Fynn Flügge

Design and Implementation of a Vulkan Engine

Case Study of Capabilities and Performance

Master's Thesis

GRIN

Bibliographic information published by the German National Library:

The German National Library lists this publication in the National Bibliography; detailed bibliographic data are available on the Internet at http://dnb.dnb.de .

Imprint:

Copyright © 2018 GRIN Verlag
ISBN: 9783668868236

This book at GRIN:

https://www.grin.com/document/456305

Fynn Flügge

Design and Implementation of a Vulkan Engine

Case Study of Capabilities and Performance

GRIN Verlag

Master Thesis

Design and Implementation of a Vulkan Engine

Case Study of Capabilities and Performance

Institute of Embedded Systems

Author:
Fynn-Jorin Flügge

September 25, 2018

Abstract

The Vulkan API, released in February 2016, is the Khronos Group's answer to Microsoft's Direct3D 12 API published in 2015. Due to the revolutionary capabilities provided by the new API's to the programmer, the releases were accompanied by an enormous hype. Vulkan and Direct3D 12 provides the programmer unprecedented control and empowerment over the GPU and its memory, which might introduce a new era in GPU computing. This elaboration deals with the design and implementation of a graphics engine along with state-of-the-art rendering features using the Vulkan API. The Vulkan engine is built upon the OpenGL engine *Oreon Engine* [1] developed in a previous thesis and used in the research elaboration *Realtime GPGPU FFT Ocean Water Simulation* [2]. Finally, an extensive study concering the capabilities of the new Vulkan API and its performance advantage compared to OpenGL is demonstrated.

Acknowledgements

I gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

Contents

List of Figures

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
FFT	Fast Fourier Transform
FLOPS	Floating point operations per second
GB/s	Gigabit per second
GFLOPS	Giga FLOPS ≙ billion FLOPS
GPU	Graphical Processing Unit
RAM	Random Access Memory
MSAA	Multisample Antialiasing
FXAA	Fast Approximated Antialiasing
FPS	Frames Per Second

Chapter 1

Introduction

Over the past years, GPU computing has become more and more relevant in a wide range of application environments. Since the early 2000s, when the first GPUs with programmable shader units were launched, the bandwidth of GPU accelerated applications has reached new dimensions. Since then, GPUs have taken on much more the "role as a processor" [3, p.18], because GPUs are significant faster than CPUs in solving tasks with high data parallelism. Figure 1.1 demonstrates the evolution of GPU and CPU performance measured in computing power (GFLOPS) and the data transfer rate (GB/s) from 2001 to 2014 and 2003 to 2013, respectively. Already in the years 2002 to 2003, the GPU performance (in GFLOPS) has risen up to the factor of 6, while the clock rate even decreased by 20% as listed in figure 1.2. Due to that enormous role change of GPUs and the resulting new possibilities in offscreen computing, many industrial sectors next to the gaming and film industry considered this progress. As a result of the increasing demand for GPU programmers, e.g. financial companies hired game programmers to meet these new challenges. [3] [4]

Over the next years, high-level GPU APIs (like *CUDA*, *Stream* or *OpenCL*) had to rely on the graphics APIs *Direct3D* or *OpenGL*, since Direct3D and OpenGL serve as the interface between CPU and GPU communication. Direct3D's and OpenGL's initial releases were in the early 1990s. Obviously computer hardware has advanced enormously until today, which reveals some disadvantages in still relying on the low-level GPU APIs Direct3D (pre Version 12) and OpenGL. Hence, *Microsoft* released Direct3D 12, an overhauled version of its graphics API Direct3D within the scope of the *DirectX* 12 release. Soon after, in early 2016, *Vulkan* was released by the *Khronos Group.* Unlike Direct3D 11 (and backwards) and OpenGL, Vulkan and Direct3D 12 were evolved under consideration of modern computer hardware's architecture and features to unleash its full power. [3] [5] [6]

This elaboration targets the development of a Vulkan graphics engine with a performance comparison of Vulkan and OpenGL. The next chapter introduces the Vulkan API and its history and points out the advantages of Vulkan compared to OpenGL and Direct3D 11 (and backwards) on modern computer hardware. Following this, the core elements of the Vulkan API specification used in the developed Vulkan engine are described in chapter three. The realtime simulation scenario including a GPGPU FFT generated ocean for the case study and the Vulkan engine implementation design is extensively explained and illustrated in the subsequent two chapters. Finally, the measured performance results of the simulation scenario with Vulkan and OpenGL are presented and compared.

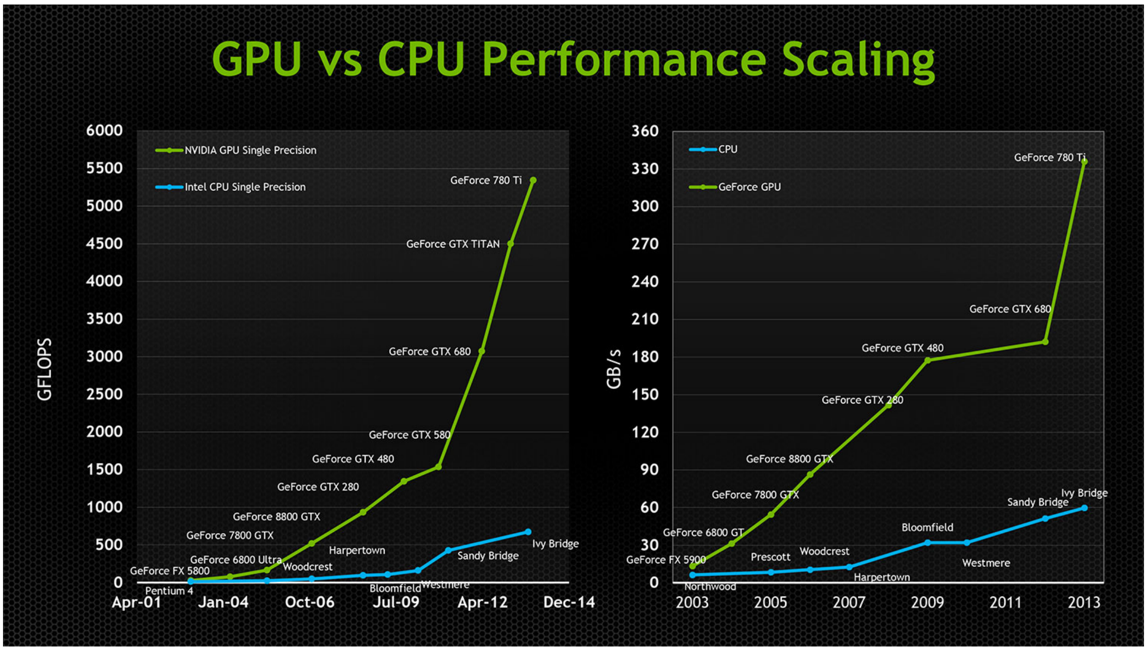
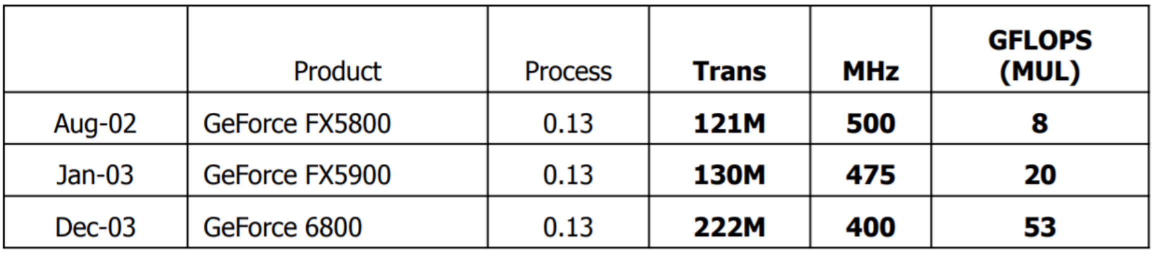


Figure 1.1: GPU vs CPU Peformance Scaling [4]
The two graphs show how much GPUs have outscaled CPUs in GFLOPS and GB/s over the past years. While the *Pentium 4* and the *Gefore FX5800* were head to head in 2002, the *Gefore 780 Ti* reaches a performance up to around 900% higher than the *Intel Ivy Bridge* CPUs in 2013. Same for GB/s: While in 2003 CPUs and GPUs were pretty much on the same level, in 2013 the 780 Ti has a lead of around 500% against the Ivy Bridge CPUs.



	Product	Process	**Trans**	**MHz**	**GFLOPS (MUL)**
Aug-02	GeForce FX5800	0.13	**121M**	**500**	**8**
Jan-03	GeForce FX5900	0.13	**130M**	**475**	**20**
Dec-03	GeForce 6800	0.13	**222M**	**400**	**53**

Figure 1.2: Comparison of Nvidia GPUs [3, p. 16]

The GFLOPS of the Geforce FX5900 has increased by the factor of 2.5 compared to the *Geforce FX5800*, while the transistor count just increased by the factor of 1.074 and the clock rate even decreased by 5%. The *Geforce 6800* reaches a performance of 50 GFLOPS, 2.65 times more than the FX5900 and 6.625 times more than the FX5800, while the transistor count has increased by the factor of 1.7 and the clock rate decreased by 15,8% compared to the FX5900.

Chapter 2

Why New API?

This chapter introduces Vulkan and its history and outlines the advantages of Vulkan over OpenGL. Finally, the question is clarified if OpenGL is still viable or if it will be outdated by Vulkan soon.

2.1 Origin and History of Vulkan

Vulkan is a cross-plattform next generation 3D graphics and compute API and is considered as the successor to OpenGL and *OpenGL ES*[1]. The official 1.0 specification of Vulkan was released on the 16th of February 2016 by the Khronos Group. [7]

In contrast to OpenGL, Vulkan is a minimal abstraction of the GPU hardware, which facilitates its portability across multiple GPU vendors and device types, such as desktop, mobile or embedded systems. The first discussions about the idea of a completely new graphics API were already held in October 2012. Until the 1.0 release in February 2016, famous hardware and software vendors like *Nvidia*, *AMD*, *Lucasfilm Ltd.*, *EA*, *Epic Games* and many more contributed to the development of Vulkan. [8] [9]

In contrast to the competing APIs DirectX 12 (*Microsoft Windows*) and *Metal* (*macOS*, *iOS*), Vulkan is the only cross-platform next generation graphics API. Further, Vulkan is not only cross-platform but also supported by mobile devices and embedded systems across multiple GPU hardware vendors. Figure 2.1 illustrates the platform support comparison of the three next generation APIs Vulkan, DirectX 12 and Metal. In 2018, Vulkan is also supported by *Apple* platforms with the *MoltenVK* API, which maps Vulkan to Metal. Additionally, many popular Game Engines like *Unreal Engine*, *Unity*, *CryEngine* and *Xenko* offer Vulkan support. [9] [10]

[1] OpenGL ES is the OpenGL congruent specification for embedded systems

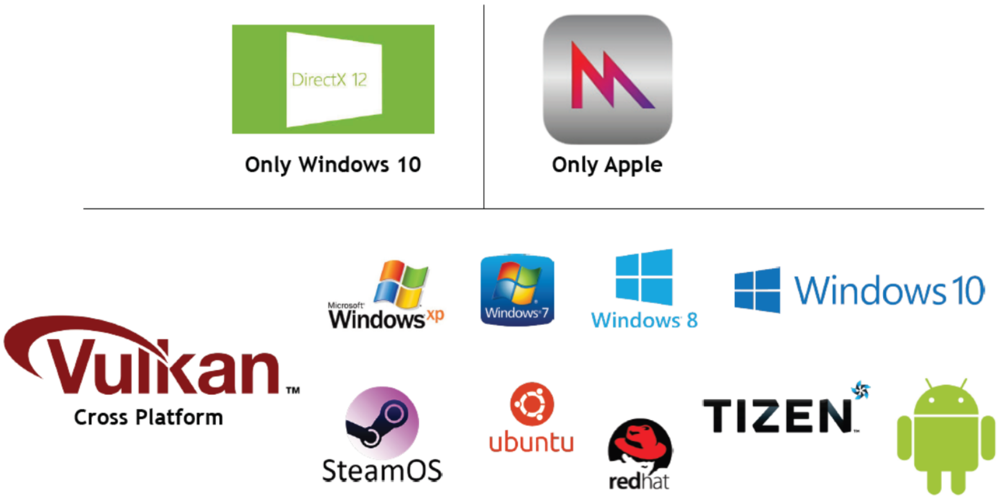


Figure 2.1: Platform Support of Next Generation GPU APIs [9, p. 21]
The illustration shows the range of Vulkan platform support in contrast to DirectX 12 and Metal. While Vulkan offers platform support for the latest four Windows versions, Ubuntu and Android (and some more), DirectX12 and Metal are dedicated to Windows 10 and MacOS respectively. Meanwhile, Vulkan runs also on Nintendo Switch consoles and Apple platforms (with MoltenVK).

The next section clarifies the question why there was such a big desire by the developers for a new graphics API and why Vulkan is considered to be next generation. In order to confirm this, a theoretical and abstract comparison of Vulkan and OpenGL in performance and efficiency is demonstrated.

2.2 Vulkan - More Performance and Efficieny

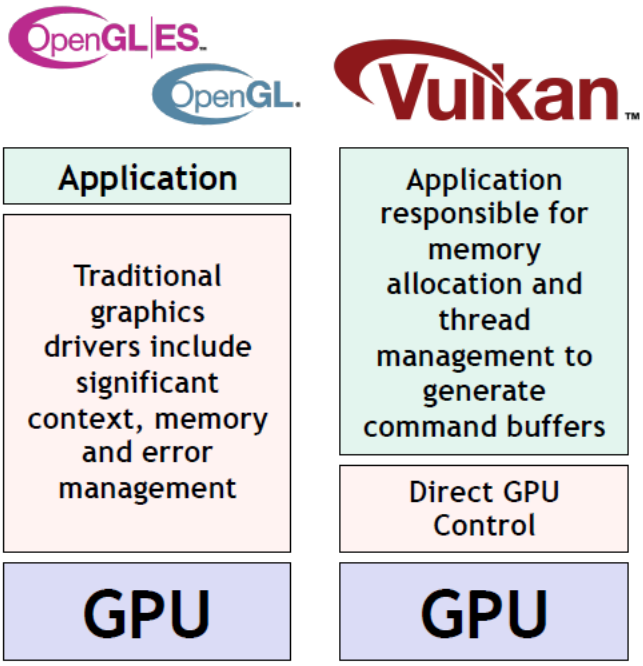
The graphics API OpenGL has been initially released in 1992. Since then, the architecture of GPUs and platforms has evolved enormously. OpenGL was fundamentally invented for fixed graphic workstations with single-threaded direct renderers and split memory. 25 years ago, no considerations about multi-core CPU architecture and multi-threading were made. Hence, OpenGL doesn't provide a parallel multithreading execution model and thus multi-core CPUs cannot be effectively used. Further, the OpenGL model doesn't match mobile device arichtectures and platforms. That's why OpenGL ES has been outsourced as a standalone API for smartphones, tablets and video game consoles. However, OpenGL ES is based on OpenGL which was not tailored for mobile GPU hardware. [11] [8] [12]

A further point is that GPU vendors are responsible for their individual OpenGL API implementation within the scope of their GPU drivers, since OpenGL is just an API specification, which describes the interface and its expected behaviour. This leads to complex and unpredictable drivers with different bugs on different GPUs. Programmers doesn't really know what is happening behind the OpenGL interface within the related driver and its hidden implementation. The driver has to do lots of work like state validation, dependency tracking and error checking. This driver overhead limits or even randomizes the performance. In addition, each driver has to provide an implementation of the GLSL shader language compiler and thus different behavior of the same GLSL shadercode across different GPU drivers can occur. As a consequence, software that uses OpenGL must be tested against multiple GPU vendors and often implementation variabilities across these vendors are necessary. [11] [13] [5]

In summary, OpenGL can no longer be considered as contemporary and thus programmers desired a completely new next-generation GPU API. The answer to the programmer's desirement is the Vulkan API. Since software in graphics, vision and deep learning across diverse devices and platforms will profit from GPU acceleration, a next generation GPU API should be flexible and portable. In contrast to OpenGL, Vulkan is designed for modern cross-platform usage on cloud, desktop, console, mobile and embedded devices. Hence, there are no separate APIs necessary for desktop and mobile devices as OpenGL and OpenGL ES. Further, Vulkan provides efficient usage of multi-core CPUs and a parallel multithreading execution model. GPU vendors does not need to provide their individual implementations of Vulkan for their drivers, because Vulkan is not only a specification that just defines the interface and its behaviour (as it is the case in OpenGL) but Vulkan is also open source with one explicit implementation for all GPU vendors. Hence, the problem that software needs often a tailored implementation for different GPU vendors, as it is the case for OpenGL, does not exist for Vulkan. Moreover GPU vendors do not even need to provide a compiler for the shader language within their drivers, because Vulkan uses precompiled *SPIR-V* shader files. [5] [9]

However, the most important innovation in the Vulkan API is the principle of explicit control. As mentioned before, one big problem with OpenGL is the driver overhead. The programmer must explicitly tell the Vulkan driver everything he is going to do in advance. This explicitness simplifies the GPU driver and favors cross vendor consistency, which makes Vulkan that much portable and flexible. Moreover, the explicit control reduces driver overhead and latency, which leads to a reduced CPU load and at the end a better performance. In contrast, OpenGL allows the programmer to change the state at any time which may result in huge performance costs, especially when the programmer changes the OpenGL render state very late just before a draw or compute command. [5] [9] [8]

As a final point, Vulkan allows the programmer to disable validation and error checking (for example in delivery versions) which reduces driver overhead even further. [5] Figure 2.2 illustrates the driver overhead comparison of Vulkan and OpenGL/OpenGL ES.



OpenGL ES
OpenGL
Vulkan
Application
Traditional graphics drivers include significant context, memory and error management
GPU
Application responsible for memory allocation and thread management to generate command buffers
Direct GPU Control
GPU

Figure 2.2: Vulkan Explicit GPU Control [5, p. 16]

Figure 2.2 delineates the driver overhead (orange-colored blocks) and the application responsibility (green-colored blocks) of Vulkan and OpenGL/OpenGL ES over the GPU. The application-block of Vulkan is multiple times larger than the OpenGL's application-block, which means that the programmer is responsible for many things in order to run a Vulkan application efficiently. As a consequence, the driver overhead of Vulkan is multiple times smaller than in OpenGL/OpenGL ES, which leads to a better performance and efficiency.

2.3 Will OpenGL Get Outdated?

After listing the strengths of Vulkan in contrast to the weaknesses of OpenGL, the question arises whether Vulkan will supersede OpenGL completely in the future. Even if it is obvious after a close examination that Vulkan has much more potential than OpenGL and solves all the problems OpenGL poses, the answer to this question is most likely: No.

Using Vulkan is extremely challenging and needs a lot of effort compared to OpenGL. For purposes where the main focus is not on performance, OpenGL is still more appropriate and the Khronos Group will not discontinue the evolvement of OpenGL and OpenGL ES soon. Nevertheless, for applications where the work can be well parallelized and the CPU load and performance is significant it is worth using Vulkan despite its many challenges and big effort.

Chapter 3

Vulkan API Overview

This chapter introduces the core elements and functions of the Vulkan API. All of the following Vulkan elements are used in various ways by the implementation of the Vulkan engine developed in the scope of this thesis.

First of all, the Vulkan syntax and its handling is mentioned. Every Vulkan function starts with the prefix `vk`. Vulkan objects or structures starts with `Vk` and enumerations with `VK`. Informations for object creations are generally handled as structures. These structures are created according to the following pattern in *C++* and *Java* and are used as parameters for object creation functions with the prefix `vkCreate`.

```
// C++ Syntax
VkXXXInfo info = {};
info.sType = VK_STRUCTURE_TYPE_XXX_INFO;
info.pNext = nullptr;
info.foo = ...;
info.bar = ...;

// Java Syntax
VkXXXInfo info = VkXXXInfo.calloc();
info.sType(VK_STRUCTURE_TYPE_XXX_INFO)
info.pNext(VK_NULL_HANDLE)
info.foo(...)
info.bar(...);
```

Listing 3.1: Vulkan Structs Pattern

`XXX` is a placeholder for the Vulkan information name (e.g. `VkFrameBufferCreateInfo` for creating a `VkFrameBuffer` object). The `sType` parameter is an enumeration with again `XXX` as the placeholder for the name of the Vulkan information. `pNext` is an optional pointer to a struct of a Vulkan extension which is rarely used (not used by the Vulkan implementaion of this thesis), hence, it can be set to a nullpointer (C++) resp. to the enumeration `VK_NULL_HANDLE` (Java). Vulkan functions generally return an enumeration which is `VK_SUCCESS` for successful execution or a specific enumeration to identify an error during function call. [14] [15] [16]

3.1 Layers

As metioned in section 2.2, all kind of validation and error checking in Vulkan can be enabled or disabled. Since Vulkan is a minimal hardware abstraction to reduce the overhead as much as possible, the driver has no built-in validation and error checking. The Vulkan driver does not provide any feedback and assumes that the programmer does everything correctly. However, to enable any kind of validation Vulkan was designed as a layered architecture. Developers can enbale validation by inserting layers between the application (top layer) and the Vulkan API (bottom layer). Layers facilitates the development with Vulkan, since they provide validation and error checking by intercepting Vulkan functions and modify or evaluate them, so that the developer can debug its application or obtain any kind feedback when calling Vulkan functions. It is even possible to cascade multiple layers into a layer chain. However, layer injection should be disbaled in released versions, since they decrease the performance. Furthermore, developers can create their own layers for individual purposes. [17] [14]

A set of useful validation layers are provided by *LunarG*[1] in its Vulkan SDK. The layer `VK_LAYER_LUNARG_standard_validation` involves a layer chain for common usage and is sufficient in most use cases. Since SDK 1.0.68, `VK_LAYER_LUNARG_assistant_layer` is available which provides feedback about potential performance issues or suspect usage patterns. [18]

Figure 3.1 shows the interaction of Vulkan layers with Vulkan drivers and applications.

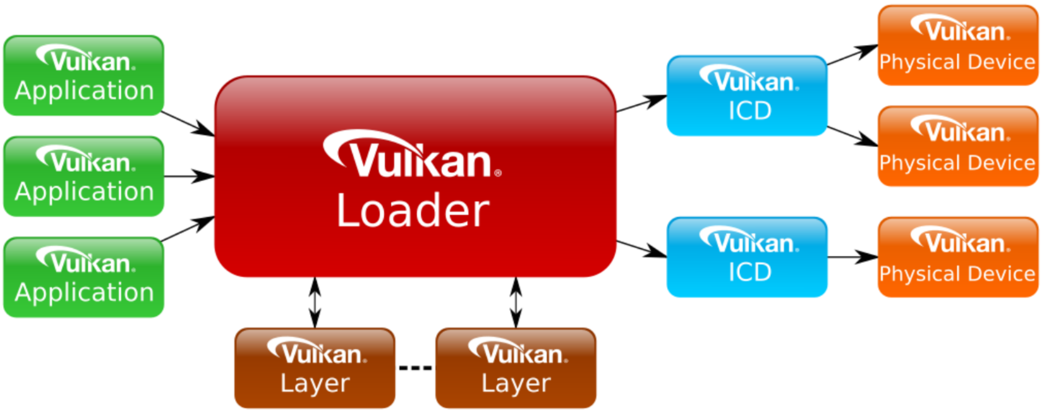


Figure 3.1: Vulkan Loader [17]
The Vulkan loader is the interface between application and Vulkan drivers (ICD) with the related devices. The loader injects enabled layers between application and driver and delivers Vulkan function calls to a specific ICD by inserting a set of layers prior to the a subsequent function call by the ICD. The Vulkan loader is responsible for supporting on or more independent Vulkan ICD's on a system. [17]

[1] Software company responsible for Vulkan runtime and SDK

3.2 Extensions

Extensions expand the functionality of the Vulkan API by providing new functions, structures or enumerations. Extensions can be created by independent developers and must be specified and registered by the Khronos Group in order to become an official extension within the published Vulkan specification. There exist two types of extensions, instance-level extensions and device-level extensions. Instance-level extensions extend the functionality of `VkInstance` objects, while device-level extenions extend the functionality of `VkDevice` objects. If the programmer intends to use extensions in the application, the extension-support must be queried and enbaled in advance at instance and device creation, respectively. [14] [17]

3.3 Vulkan Instance

The Vulkan object `VkInstance` is the starting point for every Vulkan application. It exists exactly once per application. The `VkInstance` gathers information about the application such as application name, engine name and version. Also used extensions and layers must be specified in advance at `VkInstance` creation with the `vkCreateInstance` function call. In order to specify extensions, their platform availablity should be queried with the `vkEnumerateInstanceExtensionProperties` function. Further, available physical devices (GPUs) must be queried and created as logical devices from the Vulkan instance. [14] [16]

3.4 Devices

Devices in Vulkan represent GPUs within the operating system. The GPU representation is splitted into physical and logical devices as the Vulkan objects `VkPhysicalDevice` (representing a physical device) and `VkDevice` (representing a logical device). The logical device object `VkDevice`, on which all common Vulkan operations like drawing, computing or memory allocations are executed, serves as the interface to the GPU. In order to create a `VkDevice` object for feeding the GPU with work, the operating system has to be queried for available physical devices with the `vkEnumeratePhysicalDevices` function. Since Vulkan applications can run on workstations, notebooks, tablets or mobile phones with all different graphics hardware and different performance and capabilities, the GPU capabilities must be checked against the application's needs. Apart from that, a system may have multiple GPUs installed so that one GPU needs to be selected by the programmer properly, which suits the application's needs best. Once a `VkPhysicalDevice` object has been selected and created, its properties and features can be queried. [16]

3.4.1 VkPhysicalDeviceProperties

The `VkPhysicalDeviceProperties` structure holds general information about the related physical device and is retrieved by `vkGetPhysicalDeviceProperties`. The following listing shows the `VkPhysicalDeviceProperties` of the *Titan Xp*:

```
apiVersion = 0x401046 (1.1.70)
driverVersion = 1669922816 (0x63890000)
vendorID = 0x10de
deviceID = 0x1b02
deviceName = TITAN Xp
deviceType = VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
```

Listing 3.2: VkPhysicalDeviceProperties of the Titan Xp

The field `apiVersion` indicates the Vulkan version supported by the device. The following field `driverVersion` exposes the version of the installed GPU device driver. `vendorID` and `deviceID` are unique identifiers of the GPU vendor and its manufactured device. The value `0x10de` (decimal 4318) identifies the Nvidia Corporation. The field `deviceName` denotes the vendor's GPU model name and `deviceType` indicates the type of the GPU's hardware architecture. [14]

The enumeration `VkPhysicalDeviceType` lists all device types that are differentiated by Vulkan. Listing 3.3 points out the `VkPhysicalDeviceType` enumeration:

```
typedef enum VkPhysicalDeviceType {
    VK_PHYSICAL_DEVICE_TYPE_OTHER = 0,
    VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1,
    VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2,
    VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3,
    VK_PHYSICAL_DEVICE_TYPE_CPU = 4,
} VkPhysicalDeviceType;
```

Listing 3.3: VkPhysicalDeviceType enumeration

As mentioned in Listing 3.2, the Titan Xp is of discrete type, which means that the GPU is separated from the CPU with its own VRAM. In contrast, integrated device types are embedded in or closely located to the CPU without an own RAM, since they share the system RAM with the CPU. Generally integrated GPUs are installed in mobile devices or laptops. The type `VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU` denotes that the GPU is a virtual instance and `VK_PHYSICAL_DEVICE_TYPE_CPU` are GPUs and CPUs as a common entity. Further, the `VkPhysicalDeviceProperties` structure holds an object of `VkPhysicalDeviceLimits`, which indicates the limitations of the device such as the maximum framebuffer size or the maximum number of color attachments. [14]

3.4.2 VkPhysicalDeviceFeatures

VkPhysicalDeviceFeatures, obtained from vkGetPhysicalDeviceFeatures, is a structure of boolean flags indicating the support of specific features (e.g. the availability of tessellation shaders or samplers with anisotropic filtering support). [14]

3.4.3 VkPhysicalDeviceMemoryProperties

The Vulkan execution model differentiates between the following three kinds of physical memory, where a host in Vulkan represents the CPU environment which hosts one or more devices:

- **Device local** is the video memory (VRAM). The VRAM is physically connected to the device (GPU) without direct access from the host (CPU).

- **Device local, host visible** is a unified memory phsysically connected to the device and host.

- **Host local, host visible** is the host's system memory and physically connected to the host only but also accessible by the device.

The VkPhysicalDeviceMemoryProperties object exposes information regarding available memories and is returned by vkGetPhysicalDeviceMemoryProperties. The VkPhysicalDeviceMemoryProperties object contains two arrays of the structures VkMemoryHeap and VkMemoryType. The VkMemoryHeap structure contains the size of the memory and a bitmask composed of attribute flags. VkMemoryType holds an index for indentifying to which VkMemoryHeap it corresponds and a bitmask composed of property flags. [14]

Listing 3.4 shows the VkPhysicalDeviceMemoryProperties structure of the *Titan Xp*. It reveals two found memory heaps (l. 1). The first memory is of size 11.86 GB (l. 3) with the attribute VK_MEMORY_HEAP_DEVICE_LOCAL_BIT, which indicates that the memory is physically connected to the GPU. Further, there are eleven memory types found (l. 9). The VkMemoryType at index seven and eight (ll. 13-20) belongs to the first memory (the GPU corresponding VRAM) of the two memory heaps, since heapIndex = 0 (ll. 14, 18). Since their property flags are both VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT only (lines 16, 20), it can be derived that the first memory heap is the local video memory (VRAM) of the graphics card. The second found memory heap with size of 15.96 GB (l. 6) and no attribute flags (l. 7) with corresponding memory types with indices 0 to 6 (ll. 10-12) and 9 to 10 (ll. 21-31) is the host's system memory, since the corresponding memory types at index 9 and 10 exposes the property flag VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT (ll. 24, 29), which indicates that the memory heap is directly accessible by the host. VK_MEMORY_PROPERTY_HOST_COHERENT_BIT (ll. 25, 30) specifies that no further host cache management commands are needed when flushing host writes to device memory or make device writes host-visible. VK_MEMORY_PROPERTY_HOST_CACHED_BIT (l. 31) denotes that the memory type is cached on the host which speeds up memory access. The memory types with index 0 to 6 have no property flags. Hence, these types have no usage for Vulkan. [14]

```
memoryHeapCount = 2
memoryHeaps[0] :
        size = 12734955520 (0x2f7100000) (11.86 GiB)
        flags: VK_MEMORY_HEAP_DEVICE_LOCAL_BIT
memoryHeaps[1] :
        size = 17140023296 (0x3fda00000) (15.96 GiB)
        flags: None

memoryTypeCount = 11
memoryTypes[0..6] :
        heapIndex = 1
        propertyFlags = 0x0:
memoryTypes[7] :
        heapIndex = 0
        propertyFlags = 0x1:
                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
memoryTypes[8] :
        heapIndex = 0
        propertyFlags = 0x1:
                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
memoryTypes[9] :
        heapIndex = 1
        propertyFlags = 0x6:
                VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
                VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
memoryTypes[10] :
        heapIndex = 1
        propertyFlags = 0xe:
                VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
                VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
                VK_MEMORY_PROPERTY_HOST_CACHED_BIT
```

Listing 3.4: VkPhysicalDeviceMemoryProperties of the Titan Xp

```
memoryHeapCount = 1
memoryHeaps[0] :
        size = 3805384089 (0xe2d18d99) (3.54 GiB)
        flags: VK_MEMORY_HEAP_DEVICE_LOCAL_BIT
memoryTypeCount = 2
memoryTypes[0] :
        heapIndex = 0
        propertyFlags = 0x7:
                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
                VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
                VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
memoryTypes[1] :
        heapIndex = 0
        propertyFlags = 0xf:
                VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
                VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
                VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
                VK_MEMORY_PROPERTY_HOST_CACHED_BIT
```

Listing 3.5: VkPhysicalDeviceMemoryProperties of the Intel HD 620

In contrast to the system with the Titan Xp as a discrete GPU with its local VRAM, listing 3.5 shows the memory properties of the integrated *Intel HD Graphics 620* with a unified memory. Since only one memory heap (lines 2-4) with two memory types (lines 6-18) containing the flags `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` (lines 9, 15) and `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` (lines 10, 16) is found for the Intel HD Graphics 620, it follows that the device shares the memory with the host as a unified memory, which is the common architecture for integrated GPUs. [14]

When allocating memory in a Vulkan application, it is important to consider all available `VkMemoryType`s and choose the type which suits the needs of the memory to allocate best, since it can affect the performance significantly. [16]

3.4.4 VkDevice

Once a physical device object `VkPhysicalDevice` is created, the logical device object `VkDevice` can be generated from it. Almost all GPU work in Vulkan is processed on logical devices with its corresponding queues. The work is submitted to queues as command buffers, where they are processed by the related device. Queues and command buffers are covered later in this chapter. Device-level extensions which are intended to be used by the application must be explicity specified during logical device creation. In order to do this, the extension-support by the physical device should be checked with the `vkEnumerateDeviceExtensionProperties` command. Further available features from `VkPhysicalDeviceFeatures` must be explicitly enabled during `VkDevice` creation. It also needs to be specified which queue families the application is intended to use. [14] [16]

3.5 Queues

Queues in Vulkan (`VkQueue` objects) receive and process execution commands (like draw/compute commands or memory operations). Vulkan differentiates four types of queues. Every queue type has its individual capabilities and is dedicated for specific kinds of execution commands. The queues are divided into the following types:

- **Graphics** queues are specialized for draw commands. Further, only graphics queues are able to display images onto the screen.
- **Compute** queues are optimized for offscreen compute operations.
- **Transfer** queues are optimized for memory operations.
- **Sparse** queues support sparse memory operations.

Different GPUs possess individual sets of queues, which support one or more queue types. These queue sets are seperated into queue families. The available queue families of a physical device can be queried with `vkGetPhysicalDeviceQueueFamilyProperties`. [14] [16]

```
 1  VkQueueFamilyProperties[0]:
 2          queueFlags = GRAPHICS | COMPUTE | TRANSFER | SPARSE
 3          queueCount = 16
 4          timestampValidBits = 64
 5          minImageTransferGranularity = (1, 1, 1)
 6  VkQueueFamilyProperties[1]:
 7          queueFlags = TRANSFER
 8          queueCount = 1
 9          timestampValidBits = 64
10          minImageTransferGranularity = (1, 1, 1)
11  VkQueueFamilyProperties[2]:
12          queueFlags = COMPUTE
13          queueCount = 8
14          timestampValidBits = 64
15          minImageTransferGranularity = (1, 1, 1)
```

Listing 3.6: VkQueueFamilyProperties of the Titan Xp

Listing 3.6 shows the queue families of the Titan Xp. The first queue family (lines 1-5) supports graphics, compute, transfer and sparse queue types (line 2). There are 16 queues of this queue family available (line 3). The `timestampValidBits` (line 4) field indicates how many bits are available when writing timestamps into memory. The field `minImageTransferGranularity` (line 5) indicates the minimum granularity of image texel block transfer operations for x-,y- and z-dimensions of the image. However, `timestampValidBits` and `minImageTransferGranularity` are not further considered in this thesis. The second queue family supports transfer capabilities (line 7) and contains one queue (line 8). The third queue family provides eight queues (line 11) of compute type (line 12). [14]

The Vulkan queue model facilitates concurrency. For optimal usage, the application can split its work into segments and submit these segments to multiple queues. For best performance, the work segments can be categorized into graphics, compute, transfer or sparse type and submitted to a queue of the appropriate queue family. The work must be submitted to queues as `VkCommandBuffer` objects (more on command buffers in section 3.7). Since the queues process their work asynchronously, the synchronization of the segments is completely in the developer's hands. Moreover, it is not guaranteed that one queue processes its work in the same order the work was submitted to the queue. For proper inter- and intra-queue synchronization, Vulkan provides various synchronization objects (more in section 3.15). As mentioned in section 3.4.4, it must be explicitly specified at `VkDevice` creation which queue families are intended to be used. After logical device creation, queues can be accessed via `vkGetDeviceQueue`. [16]

3.6 Window System Integration

Since Vulkan is a platform-agnostic low-level hardware abstraction, the API core itself does not provide a mechanism to present images onto a screen. In order to display rendered images, a set of instance-level and device-level extensions must be explicitly enabled. The instance-level extension `VK_KHR_surface` provides a `VkSurfaceKHR` object, which represents a logical abstraction of a native platform surface. An additional instance-level extension must be enabled to connect the generic surface object `VkSurfaceKHR` to the platform specific window system (e.g. `VK_KHR_win32_surface` for Windows platforms). After enabling the `VkSurfaceKHR` extension and the appropriate platform surface extension at instance creation, the surface can be created with `vkCreateWin32SurfaceKHR` on Windows platforms (similar for other platforms, e.g. `vkCreateAndroidSurfaceKHR` on Android platforms). [19] [20]

For presenting images onto the surface, a swapchain provided by the device-level extension `VK_KHR_swapchain` is needed. Swapchains in Vulkan are responsible for displaying rendered images and must be created explicitly by the programmer. The Vulkan core API does not provide a swapchain (or default framebuffer as OpenGL does) itself. Even though Vulkan is a graphics API, not every use case needs a graphics output, such as GPGPU accelerated applications. Further, different operating systems provide different window systems for displaying images onto a monitor. However, Vukan is a platform-agnostic API. That's why surfaces and swapchains are provided by extensions and must be created explicitly and OS dependent by the programmer if a monitor output is desired. In order to create a swapchain, a device needs to be selected, which possesses a queue family with the capability of displaying images to a window surface, but not every queue family supports presentation to a given surface. Hence, the device-extension function `vkGetPhysicalDeviceSurfaceSupportKHR` verifies the presentation support for a given logical device, surface and queue. [19] [16]

As mentioned, swapchains are responsible for the display output and holds a set of images for presenting them. To present something on the window surface, the application acquires an image from the swapchain, renders to the image and returns it afterwards back to the swapchain. A swapchain image is acquired with `vkAcquireNextImageKHR` and returned back as a present request to the swapchain with `vkQueuePresentKHR`. At present request with `vkQueuePresentKHR` a queue that supports presentation must be specified. The swapchain provides four different presentation modes specifying in which condition the images are presented. By selection of an appropriate mode, screen tearing[1] can be prevented. Since not every device supports all four presentation modes, available modes must be checked in advance. The presentation modes are presented in the following. [19] [16] [20]

[1] Screen tearing is an effect where artifacts from multiple images are displayed at once.

Immediate

In the immediate presentation mode the swapchain holds exactly one presentable image and immediately displays the image at application's present request. At high frame rates tearing may be noticeable. Figure 3.2 shows an immediate presentation scenario. The presentation engine symbolizes the swapchain with the surface resp. monitor. The swapchain holds seven images. Images 1, 5 and 7 are acquired by the application. Images 2, 3 and 6 are unused and ready to acquire. Image 4 is currently present on the monitor. Once the application calls a present request on one of the acquired images, image 4 is immediately displaced by the acquired image on which the present request is called. Simultaneously, image 4 is moved to the unused image pool of the swapchain. [14] [20]

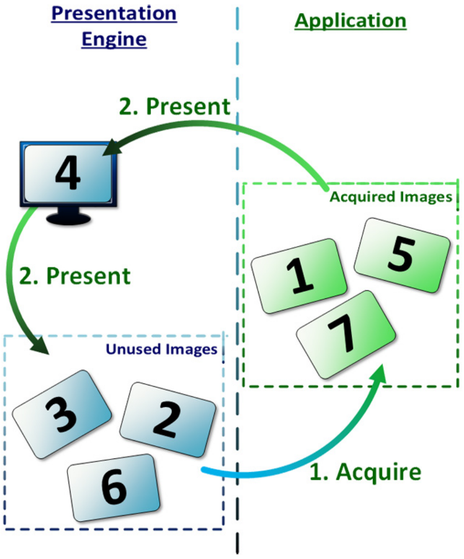


Figure 3.2: Immediate Mode [20]

FIFO

In this mode, the swapchain holds a queue of presentable images and selects the image to display according to FIFO rule. Tearing does not occur, since the swapchain is waiting for v-sync[1] signals of the monitor before replacing the displayed image by an image from the queue. Figure 3.3 shows a FIFO presentation mode example. Images 1, 5 and 7 are acquired by the application. Images 2 and 3 are unused and ready to acquire. Image 4 is currently present on the monitor. Image 6 is waiting in the FIFO-queue for being displayed. Once the application calls a present request on an acquired image, this image is added to the FIFO-queue. After v-sync is signalled, image 4 is displaced by image 6, the first (and only) image in the queue. Simultaneously, image 4 is moved to the unused image pool of the swapchain. [14] [20]

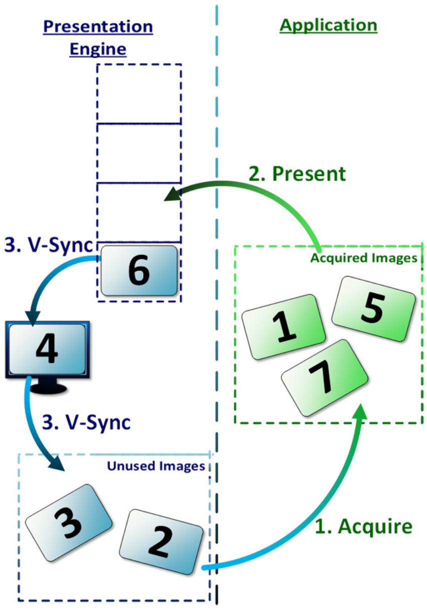


Figure 3.3: FIFO Mode [20]

[1] Vertical synchronization prevents the display image data to be updated while the monitor builds up its screen.

FIFO Relaxed

This mode is similar to FIFO, but the presented image is immediately released after the first v-sync signal since it was displayed. The FIFO Relaxed mode may expose tearing when the refresh rate of the monitor exceeds the framerate of the application. Hence, this mode is only reasonable for applications with high framerates. [20]

Mailbox

The Mailbox mode is similar to FIFO, but only one presentable image is waiting for being displayed. If the application calls a present request on an acquired image, the image that waits for being presented is replaced by the image on which the present call was executed. Figure 3.4 shows such a mailbox presentation scenario where image 6 is moved to the unused image pool instead of being presented, because the present request by the application on an acquired image occured before the v-sync signal. [14] [20]

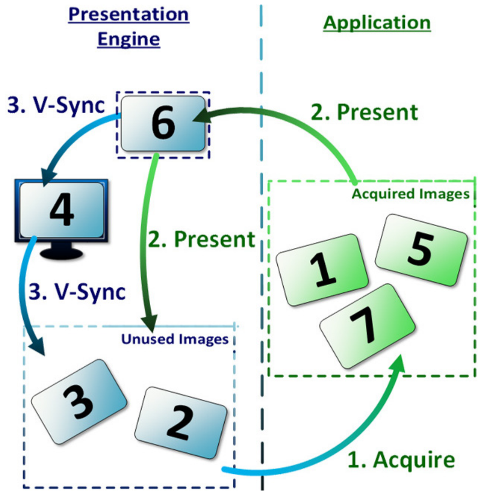


Figure 3.4: Mailbox Mode [20]

The presentation of images needs additional synchronization between swapchain and logical device queue with Semaphores [16]. Synchronization is covered in section 3.15.

3.7 Command Buffers

Command buffers, represented by `VkCommandBuffer` objects, are containers holding a set of execution commands. Command buffers must be allocated from command pools (`VkCommandPool`). All kind of work for GPUs is submitted as command buffers via `VkQueue` objects to a device. In contrast to OpenGL, where commands are implicitly collected by the driver and immediately transferred to the GPU, multiple Vulkan GPU commands might be recorded in advance into a single `VkCommandBuffer` and can be submitted all at once. Recorded command buffers can afterwards be reused as often as intended. Hence, command buffers relieves the CPU a lot by summarizing multiple GPU commands into a `VkCommandBuffer` object, which can be submitted to the GPU with only one CPU call. Vulkan command record functions have the prefix `vkCmd`.

Command buffers can be recorded concurrently and submitted via multiple threads. Further, Vulkan provides secondary and primary command buffers. Multiple secondary command buffers can be recorded in multiple threads and finally, all concurrently recorded secondary command buffers can be submitted within a primary command buffer to the GPU at once. However, every thread needs its own command pool to allocate command buffers from. [14] [16]

3.8 Render Passes

Render passes in Vulkan define the scope of one or multiple rendering commands as a `VkRenderPass` object which consists of a set of attachments and subpasses with dependencies between these subpasses. A render pass specifies how attachments are used by its subpasses. By means of render passes the Vulkan driver is able to setting up its hardware in advance, such that the rendering operations are executed under optimal conditions. The attachments of a `VkRenderPass` object specify all attachments that are used in at least one of the subpasses as a framebuffer attachment (input, color, depth/stencil). Further, the initial and final image layouts of each attachment must be specified along with the image format and number of samples. Image formats and layouts in Vulkan are covered later in section 3.14. By specifying the initial and final layouts of the attachments, the driver implicitly executes image layout transitions on the attachments at the beginning and end of the render pass. These transitions are necessary to have the attachments in the right format at the start of the render pass resp. to set up the layouts of the attachments for purposes later when the render pass has finished. A render pass must contain at least one subpass. A subpass holds references of the attachments which are intended to be used by the subpass. Further, the usage of each attachment must be specified (input, color, depth/stencil) together with its image layout. Again the driver implicitly performs an image layout transition to the specified layout before entering the subpass execution. Subpass dependencies synchronize the access to attachments between subpasses resp. between a subpass and the entry or exit of the render pass. Hence, at least two subpass dependencies must be specified. As an example, if one subpass is writing to an attachment while the next subpass wants to read from the same attachment, a subpass dependency synchronizes the two subpasses by blocking the read access for the second subpass until the first subpass has finished writing to the attachment. [16] [21]

3.9 Framebuffers

Vulkan framebuffers, represented by `VkFrameBuffer` objects, are closely connected to render passes. The attachments used in a render pass are enveloped as a set of `VkImageView` (3.14) references in a `VkFrameBuffer` object. Since a framebuffer always acts in conjunction with a specific render pass, a `VkRenderPass` reference must be specified at `VkFrameBuffer` creation. [16]

3.10 Pipelines

A Vulkan pipeline specifes what the GPU is claimed/demanded to do in a render or compute operation by encompassing a programmable shaderpipeline and a set of configurable function states along with a render pass reference in a `VkPipeline` object. By means of a pipeline object, the GPU is aware of almost all configurations of a render or compute operation in advance. `VkPipeline` objects are immutable except for some few configurable dynamic function states. If a shader or a state in the fixed function configuration needs to be marginally switched, a separate `VkPipeline` object must be created. Hence, many pipeline objects need to be created in advance for all different combinations of shaders and configurations. Pipelines are separated into graphics pipelines and compute pipelines. Graphics pipelines represent common draw operations with vertex input assembler, graphics shader pipelines and rasterization while compute pipelines define offscreen compute operations with a single compute shader module without vertex input assembler or rasterization. [14] [16]

The programmable shaderpipeline of graphics or compute pipeline objects is similar to the OpenGL shaderpipeline with the shader stages vertex, tessellation, geometry, fragment and compute. The difference is that Vulkan uses precompiled shader bytecode instead of GLSL shader files as OpenGL does. Vulkan shader stages with its precompiled bytecode are represented as `VkShaderModule` objects. At `VkPipeline` creation multiple (or one for compute pipelines) `VkShaderModule` objects are specified as a shaderpipeline. The configurable function state consists of a set of pipeline configuration state objects. Every configuration state object must be explicitly specified even if the programmer doesn't take care about some of them. These configuration state objects are listed and briefly explained below:

- **Vertex Input State** specifies the layout and format of the vertex buffer.
- **Vertex Input Assembly State** defines the topology (points, lines, triangles etc.) of the vertices.
- **Rasterization State** adjusts the configuration of the rasterizer auch as the polygon cullmode.
- **Color Blend State** specifies the color and alpha blending functions to be applied.
- **Multisample State** sets the number of samples together with some additional multisampling configurations.
- **Viewport State** sets up the viewport size and offset.
- **Depth and Stencil State** enables or diables depth and stencil test together with compare function parameters.
- **Tessellation State** defines the control point number for tessellation patches (Ignored if tessellation is disabled).

- **Dynamic State** enables dynamic state functions. Even though pipeline objects must be configured in advance and cannot be modified afterwards, a small set of states can be dynamically changed during command buffer execution. These modifiable states must be explicitly enabled within the dynamic state object.

In addition to the configuration state objects a pipeline layout must be specified at `VkPipeline` creation. The pipeline layout contains a set of descriptor set layouts and optionally informations about a push constants block. Descriptors and push constants are covered in the following sections 3.11 and 3.12. [22] [16]

Figure 3.5 illustrates the Vulkan pipeline with its programmable shader stages and configureable functions stages as a block diagram.

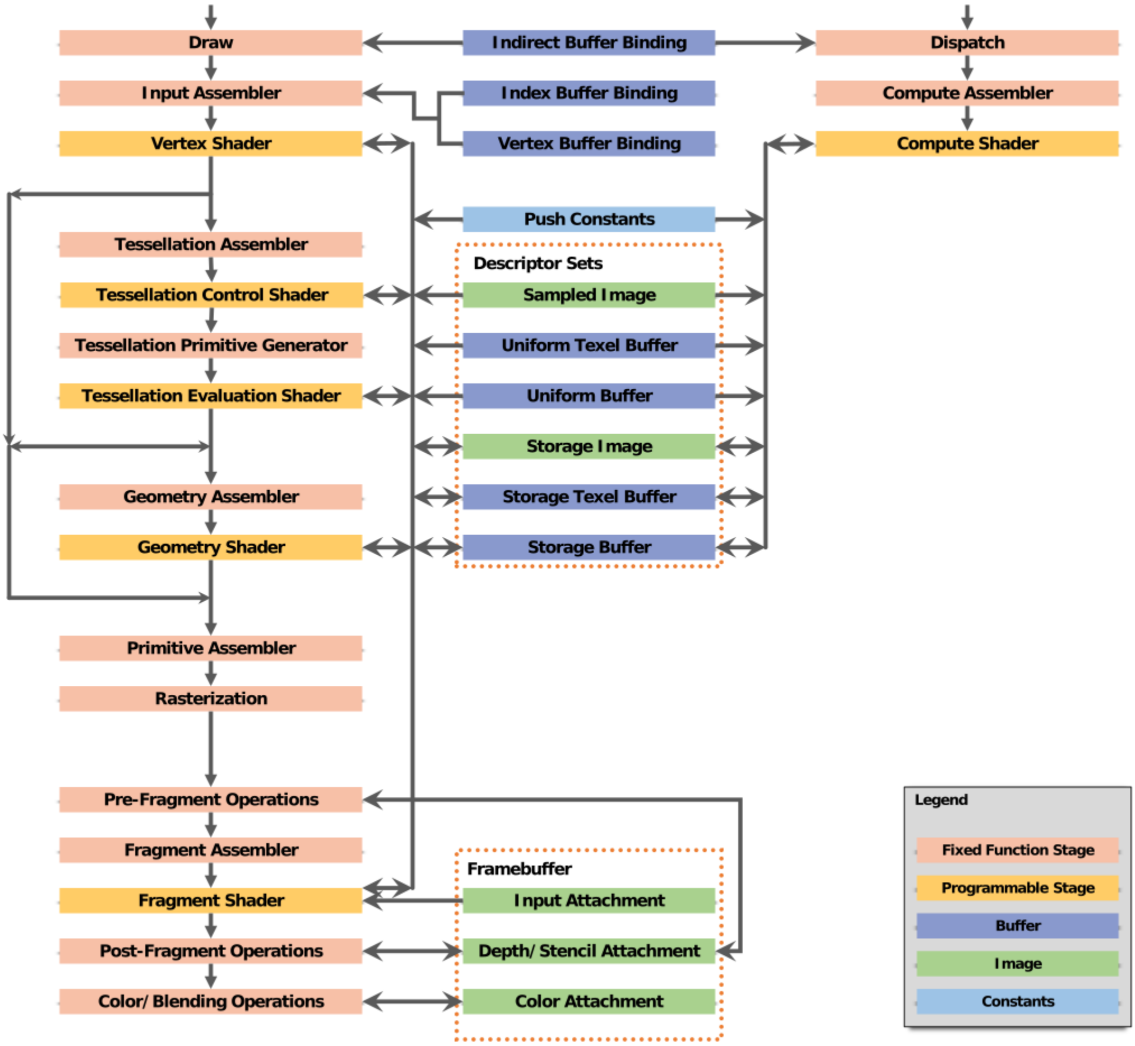


Figure 3.5: Vulkan Pipeline Block Diagram [23]

The block diagram shows the Vulkan graphics/compute pipeline as a block diagram where one block element represents a specific pipeline stage or resource. The left block flow diagram from `Draw` to `Color/Blending Operations` portrays the graphics pipeline while the right block flow diagram from `Dispatch` to `Compute Shader` illustrates the compute pipeline. An indirect buffer is an optional resource for the fixed function stages `Draw` resp. `Dispatch`. Indirect buffers are needed for indirect draw and dispatch operations, where draw resp. dispatch parameters are specified via buffers. As mentioned, graphics pipelines possess an `Input Assembler` with a vertex buffer and/or index buffer as resource. Descriptor sets (3.11) and push constants (3.12) can be used as resources in all programmable stages of graphics and compute pipelines. The graphics pipeline is capable to access framebuffer input attachments at the programmable `Fragment Shader` stage. The `Pre-Fragment Operations` stage verifies the primitive fragments of the current pipeline against existing values in framebuffer's `Depth/Stencil Attachment` (depth/stencil test) and discards or passes a fragment depending on the specified configuration of the pipeline's `Depth and Stencil State`. In the `Post-Fragment Operatins` stage the depth and/or stencil values are written to the `Depth/Stencil Attachment`. Colors are written into one or multiple `Color Attachments` at the `Color/Blending Operations` stage. [23] [14]

3.11 Descriptors

Vulkan descriptors represent shader resources. A descriptor consists of a descriptor set and a descriptor set layout. The descriptor set layout specifies which type of resources (like buffers, images or samplers) are contained in the related descriptor set. The descriptor set holds references to the actual resource data. A shaderpipeline is capable of using multiple descriptor sets by specifying the descriptor sets in the programmable shader stages with indices in the preserved order as specified in the related pipeline layout. The descriptor set layouts of the descriptor sets used by the shaderpipeline must be added to the related pipeline layout. [16] [24]

Figure 3.6 illustrates the relation of descriptor set layouts and pipeline layouts.

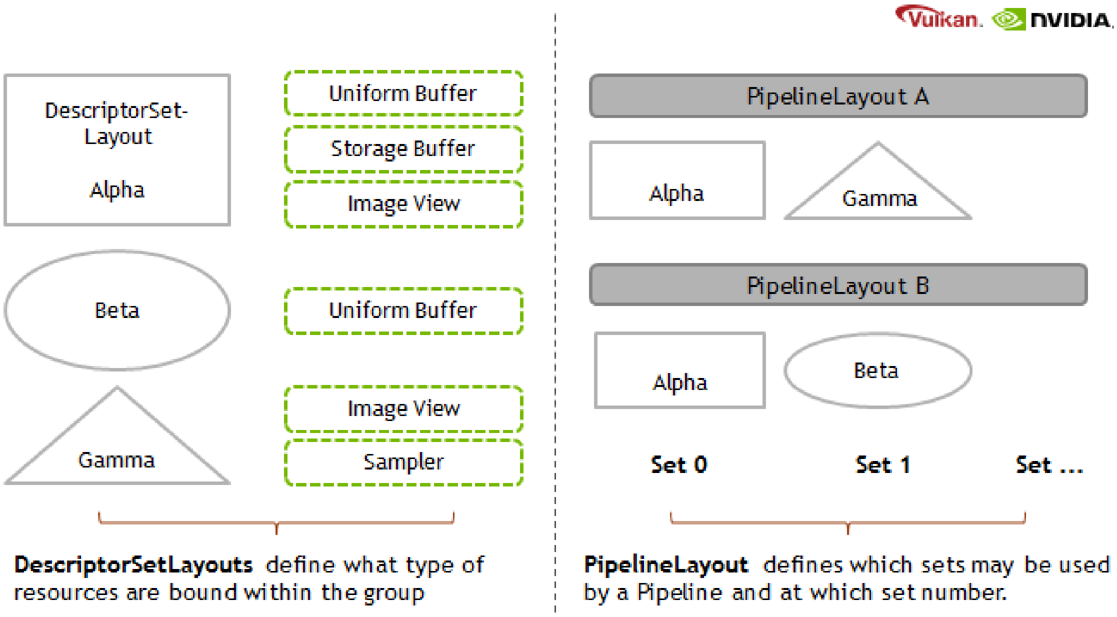


Figure 3.6: Descriptor Set Layout and Pipeline Layout [24]
The »Alpha« descripor set layout specifies the usage of a uniform buffer, a storage buffer and an image view in the related descriptor set. The »Beta« descriptor set contains a uniform buffer. »PipelineLayout B« specifies descriptor set layout »Alpha« and »Beta« which implies the usage of two uniform buffers, one storage buffer and one image view by the related pipeline.

3.12 Push Constants

Push constants are small blocks of raw data that can be passed to shaders in a very straighforward manner. Unlike uniform and storage buffers push constants are provided directly to shaders by the single command `vkCmdPushConstants` during command buffer recording without encapsulating the data in a `VkBuffer` object and referencing it in a descriptor set. However, the maximum data block size of push constants is highly limited compared to uniform or storage buffers, which can hold megabytes of data. Vulkan drivers force GPUs to offer push constants with a size of at least 128 bytes.

The Titan Xp supports a maximum push constants size of 256 byte. In contrast to uniform buffers, the push constants data cannot be altered after the command buffer was recorded. Nevertheless, if a command buffer is rerecorded, updating data with push constants is more performant than updating uniform buffers. [16] [24]

3.13 Buffers

Buffers in Vulkan are linear memory areas for arbitrary data that is accessible by the GPU and represented by `VkBuffer` objects. As mentioned in section 3.4.3, Vulkan differentiates between three kinds of physical memory. Allocating buffers on different kinds of memory exposes different limitations and advantages for the usage of these buffers. Buffers allocated on discrete video memory (device local) are accessed much faster by the GPU than buffers in system memory (host local). However, the host cannot access discrete video memory directly and hence the host is not able to write to it. To circumvent this limitation, the host writes the data to a buffer in a host visible memory, also called staging buffer in this context, and afterwards the device copies the data in the host visible memory with the `vkCmdCopyBuffer` to a preallocated device local memory buffer. [16]

`VkBuffer` objects are used as vertex- and index-buffers for the input assembler of graphics pipelines. Further `VkBuffer`s serve as read and write resources for shaders as uniform and storage buffers. Uniform and storage buffers can be much larger than push constants. The Titan Xp offers a maximum uniform buffer size of 65,536 bytes (= 0.065536 megabytes) and a maximum storage buffer size of around 4,295 megabytes. Uniform buffers provide read access to shaders and can be updated by the host directly if the buffers are located on host visible memory, otherwise a staging buffer is necessary for updating a uniform buffer. Storage buffers offer read and write operations to shaders and are commonly of much larger size than uniform buffers. [16] [14]

3.14 Images

Images in Vulkan are represented by `VkImage` objects. A `VkImage` object defines the size, format and layout of an image. Images are used as render targets by graphics pipelines. Before an image is capable to be used as a render target, the related `VkImageView` of the image must be created. `VkImageView` objects specify an explicit part of the image's memory together with the format to properly read the image. The `VkImageView` object can be referenced as an attachment of a framebuffer what makes the image a render target. [16] [14]

Shaders can read/sample from images or write to them. Storage image descriptor types offer direct texel read/write access to shaders by referencing the `VkImageView` object of an image in a descriptor set. With a `VkSampler` reference along with a `VkImageView` reference in a descriptor set as a combined image sampler descriptor type, shaders can sample from an image, where the `VkSampler` object specifies sampling parameters to control the filtering and transformations of the retrieved color. [16]

An important property of images in Vulkan are image layouts. Through image layouts the Vulkan driver can optimize the performance for specific image memory utilization. Image layouts must be explicitly specified by the programmer as a proper enumeration with the prefix `VK_IMAGE_LAYOUT`. Further, the programmer is responsible for synchronization of image layout transitions via pipeline barriers or subpass dependencies. Synchronization with pipeline barriers is explained in the next section. For instance, if an image should be used as a framebuffer color attachment, it must have the layout `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. If the image is subsecuently used as a storage image in a compute shader, the image layout must be transitioned to `VK_IMAGE_LAYOUT_GENERAL`. Such a use case realized as a subpass dependency is demonstrated in chapter 5.2 in figure 5.5. [16] [14]

3.15 Synchronization

Since Vulkan is a low-level hardware abstraction and does not provide any synchronization mechanism by default, the programmer must handle any synchronization explicitly. The command `vkQueueWaitIdle` waits for a specific queue to become idle and is a very straightforward method to perform synchronization, but in most cases queues rarely become idle. Further, it is often necessary to synchronize commands during queue execution. For convenient synchronization, Vulkan offers four distinct synchronization objects, which are listed below. [16]

3.15.1 Fences

Fences provide synchronization between host and device. A `VkFence` reference can be submitted together with a command buffer to a queue. The `VkFence` object signals to the host the completion of the command buffer execution. [16]

3.15.2 Events

Events (`VkEvent`) provide finer-grained snychronization than fences between host and device. While fences are only capable to signal the completion of a command buffer execution, events can be used to synchronize the progress of a command buffer execution by placing a `vkCmdSetEvent` command at a specific position in the command buffer. Further, by placing a `vkCmdWaitEvents` the command buffer execution can wait on one ore more events to become signalled (by the host or the device). [16]

3.15.3 Semaphores

Semaphores (`VkSemaphore`) offer a synchronization mechanism between command buffer submissions within a single queue and across multiple queues. Even though command buffer submissions to a single queue are executed in the preserved order they were submitted, GPUs parallelize the process of batches of command buffer executions as much as possible. With semaphores, the processing of one or multiple command buffers can be postponed at any specific pipeline stage until another command buffer (or batch of command buffers) has finished or reached a specific pipeline stage at the same or any other queue. [16]

3.15.4 Barriers

Barriers are the most extensive synchronization objects in Vulkan and provide execution and memory synchronization between sets of commands within a single command buffer execution. Barriers can be placed within a command buffer with the `vkCmdPipelineBarrier` command. As an example, if one command execution depends on the completion or a specific progress level of another command execution, a proper `vkCmdPipelineBarrier` between these two commands ensures synchronization based on the execution and/or memory dependencies specified in `vkCmdPipelineBarrier`. A memory dependency snychronizes read and write access to images or buffers while an execution dependency ensures a specific sequence of command executions. As mentioned in the section about render passes (3.8), the subpass dependencies and image layout transitions can be explicitly specified to induce the driver doing implicit synchronizations. These synchronizations are nothing more than barriers which are implicitly placed by the driver. [16] [25]

Figures 3.7 and 3.8 show two different use case examples of barriers.

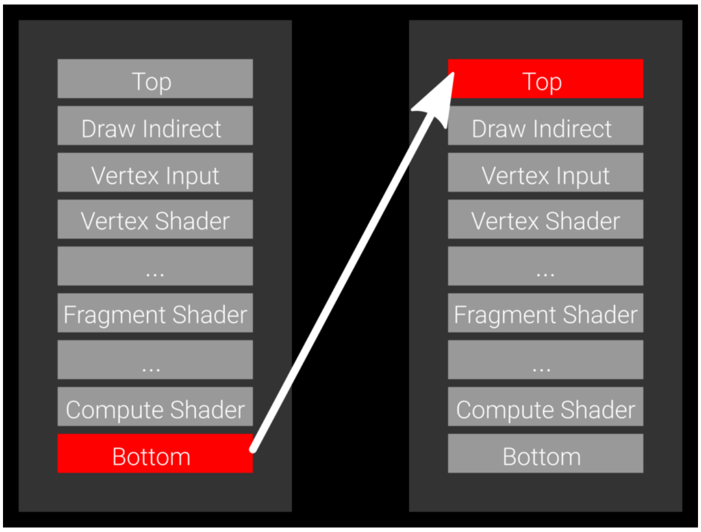


Figure 3.7: Slow Barrier Example [25]

Figure 3.7 shows two pipelines where the right pipeline is waiting on the left pipeline via a pipeline barrier. The pipeline barrier is specified with bottom stage (`VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`) as source stage and top stage (`VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`) as destination stage, which means that the waiting pipeline starts its execution not until the left pipeline has finished its whole processing.

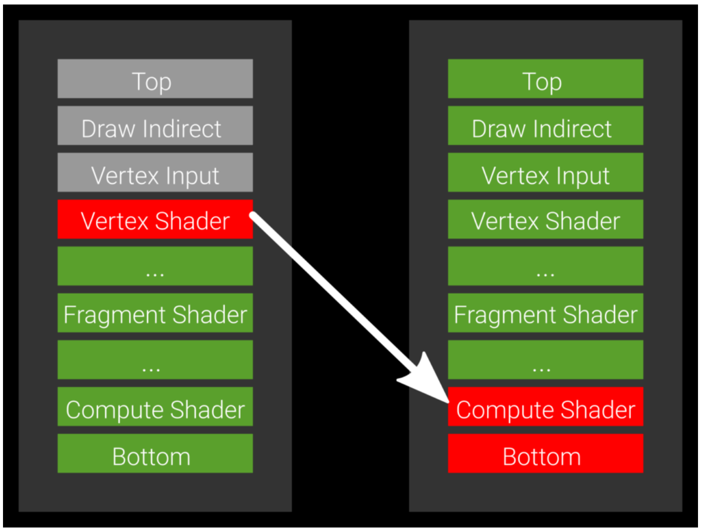


Figure 3.8: Optimal Barrier Example [25]

Figure 3.8 shows similar to the previous figure two pipelines where the right one is waiting on the left one. The pipeline barrier is specified with vertex shader (`VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`) as source stage and compute shader (`VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT`) as destination stage, which means that the waiting pipeline continues its execution at the compute shader stage not until the left pipeline has finished its processing of the vertex shader stage. For instance, if the compute shader of the right pipeline wants to read from a resource the vertex shader of the left pipeline writes to, this would be an optimal barrier, since in this case it is not necessary to wait on subsequent pipeline stages to be finished.

3.16 SPIR-V Shaders

Since Vulkan drivers don't provide a high-level shader language compiler, all shader code must be compiled to SPIR-V files in advance in order to be used by a Vulkan application. SPIR-V is an intermediate low-level language used by Vulkan drivers for assemble shader programs on the GPU. Since SPIR-V is a low-level language, it is hard to write SPIR-V shader manually. To overcome this issue the VulkanSDK provides a tool for compiling high-level shader code (like GLSL) into SPIR-V shader files. [16] [5]

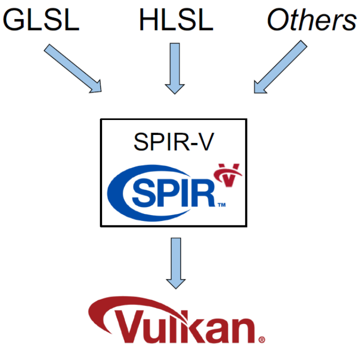


Figure 3.9: SPIR-V [26]

Chapter 4

The Case Study Scenario

This chapter outlines an abstract and theoretical perspective of the simulation scenario used in the case study and deployed by the developed Vulkan engine.

The case study simulation is rendered in terms of a successive image synthesis by the use of deferred shading along with antialiasing and post processing bloom. Entities of the simulation are:

- FFT generated ocean
- Atmosphere mapped onto a skydome
- Sun

The FFT generated ocean is based upon the GPGPU implementation design elaborated in the context of the research thesis *Realtime GPGPU FFT Ocean Water Simulation* along with the equation

$$h(n,m,t) = \frac{1}{N \cdot N} (-1)^n \sum_{k=0}^{N-1} \left[(-1)^m \sum_{l=0}^{N-1} \tilde{h}(k,l,t) \exp\left(i \frac{2\pi ml}{N} \right) \right] \exp\left(i \frac{2\pi nk}{N} \right) \tag{4.1}$$

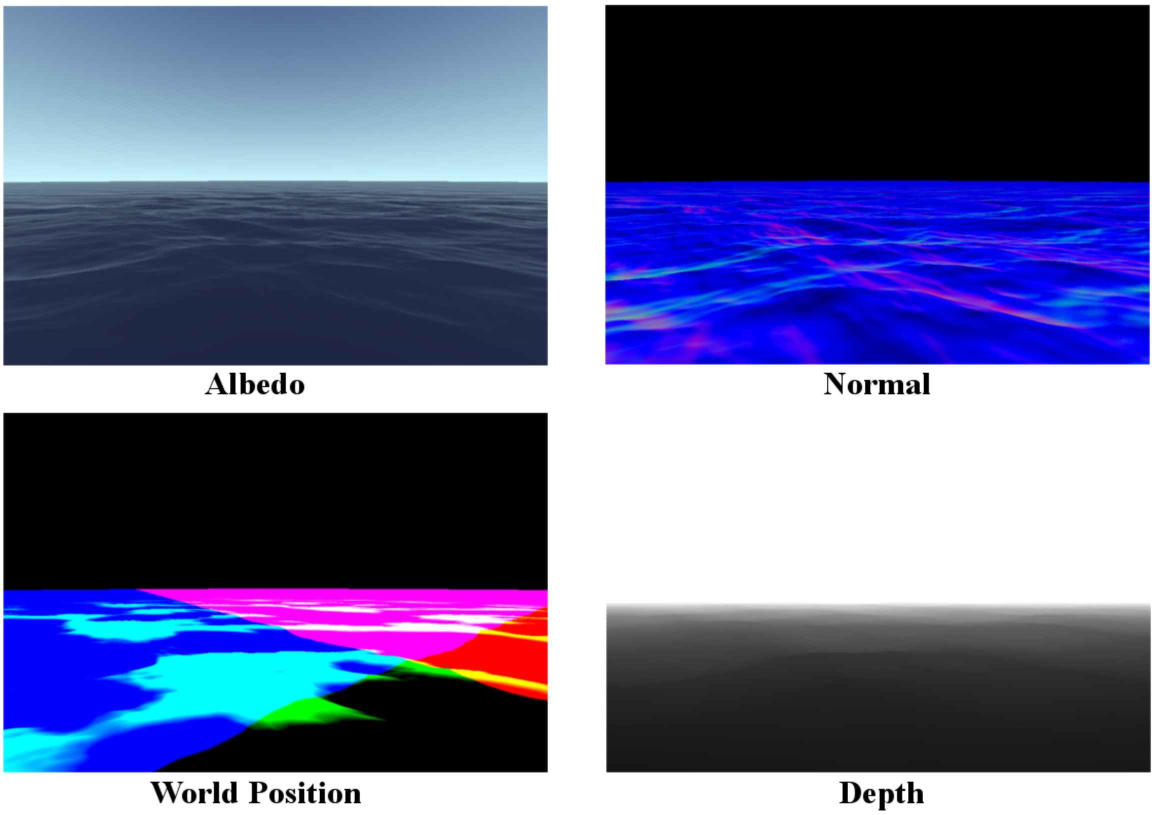
derived from the statistical, emperically-based oceanographic spatial spectrum of the ocean surface noted in the paper *Simulating Ocean Water* [2] [27].

The stages for successively synthesizing the presentation image are delineated in the following sections.

4.1 Deferred Shading with MSAA

The first stage of the image synthesis is deferred shading with multisample anitaliasing[1] (MSAA). Deferred shading separates scene lighting from geometry processing by applying light calculations in screen-space. The advantage of deferred shading is that lots of light sources in the scene will impair the performance trivially. In order to perform screen-space lighting, the scene geometry and color values are rendered by a first render pass with multiple render targets into dedicated images resp. framebuffer attachments. These images, called g-buffer, are processed by a subsequent render pass to obtain a lighted and shaded scene image. To apply MSAA along with deferred shading, the scene must be rendered with enabled hardware-antialiasing into a multisample g-buffer. [28]

The g-buffer and the depth buffer of an example scene is shown in figure 4.1.



Albedo

Normal

World Position

Depth

Figure 4.1: The top left albedo image buffer contains the raw color values of the scene. The further two elements of the g-buffer are normals and world positions of the scene geometries. The bottom right depth image is a linearized visualization of the framebuffer depth attachment. The depth buffer is needed for shadow mapping, which is not applied in the case study simulation though. Since no lighting is applied to the skydome, no normal and world position values are written to g-buffer for it.

[1] Pixel artifact reduction

A non-trivial problem with deferred shading is that multisampling must be explicitly considered during light calculations in screen-space to preserve hardware-antialiasing. Therefore, lighting must be calculated for each sample of a multisample pixel from the g-buffer with afterwards averaging the results of the lighted samples to obtain the final antialiased color value. To save performance it is convenient to consider all samples of a pixel in lighting calculations only on sharp edges where aliasing occurs. Hence, these edges with potential aliasing must be detected in order to apply the deferred shading render pass. In a separate render pass, the aliased pixels in the scene image are detected and stored in the red channel of a further image. These detected aliased pixels are called the sample coverage mask of the scene. The sample coverage mask is obtained by evaluating the g-buffer to find discontinuities between position samples of a pixel. Discontinuities are found by computing the distances between the sample position vectors. The distances are subsequently cumulated and checked against a predefined threshold. If the cumulated distances are above the threshold a potential aliased pixel is detected. To reduce the complexity of the detection of discontinuities the distances between the position samples are not computed as a complete graph $O(n^2)$ but rather as an arbitrary path $O(n)$. For more accuracy discontinuities between normals can be evaluated additionally, but this is omitted in this thesis case study simulation though. Better sample coverage accuracy results in a faster deferred shading (less false positives: when pixels are detected as not aliased, which would otherwise be detected as aliased) and in more qualitative deferred shading (less false negatives: when pixels are detected as aliased, which would otherwise be detected as not aliased). However, more accuracy leads to a more expensive sample coverage mask generation. [29] [28]

Figure 4.2 shows the image of the sample coverage mask to the related g-buffer from figure 4.1.

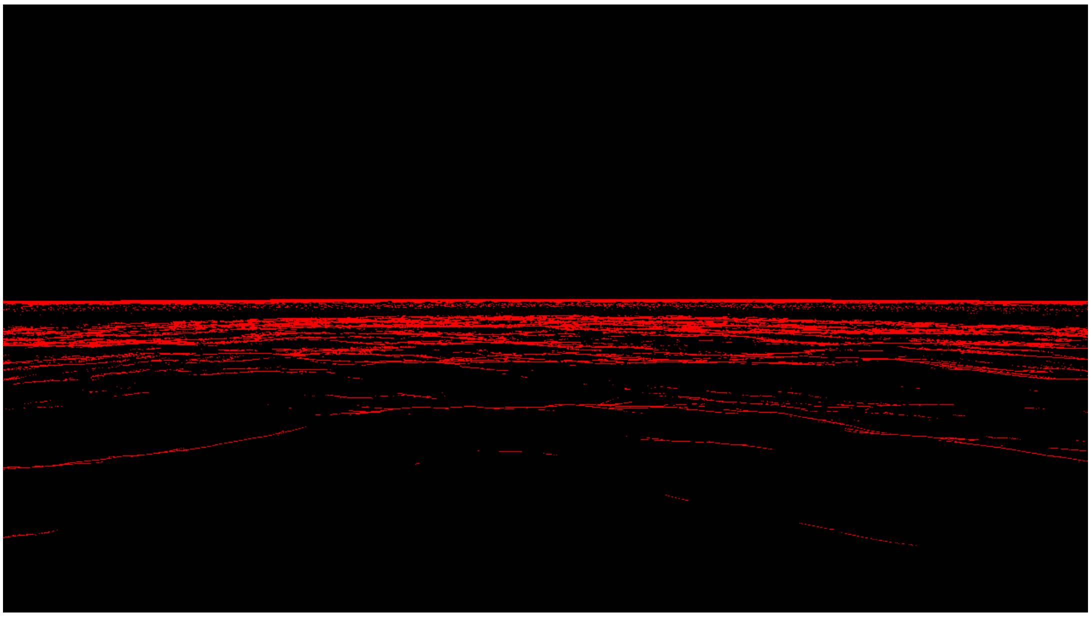


Figure 4.2: Sample Coverage Mask Image

The deferred lighted and shaded scene image from the g-buffer (4.1) and the sample coverage mask (4.2) with a directional sunlight is shown in figure 4.3. However, the sun itself is not visible yet and added to the scene in the transparency blending stage.

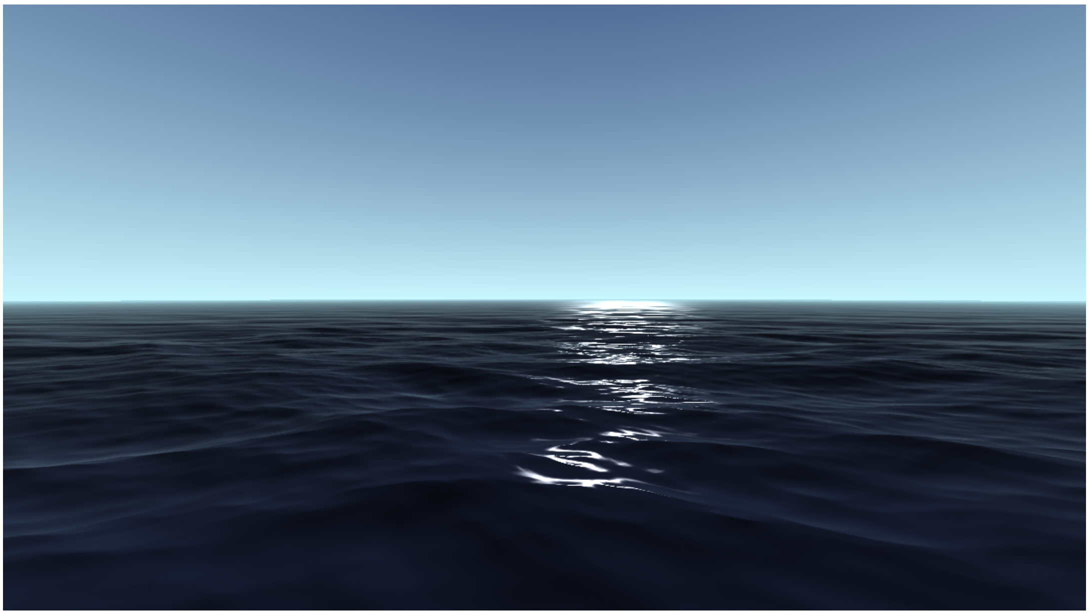


Figure 4.3: Deferred Lighting Scene

4.2 Transparency Blending

One problem that occurs along with deferred shading is that objects located behind transparent objects are not considered. A further problem is that MSAA along with deferred shading cannot properly applied to transparent objects and causes pixelations. One possible solution to overcome these issues is to render transparent scene objects into a separate image with forward lighting and blend it afterwards with an appropriate alpha blending function on top of the deferred lighted and shaded scene image. [28]

The case study simulation possesses one single transparent object, the sun, which is a simple point sprite object with the texture shown in figure 4.4. Even though the sun is always located between any opaque object and the atmospheric skydome, it is blended onto the scene in the same manner as any transparent object lying between opaque scene objects. The depth values of the opaque scene image and the transparent scene image are compared against each other and afterwards the transparent image is blended onto the opaque scene image by the color and alpha blending equations

$$RGB = R_sG_sB_s \cdot A_s + R_dG_dB_d \cdot (1 - A_d) \tag{4.2}$$

and

$$A = A_s \cdot 1 + A_d \cdot 0 \,, \tag{4.3}$$

where $R_sG_sB_sA_s$ represents the RGBA-channel of the transparent image (source) and $R_dG_dB_dA_d$ represents the RGBA-channel of the opaque image (destination). [14]

Figure 4.4: Sun texture created with the graphics editor *GIMP*.

Figure 4.5 shows the transparency scene image blended as a layer onto the deferred lighted and shaded ocean scene from 4.3. The transparency scene contains the sun object with world position derived from the sunlight direction applied in the deferred shading stage.

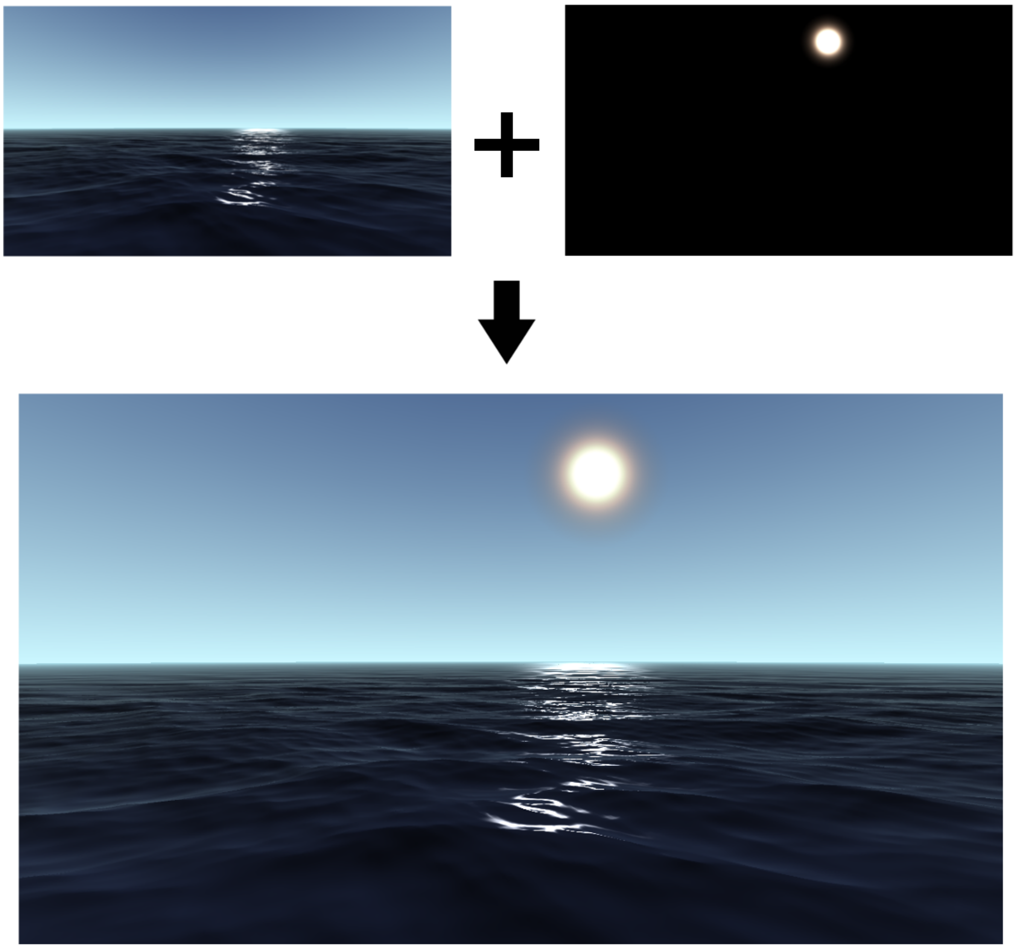


Figure 4.5: Transparency Scene Composition

4.3 FXAA

Fast approximated Antialiasing (FXAA) is a post-processing GPGPU antialiasing technique separated from multisample hardware-antialiasing. Since MSAA influences the perfomance often badly, FXAA achieves in some cases a sufficient antialiasing result with a much better performance than MSAA. However, for almost perfect antialiasing MSAA and FXAA can be applied together. The FXAA algorithm employed by the developed Vulkan engine is based on the Nvidia whitepaper *FXAA* [30].

Figure 4.6 shows the scene from 4.5 with 8x MSAA and FXAA. The case study simulation uses by configuration between 2x and 8x MSAA with enabled or disabled FXAA. Figure 4.7 shows the visual differences of various antialiasing configurations.

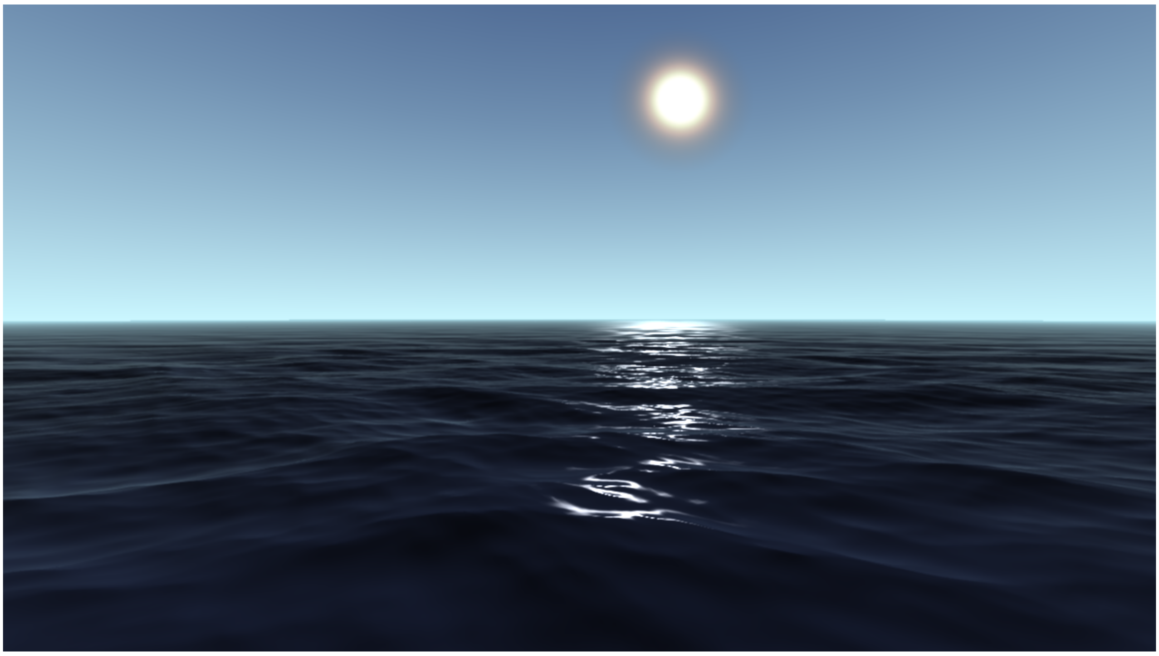
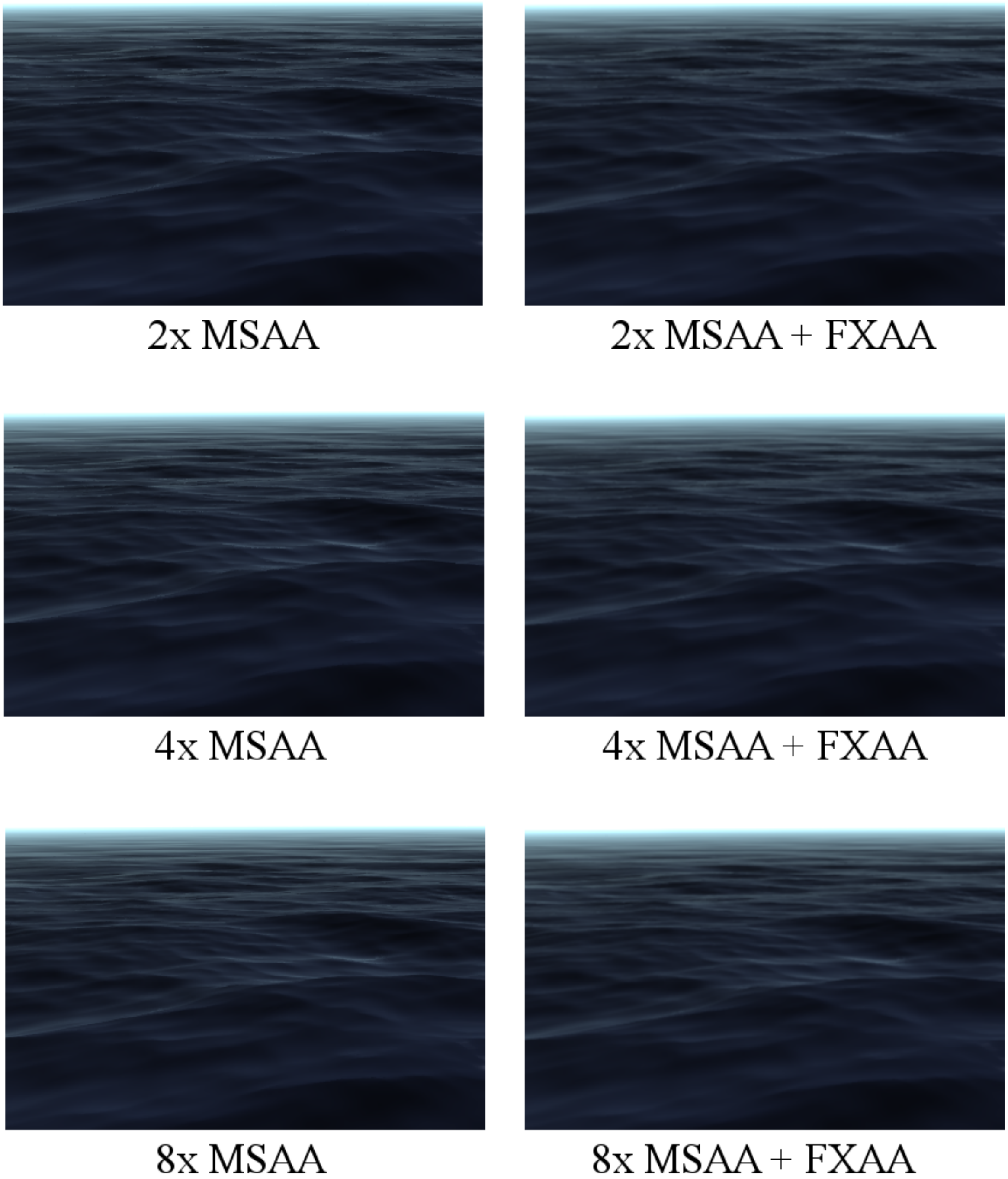


Figure 4.6: FXAA Scene Image



2x MSAA

2x MSAA + FXAA

4x MSAA

4x MSAA + FXAA

8x MSAA

8x MSAA + FXAA

Figure 4.7: The graphic shows a small section of the scene from figure 4.6 with ascending antialiasing qualities from top left to bottom right.

4.4 Bloom

After the FXAA is processed the image is ready to apply a chain of post-processing effects on it as the last stage of the scene image synthesis. The case study simulation uses a post-processing chain with a single element, a bloom effect. Bloom is an effect where bright light areas bleed over neighboured dark areas. The following presented approach of post-processing bloom generation is inspired by Nvidia's *GPU Gems - Chapter 21. Real-Time Glow* [31].

In order to apply bloom effect, a brightness image as shown in figure 4.8 is generated from the scene image with the RGB color's relative luminance

$$RGB_{luminance} = RGB_{color} \cdot \begin{bmatrix} 0.2126 \\ 0.7152 \\ 0.0722 \end{bmatrix} \tag{4.4}$$

as definced at *W3C - Relative luminance* [32]. Afterwards the luminance is squared and multiplied with the RGB color to obtain the RGB's brightness:

$$RGB_{brightness} = RGB_{color} \cdot (RGB_{luminance})^2 \ . \tag{4.5}$$

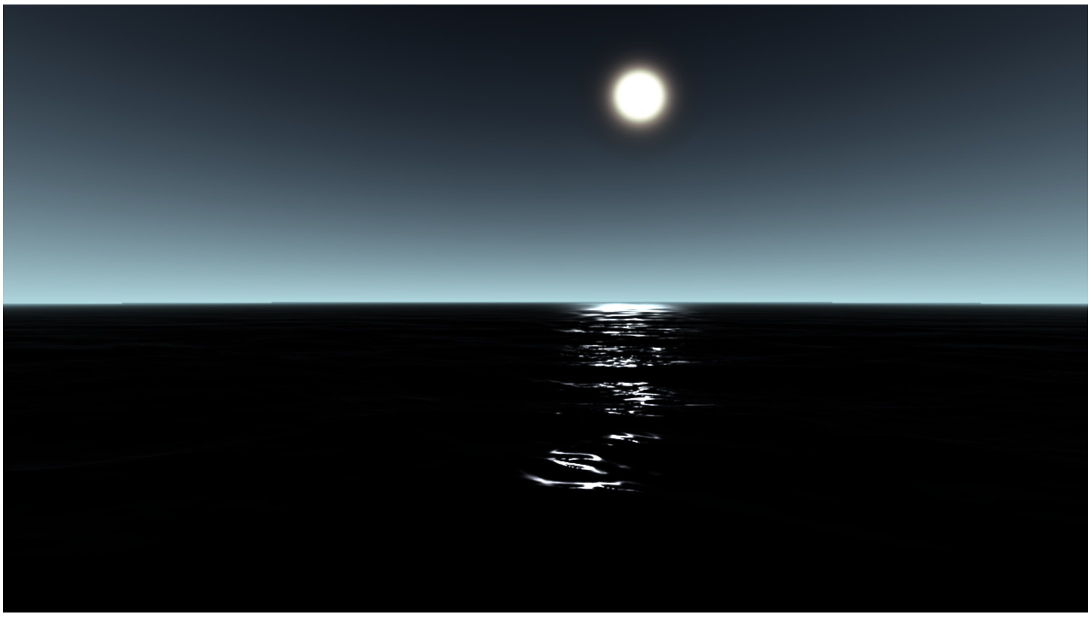
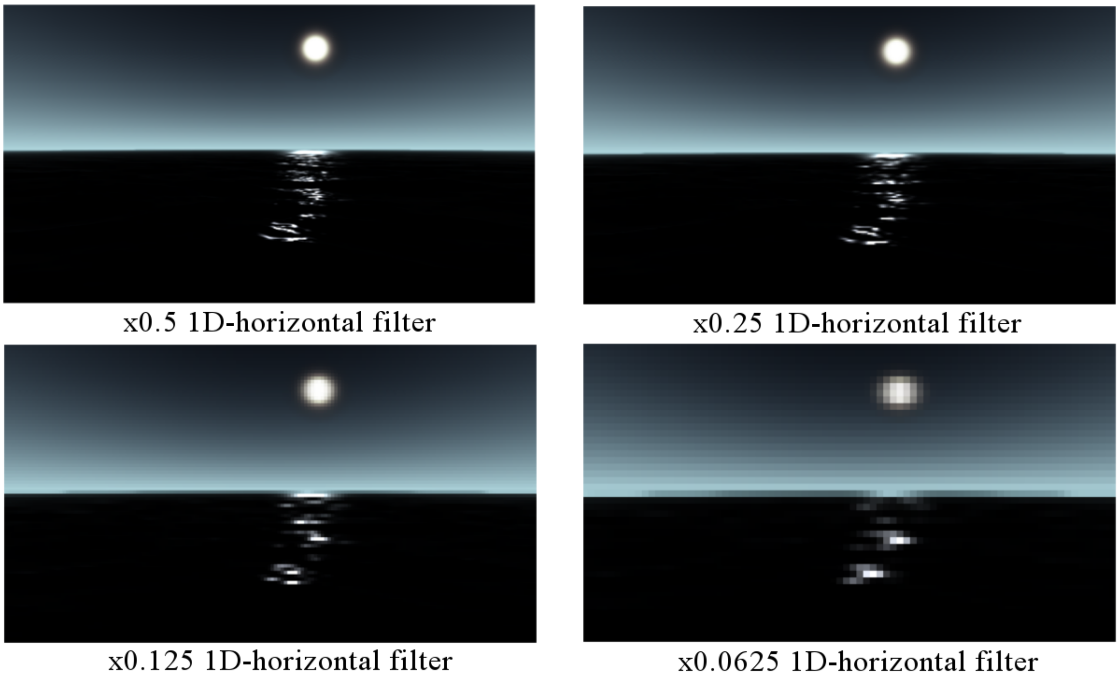


Figure 4.8: Brightness Scene Image

A *Gaussian* blur filter with $kernelsize = 9$ and $\sigma = 1$ is applied to the brightness image with decreasing resolutions as a 1-dimensional Gaussian blur filter in first horizontal and subsequently vertical direction. Figure 4.9 shows the 1-dimensional horizontal gaussian filtered brightness images with decreasing resolutions (resolution of the original image divided by 2, 4, 8, 16).



x0.5 1D-horizontal filter

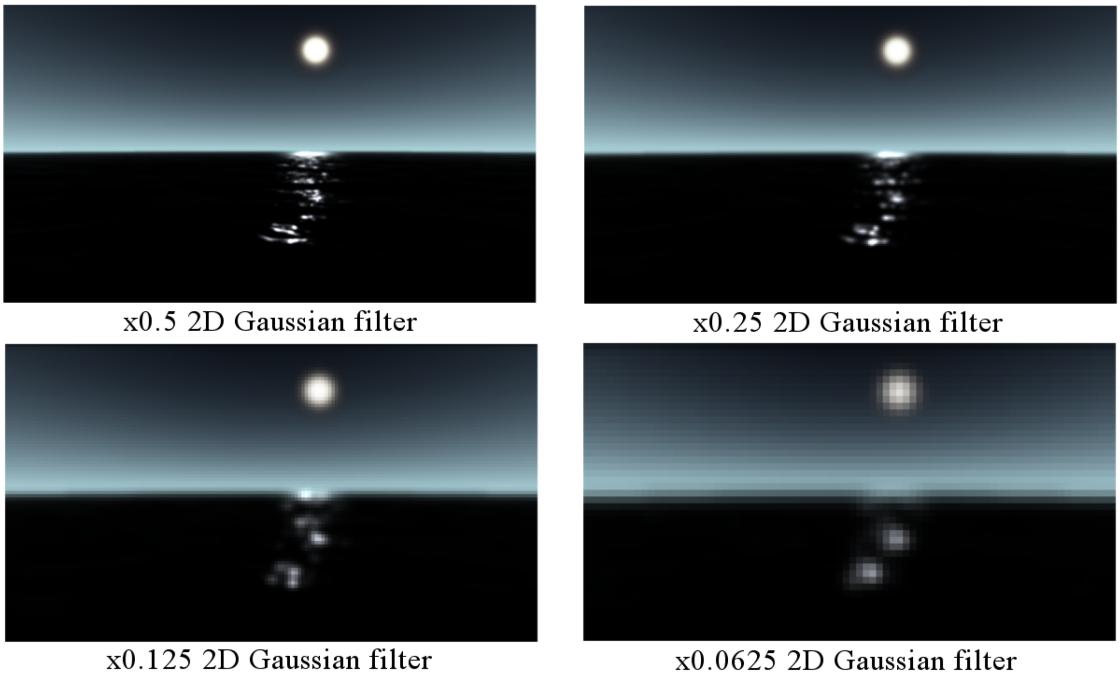
x0.25 1D-horizontal filter

x0.125 1D-horizontal filter

x0.0625 1D-horizontal filter

Figure 4.9: Horizontal Gaussian Bloom Blur

Afterwards a 1-dimensional vertical Gaussian filter is applied to the horizontal Gaussian filtered images to obtain the 2-dimensional Gaussian blurred brightness images shwon in figure 4.10. These Gaussian filtered brightness images are subsequently additively blended to obtain the bloom scene image depicted in figure 4.11.



x0.5 2D Gaussian filter

x0.25 2D Gaussian filter

x0.125 2D Gaussian filter

x0.0625 2D Gaussian filter

Figure 4.10: Vertical Gaussian Bloom Blur

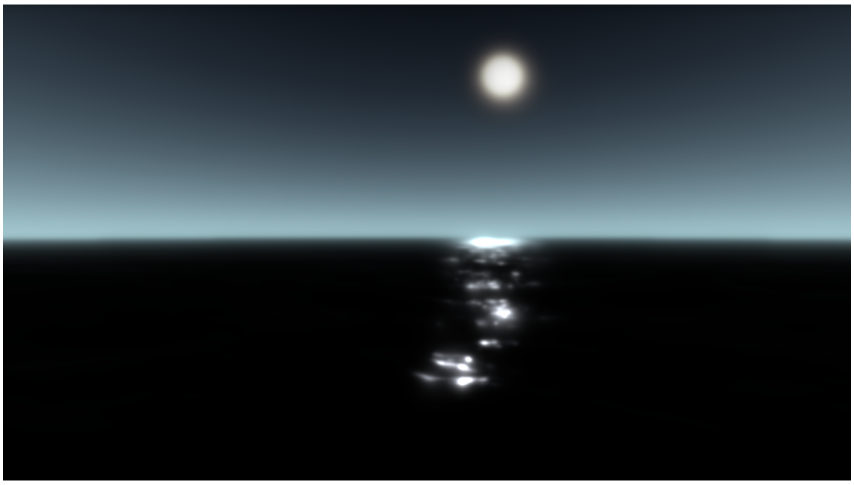


Figure 4.11: Bloom Scene Image

Finally the blurred bloom scene image is added to the scene image to obtain a bloom effect as depicted in figure 4.12.

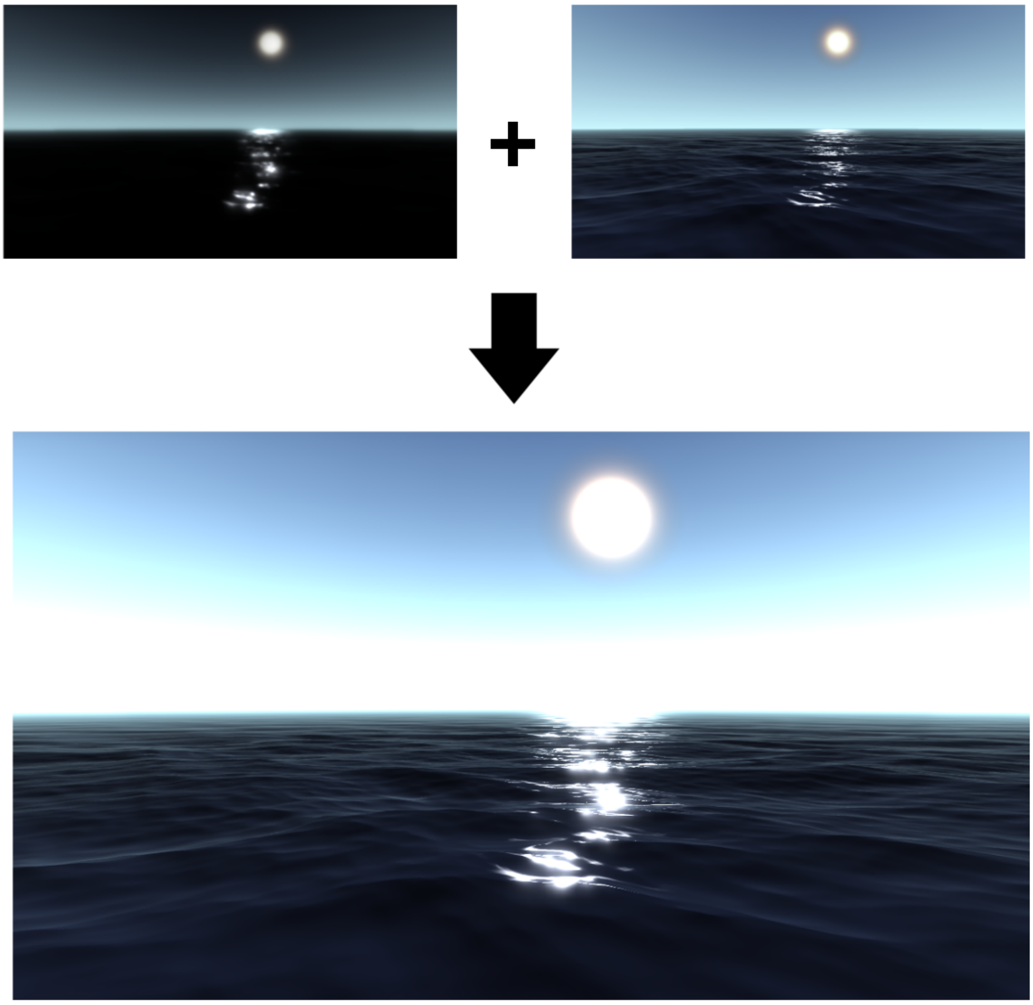


Figure 4.12: Bloom Effect Composition

4.5 Dynamic Panel Overlay

A dynamic panel overlay is rendered on top of the synthesized scene image. The overlay can be individually customized by a set of transparent or opaque color panels, text panels and image panels. The color and alpha blending functions for the overlay panels are the same equations as 4.2 and 4.3. Figure 4.13 shows a panel overlay that displays the fps[1] and the CPU load with the Vulkan logo [33].

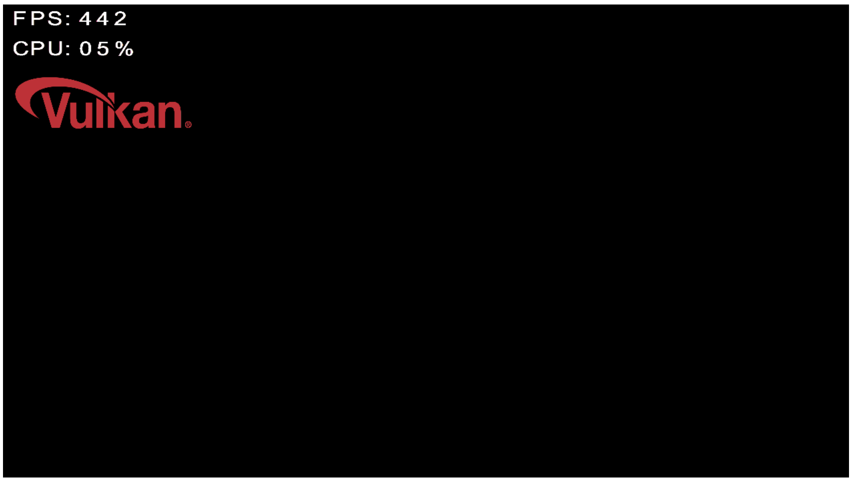


Figure 4.13: Panel Overlay

Blending the panel overlay onto the synthesized scene results in the image shown in figure 4.14.

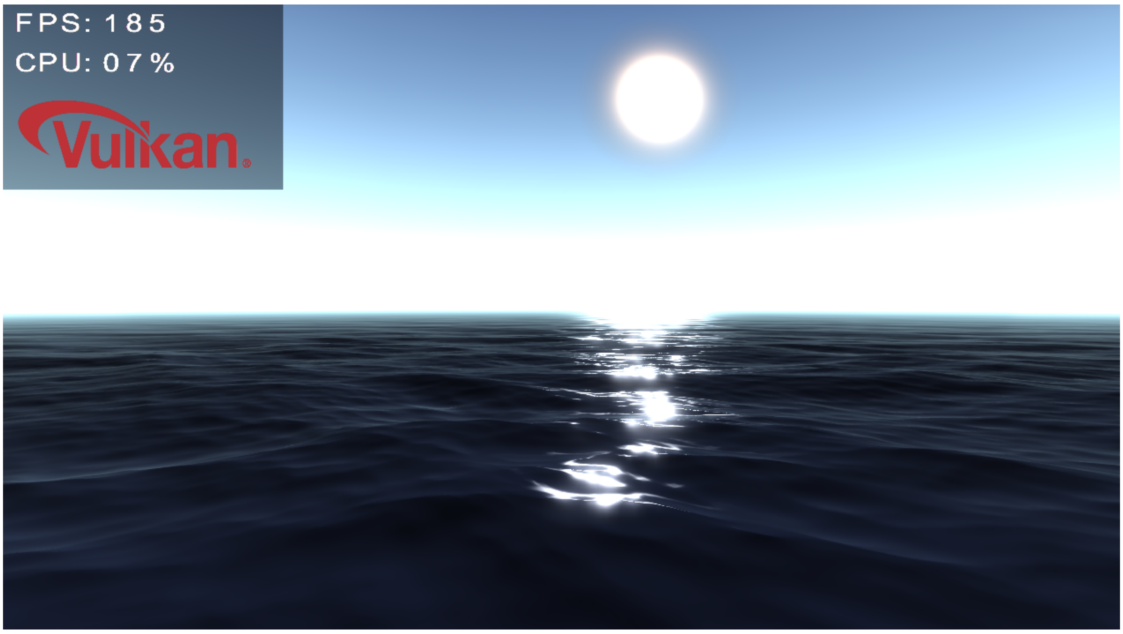


Figure 4.14: Blended Panel Overlay

[1] Frames per second

Chapter 5

Engine Design and Implementation

This chapter presents the design along with fundamental implementation details of the Vulkan engine elaborated in the scope of this thesis. The Vulkan engine is embedded into the Java graphics engine *Oreon Engine* [1] developed in the context of the research elaboration *Realtime GPGPU FFT Ocean Water Simulation* [2]. *Oreon Engine* already provides a proper OpenGL implementation of the simulation approach from last chapter which is used for the peformance comparison of Vulkan and OpenGL in chapter 6.

The process of the image synthesis presented in chapter 4 is subdivided into a batch of command buffers. The snychronization of critical sections between consecutive command buffers is done with semaphores and (rarely) fences. Pipeline barriers are used for synchronization between commands inside of command buffers

For instance, a critical section arises between g-buffer and sample coverage mask generation. The sample coverage mask generation must be postponed until the render operation of the g-buffer has finished. This synchronization is done with a semaphore, since g-buffer and sample coverage mask are generated by two separate consecutive command buffers. Further, the deferred shading pass cannot start until the sample coverage mask generation has finished. Since sample coverage and deferred shading passes are executed in one common command buffer, the synchronization is done with a pipeline barrier placed between sample coverage and deferred shading compute pipelines.

Figure 5.1 depicts a flow diagram of the command buffer execution sequence with its synchronization. The following sections give a deeper insight into the individual block elements shown in figure 5.1.

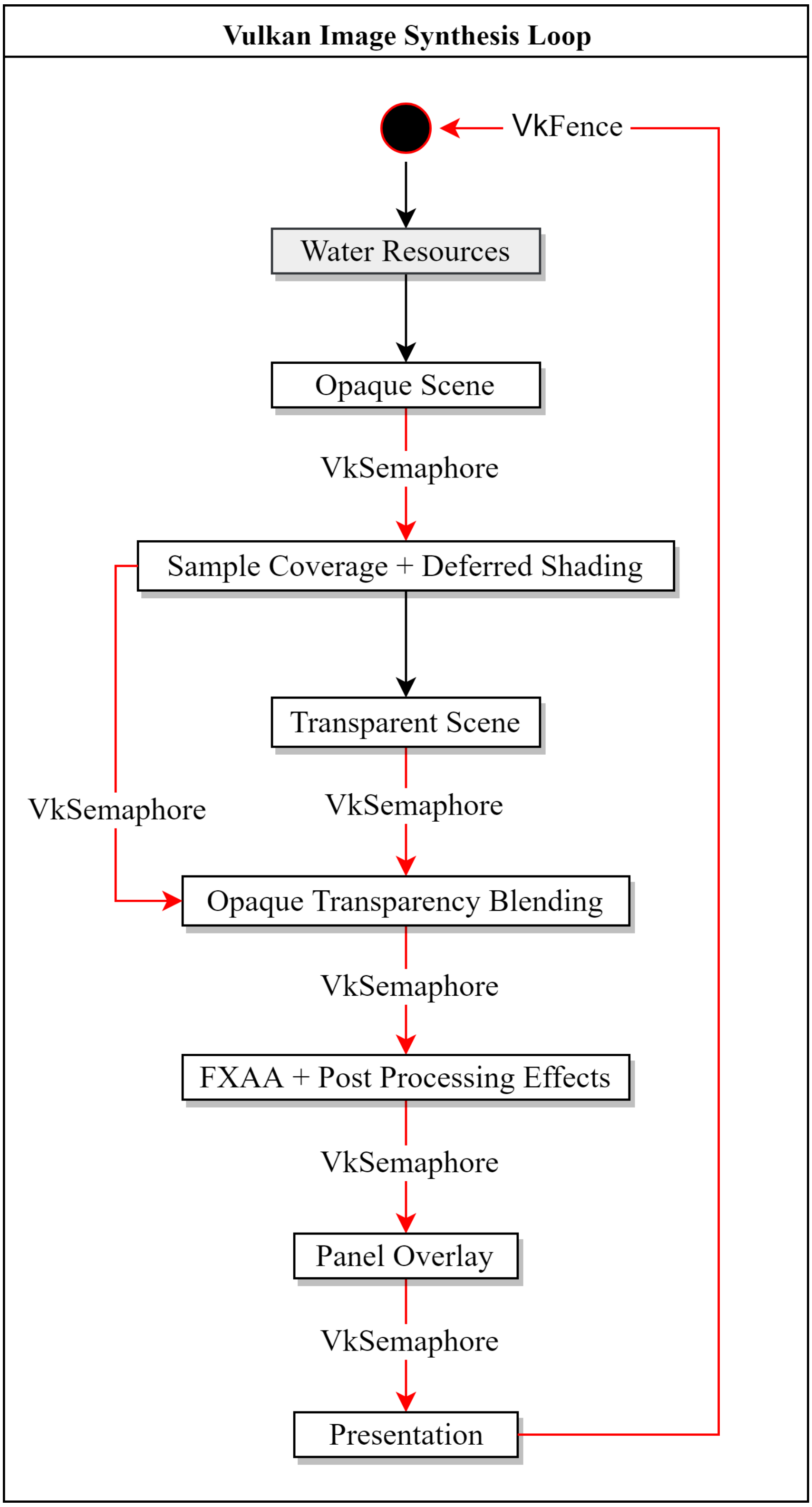


Figure 5.1: White blocks in the flow diagram represent command buffers. The grey block »Water Resources« conists of multiple synchronized command buffers. Execution flow along with synchronization is represented by red arrows with labels indicating which synchronization object is used. Black arrows represent execution flow without any synchronization.

5.1 Ocean Resources

The resources for the ocean such as displacement-, reflection- and refraction maps are generated in multiple steps with various command buffers. Figure 5.2 provides an overview of the command buffer execution sequence of the ocean resources generation. The particular block elements of figure 5.2 are described in the following subsections.

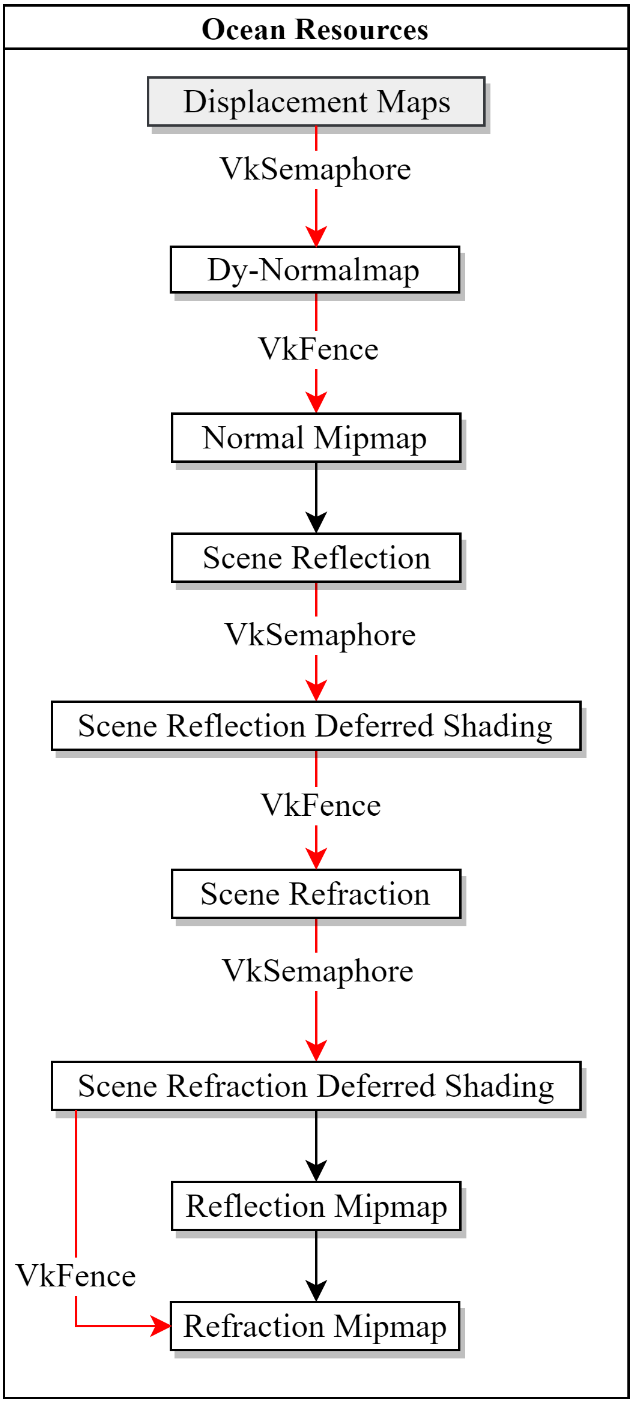


Figure 5.2: The flow diagram caption is similar to figure 5.1. The grey block »Displacement Maps« consists of multiple synchronized command buffers

5.1.1 Displacement Maps

The process of the displacement maps generation for the ocean surface is based on the approach from *Realtime GPGPU FFT Ocean Water Simulation* [2]. The FFT's for generating the displacement maps in x-, y- and z-direction are wrapped into one single command buffer. The $\tilde{h}(\mathbf{k}, t)$ components are previously generated by a separate command buffer as depicted in figure 5.3. For synchronization, the FFT command buffer waits on a `VkSemaphore` which is signalled after the $\tilde{h}(\mathbf{k}, t)$ command buffer execution has been finished.

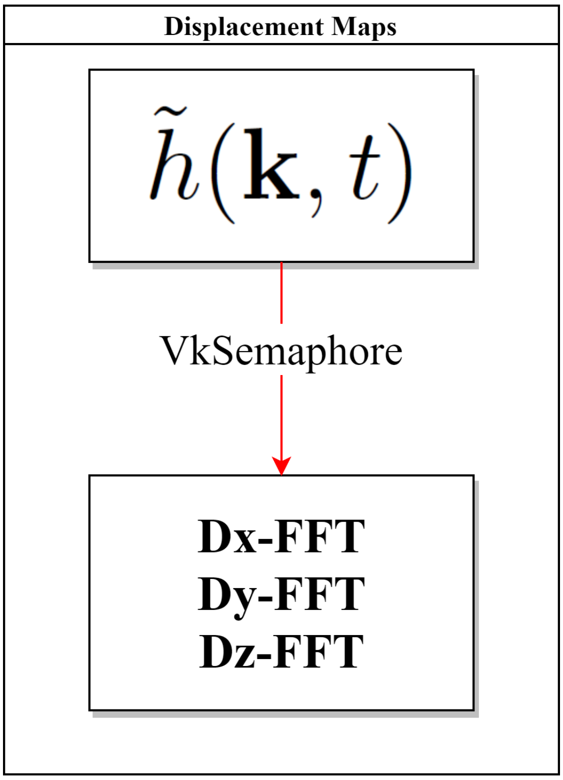


Figure 5.3: FFT Displacement Maps Generation

Listing 5.1 shows a pseudocode of the FFT command buffer record procedure with simplified input paramaters for the command buffer record functions, where "..." is a placeholder for trivial or irrelevant parameters.

```
// start command buffer record
vkBeginCommandBuffer(...);

// memory barrier structure
VkMemoryBarrier barrier = {
        .sType(VK_STRUCTURE_TYPE_MEMORY_BARRIER)
        .srcAccessMask(VK_ACCESS_SHADER_WRITE_BIT)
        .dstAccessMask(VK_ACCESS_SHADER_READ_BIT)};

// bind butterfly pipeline
vkCmdBindPipeline(..., VK_PIPELINE_BIND_POINT_COMPUTE, butterflyPipeline);
// horizontal butteflies
for i=0 to i<$log_2N$ do {
        // bind push constants
        vkCmdPushConstants(..., VK_SHADER_STAGE_COMPUTE_BIT, ...,
            horizontalPushConstants[i]);
        // dx butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dxButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // dy butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dyButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // dz butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dzButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // pipeline memory barrier
        vkCmdPipelineBarrier(..., VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
            VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, ..., barrier, ...);
}
// vertical butteflies
for i=0 to i<$log_2N$ do {
        // bind push constants
        vkCmdPushConstants(..., VK_SHADER_STAGE_COMPUTE_BIT, ...,
            verticalPushConstants[i]);
        // dx butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dxButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // dy butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dyButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // dz butterfly
        vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
            butterflyPipelineLayout, ..., dzButterflyDescriptorSet, ...);
        vkCmdDispatch(...);
        // pipeline memory barrier
        vkCmdPipelineBarrier(..., VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
            VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, ..., barrier, ...);
}
```

```
44  // bind inversion pipeline
45  vkCmdBindPipeline(..., VK_PIPELINE_BIND_POINT_COMPUTE, inversionPipeline);
46  // bind inversion push constants
47  vkCmdPushConstants(...,VK_SHADER_STAGE_COMPUTE_BIT, ...,
        inversionPushConstants);
48  // dx inversion
49  vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
        inversionPipelineLayout, ..., dxInversionDescriptorSet, ...);
50  vkCmdDispatch(...);
51  // dy inversion
52  vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
        inversionPipelineLayout, ..., dyInversionDescriptorSet, ...);
53  vkCmdDispatch(...);
54  // dz inversion
55  vkCmdBindDescriptorSets(..., VK_PIPELINE_BIND_POINT_COMPUTE,
        inversionPipelineLayout, ..., dzInversionDescriptorSet, ...);
56  vkCmdDispatch(...);
57
58  // finish command buffer record
59  vkEndCommandBuffer(...);
```

Listing 5.1: FFT Command Buffer Recording

The `vkBeginCommandBuffer` (l. 2) and `vkEndCommandBuffer` (l. 59) define start and end of the command buffer record procedure, respectively. The horizontal and vertical butterfly stages recordings are wrapped into two separate for-loops (ll. 13-27, ll. 29-43). The horizontal and vertical butterfly compute operations all use the same pipeline which is bound once with `vkCmdBindPipeline` (l. 11). `VK_PIPELINE_BIND_POINT_COMPUTE` indicates that the record command is dealing with a compute pipeline. The proper push constants block of the related stage and direction is bound with `vkCmdPushConstants` at the beginning of each horizontal and vertical for-loop iteration (l. 15, l. 31). The `VK_SHADER_STAGE_COMPUTE_BIT` enumeration configures the usage of the push constants in the compute shader stage of the pipeline. The push constants data blocks consists of the current stage index (i or j), the pingpong value (i+j mod 2) and a flag indicating horizontal or vertical butterflies (0 for horizontal, 1 for vertical). The dx-, dy- and dz-butterfly operations uses individual descriptor sets, which are bound with `vkCmdBindDescriptorSets` in order to execute the compute operation with `vkCmdDispatch`. The descriptor sets reference the corresponding pingpong images and the *twiddle* indices image as storage images (`VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`). After each horizontal and vertical for-loop iteration a pipeline barrier command is recorded (l. 26, l. 42) to prevent the usage of an image with the butterfly operation results in the subsequent stage prior to the butterfly operation executions of the previous stage has been finished. The `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT`s for both source and destination indicates that the pipeline executions before and after the pipeline barrier are synchronized both in the compute shader stage. The `VkMemoryBarrier` structure (l. 5) passed to both horizontal and vertical pipeline barrier commands specifies that the source (previous) pipeline is synchronized at shader write access (`VK_ACCESS_SHADER_WRITE_BIT`) while the destination (subsequent) pipeline is synchronized at shader read access (`VK_ACCESS_SHADER_READ_BIT`). The inversion operations use a further compute pipeline (l. 45) and a push constants block (l. 47). The inversion push constants consist of the resolution (N) and the final pingpong value. Additionally, the dx-, dy- and dz-inversion operations use separate descriptor sets. [14]

5.1.2 Dy-Normalmap and Mipmap Generation

It is sufficient to generate the normal map of the dy-displacement map, since only normals of the vertical displacement (dy-FFT) are token into account by generating the ocean surface normals for the normal buffer at g-buffer render stage. The normal map is created with a compute pipeline by a separate command buffer. The mipmap of the normal map is generated by a subsequent command buffer which is synchronized with a fence to the normal map generation.

5.1.3 Scene Reflection/Refraction and Deferred Shading

For proper water reflection and refraction, the scene is rendered two times into a g-buffer with half resolution of the display. Prior to the reflection render pass the scene is mirrored at the water surface. The reflection and refraction render command buffers are recorded by wrapping the scene objects as secondary command buffers into a single primary command buffer, respectively. The reflection/refraction scene rendering with a primary and multiple secondary command buffers is similar to the opaque and transparent scene g-buffer rendering. Detailed explanations on how to render the scene with a primary command buffer into a g-buffer as framebuffer attachments follow in section 5.2. The reflection and refraction g-buffers are used by subsequent deferred shading passes to obtain the water reflection and refraction maps. Finally, mipmaps of the reflection/refraction maps are generated by two separate command buffer executions similar to the normalmap mipmap generation in 5.1.2. However, the case study simulation does not contain any objects which are refracted by the water surface since skydome and sun are above the ocean surface and only reflected by the water.

5.2 Opaque Scene G-Buffer

The opaque scene g-buffer is rendered by the execution of a primary command buffer which wraps the opaque scene objects as a list of secondary command buffer references into a single command buffer. Secondary command buffers are created by specifying `VK_COMMAND_BUFFER_LEVEL_SECONDARY` instead `VK_COMMAND_BUFFER_LEVEL_PRIMARY` for primary command buffers at `VkCommandBuffer` creation. Further, the `vkBeginCommandBuffer` function called within the scope of a secondary command buffer recording needs the same framebuffer and render pass references (enveloped by a `VkCommandBufferInheritanceInfo` object together with some other optional configuration parameters which are not relevant here) as specified in `vkCmdBeginRenderPass` in the related primary command buffer recording procedure. [14]

Listing 5.2 shows a pseudocode of the primary command buffer record procedure for rendering the opaque scene g-buffer.

```
1  // start primary command buffer record
2  vkBeginCommandBuffer(...);
3  vkCmdBeginRenderPass(...,
       VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS);
4  vkCmdExecuteCommands(..., &SecondaryCommandBuffers);
5  vkCmdEndRenderPass(...);
6  // finish primary command buffer record
7  vkEndCommandBuffer(...);
```

Listing 5.2: Primary Command Buffer Record Procedure

Previously recorded secondary command buffers are specified as an array of references at the `vkCmdExecuteCommands` command (l. 4). The render pass used by the secondary command buffers is initiated with `vkCmdBeginRenderPass` by passing `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS` (l. 3) and terminated with `vkCmdEndRenderPass`. [14]

Listing 5.3 shows a record procedure of a secondary command buffer with graphics pipeline and indexed drawing.

```
// start secondary command buffer record
vkBeginCommandBuffer(...);
// bind push constants
vkCmdPushConstants(...);
// bind graphics pipeline
vkCmdBindPipeline(...);
// bind vertex buffer
vkCmdBindVertexBuffers(...);
// bind index buffer
vkCmdBindIndexBuffer(...);
// bind descriptor sets
vkCmdBindDescriptorSets(...);
// indexed drawing
vkCmdDrawIndexed(...);
// finish secondary command buffer record
vkEndCommandBuffer(...);
```

Listing 5.3: Secondary Command Buffer Record Procedure

The framebuffer with the g-buffer attachments is shown in figure 5.4 and the related render pass is graphically illustrated with a comprehensive explanation in figure 5.5.

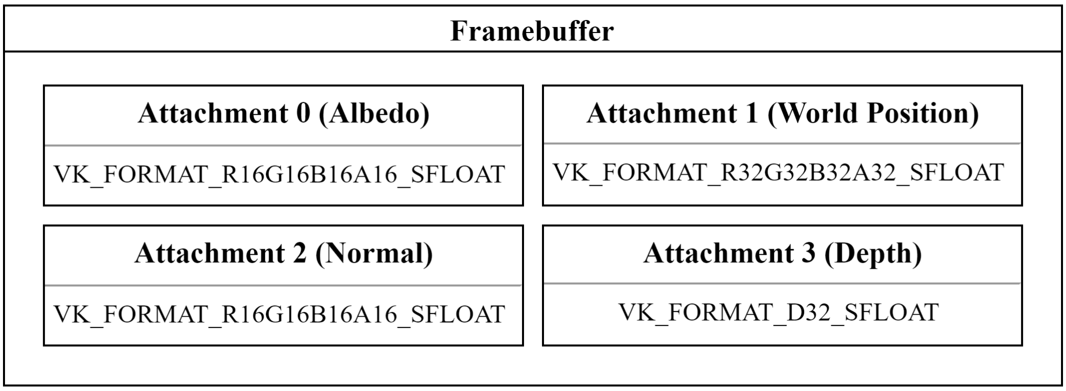
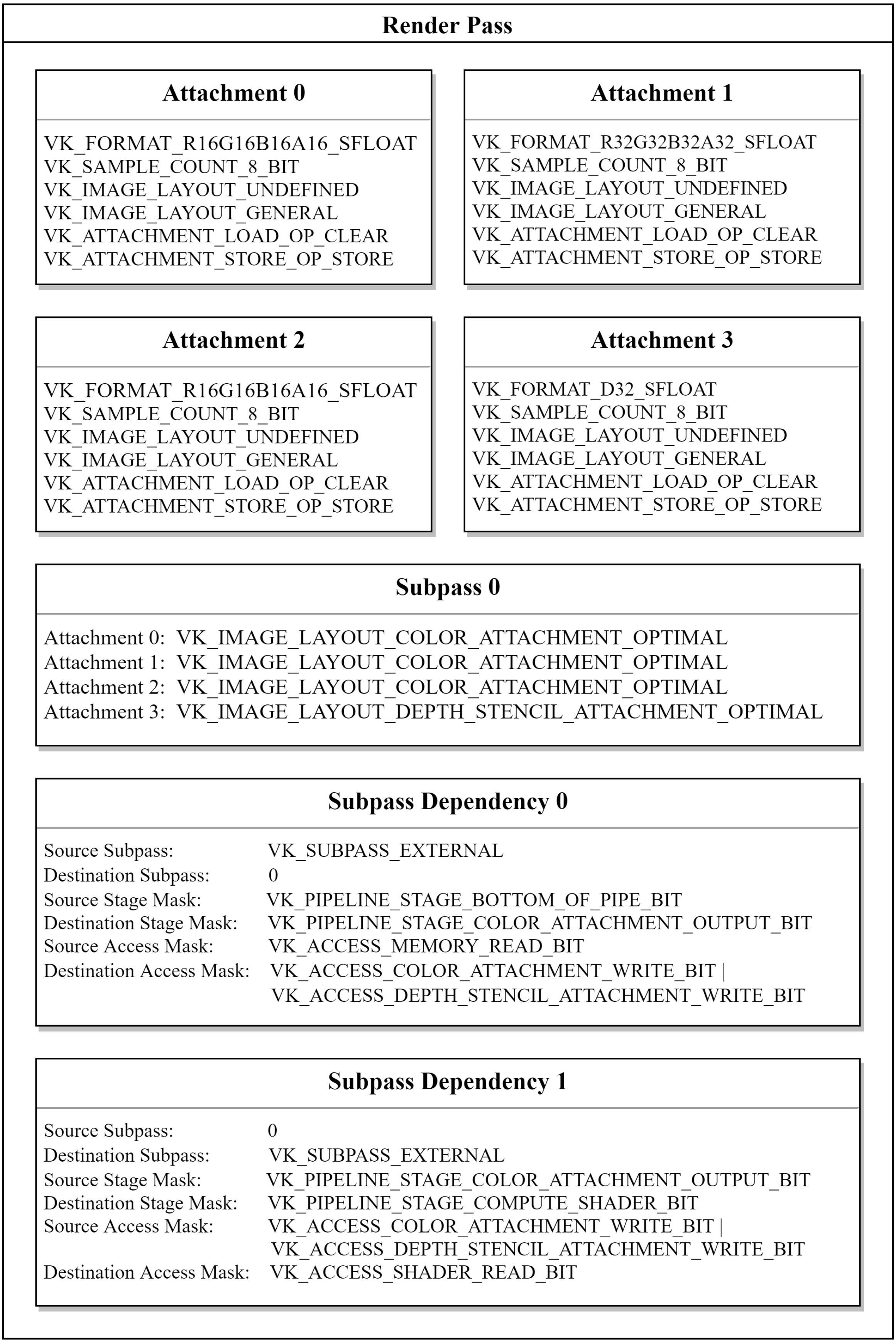

Framebuffer
Attachment 0 (Albedo)
VK_FORMAT_R16G16B16A16_SFLOAT
Attachment 1 (World Position)
VK_FORMAT_R32G32B32A32_SFLOAT
Attachment 2 (Normal)
VK_FORMAT_R16G16B16A16_SFLOAT
Attachment 3 (Depth)
VK_FORMAT_D32_SFLOAT

Figure 5.4: Four attachments are attached to the framebuffer. The color attachments 0, 1 and 2 are the albedo, normal and world position buffer of the g-buffer, respectively. Attachment 0 and 2 are specified with the enumeration `VK_FORMAT_R16G16B16A16_SFLOAT` which means that the images have the format of a 16 bit RGBA-channel interpreted as a signed float type by the Vulkan driver. Attachment 1 has a 32 bit RGBA-channel (`VK_FORMAT_R32G32B32A32_SFLOAT`). Attachment 3 is a depth attachment with a 32 bit depth-channel also interpreted as a signed float (`VK_FORMAT_D32_SFLOAT`). [14]



Render Pass

Attachment 0

VK_FORMAT_R16G16B16A16_SFLOAT
VK_SAMPLE_COUNT_8_BIT
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_ATTACHMENT_LOAD_OP_CLEAR
VK_ATTACHMENT_STORE_OP_STORE

Attachment 1

VK_FORMAT_R32G32B32A32_SFLOAT
VK_SAMPLE_COUNT_8_BIT
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_ATTACHMENT_LOAD_OP_CLEAR
VK_ATTACHMENT_STORE_OP_STORE

Attachment 2

VK_FORMAT_R16G16B16A16_SFLOAT
VK_SAMPLE_COUNT_8_BIT
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_ATTACHMENT_LOAD_OP_CLEAR
VK_ATTACHMENT_STORE_OP_STORE

Attachment 3

VK_FORMAT_D32_SFLOAT
VK_SAMPLE_COUNT_8_BIT
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_GENERAL
VK_ATTACHMENT_LOAD_OP_CLEAR
VK_ATTACHMENT_STORE_OP_STORE

Subpass 0

Attachment 0: VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
Attachment 1: VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
Attachment 2: VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
Attachment 3: VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL

Subpass Dependency 0

Source Subpass:	VK_SUBPASS_EXTERNAL
Destination Subpass:	0
Source Stage Mask:	VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
Destination Stage Mask:	VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
Source Access Mask:	VK_ACCESS_MEMORY_READ_BIT
Destination Access Mask:	VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT

Subpass Dependency 1

Source Subpass:	0
Destination Subpass:	VK_SUBPASS_EXTERNAL
Source Stage Mask:	VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
Destination Stage Mask:	VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
Source Access Mask:	VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
Destination Access Mask:	VK_ACCESS_SHADER_READ_BIT

Figure 5.5: Render Pass Diagram

Figure 5.5: The render pass uses all four attachments of the framebuffer with 8 samples (VK_SAMPLE_COUNT_8_BIT). In order to define multisample attachments in the render pass, the images of the related attachments must have been created as multisample images. The initial layouts of the attachments at the beginning of the render pass are undefined (VK_IMAGE_LAYOUT_UNDEFINED) and have a general layout (VK_IMAGE_LAYOUT_GENERAL) at the end of the render pass since the attachments are used by the subsequent stage as storage images. VK_LOAD_OP_CLEAR and VK_STORE_OP_STORE indicate the cleanup of the attachments at the begin and the storing at the end of the render pass. The render pass possesses one subpass which uses the color attachments as render targets in the fragment shader and the depth attachment as a buffer for storing the depth test results. For this usage, the attachments must be specified with proper layouts which is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL for the color attachments and VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL for the depth attachment. Additionally, subpass dependencies must be specified for implicit synchronization. Since there is only one subpass, two dependencies are sufficient to synchronize the subpass with the outside of the render pass. Subpass dependency 0 synchronizes subpass 0 with operations prior to the render pass (source subpass VK_SUBPASS_EXTERNAL and destination subpass 0). In contrast, subpass dependency 1 synchronizes subpass 0 with operatins after the render pass (source subpass 0 and destination subpass VK_SUBPASS_EXTERNAL). Further, it is specified at which stages the synchronization takes place. At subpass dependency 0 the source stage mask is VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT while the destination stage mask is VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT which means that the synchronization occurs at the end of a previous pipeline and the stage of the subpass related pipeline where the final color values are generated. Accordingly, subpass dependency 1 synchronizes the final color generating pipeline stage of subpass 0 with the compute shader stage (VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT) of a following pipeline outside of the render pass, since the subsequent pipeline (sample coverage pipeline) reads the attachments in a compute shader. In addition, it is specified which access operations are considered at synchronization. At subpass dependency 0 the source access mask is set to VK_ACCESS_MEMORY_READ_BIT meaning that all kind of memory read operations of previous pipelines are considered. The destination access mask of subpass dependency 0 is specified as VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT which synchronizes subpass 0 at color/depth attachment write access. At subpass dependency 1 the access masks are accordingly set to VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT as source access snychronization in the pipeline of subpass 0 and VK_ACCESS_SHADER_READ_BIT as destination access synchronization, since the subsequent pipeline applies read-access to the attachments as mentioned before in a compute shader. [14]

5.3 Sample Coverage and Deferred Shading

The sample coverage and deferred shading stages are encapsulated in a single command buffer. Sample coverage and deferred shading passes are defined as compute pipelines with a pipeline memory barrier amongst both dispatches.

5.4 Transparent Scene and Blending

The transparent scene objects are rendered similar to the opaque scene objects with a primary command buffer and a separate framebuffer. Since forward lighting is used, the framebuffer does not have a g-buffer attachment but instead a single color attachment. The blending of the deferred lighted and shaded opaque scene and the transparent scene is done in a further command buffer.

5.5 FXAA and Post Processing

The post processing bloom effect and FXAA is wrapped in one command buffer with a pipeline memory barrier similar to the barriers in listing 5.1 between FXAA and bloom pipelines. Further pipeline memory barriers are used to synchronize the different bloom processing stages.

5.6 Panel Overlay

The panel overlay is rendered by wrapping all the overlay elements as secondary command buffers in one primary command buffer similar to the opaque and transparent scene command buffer. Afterwards the panel overlay image is blended with an alpha blending function on top of the final synthesized scene image.

5.7 Presentation

The presentation engine uses a swapchain with `VK_PRESENT_MODE_MAILBOX_KHR` presentation mode. For each swapchain image, a `VkImageView` is created and attached to a separate `VkFrameBuffer` object. Further, the same number of command buffers are recorded for rendering into one framebuffer, respectively. When a swapchain image is acquired with `vkAcquireNextImageKHR` the command buffer with the related framebuffer is submitted to the presentation queue. The acquisition of an unused swapchain image and the submission of a command buffer that draws to that image is synchronized with a respective semaphore.

Chapter 6

Case Study: OpenGL vs. Vulkan

The case study simulation scenario presented in chapter 4 was testet against different antialiasing configurations from figure 4.7 on a system with an *Intel Core i5-6600k*, 16 GB RAM and the Titan Xp. The operating system is *Windows 10.* The camera and level of detail settings are exactly the same for both Vulkan and OpenGL with every antialiasing configuration. The FFT's have a resolution of 256x256. On a time window of 20 seconds the fps, CPU load and GPU load were measured as performance indicators. The fps were simply recorded by the rendering loop of the engine. CPU load was measured with the tool *Performance Monitor* included in Windows 10 while GPU load was measured with the tool *GPU-Z.* The following diagrams contrasts the measured average values of the simulation executed with Vulkan and OpenGL.

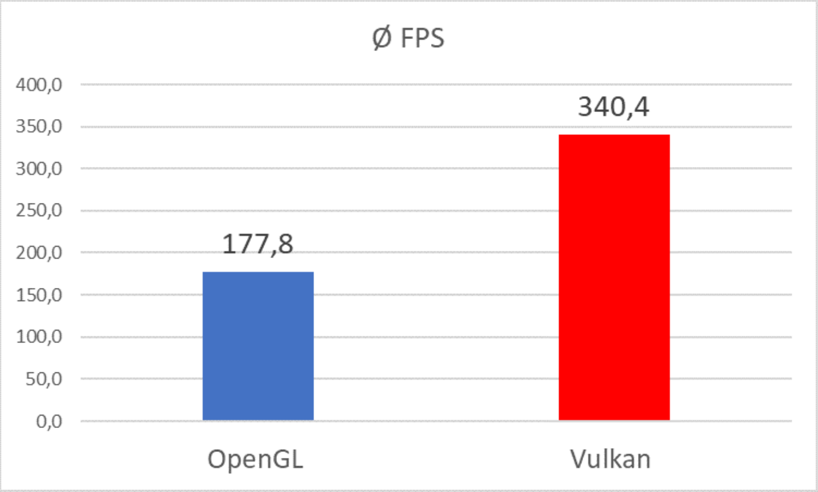


Figure 6.1: 2x MSAA - FPS

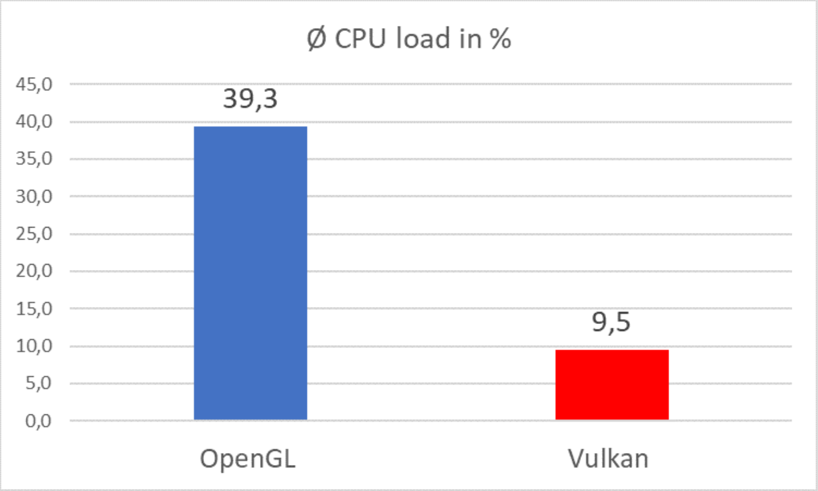


Figure 6.2: 2x MSAA - CPU Load

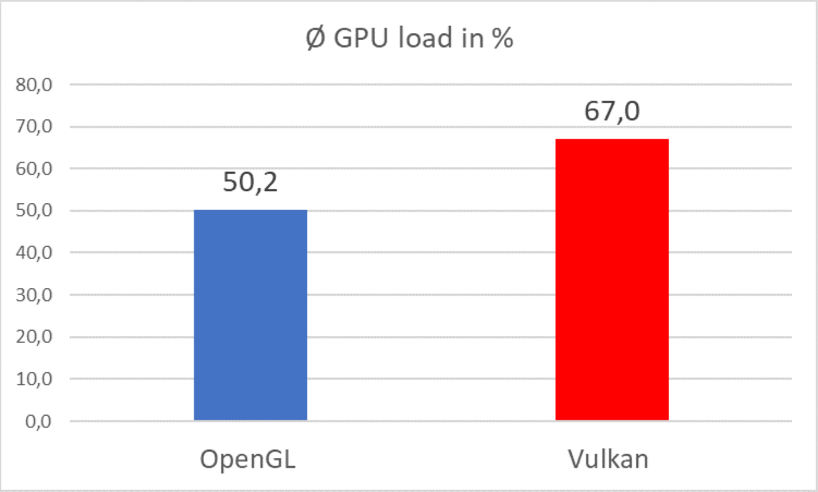


Figure 6.3: 2x MSAA - GPU Load

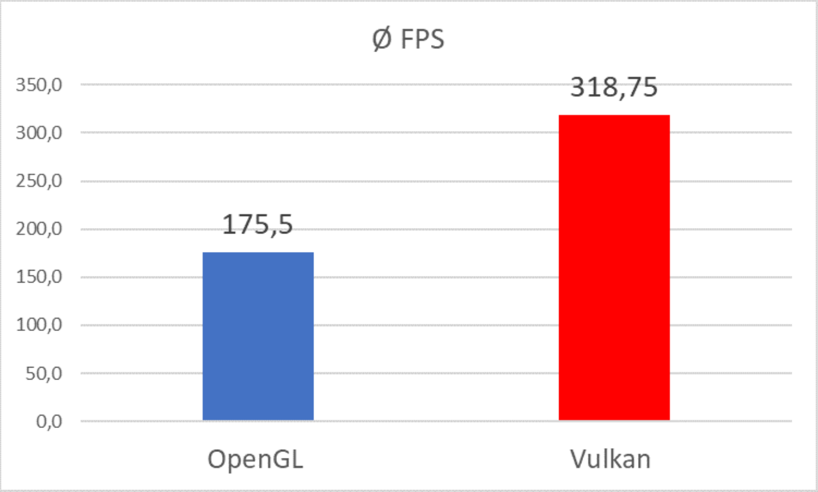


Figure 6.4: 2x MSAA and FXAA - FPS

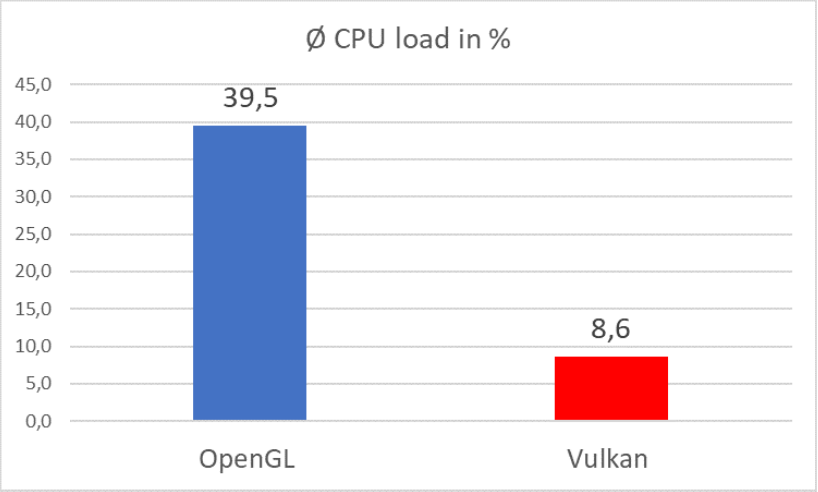


Figure 6.5: 2x MSAA and FXAA - CPU Load

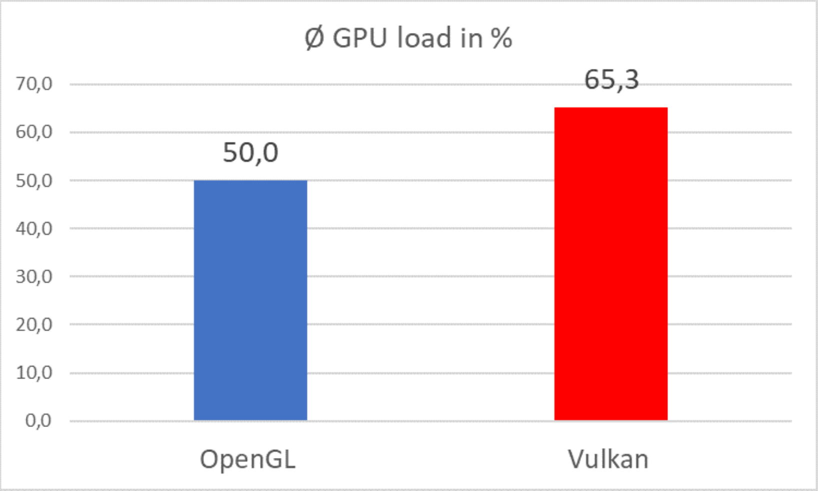


Figure 6.6: 2x MSAA and FXAA - GPU Load

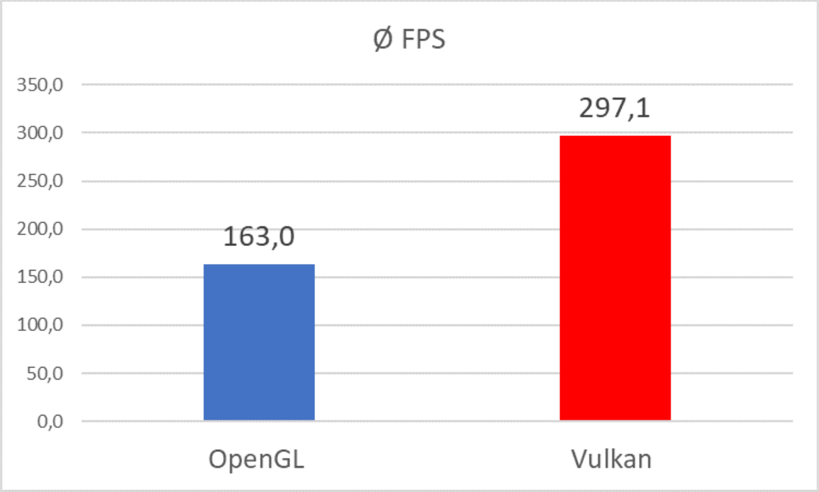


Figure 6.7: 4x MSAA - FPS

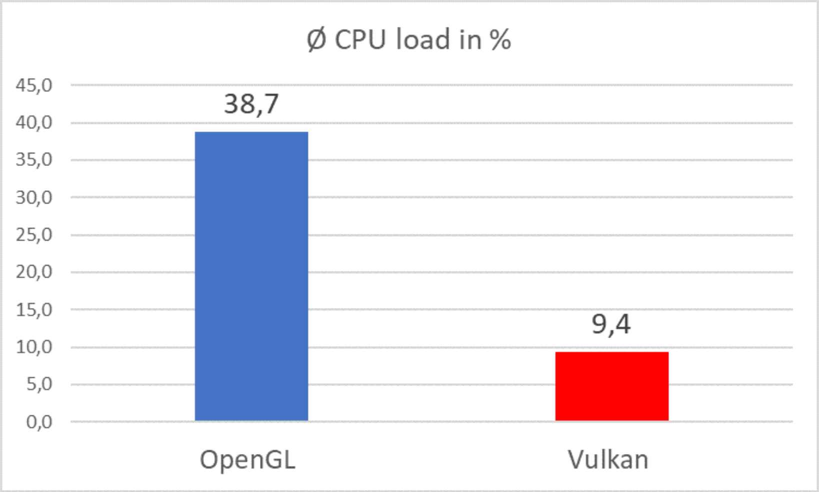


Figure 6.8: 4x MSAA - CPU Load

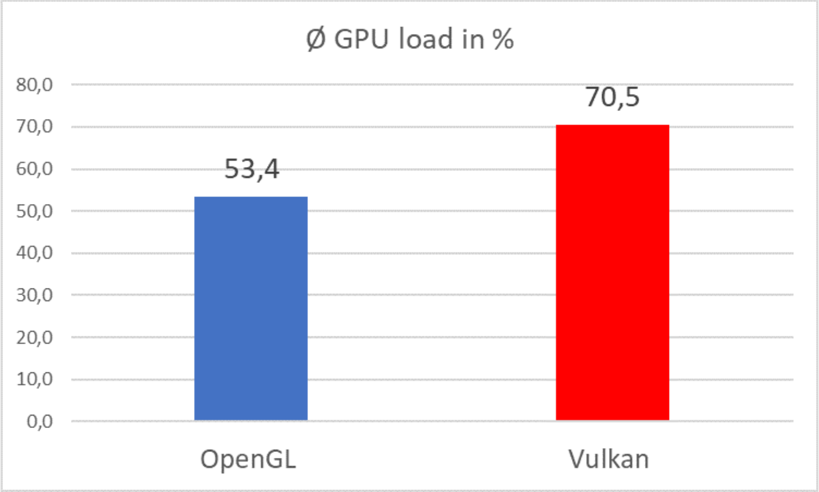


Figure 6.9: 4x MSAA - GPU Load

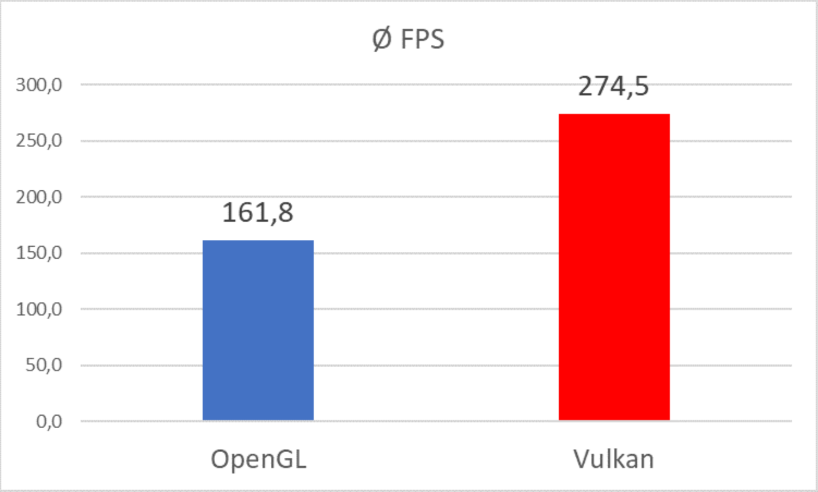


Figure 6.10: 4x MSAA and FXAA - FPS

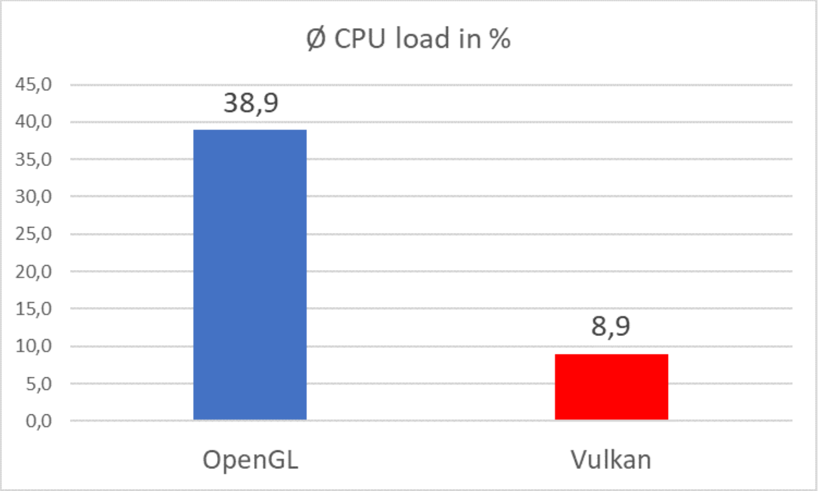


Figure 6.11: 4x MSAA and FXAA - CPU Load

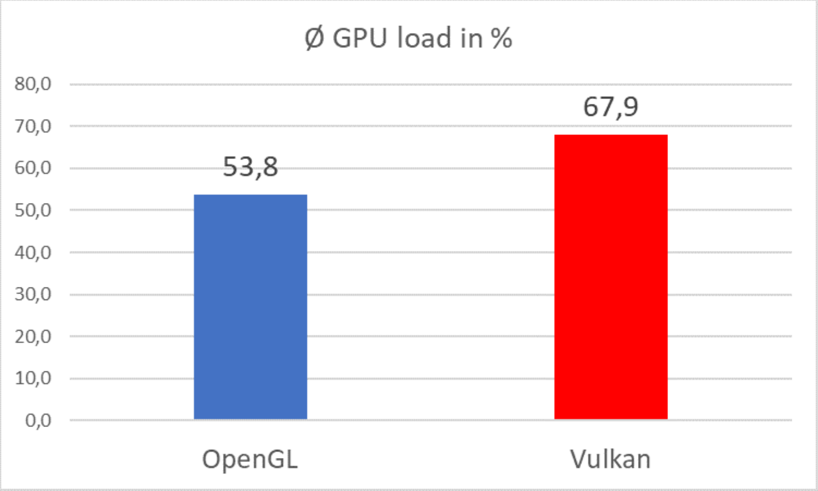


Figure 6.12: 4x MSAA and FXAA - GPU Load

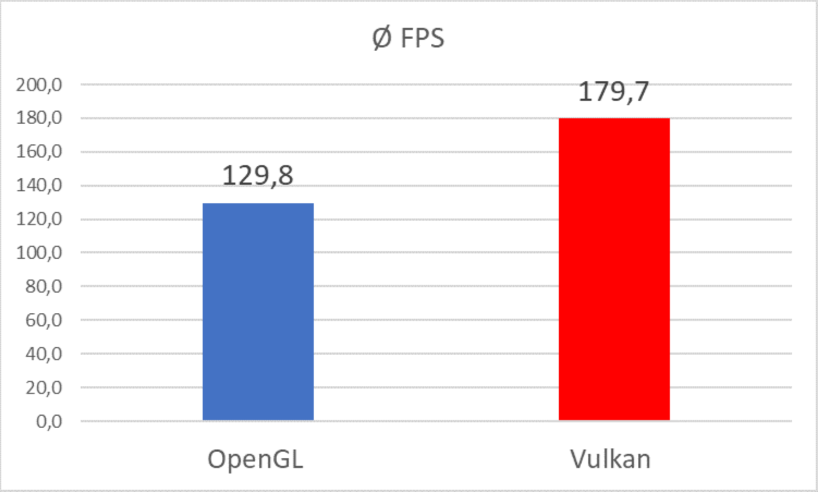


Figure 6.13: 8x MSAA - FPS

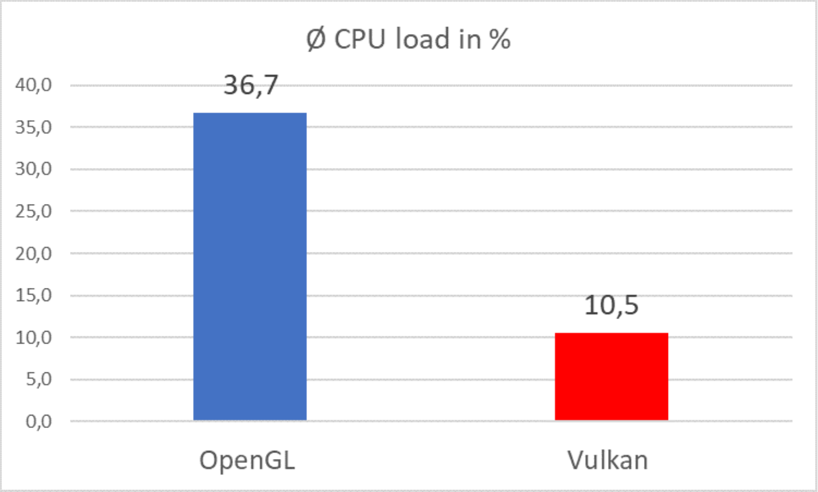


Figure 6.14: 8x MSAA - CPU Load

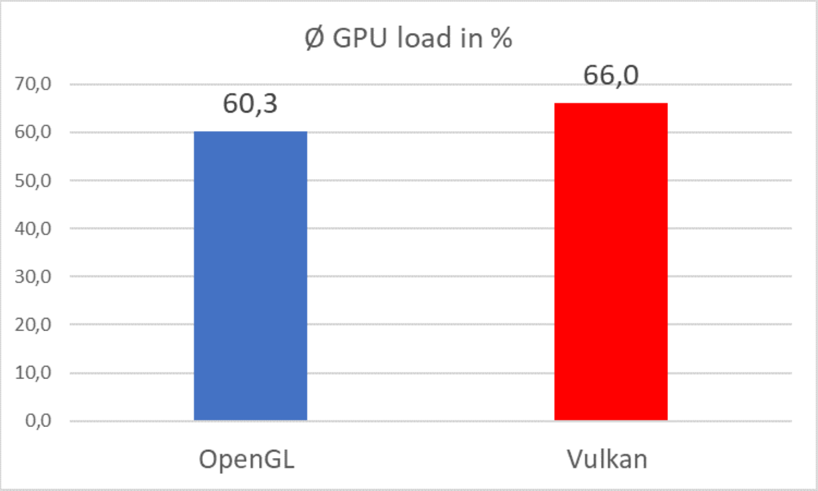


Figure 6.15: 8x MSAA - GPU Load

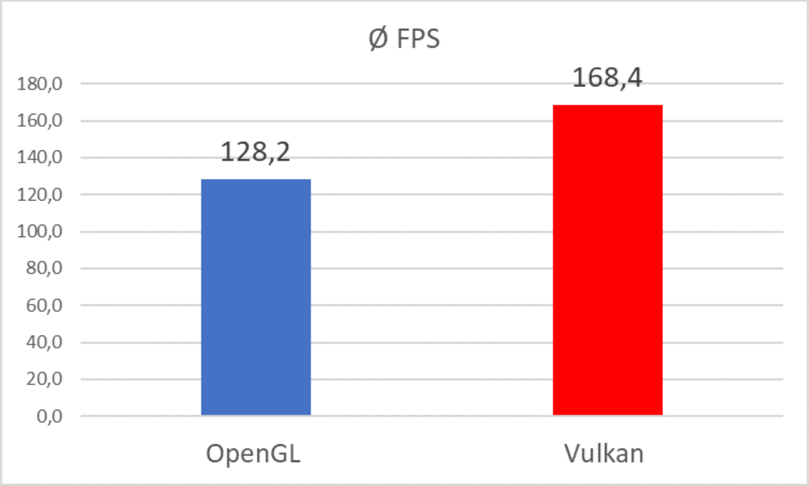


Figure 6.16: 8x MSAA and FXAA - FPS

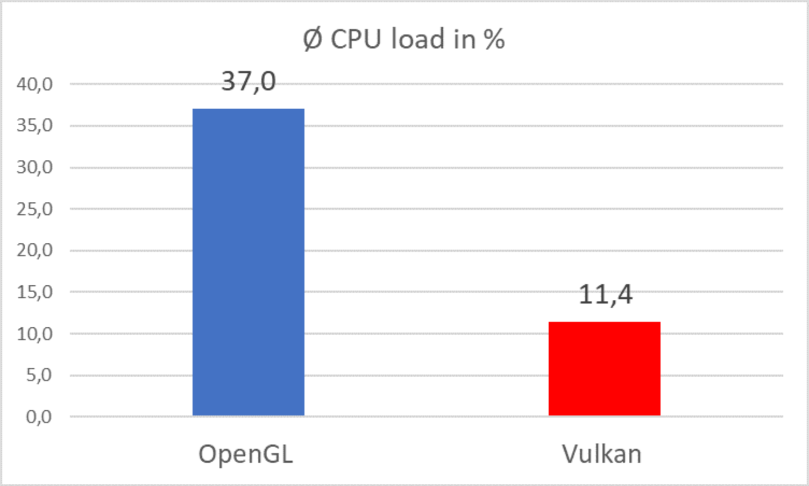


Figure 6.17: 8x MSAA and FXAA - CPU Load

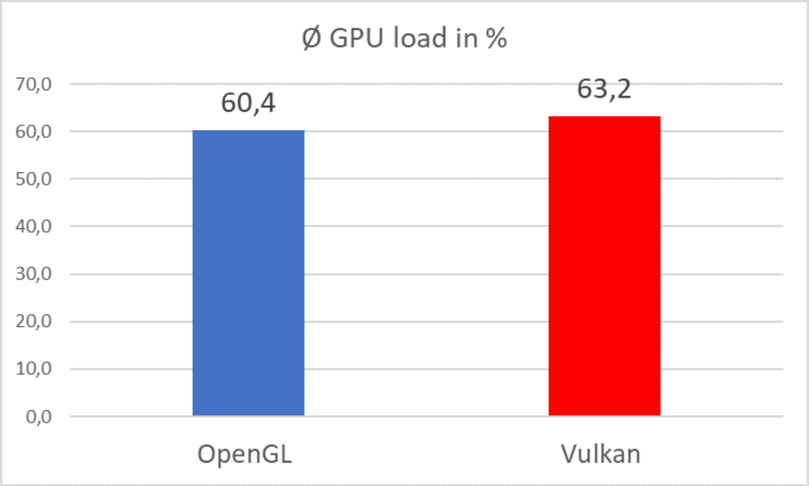


Figure 6.18: 8x MSAA and FXAA - GPU Load

Chapter 7

Evaluation

The measured performance indicators presented in the diagrams in chapter 6 expose major advantages of Vulkan over OpenGL. At the configuration with 2x MSAA, Vulkan reaches 91% more fps than OpenGL (figure 6.1). With increasing antialiasing configurations, the advantage of Vulkan decreases. For instance, at 4x MSAA Vulkan reaches 82% (figure 6.7) more fps and at the maximum antialiasing configuration with 8x MSAA and FXAA the advantage is 31% (figure 6.16). However, in consideration of the CPU load difference of Vulkan and OpenGL, the advantages of Vulkan become more significant. The CPU load of the Vulkan simulation moves constantly around 10% (± 1.4) while the OpenGL simulation constantly demands around 38% of the CPU power (± 1.5). Hence, OpenGL requires three to four times as much CPU capacities as Vulkan which is a significant difference. By the fact that Vulkan reaches 31% to 82% more performance by just consuming 25% of the CPU load compared to OpenGL, it can be considered as proven that Vulkan is more powerful and effective than OpenGL as described in chapter 2. Further, under consideration of the GPU load measurements, it can be derived that Vulkan is able to supply work to the GPU more effectively, since the GPU load of Vulkan is constantly higher than OpenGL. Though with increasing antialiasing quality, the GPU load difference decreases. While the GPU load difference is 16.8% at 2x MSAA (figure 6.3), at 8x MSAA with FXAA the difference is only 2.8% (figure 6.18). A possible explanation to this is, that the shader executions become more complex for high antialiasing configurations. As a consequence, the GPU spends more time with executing single shader programs. However, during shader execution, it does not matter if Vulkan or OpenGL is used though. This is possibly the same reason why the fps gap between Vulkan and OpenGL decreases for increasing antialiasing configurations.

Appendix

A Measured Simulation Data - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	175	39.9	49
2	179	36.4	51
3	180	44.1	51
4	179	41.8	49
5	179	39.9	48
6	178	35.6	49
7	179	41.0	50
8	177	37.5	52
9	175	39.5	49
10	177	38.7	49
11	179	40.3	51
12	177	41.8	50
13	180	34.0	50
14	178	42.2	50
15	177	36.7	52
16	177	39.7	51
17	177	39.1	51
18	180	36.4	51
19	177	39.5	50
20	175	42.6	50

Table 1: 2x MSAA - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	175	39.8	48
2	175	38.7	50
3	175	41.4	50
4	177	38.3	50
5	176	39.9	51
6	180	35.6	50
7	174	40.6	49
8	176	39.1	49
9	179	39.5	50
10	176	38.4	51
11	176	42.2	50
12	175	36.8	51
13	174	35.2	50
14	175	38.7	50
15	172	41.2	52
16	173	48.5	46
17	175	40.3	51
18	174	39.1	51
19	177	39.9	51
20	175	36.3	50

Table 2: 2x MSAA and FXAA - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	164	34.4	53
2	161	35.2	54
3	166	38.3	52
4	160	36.4	54
5	166	37.9	55
6	162	35.6	53
7	163	39.1	53
8	162	36.0	54
9	166	35.2	54
10	163	38.6	55
11	165	42.2	52
12	163	41.4	53
13	161	41.8	52
14	161	44.9	55
15	165	39.5	53
16	161	37.1	54
17	164	41.1	54
18	162	39.9	51
19	162	39.8	53
20	162	39.9	54

Table 3: 4x MSAA - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	163	38.3	54
2	163	38.3	54
3	162	49.1	54
4	163	34.0	55
5	162	38.7	54
6	162	38.7	55
7	165	40.7	54
8	162	35.2	52
9	160	40.9	55
10	161	35.2	54
11	161	41.4	48
12	162	51.6	55
13	160	35.2	54
14	161	40.3	54
15	162	38.3	53
16	158	36.7	54
17	162	34.0	55
18	162	38.7	54
19	161	42.6	54
20	162	39.9	53

Table 4: 4x MSAA and FXAA - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	130	36.3	60
2	130	35.6	61
3	128	33.6	59
4	128	36.4	60
5	129	35.6	60
6	133	36.0	59
7	130	37.1	60
8	129	38.7	60
9	129	32.4	61
10	131	31.3	60
11	130	45.3	59
12	130	36.8	61
13	129	34.1	61
14	130	39.5	60
15	132	37.1	60
16	129	35.9	62
17	129	35.5	60
18	128	39.1	62
19	131	36.8	60
20	130	38.3	61

Table 5: 8x MSAA - OpenGL

t in s	fps	CPU load in %	GPU load in %
1	128	32.5	59
2	128	36.0	61
3	128	37.1	60
4	130	36.4	61
5	132	34.4	59
6	129	33.7	59
7	129	39.1	62
8	129	36.0	61
9	130	34.0	60
10	127	49.2	60
11	126	37.1	60
12	126	36.4	63
13	129	41.4	60
14	128	32.5	61
15	128	36.8	61
16	127	41.4	60
17	127	38.3	60
18	128	34.8	60
19	128	40.6	59
20	127	33.3	61

Table 6: 8x MSAA and FXAA - OpenGL

B Measured Simulation Data - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	343	9.1	65
2	343	11.7	67
3	345	11.7	67
4	340	9.0	67
5	328	8.6	68
6	341	8.6	68
7	341	9.4	67
8	343	11.3	67
9	344	10.1	67
10	342	8.6	66
11	342	9.8	67
12	341	7.9	67
13	341	6.3	68
14	339	9.0	67
15	337	7.9	66
16	341	8.2	67
17	338	15.7	68
18	339	9.8	68
19	341	8.2	67
20	339	9.5	66

Table 7: 2x MSAA - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	316	14.4	65
2	322	5.5	65
3	321	8.6	65
4	320	9.5	67
5	326	6.3	63
6	323	7.4	66
7	321	10.6	68
8	321	7.5	66
9	319	9.8	66
10	321	11.4	66
11	318	7.1	65
12	312	7.4	65
13	313	9.4	64
14	316	5.9	64
15	320	10.2	67
16	321	9.1	65
17	315	9.4	64
18	318	7.8	64
19	321	6.3	66
20	311	9.0	65

Table 8: 2x MSAA and FXAA - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	396	5.9	70
2	392	7.0	70
3	300	9.0	70
4	298	7.8	69
5	293	5.5	70
6	300	8.3	70
7	299	9.0	70
8	300	13.3	69
9	296	9.0	71
10	297	14.1	71
11	299	7.0	71
12	295	7.8	70
13	298	10.2	71
14	293	9.5	71
15	295	8.6	71
16	300	13.7	72
17	295	8.7	71
18	293	11.7	71
19	302	10.5	71
20	300	11.3	71

Table 9: 4x MSAA - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	275	9.8	70
2	281	10.6	69
3	273	7.5	70
4	280	12.9	69
5	282	10.2	68
6	282	8.6	69
7	275	6.3	66
8	279	7.1	67
9	278	7.9	70
10	273	8.7	69
11	273	5.5	69
12	276	8.6	66
13	270	7.4	67
14	272	9.4	66
15	271	8.3	66
16	267	17.6	68
17	263	7.8	65
18	269	9.4	69
19	270	8.3	69
20	280	5.9	66

Table 10: 4x MSAA and FXAA - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	178	10.2	62
2	178	11.7	65
3	179	6.6	63
4	182	9.4	68
5	178	10.2	68
6	178	11.3	64
7	177	15.3	62
8	171	10.5	65
9	179	14.9	65
10	171	8.6	66
11	183	14.5	66
12	178	10.6	68
13	199	10.6	72
14	194	7.9	75
15	178	9.8	67
16	184	10.6	69
17	175	8.6	65
18	173	7.1	62
19	183	13.8	66
20	175	8.2	62

Table 11: 8x MSAA - Vulkan

t in s	fps	CPU load in %	GPU load in %
1	168	7.8	70
2	161	12.5	63
3	168	11.0	72
4	166	11.7	64
5	164	13.7	59
6	179	12.2	57
7	176	8.3	62
8	172	8.7	65
9	164	11.3	61
10	160	11.7	60
11	173	9.9	65
12	164	13.0	64
13	175	14.5	65
14	166	10.2	63
15	176	12.5	66
16	171	12.9	61
17	166	14.1	58
18	167	11.3	63
19	169	10.7	59
20	162	9.4	67

Table 12: 8x MSAA and FXAA - Vulkan

Bibliography

[1] Oreon Engine. https://github.com/oreonengine/oreon-engine.

[2] Fynn-Jorin Flügge. Realtime GPGPU FFT Ocean Water Simulation, 2017. http://tubdok.tub.tuhh.de/handle/11420/1439.

[3] Ian Buck. The Evolution of GPUs for General Purpose Computing. Nvidia, September 2010. In proceedings of the GTC 2010, http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf.

[4] Nav Singh. Upcoming NVIDIA driver to give AMD Mantle a run for its money - massive performance boost. *Digitalstorm*, March 2014. http://www.digitalstorm.com/unlocked/upcoming-nvidia-driver-to-give-amd-mantle-a-run-for-its-money-massive-performance-boost-idnum209, visited August 10, 2018.

[5] Neil Trevett. Vulkan, SPIR-V and OpenCL 2.1. Khronos Group and Nvidia, April 2015. In proceedings of GTC 2015, https://www.khronos.org/assets/uploads/developers/library/2015-gtc/Khronos-Overview-GTC_Mar15.pdf.

[6] Microsoft - What is Direct3D 12? https://docs.microsoft.com/en-us/windows/desktop/direct3d12/what-is-directx-12-, requested August 10, 2018.

[7] Neil Trevett. Khronos Overview. Khronos Group and Nvidia, March 2016. In proceedings of GDC 2016, https://www.khronos.org/assets/uploads/developers/library/2016-gdc/Khronos-GDC-Overview_Mar16.pdf.

[8] Tom Olson. Vulkan 101. Khronos Group, September 2016. In proceedings of Vulkan DevDay UK 2016, https://www.khronos.org/assets/uploads/developers/library/2016-vulkan-devday-uk/1-Vulkan_101.pdf.

[9] Neil Trevett. Ecosystem Overview. Khronos Group and Nvidia, April 2016. In proceedings of GTC 2016, https://www.khronos.org/assets/uploads/developers/library/2016-gtc/2016-GTC-Khronos-API-Ecosystem_Apr16.pdf.

[10] Neil Trevett. Khronos Standards Update. Nvidia, March 2018. In proceedings of GTC 2018, https://www.khronos.org/assets/uploads/developers/library/2018-gtc/Khronos-Standards-Update-Trevett-GTC_Mar18.pdf.

[11] History of OpenGL. https://www.khronos.org/opengl/wiki/History_of_OpenGL, requested August 10, 2018.

[12] *OpenGL 4.5 Core Profile.* Khronos Group, April 2017. https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

[13] OpenGL FAQ. https://www.khronos.org/opengl/wiki/FAQ#What_is_OpenGL.3F, requested August 10, 2018.

[14] *Vulkan® 1.1.82 - A Specification (with all registered Vulkan extensions).* Khronos Group, 2018. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html.

[15] Package org.lwjgl.vulkan. https://javadoc.lwjgl.org/org/lwjgl/vulkan/package-summary.html, requested August 10, 2018.

[16] Pawel Lapinski. *Vulkan Cookbook.* Packt Publishing, 2017.

[17] Architecture of the Vulkan Loader Interfaces. https://github.com/KhronosGroup/Vulkan-LoaderAndValidationLayers/blob/master/loader/LoaderAndLayerInterface.md#application-usage-of-extensions, requested August 10, 2018.

[18] Vulkan Validation Layers. https://vulkan.lunarg.com/doc/sdk/1.1.82.0/windows/layer_configuration.html, requested August 10, 2018.

[19] Create a Swapchain. https://vulkan.lunarg.com/doc/view/1.1.73.0/windows/tutorial/html/05-init_swapchain.html, requested August 10, 2018.

[20] API without Secrets: Introduction to Vulkan* Part 2: Swap Chain. https://software.intel.com/en-us/articles/api-without-secrets-introduction-to-vulkan-part-2, requested August 10, 2018.

[21] Create a Render Pass. https://vulkan.lunarg.com/doc/view/1.1.73.0/windows/tutorial/html/10-init_render_pass.html, requested August 10, 2018.

[22] Create a Pipeline. https://vulkan.lunarg.com/doc/view/1.1.73.0/windows/tutorial/html/14-init_pipeline.html, requested August 10, 2018.

[23] Vulkan 1.0.19 + WSI Extensions - A Specification. http://vulkan-spec-chunked.ahcox.com/ch09.html, requested August 10, 2018.

[24] Vulkan Shader Resource Binding. https://developer.nvidia.com/vulkan-shader-resource-binding, requested August 10, 2018.

[25] Vulkan Barriers Explained. https://gpuopen.com/vulkan-barriers-explained, requested August 10, 2018.

[26] Tom Olson, David Neto, and Dan Ginsburg. What's New in Vulkan? Khronos Group, March 2018. In proceedings of GDC 2018, https://www.khronos.org/assets/uploads/developers/library/2018-gdc-webgl-and-gltf/1-Vulkan-Whats-New-GDC_Mar18.pdf.

[27] Jerry Tessendorf. Simulating Ocean Water. 1999.

[28] Mark Harris and Sawn Hargreaves. Deferred Shading. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf, requested August 10, 2018.

[29] Nvidia Gameworks - Antialiased Deferred Rendering. https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/d3d_samples/antialiaseddeferredrendering.htm, requested August 10, 2018.

[30] Timothy Lottes. FXAA. https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf, requested August 10, 2018.

[31] Greg James and John O'Rorke. *GPU Gems - Chapter 21. Real-Time Glow*. Nvidia, 2004.

[32] Relative Luminance. https://www.w3.org/WAI/GL/wiki/Relative_luminance, requested August 10, 2018.

[33] Khronos Logos, Trademarks, and Guidelines. https://www.khronos.org/legal/trademarks, requested August 10, 2018.