Multi-GPU Rendering with Vulkan API

Lars Olav Tolo

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Science

Department of Informatics,
University of Bergen

June 2018

Abstract

Vulkan API provides a low level interface to modern Graphics Processing Units (GPUs). With this thesis, we demonstrate how to use Vulkan to send commands explicitly to separate GPUs for implementing platform- and vendor independent multi-GPU rendering. We describe how to implement the sort-first and sort-last approaches to perform parallel rendering with Vulkan. We introduce an abstraction library which we have made available, and an application for multi-GPU rendering of meshes. The introduced solution is the first publicly available implementation of heterogeneous multi-GPU rendering with Vulkan API. The introduced abstraction library supports creating partial renderers for the sort-first and sort-last approaches to multi-GPU rendering, and takes care of the details of multi-GPU synchronization and compositing. Performance benchmarks have been performed in order to evaluate the implementation. The sort-last implementation has been tested to render geometry with high-resolution textures, which would otherwise not fit in the memory of a single GPU.

Acknowledgements

I would like to thank my supervisors at Western Norway University of Applied Sciences (HVL), Daniel Patel, Harald Soleim and Atle Geitung, for providing feedback and guidance, motivating me to do my best. We had frequent meetings, where we discussed my progress. I would also like to thank my co-supervisor Ivan Viola, who is a professor at Vienna University of Technology (TU Wien). In conversation with Patel, Viola formulated the thesis topic. Viola and I had regular conversations, from which I got useful advice.

I had the opportunity to go to Vienna for a study trip, where I got introduced to the vis-group at TU Wien, who performs research on visualization. I want to thank HVL and TU Wien for providing funds for my trip to Vienna. I want to thank the vis-group for making me feel welcome, and including me in social activities. Viola provided supervision and guidance during my stay, for which I am grateful. I had the opportunity to talk to Henry Fuchs, one of the authors of the cited paper "A Sorting Classification of Parallel Rendering". Fuchs and I had conversations about multi-GPU rendering, from which I got a better understanding of the topic.

I have submitted a paper about multi-GPU rendering with Vulkan, co-authored by my supervisors Patel, Soleim, Geitung and Viola. The paper is accepted for presentation at the Norwegian Informatics Conference (NIK) 2018.

Contents

List of Figures

Listings

Glossary

AFR Alternate Frame Rendering. 9, 18, 19

API Application Programming Interface. 7–9, 12, 14, 19, 23, 26, 30–32, 34, 64, 66

BP Boilerplate. 31, 34, 35, 40, 42, 45, 48, 64, 65

CAD Computer-Aided Design. 7

CPU Central Processing Unit. 5, 11, 16, 17, 22, 26, 28, 40, 41, 47, 64, 68

culling is the process of skipping objects, or primitives that will not be visible in the final render. 15

fragment is a candidate for becoming a pixel and is the result from rasterization. 13, 15, 16, 21, 29, 44, 45, 51, 52, 62

framebuffer is a region of memory holding the resulting bitmap image from rendering. 15, 20–23, 27, 32, 36–38, 42–44, 46, 51, 61, 64, 65

GPGPU General-purpose computing on GPUs. 7, 8

GPU Graphics Processing Unit. 1, 5–13, 16–26, 28, 30, 32, 33, 37–55, 58, 59, 61–64, 66–69

HEDT High-End Desktop. 16

host the Central Processing Unit (CPU) and main memory. 26, 28, 40, 41, 64

motherboard is a circuit board connecting the components of a computer together. 16, 17

OS Operating System. 8, 22, 23, 31

PCI-E Peripheral Component Interconnect Express. 16, 17, 47

pipeline is a model describing the steps necessary to render 3D graphics on a GPU. 6–8, 13–15, 20, 21, 25, 27–29, 36, 37, 39

primitive is the representation of what gets rendered from a list of vertices, for instance points, lines, triangles. 6, 7, 13–15, 20–22, 27

rasterization is the process of converting primitives into fragments/pixels. 5, 13, 15, 27

rendering is the process of generating an image from 2D or 3D models. 3, 6–8, 10, 11, 14, 18–26, 30, 32, 36–42, 44–48, 50–58, 60, 61, 63, 64, 66–69

SFR Split-Frame Rendering. 9, 11, 18–20

shader is a program that runs for each primitive in one of the programmable rendering pipeline stages. 7, 8, 14, 15, 21, 25, 27, 28, 31, 36, 44, 45, 49, 51, 52

SLI Scalable Link Interface. 8, 9, 16

SLI, 3dfx Scan-Line Interleave. 8

sort-first is a sorting class describing an approach to implement parallel rendering by distributing primitives early in the rendering pipeline. 3, 18–20, 22, 32, 40, 42–45, 48, 50–53, 55, 61–63, 66–69

sort-last is a sorting class describing an approach to implement parallel rendering by distributing primitives at the end of the rendering pipeline. 3, 19–22, 32, 40, 42, 45–48, 50–52, 54–58, 61, 62, 66, 67, 69

sort-middle is a sorting class describing an approach to implement parallel rendering by distributing primitives in the middle of the rendering pipeline. 21

SPIR-V Standard Portable Intermediate Representation. 27, 31

TCS Tesselation Control Shader. 15

TES Tesselation Evaluation Shader. 15

vertex is a 3D point represented as a vector. 13–15, 21, 29, 36, 39, 43, 49, 52

XDMA AMD CrossFire Direct Memory Access. 17

Chapter 1

Introduction

Computer graphics is everywhere. It provides immersive video games, smooth animations in graphical user interfaces, and visualizations of 3D Computer-Aided Design (CAD). With ever increasing screen resolutions and complexity of visualizations, there is always demand for more computing power to provide a seamless experience. This is why modern computers have dedicated hardware to implement accelerated computer graphics. This is called the Graphics Processing Unit (GPU). GPUs implement functional parallelism (pipelining) to distribute the rendering workload among hundreds, or even thousands of computational units, for accelerated performance in graphics intensive tasks.

The graphics pipeline consists of a series of steps which transforms the primitives of a 3D mesh (points, lines, triangles), to the pixels you see on the screen. Some steps can be customized with shader programs. Modern GPUs also provide compute shaders, which operate outside of the pipeline, to implement General-purpose computing on GPUs (GPGPU).

A graphics Application Programming Interface (API) is a unified interface to GPUs from different vendors, and possibly different operating systems. OpenGL [1] and Vulkan [2] are examples of such graphics APIs. They provide the functionality of transferring data and commands to the GPU for implementing the rendering and compute operations. Graphics APIs can be extended with additional functionality with extensions. Extensions can provide general functionality available for all GPUs, but GPU vendors can also create extensions specific to their hardware.

As the graphics hardware has rapidly been changing, adding new features, the graphics APIs have had to adapt. Programmable graphics pipeline stages using shaders is one of the most significant changes in modern graphics. Modern OpenGL (version 3.0 and up) have shifted away from a fixed function pipeline, towards a shader based graphics pipeline. Recent versions of OpenGL now support GPGPU using compute shaders.

In 2016, Khronos Group (developers of OpenGL) launched version 1.0 of Vulkan API [2]. Vulkan API is a graphics and compute API, promising low overhead and ability to operate in parallel. The API provides a lower level access to the GPU than OpenGL.

For demanding graphics rendering on a computer, it is possible that multiple GPUs can improve performance. In 1998, 3dfx introduced their multi-GPU technology Scan-Line Interleave (SLI, 3dfx), which brought multi-GPU rendering power to the consumer. More recent multi-GPU implementations are NVIDIA's Scalable Link Interface (SLI) [3] and AMD's Crossfire [4]. However these technologies require hardware support, and only works with GPUs from the same vendor. Being able to use GPUs from different vendors in parallel requires the developer to manually divide and distribute the rendering workload. Not all graphics APIs have the features necessary to implement such a general solution to multi-GPU rendering. With recent graphics APIs providing a lower level interface than before, developers have more freedom to manage details of the rendering process. Vulkan API require the developer to select GPUs to perform rendering explicitly.

1.1 Motivation

Being able to select explicitly which GPU to assign rendering work to, is important for implementing a solution for multi-GPU rendering. We will refer to this ability as explicit selection or explicit rendering. Core OpenGL does not support explicit selection of GPUs on the Windows Operating System (OS). To be able to assign work to each GPU explicitly with OpenGL on Windows, we have to rely on extensions specific to GPU vendors. AMD's GPU association extension [5] and NVIDIA's GPU affinity extension [6] provide the functionality of creating OpenGL contexts associated with specific GPUs. An OpenGL context represent the state of OpenGL, and must be bound to the thread that should call OpenGL commands using the context [7]. Though AMD's extension target both consumer and professional GPUs [8], NVIDIA's extension is limited to their professional lineup (Quadro) [9]. Hence, these extensions can not be considered a general solution to implement explicit multi-GPU rendering on consumer GPUs, as it requires a configuration of either multiple NVIDIA Quadro GPUs, or multiple AMD GPUs.

Explicit selection of GPUs is possible with Direct3D. Direct3D is however designed for the Windows operating system and is not an open API. Vulkan API is an open API, which most likely will be supported by drivers and most operating systems for the foreseeable future. Vulkan provides the flexibility to implement multi-GPU rendering in a general solution working on all Vulkan-capable GPUs, as Vulkan supports explicit selection.

NVIDIA's SLI [3] and AMD's Crossfire [4] offer automatic, but limited utilization of several graphics cards residing in one computer. This is handled by the graphics drivers and does not require a change in the application itself. Both vendors offer two types of parallelization, Alternate Frame Rendering (AFR) where alternate frames are rendered by alternate GPUs, and Split-Frame Rendering (SFR) where the rendering window is split and each GPU renders a part of it. SLI only works with identical NVIDIA cards while Crossfire can work with different AMD cards. However, if the user wants to utilize different shaders and/or geometry on different GPUs, these technologies are not sufficient. We show in this thesis that we can implement such setups with Vulkan API.

1.2 Goal

The goal of this thesis is to explore the use of multiple GPUs on a single computer with Vulkan API, and implement an efficient solution to multi-GPU rendering. Different approaches to dividing and distributing rendering workload among multiple GPUs should be researched, for instance sort-first and sort-last. The sort-first approach allow us to divide and distribute the screen area to multiple GPUs. The sort-last approach allow us to divide and distribute the geometry to the GPUs. I want to show scenarios where the proposed solution is more performant than a single-GPU configuration. I also want to show that we can utilize the additional memory available with multiple GPUs, which is not possible with SLI or Crossfire.

1.3 Research Questions

1. How can we implement efficient multi-GPU rendering with Vulkan API?
2. How can we divide and distribute rendering workload among multiple GPUs in different ways, and what performance improvement can be expected for the different approaches compared to a single GPU?

1.4 Research Method

In order to answer the research questions, I intend to implement a prototype that support different approaches to multi-GPU rendering, and I will test performance across different hardware configurations, and input. The results of these tests will be compared to each other and to baseline performance tests on a single GPU, for quantifying the improved performance when employing additional GPUs for rendering.

Different approaches to multi-GPU rendering might respond better with different workloads. A performance test that shows increase in framerate for one approach, might not improve performance in a different scenario. In order to answer the second research question, benchmarks tailored for specific approaches will be implemented. These different benchmarks may not be comparable to each other, but they provide information about how performance scales from a single GPU to multiple GPUs with specific approaches.

An example where different workloads are necessary, is with the sort-first and sort-last approaches. With the sort-first approach, we divide the screen area, but each GPU must usually render all the geometry. The sort-last approach divides the geometry, but each GPU renders to the entire screen area. So the sort-first approach can improve performance in scenarios where the framerate increases with smaller screen areas, and the sort-last approach improves performance when the amount of geometry is the bottleneck. There might be scenarios where both approaches could improve performance, but tailored benchmarks allow us to demonstrate the implementation at its best.

1.5 Related Work

Different ways to distribute rendering workload to multiple processors has been described by Molnar et al., 1994. “A Sorting Classification of Parallel Rendering” [10] described parallel rendering as a sorting problem, and introduced the sort-first, sort-middle and sort-last approaches. These approaches differ by the location in the graphics pipeline where redistribution of primitives happen. For the sort-first approach (also known as Split-Frame Rendering (SFR)), redistribution happen during geometry processing, by dividing workload with screen regions. The sort-middle approach is similar to the sort-first approach, but redistribution of primitives happen after screen-space transformation. The sort-last approach defers redistribution until the end of the graphics pipeline, after rasterization, such that pixel fragments are the primitives to redistribute. This can be achieved by dividing and distributing the geometry to the GPUs.

There are existing solutions to multi-GPU rendering, which utilize vendor and hardware specific technologies that link GPUs together. Examples of such technologies are NVIDIA NVLink [11] and AMD XDMA [12]. In 2017, Kim et al. introduced a multi-GPU implementation for SFR rendering [13]. They claim that their multi-GPU architecture, which implements SFR, is fast, sustainable and scalable. However, the focus of this thesis is the use of unlinked GPUs to implement explicit multi-GPU rendering.

A lot of work have been done for multi-GPU volume rendering, since volume rendering is a compute intensive task, and is useful for scientific research and medical screenings. Marchesin et al. proposed a study on multi-GPU sort-last volume visualization in 2008 [14], where they utilized a configuration with single Central Processing Unit (CPU) and multiple GPUs. In 2010, Fogal et al. introduced an implementation of volume rendering with distributed memory multi-GPU clusters, in order to visualize large volumetric datasets. This thesis differ from previous research of multi-GPU volume rendering, because the introduced sort-last implementation will be based on surface rendering, rather than volume rendering.

An implementation utilizing a hybrid approach of sort-first and sort-last to perform rendering with a cluster of computers, was introduced by Samanta et al., 2000 [15]. They achieved efficiency of 70.5% with a cluster of 64 computers. A hybrid sort-first and sort-last implementation for a single computer with multiple GPUs, was introduced in “Multi-GPU Compositeless Parallel Rendering Algorithm” by Wang et al., 2011 [16]. They utilize Direct Memory Access (DMA) asynchronous transfer for image read back from the

GPUs and implicit image compositing. The multi-GPU implementation introduced with this thesis differ in that the image transfers and compositing is implemented explicitly with Vulkan API.

Khronos Group has provided documentation of the Vulkan API with the Vulkan specification [17]. The specification has been a great source when implementing the introduced multi-GPU solution with Vulkan.

Chapter 2

Background

2.1 The Graphics Pipeline

3D graphics allow us to produce a 2D image from 3D meshes. The process of producing pixels from 3D meshes is accelerated on the GPU. This is achieved with a graphics pipeline that consists of stages where the output from one is the input to the next. The graphics pipeline allows us to draw polygons and apply shading and textures.

3D meshes consists of vertices that represents 3D points, and additional attributes like normal vectors, color values, and texture coordinates. The graphics pipeline processes vertices, before combining them into primitives, like points, lines and triangles. In order to represent a point, one vertex is needed. To represent a triangle, three vertices are needed. After the final primitives are produced, the next stage performs rasterization that produces fragments, which are candidates for becoming a pixel. This stage interpolates the vertex attributes to produce fragments between the vertices in a primitive.

Figure 2.1 illustrates how the graphics pipeline produces pixels from three vertices representing a triangle primitive. After vertex processing, the primitives to render resides inside a clip space cube. primitives outside of this cube will not be rendered, and primitives that overlap the cube will be clipped. All primitives inside the clip space will be rasterized into fragments. Unless there is fragment overlap, all fragments will become pixels.

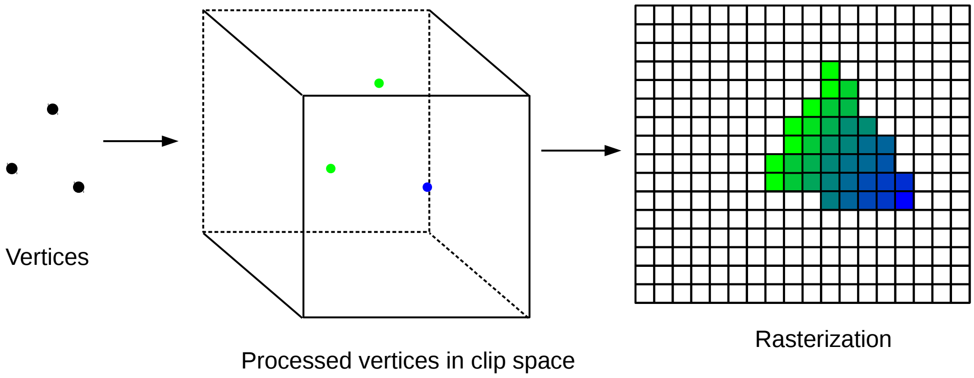


Figure 2.1: Illustration of the rendering process, from the unprocessed vertices to the final pixels on the screen.

2.1.1 Pipeline Stages

The graphics pipeline consists of stages where the output from one is input to the next. Some pipeline stages are customizable by implementing shader programs. Figure 2.2 illustrates the pipeline stages available for modern graphics APIs like OpenGL 4.0 and up, Vulkan API, and Direct3D 11 and up. The yellow stages are customizable with shader programs, while the blue stages are fixed-function.

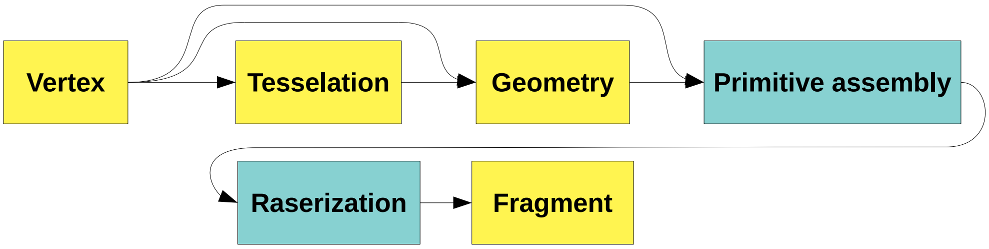


Figure 2.2: Illustration of the graphics pipeline stages, of which the yellow stages are programmable.

The first stage in the graphics pipeline is the programmable vertex shader. This shader processes the vertices before they are combined into primitives. While programming a vertex shader, one can set up vertex attributes. These attributes are scalars, points or vectors that can be used to calculate the final vertex position, and additional output. Examples of additional attributes to a vertex shader is color values, normal vectors for calculating shading, and texture coordinates used to map a 2D texture image to the polygon.

The next stage in the pipeline is the optional tessellation stage [18]. Tessellation allows us to divide the primitives into smaller primitives, to smooth out hard edges of the mesh. To use tessellation one must implement a Tesselation Evaluation Shader (TES), which is responsible for computing the values for the tessellated vertices. Optionally, a Tesselation Control Shader (TCS) can be implemented, to determine how much tessellation to apply, and transform the input data.

After the vertex shader (and tessellation if implemented), we have the option to implement a geometry shader. The geometry shader takes as input primitives from the previous pipeline stages. We can use these primitives as well as user defined attributes to create the final primitives. The geometry shader can add additional vertices, for instance generating a geometry for each point in a vertex buffer. In theory, it is possible to implement tessellation in a geometry shader, though it would not be as efficient as doing it in the tessellation stage.

The next two stages are non-programmable, fixed-function stages. First comes primitive assembly, which takes care of clipping, backface culling and viewport transform of the primitives from the previous stage(s). Clipping removes any primitives that fall outside of the clip space cube, backface culling removes primitives that face away from the user. The viewport transform, transforms the primitives to screen coordinates. These screen coordinate primitives will then be sent to the rasterization stage. This stage produces the fragments, which are candidates for becoming pixels, by interpolating positions and user provided attributes for the primitives, which will be the input of the fragment shader.

The final pipeline stage operates on fragments. The fragment shader is a programmable stage, which allows the programmer to set the final color of the fragment, based on the position and user provided attributes. Examples of additional input to the fragment shader is normal vectors and texture coordinates. The input to a fragment shader has been interpolated in the rasterization stage. This allows gradual change in normal vectors, which can be used for calculating shading. It is also useful for sampling a color value from a texture, with texture coordinates interpolated for each fragment.

After fragment processing with the fragment shader, alpha blending and depth testing is performed to produce the final pixels of the framebuffer. Alpha blending merges fragments on the same screen coordinate based upon the alpha components from the fragments. This can be used to implement transparency. An alpha component of 1 is considered 100% opaque, while 0 is considered invisible. The depth test selects the fragments with the closest depth value as candidate for the final pixel.

2.2 Multi-GPU Configurations

There are multiple ways to configure a multi-GPU system. In this thesis, we only consider configurations of a single computer with a single CPU and multiple dedicated GPUs. The GPUs are connected to Peripheral Component Interconnect Express (PCI-E) slots on the motherboard of the computer. In some configurations, the GPUs can be linked together with an additional cable, or through the PCI-E bus, such that they can share resources like buffers and textures directly with each other. Generally, we can categorize multi-GPU configurations by whether or not the GPUs are linked together, and if they are handled implicitly by the graphics driver with multi-GPU profiles for SLI or Crossfire, or explicitly by the programmer.

Dedicated GPUs are connected to the CPU through PCI-E (see Figure 2.3). The CPU has a fixed number of PCI-E lanes to distribute. With PCI-E version 3.0, each lane has a bandwidth of 985 MB/s. GPUs are connected to 16-lane slots, though in some configurations, less lanes are available from the slot. Most GPUs can utilize 16 lanes, but some low-end GPUs utilize less lanes than available. When multiple GPUs are connected, there might be less lanes available for each GPU, depending on how many lanes the CPU supports.

High-End Desktop (HEDT) CPUs usually have more PCI-E lanes than mid-range products. An example of a desktop processor is Intels Core i7-8700K [19], a 6-core CPU which supports a total of 16 PCI-E lanes. Examples of HEDT CPUs are AMDs Ryzen Threadripper processors [20], which supports a total of 64 PCI-E lanes. 64 PCI-E lanes is enough to provide full bandwidth to four GPUs, while 16 lanes is enough to provide full bandwidth to a single GPU. When connecting a second GPU in such a setup, each GPU will get 8 PCI-E lanes.

Some configurations support linking multiple GPUs together, to be able to share resources directly rather than first having to copy the resources to main memory. SLI [3] is an interface used to link NVIDIA GPUs together with a separate cable. AMD GPUs can also be linked together, using their

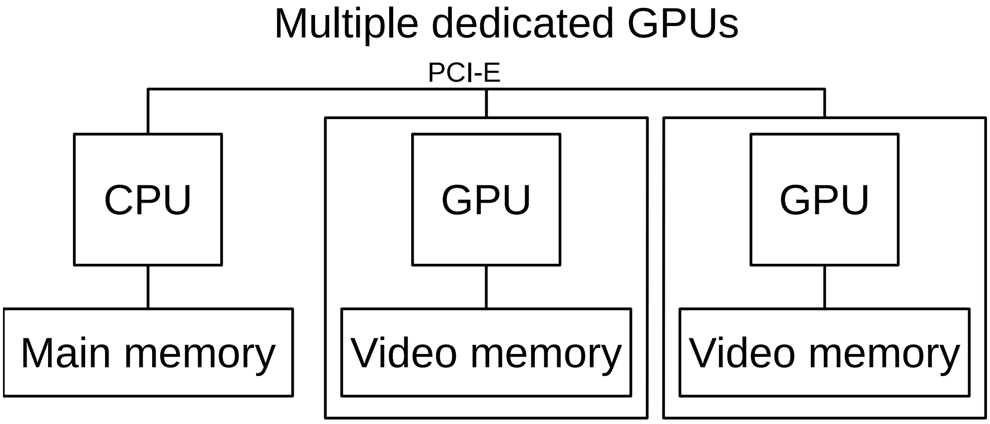


Figure 2.3: Illustration of a configuration with dedicated GPUs.

Crossfire technology [4]. AMD CrossFire Direct Memory Access (XDMA), is a more recent Crossfire implementation for AMD GPUs, where the GPUs are linked together through the PCI-E bus, rather than a separate link cable [12]. The XDMA data channel scales dynamically on demand, to provide enough communication bandwidth between the GPUs.

Configurations with linked GPUs require hardware support on the motherboard. Usually it is also required that the GPUs connected together must be very similar or the same model. Linked GPU configurations usually mirror the memory across the GPUs, such that the amount of available GPU memory doesn't increase when adding more GPUs. However with an unlinked explicit approach, as implemented in this thesis, it is possible to utilize all of the memory available for multiple GPUs.

Many modern CPUs also contain an integrated GPU (see Figure 2.4). Some configurations may allow this integrated GPU to be linked to an additional dedicated GPUs, for example some AMD configurations can utilize XDMA for multi-GPU with an integrated- and a dedicated GPU.

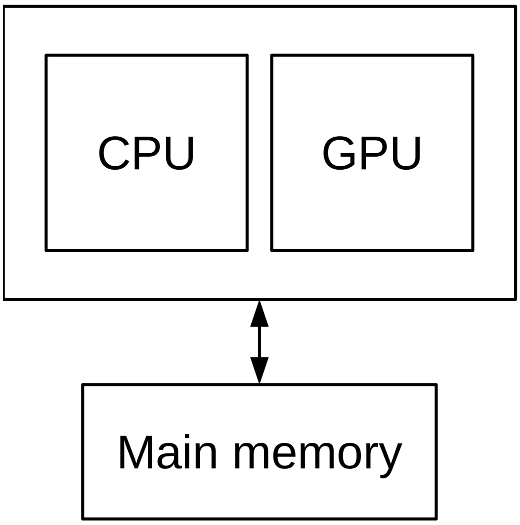


Figure 2.4: Illustration of a GPU integrated in the same package as the CPU.

2.2.1 Linked Implicit Multi-GPU

With OpenGL and Direct3D 11 or older, multi-GPU configurations with linked GPUs need support in the driver software for the GPUs. In this case the driver takes care of distributing the rendering workload among the GPUs implicitly. Linked implicit multi-GPU configurations appear to the programmer as a single GPU. The GPU drivers must provide profiles, which usually are application specific, to optimize the distribution of rendering workload.

Linked implicit multi-GPU configurations provide two approaches to divide the rendering workload, SFR and AFR. The SFR (also known as sort-first) approach subdivides the screen such that each GPU renders its partition of the screen. This reduces the time it takes to render a frame, which also lead to an increase of framerate. With the AFR approach, the GPUs renders alternate frames. While the time it takes to render a frame is not reduced with AFR, the framerate is increased, as the GPUs can render multiple frames in parallel.

2.2.2 Linked Explicit Multi-GPU

Modern lower level graphics API's, like Vulkan and Direct3D 12, do not support implicit handling of multiple GPUs. However they have introduced support for linked explicit multi-GPU configurations. These configurations support the same approaches as implicit configurations, SFR and AFR, but the programmer must explicitly distribute the rendering workload to the GPUs.

In Direct3D 12 linked explicit multi-GPU is known as explicit multi-adapter [21]. Vulkan 1.0 has experimental support through an extension (`VK_KHX_device_group`) [17, p. 1216]. As of Vulkan version 1.1, this extension is officially supported [22], with the name `VK_KHR_device_group`.

2.2.3 Unlinked Explicit Multi-GPU

The multi-GPU solution introduced with this thesis utilize unlinked explicit multi-GPU configurations. This approach can be implemented with graphics APIs that support explicit selection of GPUs, such as Vulkan or Direct3D 12. In this case, the programmer must manage the GPUs separately and explicitly, as they are not linked together. Unlinked explicit multi-GPU configurations can be heterogeneous such that GPUs of different models and from different vendors can be utilized.

When implementing a multi-GPU application with an unlinked explicit configuration, the programmer must implement the subdivision and distribution of the rendering workload, as well as combining the results from the GPUs into the final image. In order to copy resources like buffers and textures between unlinked GPUs, the resources must first be copied to the main memory, before they can be copied to the destination GPU.

2.3 Parallel Rendering Approaches

There are multiple possible approaches to implementing parallel rendering utilizing an unlinked explicit multi-GPU configuration. The approaches differ by how the rendering workload is divided and distributed among the GPUs. The introduced multi-GPU implementation can perform rendering with the sort-first and sort-last approaches.

2.3.1 Sort-first

Sort-first is defined as an approach where you redistribute primitives early in the graphics pipeline, during geometry processing [10]. This can be done by dividing the screen such that each GPU renders their own region (see Figure 2.5). When all GPUs are done rendering the frame, the final image would be composited from the contributing regions. SFR is a form of sort-first rendering. Sort-first is also referred to as image-based partitioning.

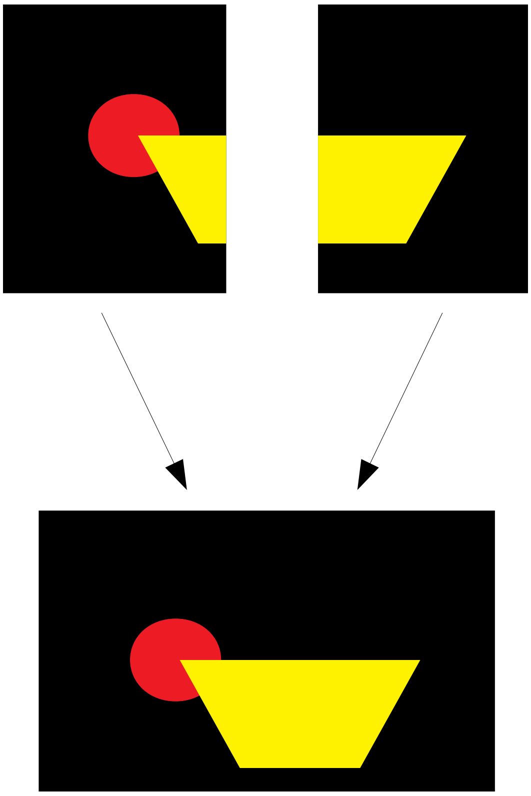


Figure 2.5: Illustration of the sort-first parallel rendering approach.

2.3.2 Sort-last

Sort-last is defined as deferring primitive redistribution until the end of the graphics pipeline [10]. An implementation of sort-last could be dividing the primitives such that the GPUs render their own portion into framebuffers. Then these framebuffers can be composited into the final image by merging them together while applying a depth test (see Figure 2.6). This is known as sort-last-full or sort-last-image, as all the pixels in the contributing framebuffers are tested upon compositing. The depth test makes sure that closer fragments are rendered in front of fragments with depth further away. This approach does not support transparency with alpha blending, as only the depth information for the frontmost fragment is stored in the depth buffer of the framebuffer. Sort-last is also referred to as object-based partitioning as the triangles (objects) are divided and distributed among the GPUs.

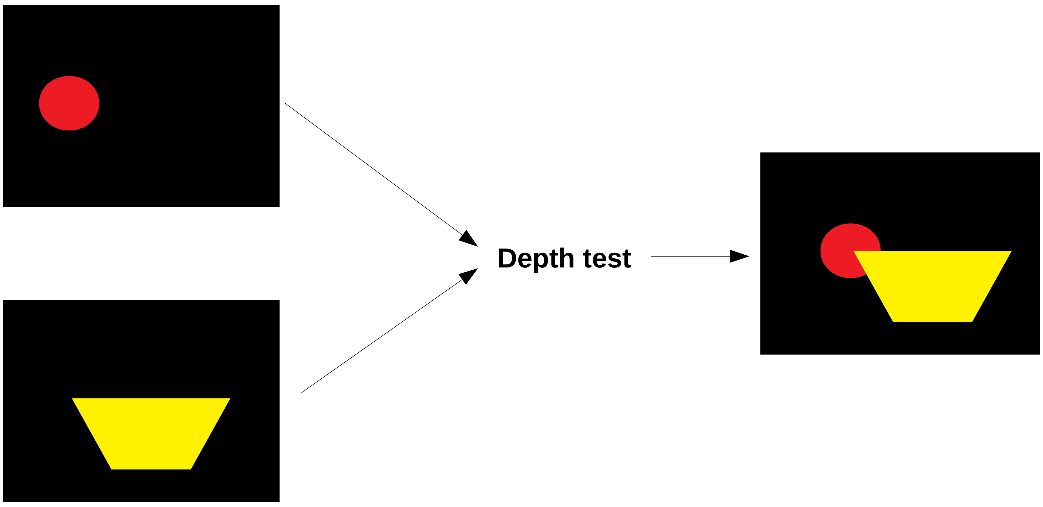


Figure 2.6: Illustration of the sort-last parallel rendering approach.

An implementation where only the drawn pixels are redistributed and tested, is known as sort-last-sparse [10]. However since we are rendering to a framebuffer, a sparse approach is not viable for our multi-GPU implementation, as only transferring the drawn pixels from a GPU is not a trivial task. To do this, we would need to know which pixels to copy up front.

2.3.3 Other Approaches

Another approach to implement parallel rendering is sort-middle [10], which implements redistribution of primitives in the middle of the graphics pipeline. The sort-middle approach could be implemented by extracting screen-space primitives from the graphics pipeline after vertex processing, for instance with transform feedback on OpenGL [23]. Vulkan does not support transform feedback, however, the same functionality could be achieved by writing the primitives to a buffer from a geometry shader. This approach has not been implemented in the solution for this thesis, due to the complexity of the primitive redistribution.

Other approaches could be scheduling different tasks of the rendering process to different GPUs, like rendering shadow maps, or computing physics calculations. Hybrid approaches can also be implemented. For instance, a hybrid of the sort-first and sort-last approaches could be to apply sort-last to partitions of the screen before merging.

2.4 Compositing

When implementing the sort-first and sort-last approaches, with an unlinked, explicit multi-GPU configuration, one significant difficulty is the lack of shared memory between the GPUs. The framebuffer that is presented to the user resides locally in the memory of a single GPU. We will refer to this GPU as the primary GPU. The other GPUs, referred to as secondary GPUs, do not have direct access to the final framebuffer. Each GPU has to render to its own framebuffer. After rendering has completed, these contributing framebuffers then needs to be combined into the final image, a step called compositing.

2.4.1 Compositing into Framebuffer

One way to combine the result is to copy the contributing image from each GPU into the final image. The image can be combined either on the CPU, or on the GPU that holds the final framebuffer. For the latter approach, the contributing images must be copied from the secondary GPUs to the primary GPU before the compositing step can be executed.

2.4.2 Using Multiple Windows or Monitors

Borderless windows are windows without titlebars or edges that surround the content. For this thesis, we had an idea that for the sort-first approach, it is possible to use multiple borderless windows carefully positioned next to each other, in order to combine the results from different GPUs. This way, compositing is handled by the window compositor implemented by the OS. This approach is not always supported, as there is no guarantee that all GPUs can present its result to an OS window. Our experience is that this approach will work for the Windows OS on a locally accessed computer, though we were unable to use this method on a computer accessed through external desktop. The experience we had with Linux (GNOME desktop environment), was that there was visible fading effects at the edge of windows. This effect could possibly be disabled in order to utilize this approach without visible edges.

For multi-monitor configurations, it is possible to assign one GPU for each monitor. This approach is compositeless, as at no point is the results combined into a single framebuffer. The result appear to the user as a single image as the monitors are placed next to each other.

2.5 Vulkan API

The Vulkan API provides low-level explicit access to GPUs allowing the developer to allocate resources, and perform rendering and compute operations. The low-level nature of the API reduces graphics driver overhead, as the API represents a closer mapping to the actual hardware. However, a lower level API expects the programmer to explicitly set up the state needed for rendering, which means more upfront work for the programmer. This is why abstractions of Vulkan objects are necessary. This thesis introduces a library that provides abstractions to the Vulkan details explained in this section, and utilizes explicit selection for implementing multi-GPU rendering.

Vulkan has no global state, as all state is represented by objects, as opposed to OpenGL that depends on a global context, which must be bound to the thread using the API [7]. Having all state stored in objects provides the opportunity to implement multi-threading without context switches, as there is no global thread-bound context to take into account. Vulkan supports recording GPU commands from multiple threads, which can be utilized to speed up the rendering process of a large amount of objects. With OpenGL, commands must be called sequentially on the thread responsible for the OpenGL context.

2.5.1 Instance and Physical Devices

When implementing an application utilizing Vulkan, we first create a `VkInstance` [17, p. 38], from which we can enumerate `VkPhysicalDevice` handles [17, p. 44] (see Figure 2.7). These handles represents the Vulkan capable GPUs in our system. The introduced multi-GPU solution implements explicit selection of which GPU to use for rendering, by selecting physical devices from this enumerated list.

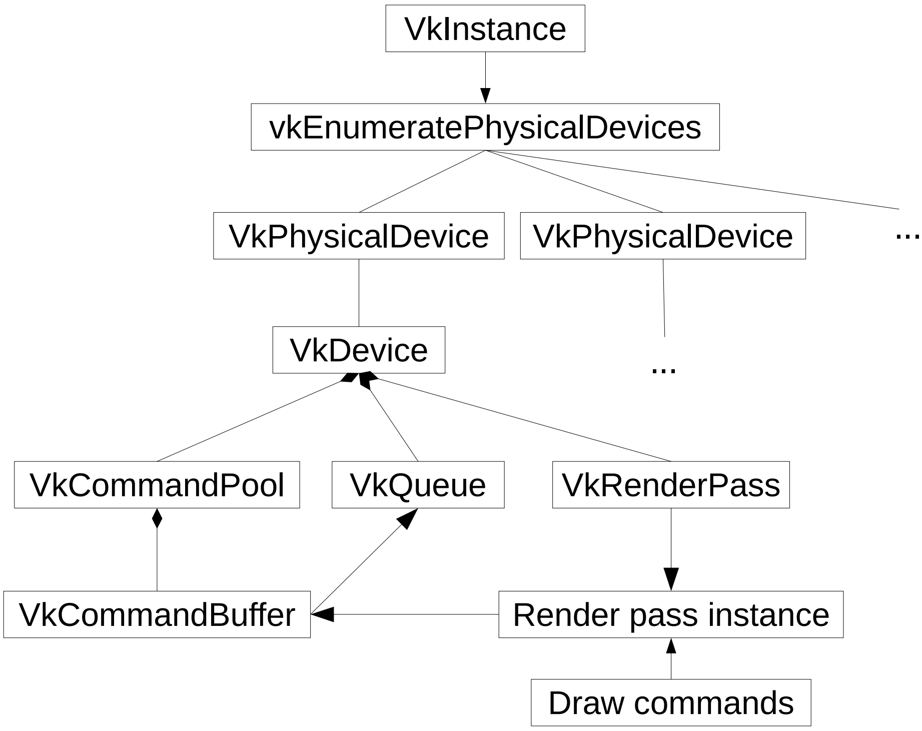


Figure 2.7: Illustration of Vulkan objects and how they relate. Draw commands and the render pass instance they belong to are recorded into a command buffer. Command buffers are submitted to queues.

2.5.2 Logical Devices, Queues and Commands

All interaction with a GPU happens through a logical device, represented by a `VkDevice` handle [17, p. 56]. The logical device represents a subset of the functionality available for a physical device. By specifying only the functionality in use, the GPU driver software can optimize the usage for minimal overhead and best performance. After we have selected which GPUs (`VkPhysicalDevice` handles) to use for rendering, we have to create one logical device for each of them, from which we can execute rendering commands.

From a logical device we can create resources like buffers and images, allocate device memory, execute commands and bind resources to pipelines. We can perform rendering with graphics pipelines, and execute compute shaders with compute pipelines. Drawing with graphics pipelines must happen inside a render pass instance (discussed in section 2.5.5).

To execute commands on a GPU, such as drawing triangles, they must first be recorded into a command buffer. A command buffer, represented by `VkCommandBuffer` handles, are batches of commands, which can be executed on a GPU. Command buffers must be allocated from command pools, represented by `VkCommandPool` [17, p. 70] handles, which must be created from the logical device. In order to fill a command buffer with commands, the command buffer must be switched to the recording state with `vkBeginCommandBuffer`, before recording the commands to it. Before we can submit the command buffer for execution it must be switched to the executable state by calling `vkEndCommandBuffer`.

In order to execute the commands in a command buffer, it must be submitted to a queue, represented by a `VkQueue` handle. We have to specify which queues to create upon creation of the logical device. Queues belong to queue families that describe the capabilities of the queue, for instance graphics-, compute- and transfer capabilities [17, p. 63]. Information about available queue families can be queried from the physical device. When multiple queues are created, we can specify queue priority in order to hint to the GPU which queues to prioritize more, when command buffers are submitted to multiple queues. Many GPUs supports creating a dedicated queue for transfer operations, which can be used for transferring data to and from the GPU in parallel with rendering or compute tasks.

2.5.3 Memory and Resources

There are two types of resources in Vulkan, buffers, represented by `VkBuffer` handles [17, p. 335], and images, represented by `VkImage` handles [17, p. 345]. The resources and the GPU memory are separated in Vulkan, which is why resources must be bound to memory objects before use. These memory objects, represented by `VkDeviceMemory` handles [17, p. 293], are allocated from the logical device.

The memory available for a GPU in Vulkan is represented by heaps and memory types. This information can be queried from a `VkPhysicalDevice` into a `VkPhysicalDeviceMemoryProperties` struct [17, p. 293]. A memory type refers to a specific memory heap, and contain flags which describes properties of the memory. Available memory properties are [17, p. 298]:

```
typedef enum VkMemoryPropertyFlagBits {
  VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
  VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
  VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
  VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
  VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
} VkMemoryPropertyFlagBits;
```

Listing 2.1: Available memory property flags

If the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` flag is set, the memory type in question represents GPU memory (device local). Otherwise, the memory type represents host local memory, which resides in main memory. The `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` flag represents whether or not the memory type is visible to the host (CPU).

Memory Transfers

Device local memory is usually not visible to the host, so in order to transfer data to GPU memory, it must first be loaded into a host local staging buffer, which can then be copied to the device local memory through the API. The reverse process must be done in order to transfer data from the GPU to the host, for instance the results from rendering. A host visible buffer can be mapped to a pointer, which the CPU can read from and write to.

Data transfers between Vulkan resources like images and buffers, must be executed with copy commands submitted to a queue. These commands does not support transferring data between resources from different logical devices, so in order to copy data from one GPU to another, we need to copy the data between host visible staging buffers. However, for supported GPUs it is possible to use an extension for sharing memory between logical devices. The extension `VK_EXT_external_memory_host` [17, p. 1247] provides this functionality, but it is not guaranteed to be supported by all GPUs.

2.5.4 Shaders and Pipelines

Vulkan only supports shaders in the open byte code format: Standard Portable Intermediate Representation (SPIR-V) [24]. To use shaders, we must create shader modules represented by `VkShaderModule` handles [17, p. 229]. The shader modules are combined in pipelines, represented by `VkPipeline` handles [17, p. 247]. We must bind a graphics pipeline in a render pass instance before our draw commands. We can dispatch compute shaders with compute pipelines.

In addition to combining the shader modules, the pipeline objects also encapsulate information about the fixed function stages of the graphics pipeline. How primitives should be assembled, how rasterization should happen, and information about blending are all represented by the pipeline objects.

Upon creation of a pipeline we must provide a pipeline layout object. A pipeline layout, represented by `VkPipelineLayout` handles [17, p. 442], holds information about what kind of resources that can be accessed in a shader. A small amount of data to pass to a shader can be represented by push constants. Push constants are data that are added directly to the command buffer for usage within a shader. A push constant range must be passed when creating a pipeline layout, to be able to use push constants for pipelines using this layout. Other resources like images and buffers, must be described by descriptors. Descriptor set layouts, which hold information about what resources can be bound to a pipeline, are passed upon creation of the pipeline layout.

2.5.5 Rendering to Framebuffers with Render Passes

Render passes, represented by `VkRenderPass` handles [17, p. 187], encapsulates attachments, subpasses and subpass dependencies. Drawing commands can only be recorded within a subpass of a render pass instance. Multiple of these subpasses can be executed in parallel, but they are recorded one at a time in a command buffer. Commands to be executed within a render pass instance must be recorded between `vkCmdBeginRenderPass` and `vkCmdEndRenderPass`. Subpasses are separated with `vkCmdNextSubpass` [17, p. 217-228]. When beginning the render pass, we must pass a framebuffer object, represented by a `VkFramebuffer` handle. This framebuffer encapsulates information about the specific attachments that will be used in the render pass.

2.5.6 Binding Resources to Shaders

When we create a pipeline layout, we can specify descriptor set layouts that describes what kind of resources that can be bound. Descriptor set layouts are represented by `VkDescriptorSetLayout` handles [17, p. 434]. A descriptor set layout encapsulate a set of bindings that describe what types of descriptors can be bound, and to which shader stages they can be bound to.

When recording the draw commands, we must bind descriptor sets for each of the descriptor set layouts that were specified upon creation of the pipeline layout for the bound pipeline. Descriptor sets are represented by `VkDescriptorSet` handles and must be allocated from descriptor pools, which are represented by `VkDescriptorPool` handles [17, p. 455-458]. Descriptors representing the specific resources to use in a shader, must be written to the descriptor set before use with descriptor updates [17, p. 459-467].

2.5.7 Synchronization

Commands recorded into a command buffer are not necessarily executed sequentially on the GPU. Commands not dependent on one another may be executed in parallel, depending on the GPU and driver software. To ensure the order of execution is correct, we must be able to synchronize between commands, queues, submissions, and between host (CPU) and device.

Synchronization between host and device can be done with fences and events. A fence is a synchronization object that can have two possible states, signaled or unsignaled. Fences are represented by `VkFence` handles [17, p. 118-140], and provide us the ability to query the state, reset the fence to the unsignaled state, or wait for the fence to be signaled on the host. Upon submitting commands to a queue, we can pass a fence to be signaled when execution of the commands have finished. The function `vkQueueWaitIdle` [17, p. 184] also allows us to wait for the commands executing on a queue to finish. Waiting for a queue is equivalent to passing a fence upon submission and waiting for the fence to be signaled on the host.

Events, represented by `VkEvent` handles [17, p. 158-170], can be used to synchronize host and device, but also synchronize between commands on the same queue. Similarly to fences, events can be either signaled or unsignaled. On the host, we can query the state, signal and unsignal an event. On the device (command buffers submitted to a queue), we can signal and unsignal an event, as well as waiting for an event to be signaled.

Semaphores, represented by `VkSemaphore` handles [17, p. 141-158], provides synchronization between queue submissions. Upon submitting command buffers to the queue, we can specify semaphores that should be waited upon before execution. Each semaphore can be waited upon in different pipeline stages. One semaphore can for instance be waited upon at the vertex pipeline stage, while another semaphore can be waited upon at the fragment stage. As well as specifying wait semaphores, we can also specify semaphores to signal once execution of the submission is done.

In order to insert dependencies between commands we can use pipeline barriers [17, p. 170-176]. Pipeline barries provide dependencies between commands submitted on the same queue, or in the case of a render pass instance, commands in the same subpass. Memory barriers provide the ability to synchronize the use of specific memory between queues, or to change image layouts [17, p. 176-184]. Memory barriers can be submitted as part of a pipeline barrier [17, p. 171], or when waiting upon an event [17, p. 167].

Chapter 3

Design and Solution

The implemented solution to multi-GPU rendering with Vulkan consists of an abstraction library, and a prototype application using the multi-GPU functionality of the library. The library provides abstractions of low-level Vulkan details, which is useful, as implementing an application utilizing Vulkan requires much upfront work to setup the objects needed to perform rendering. The library also contain a module for taking care of multi-GPU details. The prototype application supports loading 3D models from files, and specifying multi-GPU approach and how many GPUs to use.

3.1 Technologies

The solution is implemented in the C++ programming language. C++ supports using C APIs and libraries, while having a rich standard library and support for classes. The solution depends on the following APIs, libraries and frameworks:

- Vulkan API [2] - The C graphics- and compute API used to implement the multi-GPU solution.
- Qt [25] - A C++ GUI framework, used to represent a window in the application.
- GLFW [26] - A C library that can provide a window, used as an alternative to Qt for window representation. It provides a simpler way to manage windows, without depending on the large Qt framework.

- GLM [27] - A C++ math library used to implement the scene and mesh support for the solution.
- Vulkan Memory Allocator [28] - A C/C++ library for managing Vulkan memory.
- LunarG Vulkan SDK [29] - A software development kit for Vulkan API, providing function loading and debugging layers.
- stb_image [30] - A C library for loading images from files.
- tiny_obj_loader [31] - A C++ library for loading 3D models in the Wavefront obj format.
- boost [32] - A collection of portable C++ libraries. The filesystem library was used for creating a list of files to load, and the program_options library was used for parsing command line options.

In order to build the project from source, the following tools have been used:

- CMake [33] - Cross platform build system used to describe the build process and generate platform specific build files.
- glslang [34] - Reference compiler for the GLSL shader programming language, used to validate shader source and compile to SPIR-V bytecode.

3.2 A Vulkan Abstraction Library

To take care of the low-level Vulkan details, we have made publicly available the library Boilerplate (BP) [35], a cross platform abstraction library implemented along with the multi-GPU application. BP consists of C++ wrappers of Vulkan objects, and abstractions hiding complexity of the API.

The core module `bp` provides the wrappers and abstractions of Vulkan, and additional modules can be added for creating e.g. windows and scenes. The module `bpView` provides a window wrapper class based upon the library GLFW [26], while the `bpQt` module provides an abstract window class implemented with the Qt framework [25]. It is possible to use BP with other window libraries, for instance SDL [36], or OS specific APIs.

A basic scene graph implementation with mesh support is implemented in the `bpScene` module. The scene graph is represented by nodes that provide transformations relative to parent nodes. The scene module also provide classes used to render the scene in the prototype application.

Another module called `bpMulti`, implements abstractions for multi-GPU rendering with the sort-first and sort-last approaches. This abstraction allows passing renderers to a compositor, that takes care of the multi-threading and synchronization details of rendering with multiple GPUs, as well as combining the result and presenting it to a framebuffer.

While many of the classes implemented with `bp` are simple wrappers of Vulkan objects, some of the classes provide a higher level abstraction that hides complexitites of Vulkan. Examples of the latter are devices, memory allocators, attachments, subpasses and render passes.

3.2.1 Instance and Device

The first step when implementing an application utilizing Vulkan API, is to create an instance object, representing per-application state, as explained in section 2.5.1. If the intention is to use the `bpQt` module for creating a Qt window, a `QVulkanInstance` object must be created, to represent the Vulkan instance. Otherwise, we can use the abstraction `bp::Instance` together with the `bpView` module, which is a wrapper of the library GLFW [26]. We can also use other window libraries that can provide a `VkSurfaceKHR` representation of the window.

The following code will enable validation layers for debug builds and disable it for release builds. `NDEBUG` is only defined for release builds. In order for the debugging layers to be of any use, we must provide a delegate for printing errors, and connect it to the error event:

```
#include <bp/Instance.h>
#include <iostream>

void printErr(const std::string& s) {
    std::cerr << s << std::endl;
}

int main(int argc, char** argv) {
#ifdef NDEBUG
    bool enableDebug = false;
#else
    bool enableDebug = true;
#endif
```

```
    bp::Instance instance;
    instance.init(enableDebug);
    bpUtil::connect(instance.errorEvent, printErr);
}
```

Listing 3.1: Create instance with validation layers enabled

The error event provided by the instance is a `bpUtil::Event` object that provide a simple event-delegate implementation. It is implemented with the C++ parameter pack feature [37], that allows using varying numbers of template parameters. The event behaves as a function that calls all connected delegates and the template parameters defines what kind of arguments can be passed when calling the event. This event-delegate implementation behaves very similar to the well known Qt signals and slots [38]. It allows the `bp` library to have less dependencies, however Qt is necessary if it is desired to implement a window with `bpQt`.

After creating the instance object, we can query for available GPUs from the instance. To do this, we must specify what queues should be available, what features to use, and whether or not the GPU should be capable of presenting its result to a specific window surface. This information is encapsulated in a `bp::DeviceRequirements` struct. The function `bp::queryDevices` returns a list of the `VkPhysicalDevice` handles that represents the available GPUs. It will filter out unsuitable devices that does not support the required features or cannot present to the given window surface.

The next step is to create a logical device, represented by `bp::Device` for each of the GPUs that should be used. We can either select a physical device from the list return by `bp::queryDevices`, or pass the instance directly into the constructor of `bp::Device`, which queries for suitable devices and uses the first device from the list. After including the header `bp/Device.h`, we can create a logical device:

```
bp::DeviceRequirements requirements;
requirements.queues = VK_QUEUE_GRAPHICS_BIT
                    | VK_QUEUE_TRANSFER_BIT;
requirements.features.samplerAnisotropy = VK_TRUE;

bp::Device logicalDevice{instance, requirements};
```

Listing 3.2: Create logical device

We create a logical device that provides both a queue for graphics- and for transfer operations. Though graphics- and compute capable queues will always be able to perform transfer commands, by specifying the transfer queue flag, the library will try to create a separate transfer queue if possible. If a separate transfer queue is available, we can execute transfer commands in parallel with graphics commands. The device also enables the anisotropic filtering feature, used when sampling textures.

The created `bp::Device` can be used as an in-place replacement for both `VkPhysicalDevice` handles and `VkDevice` handles, as it will be implicitly casted to the proper handle. The device can be used directly in Vulkan API functions. Implicit casting to Vulkan handles has been implemented for all the wrapper classes provided by BP, for instance, a `bp::Image` object can be implicitly casted to a `VkImage` handle.

A `bp::MemoryAllocator` object is provided by the device, which is used to allocate memory for buffers or images. This abstraction is a wrapper of Vulkan Memory Allocator [28], an open source library from the GPUOpen initiative by AMD. The memory allocator allocates memory optimally in large chunks, rather than using small allocations for each resource.

3.2.2 Implementing a Qt Window

We can implement window creation and handle system events with Qt, by using the module `bpQt`. We can render to a Vulkan window by inheriting `bpQt::Window`:

```
//MyWindow.h
#ifndef MYWINDOW_H
#define MYWINDOW_H

#include <bpQt/Window.h>

class MyWindow : public bpQt::Window {
    void initRenderResources(uint32_t width, uint32_t height)
    {
        //Initialize vulkan resources
    }

    void render(VkCommandBuffer cmdBuffer) {
        //Record rendering operations into
        //command buffer
    }
};

#endif
```

Listing 3.3: Implement a Qt window

We can also override methods to select which physical device to create a logical device from, as well as a method for resizing resources. When using bpQt we must create a QVulkanInstance and pass it to the window:

```
#include <QGuiApplication>
#include <QLoggingCategory>
#include "MyWindow.h"

int main(int argc, char** argv) {
    QGuiApplication app(argc, argv);
    QVulkanInstance instance;

#ifndef NDEBUG
    //Enable logging and validation layers for debug builds
    auto filterRules = QStringLiteral("qt.vulkan=true");
    QLoggingCategory::setFilterRules(filterRules);
    auto layers = QByteArrayList();
    layers << "VK_LAYER_LUNARG_standard_validation";
    instance.setLayers(layers);
#endif

    if (!instance.create()) {
        qFatal("Failed to create instance.");
    }

    MyWindow window;
    window.setVulkanInstance(&instance);
    window.resize(1024, 768);
    window.show();

    return app.exec();
}
```

Listing 3.4: Setup QInstance and create window

3.2.3 Rendering with BP

The render pass abstraction provided by BP, `bp::RenderPass`, takes care of recording the commands for a render pass instance, and the subpasses within that render pass instance. Subpasses, represented by an abstract class `bp::Subpass`, must be added to the render pass before initialization.

Draw commands can only be executed inside a subpass of a render pass instance. To implement a subpass, we must inherit `bp::Subpass` and override the `render` method. Subpass dependencies can be added to a subpass with the `addDependency` method. Attachments to use in a subpass are represented by bp::AttachmentSlot objects. Subpass dependencies, as well as attachment slots must be set before adding the subpasses to a render pass.

```
#include <bp/Subpass>
#include <bp/GraphicsPipeline>

class SimpleSubpass : public bp::Subpass {
    bp::GraphicsPipeline& graphicsPipeline;
public:
    SimpleSubpass(bp::GraphicsPipeline& pipeline) :
        graphicsPipeline{pipeline} {}

    void render(const VkRect2D& area,
                VkCommandBuffer cmdBuffer) {
        VkViewport viewport = {
            static_cast<float>(area.offset.x),
            static_cast<float>(area.offset.y),
            static_cast<float>(area.extent.width),
            static_cast<float>(area.extent.height),
            0.f, 1.f
        };

        vkCmdBindPipeline(cmdBuffer,
                          VK_PIPELINE_BIND_POINT_GRAPHICS,
                          graphicsPipeline);
        vkCmdSetViewport(cmdBuffer, 0, 1, &viewport);
        vkCmdSetScissor(cmdBuffer, 0, 1, &area);
        vkCmdDraw(cmdBuffer, 3, 1, 0, 0);
    }
}
```

Listing 3.5: Implement a simple subpass

The above code implements a simple subpass example, for executing a single draw command. Note that this subpass does not bind any resources to the pipeline. It simply executes a draw command for drawing three vertices with the graphics pipeline passed to the constructor. This would work for an example where the vertices are provided by, or calculated in the vertex shader. For rendering a mesh, the subpass would need to bind vertex buffers, and possibly an index buffer to the pipeline. A small amount of data, for instance a few matrices can be passed to the pipeline with push constants. Other resources, like textures or uniform buffers, can be bound to the pipeline with descriptor sets.

The graphics pipeline wrapper bp::GraphicsPipeline uses dynamic viewport and scissor, which allows reusing the pipeline for different framebuffer sizes. The viewport and the corresponding scissor rectangle determines the viewport transformation, and region of the framebuffer to render to. In this case we fill the entire render area passed from the render pass.

When executing a render pass instance, we must provide a framebuffer, represented by `bp::Framebuffer`, that holds attachments for each of the attachment slots in use by the subpasses. The abstract class `bp::Attachment` represents a framebuffer attachment, and is inherited by `bp::Texture` and `bp::Swapchain`. A `bp::Texture` object encapsulates a single offscreen image, and a `bp::Swapchain` represents a series of images used to render to a window or display. A simple renderer would usually implement a swapchain as color attachment, and a texture as depth attachment.

To execute the commands on the GPU we must submit the recorded command buffer to a graphics-capable queue. When the commands have finished executing, our attachments will contain the rendering result.

To set up a render pass, we must first create a subpass and set up the attachment slots that represents the attachments to use in the subpass. To render we must first create a command buffer to record the draw commands to, as well as creating a framebuffer we can use as a render target. Given that we have already created a graphics pipeline, the following code will set up a render pass and execute the rendering commands on the GPU:

```
#include <bp/RenderPass.h>
#include <bp/OffscreenFramebuffer.h>
#include <bp/CommandPool.h>

...

SimpleSubpass subpass{graphicsPipeline};

//Create attachment slot for RGBA texture with clear enabled
bp::AttachmentSlot colorAttachmentSlot{
    VK_FORMAT_R8G8B8A8_UNORM,
    VK_SAMPLE_COUNT_1_BIT,
    VK_ATTACHMENT_LOAD_OP_CLEAR,
    VK_ATTACHMENT_STORE_OP_STORE,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};

//Create attachment slot for depth texture
bp::AttachmentSlot depthAttachmentSlot{
    VK_FORMAT_D16_UNORM,
    VK_SAMPLE_COUNT_1_BIT,
    VK_ATTACHMENT_LOAD_OP_CLEAR,
    VK_ATTACHMENT_STORE_OP_STORE,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL};

subpass.addColorAttachment(colorAttachmentSlot);
subpass.setDepthAttachment(depthAttachmentSlot);

//Setup render pass
bp::RenderPass renderPass;
renderPass.addSubpassGraph(subpass);
renderPass.init(device);
renderPass.setRenderArea({{}, {WIDTH, HEIGHT}});

//Setup offscreen framebuffer
bp::OffscreenFramebuffer framebuffer;
framebuffer.init(renderPass, WIDTH, HEIGHT,
                 colorAttachmentSlot, depthAttachmentSlot);

//Create command pool and allocate command buffer
bp::Queue& graphicsQueue = device.getGraphicsQueue();
bp::CommandPool commandPool{graphicsQueue};
VkCommandBuffer commandBuffer =
    commandPool.allocateCommandBuffer();

//Record rendering commands
renderPass.render(framebuffer, commandBuffer);

//Execute rendering commands on GPU
graphicsQueue.submit({}, {commandBuffer}, {});
graphicsQueue.waitIdle();
```

Listing 3.6: Execute rendering to an offscreen framebuffer

After executing the code above, the offscreen framebuffer will hold the result from rendering. The `bp::OffscreenFramebuffer` object that represents the framebuffer, provides two textures, one color attachment and one depth attachment. We can implement a renderer that takes care of setting up color- and depth attachment slots, as well as the render pass, by inheriting `bp::Renderer`.

3.2.4 Scene Module

To represent the scene to be rendered, and provide classes to represent the GPU resources in use for rendering the scene, the module `bpScene` was implemented. The scene module provide a basic scene graph implementation represented by nodes. These nodes can be used to represent the position and rotation of objects and the camera.

The `bpScene` module provides the ability to load 3D models from files stored in the Wavefront obj format. This is achieved with the library tiny_obj_loader [31]. Some models in this format also specifies material files that contain information about colors and textures the 3D model uses. The class `bpScene::Mesh` allows us to load an obj file into a single 3D mesh, without considering the materials. The class `bpScene::Model` can load a 3D model from an obj file, taking into consideration how different parts of the model can use different materials. A model holds meshes, and materials, represented by `bpScene::Material`, as well as information about which material is applied to each mesh.

While the mesh, material and model classes represent the 3D geometry we want to render, they must be loaded onto a GPU before they can be rendered. To represent the GPU resources used for rendering the 3D geometry, resource classes have been implemented for meshes, materials and models. The class `bpScene::MeshResources` provide the GPU buffers used for vertex data and indexing. The class `bpScene::MaterialResources` hold the texture and color data for a material, as well as a descriptor set used to bind the color data to the graphics pipeline. The class `bpScene::ModelResources` represents mesh resources for all the partial meshes the model consists of, material resources for all the materials in use by the model, as well as a uniform buffer holding all the color values for the materials. The class `bpScene::PushConstantResource` allow us to bind matrices used for vertex calculation to the pipeline, based on a scene graph node.

In order to render the loaded 3D models, a subpass for rendering multiple objects referred to as drawables, has been implemented. This subpass, represented by `bpScene::DrawableSubpass`, will draw a list of drawables, represented by the abstract class `bpScene::Drawable`. Two classes deriving the drawable class have been implemented: `bpScene::MeshDrawable`, used for rendering a single mesh, and `bpScene::ModelDrawable`, for rendering a model with materials.

3.3 Multi-GPU Implementation

We have made available the source code [39] of an application supporting loading a 3D meshes from file, and rendering with either the sort-first or sort-last multi-GPU approaches. The source code builds on the abstraction library BP. Each GPU renders its contribution to a texture. The textures are composited to the final frame by one of the GPUs, before it is presented to the screen.

The current implementation of sort-first and sort-last multi-GPU rendering executes five steps in order to render a frame (see Figure 3.1):

1. Rendering the contributions for each GPU into textures.
2. Copy the contributing textures from the secondary GPUs to the host (CPU and main memory).
3. Redundant memory to memory copying step (discussed later) of the contributing textures.
4. Copying the textures from host to the primary GPU.
5. Combining the contributions into the final image in a compositing step.

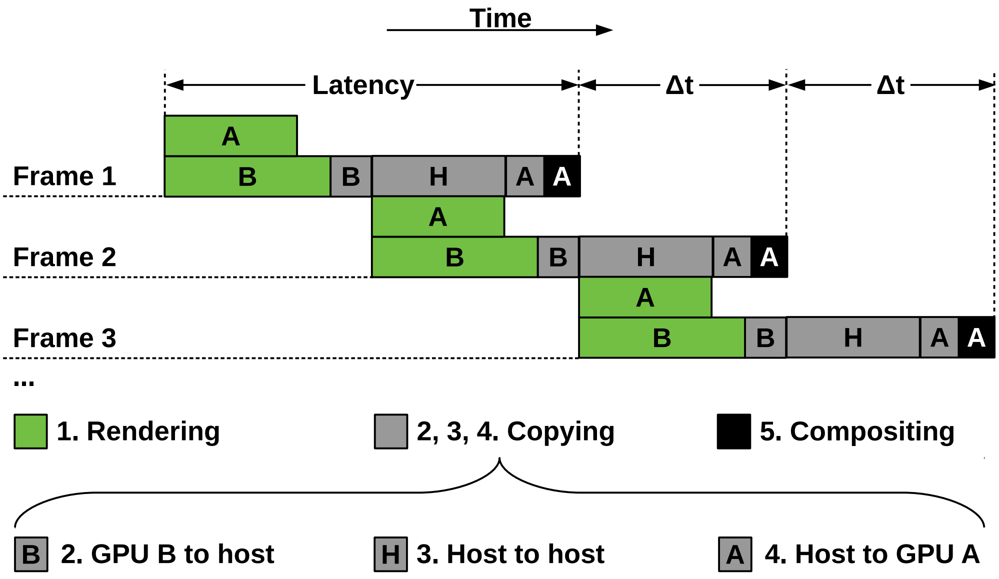


Figure 3.1: Timeline of the multi-GPU rendering steps for two GPUs A and B, and host CPU and main memory, H.

Figure 3.1 illustrates a timeline of the rendering process across a few frames. Tasks are distributed onto two GPUs, A and B, and host CPU and main memory H. GPU A is responsible of compositing the result into the final image. We refer to this GPU as the primary GPU. The other GPUs, which are only responsible for their partial rendering, are referred to as secondary GPUs.

If the 5 steps are executed sequentially every frame, the overhead from the three copying steps would be a bottleneck for performance. In order to increase the frame rate, functional parallelism (pipelining) is introduced, such that some stages can execute in parallel. This approach increases frame rate, however the latency from starting rendering a frame until presenting it on the screen is not improved.

It is possible to parallelize stages executing on a single GPU, by submitting commands to different queues. This would require the GPU to support multiple queues that can operate in parallel. When executing commands on multiple queues in parallel, the graphics driver software takes care of scheduling the tasks submitted to the queues. Many dedicated GPUs supports separate transfer queues, such that transferring data between the host and GPU can execute in parallel with rendering. This has been utilized to allow the primary GPU to perform rendering in parallel with the third copying step, when supported.

The GPUs render their contributions into textures. Each GPU needs two textures: the color attachment, and the depth attachment. In our code, the color attachment is implemented as a 32-bit RGBA texture and the depth attachment as a 16-bit depth texture. Both are represented by `bp::Texture` objects. These textures are provided by `bp::OffscreenFramebuffer` objects, which usage was described in section 3.2.3.

The application uses the `bpScene` module for rendering a list of 3D meshes or models loaded by the application. A `bpScene::DrawableSubpass` records the draw commands to execute in a render pass instance by each of the partial renderers. After the partial rendering has been executed, the result from rendering must be copied from the secondary GPUs into textures on the primary GPU. At this point, the compositing step can be executed on the primary GPU to produce the final image.

3.3.1 Multi-GPU Abstraction

To be able to implement an application utilizing the multi-GPU implementation quickly, the BP module `bpMulti` has been implemented. This module provides the ability to implement partial renderers for the sort-first and sort-last multi-GPU approaches. These renderers can then be passed to a compositor that takes care of the multi-GPU details of parallelization and synchronization, as well as combining the result into the final image.

Figure 3.2 illustrates the architecture of the `bpMulti` abstraction. The abstract class `bpMulti::Compositor` inherits from `bp::Renderer`, which allows us to specify a framebuffer to hold the result. The sort-first and sort-last compositor classes hold renderers that can be implemented by inheriting the sort-first and sort-last renderers respectively. The sort-first and sort-last renderers also inherits from `bp::Renderer`, but their intended use is for performing the per-GPU rendering into textures (partial rendering), that the compositor should combine into the final image.

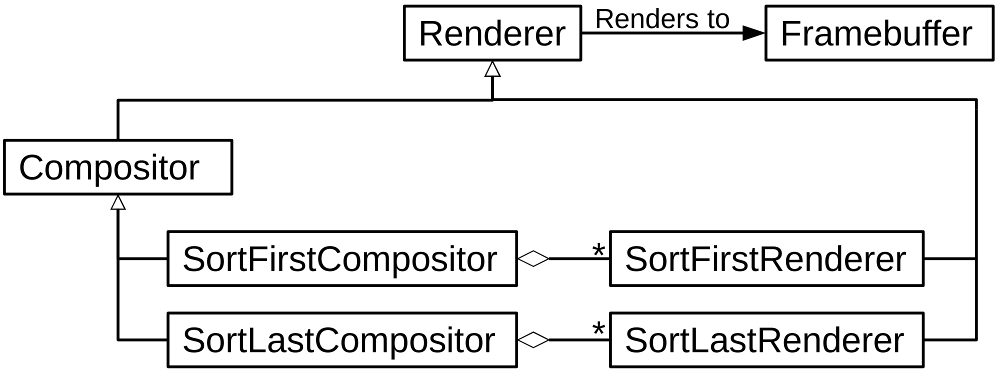


Figure 3.2: Multi-GPU abstraction architecture.

Each partial renderer is paired with a `bp::Device` object that represent the GPU that should perform rendering. Upon creation of the compositor, we have to specify the GPUs to use and pair them with a renderer. The first specified GPU is considered the primary GPU, while the remaining ones are considered secondary GPUs. The compositor will take care of partial rendering, copying contributing textures to the primary GPU, and compositing the result.

3.3.2 Sort-first Implementation

The sort-first approach is implemented by partitioning the screen into horizontal tiles of fixed height (see Figure 3.3). Each GPU renders its tile to the framebuffer consisting of the color- and depth attachment textures. Then the color textures rendered on the secondary GPUs are copied into the primary GPU before composition.

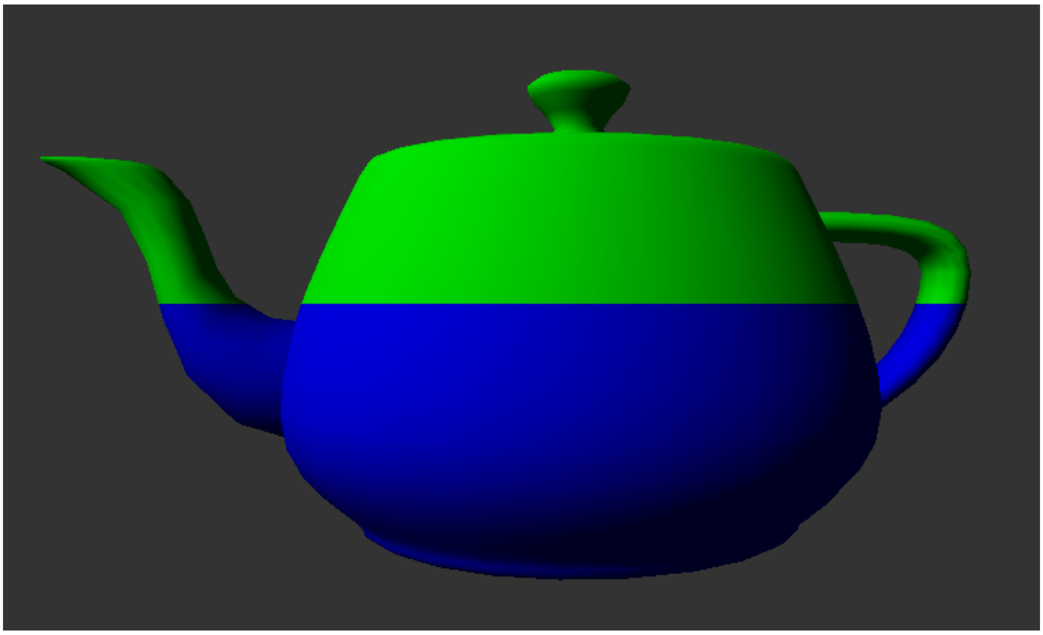


Figure 3.3: Screenshot illustrating the sort-first implementation. We render with two GPUs where the result from one GPU is shown in green and the other shown in blue.

In order to split the screen into tiles, each GPU must also transform the vertices such that the result is projected correctly, rather than having each GPU render the same result. This is done by multiplying the model-view-projection matrix with a matrix that translates and scales the clip space, such that the partition of the screen is stretched to fit the entire clip space. The following code calculates this clip space transform matrix:

```
float xTranslate = (2.f * x + w - 1.f) / w;
float yTranslate = (2.f * y + h - 1.f) / h;
mat4 translation = translate(mat4{},
                             {xTranslate, yTranslate, 0.f});
clipTransform = scale(translation, {1.f / w, 1.f / h, 1.f});
```

Listing 3.7: Calculate transformation matrix for partial sort-first rendering

The screen partition rectangle is represented by `x` and `y` offset coordinates, width `w` and height `h`. These values are normalized between 0 and 1.

After the secondary GPUs have performed the partial rendering, each texture contains the finished rendering for each tile. To combine the result, these textures must be copied into the primary GPU before compositing. The compositing step is implemented by drawing a rectangle for each of the partial render results, while sampling the color from the contributing texture in the fragment shader.

Using the abstraction implemented in `bpMulti`, we can implement a sort-first renderer by inheriting `bpMulti::SortFirstRenderer`:

```
class MySortFirstRenderer : public bpMulti::SortFirstRenderer
{
    void render(Framebuffer& fbo, VkCommandBuffer cmdBuffer)
    {
        glm::mat4 clipTransform =
            getContributionClipTransform();
        //Record rendering commands into cmdBuffer
    }
};
```

Listing 3.8: Implement a sort-first renderer

After implementing a sort-first renderer, we can create a renderer for each GPU and setup a compositor to combine the results. Given that we have two device objects available, we can setup sort-first rendering with two GPUs as follows:

```
MySortFirstRenderer renderer1, renderer2;
bpMulti::SortFirstCompositor compositor;
compositor.init({{&device1, &renderer1},
                 {&device2, &renderer2}},
                FORMAT, WIDTH, HEIGHT);
```

Listing 3.9: Initialize a sort-first compositor

To execute, we must create a suitable framebuffer and a command buffer, which we pass to the `render` method of the compositor. Then we can execute the commands for compositing by submitting the command buffer to a queue as explained in section 3.2.3.

A different way to composite the result for the sort-first approach is to create a borderless window for each GPU. These windows can then be resized and positioned next to each other to represent the final image. This approach will only work if all the GPUs supports presenting directly to a window. Then it is possible to skip the explicit copying steps, as it will be handled automatically by the operating systems window compositor. The BP abstraction library does not support this compositing approach currently, but a prototype has been implemented.

3.3.3 Sort-last Implementation

Sort-last parallelization is implemented by partitioning the geometry (see Figure 3.4). Each GPU renders its portion of the geomerty into textures of the same size as the render window. The contributing textures are copied from the secondary GPUs to the primary GPU. For sort-last, both the color- and depth textures must be copied, and the compositing must compare the fragment depths to decide from which texture to sample the color.

In order to composite the partial rendering results into the final image, we draw a full screen quad for each contribution. In the fragment shader we sample the color from the color texture, and set the fragment depth to the value sampled from the depth texture. Then the depth test takes care of selecting the closest fragment.

As with the sort-first implementation, we can use the abstraction implemented in `bpMulti` to implement renderers and handle compositing. By inheriting `bpMulti::SortLastRenderer`, we can create a sort-last renderer for partial rendering:

```
class MySortLastRenderer : public bpMulti::SortLastRenderer {
    void render(Framebuffer& fbo, VkCommandBuffer cmdBuffer)
    {
        ...
    }
};
```

Listing 3.10: Implement a sort-last renderer

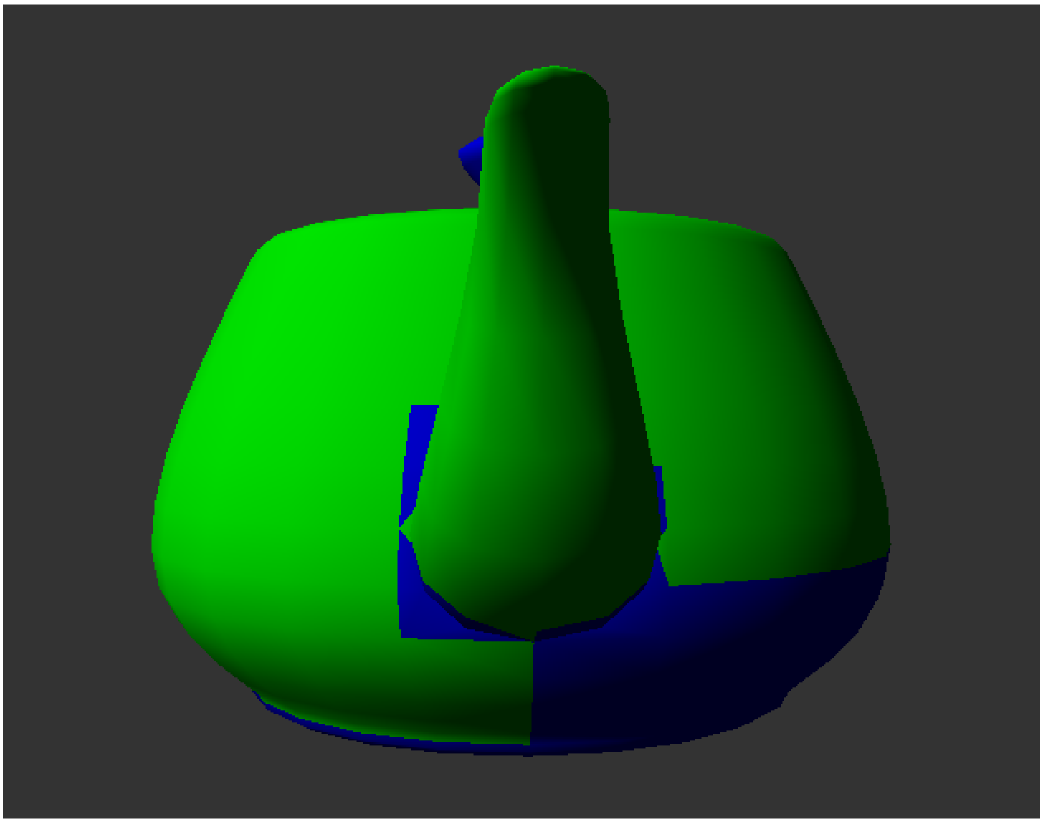


Figure 3.4: Screenshot illustrating the sort-last implementation. We render with two GPUs where the result from one GPU is shown in green and the other shown in blue.

We can set up a sort-last compositor for handling the multi-GPU details as follows:

```
MySortLastRenderer renderer1, renderer2;
bpMulti::SortLastCompositor compositor;
compositor.init({{&device1, &renderer1},
                 {&device2, &renderer2}},
                FORMAT, WIDTH, HEIGHT);
```

Listing 3.11: Initialize a sort-last compositor

Before we can execute sort-last rendering, an appropriate framebuffer must be provided, as well as a command buffer to record the composition commands into. In order to execute the commands, the command buffer must be submitted to a graphics capable queue, as described in section 3.2.3.

3.3.4 Copying Contributing Textures

The time it takes to copy a texture from one GPU to another increases with screen resolution, as more data must be transferred over the PCI-E connection. For the sort-last approach, the amount of data to copy between GPUs increases as more GPUs are used for rendering. The three steps implemented to copy the contributing textures from the secondary GPUs to the primary GPU can quickly become a bottleneck.

Naively Copying Through Staging Buffers

The current solution to copy data between GPUs is to have a host-allocated staging buffer for each of the GPUs, then map these buffers and copy between them. The process of copying data between GPUs is to first copy from GPU memory to a staging buffer, then copy between the two staging buffers, before finally copying from the second staging buffer to the destination GPU. Both of the staging buffers are located in main memory. Therefore, copying between them is in theory a redundant step.

In order to speed up the copying process for the redundant step, the memory is copied chunk-wise on multiple threads as a single thread is not able to utilize the entire memory bandwidth available. The tradeoff is high CPU usage.

Using External Memory Extensions

To avoid the redundant copying step would mean one less step to execute, increased frame rate and reduced CPU usage. To be able to do this, we would need a staging buffer shared between the source and destination GPUs. This could be possible using the Vulkan device extension `VK_EXT_external_memory_host` [17, p. 1247]. However, the extension requires graphics driver support which is currently limited. AMD and NVIDIA have contributed in developing the extension [40]. Therefore it may be supported in future driver releases.

3.3.5 Prototype Application

The implemented prototype application [39], allows us to load 3D models from file to render with a single GPU or multiple GPUs using either the sort-first or the sort-last approaches. It has been used to perform the sort-last benchmarks, and a modified version was used to perform the sort-first benchmarks. The application utilizes BP for implementing rendering and multi-GPU compositing. The `bpScene` module is used to represent and render the scene, and the `bpMulti` module is used to implement the multi-GPU details of rendering with multiple threads and combining the results.

Application Usage

The prototype application provides a simple command line interface to select 3D models to load, multi-GPU strategy, GPU count, initial screen resolution, and flags for changing how models should be loaded and rendered. This interface has been implemented with the "program_options" boost library [32]. By executing the application with the `--help` flag, the application will print a summary of available options and flags.

In order to specify to the application which 3D models to load, we use the `--path` option. By default this loads a single Wavefront obj model, however it is possible to load multiple models in to ways. The `--list` flag specifies that the path is a file that lists paths to the obj files that should be loaded. The format of this file is file paths separated by newlines. The second way to load files is to specify a folder to load obj files from recursively, with the `--directory` flag. The recursive file loading has been implemented with the boost library "filesystem". In order to limit the amount of loaded files, it is possible to use the `--max` flag set a maximum number of files to load.

The default approach to rendering is to use a single GPU. In order to use one of the multi-GPU approaches, we must specify the `--strategy` option, which supports specifying a single GPU (default), sort-first or sort-last. The default number of GPUs used for rendering with the sort-first and sort-last approaches is two. To specify GPU count we can use the `--count` option.

Though the window used for presenting the result is resizable, being able to specify an exact window resolution is necessary to achieve consistent benchmark results. The default resolution is 1024 × 768, but a different resolution can be specified with the `--resolution` option. During development, sometimes a multi-GPU setup is not available. To be able to test the multi-GPU implementation on a single-GPU system, the flag `--simulate-mgpu` will tell the application to simulate multiple GPUs by creating multiple logical devices for a single GPU.

The default behavior of the application is to load 3D models with materials. In order to support loading models without materials, the prototype application accepts a command line flag, `--basic`, to specify to the application not to load materials. To be able to render meshes that does not provide vertex normals, we can specify the `--generate-normals` flag, which uses a geometry shader to generate normals which can be used for shading. The application interprets the Y-axis as pointing upwards. However some 3D models interpret the Z-axis as pointing upwards instead. To support such models, the `--z-up` flag specifies to the application to rotate the model such that the correct axis points up in the scene.

Chapter 4

Results

In order to evaulate the implementation, performance has been measured for benchmarks tailored specific to the sort-first and sort-last approaches. Two different hardware configurations have been tested, both utilizing only dedicated GPUs. The first testing computer, referred to as TC1, has three relatively powerful GPUs (see Table 4.1). The second testing computer, referred to as TC2, has two weaker GPUs from different vendors (see Table 4.2), which shows that the implementation is vendor agnostic and can work with any Vulkan capable GPUs.

CPU	Intel Core i7-6850K
RAM	64GB
GPU 1	Nvidia GeForce GTX 1080 8GB
GPU 2	Nvidia GeForce GTX 1080 8GB
GPU 3	Nvidia GeForce GTX 1080 8GB

Table 4.1: Specification of testing computer 1 (TC1)

CPU	Intel Core i7-7700K
RAM	32GB
GPU 1	Nvidia GeForce GTX 760 2GB
GPU 2	AMD Radeon RX 460 2GB

Table 4.2: Specification of testing computer 2 (TC2)

An ideal multi-GPU implementation would be able to utilize all GPUs 100% for rendering. However, due to the overhead from copying contributing textures from secondary GPUs to the primary GPU, as well as the compositing step, it is to be expected that the utilization decreases when executing one of the implemented multi-GPU approaches. The GPU utilization for the performance results is calculated from the measured framerates of the multi-GPU results, compared to what an ideal framerate would be based upon the results from rendering with each GPU individually. We assume that the GPUs are fully utilized when rendering individually without the multi-GPU overhead.

For both the sort-first and sort-last approaches, each GPU renders to off-screen framebuffers. The pixels from these framebuffers are combined to produce the final image. The difference between them is that the sort-first approach distributes the pixels to the GPUs, while the sort-last approach distributes geometry. So a benchmark that scales well the sort-first approach might not scale well for the sort-last approach. This is why two different benchmarks have been implemented. Both benchmarks were executed with a screen resolution of 1024 × 768.

4.1 Sort-first Benchmark

A rendering workload that would scale well for the sort-first algorithm is one that has high computational load per fragment. Examples of such workloads are complex triangle shading, or ray tracing, where the computation load is high per pixel. To simulate this situation, the sort-first benchmark draws a rectangle filling the framebuffer for each GPU, and performs demanding calculations in the fragment shader. This approach implements a high per-pixel computational workload, which allow us to evaluate the performance of the implemented multi-GPU compositing. The following fragment shader code was used in a modified version of the prototype application [41], to benchmark the sort-first implementation:

```
#version 450 core
layout (location = 0) in vec2 uv;
layout (location = 0) out vec3 color;
void main() {
    vec2 uv2 = uv;
    for (int i = 0; i < 10000; i++) {
        uv2 *= 1.00001;
        uv2 *= 0.999998;
    }
    color = vec3(uv2, 0.0);
}
```

Listing 4.1: Sort-first benchmark fragment shader code

The input variable uv is passed from the vertex shader and behaves like a texture coordinate to apply a texture to the entire screen. In this case it is used to produce a gradient. As this program executes once per fragment, the loop will provide a high computational load. The calculations performed in the loop needed to change the result somewhat, otherwise, the loop would automatically be removed by the shader compiler as a performance optimization. When running the benchmark, we changed the number of iterations to increase or decrease the rendering workload.

Figure 4.1 shows the results for the sort-first implementation on TC1. Two GPUs yielded a 58% increase of frame rate compared to a single GPU. Adding a third GPU further increased the frame rate 22%, or an increase of 94% compared to a single GPU. Compared to an ideal implementation, the GPU utilization is 79% and 65% for two and three GPUs respectively.

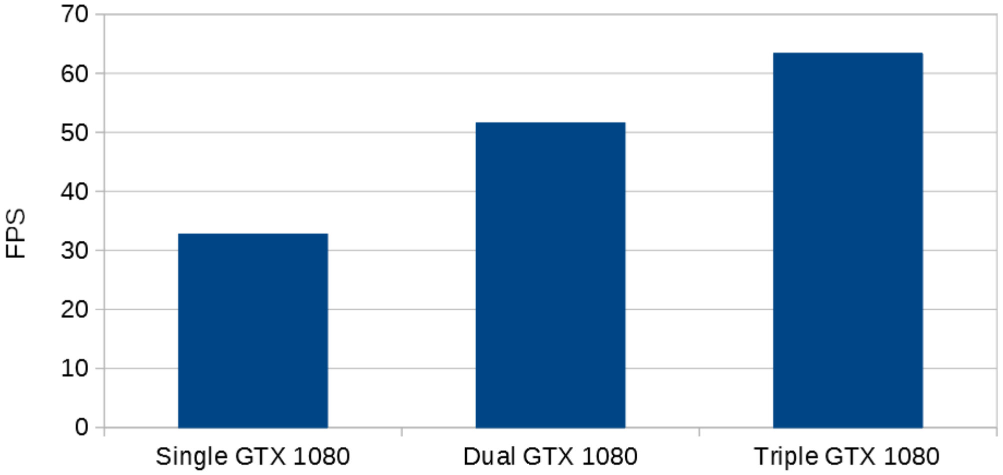


Figure 4.1: Graph of the sort-first performance for TC1.

On TC2, the sort-first results did not improve performance, as the difference in performance of the GTX 760 and the RX 460 was too large to get performance improvement in this benchmark. This is perhaps due to the difference in Raster Operations Pipeline (ROP) count, of which the GTX 760 has 32, and the RX 460 has 16. However for the demanding geometry implemented in the sort-last benchmark, the two GPUs performed similarly, as the GPUs have similar compute performance.

The prototype implementing the compositing method using borderless windows was not able to run on TC1. We suspect that this is because the computer was accessed through external desktop. The configuration was only able to present its result to a window from one of the GPUs, but the approach using borderless windows require all GPUs to be able to do so. We were able to run the borderless window prototype on TC2, but we still had issues with the performance difference between the two GPUs. However, we can evaluate the borderless window compositing method by comparing it against the implementation that composites the result on a single window.

By rendering a small mesh, the compositing method became the bottleneck for framerate, and we can compare the maximum framerate for the two methods at different screen resolutions. Figure 4.2 shows the measured framerates for the two compositing approaches on TC2. When running this we noticed that the borderless window approach achieved higher framerates at first, but settled at a lower framerate when the application had run for a few seconds. At the lowest resolution of 1024×768 the framerate started above 300FPS, but after a few seconds it got reduced to 207FPS. However, this was not the case when run at a screen resolution of 3840×1080. In this case, the framerate was measured to be faster than when run at a screen resolution of 1920×1080. We don't know the reason for this behavior. It could be caused by throttling of resources, for instance a framerate limit. However, for the implementation rendering into a single window, the framerate is reduced when screen resolution increases.

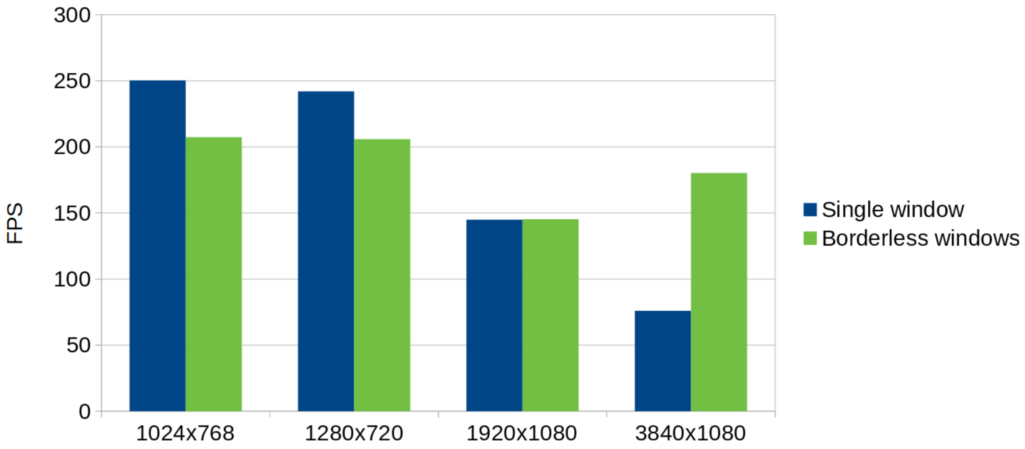


Figure 4.2: Graph of the performance of different sort-first compositing methods for TC2.

Mesh	**Triangle count**
Statue of Lucy	28 055 742
Boeing Airplane	282 871 419

Table 4.3: Table of the meshes used to test the sort-last performance.

4.2 Sort-last Benchmark

A rendering workload that would scale well for the sort-last algorithm is one that has a large amount of geometry. The benchmark used to evaluate the sort-last implementation draws large meshes, such that a single GPU is not powerful enough to achieve good performance. Then the geometry can be distributed between the GPUs to improve performance.

Two meshes have been used for executing the performance benchmarks (see Table 4.3). One of these meshes is a reconstructed mesh of a statue of Lucy (see Figure 4.3), with more than 28 million triangles, provided by the Stanford 3D Scanning Repository [42]. The second mesh is a large mesh of a Boeing airplane (see Figure 4.4), with more than 280 million triangles. This mesh was used with permission by the Boeing company. The Boeing airplane mesh does not contain all geometry for the entire airplane, but it provided a large enough geometry to be a demanding benchmark for the sort-last implementation.

Figure 4.3: Screnshot from rendering the statue of Lucy.

Since TC1 has significantly more powerful GPUs than TC2, the benchmark was run with different amount of geometry for each of the computers. For TC1, the Lucy mesh was rendered 8 times at different positions, for a total of more than 224 million triangles. This allowed us to demonstrate a scenario where the performance could increase with three GPUs, as the amount of geometry was too large to achieve good performance with a single GPU. TC2 rendered the mesh twice, for about 56 million triangles. TC1 was also tested with the the Boeing airplane mesh. Once loaded into a single GPU, the Boeing mesh has a memory usage of about 6GiB, so TC2 does not have enough GPU memory to render this mesh.

The sort-last benchmark results shows similar increases in performance as the sort-first benchmark for TC1 (see Figure 4.5). We got a 69% improvement

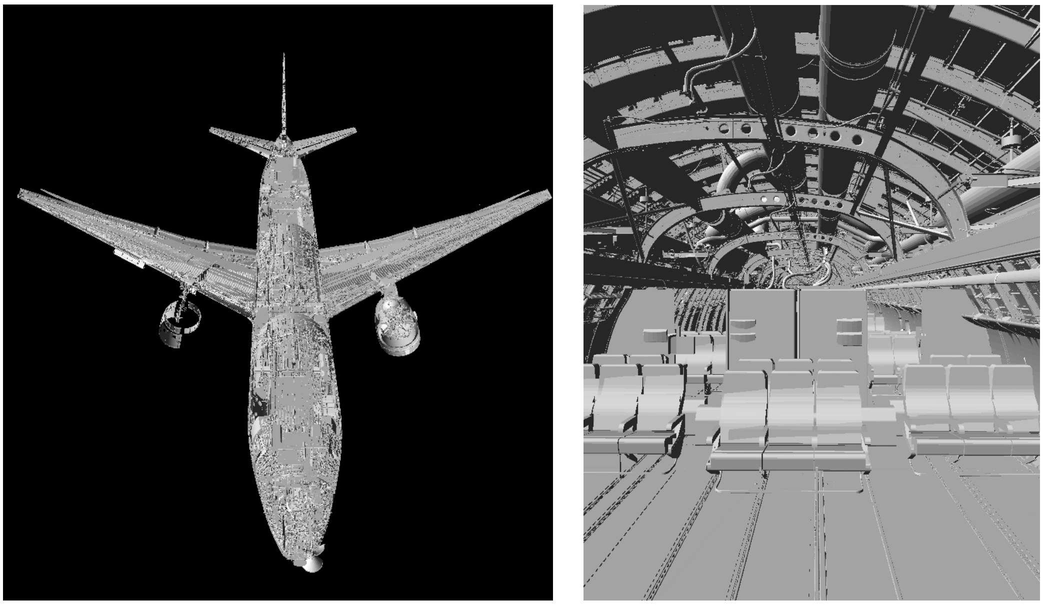


Figure 4.4: Screenshots from rendering the Boeing airplane.

from one to two GPUs, and a 16% improvement from two to three GPUs, which is a 97% improvement from one to three GPUs. The GPU utilization compared to an ideal multi-GPU implementation is in this case 85% and 66% for two and three GPUs respectively.

The results from the sort-last benchmarks on TC2 (see Figure 4.6) yielded a performance increase of 52% when utilizing both GPUs, compared to the GTX 760 GPU alone, or 71% compared to the RX 460 GPU on its own. The GPU utilization is 80% compared to an ideal implementation.

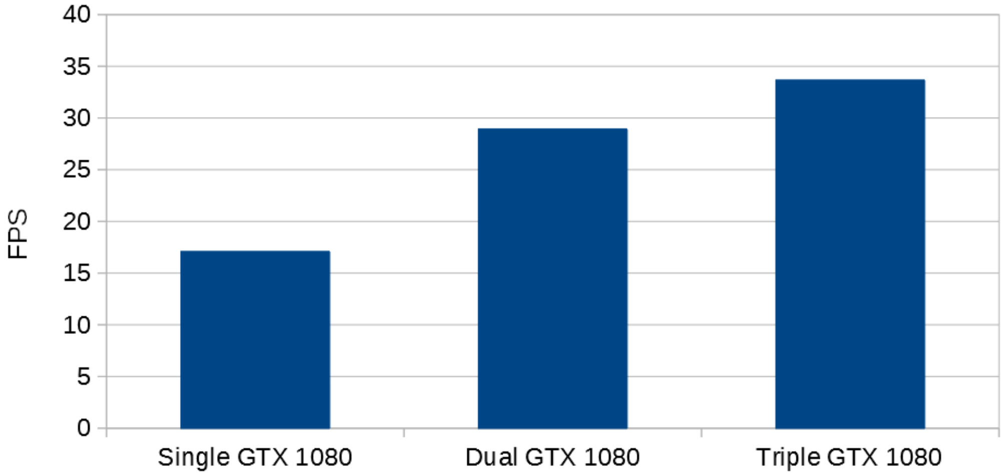


Figure 4.5: Graph of the sort-last performance for TC1, rendering the Lucy statue.

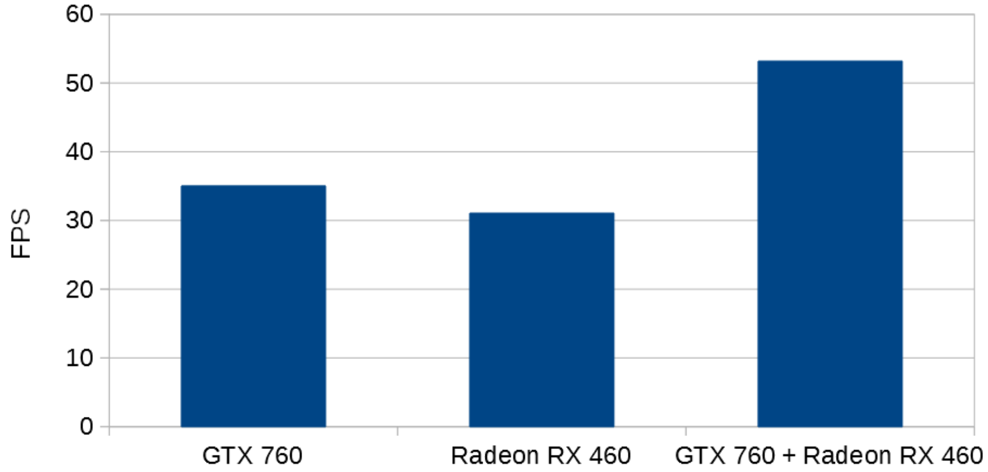


Figure 4.6: Graph of the sort-last performance for TC2, rendering the Lucy statue.

When tested with the larger Boeing airplane mesh, we achieved better scaling than with the Lucy statue mesh (see Figure 4.7). This is perhaps because the amount of geometry rendered was larger, such that the overhead from compositing was smaller compared to the rendering workload. The sort-last implementation achieved a 76% increase in framerate from one to two GPUs, 27% increase from two to three, which is an improvement of 124% from a single GPU to three GPUs. The GPU utilization was 88% and 75% for two and three GPUs respectively, compared to an ideal implementation.

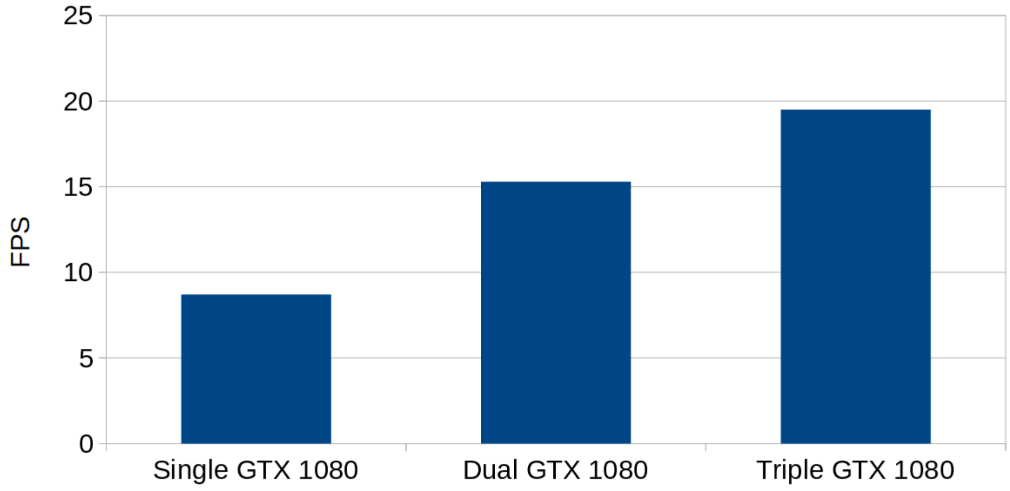


Figure 4.7: Graph of the sort-last performance for TC1, rendering the Boeing airplane.

4.3 Rendering Geometry with High-resolution Textures

One way to utilize multiple GPUs is to speed up the rendering process, increasing framerate. Another use case for multiple GPUs is to utilize the available GPU memory to render geometry with high-resolution textures, which would not fit in the memory on a single GPU. This can be done with the sort-last approach, by distributing the geometry and textures to multiple GPUs.

The data set we used in order to perform rendering of geometry with high-resolution textures, consists of 3D models of the Beckwith Plateau (Book Cliffs, UT, USA) mountain side. The Virtual Outcrop Geology group, Uni

Research, Bergen is acknowledged for access to the 3D models used within this work. The data was acquired with the support of the Research Council of Norway and FORCE Sed/Strat group through the EUSA/SAFARI project (grant number 193059). This model is part of a data set acquired with light detection and ranging (LIDAR) scanning from a helicopter, as described by Eide et al., 2014 [43].

The model of Beckwith Plateau consist of 12 sections of the mountain side, two of which are split into two sections, and high resolution textures (see Table 4.4). With TC1, we were able to render 9 of these sections (see Figure 4.8 and 4.9) with three GPUs, at a framerate of 50FPS. If we had access to one, or two more GPUs, we could render all 12 sections. With a single GPU we were only able to render 3 sections.

Section	**Triangle count**	**Textures (compressed JPG files)**
1	2 005 007	300MB
2	1 918 131	308MB
3	1 838 960	304MB
4	1 798 272	290MB
5	1 520 325	233MB
6	1 716 656	288MB
7a	1 404 145	253MB
7b	1 819 675	335MB
8a	1 241 730	242MB
8b	1 057 564	192MB
9	2 152 939	422MB
10	2 074 731	324MB
11	1 952 748	380MB
12	1 871 221	430MB
Total	**24 372 104**	**4.3GB**

Table 4.4: Triangle counts and amount of textures for the Beckwith Plateau model sections. The textures are compressed JPG files, that takes more memory when uncompressed and loaded on a GPU.

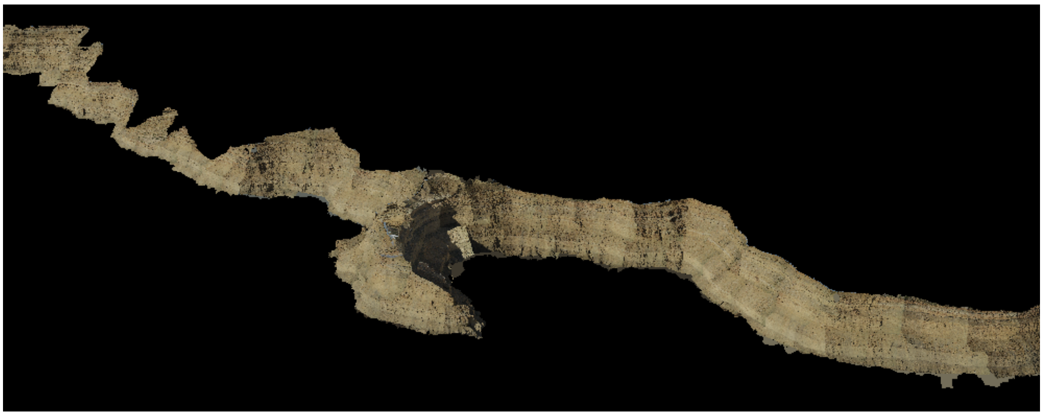


Figure 4.8: Screenshot from rendering 9 sections of the Beckwith Plateau, Book Cliffs, UT, USA.

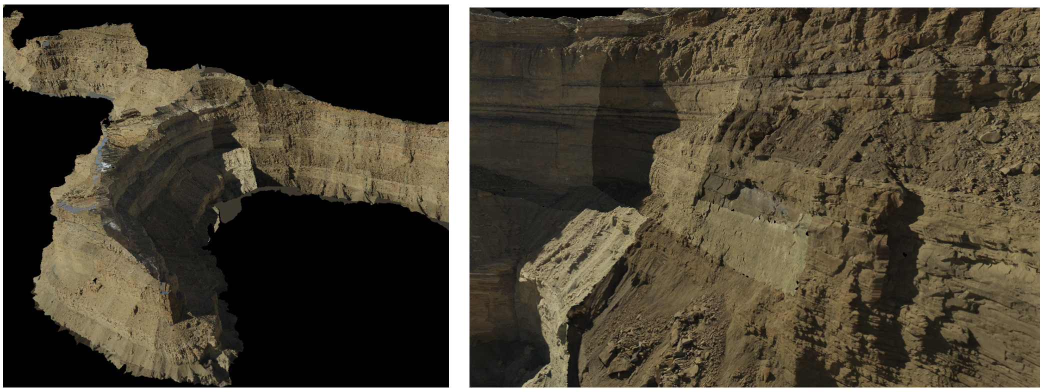


Figure 4.9: Closer screenshots from rendering the Beckwith Plateau, Book Cliffs, UT, USA.

Chapter 5

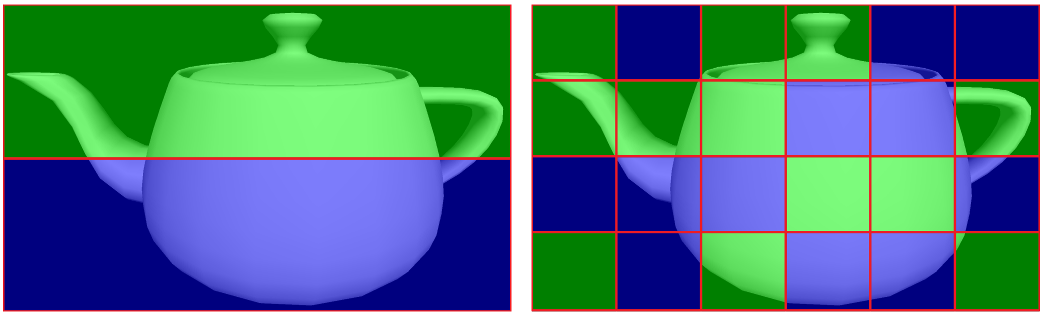
Discussion

The measured framerates for the sort-first and sort-last implementations show that performance can be improved when adding a second and third GPU. However, GPU utilization is reduced when adding more GPUs, due to overhead from copying contributing textures and compositing the result. Improvements are also limited when the resolution is increased. Though it is worth noting that for the sort-first approach, the amount of data that needs to be copied between GPUs stays the same when GPU count increases, as the screen is divided into smaller partitions when more GPUs are used for rendering. For the sort-last approach, the amount of data that needs to be copied increases with GPU count, as the contributing textures from the secondary GPUs are the same size as the destination framebuffer.

5.1 Sort-first Screen Partitioning

The sort-first implementation divides the screen in horizontal tiles of equal size (see Figure 5.1a). Each GPU must render its own tile. However, the workload of rendering each tile might differ as the scene changes. Some parts of the screen can potentially be much more demanding to render than others, such that the workload distribution becomes unbalanced. The sort-first benchmark does not simulate this behavior, as the workload for this benchmark is uniform for the entire screen.

One way to mitigate unbalanced workload distribution for the sort-first approach, could be to partition the screen into smaller tiles which can be distributed to the GPUs randomly (see Figure 5.1b). Instead of rendering only a single tile, each GPU could render multiple smaller tiles. With this approach, load balancing could be introduced by changing the number of tiles each GPU renders, based upon measurements of the time it takes to render for each GPU. It would also be wise to take into consideration the amount of time spent copying the data between the secondary and primary GPUs when implementing this load balancing. It is possible to implement load balancing for the current screen partitioning strategy, by resizing the tiles. However, using randomly distributed, smaller tiles, we can distribute the workload more evenly, for instance if the top half of the screen contain all the geometry.



(a) The current screen partitioning strategy. Each GPU renders one horizontal tile.

(b) Possible screen partitioning strategy where each GPU renders multiple smaller tiles.

Figure 5.1: Illustration of different ways to partition the screen for sort-first compositing with two GPUs. One GPU renders the green tiles, while the other renders the blue tiles.

5.2 Sort-last Alpha Blending

The current implementation of the sort-last approach does not support correct compositing of transparent geometry with alpha blending. Alpha blending uses an additional pixel component known as the alpha component. This component represents the opacity of the pixel, which allows us to blend colors of fragments that fall within the same pixel.

Alpha blending could be performed on the geometry distributed to each of the GPUs, however the necessary depth information required to merge the alpha blended geometry is not available. The depth component of the sort-last contribution only contains depth information about the closest rendered geometry. For transparent geometry to be rendered correctly, all opaque geometry must be rendered first, but each GPU renders only part of the geometry.

One way to solve this issue could be to render all transparent geometry on the primary GPU responsible for compositing. All opaque geometry must be rendered and composited before drawing the transparent geometry. This approach will become inefficient if the scene mostly consists of transparent geometry, for instance if a configuration of two identical GPUs should render a scene where more than half of the geometry is transparent. In this case, the secondary GPU would not be fully utilized, as the primary GPU has to render most of the geometry.

5.3 Post-processing

Utilization of post-processing effects like motion blur, bloom, depth of field, and so forth, has not been implemented for the multi-GPU solution introduced with this thesis. Post-processing requires all the geometry to be rendered, so post-processing must happen after compositing the final image on the primary GPU. If the rendering workload is evenly distributed among the GPUs, the additional overhead from post-processing on the primary GPU can cause idle time on the secondary GPUs. This issue can be mitigated by implementing load balancing.

Having each GPU execute post-processing for its own screen region could be possible for the sort-first approach. However this approach can produce sligthly different results on the borders of the screen partitions, causing the partitioning to be visible.

5.4 Copying between GPUs

The implemented compositing requires contributing textures from secondary GPUs to be copied to the primary GPU responsible for combining the result. The optimal way to do this would be to copy the textures directly from one GPU to the other. However this is not possible for an unlinked explicit configuration. The contributions must be copied into main memory, before copying it back to the second GPU.

Due to a limitation with core Vulkan API, where the memory objects are bound to specific device objects, an additional redundant CPU to CPU copying step has been added. The process of copying a texture from one GPU to another is implemented with three steps. First, we copy the texture from the source GPU to a host staging buffer. Then we copy the texture from this host staging buffer to another staging buffer bound to the destination GPU. The last step is to copy the texture from the staging buffer to the destination GPU.

The redundant copying step is accelerated by copying blocks of memory in parallel. This speeds up the copying process, as a single thread is not capable of utilizing the entire available memory bandwidth. When the amount of data to copy increases, the amount of CPU time spent on copying increases as well. So if screen resolution increases, or more GPUs are used, CPU utilization can increase significantly. This limits the available CPU time for other tasks like physics calculations. It also limits the performance as screen resolution increases, which is the reason the benchmarks were run at a relatively low resolution (1024 × 768).

Eliminating this redundant copying step will improve performance, efficiency, as well as latency. It is possible to implement shared host-allocated memory between GPUs with the Vulkan extension `VK_EXT_external_memory_host` [17, p. 1247]. This shared staging buffer would enable us to eliminate the redundant copying step.

5.5 Attempt at Integrating into Existing Framework

During my study trip in Vienna, I attempted to integrate the multi-GPU solution into an existing rendering framework. This framework, used by the vis-group at TU Wien, was implemented with OpenGL. In order to integrate my solution, I would need to port the framework to support Vulkan API, as well as implement the ability to divide the rendering workload for distributing onto multiple GPUs.

The framework relies on being able to render to render to different framebuffers. At the time, my abstraction library, BP, automatically created framebuffers as part of the render pass abstraction. Instead of assigning attachments to framebuffers, the attachments were assigned directly to the render pass, which made the library inflexible for integrating into the framework at the time. Due to this limitation, I was not able to integrate the multi-GPU solution into the existing framework. However, with a more recent implementation of BP, it could be possible to implement, as the library was refactored to support framebuffers when the `bpMulti` module was implemented.

Chapter 6

Conclusion

With this thesis we have introduced a solution to multi-GPU rendering with Vulkan API. This is the first publicly available solution implementing heterogeneous multi-GPU rendering, using explicit selection of the Vulkan capable GPUs in a single computer. Both the Linux and Windows operating systems are supported, and the solution can use GPUs of different models, from different vendors. We have shown that performance can be improved compared to a single GPU, using the sort-first and sort-last approaches to multi-GPU rendering. Using the sort-last approach, we have also shown that we can utilize the additional GPU memory from multiple GPUs, to render larger data sets than possible with a single GPU.

With the first research question, we wanted to know how we can implement efficient multi-GPU rendering with Vulkan API. This thesis describes how the introduced multi-GPU solution has been implemented. In terms of efficiency, there is still some room for improvement, due to the redundant step for copying data between GPUs. However, the performance tests have shown that it is possible to achieve higher framerates using the introduced multi-GPU solution, compared to using a single GPU.

With the second research question, we wanted to know how we can divide and distribute rendering workload among multiple GPUs in different ways, and what performance improvement can be expected for the different approaches compared to a single GPU. The two approaches to dividing and distributing rendering workload implemented for the solution, are sort-first and sort-last. The solution shows increase in performance compared to a single GPU, with benchmarks tailored to utilize the specific approaches. The performance will only increase when the rendering workload is sufficiently demanding for a single GPU, and that the multi-GPU approach reduces the amount of work for each GPU. For small workloads, a single GPU is faster than multiple GPUs because of the additional overhead from copying data between GPUs, and compositing of the result.

The sort-first approach improved framerates with a workload demanding per pixel. This could be useful for accelerating a ray tracer. The sort-last approach improved framerates when rendering large amounts of geometry. The results from rendering the Boeing airplane achieved good scaling, with GPU utilization of 88% and 75% for two and three GPUs respectively.

Another way to utilize multiple GPUs other than for increased graphics processing power, is to get access to a larger amount of GPU memory. The results from rendering the Beckwith Plateau mountain side, shows that with the sort-last implementation, it is possible to utilize the extra GPU memory to render larger data sets than possible with a single GPU.

Chapter 7

Further Work

There is room for improvement for the introduced multi-GPU implementation, both in terms of performance, and in terms of support for different rendering techniques. Performance can be improved by implementing optimizations for increasing the GPU utilization.

To reduce latency, CPU usage, and improve performance, one could utilize the `VK_EXT_external_memory_host` extension [17, p. 1247] to investigate CPU allocated memory shared between GPUs, or host-mapped memory for a secondary GPU, imported for the primary GPU to read. The last approach only require the primary GPU to support the extension. Another extension, `VK_EXT_external_memory_dma_buf` [17, p. 1245] might also be suitable for linux systems, but would require all GPUs to support the extension. If implemented successfully, shared host memory between GPUs would allow us to eliminate the redundant copying step for copying data between GPUs.

In order to optimize the performance, dynamic load balancing could be implemented. This load balancing would redistribute rendering workload for the GPUs based on the amount of time spent rendering, copying and compositing per GPU. The purpose of balancing the workload is to reduce the amount of time to render a frame, by relieving rendering workload from GPUs that becomes bottlenecks, and redistributing that workload to GPUs that have some headroom. This decreases GPU idle time and increases GPU utilization.

For the sort-first approach, one could improve distribution of uneven rendering workloads, by subdividing the screen into smaller tiles. Each GPUs would then be responsible for rendering multiple such tiles. The tiles can be distributed dynamically with load balancing to achieve optimal performance.

Post-processing effects could be investigated for the sort-first approach, to see if the result becomes a uniform image, or if the screen partitioning becomes visible.

For the sort-last approach, one could investigate using the primary GPU for different rendering tasks than the secondary GPUs. The primary GPU could for instance render transparent geometry, after all the opaque geometry have been rendered by the secondary GPUs. Post-processing effects could also be implemented on the primary GPU. This process would be pipelined, such that the primary GPU performs rendering of transparent geometry and post-processing for the previous frame, while the secondary GPUs renders the opaque geometry for the current frame.

Bibliography

[1] Khronos Group. *OpenGL Overview.* https://www.opengl.org/about/. Accessed: 2017-11-27.

[2] Khronos Group. *Vulkan.* https://www.khronos.org/vulkan/. Accessed: 2017-11-13.

[3] NVIDIA Corporation. *Introduction to SLI Technology.* https://www.geforce.com/whats-new/guides/introduction-to-sli-technology-guide. Accessed: 2017-11-15.

[4] Advanced Micro Devices Inc. *AMD Crossfire Technology.* https://www.amd.com/en/technologies/crossfire. Accessed: 2017-11-15.

[5] N. Haemel. *AMD_gpu_association.* https://www.khronos.org/registry/OpenGL/extensions/AMD/WGL_AMD_gpu_association.txt. Accessed: 2017-11-24. Mar. 2009.

[6] B. Lichtenbelt. *WGL_NV_gpu_affinity.* https://www.khronos.org/registry/OpenGL/extensions/NV/WGL_NV_gpu_affinity.txt. Accessed: 2017-11-24. Nov. 2006.

[7] Khronos Group. *OpenGL Context.* https://www.khronos.org/opengl/wiki/OpenGL_Context. Accessed:2018-05-20.

[8] Advanced Micro Devices Inc. *AMD - GPU Association - Targeting GPUs for Load Balancing in OpenGL.* 2010.

[9] Derivative.ca. *Using Multiple Graphic Cards.* https://www.derivative.ca/wiki088/index.php?title=Using_Multiple_Graphic_Cards#Quadro_Cards_and_GPU_Affinity. Accessed: 2017-11-24.

[10] S. Molnar et al. “A sorting classification of parallel rendering”. In: *IEEE computer graphics and applications* 14.4 (1994), pp. 23–32.

[11] NVIDIA Corporation. *NVLink Fabric.* https://www.nvidia.com/en-us/data-center/nvlink/. Accessed: 2018-04-30x.

[12] R. Hallock. *Modernizing multi-GPU gaming with XDMA*. https://community.amd.com/community/gaming/blog/2015/05/11/modernizing-multi-gpu-gaming-with-xdma. Accessed: 2018-04-30.

[13] Y. Kim et al. "GPUpd: A Fast and Scalable multi-GPU Architecture Using Cooperative Projection and Distribution". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 574–586. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123968. URL: http://doi.acm.org/10.1145/3123939.3123968.

[14] S. Marchesin, C. Mongenet, and J.-M. Dischler. "Multi-GPU sort-last volume visualization." In: *EGPGV*. 2008, pp. 1–8.

[15] R. Samanta et al. "Hybrid Sort-first and Sort-last Parallel Rendering with a Cluster of PCs". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '00. Interlaken, Switzerland: ACM, 2000, pp. 97–108. ISBN: 1-58113-257-3. DOI: 10.1145/346876.348237. URL: http://doi.acm.org/10.1145/346876.348237.

[16] P. Wang et al. "Multi-GPU Compositeless parallel rendering algorithm". In: *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on*. IEEE. 2011, pp. 103–107.

[17] Khronos Vulkan Working Group. *Vulkan 1.0.66 - A Specification (with all registered Vulkan extensions)*. 1.0.66. Nov. 2017.

[18] Khronos Group. *Tesselation*. https://www.khronos.org/opengl/wiki/Tessellation. Accessed: 2018-01-15.

[19] Intel Corporation. *Intel Core i7-8700K Processor*. https://ark.intel.com/products/126684/Intel-Core-i7-8700K-Processor-12M-Cache-up-to-4_70-GHz. Accessed:2018-05-15.

[20] Advanced Micro Devices Inc. *AMD Ryzen^{TM} Threadripper^{TM} Processors*. https://www.amd.com/en/products/ryzen-threadripper. Accessed:2018-05-15.

[21] Microsoft Corporation. *Multi-Adapter*. https://msdn.microsoft.com/en-us/library/windows/desktop/dn933253(v=vs.85).aspx. Accessed: 2017-11-27.

[22] P. Bright. *Vulkan 1.1 out today with multi-GPU support, better DirectX compatibility*. https://arstechnica.com/gadgets/2018/03/vulkan-1-1-adds-multi-gpu-directx-compatibility-as-khronos-looks-to-the-future. Accessed:2018-05-08.

[23] Khronos Group. *Transform Feedback.* https://www.khronos.org/opengl/wiki/Transform_Feedback. Accessed:2018-05-25.

[24] Khronos Group. *SPIR Overview.* https://www.khronos.org/spir/. Accessed: 2018-03-03.

[25] Qt. *About Qt.* https://wiki.qt.io/About_Qt. Accessed:2018-03-06.

[26] GLFW. *GLFW.* http://www.glfw.org/. Accessed:2018-03-06.

[27] G-Truc Creation. *OpenGL Mathematics.* https://glm.g-truc.net/0.9.8/index.html. Accessed:2018-05-21.

[28] Advanced Micro Devices Inc. *Vulkan Memory Allocator.* https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator. Accessed: 2018-02-26.

[29] LunarG. *LunarG Vulkan SDK.* https://www.lunarg.com/vulkan-sdk. Accessed:2018-05-21.

[30] S. Barrett. *stb.* https://github.com/nothings/stb. Accessed:2018-05-21.

[31] S. Fujita. *tinyobjloader.* https://github.com/syoyo/tinyobjloader. Accessed:2018-05-21.

[32] D. A. Beman Dawes and R. Rivera. *Welcome to Boost.org!* https://www.boost.org. Accessed:2018-05-28.

[33] Kitware. *Build, Test and Package Your Software With CMake.* https://cmake.org. Accessed:2018-05-21.

[34] Khronos Group. *OpenGL / OpenGL ES Reference Compiler.* https://www.khronos.org/opengles/sdk/tools/Reference-Compiler. Accessed:2018-05-21.

[35] L. O. Tolo. *C++ Abstraction library for Vulkan API.* https://github.com/larso0/bp. Accessed: 2018-03-03.

[36] SDL. *About SDL.* https://www.libsdl.org/. Accessed:2018-03-06.

[37] cppreference.com. *Parameter pack.* http://en.cppreference.com/w/cpp/language/parameter_pack. Accessed:2018-05-30.

[38] Qt. *Signals & Slots.* http://doc.qt.io/qt-5/signalsandslots.html. Accessed:2018-05-30.

[39] L. O. Tolo. *Vulkan Multi-GPU Application.* https://github.com/larso0/vmgpu. Accessed: 2018-03-03.

[40] M. Larabel. *Vulkan 1.0.66 Introduces Three New Extensions.* https://www.phoronix.com/scan.php?page=news_item&px=Vulkan-1.0.66-Released. Accessed: 2018-02-25.

[41] L. O. Tolo. *Vulkan Multi-GPU Application.* https://github.com/larso0/vmgpu/tree/sfbench. Accessed: 2018-05-30.

[42] Stanford University. *The Stanford 3D Scanning Repository.* http://graphics.stanford.edu/data/3Dscanrep/. Accessed: 2018-03-01.

[43] C. Eide, J. Howell, and S. Buckley. "Distribution of discontinuous mudstone beds within wave-dominated shallow-marine deposits: Star Point Sandstone and Blackhawk Formation, Eastern Utah". In: *AAPG Bulletin* 98.7 (July 2014), pp. 1401–1429. ISSN: 0149-1423. DOI: 10.1306/01201413106.