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 2016) 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.*

Chapter 3

Technical Background

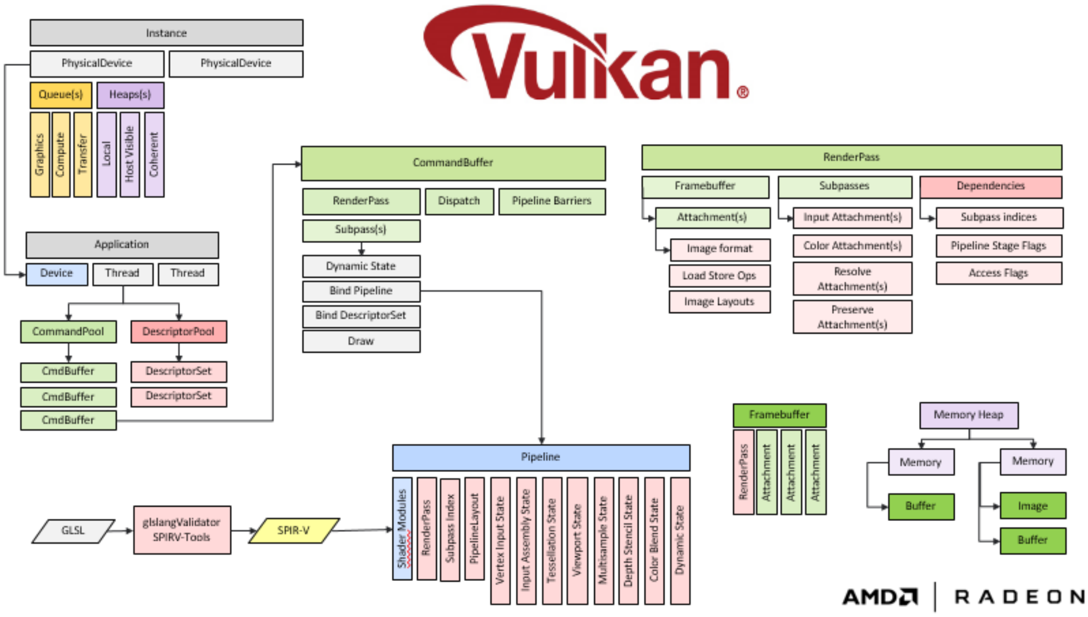

Figure 3.1: Vulkan API objects and their interactions (AMD 2018, p.1).

3.1 Limitations of OpenGL

OpenGL, the current cross-platform industry standard, was first released in 1992.

Chapter 4

The evulkan Library

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

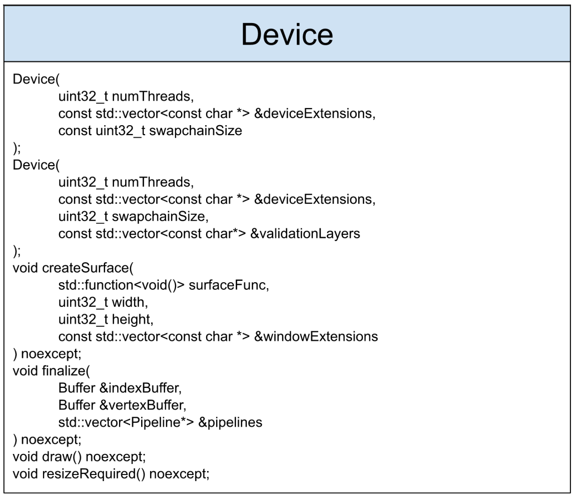

Figure 4.1: Device class diagram.

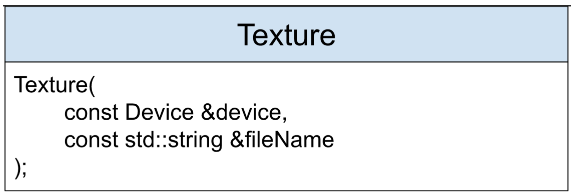

Figure 4.2: Texture class diagram.

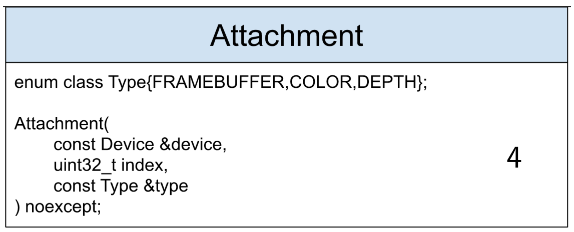

Figure 4.3: Attachment class diagram.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

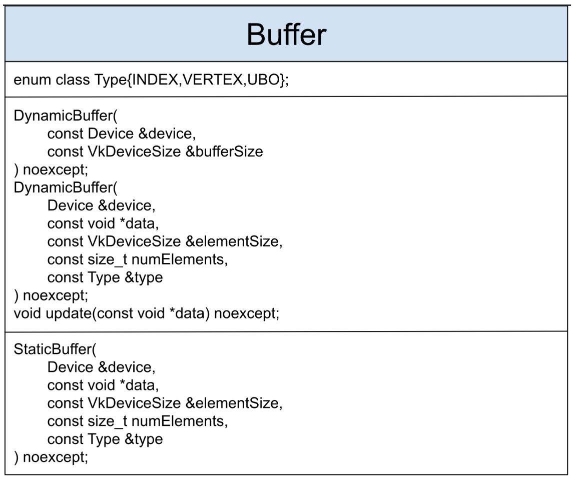

Figure 4.4: Buffer class diagram.

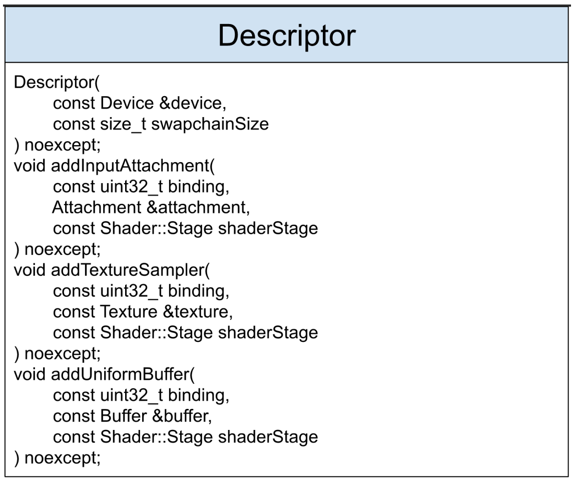

Figure 4.5: Descriptor class diagram.

Subpass

```
typedef uint32_t Dependency;

Subpass(
  const uint32_t index,
  const std::vector<Dependency> &dependencies,
  const std::vector<Attachment*> &colorAttachments,
  const std::vector<Attachment*> &depthAttachments,
  const std::vector<Attachment*> &inputAttachments
) noexcept;
```

Figure 4.6: Subpass class diagram.

Renderpass

```
Renderpass(
      const Device &device,
      std::vector<Subpass*> &subpasses
) noexcept;
```

Figure 4.7: Renderpass class diagram.

Pipeline

```
Pipeline(
      Device &device,
      Subpass &subpass,
      Descriptor &descriptor,
      const VertexInput &vertexInput,
      Renderpass &renderpass,
      const std::vector<Shader*> &shaders
) noexcept;

Pipeline(
      Device &device,
      Subpass &subpass,
      const VertexInput &vertexInput,
      Renderpass &renderpass,
      const std::vector<Shader*> &shaders
) noexcept;
```

Figure 4.8: Pipeline class diagram.

Shader

```
enum class Stage{VERTEX,FRAGMENT};
Shader(
      const Device &device,
      const std::string &fileName,
      const Stage &stage
);
```

Figure 4.9: Shader class diagram.

VertexInput

```
VertexInput(uint32_t stride) noexcept;
void setVertexAttributeVec2(
      uint32_t location, uint32_t offset
) noexcept;
void setVertexAttributeVec3(
      uint32_t location, uint32_t offset
) noexcept;
```

Figure 4.10: VertexInput class diagram.

Chapter 5

Conclusion

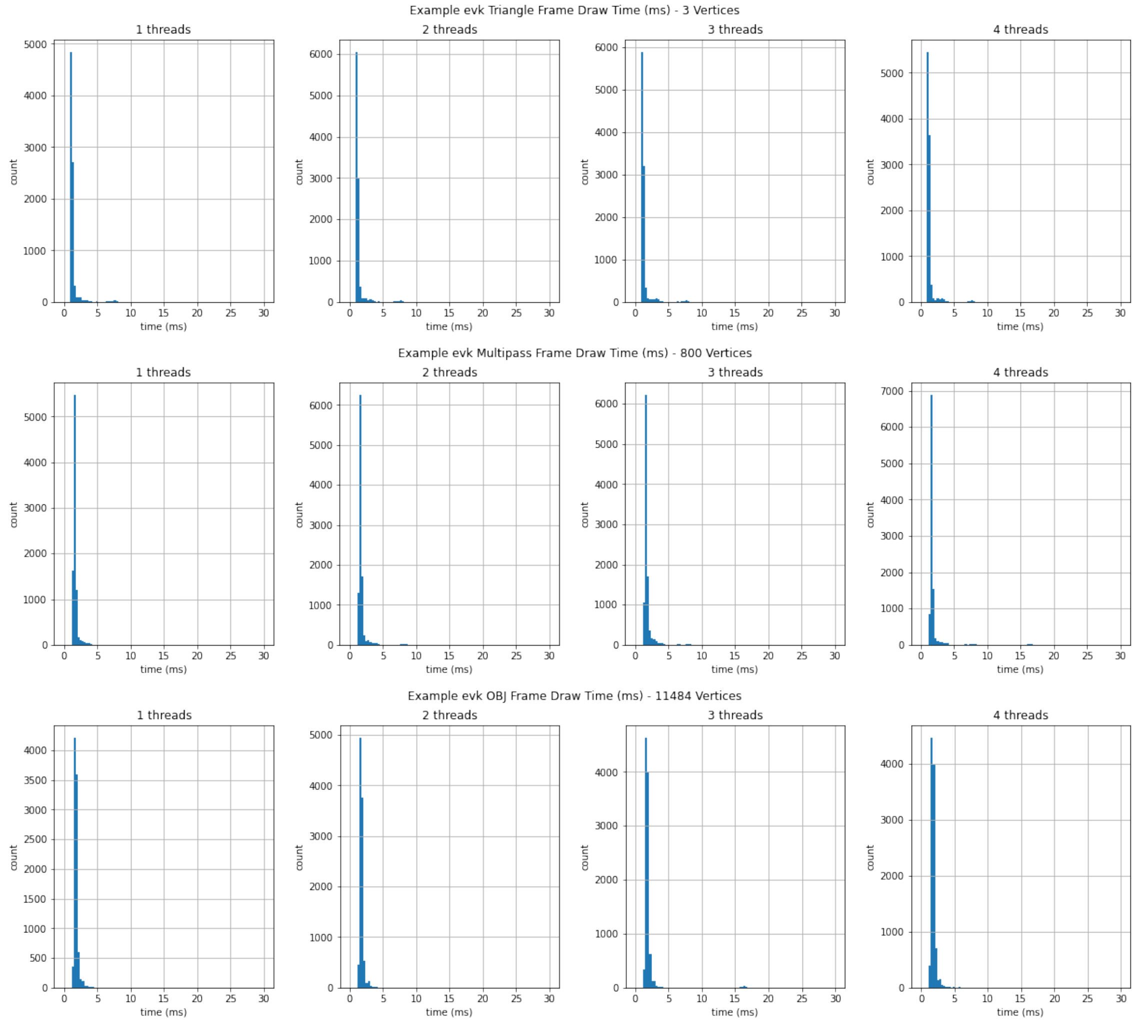

Figure 5.1: Draw time for different examples over multiple threads.

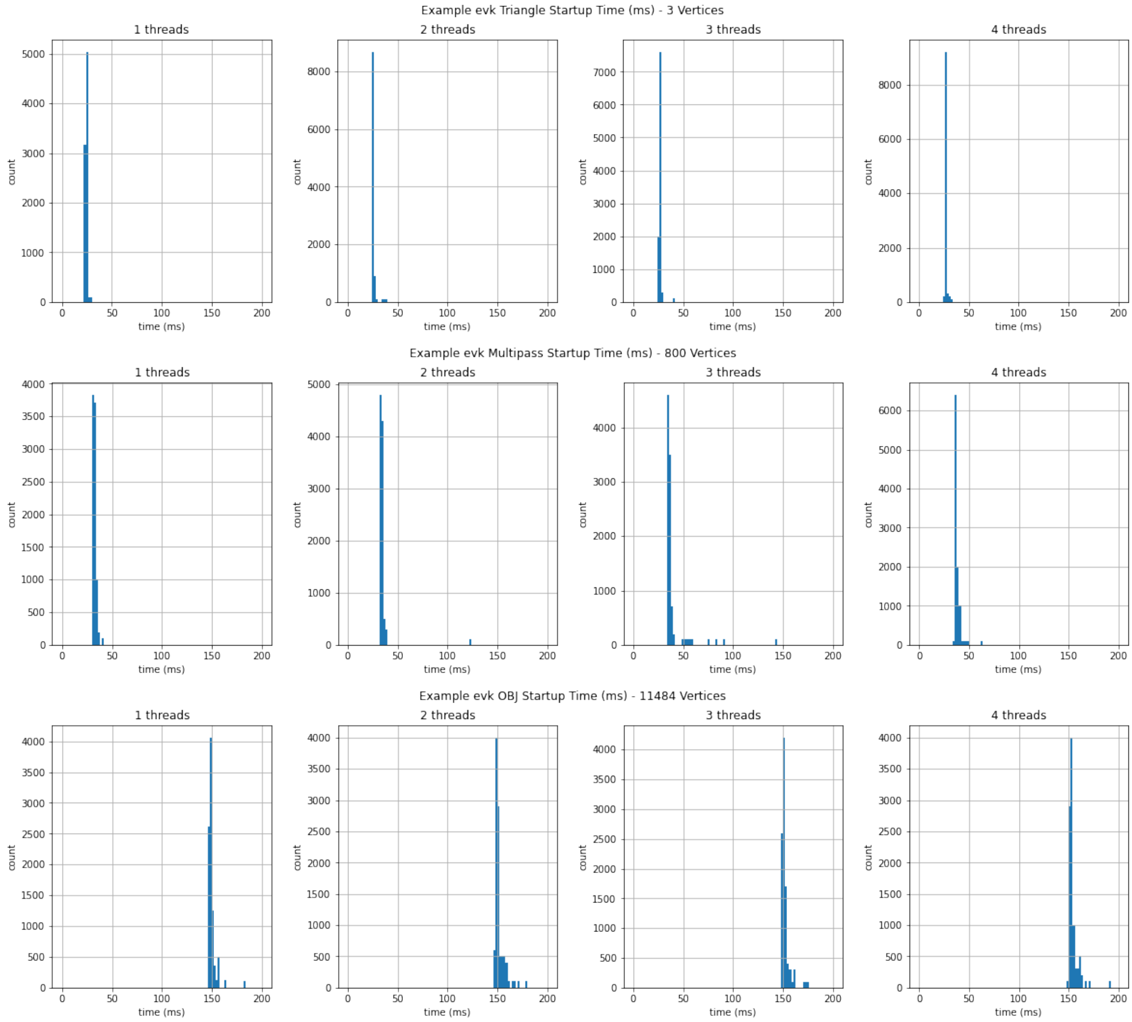

Figure 5.2: Setup time for different examples over multiple threads.

Bibliography

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

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

Appendices