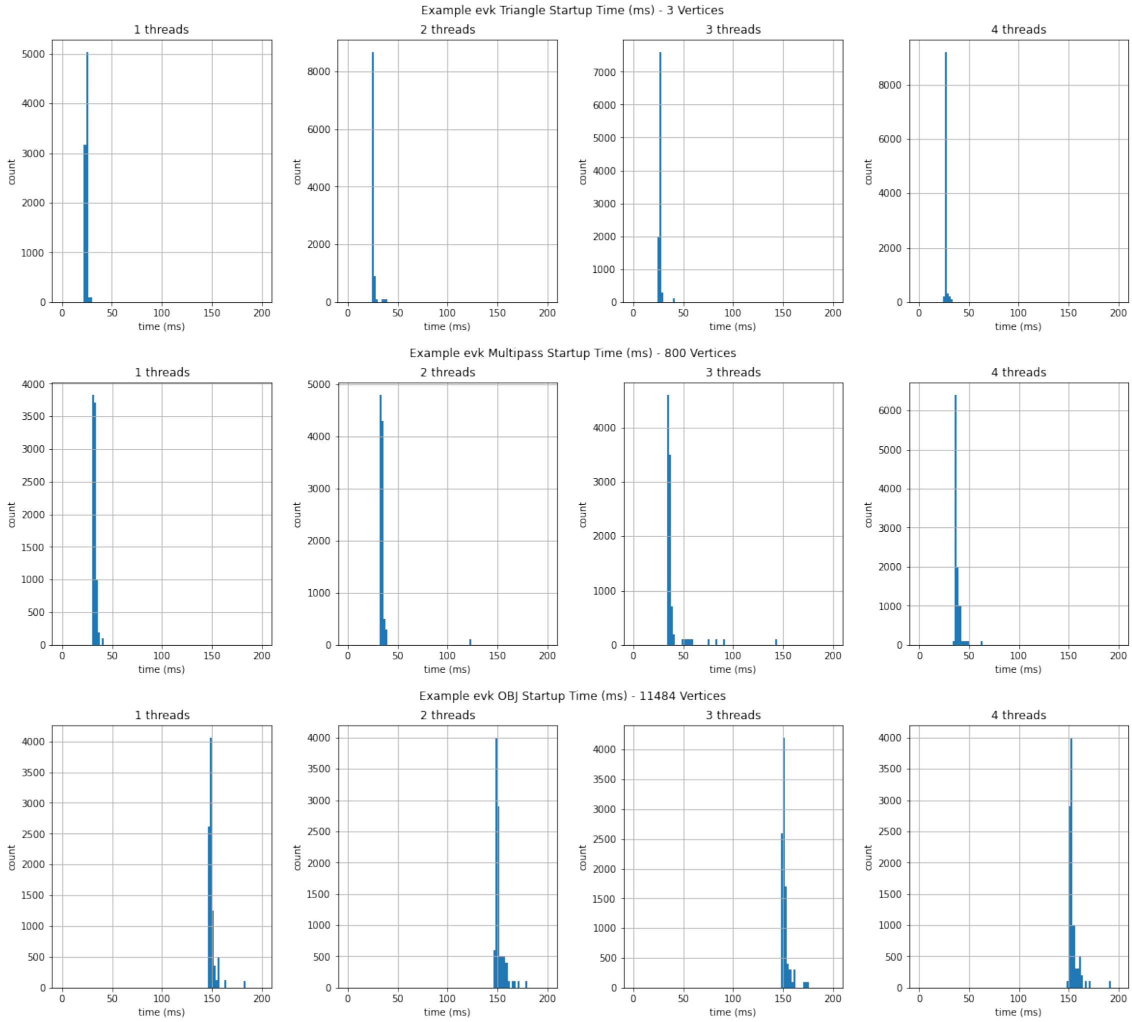

Figure 6.5: Setup time for different examples over multiple threads.