AN EXPLORATORY STUDY OF HIGH PERFORMANCE GRAPHICS APPLICATION PROGRAMMING INTERFACES

By

JOSEPH SHIRAEF

Dr. Yu Liang
Professor of Computer Science
(Chair)

Dr. Joseph Kizza
Department Head and Professor
of Computer Science
(Committee Member)

Dr. Craig Tanis
Professor of Computer Science
(Committee Member)

AN EXPLORATORY STUDY OF HIGH PERFORMANCE GRAPHICS APPLICATION PROGRAMMING INTERFACES

By

JOSEPH SHIRAEF

A Thesis Submitted to the Faculty of the University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements of the Degree of
Master of Science: Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

May 2016

ABSTRACT

This study was conducted to take an in depth look at the newest application programming interfaces (API) offered to graphics programmers. With the recent releases of Vulkan (2016) and DirectX 12 (2015) from industry giants like the Khronos Group and Microsoft, it's clear they are pushing for a much lower-level, closer-to-hardware approach for future graphics programming solutions. These changes can be credited to the drastic improvements we've seen in graphics processors over the last 5 years. It will take a significant amount of time for these API's to become industry standard. The goal of this research is to verify the value and benefits of developing with these API's as opposed to using the current industry standard OpenGL or DirectX 11. Several GPU & CPU benchmark performance tests have brought interesting results. Furthermore, many advanced computer graphical techniques and algorithms which are implemented using C++ and Vulkan, help to shine a spotlight on the glaring contrast between Vulkan and OpenGL. This research attempts to be one of the first validations for advantages or disadvantages the Vulkan API offers in comparison to its predecessors.

DEDICATION

This thesis is dedicated to my fellow graduate students and the staff of the Department of Computer Science and Engineering at the University of Tennessee at Chattanooga. I would like to express my deepest gratitude to the computer science scholars who work on the 3rd floor of the EMCS building. Thank you for your kind words and support over the last 3 years. I would not have made it this far without you. This thesis is also dedicated to my patient and wonderful wife, Kristin Shiraef.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Yu Liang for his feedback, support, and encouragement throughout this research. My thanks also goes to Dr. Joseph Kizza for serving on my committee and providing valuable advice and support at various stages of my Master of Science. My thanks also goes to Dr. Craig Tanis for serving on my committee and providing his assistance and expertise during my studies. I would also like to thank my friends and peers whose enthusiasm and support have helped me to survive the highs and lows of the past 3 years in the College of Computer Science and Engineering.

TABLE OF CONTENTS

APPENDIX

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACE, Asynchronous Compute Engines

API, Application Programming Interface

APU, Accelerated Processing Unit

AWS, Amazon Web Service

CDT, Constrained Delaunay Triangulation

CGT, Computer Graphics Technology

DX, Direct X

FPGA, Field Programmable Gate Array

FPS, Frames Per Second

GLSL, OpenGL Shading Language

HLSL, High Level Shading Language

GPU, Graphical Processing Units

GUI, Graphical User Interface

IHV, Independent Hardware Vendor

ISV, Independent Software Vendor

KHR, Khronos Registry

MSVS, Microsoft Visual Studio

MoCap, Motion Capture

OS, Operating System

OpenGL, Open Graphics Library

OpenCL, Open Computing Language

PCA, Principle Components Analysis

PFN, Pointer to Function

POM, Parallax Occlusion Mapping

PSO, Pipeline State Object

RAM, Random Access Memory

ROM, Read-Only Memory

SPIR-V, Standard Portable Immediate Representation - Vulkan

UTC, University of Tennessee at Chattanooga

UX, User Experience

VR, Virtual Reality

CHAPTER 1

INTRODUCTION

As a developer, your application's programming interface (API) is your most important toolset and its documentation is essentially your bible. You will constantly be referring to it and keeping tabs on the latest updates. Vulkan and DirectX 12 aim to give the programmer more lower-level access to the GPU and CPU cores. This should greatly increase the multithreading capabilities for graphics intensive applications. The central focus of this new generation of APIs is to increase the amount of draw calls possible while decreasing the amount of overhead for the CPU. This could potentially change the way procedural graphics programmers work and think, because optimization is no longer an issue that is magically handled behind-the-curtain by the API.

High performance computing infrastructure has been of significant importance in solving scientific, engineering and data analysis problems. Practical applications such as 3D visualization or 3D modeling software will almost always demand graphically intense rendering. As the hardware of desktops and mobile devices continues to improve so does the possibility for drastic improvements in number or draw calls from the renderer. (Lindholm, 2008). For decades OpenGL has become more powerful and accessible enabling graphics programmers to build graphically intense applications on standard desktops or even mobile devices using OpenGL ES.

Early this year (2016) the Khronos Group, the consortium behind OpenGL, rolled out Vulkan (previously known as nextGL). Vulkan is intended to provide a variety of advantages over other APIs including the industrial standard OpenGL. Most importantly it will offer lower overhead, more direct control over the GPU, and lower CPU usage.

The emergence of this new Vulkan API along with a new ecosystem has excited many developers with the belief that it will create new business opportunities for both the graphics and computing industry. Vulkan plays a greatly significant part in eliminating expensive driver operations, which translates to less CPU overhead thus reducing or completely eradicating unexpected frame rate discontinuities or hiccups. Therefore, the evolution of this new approach of explicit GPU control will play a critical role in enabling developers to avail the best probable experience to clients on multiple platforms. Work for completion of Vulkan 2.0 SDKs for Android, windows and Linux is in progress. Furthermore, Google has also upgraded to Promoter membership partnering with the Khronos Board in steering Vulkan strategy for both the wider android industry. Considerable energy is therefore applied towards driving the operation necessary to release to the broader computing world and all platforms (Khronos, 2015). To make this a success, the Khronos group is planning for Vulkan sessions and demonstration at key industrial events throughout the year.

One key advantage that Vulkan claims to hold over OpenGL includes its capability in generating work for the GPU across many CPU threads. This will make Vulkan distinctively useful for developers who find themselves CPU-bound. Vulkan enlarges the Khronos 3D APIs family by supplementing OpenGL and OpenGL ES, which are two standards which provide access to millions of GPUs today. More so, NVIDIA, the leading GPU manufacturer, is said to be operating within the confines of Khronos ensuring evolution of Vulkan to meet industrial demands (Khronos, 2015).

OpenGL uses the high-level language GLSL for writing shaders which forces each OpenGL driver to implement its own compiler for GLSL that executes at application runtime to translate the program's shaders into executable code for the target platform. Vulkan will instead provide an intermediate binary format called SPIR-V (Standard Portable Intermediate Representation), analogous to the binary format that HLSL shaders are compiled into in DirectX. This reduces the workload on the drivers made from software vendors. This will also in turn allow shader pre-compilation as well as permit application developers to write shaders in languages other than only GLSL. Other obvious advantages include cross-platform API supported on both mobile devices and high-end graphics cards and better support for modern systems that use multithreading. Most of all, having direct control over the GPU will reduce the load on CPUs in situations where the CPU is the bottleneck, allowing higher throughput for GPU calculations and rendering.

The introduction of DirectX 12 by Microsoft has also promised to be a graphics API which enables a console-like low-level access to the CPU and GPU, thus improving the performance for existing graphics cards. It is unlikely that not all graphic cards support each feature or attribute of DX 12 due to the fact that the APIs are split into disparate feature levels. Nevertheless, the most significant features of DX 12 are supported throughout or across the board. Theoretically, it is evident that before people move to DX 12, they should see some promising sort of performance uplift. Additionally, DirectX 12 introduces command lists or instruction sets which are essentially important in the execution of particular workload on the CPU. Since each of the command lists is sufficient in itself, the pre-computation of all the necessary GPU commands by the driver up front and in a free threaded manner across any CPU core is possible (William, 2008). The sole serial process involved includes submission of the final command lists or instructions set to the GPU, which theoretically is a highly efficient and effective process. Execution process of command lists in the GPU take places in a parallel manner and therefore serial execution is eliminated hence DirectX 12 increases performance.

Because DirectX 12 is proprietary to Microsoft, there is always some competition to be realized, which means the reinventions of these APIs are surrounded by much promise and hype as the release draws near. DirectX 12 was released during Summer 2015. As expected there were many proclamations about how this new iteration will offer "astounding" new features which will enhance graphics capabilities across all platforms while lowering overhead. They were claiming to achieve massively increased framerates. They also claim to have the ability to combine performance among GPU's which would mean a drastic reduction in CPU bottleneck.

During the DX 11 generation, Nvidia was considered the undisputed leader, but the rolling out of DX 12 will provide greatly awaited news for AMD. Under DX 12, the organization's GCN architecture which featured the underperforming asynchronous compute engines (ACE) will finally operate with tasks such as lighting, physics and post processing. These tasks are divided into various queues and time lined independently for executing or processing by the GPU. Another big feature in DX 12 that will be of great importance and interest to those with iGPU or APU is the Explicit Multiadaptor. With DirectX 12, support for numerous GPUs is molded into the Application Program Interface, allowing distinguishable and adjacent workloads to be performed and executed in parallel on various GPUs, regardless of where they come from; either from AMD, Intel, or Nvidia. Finally, DirectX 12 is expected to allow for many GPUs to puddle their memory.

So it seems this year, 2016 and for years to come, graphics programmers and game developers should have a bunch of new toys to keep them busy. According to recent tech demos performed by seemingly unbiased journalists and tech reviewers alike, DirectX 12 does in fact dramatically increase the amount of draw calls possible while utilizing the available hardware. This has been tested with processors from Nvidia, AMD, and Intel. If it is true that DX12 and Vulkan, the new generation of OpenGL, can achieve much lower overhead while performing 4X or 5X the normal amount of draw calls, then the Golden Age for advanced graphics capabilities has arrived.

1.1 Scope

The scope of this research is limited to creating simple graphical programs using next generation APIs and comparing performance, syntax, and the overall benefits of the API concerning optimization. Based on the simple tests and examples, I hope to provide interesting results which can verify the potential increase of framerates, timing, and overall performance thus presenting a new and more optimum way for building graphics intensive applications in the future.

1.2 Significance

There are three major contributions of this work: analysis of the current status in the field of computer graphics, high performance optimization of modern hardware, and conducting research and test to advance the improvement of future software applications through exploitation of next generation APIs. The current paradigm for loading graphics drivers is the real problem. A developer should not be forced to draw a primitive with every tick of the GPU. At times, they might want to perform a workload of linear algebra through it, while other requests could simply be pushing memory around to set up a later task. More importantly, as hardware continues to significantly improve in desktop and mobile devices, it is important or programmers to be able to take full advantage of the multiple available cores. The overall agenda here is to increase familiarity with these new APIs and the opportunity to greatly benefit software developers and programmers alike.

1.3 Research Questions

The major questions behind the project are:

1. What is the optimum way to utilize your hardware when using the latest graphics APIs (Vulkan, DirectX 12)?
2. How significant of an increase in performance can be achieved by using next generation graphics APIs?

1.4 Assumptions

The assumptions we have in this thesis are:

1. The 3DMark/futuremark test is an effective method for stress testing modern hardware with accurate results

2. Modern GPUs operate closer to a GPU Compute APIs than they did in the 1990s versions of OpenGL and DirectX.

3. The true benefits of next generation APIs may not be realized until they grow in popularity.

4. The APIs will release updated iterations (Vulkan 1.1, 1.2) throughout 2016, Here benchmark results are limited to 2015 standards.

1.5 Limitations

1. In this study benchmark results are limited to 2015 standards. The next generation APIs will eventually release updated iterations (Vulkan 1.1, 1.2) throughout 2016.
2. Any software applications built using Vulkan before 2016 were using an early look at Vulkan (beta) and cannot be considered a prime example of CPU utilization or draw call improvements.

1.6 Delimitations

1. Differentiation of proprietary hardware, software, operating systems, etc. is not considered.
2. The benchmark tests will be performed on modern hardware and the latest versions of Operating Systems

1.7 Definitions

draw call – To draw a line, curve, or object on the screen, the application must issue a function call to the graphics API(e.g. OpenGL or Direct3D). Draw calls can often be expensive, with the graphics API doing significant work for every draw call, causing performance overhead on the CPU side.

Metal - is a low-level, low-overhead hardware-accelerated graphics and compute API that debuted in iOS 8. It combines functionality similar to OpenGL and OpenCL under one API.

mental ray - production quality rendering application capable of achieving high performance graphics computing through parallelism on multiprocessor machines developed by Mental Images.

trajectory – A curve made by collected points of joint locations in time.

key frame – "In computer animation, the term *key frame* has been generalized to apply to any variable whose value is set at specific key frames and from which values for the intermediate frames are interpolated according to some prescribed procedure." (Parent, 2007)

overhead – any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to attain a particular goal or job.

sequence – "Here, the overall animation–the entire project–is referred to as the *production*. Typically, productions are broken into major parts referred to as *sequences*. A *sequence* is a major episode and is usually identified by an associated staging area; a production usually consists of one to a dozen sequences."

1.9 Summary

This chapter presents an introduction to the current status of the most popular programming APIs being used within the field of computer graphics. Then some brief insight and analysis is offered concerning the fresh new graphics programming APIs DirectX 12 and Vulkan, which have just recently been launched in 2015/2016. I discuss my observations about how these APIs could potentially benefit the field of computer graphics, specifically programmers. Also briefly discussed in this introductory chapter are ideas for tests, the scope of this project, limitations, delimitations, and my contribution in this thesis. The terms and acronyms for the following chapters are defined.

CHAPTER 2

REVIEW OF RELEVANT LITERATURE

In this chapter, we review the previous work related to recent advancements and developments in the field of computer graphics. This chapter consists of four sections. In section 2.1, the most popular graphics API, OpenGL is introduced and discussed. In section 2.2 the inception of the Vulkan API and the purpose it serves for developers. In the third section (Section 2.3), we take a look at Mantle, the original API from AMD which helped spark the Khronos Group to launch a new APIs. Next in section 2.4 through 2.8, the limitations of each API are analyzed, compared, and further discussed. In Section 2.9 we are introduced to SPIR-V, Vulkan's answer for shader translation and compilation. The final section presents the various pros and cons to be offered from Vulkan and DirectX 12 (Section 2.11).

2.1 OpenGL Overview

Since OpenGL was released in 1992 by Silicon Graphics Inc., it has been widely adopted throughout the entire industry as well as academia. Over the past 20 years, the Khronos Group has managed the API, and released many significant updates. OpenGL consists of a library of over 500 function calls which perform 3D tasks (Guha, 2014). OpenGL can be accessed from applications written in various programming languages and it can be run across various operating systems. It was essentially the first API to popularize 3D graphics programming across the software industry. It has always been a high-level language in that it frees the programmer from having to perform low-level tasks such representing triangles in the rasterizer, or rendering them to the window. There is no doubt its ease of use has made graphics programming much more accessible over the years.

Under the hood, OpenGL normally focuses on the drawing of graphics into the frame memory buffers and also the reading of the back values already stored in the frame buffer (Guha, 2014). It is very unique in the sense that its designs support the drawing of the 3D geometry such as polygons, lines and the points which are collectively referred to as PRIMITIVES, and also the drawing of the bitmaps and images. The commands in the OpenGL are processed in the same order as they are received by the inner state machine, but the completion of the commands in some cases can be delayed as a result of the intermediate processes causing the buffering of the OpenGL commands. The execution of out-of-order OpenGL commands is always not permitted (Guha, 2014). Also the execution of in-order normally applicable to queries frame buffer and state read operations. These commands only return results in consistency to the complete execution of the preceding commands. The interpretation of the data which is passed to any command of OpenGL is only possible once it is copied in the memory of OpenGL at times when it is needed. Any successive change to the data on this application does not affect the data as long as it has been stored by the OpenGL (Dobersberger, 2015).

2.2 The Origin of Vulkan

Vulkan was initially unveiled by the Khronos Group in March 2015, and version 1.0 was finally released in January 2016. I was present for the initial presentation at the Game Developer's Conference in San Francisco, and I was able to participate in a Q & A with Piers Daniell, Nvidia software engineer and GPU expert. As mentioned before, Khronos is a non-profit industry consortium founded fifteen years ago by some of the biggest names in the graphics industry, including AMD, Nvidia, Intel, and Sun Microsystems. Even if you haven't heard of Khronos, you've most likely heard of some of their industry standard APIs, such as: OpenGL, OpenGL ES, WebGL, OpenCL, SPIR, and glTF.

Unlike its predecessors, Vulkan is designed from the ground up to run on diverse platforms, ranging from mobiles and tablets, to gaming consoles and high-end desktops. The underlying design of the API is layered, or should we say modular, so it enables the creation of a common, yet extensible architecture for code validation, debugging, and profiling, without impacting performance. According the Piers Daniell , Khronos claims the layered approach will deliver a lot more flexibility, catalyze strong innovation in cross-vendor GPU tools, and provide more direct GPU control demanded by sophisticated game engines.

What is so special about Vulkan is that it is the only multi-platform solution, meaning it is envisioned as an API for the masses, from kids gaming on smartphones, to their parents designing buildings and games on workstations.

In theory, Vulkan could be used in parallel computing hardware, to control tens of billions of GPU cores, in tiny wearables and toy drones, in 3D printers, cars, VR kits, and just about anything else with a compatible GPU inside.

2.3 Overview of AMD's Mantle

To fully understand the origin of Vulkan, it's important to know of the API Mantle, which sparked the revolution for a lower-level graphics programming (Keckler 8). Mantle refers to the API developed by AMD which provides the tools for low-level rendering functionality for the developer. It began as an alternative to Direct3D and OpenGL. Mantle was first announced in 2013 at a press conference providing DICE with a console like rendering API for reducing CPU overhead in games (Dobersberger, 2015). AMD has constantly been working in conjunction with other game developers in providing them with the Mantle API. However, in 2015, AMD announced that there would be no more public SDK thus availing the API to selected partners. This action left developers with the option of focusing on Vulkan API or DirectX 12 which were later specified the same year. Additionally, the Mantle API was the foundation for the Vulkan API that operates for numerous hardware platforms and vendors. AMD supports the open and cross platform Vulkan API by providing parts of the Mantle API like reducing driver overhead and power consumption. Mantle also supports multiple core CPUs and features such as split frame rendering. Since most real time rendering applications require addressing almost similar features of graphics accelerator, APIs were developed to provide for portability of the applications allowing sending of commands to various graphic devices (William, 2008). Microsoft's strategies for switching to a universal windows platform are largely documented and DirectX 12 is on the forefront of this shift. Development of DirectX 12 is aimed at improving game performance, releasing creativity and maximizing code sharing between pc and Xbox. In D3D11, all things were finely grained and fundamentally all states were orthogonal objects, for example the rasterizers, pixel shader and the assembler (fabrication) state. This approach nevertheless, did not couple well with contemporary and recent hardware as present graphics cards face the tendency of merging certain operations or handling them differently to what direct 11 is compatible and agreeable with. This clearly indicates that the drivers will not determine things until states are complete, leading to bigger overhead and therefore lesser draw calls (William, 2008).

Vulkan and DirectX 12 seek to mend this by applying Pipeline State Objects (PSO), whereby they are provided on a distinctive thread and assembled in real time. According to Mark and William (# 21), the entire graphics pipeline is integrated and thus the drivers and hardware translate the Pipeline State Objects into whatever the state is expected. The greatest advantage of Microsoft's updated API is essentially gaining dominance over the Graphical Processing Unit (Dobersberger, 2015). Application and game developers are capable of controlling how the GPU's resources are allocated. DirectX 12 significantly changes things over the previous guarded, therefore enabling it to aim or target multiple GPUs as well as providing vast performance boosts. It is impossible for developers and programmers to make use of multiple sets (array) of threads to supply these GPUs as they share individualistic and independent memory location. That implies that each can be allocated its own assets depending on what is required from them, while still sharing the same information once the programmers decide too by establishing up special and unique memory space. Thus far, multiple GPU layouts can be managed two ways within DX 12, creating the curiosity how developers will opt to harness the functionality. It is evidently clear that should the new techniques and technologies manifest to be well accepted among developers and programmers, they will offer extremely compelling justifications to consider while pursuing multiple Graphical Processing Unit setups, especially where high resolution and Virtual Reality gaming are the goals. Connected (Linked) to the final application seems like an individual GPU, but programmers may generate their own queues for every processor. Additionally, each graphical processing unit can possess local assets and resources while at the same time the card can both access and read the memory from a different card (Dobersberger, 2015).

On the other hand Vulkan is a modern cross platform graphics and compute API currently in development by the Khronos group. Vulkan which is a low overhead, close-to-metal API for 3 dimensional graphics and compute applications is fundamentally a different approach to graphics than OpenGL (Khronos, 2015). Vulkan is expected to provide several advantages over all other GPU APIs, facilitating higher ranking cross sectional platform support, appropriate support for multiple threaded processors, lower CPU loading, and Operating System agnosticism. Moreover, it should make easier the development of drivers allowing the pre compilation of drivers comprising the application of shaders written or coded in diverse languages (Lindholm, 2008). Unlike its predecessor, Vulkan is designed to run on several platforms traversing from tablets and mobiles, high-end desktops to gaming consoles. In order for a low-level API to be successful, it should be layered to enhance the creation of ordinary, but yet an extensible architecture or framework for code validation, profiling and debugging, without affecting performance. Vulkan permits applications to get closer-to-hardware, therefore eliminating the necessity for a huge quantity of memory and error/inaccuracy management, in addition to shading language sources (Dobersberger, 2015). Figure 2.1 provides an example for how draw calls are issued to the GPU via the classic OpenGL model. Looking closely to both Vulkan and DirectX 12 APIs, it is evident that both are moving towards providing developers with better performance by utilizing the CPUs and GPUs, therefore meaning Mantle is essentially their spiritual predecessor.

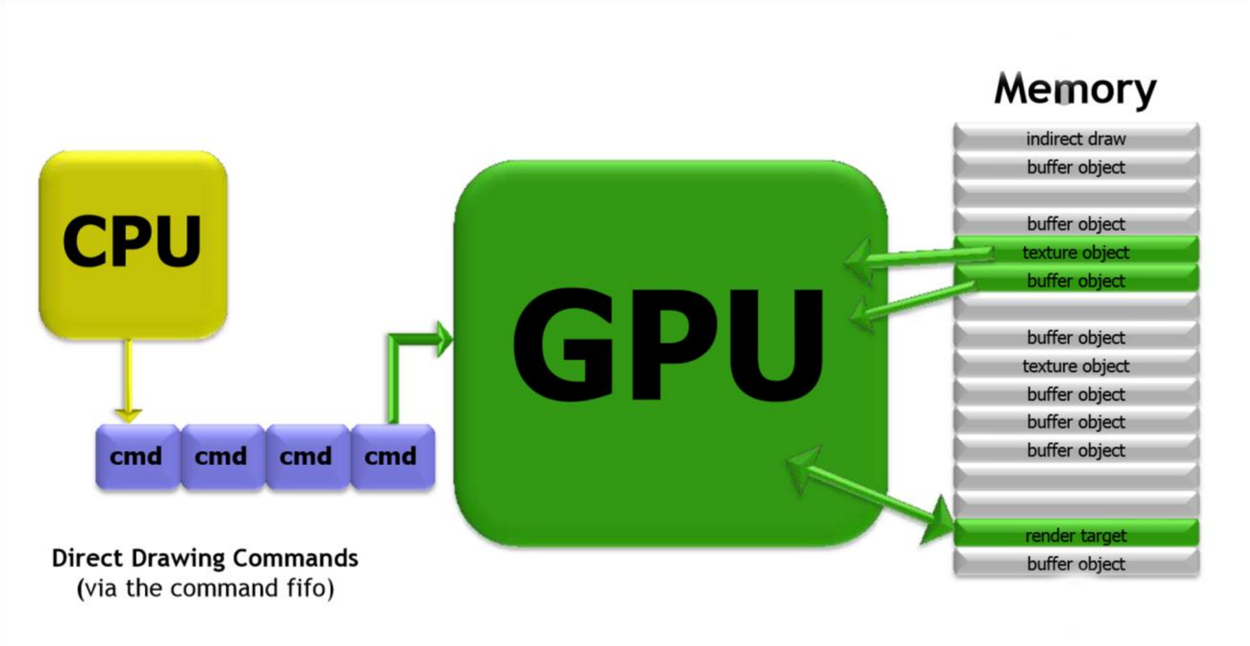


Figure 2.1 Classic OpenGL Model (Khronos Group, 22)

2.4 OpenGL's Limitations

Over two years ago the Khronos group launched the AZDO (Approaching Zero Driver Overhead) initiative. They were given the task of creating a modern graphics and compute API that addressed OpenGL's biggest issues, including:

- Creating a common API for all platforms (desktop, console, mobile & embedded)
- Mapping closely to modern GPU architectures
- Providing a console-like, low-overhead, explicit API

Advances in OpenGL, such as the AZDO initiative and the VK_KHR_no_error extension, demonstrated that it was possible for a lot of the CPU overhead usually associated with the API to be avoided. Creating a brand new OpenGL focused on those ideas could have given developers a decent performance boost. However, this route has not been without issues. For example:

- Buffer allocation and synchronization would still be done by an opaque driver. Stalling and buffer ghosting would still occur in vendor specific ways
- States would still be global. Developers would need a clear understanding of how each IHV's GPU state is set to ensure OpenGL state is configured efficiently
- Efficient multi-threaded command submission would have been tricky to solve in an OpenGL-like API. Figure 2.2 shows a recent comparison of multi-threading capabilities on the CPU between Vulkan and OpenGL 4.1

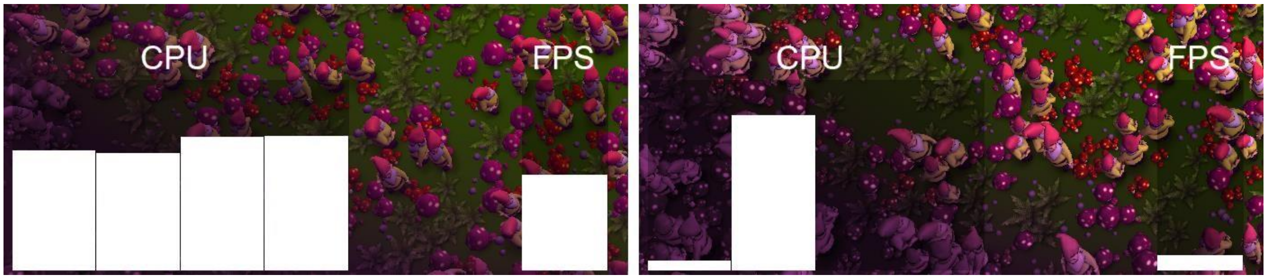


Figure 2.2 Vulkan vs OpenGL: Multi-threading

Source: Imagination Technologies (Smith, 2015)

2.5 Vulkan Relevance to Developers

According to Piers Daniell, hardware manufacturers seem to agree that providing an API where memory allocations, state setting, and call validation are developer controlled will enable better run-time performance while making driver behavior much more predictable across platforms. This should drastically reduce maintenance costs (Daniell, 2015).

However, software vendors appear to be less certain. Nvidia and AMD have spent years optimizing their OpenGL and DirectX drivers. Approaching equivalent or better performance will require software developers to invest significant resources to designing and optimizing their rendering engines for explicit APIs, for example efficiently caching state. Additionally, existing middleware will need to continue supporting older APIs for a long time. For now, Vulkan will be yet another standard to implement and support, which will increase the total cost of maintaining rendering engine code.

The restrictions that affect the big engines creates an interesting opportunity for the development of new middleware that can be more flexible. Figure 2.3 shows Protostar, a Vulkan demo built using Unreal Engine 4. It has been discussed among engine developers (Unity, Unreal, etc.) that today the only way to remain on the cutting edge of graphics APIs is by starting from a clean slate. It could take a long time before the big engines approach the same efficiency.

Figure 2.3 Protostar: Vulkan-Supported Unreal Engine Demo (Smith, 2015)

So one of the fundamental questions is who will this API be most relevant to? Engines will definitely support it, but those who have to support old APIs may be slow to evolve. It’s likely that projects with the simplest use cases will be the first to benefit. 2D rendering engines that spend a lot of time on screen could support the API to reduce power consumption.

Developers targeting closed systems will also have a lot to gain in the short-term from the lower overhead offered by the API. For example, programmers working on in-car navigation systems with tight CPU, GPU and memory budgets.

2.6 Quality of Development Tools

According the Khronos Group's presentation at SIGGRAPH 2015 (Khronos, 2015), software vendors do agree that the quality of development tools can have a more significant impact on an API's success than the API itself. For example, software engineers often single out Sony's platforms and APIs first as they have (according to Daniell) the industry's best tools. OpenGL development has historically been quite difficult. There are now a wealth of utilities maintained by platform owners, IHVs, ISVs and the community, but it has taken years to reach this stage.

With the advent of Vulkan, the Khronos group has put more thought into developing tools than with previous standards. The loader layer design, for example, enables developers to use Khronos supplied debugging layers, those provided by 3rd parties or custom solutions. Unfortunately, Vulkan – like OpenGL – lacks canonical utilities for many debugging tasks, such as analyzing and replaying call streams. Unless the Khronos creates working groups specifically for tools in parallel to the design of new APIs, it's unlikely this will change.

Ever since Vulkan's release, there are already a number of tools available and many more on the way developed by Khronos. Platform owners such as Google and Valve are planning to provide cross-vendor tools for their operating systems. Useful community maintained tools, such as RenderDoc, have allowed developers to debug Vulkan across a variety of operating systems and GPUs.

It is still unclear however if fully accurate GPU performance timing will be exposed through Vulkan. Unfortunately it doesn’t seem to have hasn’t happened yet, but it will hopefully be addressed through an extension in the future. Good tools are obviously essential for Vulkan to succeed. Hopefully we will see better tools which enable you to find a more legible debugging workflow to suit necessary requirements.

2.7 Optimizing Graphics Performance

Today, GPUs in mobile devices are continuously becoming increasingly powerful. The greatest concern in the mobile space is the lifespan of the battery and one of the most significant consumers of battery include the external memory access (Lindholm, 2008). Modern mobile games apply post-processing implementations (effects) in numerous ways and while the Graphical Processing Unit itself is able of performing this, the bandwidth obtainable to the GPU is typically not. Demand for impressive graphics in mobile applications has given rise to ever-more cogent and powerful GPUs and current graphics APIs like Vulkan. Through the use of these advancements, programmers are capable of writing cleaner, well-articulated and more efficient implementations of computer graphics algorithms than ever before. Mantle was a relatively young industry-backed API that aimed at providing functional portability across systems equipped with computational accelerators such as Graphical Processing Units. A standard or benchmark conforming Mantle program can be operated on standard conforming/matching Mantle implementation. However, the issue of performance portability is not addressed by AMD’s Mantle API. Therefore, transformation of a multithreaded graphics program in order to achieve higher performance on a device might have certainly lead to lower performance on other devices, since performance may significantly depend on low level details namely; data layout and space mapping. In addition to popularity of certain

GPU architectures, some GPUs' optimizations have become hallmarks of GPU computing (Dally, 2010).

2.8 GPU Accelerated Computing

GPU accelerated computing refers to the application of a graphics processing unit (GPU) in conjunction with a Central Processing Unit (CPU) to accelerate analytics, scientific, engineering, enterprise and consumer applications. GPU accelerators were pioneered by NVIDIA in 2007 have gained popularity in powering energy efficient and effective datacenters like universities, government labs, enterprises and businesses internationally. In a wide scope, GPUs are hastening applications in stages or platforms varying from tablets, mobile phones, cars, robots and drones. GPU accelerated computing therefore, offers unprecedented application performance and operation by offloading computer intensive segments of the applications to the GPU, while the prevailing code still executes on the CPU. From both users' and developers' perspective, applications simply execute or run significantly faster (Dally, 2010).

Most CPU's are comprised of two to eight cores optimized for sequentially serial processing whereas the GPU possesses a massively parallel structure or architecture composed of thousands of tiny and efficient cores designed for controlling multiple tasks concurrently (Dally, 2010). GPUs have several thousands of cores to manage parallel workloads effectively and efficiently. GPU computing is made possible today because the majority of today's advanced GPUs do much more than render graphics

2.9 SPIR-V - Transforming the Shader Ecosystem

One of the key features of Vulkan is the ability to setup a new pipeline for graphics. Usually the pipeline for 3D graphics involves the compiling of shaders. In Vulkan the only way to pass in a shader is by way of SPIR-V. SPIR-V is a new standard defined by Khronos as an "intermediate shader representation". It is a portable representation which can be used across all platforms. It will support a wide variety of high-level shader languages including GLSL. One of the recurring complaints about OpenGL is the inconsistent compiling of GLSL shaders between implementations. In addition to GLSL, Vulkan can support OpenCL C kernels, as well as basic shader in C++. Future domain-specific languages, frameworks and tools are another option.

Khronos even mentions the possibility of developing new experimental languages.

It's too early on to say whether or not Vulkan and SPIR-V will become industry standards, but the idea of immediate portability across a multitude of different devices is appealing to any ISV. Since there will be no need for every hardware platform to feature a high-level language translator, developers will deal with less of them. An individual ISV can generate SPIR-V using a single tool set, thus eliminating portability issues of the high-level language. SPIR-V is simpler than a typical high-level language, making implementation and processing easier (Lorach, 2014). Performance will be improved in a number of ways, depending on how Vulkan is implemented:

- No more compiler front-end, a lot of processing can be done offline
- Optimization passes can settle faster, optimizations executed offline
- Multiple source shaders reduce to the same intermediate language instruction stream

2.10 How Does Vulkan Compare to OpenGL?

Table 2.1 Vulkan vs. OpenGL

OpenGL	Vulkan
Originally created for graphics workstations with direct renderers, split memory.	A better match for modern platforms, including mobile platforms with unified memory and tiled rendering support.
Driver handles state validation, dependency tracking, error checking. This may limit and randomize performance.	The application has direct and predictable control over the GPU via an explicit API.
Obsolete threading model does not allow generation of graphics commands in parallel to command execution.	API designed for multi-core, multi-thread platforms. Multiple command buffers can be created in parallel.
API choices can be complex, syntax evolved over twenty years.	Removal of legacy requirements simplifies API design, simplifies usage guidance, and reduces specification size.
Shader language compiler is a part of the driver, and it only supports GLSL. The shader source has to be shipped.	SPIR-V is the new compiler target, enabling front-end language flexibility and reliability.
Developers have to take into account implementation variability between vendors.	Due to the simpler API and common language front-ends, more rigorous testing will increase cross-vendor compatibility.

Source: Nvidia (Daniell, 2015)

2.11 Pros and Cons

Potential Benefits of next generation API's

- Better use of multi-core CPUs
- More draw calls = more on-screen detail
- Highers min/mix/avg framerates
- More efficient use of GPU hardware
- More efficient use of integrated (CPU) hardware
- Reduced system power draw
- Allows for new architecture designs previously considered impossible due to technical limitations of past OpenGL/DirectX APIs

There are a number of attributes that should make Vulkan and SPIR-V popular amongst programmers, and Khronos is often iterating over these features. The idea of using the same skills and toolset to build for various platforms is appealing, especially now that the hardware performance gap between different devices is closing.

According to the Khronos Group, the following advantages are made possible by SPIR-V:

- The front-end compiler remains the same for developers across multiple platforms. This eliminate portability issues across ISV's

- Runtime shader compiler time will always be minimal because the driver only has to process SPIR-V

- Developers receive an additional level of IP protection, because there is no need to distribute shader source code

- Drivers are simpler and more reliable since there is no need to include front-end compilers
- Developers can issue command buffers and can tweak memory allocation accordingly to better fit their application's approach.

One example of a developer taking advantage of this is Imagination Technologies who displayed one of the most impressive tech demos portraying improved graphical performance (Figure 2.4). In a presentation at SIGGRAPH, they explained how Vulkan was used to batch draw calls into a block of memory they called "tiles", and the renderer was able to draw multiple "tiles" at a time, drastically increasing the amount of draw calls. When the frame is drawn, depending where the "tile" it can be moved or transformed. On the other hand, in OpenGL, all draw calls are dynamic, meaning they are submitted with each frame no matter what is in the field of view. It is not possible to reuse or cache draw calls which are already executed.

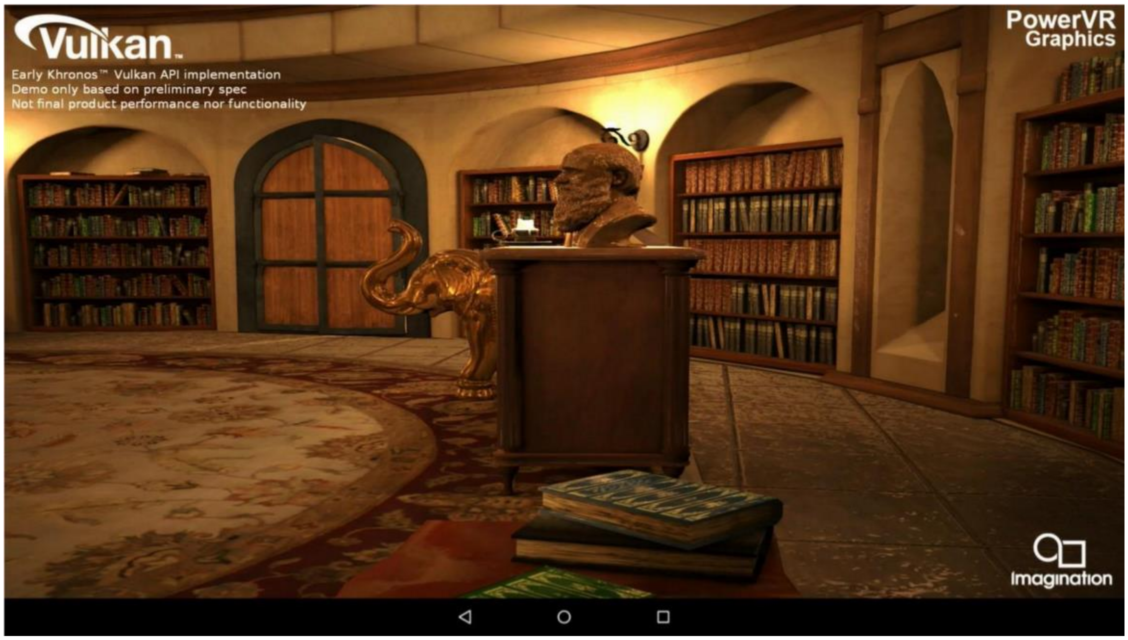

Vulkan™
Early Khronos™ Vulkan API implementation
Demo only based on preliminary spec
Not final product performance nor functionality
PowerVR
Graphics
Imagination

Figure 2.4 Imagination Technologies Vulkan Tech Demo (Smith, 2015)

Because of this, several calls must be issued into kernel mode in order to validate or alter the state of the driver. In Vulkan, pre-processed command buffers are in place to deduce CPU overhead and there is never a need to compile during a render loop.

Another major benefit is *parallel buffer generation*, which enables Vulkan to harness the power across all CPU cores. OpenGL was conceived before processor contained multiple cores.

Over the past 4-5 years, the industry has gone from dual cores to quad cores to even 8 or 10 CPU cores. It just makes sense for a graphics API to have to ability to maximize a CPU's potential if necessary.

Unfortunately, there are a few cons to this approach, some of which are rather significant. For those developers eager to jump on the lower-level API bandwagon, it would be wise to consider these:

- Verbose code and increased complexity
- Still early in its lifespan
- Level of industry support

- Vulkan may not be as relevant or effective on some platforms (desktops)

Until Software leaders are able to simplify the process of allocating memory for parallel processing in Vulkan, developers will have to write numerous lines of additional code. In chapter 3 you can see why it takes over 500 lines of code just to write a simple “Hello Triangle” in Vulkan and C++. In order to take advantage of the great features Vulkan offers, the programmer must take an in depth look at the function calls and their uses. For example, to even begin drawing to a window first the display must be created, memory must be allocated for resources and the 3D pipeline (shaders, state machine), and then command buffers must be initialized (Lorach, 2014).

Needless to say, if a developer wants to use Vulkan to its full potential it will entail a fair amount of code setup work. It is likely that over time this process will be made easier. But currently the API is only at version 1.0 meaning the widespread market has yet to even adopt using it commercially. As of March 2016 there is only a few games which are running on Vulkan and most of those had to use a Beta version.

As more software vendors see the performance benefits, industry support should not be an issue; After all, this is a Khronos standard. Mobile software and hardware evolve more quickly, and it may take a few more quarters before we see Vulkan making an impact on desktop platforms. While Vulkan can do wonders in a CPU-bound setting, especially with multi-core mobile SoCs, these performance gains will be limited on desktop platforms (Lorach, 2014). Desktops handle multi-core processors with a greater level of efficiency, and most graphically demanding applications are GPU-bound.

2.12 Summary

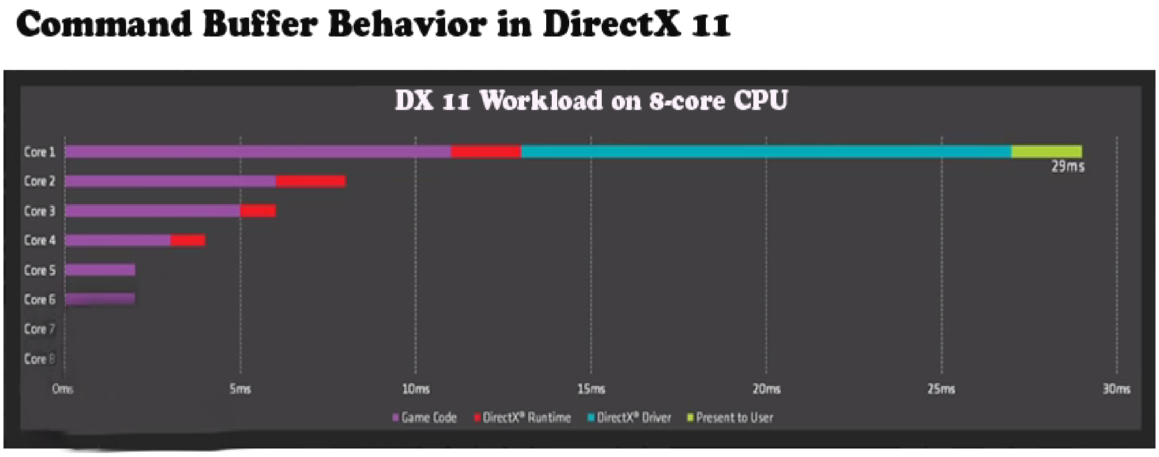
In this chapter, the practicality of Vulkan is discussed and the pros and cons of transitioning to the new API are considered. SPIR-V, the shader language handler is introduced and discussed as well. There are definite benefits to be seen in Vulkan over OpenGL but is it worth the extra work involved?

CHAPTER 3

METHODOLOGY

This chapter presents theories and methods used to examine the abilities of DirectX 12 and Vulkan. The test methodology used for performance testing is further explained and the data in question is presented here. Also in this chapter, we make an effort to obtain an accurate assessment of the relationships between the current API (OpenGL) and the newer APIs in question (DirectX 12, Vulkan).

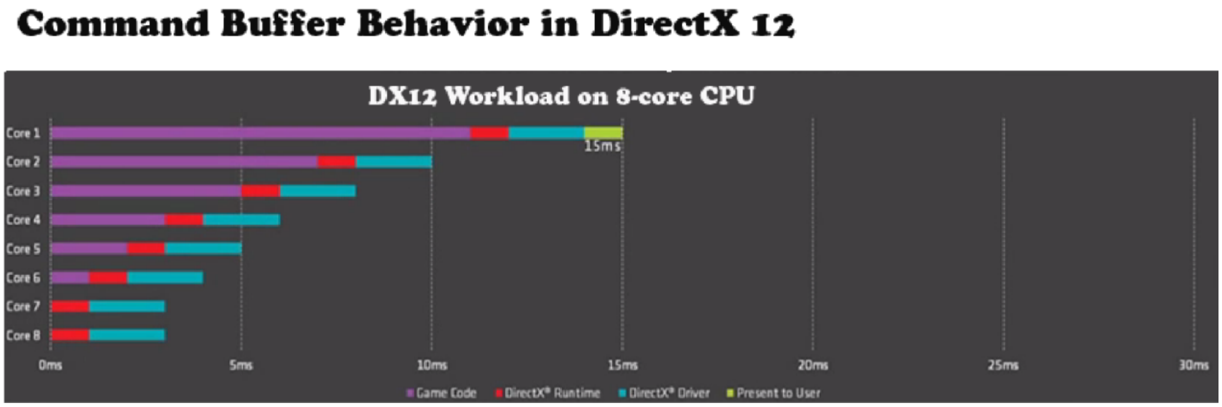
3.1 CPU Overhead Reduction



- Frame rendered in 29ms
- 29ms = 34 frames per second
- Cores 7 and 8 unused
- Core 1 overloaded with most of the work
- DirectX work(red/blue) consumes disproportionate time
- This is “high” API overhead

Figure 3.1 Command Buffers in DirectX 11 (AMD)

Multithreaded graphics in OpenGL & DX11 do not allow for multiple tasks to be scheduled simultaneously without adding considerable complexity to the design. In theory, this means that a great number of GPU resources are spending time idling with no task to process because the command stream simply can't keep up. This in turn means that GPUs running on DirectX 11 or OpenGL will never be fully utilized. This leaves a deep well of untapped performance and potential that programmers should be able to reach (Moammer, 2015). In order to get the most out of GPU performance, we will implement simple C++ programs for rendering images in Vulkan & DirectX 12. The difference in CPU work vs. GPU work will be documented. The data in question is shown in Figures 3.1 and 3.2. This data was gathered from an AMD processor using a program called Geekbench which is a widely used CPU benchmarking tool (Moammer, 2015).



- ➢ Frame rendered in 15 ms
- ➢ 15ms = **66 frames per second**
- ➢ All 8 cores utilized
- ➢ API work (red and blue) very modest vs. code

This is low API overhead

Figure 3.2 Command Buffers in DirectX 12 (AMD)

3.2 Benchmarking Test Methodology

3.2.1 Ashes of Singularity

As stated before, DX12's approach as a new graphics API is very similar to that of Vulkan. Because it is tough to find any games or application which fully utilized Vulkan, for our benchmark test we use a game that was built in DX 12 as well as in DX 11. Ashes of Singularity is a real-time strategy computer game which boasts the first game released with full DirectX 12 support. In chapter 5 (Figure 5.3) you can see the results of a Fraps framerate benchmark test performed on both Nvidia and AMD advanced graphics processors. For diverse results, we tested two separate advanced GPUs at 3 different resolution settings (1080p, 1440p, 2160p). The separate GPUs used in this test were AMD's R9 290X and Nvidia's Geforce GTX 970ti.

3.2.2 Talos Principle Test

One of the first games to offer Vulkan support was the The Talos Principle from Croteam (Stephen W. Keckler). This provided a good opportunity for a simple benchmark test for applications running on Vulkan compared to its OpenGL supported counterpart. For this testbed we used an Nvidia 970ti GPU along with the beloved Windows 8.1 Operating System. This OS showcases the game on a platform that is unable to support DirectX 12. The main goal here was to witness the difference between an OpenGL version of a game next to its cloned Vulkan version. The test results are presented and analyzed in chapter five.

3.2.3 CPU Diagnostics Test

In effort to answer the first research question we conducted a diagnostics test across the CPU (Intel i7-4790k). The tool used to perform the test was the CPU diagnostics test from Microsoft Visual Studio. It analyzes the program during run-time and estimates what percentage of the program is utilizing the CPU and its multiple cores. This test examined two 3D applications. One of the applications tested was a simple 3D game built in OpenGL using C++ for a recent project in my Computer Graphics class. The other application is a rudimentary 3D scene built in Vulkan using C++. The test result for this experiment helped to magnify the extreme difference in driver overhead between the 2 APIs.

The Computer used for the following tests has the following characteristics.

Table 3.1 Benchmark PC Specs

Motherboard	Asus Z97-A
CPU	Intel Core i7 4790k 3.6 GHz Quad Core
RAM	4x Kingston DDR3 – 1600 8GB
GPU	Nvidia GeForce GTX 970 4GB
OS	Windows 8.1

Multi-threaded GPU theories

- **Submitting tasks from multiple queues may add complexity to scheduling**
- **Some GPUs can only process one command stream at a time**
- **This could possibly make it difficult to keep GPU resources fully utilized**
- **Lack of prioritization for more urgent tasks**

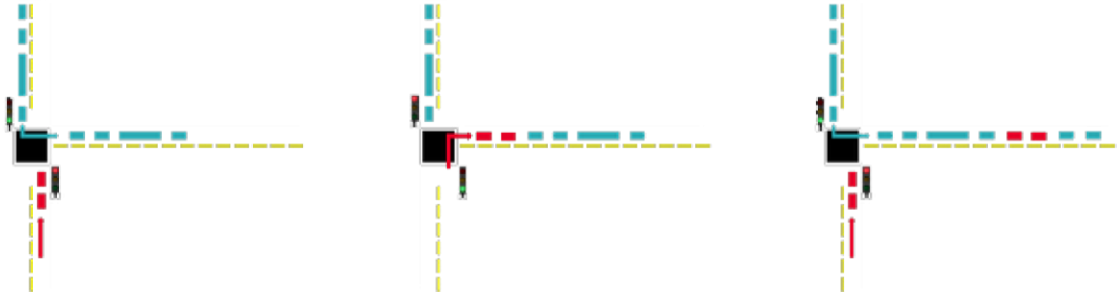


Figure 3.3 Traffic Light Analogy

In the past, other technologies have attempted to improve the idle GPU-space situation by enabling prioritization of certain tasks over others. Graphics pre-emption allowed for prioritizing tasks but just like multithreaded graphics in DX11 it did not solve the fundamental problem. As it could not enable multiple tasks to be handled and submitted simultaneously independently of one another. A crude analogy for this theory, would be that what graphics pre-emption does is merely add a traffic light (Figure 3.3) to the road rather than add an additional lane (Moammer, 2015).

3.2.4 Draw Call Overhead Test

In an OpenGL renderer, when drawing a numerous amount of dynamic objects to the screen simultaneously, I have witnessed performance bottleneck which in turn can cause a screen tearing effect.

In order to test the difference in draw call overhead between the two APIs, we tested our basic Vulkan renderer (Appendix C, page 77), against our OpenGL renderer. The Vulkan GPU numbers are measured as the difference between a `vkCmdWriteTimestamp` at the start and at the end of the render command buffer. For a draw call measurement in OpenGL we used `glBeginQuery/glEndQuery(GL_TIME_ELAPSED)` which pairs around the render loop. This does not entirely account for GPU idles when it is starved between draw calls, but it is how we measured draw calls for this particular test.

3.3 Explicit GPU Control

The new Vulkan interface is designed to be as close to the architecture of modern GPUs as possible. This means that both the code size and the amount of work going on in user and kernel space for the Vulkan driver is very small and therefore will be more efficient than OpenGL. For example, there are no glUniform*() equivalent entry points in Vulkan; instead, writing to GPU memory is the only way to pass data to shaders.

When you call glUniform*(), the OpenGL driver typically needs to allocate a driver managed buffer and copy data to it, the management of which incurs CPU overhead. In Vulkan, you simply map the memory address and write to that memory location directly (Boudier, 2015).

Figure 3.4 presents a chart showing the difference in CPU usage between Vulkan and OpenGL for The Talos Principle demo.

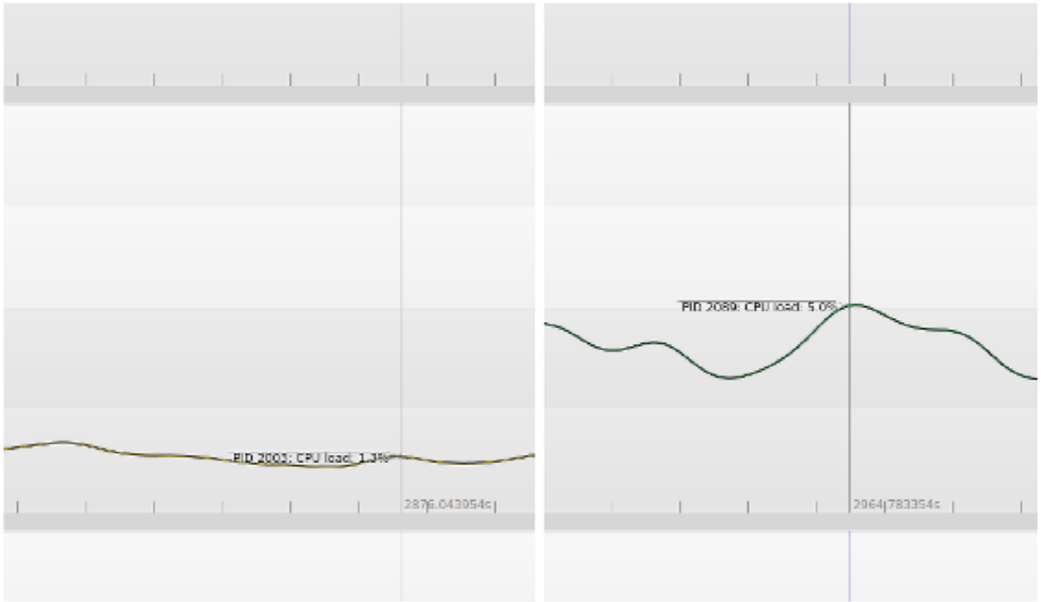


Figure 3.4 CPU Usage: Vulkan vs OpenGL

3.3.1 Leaner, More Explicit Driver

On the front end, driver overhead is greatly decreased in comparison to OpenGL due to the API being designed around hardware. This is where the opportunity arises to deliver more draw calls and hardware vendors can achieve better stability and faster driver boot up time.

In Vulkan, higher level control of the GPU needs to be accomplished by the application. The driver is essentially non-existent and basically does what the code tells it to do. While this may result in more verbose application code, the benefit therein lies in not having to work *around* the driver (e.g. shader pre-processing in OpenGL). Vulkan is clearly designed to resemble modern command buffer-based APIs. Low-level software engineers and engine programmers should see immediate speed-up benefits due to ease of portability and a familiar programming interface. Over time Vulkan should provide more efficient renderers for game engines, 3d modeling software, image editors, etc.

Another major advantage for this API is that less CPU-confined jobs take place when a draw call is issued. Any CPU-related issues when making a large amount of draw calls will initially be taken care of.

However the main benefit here for developer is that programming 3D graphics will become much more predictable. For example: when you call glBlendFunc() in OpenGL, different things can happen depending on the underlying graphics architecture that is running that code. Some GPUs could delay setting up the blending until the first time the bound shader is used; others might not. This makes achieving consistent performance across different GPU vendors quite difficult (Boudier, 2015).

Vulkan should make solving this problem easier because the entry points to the API are designed to allow the driver to do work in consistent places. When you fill in a struct describing some state using Vulkan, you know that there is no driver work going on; the code is all application code. The API is designed to fit as best as it can to all GPU vendor's architectures so there are fewer opportunities for unknown performance hiccups. The glBlendFunc() problem becomes obsolete because the blend function is specified in a struct during pipeline setup. The driver work will happen early, when the function to create the pipeline is called, instead of sometime during rendering causing a stutter.

Actually, a lot of the Vulkan API is aimed at being able to specify everything up-front if possible. For example you can record a list of render commands and state setting commands into a *command buffer* and replay that every frame with just one call. The driver has more opportunities to optimize this usage case because it knows it can do more work when creating the command buffer, rather than when executing it. Another consequence of the explicit nature of Vulkan is that there is no resource renaming (or *ghosting*) behind the application's back – multibuffering needs to be performed explicitly. Multi-buffering is the process whereby a graphics driver may have a number of frames being processed at the same time.

The data attached to those frames (e.g. uniform data and attached textures) needs to be kept around until the frame it is bound to is complete; this will need to be performed by the application. On the plus side, the data that you know will not be modified between frames (e.g. brightness or contrast) can be specified as *const* for possible optimizations (Boudier, 2015).

Vulkan reimagines how well a graphics program can utilize hardware by using something called *render pass*. This implicitly decreases the amount of work needed to be done without the application knowing about it. A render pass consists of a framebuffer state (as opposed to a specific render memory address) which loads render targets in and out of the GPU at the front and back of each render. This structure is the key feature that allows graphically intense application to run at extremely high efficiency on advanced graphics architectures (e.g. Kepler, Maxwell). According to PowerVR, this functionality has already been a major factor for solving latency problems for VR-compatible software.

In OpenGL during rendering, several things can cause implicit flushes of tile buffers to main memory; a bandwidth heavy operation that's usually unnecessary. In Vulkan, the only time such a flush can happen is between render passes, making it obvious to both the application and the driver. More importantly – it tells the GPU exactly what an application wants to do with each render target. Command buffers can be created on a different thread to the thread they are submitted on. This means rendering commands could be created on all cores of a CPU (Smith, 2015). There is no extra work or locking required to do this – a feature that was not previously possible with OpenGL. This may be of use to games which need to recreate their render commands quite often (e.g. Minecraft).

Vulkan gives you the advantage of knowing exactly the state that you are setting. Take for example the glActiveTexture() function in OpenGL: it is not obvious whether this function will change the state globally for all shaders or maybe change the state just for the current shader program. In Vulkan, this is explicitly defined: you know that when you bind your resources, it is changing the state for the bound command buffer because that is the first parameter to the function.

A common pattern in Vulkan is to set the initial parameter to all entry points as a representation of the state which you are changing in the following function call. For example:

```
vkCmdBindDescriptorSet(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
textureDescriptorSet[0], 0);
vkQueueSubmit(graphicsQueue, 1, &commandBuffer, 0, 0, fence);
vkMapMemory(staticUniformBufferMemory, 0, (void **)&data);
// ... vkUnmapMemory(staticUniformBufferMemory);
```

3.2.2 Explicit Memory Management

When you call glTexStorage() in OpenGL, the driver automatically has to allocate memory for a one or two-dimensional texture array. The method and the allocation of memory take place behind the curtain.

In Vulkan however, the memory allocation is done by the application. This means that the application knows more about what type of memory it is using and more importantly how much memory it is using, which should be useful for applications that are memory-bound. This is in contrast to receiving an “out of memory” error in OpenGL and needing to reduce resource usage by an unknown value. Explicit memory management in Vulkan allows applications to use custom allocation strategies. For example to allocate all memory up-front and avoid any allocations during rendering (Boudier, 2015).

3.3 Synchronization - Fence, Semaphore, Event

Outsourcing overhead between the GPU and CPU is one of the key features found in Vulkan. This is handled using three synchronization tools known in Vulkan as fences, semaphores, and events.

Semaphores and Fences are very similar synchronization tools with a few key differences. Fences are used for synchronization from GPU to CPU, while semaphores are for syncs within the GPU only. Semaphores are specifically used to sync queue submissions, either the same queue or a separate one. Fences can only be waited on by the CPU, while semaphores can only be waited upon by the GPU. Both fences and semaphores can be signaled from the GPU. Another difference between the two is fences must be reset manually, while semaphores will reset automatically after being waited on.

An event type is much more general. They can be set, reset and checked by both the CPU and GPU, however they can only be waited on by the GPU. Events are limited to only working within a single queue. Events differ from fences and semaphores especially, because they can be used for syncing within a command buffer.

3.4 Loading Vulkan Functions

It is clear that applications cannot yet fully rely on the Vulkan library because even after version 1.0 release in January, the library has been updated on 4 occasions. Even when it is included in an official Android API level, applications that want to run on earlier platform versions (e.g. with a fallback to OpenGL ES) are not able to directly link against the API. In both cases, applications need to load Vulkan dynamically, and handle the possibility that it might not be present:

```
void* vulkan_so = dlopen("libvulkan.so", RTLD_NOW | RTLD_LOCAL);
if (!vulkan_so) {
    LOGD("Vulkan not available: %s", dlerror());
return false;
}
```

Function pointers for the global Vulkan commands (those that do not take a dispatchable object as their first parameter) and `vkGetInstanceProcAddr` had to be loaded dynamically (Hajdarbegovic, 2015):

```
PFN_vkEnumerateInstanceExtensionProperties
vkEnumerateInstanceExtensionProperties =
        reinterpret_cast<PFN_vkEnumerateInstanceExtensionProperties>(
dlsym(vulkan_so, "vkEnumerateInstanceExtensionProperties"));

PFN_vkEnumerateInstanceLayerProperties vkEnumerateInstanceLayerProperties =
reinterpret_cast<PFN_vkEnumerateInstanceLayerProperties>(        dlsym(vulkan_so,
"vkEnumerateInstanceLayerProperties"));

PFN_vkCreateInstance vkCreateInstance =
reinterpret_cast<PFN_vkCreateInstance>(
        dlsym(vulkan_so, "vkCreateInstance"));

PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr =
reinterpret_cast<PFN_vkGetInstanceProcAddr>(
dlsym(vulkan_so, "vkGetInstanceProcAddr"));
```

All other Vulkan commands and the commands in the VK_KHR_swapchain and VK_KHR_device_swapchain extensions can be obtained in the same way; function pointers obtained this way can be used with any Vulkan instance. Alternately, vkGetDeviceProcAddr and any commands that take VkInstance or VkPhysicalDevice as their first parameter can be obtained by calling vkGetInstanceProcAddr. The function pointers returned are specific to the instance used to retrieve them, and avoid a dispatch indirection. Similarly, commands that take a VkDevice, VkQueue, or VkCommandBuffer as their first parameter (except vkGetDeviceProcAddr) can be obtained from vkGetDeviceProcAddr, are specific to a particular device, and avoid a dispatch indirection (Hajdarbegovic, 2015).

> **Note:** This alternative process reflects a beta version of the Vulkan code base; the Windows implementation should now have updated to the finally required behavior. I was able to use `dlsym` to obtain `vkGetInstanceProcAddr`, and through that obtain function pointers for all other core and extension commands. A call to `vkGetDeviceProcAddr` was able to return device specific function pointers which help to avoid dispatch overhead.

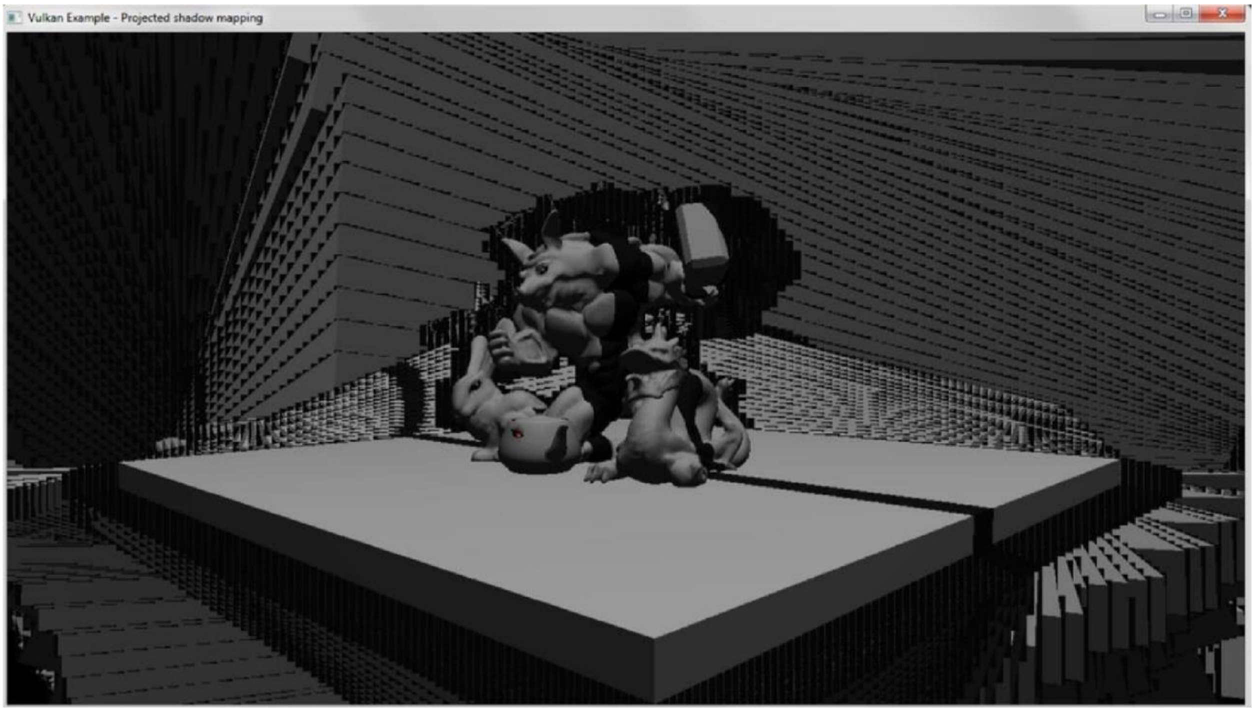


Figure 3.5 Vulkan AMD Error

3.4.1 Window System Integration

For my Vulkan examples I used the VK_surface and VK_swapchain extensions to allow Vulkan to render to onscreen windows represented by Windows win32 API. Figure 3.5 is a screenshot of a common error which occurred when using an AMD GPU. The functionality of VK_swapchain seems to clash with AMD's Fiji architecture.

In future versions of Vulkan there should be no need to call any SetupWindow() functions on the window directly when using Vulkan; all queries and configuration can be done through the VkSurface object and VkSwapchain creation. The vkCreateSurface function in the VK_surface extension is used to create the VkSurface from an ShowWindow(). Surface properties and swapchain creation have some platform specific behaviors on Windows:

- VkSurfacePropertiesKHR::currentExtent is the default size of the window; a swapchain with this size will not be scaled during presentation.
- VkSwapchainCreateInfoKHR::minImageCount should be set to 3 for best performance on current Android devices when attempting to render at the display refresh rate.
- If VkSwapchainCreateInfoKHR::imageExtent is not the same as VkSurfacePropertiesKHR::currentExtent, the swapchain images will be scaled to the window size during presentation. The scaling filter is not specified, but is bilinear or

  better. If the image and surface aspect ratios are different, images will be scaled non-uniformly rather than letterboxed.
- On Windows there is no performance advantage to setting VkSwapchainCreateInfoKHR::clipped to VK_TRUE , though there may be on other platforms.

3.5 Summary

This chapter was a presentation of the overall methodological approach for investigating Vulkan and DirectX 12. This approach fits our research design because we have meticulously analyzed the contrast between OpenGL and Vulkan. The user or developer should have a decent understanding of the high level concepts involving the Vulkan programming process. Additionally some key differences were noted about Vulkan’s CPU advantages over OpenGL.

CHAPTER 4

IMPLEMENTATION

This chapter demonstrates and briefly exhibits what it is like to begin writing code in Vulkan. The brief demonstration consists of writing a simple working program which creates a pipeline and draws a triangle. Furthermore, in section 4.2 we present several high-performance rendering techniques which are often used today for high performance graphics rendering. These algorithms were written solely using C++ and Vulkan. The methods of implementation are discussed and results are analyzed.

4.1 Vulkan PsuedoCode Overview

One of the issues with previous APIs like OpenGL and DirectX 11 is that they are not a good abstraction of modern graphics hardware, resulting in unnecessarily complex drivers that sort-of guess what the software actually wants to achieve with its given architecture. A few years ago when AMD began working on Mantle, the main goal was to alleviate the bottleneck of a graphics driver by offering a low-level API and a simplified driver that doesn't get in the way. As stated before, Vulkan is essentially an evolution of Mantle. Sinc the SDK for Vulkan has been released (early 2016), it has become more evident that similar implementations and method calls of its predecessor Mantle have carried over with a few changes in its naming conventions. After viewing a few demos and taking a look at Vulkan's programming guide (Daniell, 2015), you can manage to write a "Hello Triangle" demo in C++. The experiment ended up calling for over 500 lines of code. Some past experience with DirectX or OpenGL will really help to understand the code.

The first task a graphics programmer must do is load the function entry points from the .dll. You set up a helper function that initializes function pointers, much like when using GLEW or OpenGL. These helper functions can be instantiated in a header file (i.e. Vulkan.h). In the past, Mantle functions used a *gr* prefix. Many Vulkan methods have the same name while simply replacing the *gr* with *vk*.

Vulkan is initialized by calling `vkInitAndEnumerateGpus()` with some information that describes your application such as the version, the engine name, or the application name.

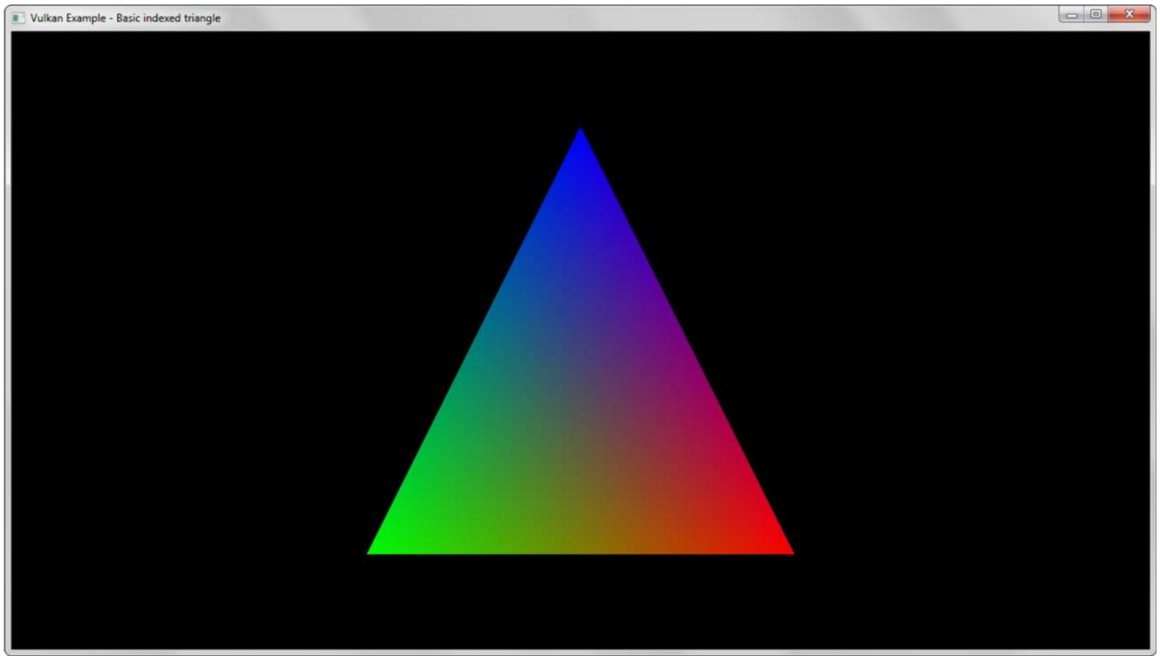


Figure 4.1 Vulkan Basic Triangle

See Appendix A for my C++ source code implementations. Also the complete source code for this triangle is at: *github.com/jshiraef/VulkanTester* (See *VulkanExampleBase.cpp & triangle.cpp*)

While sticking to the high level concepts, Figure 4.2 is a simple flowchart which presents the steps necessary to draw this triangle in Vulkan. First a display must be created to enable some sort of window presentation. This is done by using Vulkan’s Window System Integration (WSI). WSI is an extension which ties the Vulkan display data to whatever windowing systems that is provided by the current operating system. The next step is to setup your resources. For drawing a simple triangle the only resources needed are vertex data and an index buffer (Appendix A p. 76). In C++ this is taken care of by placing basic vertex data within a struct and allocating memory (`VkMemoryAllocateInfo` & `VkMemoryRequirements`). All memory in Vulkan is automatically handled via the asynchronous queue.

4.1.1 Pipeline State Objects

Finally it’s time to setup the 3D pipeline. To understand the 3D pipe you must instantiate the pipeline state object (PSO). The pipeline state object consists of shaders and separate 3D states. This is where SPIR-V, the translator for all shader languages, is applied and compiled into machine code. The PSO is the object which represents all static states in the entire 3D pipeline (Shaders, vertex data, rasterization, colors, depth, etc). A PSO can be initialized by instantiating a `vkPipelineLayoutCreateInfo` object (See *triangle.cpp*). A key feature of the pipeline state object is that all shader data and state data are compiled ahead of drawtime. The data is pre-baked into the PSO.

Another unique feature of the pipeline state object is that they can be cached for reuse if necessary, and they also can contain multiple entry points to be called-on as needed. As multiple threads are launched and properly synchronized in Vulkan, the reusability of PSOs and other resources should bring about a significant timing advantage for advanced GPUs.

There is also dynamic state which contains data the can be changed very quickly. These changes will not affect the pipeline state. (This data can consist of viewport transformations, color blending, polygon offsets, stencil masks, etc).

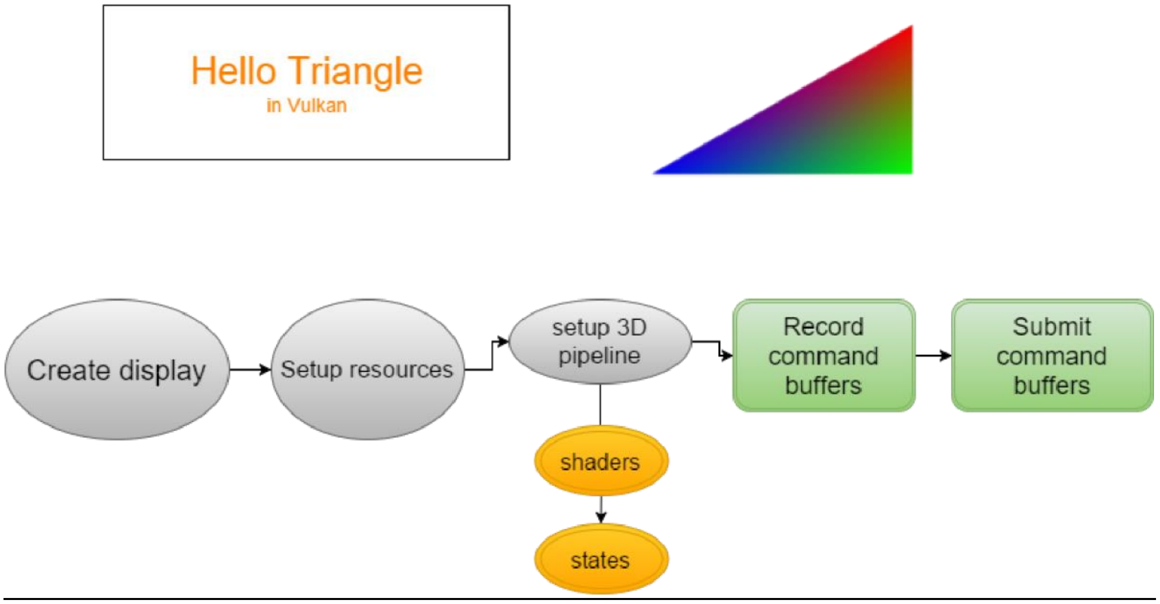


Figure 4.2 Simplified Overview of Graphics Pipeline

4.1.2 Command Buffers

Next the developer must record the commands for rendering a triangle within a designated command buffer. A command buffer will typically consist of starting a render pass, binding your resources (e.g. vertex buffers, pipeline state object, descriptor sets), modifying your dynamic state, draw calls, and ending the render pass (See Figure 4.2 for flowchart visualization). These command buffers are designed to be multithreaded-friendly and recording them is meant to be very fast.

Lastly, a queue submission command buffer must be issued. This is where commands for the GPU are scheduled. Queue execution is set to be cheap for the processor and is always asynchronous. Additionally, `vkSemaphore` can be used to synchronize dependencies between command buffers. The presentation is now ready to be made using the WSI extension.

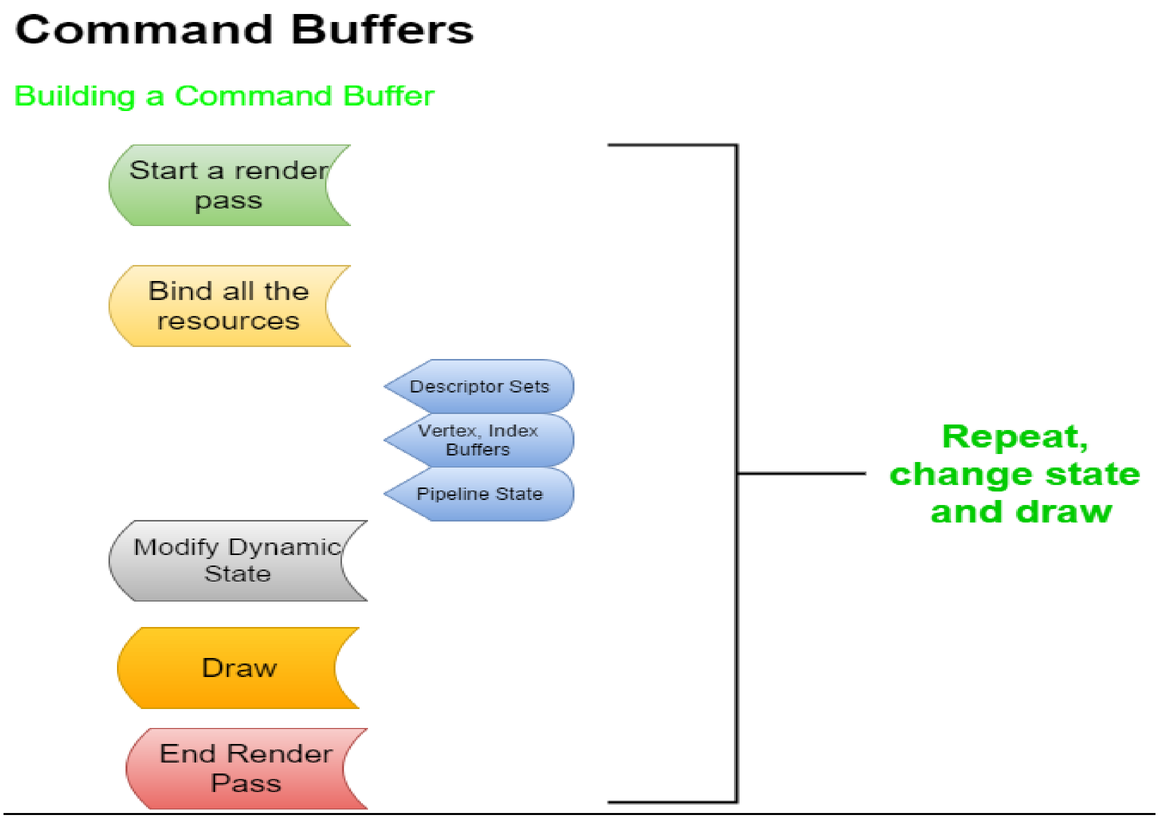


Figure 4.3 Building Command Buffers

4.2 Implementing Vulkan - Advanced Techniques and Algorithms

4.2.1 Global Illumination

During creation of high quality renders, thorough understanding and comprehension of global illumination acts as a vital or pivotal step in that process (Dutre, 2006). Developers must therefore familiarize themselves thoroughly on the operation and execution with global illumination. Global illumination refers to the process which stimulates indirect lighting such as color bleeding and light bouncing. In order to achieve the best of its effect, global illumination wholly relies upon the use of photons in mental ray. Basically, a photon refers to the particle that is accountable for light energy (Dutre, 2006). Once a photon has been emitted, it comes in contact with the surface in the scene inheriting that surface's color as well as the energy value. Upon bouncing up the first surface, the photon carries those energy values or packets to the next surface it comes in contact with, and continuously bounces until its absorption creating an indirect illumination effect. This technique of using global illumination gives a developer the capability of capturing indirect illumination, which demonstrates clearly the real world occurrences where light reflects or bounces off everything in its propagation path it is entirely absorbed. For instance, cracks at the lower part of a door causes light to penetrate into the room or red walls reflecting light from the light source can cause the floor to have a red hue. Therefore, the use of global illumination to achieve these kinds of effects creates a much greater level of believability and realism in the renderer.

During the rendering process with mental ray, developers need to turn on emission of photons for both the render settings and the light source. Failure to complying with this process, indirect lighting effects will not be applied to the render. Typically, every object will cast/throw and receive photons but not all objects really have to both cast and receive photons for the developer to achieve the desired look. While reducing render times or fine tuning the outlook for global illumination, developers must specify precisely which objects ought to cast or receive photons. For example, the developer can arrange and organize only certain particular items within their scene to receive global illumination. A common happening with global illumination is that the renders may come out splotchy and speckled creating a smooth effect. However, this can be relatively easy fix. Increasing the bounces or photon emission level from the light source will create a smoother cleaner indirect lighting effects. Increasing the number of photons emitted therefore will noticeably incline the rendering time (Dutre, 2006).

Global illumination can also be defined as the technique for modeling how light is reflected and bounces off of surfaces. Modeling indirect lighting allows for events that make the virtual or simulated world seem connected and more realistic since objects influence each other's appearance. For a long time, real-time graphics applications and video games were limited only to direct lighting. On the contrary, calculations necessary for indirect lighting were deemed to be slow hence were only applied in non-real-time situations such as computer graphics animated films. Most global illumination algorithms require extra preprocessing before actual rendering which can often slow run-time performance. In Vulkan, this shouldn't be an issue because the pipeline is setup and compiled before run-time preferably on multiple cores. In the past developers have sought for a way to work around the limitation by calculating indirect light for surfaces and objects which are known before time to be stationary. During global illumination, the number of lights, its direction, position and other attributes can be altered while indirect lighting updates accordingly. Likewise, alteration of the material's properties of items (objects) such as color, the amount of light absorption or emitted is made possible (Dutre, 2006).

While pre computed real-time global illumination results to soft shadows, typically they will be more coarse-grained compared to those achieved with Baked Global Illumination (GI) unless the scene be very small (Dutre *et al* 96-134). It should be noted that while pre computation of real-time global illumination does the last lighting at run time, iteratively it is carried out over many frames. Therefore, when a big change is made to the lighting, more frames will be taken for the global illumination to fully take effect. Although global illumination for real time applications is relatively faster, Baked GI would produce a better run-time performance for constrained target platform. The major limitation of Baked Global Illumination and Pre computed Real-time Global Illumination is that static or stationary objects are not includes in pre computation. This implies that objects in motion cannot bounce or reflect light onto adjacent objects and vice versa (Dutre, 2006). The source code for Figure 4.4 can be found in Appendix B.

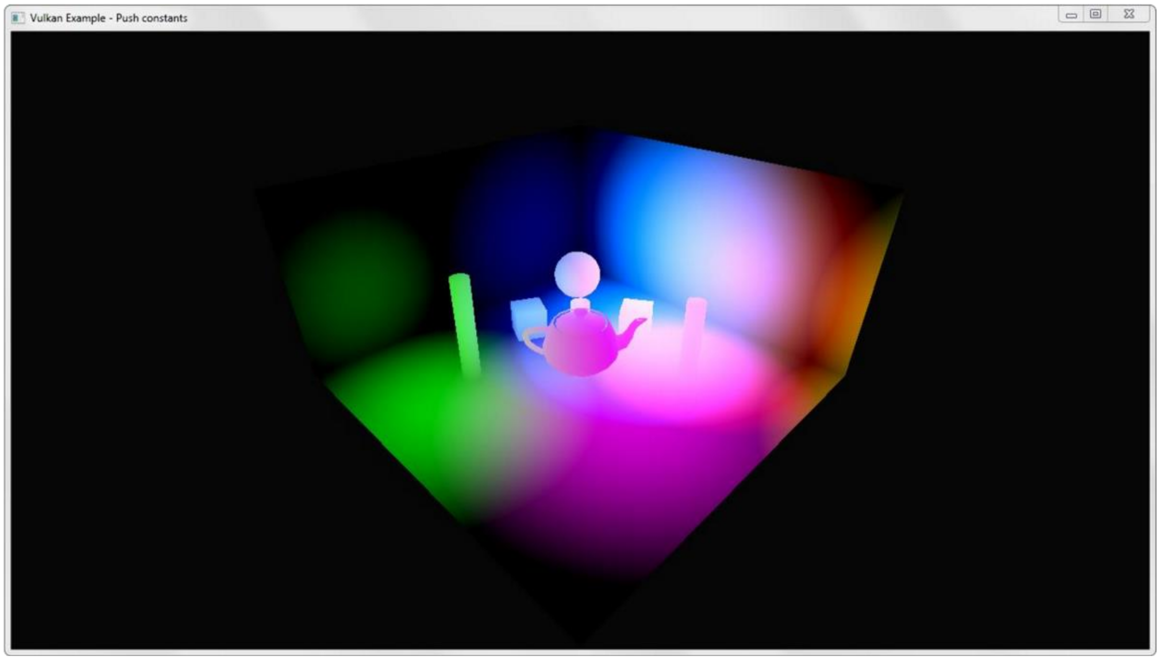


Figure 4.4 Global Illumination Vulkan Example

4.2.2 Deferred Rendering

Also referred to as deferred shading is a technique that differs from forward shading or forward rendering. Deferred rendering postpones light's computation to the image space and to end of the rendering pipeline, similarly posting processing techniques (Lauritzen, 2010). The underlying ideas of this technique imply that if the pixel does not get to the screen then there is no need in shading it. When a pixel gets into the image space the developer knows that it is going to be visible therefore they can calculate safely all the lights' influence on it. According to Lauritzen and Andrew (1-34), shading is made much faster by this concept coupled with the per pixel accuracy. On the contrary, this technique introduces series of issues which forward rendering does not have. For instance, it is quite difficult when dealing with transparencies as handling large numbers of materials requires storage of more information in the q-buffer which can easily blot the video memory when using OpenGL. Deferred shading is commonly comprised of two passes namely geometry pass and lighting pass. Geometry pass involves rendering the scene and retrieving all kinds of geometrical information from the objects that we store in a collection of textures known as the g-buffer. The geometric information of a scene stored in the G-buffer is then later used for more complex lighting calculations (Lauritzen, 2010).

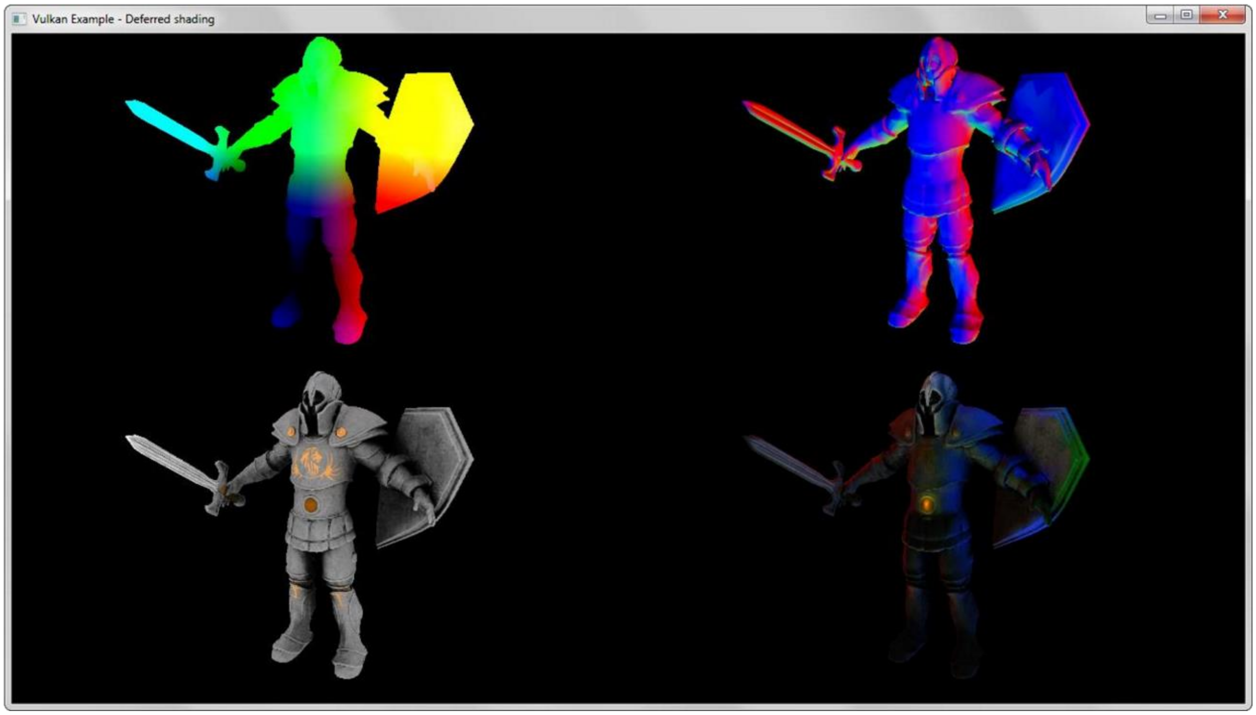


Figure 4.5 Deferred shading

On the other hand, a lighting pass involves use of textures from the G-buffer where a developer renders a screen-filled quad and calculates the scene's lighting for each fragment using stored in the G-buffer; that is, the developer iterates over the G-buffer pixel by pixel. Instead of taking each object all the way from the vertex shader to the fragment shader, developers decouple its advanced fragment process to a later stage. The lighting calculations remain exactly the same but this time all the input variables required are taken from the corresponding G-buffer textures instead of the vertex shader (Lauritzen, 2010). A major advantage of this approach is that whatever fragment ends up in the G-buffer is the actual information that ends up as a screen pixel, as the depth test already concluded this fragment information as the top-most fragment. This ensures that for each pixel, the developer's process in the lighting pass is done once therefore saving them a lot of unused render calls. Furthermore, deferred rendering opens up the possibility for further optimizations that allow us to render a much larger amount of light sources than developers would be able to use with forward rendering. In OpenGL, deferred shading had a few disadvantages. The G-buffer required developers to store a relatively large amount of scene data in its texture color buffers which usually consumes a significant amount of memory, especially since scene data like position vectors require a high precision (Lauritzen, 2010). However when this technique was implemented in Vulkan it was simple to distribute the texture color buffers across multiple cores without stalling memory or framerate (See `buildDeferredCommandBuffer()` & `prepareMultiThreadedRenderer()` in Appendix C, p 77 - 78). Figure 4.5 shows a screenshot of the deferred shading example built in C++ and Vulkan.

4.2.3 Instancing

For advanced real-time computer graphics, it is sometimes necessary to render multiple copies of the exact same mesh repeatedly throughout a scene. For example when creating a detailed environment there may be a need for thousands of trees, blades of grass, or buildings. This can be accomplished through the practice of geometric instancing, which means rendering numerous copies of the same exact mesh in one scene simultaneously. This can be represented using a repeated geometry shader without appearing too repetitive. Each instance can be issued different attributes (e.g. color or position) in effort to decrease the spectacle of repetition. Figure 4.6 a simple example of instancing while rendering the same mesh in C++ and Vulkan with differing uniforms all within one single draw command. This saves significant performance when the same mesh had to be rendered multiple times.

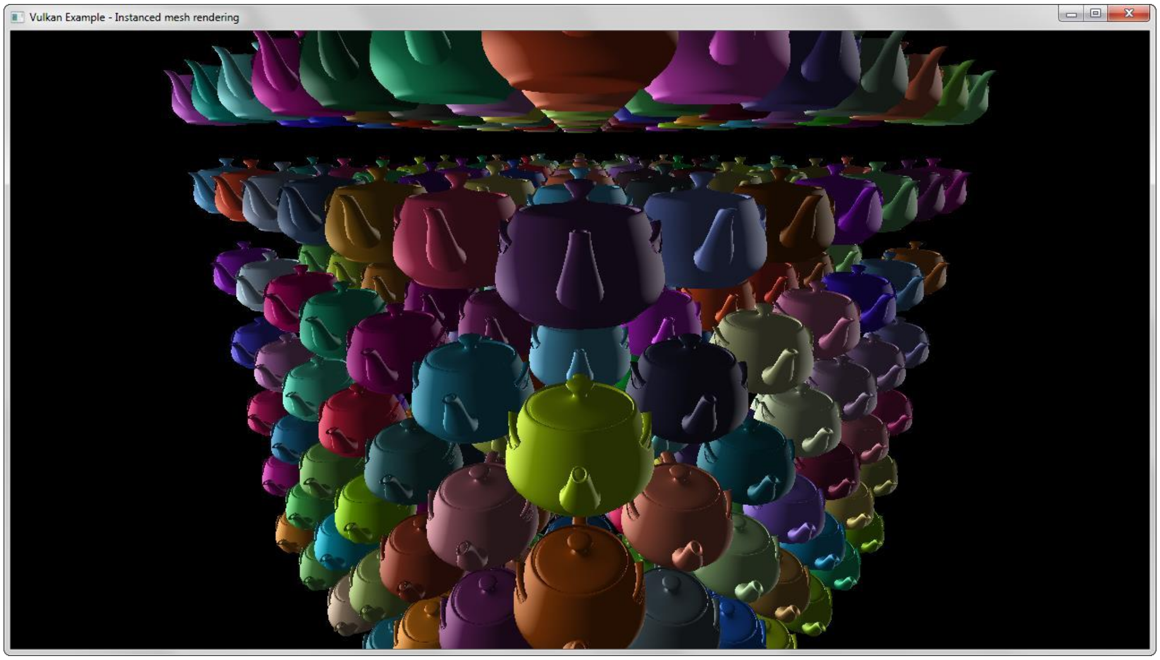


Figure 4.6 Mesh Instancing

See Appendix c page 85 for Instancing source code. The complete source code for this example can be viewed at *github.com/jshiraef/VulkanTester/instancing*

4.2.4 Bump mapping

Bump mapping, also referred to normal mapping, is much like texture mapping. The difference is that texture mapping involves adding color to a polygon, while bump mapping adds what appears to be surface roughness. This process encompasses lighting calculations that lead to introduction of perturbations on the surface of the object. The object going through this process remain unchanged even the whole surface is evenly made bumpy (Dobashi, 2002). There is no modification on the surface geometry of the object in question. An advantage of bump mapping is that it can add minute details to objects which would otherwise require a large number of polygons. Obviously the polygon is still physically flat but appears to be rigid.

4.2.5 Parallax Occlusion Mapping

Parallax Occlusion Mapping refers to a method that is used reducing a geometric representation's complexities by encoding outside/surface detail in a texture. The height-map representation is typically the surface information for replacement of the geometry. When rendering the model, all the details concerning the surface are reconstructed in the pixel shader from the height-map texture information (Tatarchuk, 2006). The primary idea behind parallax occlusion mapping (POM) is relatively simple. Computation of the effects of motion parallax for a surface is done by applying a height map and offsetting each pixel in the height map using the geometric normal and eye vector. As the geometry is moved away from its original or former position using that ray, a parallax is achieved. This technique explains that all the highest points on the height map would be moved the farthest along that ray while the lower extremes would seem to be stationary. To obtain satisfying result or outcomes for true perspective simulation, a developer would have to shift every pixel in the height map using both geometric normal and the view ray. Essentially parallax occlusion mapping is a replicated displacement mapping approach that takes place in texture space (Tatarchuk, 2006).

Parallax mapping in Vulkan can fully utilize the programmable GPU pipeline to offer interactive and responsive rendering rates. Recent algorithms have variety of significant improvements and advancement over the earlier techniques. Application of this method can be used in animation of objects as it fits correctly within established artwork pipelines of computer games as well as effects rendering. Implementation of the current algorithms makes effective and efficient use of current GPU pixel conduits and texturing hardware or equipment for interactive and responsive rendering. This algorithm thus provides scalability for a given range of existing GPU products (McGuire, 2005). Parallax mapping is an enhanced version or model of Normal Mapping in computer graphics which changes the behavior of lighting as well as creating illusion of 3-D details on plain polygons. Parallax mapping hence offsets surface coordinates which are used in assessment of textures with dispersed colors and normals. In implementing parallax mapping, the developer requires a heightmap surface or texture. The height map in every pixel contains statistics about elevation of the surface. The predominant duty of parallax mapping methods involves modifying the texture coordinates in a way that plain surfaces will appear like 3-D.

Parallax Occlusion Mapping simply interpolates between results of Normal Mapping. (See Figure 4.7. For interpolation POM uses depth of the layer after intersection (0.375, where Normal Mapping has stopped), previous $H(T_2)$ and next $H(T_3)$ depths from heightmap. As you can see from the image, the result of Parallax Occlusion Mapping interpolates in on the intersection of view vector V and the line between heights H(T3) and H(T2). Intersection Tp is close enough to real point of intersection (marked with green).

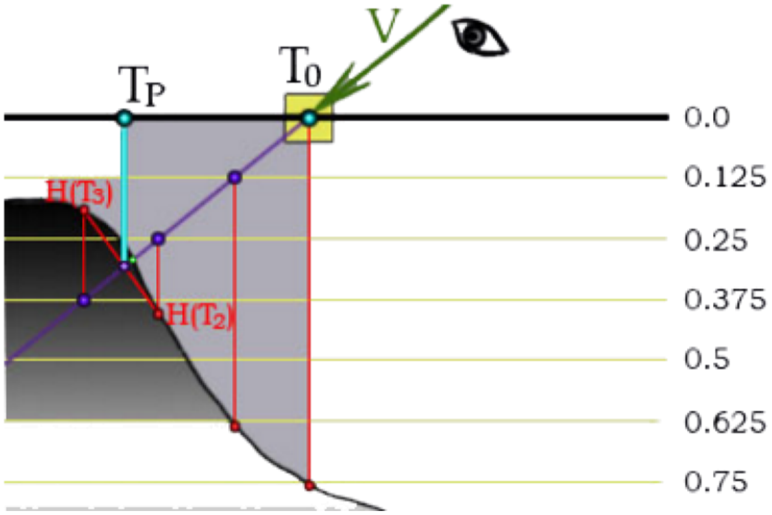


Figure 4.7 Parallax Occlusion Mapping Diagram

Parallax Occlusion Mapping is therefore an alternative advancement for Bump Mapping. POM is aimed at producing better outputs in comparison to Relief Parallax Mapping which provides better and excellent outcomes than Steep Parallax Mapping (McGuire, 2005). However, the POM results are unsatisfactorily worse compared to results for Relief Parallax Mapping or Steep Parallax Mapping. Additionally, Parallax Occlusion Mapping basically interposes between outcomes of normal mapping as well as producing desirably better results with comparatively small quantities of samples/specimen from heightmap. Notwithstanding, Parallax Occlusion Mapping (POM) may skip finer details of the heightmap more than normal mapping producing incorrect outputs for abrupt or sudden changes of variables in the heightmap (Tatarchuk, 2006).

Like normal mapping, parallax mapping simulates geometry on a flat surface. In addition to normal mapping a heightmap is used to offset texture coordinates depending on the viewing angle giving the illusion of added depth.

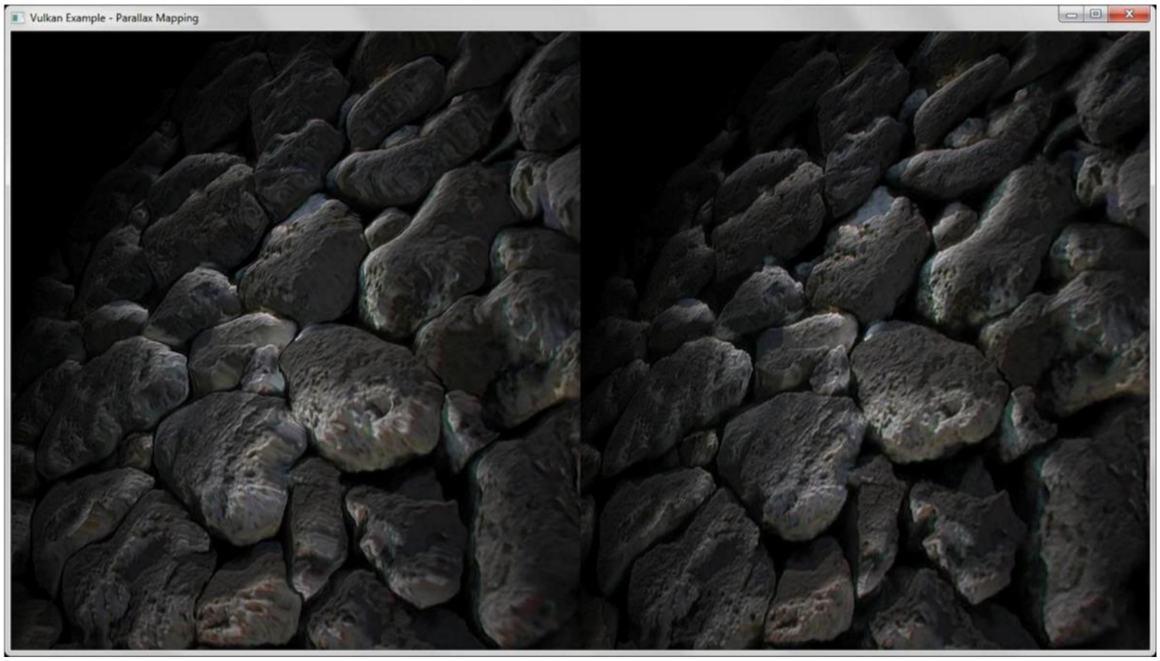
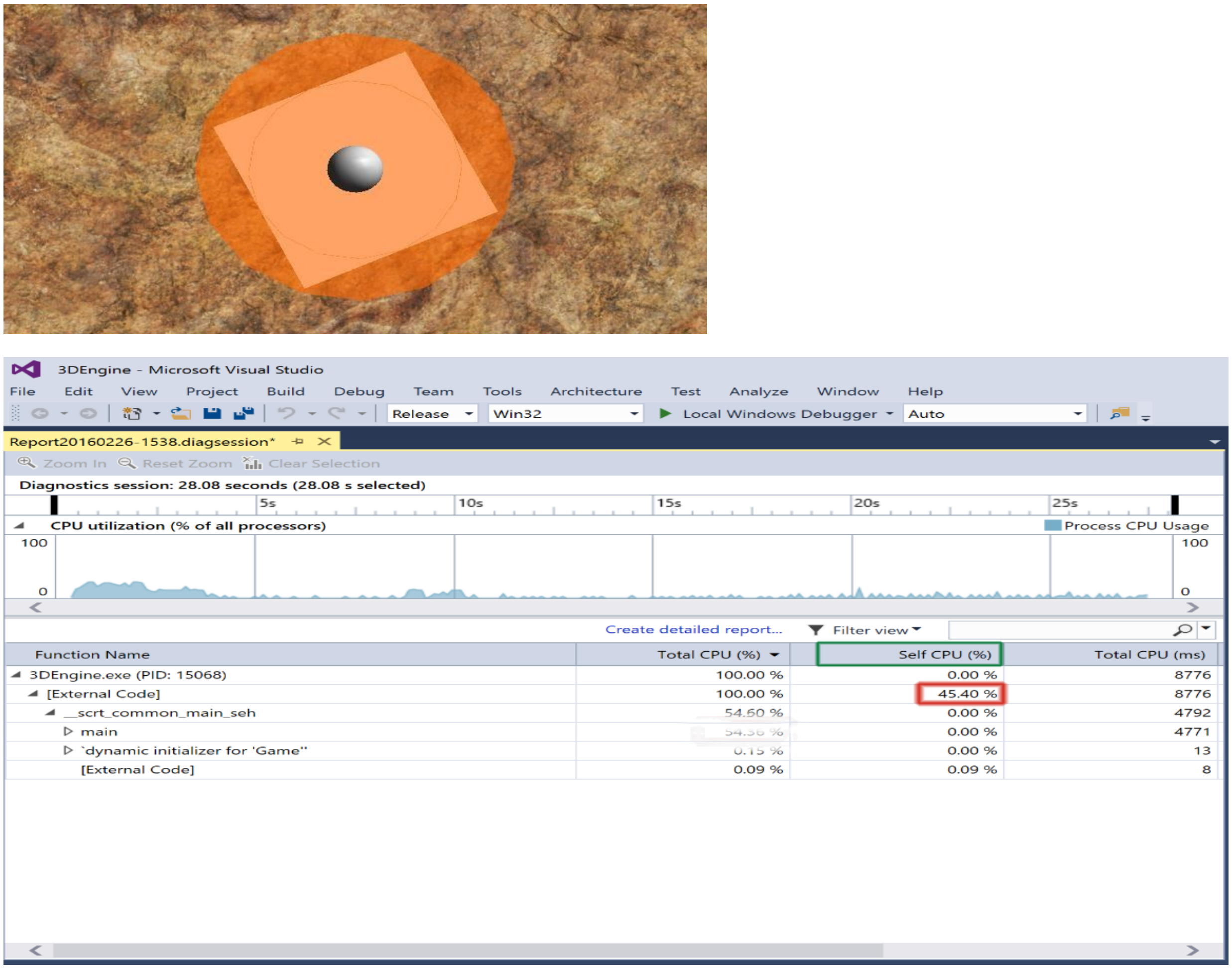


Figure 4.8 Parallax Occlusion Mapping

CHAPTER 5

TEST RESULTS AND ANALYSIS

The data presented in this chapter are the results of diagnostics and performance tests mentioned in chapter 3. These tests were conducted in effort to validate the performance and optimization improvements which can be achieved when transitioning to Vulkan or DirectX 12.

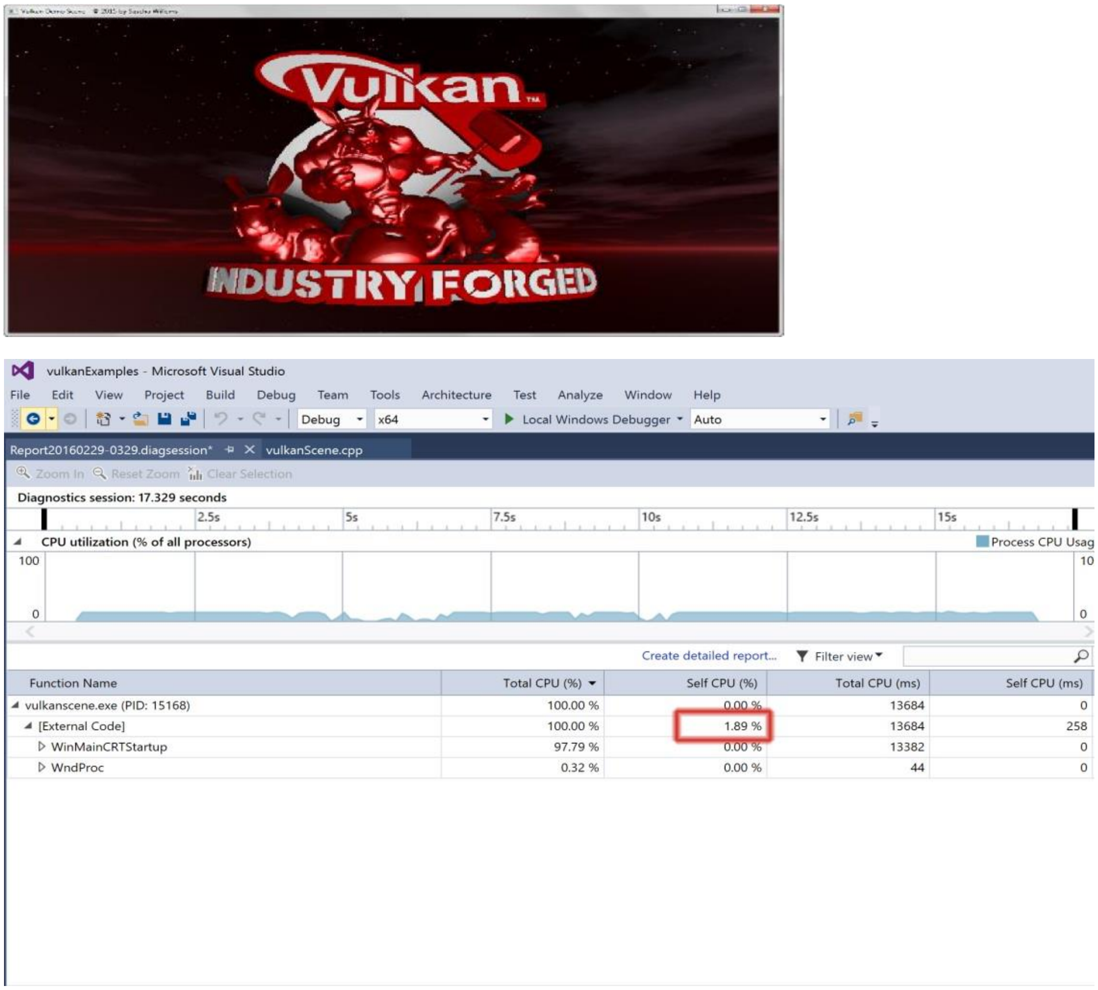

3DEngine - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Release
Win32
Local Windows Debugger
Auto
Report20160226-1538.diagsession*
Zoom In
Reset Zoom
Clear Selection
5s
10s
15s
20s
25s
CPU utilization (% of all processors)
Process CPU Usage
100
0
100
0
Create detailed report...
Filter view

Diagnostics session: 28.08 seconds (28.08 s selected)

Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)
3DEngine.exe (PID: 15068)	100.00 %	0.00 %	8776
[External Code]	100.00 %	45.40 %	8776
__scrt_common_main_seh	54.60 %	0.00 %	4792
main	54.36 %	0.00 %	4771
`dynamic initializer for 'Game''	0.15 %	0.00 %	13
[External Code]	0.09 %	0.09 %	8

Figure 5.1 OpenGL 3DEngine CPU Utilization

These results were pulled from personal screenshots after running a Microsoft Visual Studio CPU diagnostics test. Figure 5.1 shows the percentage of CPU usage while running an OpenGL application which was a simple 3D game I made in C++ and OpenGL as a project for my Computer Graphics class. It is evident that almost half (45%) of the application's code is CPU bound. This is a large amount of overhead and could very easily contribute to CPU bottleneck.



Diagnostics session: 17.329 seconds

Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)
vulkanscene.exe (PID: 15168)	100.00 %	0.00 %	13684	0
[External Code]	100.00 %	1.89 %	13684	258
WinMainCRTStartup	97.79 %	0.00 %	13382	0
WndProc	0.32 %	0.00 %	44	0

Figure 5.2 Vulkan 3D Scene CPU Utilization

On the other hand, when running a simple 3D Scene built in C++ and Vulkan, **less than 2 percent** of the application is using the CPU as shown in Figure 5.2. This was a result of the exact same MSVS diagnostics test running immediately after the previous one. This makes it clear that any Vulkan application will not be CPU bound unless work is specifically allocated and distributed amongst CPU cores. To view my code which implements multi-threading in Vulkan see `prepareMultiThreadedRender()` (Appendix C, page 81) The complete source code for this Vulkan scene can be viewed at *github.com/jshiraef/VulkanTester/VulkanScene*.

5.2 Ashes of Singularity Test Results

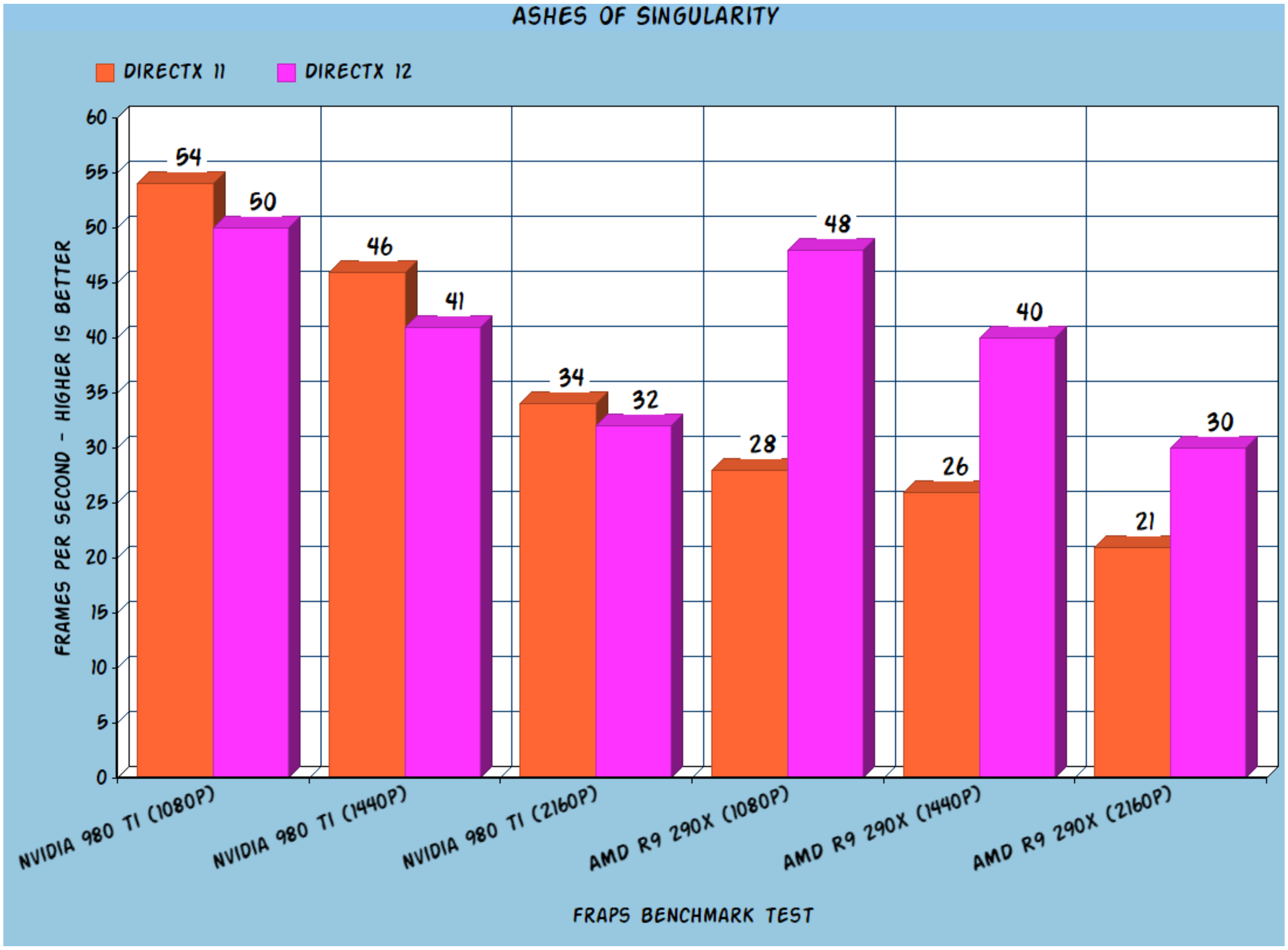


Figure 5.3 Ashes of Singularity Test

It was unexpected to discover that performance for Nvidia drivers using DirectX 12 was actually slightly worse than its performance with DirectX 11. Theoretically, this could be because this new architecture of utilizing CPU overhead is not being fully exploited yet by Nvidia. But because AMD has been working on “Mantle” for so long, its current GPU architecture seems to be ahead of the game for exploiting the increased optimization potential of the new graphics API’s (Vulkan, DirectX 12).

What these test results have indicated, along with other publications, is that AMD GPUs consistently showed significantly greater performance gains than their Nvidia counterparts and in many instances the AMD cards matched or outperformed more expensive Nvidia offerings. As has been the case with many benchmark tests registering a performance loss when Nvidia hardware is running the DX12 version of the benchmark compared to DX11. I have learned from further research that this was down to a hardware feature called Asynchronous Shaders/Compute.

Asynchronous Shaders/Compute, also known as Asynchronous Shading is one of the more useful hardware features in Vulkan and DirectX 12. This feature allows tasks to be submitted and processed by shader units inside GPUs (something Nvidia calls CUDA cores and AMD dubs Stream Processors) simultaneous and asynchronously in a multi-threaded fashion.
One would’ve thought that with multiple thousands of shader units inside modern GPUs that proper multi-threading support would have already existed in DX11. In fact one would argue that comprehensive multi-threading is crucial to maximize performance and minimize latency (Moammer, 4). But the truth is that DX11 only supports basic multi-threading methods that can’t fully take advantage of the thousands of shader units inside modern GPUs. This means that until now it has never really been possible for GPUs to reach their full potential.

5.2.1 Asynchronous Shaders

- **Next generation APIs can process multiple command streams in parallel**

  Enabled by Asynchronous Compute Engines

- **Each queue can submit commands without waiting for other tasks to complete**
- **Independent command streams can be interleaved on the GPU Shaders and execute simultaneously**
  - This increases utilization and performance by filling gaps in the pipeline

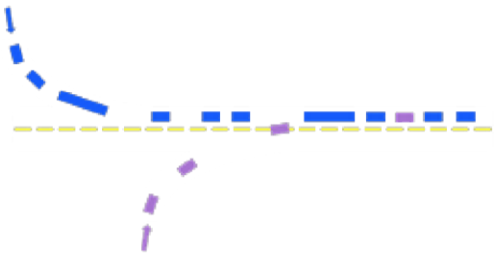


Figure 5.4 Asychronous Shaders Merging

To enable asynchronous shaders, the GPU must be built from the ground up to support it. In AMD’s Graphics Core Next based GPUs this feature is enabled through the Asynchronous Compute Engines integrated into each GPU. These are structures are built directly into the GPU itself. And they serve as the multi-lane highway by which tasks are delivered to the stream processors.

5.3 Talos Principle Results

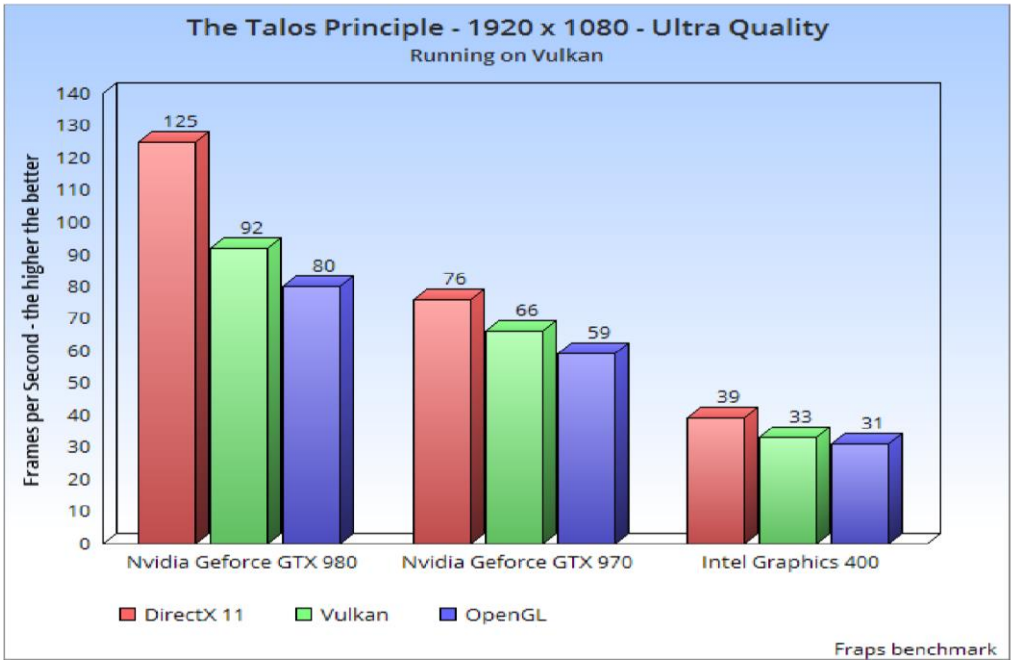


Figure 5.5 Talos Principle Results

To be fair, this is an extremely early look at Vulkan performance; The developers Croteam obviously originally made The Talos Principle to run most efficiently in DirectX 11, which explains the significant gap in increased framerate. Furthermore The Talos Principle is not a title that's designed to exploit the CPU utilization and draw call improvements that are central to Vulkan. As expected, in Figure 5.5 we see that Vulkan's performance here does not in any way stack up to the DirectX 11 version. The interesting takeaway here is that a simple 3D game built by a small team of developers using a beta version of Vulkan managed to gain an overall performance increase over OpenGL. Overall the visual quality did not appear to display any significant differences. The Vulkan version did crash once the first time it was booted up. In the future, it will be very interesting to see what a large AAA team of developers can do with the power of Vulkan and its increased scalability.

5.4 Draw Call Overhead Results

A normal inner drawing loop for OpenGL looks like this:

```
for (Mesh* i : meshList){
   glBufferSubData(...)  // Mesh-specific Uniform Data
   glBindTexture(...)
   glDrawElementsBaseVertex(...)
}
```

and with Vulkan:

```
for (Mesh* i : meshList){
   vkCmdPushConstants(...)  // Mesh-specific Uniform Data
   vkCmdBindDescriptorSets(...)  // Texture
   vkCmdDrawIndexed(...)
}
```

The shaders used here were basically identical and as simple as possible. Both implementations rendered the specified amount of objects, each consisting of 2 draw calls (with different textures) with 44 triangles in total. Here are the results:

Table 5.1 Draw Call Overhead

Object Count	ms/frame Vulkan	ms/frame OpenGL
256	0.57	0.84
1k	0.95	2.50
4k	3.18	8.33
16k	12.94	30.86
64k	47.44	123.62

Impressively, the Vulkan implementation ran about 2 ½ times faster. Especially since the Nvidia OpenGL driver it is competing with is quite well optimized and we are only using a single thread for building the Vulkan command buffer yet.

To gain a better understanding of how the total frame times came together, let's take a look at the CPU time spent in the draw loop and the GPU time used:

Table 5.2 Draw Call Overhead: CPU vs. GPU

Object Count	ms CPU Vulkan	ms CPU OpenGL	ms GPU Vulkan	ms GPU OpenGL
256	0.13	0.56	0.06	0.08
1k	0.48	2.04	0.12	0.42
4k	2.06	7.74	0.55	5.92
16k	8.90	30.41	2.87	28.57
64k	35.99	123.14	10.3	121.13

Apparently Vulkan's timing advantage is actually closer to 3x when looking at CPU time only. GPU time looks very high on the OpenGL side, but that's merely a measuring benchmark since the GPU is idling a lot between draw calls, and it is measured as time between GPU-side execution of first and last draw call. Lastly we switched the low-polygon mesh for a 6.5 triangle version and reran the tests:

Table 5.3 Draw Call Overhead: High Polygon count

Object Count	ms Frame Vulkan	ms Frame OpenGL	ms CPU Vulkan	ms CPU OpenGL	ms GPU Vulkan	ms GPU OpenGL
1k hipoly	71.12	108.92	0.49	2.10	70.25	106.87
1k hipoly	84.95	107.32	0.49	2.09	84.09	105.27
1k hipoly	81.66	108.15	0.49	2.09	80.80	106.10

As you can see, the Vulkan version produced quite some performance variation when it was within the GPU limit. This was not happening during previous tests. Surprisingly, there is almost no frame-to-frame variation, performance only changes when restarting the application. One would have to guess that memory allocation is randomly placing or fragmenting the vertex data or the command buffer in some cases more and in some cases less fortunate areas. It is interesting to see from a very basic render test, Vulkan delivers a better performance here as well. These results were unexpected

5.5 Takeaways

The landscape for computer graphics programming is quite different than it was in the 90's when OpenGL was released. GPUs have been improved unimaginably over the past decade. The industry has called for another advanced API standard, and the Khronos Group has answered with Vulkan. OpenGL will still remain the most popular 3D API for years to come and it will continue to see benefits from future driver optimizations. Developers who want to transition from OpenGL to Vulkan will witness a huge departure from their comfort zone. Switching to multi-core enable programming is a profound paradigm shift, and one which requires the developer to think very differently. Switching to Vulkan or DX12 can certainly offer a runtime benefit for your game or application, because now any jobs can be executed simultaneously. However one must consider the work it will take to overcome issues such as code complexity or interlocking of threads.

While OpenGL's state machine has done wonders for 3D applications it is no longer the most efficient approach to advanced graphics programming. Instead of making function calls to a huge state machine, you will be designing a rendering pipeline to hopefully be compiled and cached at once. This will allow the GPU to fully maximize its potential by optimizing, storing, or accessing whatever it is told to. An immediate benefit which can be exploited when switching to Vulkan is a massive increase in draw calls. However, the only way to truly harness impressive performance gains is to become a master of multi-threading & resource management.

5.5 Summary

This chapter presented several benchmark performance tests. It was discovered that Vulkan and DirectX 12 do provide a performance benefit over OpenGL. A CPU diagnostics test was also analyzed for both Vulkan and OpenGL 3D applications. The results indeed proved that a Vulkan application is not CPU bound in any way. While there are few application which have been completed that fully utilized Vulkan we were able to run a framerate benchmark tests for 2 games. According to the benchmark results, it is clear Vulkan does increase performance and optimization, however currently the margin of improvement is minimal. It is worth noting the applications used in the benchmark tests were built on beta versions of their API's.

CHAPTER 6

CONCLUSION

This thesis has aimed to provide an in-depth look at two new computer graphics API's. It seems the industry standard in the field of computer graphic might be headed down a new path for creating software with advanced graphical capabilities. A few demonstrations of advanced computer graphics techniques were presented, and an attempt was made to present a novel approach for measuring and assessing the API's performance benefits. This analysis should make it easy for a programmer or developer to assess the difficulties of transitioning to Vulkan or DirectX 12. Furthermore, this research and methodology should hopefully clarify the value of these next generation APIs, and spark intrigue for future projects or ideas.

6.1 Future Work

The first idea for future work which comes to mind would be to test the portability of these new API's by converting previous projects built in OpenGL to see how they run in Vulkan. The process of porting code from OpenGL to Vulkan has proven to be time consuming, but will hopefully become a simpler process as more tools are developed and future versions are released.

REFERENCES

Boudier, P., Kubisch, Cristoph (2015). GPU Driven Large Scene Rendering. *Nvidia GPU Technology Conference*, 22- 34.

Dally, J. N. a. W. J. (2010). The GPU Computing Era. *IEEE Micro, 30*(2), 56-69.

Daniell, P. (2015). Vulkan on Nvidia GPUs. *GDC Vault 2015, Nvidia*, 2 – 52.

Dobashi, Y., Tsuyoshi Yamamoto, and Tomoyuki Nishita. (2002). Interactive rendering of atmospheric scattering effects using graphics hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 99-107.

Dobersberger, S. (2015). Reducing Driver Overhead in OpenGL, Direct3D and Mantle. *University of Applied Sciences Technikum Wien*, 2-32

Dutre, P., Philippe Bekaert, and Kavita Bala. (2006). Advanced global illumination. 96-135.

Guha, S., Dr. . (2014). Computer Graphics: Through OpenGL: From Theory to Experiments. 11-51. from http://www.sumantaguha.com/

Hajdarbegovic, N. (2015). A Brief Overview of Vulkan API. *Toptal Engineering*(SIGGRAPH 2015), 6 – 15

Khronos, Group. (2015). Vulkan-Overview. Retrieved from https://www.khronos.org/assets/uploads/developers/library/overview/vulkanoverview.pdf

Lauritzen, A. (2010). Deferred Rendering for Current and Future Rendering Pipelines. *SIGGRAPH Course: Beyond Programmable Shading*(NVIDIA Tesla: A unified graphics and computing architecture.), 39-55.

Lindholm, E. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 39-55.

Lorach, T. (2014). Approaching Zero Driver Overhead. *Nvidia Professional Visualization Group*(SIGGRAPH 2014).

McGuire, M., and Max McGuire. (2005). Steep Parallax Mapping. *I3D 2005 Poster*, 23- 24.

Moammer, K. (2015). AMD Improves DirectX 12 Performance. *WCCF Technologies*(WCCF Hardware & Tech), 3 - 9

Parent, R. (2007). Computer Animation: Algorithms and techniques (2nd Edition).

Smith, A. (2015). Trying out the new Vulkan Graphics API. *Imagination Technologies*(PowerVR Developers 2015), 4 – 10. Retrieved from www.imgtec.com.

Stephen W. Keckler, W. J. D., Brucek Khailany, Michael Garland, David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, *31*(5), 7-17.

Tatarchuk, N. (2006). Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. *Proceedings of the 2006 symposium on Interactive 3D graphics and games*(ACM 2006), 63-69

William, M. (2008). Future graphics architectures. (Queue 6.2), 54-64.

APPENDIX A

DATA STRUCTURES

In this section, the core data structures used for Vulkan and OpenGL rendering techniques are listed in detail. These are references for Chapter’s 3 & 4.

Triangle Data Struct:

```
struct {
        VkBuffer buf;
        VkDeviceMemory mem;
        VkPipelineVertexInputStateCreateInfo vi;
        std::vector<VkVertexInputBindingDescription> bindingDescriptions;
        std::vector<VkVertexInputAttributeDescription> attributeDescriptions;
    } vertices;

    struct {
        int count;
        VkBuffer buf;
        VkDeviceMemory mem;
    } indices;

    struct {
        VkBuffer buffer;
        VkDeviceMemory memory;
        VkDescriptorBufferInfo descriptor;
    }  uniformDataVS;

    struct {
        glm::mat4 projectionMatrix;
        glm::mat4 modelMatrix;
        glm::mat4 viewMatrix;
    } uboVS;

    struct {
        VkPipeline solid;
    } pipelines;

    VkPipelineLayout pipelineLayout;
    VkDescriptorSet descriptorSet;
    VkDescriptorSetLayout descriptorSetLayout;

    VulkanExample() : VulkanExampleBase(ENABLE_VALIDATION)
    {
        width = 1280;
        height = 720;
        zoom = -2.5f;
        title = "Vulkan Example - Basic indexed triangle";
        // Values not set here are initialized in the base class constructor
    }
```

Instancing Struct:

```
struct {
        VkPipelineVertexInputStateCreateInfo inputState;
        std::vector<VkVertexInputBindingDescription> bindingDescriptions;
        std::vector<VkVertexInputAttributeDescription> attributeDescriptions;
    } vertices;

    struct {
        vkMeshLoader::MeshBuffer example;
    } meshes;

    // Number of mesh instances to be rendered
    uint32_t instanceCount;

    struct UboInstanceData{
        // Model matrix for each instance
        glm::mat4 model;
        // Color for each instance
        // vec4 is used for memory alignment
        // GPU aligns at 16 bytes
        glm::vec4 color;
    };

    struct {
        // Global matrices
        struct {
            glm::mat4 projection;
            glm::mat4 view;
        } matrices;
        // Seperate data for each instance
        UboInstanceData *instance;
    } uboVS;

    struct {
        vkTools::UniformData vsScene;
    } uniformData;

    struct {
        VkPipeline solid;
    } pipelines;
```

APPENDIX B

VULKAN FUNCTIONS

Appendix B. Vulkan Functions

These are functions from the Vulkan example of deferred shading (chapter 4.2.2). To view the entire deferred.cpp source code see github.com/jshiraef/VulkanTester

```
void buildDeferredCommandBuffer()
    {
        VkResult err;

        // Create separate command buffer for offscreen
        // rendering
        if (offScreenCmdBuffer == VK_NULL_HANDLE)
        {
            VkCommandBufferAllocateInfo cmd =
vkTools::initializers::commandBufferAllocateInfo(
                cmdPool,
                VK_COMMAND_BUFFER_LEVEL_PRIMARY,
                1);
            VkResult vkRes = vkAllocateCommandBuffers(device, &cmd,
&offScreenCmdBuffer);
            assert(!vkRes);
        }

        VkCommandBufferBeginInfo cmdBufInfo =
vkTools::initializers::commandBufferBeginInfo();

        // Clear values for all attachments written in the fragment sahder
        std::array<VkClearValue,4> clearValues;
        clearValues[0].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
        clearValues[1].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
        clearValues[2].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
        clearValues[3].depthStencil = { 1.0f, 0 };

        VkRenderPassBeginInfo renderPassBeginInfo =
vkTools::initializers::renderPassBeginInfo();
        renderPassBeginInfo.renderPass =  offScreenFrameBuf.renderPass;
        renderPassBeginInfo.framebuffer = offScreenFrameBuf.frameBuffer;
        renderPassBeginInfo.renderArea.extent.width = offScreenFrameBuf.width;
        renderPassBeginInfo.renderArea.extent.height = offScreenFrameBuf.height;
        renderPassBeginInfo.clearValueCount = clearValues.size();
        renderPassBeginInfo.pClearValues = clearValues.data();

        err = vkBeginCommandBuffer(offScreenCmdBuffer, &cmdBufInfo);
        assert(!err);

        vkCmdBeginRenderPass(offScreenCmdBuffer, &renderPassBeginInfo,
VK_SUBPASS_CONTENTS_INLINE);

        VkViewport viewport = vkTools::initializers::viewport(
            (float)offScreenFrameBuf.width,
            (float)offScreenFrameBuf.height,
            0.0f,
            1.0f);
        vkCmdSetViewport(offScreenCmdBuffer, 0, 1, &viewport);

        VkRect2D scissor = vkTools::initializers::rect2D(
            offScreenFrameBuf.width,
            offScreenFrameBuf.height,
            0,
            0);
        vkCmdSetScissor(offScreenCmdBuffer, 0, 1, &scissor);

        vkCmdBindDescriptorSets(offScreenCmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
pipelineLayouts.offscreen, 0, 1, &descriptorSets.offscreen, 0, NULL);
        vkCmdBindPipeline(offScreenCmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
pipelines.offscreen);

        VkDeviceSize offsets[1] = { 0 };
        vkCmdBindVertexBuffers(offScreenCmdBuffer, VERTEX_BUFFER_BIND_ID, 1,
&meshes.example.vertices.buf, offsets);
        vkCmdBindIndexBuffer(offScreenCmdBuffer, meshes.example.indices.buf, 0,
VK_INDEX_TYPE_UINT32);
        vkCmdDrawIndexed(offScreenCmdBuffer, meshes.example.indexCount, 1, 0, 0, 0);

        vkCmdEndRenderPass(offScreenCmdBuffer);

        blit(offScreenFrameBuf.position.image, textureTargets.position.image);
        blit(offScreenFrameBuf.normal.image, textureTargets.normal.image);
        blit(offScreenFrameBuf.albedo.image, textureTargets.albedo.image);

        err = vkEndCommandBuffer(offScreenCmdBuffer);
        assert(!err);
    }
```

Multi-threading Render function :

```
// Create threads and initialize
    void prepareMultiThreadedRenderer()
    {
        VkResult err;

        renderThreads.resize(numThreads);
        uint32_t index = 0;
        for (auto& thread : renderThreads)
        {
            // Command pool
            VkCommandPoolCreateInfo cmdPoolInfo =
vkTools::initializers::commandPoolCreateInfo();
            cmdPoolInfo.queueFamilyIndex = swapChain.queueNodeIndex;
            cmdPoolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
            err = vkCreateCommandPool(device, &cmdPoolInfo, nullptr,
&thread.cmdPool);
            assert(!err);

            // Command buffer
            // Use secondary level command buffers
            thread.cmdBuffers.resize(swapChain.imageCount);
            VkCommandBufferAllocateInfo cmdBufAllocateInfo =
                vkTools::initializers::commandBufferAllocateInfo(
                    thread.cmdPool,
                    VK_COMMAND_BUFFER_LEVEL_SECONDARY,
                    (uint32_t)thread.cmdBuffers.size());

            err = vkAllocateCommandBuffers(device, &cmdBufAllocateInfo,
thread.cmdBuffers.data());
            assert(!err);

            // Push constant block

            // Color
            // todo : randomize
            thread.pushConstantBlock.color = glm::vec3(1.0f, 1.0f, 1.0f);

            // Model matrix
            float rot = (float)(rand() % 360);
            float deltaT = (float)(rand() % 255) / 255.0f;

            glm::mat4 modelMat = glm::translate(glm::mat4(), glm::vec3((float)index
* 4.0f - (float)(numThreads-1) * 2.0f, 0.0f, 0.0f));
            modelMat = glm::rotate(modelMat, -sinf(glm::radians(deltaT * 360.0f)) *
0.25f, glm::vec3(1.0f, 0.0f, 0.0f));
            modelMat = glm::rotate(modelMat, glm::radians(rot), glm::vec3(0.0f,
1.0f, 0.0f));
            modelMat = glm::rotate(modelMat, glm::radians(deltaT * 360.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
            thread.pushConstantBlock.model = modelMat;

            thread.thread = std::thread([=] { threadUpdate(index); });
            index++;

            // Viewport and scissor rect are shared
            VkViewport viewport = vkTools::initializers::viewport((float)width,
(float)height, 0.0f, 1.0f);
            VkRect2D scissor = vkTools::initializers::rect2D(width, height, 0, 0);

            // Fill command buffers
            for (uint32_t i = 0; i < thread.cmdBuffers.size(); ++i)
            {
                // Inheritance infor for secondary command buffers
                VkCommandBufferInheritanceInfo inheritanceInfo =
vkTools::initializers::commandBufferInheritanceInfo();
                inheritanceInfo.renderPass = renderPass;
                inheritanceInfo.framebuffer = frameBuffers[i];
                VkCommandBufferBeginInfo beginInfo =
vkTools::initializers::commandBufferBeginInfo();
                beginInfo.flags = VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT;
                beginInfo.pInheritanceInfo = &inheritanceInfo;

                vkBeginCommandBuffer(thread.cmdBuffers[i], &beginInfo);

                vkCmdSetViewport(thread.cmdBuffers[i], 0, 1, &viewport);
                vkCmdSetScissor(thread.cmdBuffers[i], 0, 1, &scissor);

                vkCmdBindPipeline(thread.cmdBuffers[i],
VK_PIPELINE_BIND_POINT_GRAPHICS, pipelines.phong);

                // Update shader push constant block
                // Contains model view matrix
                vkCmdPushConstants(
                    thread.cmdBuffers[i],
                    pipelineLayout,
                    VK_SHADER_STAGE_VERTEX_BIT,
                    0,
                    sizeof(ThreadPushConstantBlock),
                    &thread.pushConstantBlock);

                // Render mesh
                VkDeviceSize offsets[1] = { 0 };
                vkCmdBindDescriptorSets(thread.cmdBuffers[i],
VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, 1, &descriptorSet, 0, NULL);
                vkCmdBindVertexBuffers(thread.cmdBuffers[i], VERTEX_BUFFER_BIND_ID,
1, &meshes.ufo.vertices.buf, offsets);
                vkCmdBindIndexBuffer(thread.cmdBuffers[i], meshes.ufo.indices.buf,
0, VK_INDEX_TYPE_UINT32);
                vkCmdDrawIndexed(thread.cmdBuffers[i], meshes.ufo.indexCount, 1, 0,
0, 0);

                vkEndCommandBuffer(thread.cmdBuffers[i]);
            }
        }

        for (auto& thread : renderThreads)
        {
            thread.thread.join();
        }

    }

    void buildCommandBuffers()
    {
        VkCommandBufferBeginInfo cmdBufInfo =
vkTools::initializers::commandBufferBeginInfo();

        VkClearValue clearValues[2];
        clearValues[0].color = defaultClearColor;
        clearValues[1].depthStencil = { 1.0f, 0 };
        VkRenderPassBeginInfo renderPassBeginInfo =
vkTools::initializers::renderPassBeginInfo();
        renderPassBeginInfo.renderPass = renderPass;
        renderPassBeginInfo.renderArea.offset.x = 0;
        renderPassBeginInfo.renderArea.offset.y = 0;
        renderPassBeginInfo.renderArea.extent.width = width;
        renderPassBeginInfo.renderArea.extent.height = height;
        renderPassBeginInfo.clearValueCount = 2;
        renderPassBeginInfo.pClearValues = clearValues;

        VkResult err;

        for (int32_t i = 0; i < drawCmdBuffers.size(); ++i)
        {
            // Set target frame buffer
            renderPassBeginInfo.framebuffer = frameBuffers[i];

            err = vkBeginCommandBuffer(drawCmdBuffers[i], &cmdBufInfo);
            assert(!err);

            // The primary command buffer does not contain any rendering commands
            // These are stored (and retrieved) from the secondary command buffers

            vkCmdBeginRenderPass(drawCmdBuffers[i], &renderPassBeginInfo,
VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS);

            // Execute secondary command buffers
            for (auto& renderThread : renderThreads)
            {
                // todo : Make sure threads are finished before accessing their
command buffers
                vkCmdExecuteCommands(drawCmdBuffers[i], 1,
&renderThread.cmdBuffers[i]);
            }

            vkCmdEndRenderPass(drawCmdBuffers[i]);

            VkImageMemoryBarrier prePresentBarrier =
vkTools::prePresentBarrier(swapChain.buffers[i].image);
            vkCmdPipelineBarrier(
                drawCmdBuffers[i],
                VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
                VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
                VK_FLAGS_NONE,
                0, nullptr,
                0, nullptr,
                1, &prePresentBarrier);

            err = vkEndCommandBuffer(drawCmdBuffers[i]);
            assert(!err);
        }
    }

    void draw()
    {
        VkResult err;
        VkSemaphore presentCompleteSemaphore;
        VkSemaphoreCreateInfo presentCompleteSemaphoreCreateInfo =

vkTools::initializers::semaphoreCreateInfo(VK_FENCE_CREATE_SIGNALED_BIT);

        err = vkCreateSemaphore(device, &presentCompleteSemaphoreCreateInfo,
nullptr, &presentCompleteSemaphore);
        assert(!err);

        // Get next image in the swap chain (back/front buffer)
        err = swapChain.acquireNextImage(presentCompleteSemaphore, &currentBuffer);
        assert(!err);

        VkSubmitInfo submitInfo = vkTools::initializers::submitInfo();
        submitInfo.waitSemaphoreCount = 1;
        submitInfo.pWaitSemaphores = &presentCompleteSemaphore;
        submitInfo.commandBufferCount = 1;
        submitInfo.pCommandBuffers = &drawCmdBuffers[currentBuffer];

        // Submit draw command buffer
        err = vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
        assert(!err);

        err = swapChain.queuePresent(queue, currentBuffer);
        assert(!err);

        vkDestroySemaphore(device, presentCompleteSemaphore, nullptr);

        submitPostPresentBarrier(swapChain.buffers[currentBuffer].image);

        err = vkQueueWaitIdle(queue);
        assert(!err);
    }
```

Instancing functions (chapter 4.2.3):

```
void prepareUniformBuffers()
    {
        instanceCount = pow((INSTANCING_RANGE * 2) + 1, 3);
        uboVS.instance = new UboInstanceData[instanceCount];

        VkResult err;

        // Vertex shader uniform buffer block
        uint32_t uboSize = sizeof(uboVS.matrices) + (instanceCount *
sizeof(UboInstanceData));

        createBuffer(
            VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
            uboSize,
            nullptr,
            &uniformData.vsScene.buffer,
            &uniformData.vsScene.memory,
            &uniformData.vsScene.descriptor);

        VkBufferCreateInfo bufferInfo = vkTools::initializers::bufferCreateInfo(
            VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
            uboSize);

        // Colors and model matrices are fixed
        float offset = 5.0f;
        uint32_t index = 0;
        for (int32_t x = -INSTANCING_RANGE; x <= INSTANCING_RANGE; x++)
        {
            for (int32_t y = -INSTANCING_RANGE; y <= INSTANCING_RANGE; y++)
            {
                for (int32_t z = -INSTANCING_RANGE; z <= INSTANCING_RANGE; z++)
                {
                    // Instance model matrix
                    uboVS.instance[index].model = glm::translate(glm::mat4(),
glm::vec3(x * offset, y * offset, z * offset));
                    uboVS.instance[index].model =
glm::rotate(uboVS.instance[index].model, deg_to_rad(-45.0f), glm::vec3(0.0f, 1.0f,
0.0f));
                    // Instance color (randomized)
                    uboVS.instance[index].color = glm::vec4((float)(rand() % 255) /
255.0f, (float)(rand() % 255) / 255.0f, (float)(rand() % 255) / 255.0f, 1.0);
                    index++;
                }
            }
        }

        // Update instanced part of the uniform buffer
        uint8_t *pData;
        uint32_t dataOffset = sizeof(uboVS.matrices);
        uint32_t dataSize = instanceCount * sizeof(UboInstanceData);
        err = vkMapMemory(device, uniformData.vsScene.memory, dataOffset, dataSize,
0, (void **)&pData);
        assert(!err);
        memcpy(pData, uboVS.instance, dataSize);
        vkUnmapMemory(device, uniformData.vsScene.memory);

        updateUniformBufferMatrices();
    }

    void updateUniformBufferMatrices()
    {
        // Only updates the uniform buffer block part containing the global matrices

        // Projection
        uboVS.matrices.projection = glm::perspective(deg_to_rad(60.0f), (float)width
/ (float)height, 0.001f, 256.0f);

        // View
        uboVS.matrices.view = glm::translate(glm::mat4(), glm::vec3(0.0f, 0.0f,
zoom));
```

```
        uboVS.matrices.view = glm::rotate(uboVS.matrices.view,
deg_to_rad(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
        uboVS.matrices.view = glm::rotate(uboVS.matrices.view,
deg_to_rad(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
        uboVS.matrices.view = glm::rotate(uboVS.matrices.view,
deg_to_rad(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));

        // Only update the matrices part of the uniform buffer
        uint8_t *pData;
        VkResult err = vkMapMemory(device, uniformData.vsScene.memory, 0,
sizeof(uboVS.matrices), 0, (void **)&pData);
        assert(!err);
        memcpy(pData, &uboVS.matrices, sizeof(uboVS.matrices));
        vkUnmapMemory(device, uniformData.vsScene.memory);
    }
```

APPENDIX C

ADDITIONAL CREDITS/ATTRIBUTIONS

ADDITIONAL CREDITS/ATTRIBUTIONS

- Cubemap used in cubemap example by Emil Persson(aka Humus)
- Armored knight model used in deferred example by Gabriel Piacenti
- Textures used in some examples by Hugues Muller
- Sascha Willems Vulkan tutorials and examples
- Vulkan scene model (and derived models) by Dominic Agoro-Ombaka
- Vulkan and the Vulkan logo are trademarks of the Khronos Group Inc.
- OpenGL Mathematics (GLM)
- OpenGL Image (GLI)
- Open Asset Import Library
- Tiny obj loader

VITA

Joseph A. Shiraef was born in Chattanooga, TN on December 24, 1988. Raised in the community of Highland Park near downtown Chattanooga, he attended Tennessee Temple Academy High School. He graduated from The University of Tennessee at Chattanooga in May 2011 with a Bachelor's Degree in Finance. Joseph has worked for Northwestern Mutual in various positions including Financial Analyst & Financial Planner. In 2015, Joseph launched an independent game development studio Inktale Studios. Joseph graduated in May 2016 with a Master of Science Degree in Computer Science from The University of Tennessee at Chattanooga and plans to advance his career with Amazon.com Inc.