Projekt Resumé

I 2016 udgav The Khronos Group Vulkan APIen, med det formål at øge hastigheden på grafikapplikationer, der er CPU-begrænsede. En grafikapplikation er CPU-begrænset, som er når GPUen udfører instrukser hurtigere end CPUen kan sende dem til GPUen. I modsætning til tidligere grafik-APIer, så har Vulkan et lavere abstraktionsniveau, og ifølge vores tidligere forskning gør dette APIen svær at anvende.

Vi præsenterer PapaGo APIen, der forsøger at øge Vulkans abstraktionsniveau, mens den anvender førnævnte API som sin backend. Vores mål er at skabe en API, hvis abstraktionsniveau ligner OpenGLs. Samtidig ønsker vi dog at bibeholde Vulkans statelessness, dens eksplicitte kontrol af GPU-ressourcer samt dens GPU command buffers. Sidstnævnte anvendes til at sende kommandoer til GPUen, og de er centrale i at øge hastigheden af CPU-kode, da de kan genbruges over flere frames og kan optages til i parallel.

Vi evaluerer PapaGo's brugbarhed, hvormed vi introducerer en ny opgavebaseret API-evalueringsmetode baseret på en forgrening af Discount Evaluation. Vores evaluering viser, at vores deltagere, der er mere vante til at programmere med OpenGL eller lignende, hurtigt indpassede sig med vores API. Vi oplevede også at vores deltagere var hurtige til at udføre opgaverne givet til dem.

Et performance benchmark blev også udført på PapaGo ved brug af en testapplikation. Testapplikationen blev kørt på to forskellige systemer, hvor den første indeholder en AMD Sapphire Radeon R9 280 GPU, imens den anden har en mere kraftfuld NVIDIA GeForce GTX 1060 GPU. I det CPU-begrænsede tilfælde af testen, blev hastigheden af applikationen øget på begge systemer når kommandoer optages i parallel. Samtidig kørte applikationen med omtrent samme hastighed på begge systemer, hvor AMD-systemet var omkring 5 millisekunder hurtigere ved højere loads. I det GPU-begrænsede tilfælde har parallel kommandooptagelse kun en lille effekt på applikationens hastighed, mens den kører tre gange hurtigere på NVIDIA-systemet i forhold til AMD-systemet. Ved at sammenligne med den samme applikation skrevet i ren Vulkan, ser vi at PapaGo pålægger en del overhead under eksekvering. I det CPU-begrænsede tilfælde kører Vulkan mange gange hurtigere på begge systemer. Dog i det GPU-begrænsede tilfælde kører PapaGo og Vulkan med samme hastighed på NVIDIA systemet, alt imens at PapaGo kører en tredjedel langsommere sammenlignet med Vulkan på AMD-systemet.

Vi konkluderer at PapaGo har et godt potentiale på grund af de positive resultater fra vores brugbarhedsevaluering, men der skal i fremtiden være et fokus på at reducere det overhead som APIen introducerer.

PapaGo

A graphics API built on top of Vulkan, developed with a focus on programmability

Master Thesis
DPW105F18

Aalborg University
Computer Science

Computer Science
Aalborg University
http://www.aau.cs.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
PapaGo - A graphics API built on top of Vulkan, developed with a focus on programmability

Theme:
Programming Technology

Project Period:
Spring Semester 2018

Project Group:
DPW105F18

Participant(s):
Alexander Brandborg
Michael Wit Dolatko
Anders Munkgaard
Claus Worm Wiingreen

Supervisor(s):
Bent Thomsen

Page Numbers: 124

Date of Completion:
June 8, 2018

Abstract:

In 2016, Khronos Group's Vulkan API was released with the purpose of speeding up CPU-bound graphics applications. In comparison to earlier APIs, Vulkan is more low level, and according to our research this makes it difficult to use.
We present the PapaGo API, which attempts to raise the abstraction level of Vulkan, while using it as a backend. The aim is to create an API with a similar level of abstraction to OpenGL, while retaining the statelessness of Vulkan, its explicit control of resources and command buffers, which can be recorded in parallel.
Evaluating the usability of PapaGo, we develop a new task-based API evaluation method based on Discount Evaluation. Our evaluation finds that our participants, who were more used to an OpenGL-style of programming, quickly adapt to our API.
A performance benchmark is also performed on PapaGo using a test application. The test application is run on two different systems, the first containing an AMD Sapphire Radeon R9 280 GPU, while the second used the more powerful NVIDIA GeForce GTX 1060 GPU. In the CPU-bound case, the application is sped up on both systems by recording commands in parallel, and it runs with about the same speed on both systems, differing with about 5 milliseconds. In the GPU-bound case, parallel command recording has little impact, and the application runs three times faster on the NVIDIA system. Comparing to the same application written in pure Vulkan, we see that PapaGo adds some overhead. In the CPU-bound case, Vulkan runs many times faster on both systems. Yet in the GPU-bound case PapaGo runs at the same speed as Vulkan on the NVIDIA system, while PapaGo is a third slower than Vulkan on AMD.

Contents

Preface

This Master's Thesis is written by group DPW105F18 at Aalborg University. The Group includes Alexander Brandborg and Claus Worm Wiingreen, who present the thesis as part of their Master's degree in Computer Science and Software Engineering respectively. The group also includes Anders Munkgaard and Micheal Wit Dolatko, who present this thesis as part of their Master's degree in Computer Science (IT). Work on the thesis has spanned from February to June 2018. The overall theme of the thesis is programming technologies, while its specific focus has been on graphics programming. In the report, we describe a new graphics API, PapaGo, which builds on top of Khronos Group's Vulkan. Source code and documentation for the finalized API can be found at the following Github repository: [12]. The data referred to in this report can be found at [9].

Acknowledgements

We would like to thank Bent Thomsen for supervising us during the semester. In addition, we would like to thank Martin Kraus for helping us evaluate our initial design of PapaGo. We would also like to thank Kim Johannsen, Christian Nygaard Daugbjerg and a third participant, who wishes to remain anonymous, for participating in our usability evaluation of the implemented API.

Aalborg University, June 8, 2018

Alexander Brandborg
<abran13@student.aau.dk>

Michael Wit Dolatko
<mdolat16@student.aau.dk>

Anders Munkgaard
<amunkg16@student.aau.dk>

Claus Worm Wiingreen
<cwiing13@student.aau.dk>

Chapter 1

Introduction

Graphics programming involves the development of applications, which project images on screen in rapid sequence as to create the illusion of movement. When you see images presented to screen, they may be rendered in real-time, as is the case with video games, or they may be pre-rendered, as is the case with animated movies. In this report we focus on the challenges of real-time rendering, with the intended purpose of being used in games.

Real-time graphics applications are split into two parts: Host-program running on the CPU and shader-programs running on the GPU. Graphics APIs like OpenGL [24] and Direct3D [38] are used to write host-side code, while shader programs are commonly written in C-like shader languages such as GLSL [25], HLSL [39] or Cg [35].

Applications with graphic interfaces use a render loop pattern shown in Figure 1.1. Another term for this pattern is "game loop". The first thing the application does is to query user input, and set the input state accordingly. Then all objects in the game world are updated to reflect the current input state. Lastly, the new state is rendered to the screen by using a graphics API to communicate with the GPU.

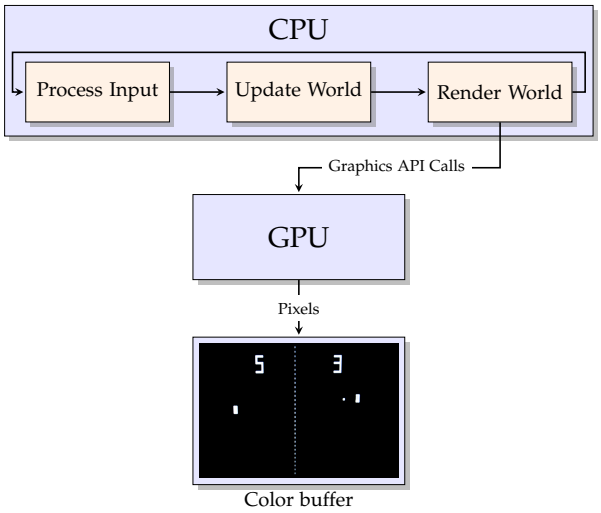

Figure 1.1: Relationship between CPU and GPU in the context of real-time graphics applications.

API calls may serve one of two general purposes: Setting GPU state or drawing to a part of GPU memory known as the color buffer. The state includes, which shader programs should be executed on the GPU, in addition to graphics data like textures and objects. When the state has been set, draw calls are used to signal the GPU to draw. In response, the GPU processes the given graphics data into pixels using the user-defined shader programs as part of an overall graphics pipeline. These pixels are then written to the color buffer. Once all necessary draw calls have been made, the color buffer is presented to screen as a new frame, and the host-side code continues to the next iteration of the render loop.

The number of iterations per second is referred to as the frame rate. For PC games, a frame rate of 60 is commonly targeted by professional game developers, meaning that the host-code only has around 16 milliseconds to process each frame.

When optimizing a graphics application to meet this goal, the programmer must be aware of whether the program is CPU or GPU-bound. In the CPU-bound case, the host-side code acts as a bottleneck leaving the GPU idle. Optimizing shader code will not affect performance in this case, instead the host-code should be optimized. The opposite is true when the application is GPU-bound, where the GPU cannot keep up with the amount of instructions submitted to it from the CPU.

1.1 Modern APIs

In later years, a new generation of APIs for host-side code has been introduced. This includes AMD's now deprecated Mantle [56], its non-proprietary successor Vulkan from the Khronos Group [59], Apple's Metal [57], and Microsoft's Direct3D 12 [55].

Their main design goal is to take advantage of modern GPU architectures to speed up CPU-bound applications. This is done by taking tasks such as low-level graphics command submission, which was previously handled by the graphics driver, and placing them in the hands of developers. Thus developers are granted a more fine grained control over their applications, while driver overhead is decreased.

In addition, by making low-level graphics commands available to developers, lists of commands may now be constructed quickly in a multithreaded manner before being submitted sequentially to the GPU.

While the potential for higher performance is positive, the additional complexity introduced to the APIs risks reducing their programmability. Other authors have already investigated the amount of additional performance granted by the new APIs [19, 45, 4]. In our previous work, we made comparisons of Direct3D 12 and Vulkan not only in regards to performance, but also programmability [10].

Investigating performance, we found that through multithreaded command construction, we were able to increase host-side performance. Running our applications on NVIDIA GPUs, we found that Vulkan was slightly faster than Direct3D 12, and much faster when running on an AMD GPU. This may be because Vulkan, which was developed from AMD's Mantle, took better advantage of the AMD architecture.

To look into programmability, we used a mixed-method approach involving the evaluative methods API Walkthrough [47] and Cognitive Dimensions [17] in addition to static code analysis. We found that, the major distinction between the APIs was that Vulkan has a cleaner interface, while also requiring additional effort from developers. This was thanks to Vulkan being compatible with a larger array of systems, including Windows 7, 8 & 10,

OSx, Ubuntu, iOS and Android, while Direct3D 12 is only compatible with Windows 10 and the Xbox One.

Thanks to their similarity, the APIs suffer from the same programmability issues. Using the terms found in [17], both require a top-down *Learning Style*. This means that users need to gain a deep understanding of graphics programming and GPU architecture to use them effectively.

Both APIs suffer from *Premature Commitment*, as many objects need to be instantiated before any rendering takes place. For instance, shaders used together in the graphics pipeline need to be defined alongside each other in a pipeline state object, meaning that it is not possible to switch out individual shaders at run-time. This requires the developer to know in advance, which combinations of shaders are needed.

The APIs also turned out to have a high *Viscosity*, meaning that it takes a lot of work to make a simple change. For instance, if a developer wants his shader to have access to an additional texture, he must first update the shader code itself, followed by uploading the texture resource to the GPU through the host code. In addition, to make the resource available to the shader, a parameter binding must be defined on a pipeline state object.

For most graphics programmers, these APIs will be difficult to use and add another burden onto the already complex task of graphics programming. However, as the APIs are very close to hardware, they provide a good base for building new high-level graphics APIs, as suggested by Sampson [50]. The amount of control granted by these tools, means that it should be possible to build APIs, which not only have good programmability, but also good performance.

To test this out, we decide to build our own graphics API on top of Vulkan. Vulkan is chosen instead of Direct3D 12, as it is an open platform and has a cleaner interface. In addition, Vulkan had worse programmability, so it would be more beneficial to have an API built on top of it.

1.2 Going Above Vulkan

When building on top of Vulkan, we must define the general level of abstraction to aim towards. We partition graphics APIs into three categories of abstraction, low, middle and high.

Low is represented by APIs like Vulkan, where a detailed understanding of GPU architecture is required. These APIs are characterized as having low driver overhead, pushing low-level operations like command submission onto the developer. They are also stateless, meaning that GPU state is defined through concrete objects up front before any rendering takes place. Through this, state is altered by switching between objects. Wrappers for Vulkan are already in development, such as NVIDIA's VkHLF [43] and AMD's Anvil libraries [1]. However their purpose is to assist developers who already use Vulkan, such as in the case of Anvil, which makes it easier to write portable Vulkan applications.

A middle-level of abstraction is put forward by APIs like OpenGL and Direct3D 11. In this case, developers still need to write both host and shader code, and they must have an understanding of the graphics pipeline. Yet, they do not need a low-level understanding of modern graphics hardware, and the graphics driver takes care of low-level operations. These APIs are also characterized as being stateful, containing an implicit GPU state, which is interacted with through API calls. This allows the developer to make less setup before

rendering, and grants them the ability to alter the underlying state at runtime without much restriction.

At the highest level of abstraction, we have tools where developers do not need an understanding of the graphics pipeline. This is the case with tools like the OpenSceneGraph [46] API for C++, where a graphics scene is structured programmatically through a graph structure. Nodes in the graph represent objects being rendered or operations made upon them. This may include object lighting, which is normally implemented as part of a shader. Thus, entire graphics applications can be written without defining any shaders. Yet, developers are allowed to write GLSL shaders when necessary. We also see shader programming being hidden away in a similar manner in modern game engines like Unreal Engine, which allows for shaders to be defined in a GUI using data flow diagrams or in HLSL for maximum flexibility [23].

In this report, we focus on developing an API with a middle level of abstraction, similar to OpenGL. This is because, there is a great benefit in the flexibility provided by shader programming, and therefore we do not wish to hide it away by raising the level of abstraction further.

Unlike OpenGL, we still want to make a stateless API in the spirit of Vulkan. Thus we are provided with a great deal of flexibility, but by staying explicit it also makes it easier for the developer to keep track of GPU states. However, we want to hide away some of the low-level details and boilerplate code introduced by Vulkan, such as allocating space for submitted commands and resources on the GPU.

Our perhaps biggest issue with Vulkan is how difficult it is to match up host code and shader code, which is in part thanks to a rigid parameter binding system. This has also been an issue when using tools like OpenGL, but it is accentuated when using Vulkan.

Earlier attempts at bringing together host and shader code, has resulted in shader languages being embedded in the host code [37, 20]. This allows shaders to make use of the scoping rules of the host language, in order to make parameter binding less explicit. The downside to this approach is that shaders written in established languages need to be rewritten.

This will also be an issue, if a new stand-alone shading language is introduced. Although in this vein, we take an interest in the recently developed Spire language [26, 27], which allows for modular shaders to be written. Spire is made for use with Vulkan, and allows developers of host code to look up information on parameters of shader modules at runtime. This added information makes it easier to establish parameter bindings to shaders.

Taking inspiration from this system, we decide to add an element of shader analysis to our API. We want to design a system for analyzing GLSL code at runtime, which makes it easier to bind shader parameters on the host side. The reasoning for sticking with GLSL is that we allow developers to reuse their old shaders, which makes it easier to transfer development to our tool.

1.3 Problem Statement

In light of our previous discussion, we decide to build a stateless API on top of Vulkan with a level of abstraction similar to OpenGL, and it should be designed for programmers familiar with that level of abstraction. We hope to hide away some of the lower-level detail of Vulkan, while also retaining some of its beneficial characteristics. In part we wish to focus on bringing host and shader code together, by supporting parameter binding through runtime analysis of GLSL code. Once the API has been build, we wish to evaluate it on both performance and programmability.

With this in place, we formalize our problem statement as:

How may we implement a stateless API on top of Vulkan, which removes unneeded complexity and supports easy parameter binding through runtime analysis of GLSL shaders? How may we investigate the programmability for this new API targeted at developers new to graphics programming?

In the rest of this report, we seek to answer this problem statement in our design, implementation and evaluation of the API.

Chapter 2

Background and Related Work

This chapter is dedicated to set up the rest of the report, discussing both the background of graphics programming and Vulkan, as well as works in the literature relating to our own efforts.

2.1 Background

In this section, we describe some background regarding graphics programming, which is needed to understand concepts in the following chapters. Many of the explanations and figures presented are based on ones found in our earlier work [10, 8].

2.1.1 Graphics Processing Unit

The GPU is the central piece of hardware when discussing graphics programming, as it is responsible for drawing and presenting images on screen.

GPUs may appear in one of two configurations; dedicated or integrated. A dedicated graphics card contains not only the GPU, but also its own BIOS, dedicated Video RAM (VRAM) in addition to output ports like VGA or HDMI. An integrated GPU sits directly on the same motherboard or die as the CPU, and the two processors share system RAM. While the dedicated card is often more powerful, it is also the more expensive option both in retail price and power consumption. The two major vendor's of dedicated graphics cards are NVIDIA and AMD. Unfortunately each vendor has defined their own terminology for the hardware. In the rest of this section, we use NVIDIA terminology for the sake of consistency.

The CPU and GPU are similar in that they are computational units containing multiple processing cores. However as the CPU is used for more general computations and the GPU is mostly used for graphics, they are designed in different ways.

The CPU has a Multiple Instruction Multiple Data (MIMD) architecture, meaning that it is developed for executing different processes, with their own instruction and data, in parallel. For instance, one core may be processing instructions for the operating system, while another is processing a user application. The GPU in turn has a Single Instruction Multiple Data (SIMD) architecture, where the same instruction needs to be executed on

many different data elements. This type of architecture fits well with graphics processing, as the same instruction must be applied to many different graphics elements.

This is the case when moving 3D models on screen. A model is made up of many vertices in 3D space, meaning that they are represented by vectors with three elements. To move a model, every vertex must be multiplied with the same matrix to calculate its new position. Thus we are applying the same operation, matrix multiplication, to multiple different pieces of data, the vertices. The use of linear algebra within graphics programming will be discussed further in Section 2.1.2.

The general GPU architecture can be seen in Figure 2.1. It consists of several Streaming Multiprocessors (SMs), each containing multiple Arithmetic Logic Unit (ALU) cores used to process SIMD instructions. This component can be thought of as an equivalent to the CPU's cores, but the GPU usually contains more than the four or eight cores of modern CPUs. For instance, the NVIDIA GeForce GTX 1060 has 20 SMs, each containing 64 internal cores.

Each core in an SM has some available registers, as well as access to a shared L1 cache. The SMs communicate through a shared L2 cache, or even global memory where elements like the color buffer resides. The SM also contains one or more texture mapping units (TMU). These units are built to sample colors from bit-mapped images, and they are used to texture models being drawn by the SM [58].

The SMs are grouped into different engines, which are scheduled together for execution [32]. On Figure 2.1, we show the two most common engines in modern GPUs, which is the 3D engine used for graphics operations and the compute engine used for GPGPU. Other engines available are the copy engine, used for transferring data, and the decoding engine, used for processing video signals. What engines are available, and how many SMs they contain depend on the individual GPU. It should also be noted that different engines may map down to the same unified set of SMs, meaning that some SMs may play a dual-engine role.

From the CPU-side, we most often communicate with these engines by submitting commands to their specific queues. For instance, if we want to execute some graphics commands, we push these to the graphics queue, and they will be executed in their given order by the 3D engine. While each engine has a logical queue, it may be possible for them to map down to the same physical queue on hardware.

In the following we discuss the SM in more detail.

The Streaming Multiprocessor

Figure 2.2 gives a more detailed view of an SM with eight ALU cores. A collection of threads being executed on these cores are referred to as a warp. The SM contains a single Fetch/Decode component for fetching instructions, as the same instruction must be executed on every core. This is more efficient than having each core fetch its own instructions. The SM also contains storage for an execution context, which includes all data used by a warp such as register contents as well as data shared between the cores.

A user-defined program being executed on an SM is known as a shader. Its name refers to its original usage of coloring/shading objects, but it is now used as a common term for any SIMD program being run on the GPU. On Figure 2.2, we see how a shader is executed by each of the SM's eight cores to color eight pixel fragments of an image. The shader in question is a diffuse shader, which is used to color a surface when a light shines upon it.

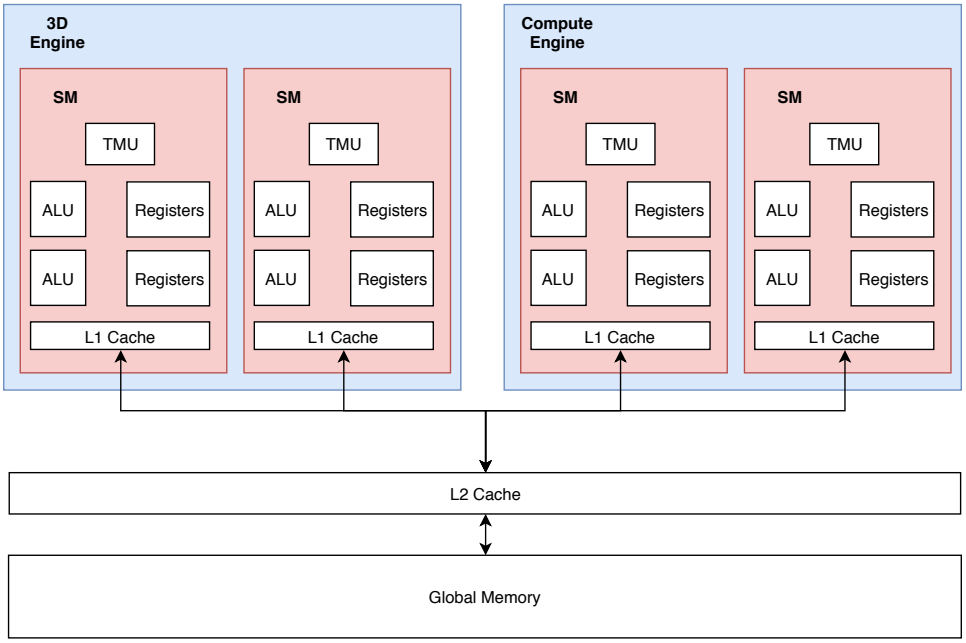

Figure 2.1: A simple GPU architecture with multiple engines. This is an altered version of a figure found in [3].

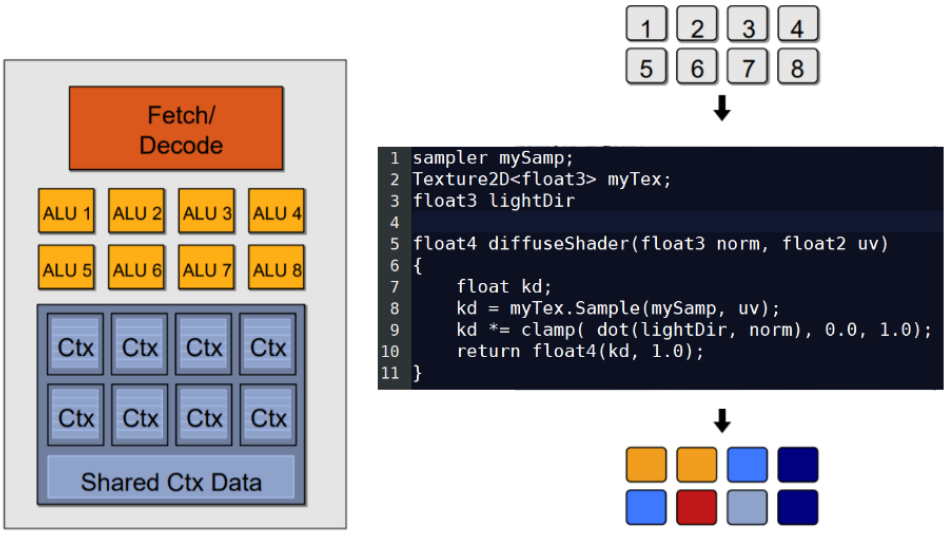

Figure 2.2: SM with eight cores and storage for one execution context used for executing a diffuse shader on eight fragments. This figure is based on ones found in [49].

The structure of shaders is described in detail in Section 2.1.2.

Compared to a CPU-style core, the SM does not contain any components for predicting execution branches or for pre-fetching memory. When a stall occurs on the CPU, such as when a read/write operation is executed, these components allow the CPU to execute

other instructions during the wait. This allows the latency of individual processes to be decreased, lowering their total execution time.

However low latency is not as important on the GPU, as the frequency at which images can be presented on screen is not determined by how fast the first pixel is drawn to the color buffer, but how long it takes for the last pixel to be drawn. Because of this, the GPU is more concerned about throughput, which is the number of instructions that can be executed per unit of time. Therefore space is not wasted on components which lower latency, instead using it for placing more SMs on the GPU, which increases the degree of parallelism.

As on the CPU, throughput is also increased through hyperthreading. While on Figure 2.2 only one execution context may be stored, it is common for SMs to contain space for several contexts, so that it can switch between entire warps in the case of stalls.

2.1.2 Graphics Pipeline

To draw an image to the color buffer, which will later be presented on screen, we use the graphics pipeline. While each stage of this pipeline used to have corresponding units on the GPU, the introduction of the unified shader model means that the same SMs can handle many different stages. We have depicted a simplified version of the pipeline in Figure 2.3, which uses data submitted through host-code to draw a rainbow-textured triangle to screen. Note that we do not depict or discuss the geometry or tessellation stages, which occur after the vertex shader stage. These are mainly used to add detail to models being rendered on screen, but as we do not include these in our project we forego a discussion of them here.

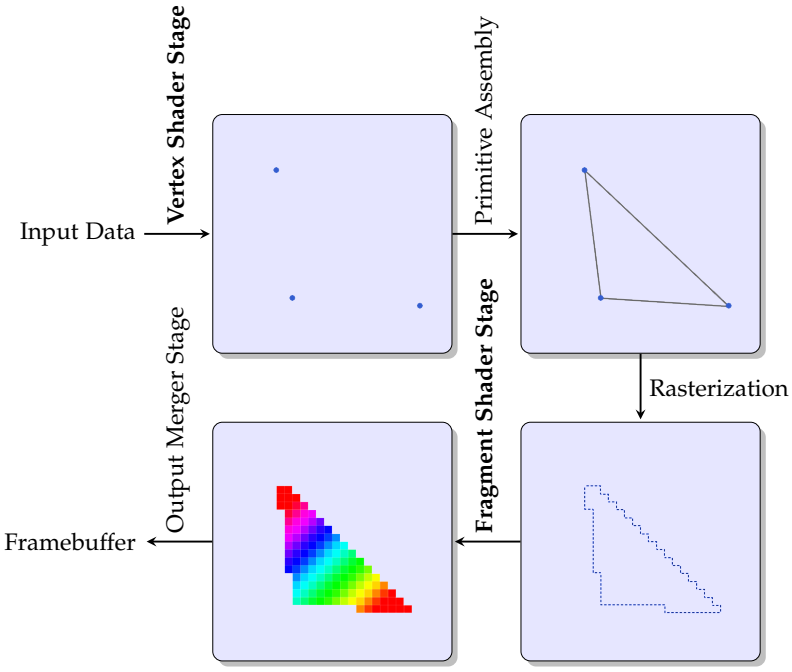

Figure 2.3: Graphics Pipeline with 32 by 32 pixel output. Stages marked with bold are programmable, while the others are fixed-function. This figure is inspired by one found in [54].

Through host-code the developer may arrange how each stage works. Programmable stages, shown in bold on the figure, are programmable, meaning that their behavior is defined by writing shader programs in languages like GLSL, HLSL or Cg. Fixed function stages, non-bold on the figure, are configurable, meaning that their behavior can only be set through function calls.

To draw a model to screen, a single draw call is made from host, which usually requires the vertices of the model to be passed into the pipeline through a vertex buffer.

This buffer defines data elements of each vertex of the model. At minimum this will include the position in 3D space of each vertex, collectively called a mesh, which describe the shape of the model. In Figure 2.4, we show how vertices with positions make up a model, which may later be fleshed out by connecting primitives between points.

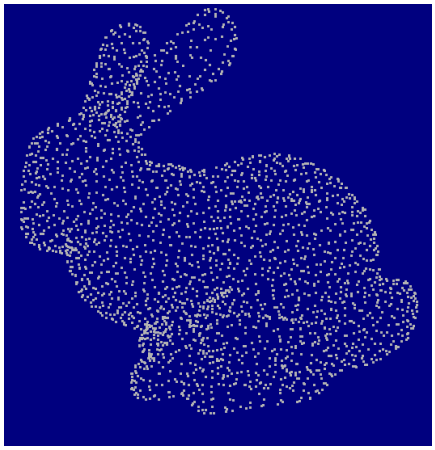

Figure 2.4: Point cloud of the Standford bunny model [34]. Each point represents a vertex defined with an x, y, and z coordinate.

Another element often associated with a vertex is a texture coordinate, which is used to map a 2D texture to a model. The modern process for doing this is called UV mapping, where uv refers to the width- and height-coordinates of the texture. As xyz are commonly used for points in a mesh, uv is traditionally used for texture coordinates. Figure 2.5 shows how a texture can be mapped to a mesh, where each vertex of the model contains a uv-coordinate that maps directly to a point in the texture.

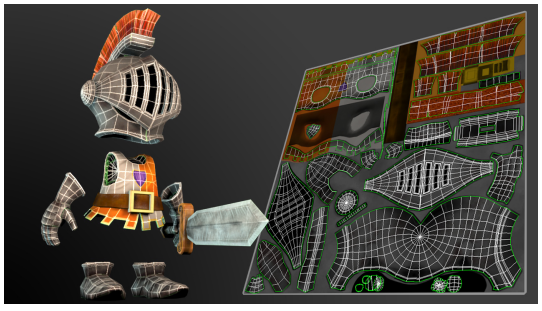

Figure 2.5: Depiction of a model being textured by a texture image through uv-mapping [29]. Each vertex in the model has a uv-coordinate mapping to a location on the texture.

Depending on the primitives being drawn by the pipeline, we may require the same vertex to appear several times in the vertex buffer. For instance when drawing a square using two triangles, six vertices have to reside in the vertex buffer. To save memory, an index buffer can be used to draw the triangles through lookups in the vertex buffer. In this case the vertex buffer only needs four unique entries, and the index buffer needs six entries, three per triangle.

In the following, we describe how each stage is used to draw our rainbow triangle.

Vertex Shader Stage

Data from the vertex buffer is passed directly to the vertex shader stage, which is configured using a vertex shader found in Listing 2.1.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

// Input parameters global to all shader instances
layout(binding = 0) uniform uniformBuffers {
  mat4 projection;
  mat4 view;
  mat4 model
} uboView;


// Input parameters local to a specific shader instance
layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec2 inTexCoord;

// Output parameters to next pipeline stage
layout(location = 0) out vec2 fragTexCoord;

out gl_PerVertex {
  vec4 gl_Position;
};

void main() {
    // Send uv-coordinate unchanged to next stage
  fragTexCoord = inTexCoord;
  // Calculate position in clip space
  gl_Position =
      uboView.projection * uboView.view * uniformBuffers.model * vec4(inPosition, 1.0);
}
```

Listing 2.1: Example of a vertex shader. The shader uses the three transformation matrices; model, view, and projection, to position the vertex in clip space, and forwards the given texture coordinate to later shader stages.

Note that the shaders shown in this section are written in GLSL 4.5, and they conform to the standards set by Vulkan. To be compatible with Vulkan the shader enables the *GL_ARB_seperate_shader_objects* extension, which allows the shader to be compiled into its own object rather than being linked together into a shader program with shaders from the other stages.

As can be seen, most of the shader code is used to define input and output parameters. Shaders take two kinds of input, which are set in the host-code, uniform data and per-vertex data.

Uniforms are global data objects, accessible to all instances of the same shader. Thus, while our shader runs once for each of the three vertices of the triangle, the uniform input will not differ between them. In our code, the uniforms are three 4x4 matrices calculated in host-side, which are defined within the same *uniformBuffers* interface block. The block resides at binding point 0, meaning that host-code must pass in uniforms at this index.

Per-vertex data is sourced from the vertex buffer, and it is passed into *inPosition* and *inTexCoord* input variables. *inPosition* contains the position of the vertex in 3D space as xyz-coordinates. *inTexCoord* contains uv-coordinates for look up into the 2D rainbow texture. The layout location of both variables indicates, where they can be found in the submitted vertex buffer.

Finally there are output variables, defining data being passed onto the next stages. Here we have the variable *fragTexCoord* in addition to the built-in *gl_Position* variable within the *gl_PerVertex* interface block. Note that variables beginning with *gl_* in GLSL are built-in, meaning that they may have special semantics. For instance *gl_Position* is exclusively used for the vertex position passed on through the pipeline.

Using these inputs and outputs in main, the uv-coordinate is passed unchanged through the pipeline and then the position of the vertex in clip-space is calculated by doing matrix factorization using the uniforms.

The position has its xyz-coordinates padded with the homogeneous component w, which is set equal to 1. Then it is transformed through different coordinate systems as depicted in Figure 2.6.

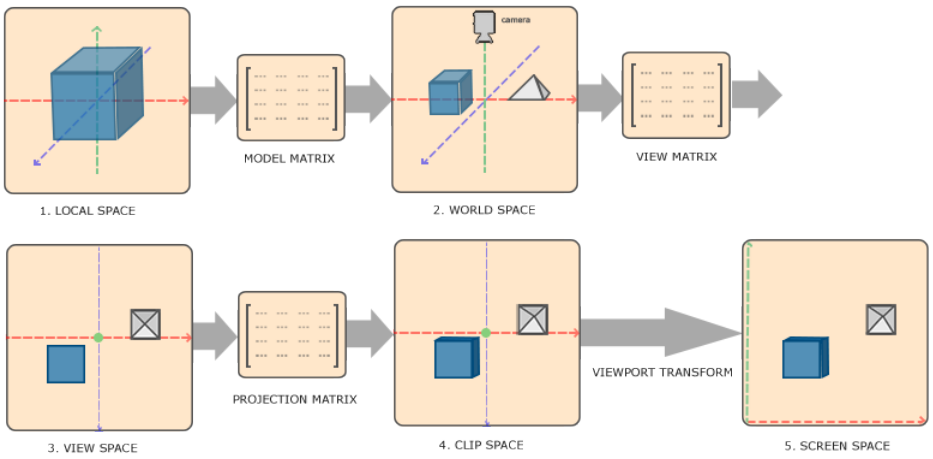

Figure 2.6: Transformations from local space to screen space [54]. The x-, y- and z-axises are the red, green, and blue lines, respectively.

The model is initially in local-space, which means that its positions are defined in relation to the origin (0, 0, 0). If the positions are passed on like this, the model being rendered would have its origin in the middle of the screen. Because of this, the model needs to be moved into world-space, where it will reside in relation to other models in the world. This occurs by scaling, moving/translating and rotating the model by factorizing its vertex positions with special 4x4 matrices. Instead of applying these operation matrices in sequence, they may be factorized together into what is commonly called a model matrix, which is then used to make the transformation.

Once the model is situated in world-space, it must be transformed into view-space, so that it is placed in accordance with a viewpoint. The viewpoint is sometimes referred to as a camera. This transformation is done by factorization with the view matrix uniform, which is defined by the camera position, where it looks at its orientation. To the viewer this emulates the process of changing the viewpoint, while it is actually the model moving around a static camera.

Afterwards the model is moved into clip-space by factorizing with the projection matrix. This transform transforms the vertex positions so that the xy-coordinates lie between -1 and 1, and z between 0 and -1. Any vertex not within these ranges will be clipped and not passed on through the pipeline. If a model i partially within the clip-space, new vertices will be interpolated, which clamps the model to the edge of the screen.

The projection matrix used may either be an orthographic matrix or a perspective matrix. Each of these matrices are defined by a far and near plane, meaning that a vertex too close or too far away from the camera is clipped. The orthographic matrix does not alter the w-component of the vertex position, while perspective makes it scale the component up, the closer it is to the camera and vise versa.

This comes into play during vertex post-processing, which takes place automatically after the vertex shader, but before vertices are passed onto the primitive assembler. A part of this process is the perspective divide, where the xyz-coordinates are divided by w. Here, if $w > 1$ then the positions of our model will be pulled together towards origin, and if $w < 1$ they will be pushed away towards the edges of the screen creating the illusion of perspective. After this, the vertex positions are transformed into screen-space, meaning that they now have the actual coordinates of the viewport on screen to which they are being rendered.

Through all this, our triangle is moved into position as can be seen in Figure 2.3.

Primitive Assembly

When the vertices are in screen-space, they are collected into primitives at the primitive assembly stage. Such primitives may include points, lines, triangles or polygons. In our case, as seen in Figure 2.3, we use the triangle primitive to collect our three vertices together.

After assembly, some of these primitives may be discarded/culled through face culling. A cube is made up of 12 triangles, but a maximum of six can be seen from the viewpoint at any angle. Thus we would like to cull the other six, which are facing away from the viewpoint. Finding out what triangles are front-facing and back-facing is done in this stage by looking at the winding order of the primitive. Commonly the order is specified in a counter-clockwise manner for each primitive in the index buffer, but if a primitive is facing away from the viewer it will naturally be rendered clockwise and therefore culled.

Rasterization

During rasterization, the primitives from the assembler are broken up in small square fragments as can be seen in Figure 2.3. In the simplest case, each fragment maps directly to an entry in the color buffer.

When using triangles as primitives, the position of each fragment is calculated from the position of the enclosing three vertices through barycentric interpolation. This is a form of linear interpolation between three points, where the closer a fragment is to a certain vertex,

the more that vertex influences the value of the fragment. If the fragment is in the middle of a triangle, the three vertices will have equal impact upon the value of the fragment.

Not only the position of the fragment is calculated in this manner, but any vertex data sent on from the vertex shader down the pipeline. In this case it means that the *fragTexCoord* of each fragment is also interpolated using the texture coordinates of the three surrounding vertices.

Because the input from the vertex buffer ends up being changed in this manner during pipeline execution, it is sometimes referred to as varying input.

Fragment Shader Stage

Listing 2.2 shows the fragment shader, which we use to color each fragment of our triangle. This is done using a texture with a rainbow gradient and results in the output in Figure 2.3.

```
1  #version 450
2  #extension GL_ARB_separate_shader_objects : enable
3
4  // Input parameters global to all shader instances
5  layout(binding = 2) uniform sampler2D texSampler;
6
7  // Input parameters local to a specific shader instance
8  layout(location = 0) in vec2 fragTexCoord;
9
10 // Output parameters to next pipeline stage
11 layout(location = 0) out vec4 outColor;
12
13 void main() {
14   // Sample color from Sampler based on uv-coordinates
15   outColor = texture(texSampler, fragTexCoord);
16 }
```

Listing 2.2: Example fragment shader. The shader uses the given texture coordinate to look up the corresponding color in a texture via a sampler.

The only uniform used by the shader is a combined image and sampler object, known as a *sampler2D*. The image is the actual rainbow texture, made up of coloured texels, while the sampler is an object constructed by host that defines how a sample color should be filtered from the image texture. Often the fragment may be either larger or smaller than a texel in the image, meaning that a filtering operation like nearest neighbour or linear interpolation must be used to sample a color.

In addition to the uniform, we also get the interpolated input *fragTexCoord* and the output *outColor*. GLSL implicitly knows that the first output variable specifies fragment color. It is calculated by sampling from the texture with the coordinates using the GLSL *texture* function.

The color format will match with the format of the color buffer, which is often RGBA. Here RGB consist of the three primary colors (red, blue, green) and A is the alpha value indicating the opaqueness of the fragment. Note that the binding index (line 5) is a continuation of the index count from the vertex stage.

Output Merger Stage

This final stage will determine if a fragment should be written to the color buffer and blend its color if this is the case.

The fragment here has to go through different tests, which may be enabled from host. This includes the scissor rect test, which discards a fragment if it resides outside of a rectangle within the color buffer. Afterwards there is the stencil test, which is backed by a stencil buffer with the same dimensions as the color buffer. Here the fragment's stencil value, which may be output from the previous shader, is checked against the corresponding entry in the stencil buffer with a logical operation. If the operation returns false, the fragment is discarded. The content of the stencil buffer and logical operation is defined in host-side and is commonly used to block parts of the color buffer from being written to.

Finally there is the depth test, which is backed by a depth buffer with the same dimensions as the frame buffer. Here the depth coordinate of the fragment z is compared to the corresponding value in the depth buffer. If z is higher than the reference value, it means that something closer to the camera has already been drawn and the fragment is discarded. If it passes the test, its z value is written to the depth-buffer to block other fragments. Often the depth buffer and stencil buffer are combined into a depth/stencil buffer to save space, where each 32-bit entry in the buffer holds both depth and stencil information. A common split is to let the first 24 bits contain depth info, and the remaining 8 bits contain stencil info.

A framebuffer is a collective term for all texture buffers being written to during a frame. In some cases the framebuffer only contains the color buffer, but most often it also contains a depth/stencil buffer. When we say that we draw to a framebuffer, we refer to all contained buffers being written to.

Having passed all tests, the fragment has its colors blended before being written to the color buffer. If its alpha value is 1, it is opaque, and its own color will be written. But if alpha is lower its color is blended with the color already in the color buffer.

The fragment color then resides in the color buffer until it is presented on screen, or if a subsequent draw call within the frame causes it to be overwritten.

2.1.3 Vulkan

This section is dedicated to explaining some of Vulkan's central aspects.

Vulkan was released in 2016 by Khronos Group, and it is the successor to AMD's Mantle [59]. Its main design goal is to lower driver overhead, meaning that a lot of operations which happen automatically in earlier APIs, like OpenGL, now have to be handled by the developer. This becomes apparent, when discussing some of the features of the API in the following.

The explanations and figures of this section is sourced from our previous work [10, 8].

Statelessness

When programming graphics, we configure the GPU state so draw calls have the desired effect. This state contains all aspects of the pipeline configuration, including shaders and fixed-function input, as well as parameter bindings, which link shaders with uniform data. APIs like OpenGL are stateful, meaning that the GPU state is represented by a single underlying API state, which is altered through API calls. The downside to this is that

many function calls must be made, when a large part of the state has to be changed for the next draw call. As the state is altered several times, when a frame is drawn, this creates a risk of becoming CPU-bound.

To get around this, Vulkan discards this global state and instead exposes it as Pipeline State Objects (PSOs). Developers are meant to define every state that they need to use as a PSO before any rendering takes place. Because each PSO is converted to hardware instructions and state at initialization, switching between them only requires some pre-computed state to be copied into hardware registers. Thus, the state switches enacted by host become much faster.

The downside to this method is that the PSO cannot be changed in great part after initialization, which means that developers need a lot of foresight.

Parameter Binding

In APIs like OpenGL, developers may make API calls to construct uniform resources on the GPU, but it is the driver, which decides when these resources are actually going to be uploaded.

To avoid this overhead, developers in Vulkan explicitly allocate memory on the GPU for their resources and upload to it. Resources may be uploaded to two different types of memory, host-visible staging buffers, and device buffers which are only accessible to the GPU. The CPU may quickly upload to a staging buffer using a *memcpy*, but this memory is slow to read for the GPU. It is slower to upload to a device buffer, as the CPU must first upload to a staging buffer and from this copy to the device buffer. However, once uploaded the GPU may quickly read data from the device buffer. The staging buffer is usually used for data which changes frequently, like transformation matrices, while the device buffer is used for data which is more constant, like textures and vertex buffers. This gives developers a lot of flexibility, they need to put extra effort into resource handling. They must also be aware of certain memory requirements, which includes ensuring spacing between data objects in a buffer.

While a vertex or index buffer can be bound directly to a PSO, the process of binding resources for uniforms is more involved. Figure 2.7 shows a PSO with its shaders along with two kinds of parameter bindings: push constants and descriptor sets.

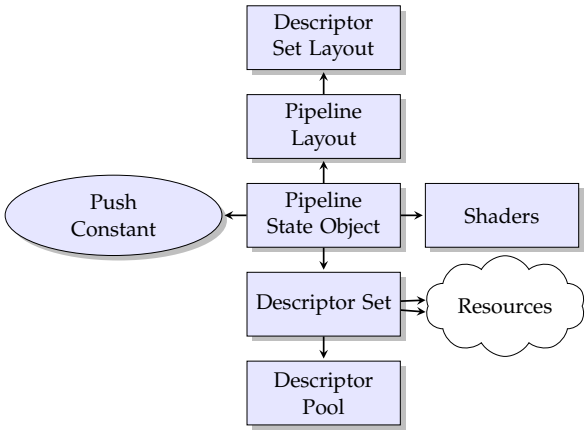

Figure 2.7: Parameter binding in Vulkan with the two kinds of bindings; push constants, and descriptor sets [8].

Push constants are resources pushed directly onto the PSO, and thus they may be quickly accessed from the shaders. However there is not much memory for push constants, we have observed memory size for push constants as low as 256 bytes. Thus they are usually used for small vectors and matrices.

The more commonly used type of binding is the descriptor set, which contains several pointers, known as descriptors, pointing to uploaded resources on the GPU. By using a descriptor set developers can quickly switch out several bindings by simply swapping between sets with a single operation. This is faster compared to parameter bindings in APIs like OpenGL, where the update of 10 parameters would require 10 API calls. This speeds up host code and decreases the risk of CPU-boundness.

However this ability comes at the cost of additional planning time for the developer. Optimally they should ensure that each descriptor set contains parameters, which are often used together. In addition, the sets should be initialized before any rendering takes place.

The developer must also ensure that each descriptor set has space allocated on the GPU through a descriptor pool object. Likewise, when initializing the PSOs they must ensure that it has a pipeline layout that allows it to bind the descriptor sets, they want to use with the pipeline's shaders. Each set we may want to bind has an entry in the PSO's pipeline layout, which is defined by a descriptor set layout object. This object determines the number and types of descriptors contained within the corresponding set.

Command Submission

With APIs like OpenGL, API calls are used to draw to screen, but as in the case with creating resources, the driver determines when these instructions are sent to the GPU. In addition, the driver must convert these instructions to GPU commands and submit them to the queue of the corresponding GPU engine for execution.

Having the driver act as a middle-man creates overhead, so Vulkan instead has developers themselves submit commands to GPU queues. This process can be seen in Figure 2.8, where host creates the commands, collects them into buffers and submits them to a queue on the GPU.

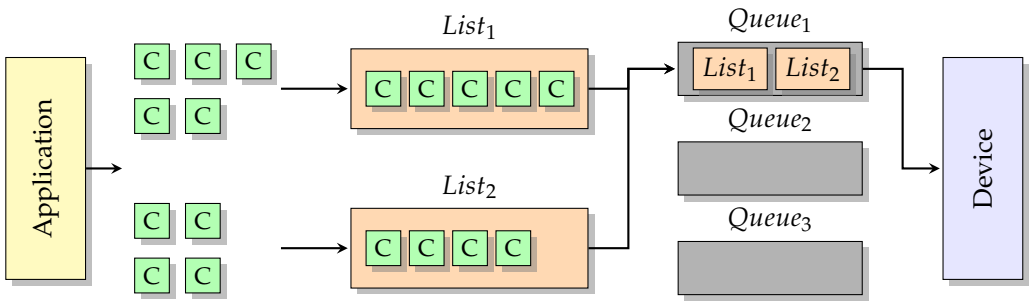

Figure 2.8: The general command submission with command lists and command queues. Figure is sourced from [33, p. 14]

In Vulkan we have two types of command buffers, primary and secondary. Primary buffers are submitted directly to queues, while secondary buffers can only be submitted as part of a primary. The benefit is that all the secondary buffers can be constructed in parallel by the developer as to speed up the host code. Command buffers may also be

retained between frames and reused several times, further cutting down on the time it takes to build buffers.

In addition to containing secondary buffers, a primary command buffer may also contain its own commands, and it is used to begin and end render passes. A render pass encapsulates several draw calls being used to draw to the same framebuffer. The output of a render pass can be presented on screen, or it may be used in another render pass as a uniform resource. The latter is a feature required to implement most lighting systems.

In Vulkan, a render pass is defined through a *RenderPass* object, which may contain several passes within it. It also determines what type of framebuffer should be attached to the render pass.

Listing 2.3 shows how a primary command buffer uses a render pass during recording. Commands are recorded directly in the primary instead of using secondary buffers, as to keep the example simple. In lines 2 – 5, we configure the render pass using a *RenderPassBeginInfo* object, where we set the render pass, what framebuffer to use and the area that should be drawn upon. In this case the frame buffer consists of both a color buffer and a depth buffer. In lines 6 – 10 we define which values to clear the color and depth buffer with, when the render pass starts. Lines 12 – 13 sets up a *CommandBufferBeginInfo* object, allowing it to be submitted several times to the same queue. In lines 15 – 28 we write to our command buffer, which we make sure to reset before beginning to write commands, and we also begin the render pass. Afterwards we bind the PSO we want to use, set up the vertex and index buffers, attach a descriptor set to the PSO and finally make a single draw call using the index buffer. Then the buffer is closed off, and it is ready to be submitted on a queue, where the commands are finally executed.

```
1  //Set up render pass info
2  vk::RenderPassBeginInfo renderPassInfo = vk::RenderPassBeginInfo()
3      .setRenderPass(renderPass)
4      .setFramebuffer(swapChainFramebuffers[frameIndex])
5      .setRenderArea(vk::Rect2D(vk::Offset2D(0,0), SwapChainExtent));
6  vk::ClearValue clear_values[2];
7  clear_values[0].color = vk::ClearColorValue({1.0f, 0.0f, 0.0f, 1.0f});
8  clear_values[1].depthStencil = vk::ClearDepthStencilValue{ 1.0f, 0 };
9  renderPassInfo.clearValueCount = 2;
10 renderPassInfo.pClearValues = clear_values;
11 //Begin info
12 vk::CommandBufferBeginInfo beginInfo = {};
13 beginInfo.flags = vk::CommandBufferUsageFlagBits::eSimultaneousUse;
14 //Record commands
15 vk::CommandBuffer commandBuffer = commandBuffers[frameIndex];
16 commandBuffer.reset(vk::CommandBufferResetFlagBits::eReleaseResources);
17 commandBuffer.begin(beginInfo);
18 commandBuffer.beginRenderPass(renderPassInfo, vk::SubpassContents::eInline);
19 commandBuffer.bindPipeline(vk::PipelineBindPoint::eGraphics, pso);
20 commandBuffer.bindVertexBuffers(0, {vertexBuffer},{0});
21 commandBuffer.bindIndexBuffer(indexBuffer, 0, vk::IndexType::eUint16);
22 // Set shader parameters
23 commandBuffer.bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
24     pipelineLayout,  0, {descriptorSet}, {0});
25 // Draw object
26 commandBuffer.drawIndexed(numOfIndices, 1, 0, 0, 0);
27 commandBuffer.endRenderPass();
28 commandBuffer.end();
```

Listing 2.3: Recording commands to a command Buffer in Vulkan. [8]

The submission of command buffers to queues can be seen in Figure 2.9. Both the graphics queue and present queue shown initially need to be set up by the developer as objects in host code. The figure shows that, like with descriptor sets, command buffers need to have memory allocated on the GPU through command pool objects. When the command buffers have been constructed, they are submitted to the queue of the graphics engine, which will then draw to a frame buffer. However, the developer has to instruct the present engine through its queue to present the image on screen. To make sure an image is not presented, while it is being drawn to, developers must synchronize between the two queues using semaphores.

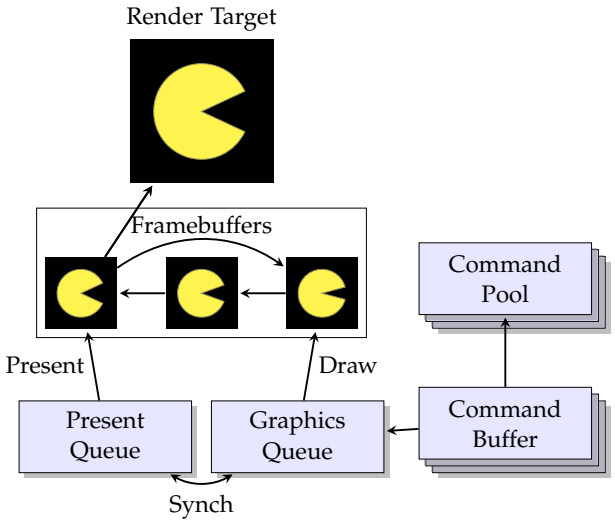

Figure 2.9: The command submission in Vulkan [8]. The names are different from Figure 2.8, but the structure is the almost same.

In this example we use three framebuffers in total as to increase the throughput of frames. If a single frame buffer was used, we would have to wait for the graphics queue to finish before presenting, which is inefficient. Adding one more buffer, ensures that a front buffer can be presented at the same time as a back buffer can be drawn to. After presentation, the two buffers swap roles, and the buffer just drawn to is presented. In our example we add one more framebuffer for what is called triple buffering. This ensures that there is always one frame being presented, one ready to be presented and one being drawn to. However with the additional buffers some delay is introduced. This manner of structuring framebuffers is commonly referred to as a swap chain.

In Listing 2.3, we also use several buffers, picking the framebuffer to render to through an index. The same index is also used to pick the command buffer object to use. We have separate command buffers for each frame, as to ensure that when recording commands for the next frame, we are not overwriting any commands, which have not been executed. If the application becomes GPU-bound, we risk that host begins to overwrite graphics commands of a framebuffer, which has not had its last batch of commands executed. To ensure that the old commands are allowed to be executed, fencing can be used in host code, to halt it from proceeding until a frame has been drawn, and its commands can be overwritten.

2.2 Related Works

Our work in this report relates the development of other graphics programming tools within the literature, which focus on improving the programmability of current offerings. The attempts described can be categorized into two sections, those focusing on increasing the programmability of shaders and those focusing on the host code. While our work concerns the programmability of host code, both categories are covered in this section.

2.2.1 Shader-centric

Tools, which focus on shaders and shader languages are by far the most plentiful within the literature.

Spire is a new experimental shading language used for multi-rate programming [26]. Shader programming in languages like GLSL are single-rate, as each shader corresponds to a specific stage in the pipeline. This results in operations within the shader occurring at the same rate, vertex shaders execute per vertex, fragment shaders execute per fragment and so on. In multi-rate shader programming operations within a shader may map to several different shader stages. This allows for all GPU code to be written within a single program.

Spire's claimed reason for introducing multi-rate programming is to allow developers to experiment, with what operations occur at which stage. Something like light can be calculated per vertex or per fragment. If a developer wants to switch between these in GLSL, they need to write two sets of vertex and fragment shaders. In a Spire shader an expression can per default be executed at any stage. Before the shader is run, the developer chooses what stage to run the expression in using a separate configuration file. In this manner a developer can easily experiment with switching the rates of different operations, while using the same shader. As some expressions only make sense when executed in certain stages, the developer may annotate them to restrict, where they are used.

A common complaint within shader programming, is that to develop shaders in a modular way, they have have to be pieced together as strings in host code with preprocessor directives [50]. As the number of modules grows, this becomes difficult to manage. Therefore, Spire was later extended with modular shaders [27]. Modular shader programming is beneficial as some shader concerns like lighting may stretch across several shader stages. Thus to separate concerns, developers want a good way to write shader modules across stages. In Spire, modules may now be written in a multi-rate manner, which are collected together in a main shader file. This makes it easy to switch between different modules. An added benefit of the modules is that developers can use the Spire API to perform runtime reflection of shaders, looking up which uniforms each module needs. Since the uniforms of modules are used together, it makes sense for developers to build them together into descriptor sets, leading to more performant code. Without this sort of guidance, it is possible that developers will group all parameters into the same set, foregoing any performance benefits.

In light of this, Spire has proven to be the only shader language, which in its design concerns itself with modern graphics APIs like Vulkan. Other publications in this section are older, and are thus designed for APIs like OpenGL.

While Spire is imperative in nature like GLSL, we also see functional shading languages like Renaissance [2]. The philosophy of this language is that each pipeline stage should be

seen as a function with no side-effects, which takes input from an earlier stage and outputs data to the next in the pipeline. Therefore it is argued that writing shaders as functions is more natural, than writing them in an imperative manner. This approach however does not seem to have caught on.

In addition to dedicated shader languages, the literature also presents some languages which are embedded in host code, such as Sh for C++ [37, 48, 36] and Vertigo for Haskell [20]. This allows the host code and shaders to tie neatly together, as the embedded shader can make use of the scoping rules of the host language to gain access to GPU resources defined in host-code. This foregoes shader parameter bindings, which in our own experiences with Vulkan proved to be difficult [10]. The embedded language is also able to make use of other features of the host language such as functions and classes. One disadvantage of embedded shaders, as mentioned in [2], is that they cannot be easily shipped between projects, as they are heavily tied to host code.

While these experimental shader languages are interesting, they suffer under the fact that languages like GLSL, HLSL and Cg are popular for shader programming. Many shaders have already been written in these languages, and they are often reused and migrated between projects. This ensures that new shading languages need to have a great benefit attached for developers to begin using them. This is in part why, we in this report focus on a tool for writing host code instead.

2.2.2 Host-centric

Literature describing tools built on top of modern graphics APIs like Vulkan and Direct3D 12 seems almost non-existent. In great part this must be a result of these tools being rather young.

Bossard [7] describes the development of a system for using modern Direct2D, which builds on top of Direct3D 12, within the functional language Racket. However unlike our tool, this system functions more like a wrapper around Direct2D and does not create an entirely new API.

Looking back to the earlier generation of graphics APIs, we do find a small handful of tools using OpenGL as a backend. This includes Halide, which is used for digital image processing, however it cannot be used for graphics programming as described in [16].

While the field of research into host-side graphics programming has proven to be rather small, the similar field of GPGPU programming has grown greatly since its initial spark of interest around 2010 [41]. GPGPU refers to the usage of GPUs for more general computations rather than strictly for graphics. Here developers use frameworks like CUDA and OpenCL to write both host code as well as programs to be run on the GPU, known as kernels. Yet, these tools have proven difficult to use because of their low level of abstraction, and thus there are plenty of tools, which seek to simplify GPGPU programming[30, 51, 44]. Interestingly, while the literature within graphics programming focused mostly on code running on the GPU, most of the literature for GPGPU programming focuses on tools for writing host code.

One example is HPL, which is a library meant to be used with OpenCL [53], assisting in creating GPU resources and synchronizing between CPU and GPU. A similar library also exists for CUDA known as CuPP [14].

In regular OpenCL, the user has to manage GPU resources and data transfers between CPU and GPU manually like in Vulkan. Given the context of OpenCL, data is transferred

to the GPU as input to kernels, and back again when the kernel has stopped running. Rather than handling this manually, HPL allows developers to store their data in a host-side *Array* object, which will then lazily handle data transfers between the devices. Thus data is only uploaded to the GPU, when a kernel is invoked, and data is only transferred back when the user needs to use it. If the user requests output from a kernel that is still running, the host will halt until the kernel finishes. It is also possible for the developer to specify, if data should be uploaded to GPU memory local to a specific SM, or if it should be globally accessible.

When executing a kernel in OpenCL, the developer must have set up and specified the device to use. HPL makes this a bit easier, implicitly using the first device on the system, unless otherwise specified by the user.

Chapter 3

API Design

This chapter concerns the design of our API, which we name PapaGo. Here we will describe both the design process behind the API, as well as its specification, in detail.

In the past, APIs were designed in a more ad-hoc fashion to fit their purpose. Yet in recent years, there has been a surge of interest in evidence-based research of API design and evaluation. This is in part thanks to calls for research, such as Daughtry et al. [18], into the field. However, while the research has become more popular, it has not grown to a size comparable to that of research into the design and evaluation of graphical user interfaces. We experienced this lack of publications ourselves, when we had to compare the programmability of Direct3D 12 and Vulkan, where we were only able to find a small handful of evaluating methods [10].

Myers and Stylos [40] describes some of the main challenges concerning API design. The first one is that we must balance the conflicting wants and needs of the three groups impacted by the API: API designers, API users and product consumers. For instance, an API designer may want to redesign parts of the API specification, but an API user does not want to update their code in response to the change. The second challenge is that the API design must balance usability with power. Usability defines the ease with which the API may be used, while power defines the flexibility of the API and the performance, which it may bring. An API that is easy to use, may end up abstracting away too many details of the underlying system, which can end up hurting flexibility and vice versa. The third challenge is that many design decisions have to be made at different levels, all affecting usability. At a high-level, decisions may be made about architecture and design patterns, while at the low level decisions about function parameters and variable naming must be made.

In our design, we focus on the user and their ability to make programs for the consumer. In Vulkan's case, power was favored over usability, but in our design we aim to balance these two properties, putting a greater emphasis on usability. To make sure that the many different design decisions being made are not performed in an ad-hoc fashion, we develop the API according to the process described in the next section.

3.1 Design Process

Our design process is based on the nine-step method for API design as described in [5]. It includes the following steps:

1. Define user requirements.
2. Write use cases for the API.
3. Take inspiration from similar APIs.
4. Define the API.
5. Have your peers review the API.
6. Write examples with the API.
7. Ready it for extension.
8. Do not make the API public before the time is right.
9. Leave out functionality, which may not be necessary.

Our approach uses the six first steps of the method. Step seven is excluded, as we are not expecting to extend the API in the future. Steps eight and nine function more as overall guidelines and not steps in the design process. Thus, while they are kept in mind during our design, they are not an explicit part of our design process.

By following the six first steps of the method, we split the design process into two; the initial design through steps 1 – 4 as well as the peer review and redesign in steps 5 – 6. These are described in detail in the following subsections.

Initial Design

In Section 1.3, we describe our target user group as being new to graphics programming. [5] suggests discovering user requirements from potential users, but as we ourselves fall into the user group, we gather the initial requirements from ourselves. Our requirements are based upon our own experiences with Direct3D 12 and Vulkan as described in [10]. They are defined in the following:

1. The API should be able to speed up CPU-bound applications.
2. Parameter binding should be intuitive.
3. Resource handling should be implicit.
4. The API should be stateless.

As Vulkan's main benefit is to speed up CPU-bound applications, we would like this ability to persist in our API. Because of our own experience with coding in Vulkan, we wish to raise the level of abstraction so that it is similar to OpenGL. Therefore we do not want the user to worry about the intricacies of parameter binding with descriptor sets, layouts and pools. We also want resources to be transferred implicitly between the CPU and GPU, so the user does not have to allocate memory on the GPU explicitly.

On the other hand, we wish to keep the benefits of using Vulkan. This includes having concrete PSOs rather than the implicit state of OpenGL, as we judge this to make it easier to understand program execution.

Having now set up these requirements, we define some use cases, which the API must support:

1. Draw several textured cubes.
2. Draw several textured cubes with parallel command submission.
3. Draw a shaded cube using shadow mapping to cast shadow on other objects.

In regards to the first case, it is very common for users to want to draw a textured object which requires most of the API's basic features to be used. The second case focuses on the ability to record commands in parallel, thus making the application less CPU-bound. The third case describes a more complex application, where several sets of shaders must be used to create lighting. Shadow mapping is explained in more detail in Section 6.2.

Before defining our API in full, we implement these use cases, coding up against the undefined API. In this manner, we ensure that we are not restricted by API implementation details during our design process. This method of design is also described in [6].

As we have four members in our group, we do not code against a single undefined API, as described by the method. Instead each member writes their own use case implementations, effectively trying out four different API designs. This allowed us to explore many ideas quickly, which were further discussed collectively within the group. After our discussion, we were able to agree upon some common features and rewrote the implemented use cases accordingly.

These implementations can be seen in Appendix A. It is here noticeable that we have taken a lot of inspiration from Vulkan, keeping the use of the state-encompassing PSOs in addition to command buffer. The PSOs are now also used in a similar manner to render pass objects in Vulkan.

Combating CPU-boundness still occurs by recording commands in parallel using a thread pool. However parameter binding is made more intuitive, occurring by name look-up into the bound shaders. Resource handling is also kept simple with framebuffers being created implicitly.

From these implemented use cases, we were later able to create a full specification for the API. This was put to the test during our peer review.

Peer Review

As no member of the group had extensive experience with graphics programming, we were not sure how our users would perceive PapaGo. Therefore we got in contact with associate professor Martin Kraus, who is a lecturer at Aalborg University's Department of Architecture, Design and Media Technology, to help review our initial design. He has about 20 years of graphics programming experience, and while he had only heard of Vulkan, he is very well versed in OpenGL. While he is not a part of our target user group, but is very familiar with this group through his lectures, we assume his input as useful.

To perform the review, we used the API Walkthrough method, which is designed to evaluate APIs, which have not been implemented yet [47]. The method entails having our participant walk through code, which uses our API. Meanwhile they should be using

think-aloud protocol to describe their thought process. This gives us an idea, of how the participant is thinking. During the review session a facilitator sits next to the participant to facilitate the process, answer questions from the participant and keep them talking.

Executing this method, we initially made a small presentation of Vulkan and the purpose of PapaGo. Then we had our participant walk through the implemented use cases from Appendix A. After the walkthrough, we performed a follow up interview to get some final thoughts on our design. The session took a bit over two hours, and points of interest were noted down in a document, which can be found in [9].

Our participant had heard of the complexity of Vulkan and found our API to be an interesting development. However, he had concerns with how we handled our resources implicitly, not being sure when and where off-screen buffers were defined in our examples. Therefore, we chose to redefine our user requirement, so that instead of handling resources implicitly, users must construct them explicitly and handle their transfer between CPU and GPU.

In Vulkan, this required the user to do pointer arithmetic and consider device memory requirements, which we want to hide. The participant also had difficulty understanding parallel command submission with threads, suggesting that even experienced graphics programmers are not always well-versed in parallel programming. Thus, there was a need for a simpler manner of submitting commands in parallel without using threads explicitly. In response to this, we update our design requirements as follows:

1. The API should be able to speed up CPU-bound applications.
2. Parameter binding should be intuitive.
3. Resource handling should be explicit but intuitive.
4. The API should be stateless.

Having done this, we implemented the use cases again and updated the API specification, which we then began to implement. [5] does not describe how to implement APIs, but does mention that implementation details are allowed to leak through the specification for added performance. This may involve the user being allowed to give non-default input to functions as to gain additional control and move away from default behavior. During implementation, we found some edge cases, which our specification did not cover. Because of these factors, our specification needed to change quite a bit during the implementation proces. The main features of our final design are discussed in the following section.

3.2 The Specification

This section describes the overall model of execution of PapaGo as well as some of its central concepts. The version of the specification described here matches the implementation of the API found in [11].

3.2.1 Overall Model of Execution

In Figure 3.1, we have depicted how an application written in PapaGo should be structured and executed. By explaining this structure, we aim to give an overview of the API, before covering some its the core concepts.

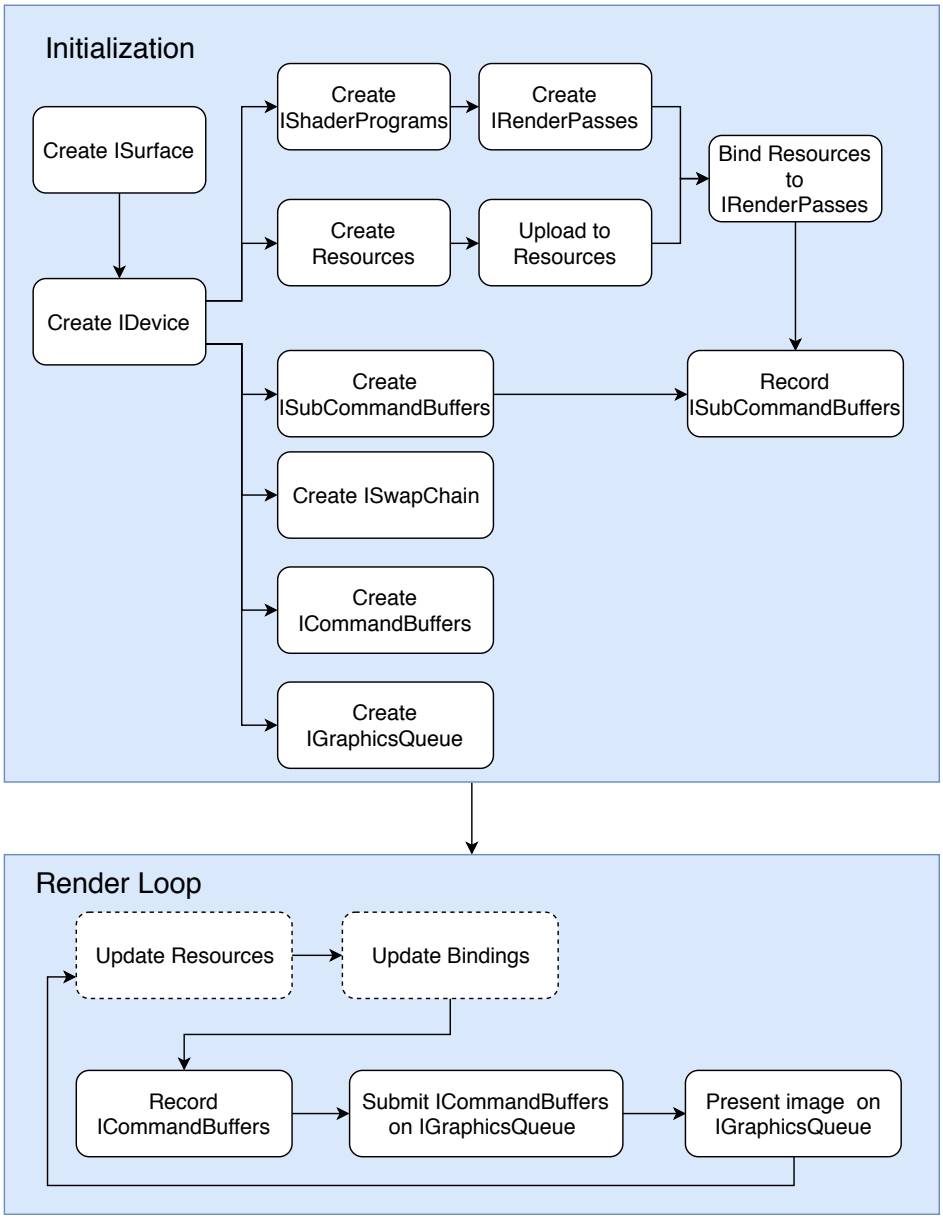

Figure 3.1: Regular execution process in PapaGo.

Figure 3.1 is split into two parts: initialization and render loop. Like in Vulkan, most objects should be created during initialization. This allows for minimal computation to be done at each iteration of the render loop, which renders new images and presents them on screen and yields a higher frame rate.

Initialization begins by creating an *ISurface*, which represents an area of the screen, which the application should render to. Afterwards this surface is used to create an instance of the *IDevice* class, which represents the GPU used for rendering images and pre-

senting them on screen. This is used to create most other API objects, functioning as the central class of PapaGo. Objects like *ICommandBuffer*, *ISwapChain* and *IGraphicsQueue* function similar to their Vulkan equivalents. They are created at initialization, but otherwise not used until the render loop is entered.

In addition to creating these objects, the initialization is mainly in place to prerecord commands, which are used several times during application lifetime. By prerecording the commands, we further minimize the amount of computation needed per iteration of the render loop. These commands are recorded into one or more *ISubCommandBuffer* objects. To fill an *ISubCommandBuffer* with commands, the developer needs to create both an *IRenderPass* object and GPU resources, which will be used as shader parameters. In PapaGo, an *IRenderPass* represents both a pass over a given render target, which may reside on the established *ISwapChain*, but also the GPU render state. Therefore it needs to be constructed using an *IShaderProgram*, which includes the shader modules that define the underlying graphics pipeline to render with.

Resources may be vertex and index buffers, as well as buffers containing vectors, matrices or image textures. Some of these resources are initialized to be empty, which requires developers to upload data to them before they can be used. Once both an *IRenderPass* object and resource objects have been constructed, the resources can be bound to the render pass, which pairs the resources up with the shader uniforms in the underlying graphics pipeline. After the *IRenderPass* has been bound to, it may be used to record commmands into one or more *ISubCommandBuffer* objects. When initialization ends, several objects are ready for the render loop, including prerecorded *ISubCommandBuffers*.

During the render loop, there are some optional initial steps of each iteration, where the resources may be updated with new data. This may be necessary, e.g when a rendered object needs to be moved by updating its model matrix. It is also possible that other resources can be bound to a render pass at this stage. This could be done for performance reasons, instead of updating a resource, it would be faster to switch to one, which already contains the needed data. When the resources have been updated and bound, one or more *ICommandBuffer* objects are recorded. Into these we may record commands directly, or we may emplace the commands found in *ISubCommandBuffers* created during initialization. After recording, these buffers can be submitted on the *IGraphicsQueue*, which will execute the commands and draw them to a given render target on the *ISwapChain*. When the swap chain has been drawn, it is then possible to present one of its images to screen through the *IGraphicsQueue*.

Having now described PapaGo in broad terms, the following sections focus on some of the API's core concepts.

3.2.2 IDevice

IDevice is the central class in PapaGo, as it represents a specific GPU device, which renders images and presents them to screen. Therefore an *IDevice* object is used to create all objects in the API, which require memory to be allocated on the GPU. The idea behind this class comes from both Direct3D 12 and Vulkan, which have equivalent classes. While Direct3D 12 has a single device class, Vulkan splits the functionality into two classes: *PhyscialDevice* and *Device*. Thus, our design takes more inspiration from Direct3D 12 than Vulkan, as this is a simpler way of initializing devices.

As multiple graphics devices may reside on a given system, we want the programmer

to be able to select from all of them at once. Therefore we take inspiration from the *IDXGIFactory::EnumAdapters* method of the Windows SDK. In PapaGo the equivalent method is called *IDevice::EnumerateDevices* and returns a vector of *IDevice* objects, representing all available devices.

Listing 3.1 shows how the method is used to get a collection of devices, which can then be picked from. It requires three parameters, a surface to draw to, a set of features to enable, and a set of extensions to enable.

The surface is represented by an *ISurface* object, which wraps around a window created by the operating system. Right now PapaGo only supports creating surfaces from Win32 windows, which is done by using the static *ISurface::createWin32Surface* method. Because of this, the current version of PapaGo will only work on systems running the Windows operating system.

The features used describe functionality of a GPU, which needs to be enabled before being used. In the example, we enable anisotrophic filtering for textures; any GPU which does not support this is not returned.

The extensions describes extensions to Vulkan that the device can use. In the example, we enable the use of a swap chain for arranging render targets; any GPU not supporting this is not returned.

In Listing 3.1, every object created by PapaGo is returned within a unique pointer. This is done throughout the API as to assist in memory management, which we will go into details with in Section 4.2. In the rest of this chapter, we use the auto keyword and reference variables to simplify the examples presented.

```
#define HEIGHT 600;
#define WIDTH 800;

int main() {

  /* Creation of win32 window and hwnd handle to it omitted */

  std::unique_ptr<ISurface> surface = ISurface::createWin32Surface(
    WIDTH, HEIGHT,
    hwnd);

    IDevice::Features features;
  features.samplerAnisotropy = true;

  IDevice::Extensions extensions;
  extensions.swapchain = true;

  std::vector<std::unique_ptr<IDevice>> devices = IDevice::enumerateDevices(
    *surface, feature, extensions);
  std::unique_ptr<IDevice> device = devices[0]; // Pick first available device
}
```

Listing 3.1: Enumeration of available *IDevice* objects that support the anisotropy feature and swapchain extension.

3.2.3 IShaderProgram

An *IShaderProgram* contains a number of shader objects, each representing a programmable stage of the graphics pipeline to be used.

Listing 3.2 shows the construction of an *IShaderProgram* object. This is done by first creating a *Parser* object, which requires the path of a GLSL to SPIR-V compiler. While shaders are supposed to be written in modern GLSL for PapaGo, the Vulkan backend requires them to be compiled to the intermediate SPIR-V language. For our own programs, we have used the Khronos Group's glslangValidator executable.

The parser can then be used to compile the shaders, which requires shader source code and the name of an entry point into the given shader. Commonly the *main* function is used as an entry point, but any shader function may take this role. Using the parser to compile the shader code, it is returned as an *IShader*. Multiple shader objects can then be used to construct a *IShaderProgram* object.

The concept of a shader program can also be found in OpenGL, denoting a group of individual shaders, which have been linked together in a program. Unlike the OpenGL variant, our shader program does not perform any linking of the shaders. These are instead kept as individual shader modules, as required by Vulkan, which does not use fully linked shader programs.

At the moment, only two types of shaders are supported by PapaGo: Vertex shaders and fragment shaders. While support for tessellation, geometry and compute shaders would increase flexibility, many non-trivial graphics applications can be developed without them.

```
char vertexShaderSource[] = {/* GLSL source code */};
char fragmentShaderSource[] = {/* GLSL source code */};

// Instantiating parser
auto parser = Parser(/* Path to GLSL to SPIR-V compiler */);

// Compiling shaders and creating an IShaderProgram
auto vertexShader = parser.compileVertexShader(vertexShaderSource, "main");
auto fragmentShader = parser.compileFragmentShader(fragmentShaderSource, "main");
auto shaderProgram = device->createShaderProgram(*vertexShader, *fragmentShader);
```

Listing 3.2: Creation of an *IShaderProgram* object.

3.2.4 Resource Management

In Section 3.1, one of the original requirements for PapaGo was that resources uploaded to the GPU should be handled implicitly. In response to our peer review, we later updated this requirement, which states that resource handling should be explicit like in Vulkan. However, it should not require the developer to perform pointer arithmetic or worry about memory requirements.

Figure 3.2 shows the relationship between the different resources, which are available through PapaGo. In this chapter, the API is explained from a user point of view. Please note that in the underlying implementation *IBufferResource* and *IImageResource* inherit from the same *Resource* class. Thus all resources, which are instances of *IBufferResource* or *IImageResource*, have the same methods for uploading to, and downloading from, the allocated

data on the GPU. As can be seen in this section, all upload methods are templated, allowing developers to upload any data format to the GPU.

In the following, we describe the different types of resources, and how they are created. How the different resources are used for rendering, will be described in Section 3.2.5 and Section 3.2.6.

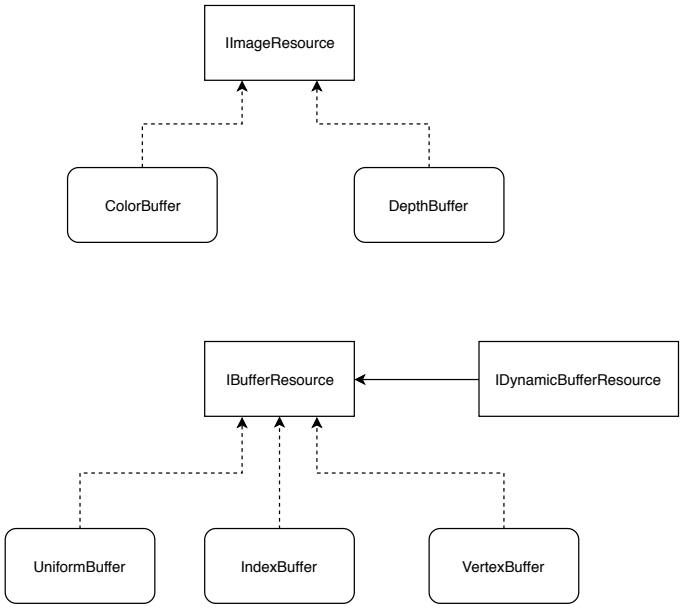

Figure 3.2: Relationship between resources in PapaGo. Square boxes represent API classes. Solid arrows between classes indicate that the pointing class contains an object of the class pointed to. Each rounded box represents an instance of a class. A dashed arrow from an object to a class, indicates the class which the object is an instance of.

Vertex and Index Buffers

As explained in Section 2.1.2, vertex buffers are used to submit vertex data to the graphics pipeline, and for each vertex a single vertex shader instance is executed. Each element in a buffer may contain several vertex attributes of a given vertex, including position, texture coordinates and surface normals.

An example of how a vertex buffer is created within PapaGo can be seen in Listing 3.3. First a vector of floats with vertex data is initialized, which describes the four corners of a square. In the example, this is styled so each row corresponds to a particular vertex with the first three elements representing a position attribute and the following two elements representing a texture coordinate attribute.

```
1 // Set up vertex buffer
2 auto vertices = std::vector<float>{
3     { -0.5, -0.5, 0.5,  0.0, 0.0,  // Upper left corner
4       -0.5,  0.5, 0.5,  0.0, 1.0,  // Lower left corner
5        0.5,  0.5, 0.5,  1.0, 1.0,  // Lower right corner
6        0.5, -0.5, 0.5,  1.0, 0.0}; // Upper right corner
7 auto vertexBuffer = device->createVertexBuffer(vertices);
```

Listing 3.3: Creation of *IVertexBuffer* containing a vertex data for a textured square.

This vector is then submitted to the *IDevice::createVertexBuffer* method on an *IDevice* instance, which creates an *IBuffeResource* object representing a vertex buffer on the GPU containing the vertex data. While the vector contains floats in this example, the method is templated like other upload methods so any type of vector can be given as input. No matter the data type submitted it is converted to a byte stream, uploaded to the GPU, and later interpreted by the vertex shader as floats.

In APIs like OpenGL and Vulkan, the developer must define how attributes are to be read from the vertex buffer. This involves the developer defines the size of each attribute and the strides between them. One of the requirements for PapaGo is *Parameter binding should be intuitive*, so PapaGo instead automizes this process. This is done by having the API analyze the vertex shader being used, and reading the vertex buffer in the manner expected by the shader. So when using the buffer created in Listing 3.3, the corresponding vertex shader must expect the first attribute of a vertex to contain three elements, and the second attribute to contain two elements.

While developers may render using only a vertex buffer, PapaGo also provides the use of an index buffer. Creation of an *IBufferResource* object representing an index buffer can be seen in Listing 3.4. This requires a vector of integers, either *uint16_t* or *uint32_t*, to be submitted as a parameter to the *IDevice::createIndexBuffer* method. In the example, the index buffer defines the two triangles making up the square, defined within the vertex buffer in Listing 3.3.

```
// Set up index buffer
auto indices = std::vector<uint16_t>{
  0, 1, 2, // 1st triangle
  0, 2, 3  // 2nd triangle
};
auto indexBuffer = device->createIndexBuffer(indices);
```

Listing 3.4: Creation of *IIndexBuffer* for the square from Listing 3.3.

Uniform Resource

As stated in Section 2.1.2, uniforms are shader variables global to all shader instances. The two main types of uniforms available are buffers and textures. Uniform buffers contain data like light positions and transformation matrices, while textures contain image data.

The creation of an *IBufferResource* object, representing a regular uniform buffer, can be seen in Listing 3.5. Here a 4x4 matrix is constructed using glm, a math library designed for use with OpenGL. The buffer is then created using the *IDevice::createUniformBuffer* method, sizing the memory allocated on the GPU to the matrix size. Once the buffer has been created, the matrix is uploaded to it.

```
glm::mat4 uniform_data = /* definition of transformation matrix */;
auto uniform_buffer = device->createUniformBuffer(sizeof uniform_data);
uniform_buffer.upload(uniform_data);
```

Listing 3.5: Creation of a uniform buffer, which can contain a 4 by 4 matrix.

This type of uniform buffer is supposed to be used for uniforms, where the contents do not change over a render pass. Thus it is meant to store data like light positions, in addition to view and projection matrices. If the developer attempts to upload data to this

type of buffer during a render pass, he will not be able to control what draw calls this change will affect.

Therefore this type of buffer cannot be used for uniforms, which need to change between draw calls, such as model matrices. PapaGo includes a feature from Vulkan, known as a dynamic uniform buffer. Unlike a regular uniform buffer, this buffer may contain multiple objects. Thus the developer can upload all model matrices, they need before executing a render pass. During the pass, they may then switch between the elements of the buffer, which are to be used as uniforms.

Listing 3.6 shows the creation of an *IDynamicUniformBuffer* object. It is sized, so that it may contain two 4x4 matrices. After creating the buffer, the matrices defined are uploaded to it by index. It is also possible to upload to the buffer at once, by submitting a vector of entries instead.

```
glm::mat4 uniform_data0 = /* definition of model matrix */;
glm::mat4 uniform_data1 = /* definition of model matrix */;
auto dynamicBuffer = device->createDynamicUniformBuffer(sizeof(glm::mat4), 2);

dynamicBuffer->upload<glm::mat4>(uniform_data0, 0);
dynamicBuffer->upload<glm::mat4>(uniform_data1, 1);
```

Listing 3.6: Creation of a dynamic uniform buffer. It is also possible to initialize the buffer with a vector.

In addition to buffers, textures may also be used as uniforms in shaders. Listing 3.7 shows the creation of an *IImageResource* object, representing a two dimensional color texture, using the *IDevice::createTexture2D* method. This requires the texture to have its dimensions defined, along with its color format. After creation, the texture's pixels can be uploaded to the GPU. This uniform is most often used to texture the surface of models being rendered.

```
int textureWidth = 800, textureHeight = 600;
std::vector<char> pixelData = /* read pixel data from file */;
auto tex = device->createTexture2D(textureWidth, textureHeight, Format::eR8G8B8A8Unorm);
tex->upload(pixelData);
```

Listing 3.7: Creation of a color texture.

A similar *IDevice::createDepthTexture* method is also available, which instead of a color buffer creates a depth buffer. Such a uniform is used for implementing some lighting systems, such as shadow mapping.

Unlike a uniform buffer, a texture cannot be used as a uniform by itself. It needs to be associated with an *ISampler* object, which defines how the shader should read from the texture. A similar setup is used in Vulkan.

The creation of an *ISampler*, using the *IDevice::createTextureSampler2D* method, can be seen in Listing 3.8. Sampler options are kept rather simple in PapaGo allowing for magnification and minification filters to be defined, as well as texture wrap modes on the both axes. Samplers for 1 dimensional and 3 dimensional textures are also available, but the corresponding texture types have not yet been implemented.

```
auto sampler2D = device->createTextureSampler2D(
  Filter::eNearest, Filter::eNearest,
  TextureWrapMode::eClampToEdge, TextureWrapMode::eClampToEdge);
```

Listing 3.8: Creation of an *ISampler* using the nearest neighbor algorithm to determine the color when sampling.

Render Targets

Render targets represent framebuffers, which are rendered to by the graphics pipeline. A framebuffer may contain a color buffer, or a combination of a color buffer and a depth buffer.

In addition to being used as uniforms, the color and depth textures described in the previous section can be used as render targets. However, PapaGo does not support presenting these on screen directly. Therefore these textures will most often be used as off-screen buffers, to store results between render passes.

To present rendered images on screen, the developer must create a swap chain, which contains one or more framebuffers. The creation of an *ISwapChain* object is shown in Listing 3.9. In the example, the swap chain is given both the color buffer and depth buffer format of its framebuffers, in addition to the number of framebuffers to create within the chain. The present mode defines, how the images of the chain are presented. *eMailbox*, the only present mode implemented as of now, switches the front and back buffer of the swap chain at each present, and makes sure that the framebuffer presented is the most recently submitted.

```
1 auto swapChain = device->createSwapChain(
2                         Format::eR8G8B8A8Unorm,
3                         Format::eD32Sfloat,
4                         3,
5                         IDevice::PresentMode::eMailbox);
```

Listing 3.9: Creation of a *ISwapChain* with three framebuffers, each containing a color and depth buffer.

Unlike in Vulkan, PapaGo keeps track of the index of the back buffer, meaning that developers never needs to explicitly access specific framebuffers within the swap chain.

3.2.5 IRenderPass

One of the requirements of PapaGo is that *The API should be stateless*. Therefore, like in Vulkan and Direct3D 12, the render state is represented by a concrete object, which makes state changes faster. We also assume that this will make it easier for developers to keep track of their render state, in comparison to APIs with implicit render states such as OpenGL.

In PapaGo, a render state is encapsulated in a *IRenderPass* object. Once such an object has been initialized, the render state it represents cannot be changed, except for shader parameter bindings. If developers want to use multiple render states, they need to construct additional *IRenderPass* objects.

The name of *IRenderPass* hints that it is used for completing render passes. Within computer graphics, a render pass describes the execution of several draw calls, which use the same render state to draw to a render target. By stringing together render passes, the same render target can be drawn to multiple times to create different effects.

Creation of an *IRenderPass* object, using the *IDevice::createRenderPass* method, can be seen in Listing 3.10. The example presents an overload of the method, which creates a render pass that can be used to draw to a framebuffer containing a color buffer and a depth buffer. Another overload is available for drawing to framebuffers, which only include a color buffer.

```
auto shaderProgram = /* Create shader program */
auto renderPass = device->createRenderPass(
                        *shaderProgram,
                        800, 600,
                        Format::R8G8B8A8,
                        Format::eD32Sfloat);
```

Listing 3.10: Creating a *IRenderPass* object, which can be used to draw to a framebuffer consisting of a color buffer and a depth buffer.

The render target dimensions and format used to construct the render pass, must match with the render targets it will be used to render to. In addition to framebuffer information, the render pass also takes an *IShaderProgram*, which describes the underlying render state. Because of this, the developer can only configure the programmable pipeline stages of the render state. The fixed function stages retain a set of default values, which cannot be altered by the developer in the current version of PapaGo. For example, the primitives rendered are always triangles. While this decreases the flexibility of the API, it allows this first version of PapaGo to stay simple, yet still being able to create common graphics applications.

Before a render pass can commence, developers must bind uniform resources, needed by the shaders, to the corresponding *IRenderPass* object.

Here one of the requirements of the API is that *Parameter binding should be intuitive*. In Vulkan, developers can bind parameters through both push constants and descriptor sets. In the case of descriptor sets, the developer must make sure that they are compatible with the layout of the pipeline state object, and they need to allocate memory for the bindings on the GPU using descriptor pool objects. Writing in GLSL for Vulkan, every uniform needs a binding location or an offset from such a location, forcing developers to remember locations and calculate offsets to bind parameters.

In PapaGo, bindings reside on the *IRenderPass* object, so developers need not worry about descriptor sets or allocating space for them. In addition, bindings are not made to locations in the shaders, but the names instead. This feature is in part inspired by OpenGL, where the developer can ask for the location of a uniform by name.

Listing 3.11 shows how to bind resources to a render pass, when the underlying render state includes the vertex shader in Listing 3.12 and the fragment shader in Listing 3.13. Note that when writing GLSL for use with Vulkan, uniform buffers need to be defined within named shader interface blocks, as can be seen in Listing 3.12. At the moment, PapaGo only supports interface blocks containing a single element, otherwise the developer must do some padding of the uniform buffer themselves.

```
auto textureResource = /* Create texture */;
auto sampler2D = /* Create 2D sampler */;
renderPass->bindResource("sam", *textureResource, *sampler2D);

auto uniformBufferResource = /* Create buffer with view/projection matrix */;
renderPass->bindResource("view_projection_matrix", *uniformBufferResource);

auto dynamicUniformBufferResource = /* Create buffer with multiple mode matrices */;
renderPass->bindResource("model_matrix", *dynamicUniformBufferResource);
```

Listing 3.11: Different types of objects being bound to names in an *IRenderPass* object.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 pos;
layout(location = 1) in vec2 uv;

layout(binding = 0) uniform UniformBuffer {
  mat4 view_projection_matrix;
} uniforms;

layout(binding = 1) uniform InstanceUniformBuffer {
  mat4 model_matrix;
} instance;

layout(location = 0) out vec2 out_uv;

void main() {
  gl_Position = uniforms.view_projection_matrix * instance.model_matrix * vec4(pos, 1);
  out_uv = uv;
}
```

Listing 3.12: Vertex buffer that the program binds to in Listing 3.11

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 uv;

layout(binding = 2) uniform sampler2D sam;

layout(location = 0) out vec4 col;
void main() {
  col = texture(sam, uv);
}
```

Listing 3.13: Fragment shader that the program binds to in Listing 3.11

3.2.6 Command Construction and Submission

The last sections have covered the creation of different resources as well as render passes, which encapsulate state. What ties all these concepts together is the actual process of command constructions and submission, which enacts the rendering of images.

Overall, PapaGo takes inspiration from Vulkan in terms of handling commands, as this was a feature, we ourselves liked using in that API. By rendering with recorded commands instead of API function calls, as in OpenGL, GPU instructions can easily be reused between frames, which speeds up each iteration of the render loop. In addition, PapaGo allows for some commands to be recorded in parallel, like in Vulkan, which fits with the requirement *The API should be able to speed up CPU-bound applications*.

Like in Vulkan, the main objects of commands handling are primary and secondary command buffers, in addition to a GPU queue to execute them on. In PapaGo these are referred to as *ICommandBuffer*, *ISubCommandBuffer* and *IGraphicsQueue* classes, respectively. Their creation is depicted in Listing 3.14, where the queue needs to know the swap chain, which contents should be presented from the queue. As opposed to Vulkan, developers

need not explicitly allocate memory on the GPU for the command buffers by the use of command pools. Neither is the queue split into a graphics queue and a present queue, which the developer needs to synchronize between. Like in Direct3D 12, developers using PapaGo need only worry about a single type of queue.

```
auto commandBuffer = device->createCommandBuffer();
auto subCommandBuffer = device->createSubCommandBuffer();
auto graphicsQueue = device->createGraphicsQueue(*swapChain);
```

Listing 3.14: Creation of the objects for command submission

In Vulkan, primary command buffers and secondary command buffers can be used to execute the same commands. Primary command buffers can be executed directly on the queue, while secondary command buffers need to be executed through a primary buffer. While the buffers are similar, only sub command buffers can be made thread safe and recorded to in parallel.

We found this setup to be un-intuitive, as the purpose of the two kinds of command buffers overlap a great deal. Therefore, in PapaGo, the only purpose of the primary buffer type, *ICommandBuffer*, is to record commands for clearing color and depth buffers, as well as the contents of *ISubCommandBuffer* objects. This secondary type of command buffer instead has the purpose of setting up and recording draw calls. This can be seen in Listing 3.15, where the sub command buffer is recorded to in line 1 – 6 with commands, which set the vertex and index buffer, set the index of an element of a dynamic buffer, and then performs a draw call. After the sub command is recorded, its contents are recorded to the primary buffer in line 8 – 12, which also clears the color and depth buffer of the render target. Note that both buffer types need to know the render pass used for recording. However only the primary buffer needs to know the render target. In this case the render target is a swap chain, but overloads exist for *ICommandBuffer::record*, which allow for individual color and depth buffers to be used as targets.

```
subCommandBuffer->record(*renderPass, [&](IRecordingSubCommandBuffer& subRec) {
    subRec.setVertexBuffer(*vertexBuffer);
    subRec.setIndexBuffer(*indexBuffer);
    subRec.setDynamicIndex("model_matrix", 0);
    subRec.drawIndexed(36);
});

commandBuffer->record(*renderPass, *swapChain, [&](IRecordingCommandBuffer& recCommand) {
    recCommand.clearColorBuffer(0.0f, 0.0f, 0.0f, 1.0f);
    recCommand.clearDepthBuffer(1.0f);
    recCommand.execute({ *subCommandBuffer});
});
```

Listing 3.15: Using command buffers and sub command buffers.

Note that in the case of both buffers, the developers records to them using a lambda expression, which takes a corresponding recording object as input. This is done in response to Vulkan, where the developer needs to open a command buffer before recording to it, and then closing it before submitting it for execution. This allows the command buffer to be in an incomplete state, which can lead to errors performed by the developer. By using lambda expressions, PapaGo is able to handle the opening and closing of the buffers, allowing the developer to only focus on the commands.

Once a primary command buffer has been recorded to, it can be submitted for execution on a graphics queue, which will render an image to the given render target. After drawing, the swap chain bound to the queue may then be presented on screen. This can be seen in Listing 3.16. Note that when a present occurs, the swap chain automatically swaps around its framebuffers.

```
graphicsQueue->submitCommands({ *commandBuffer });
graphicsQueue->present();
```

Listing 3.16: Submitting command buffer for execution on GPU, and presenting the next image of the swap chain.

The examples just depicted, show how to record to sub command buffers in a sequential manner. As with the Vulkan equivalent, the *ISubCommandBuffer* object is thread safe, and multiple instances of this class can be recorded to in parallel. Therefore PapaGo can be used for parallel command recording, as depicted in Listing 3.17. For this example we have included a helper *ThreadPool*, which is part of a helper library, we include with PapaGo. This object can be used to enqueue lambda expressions, which need to be executed on the threads allocated on the pool. The act of enqueuing returns an *std::future* object, which provides a way of accessing the result of asynchronous operations and waiting for the lambda to finish executing.

The example shows two threads being used to draw 1000 objects each. First the thread pool is constructed, along with vectors containing sub command buffers and the aforementioned futures. The lambda for recording the sub command buffers takes three parameters of input: The command buffer to record to, the offset into the dynamic uniform buffer, and the number of objects to draw. Two instances of this lambda are given input and enqueued on the thread pool. The futures returned from the enqueue operation are used to wait until both lambda instances have been executed and the buffers recorded. From this point on, these command buffers can be safely executed.

By forcing developers to write the majority of their commands on sub command buffers, we assume that it becomes more intuitive to turn a sequential program into a parallel one.

```
auto threadPool = ThreadPool(2);
auto subCommandBuffers = std::vector<ISubCommandBuffer>(2);
auto subCommandBufferFutures = std::vector<std::future>();

auto func = [&] (int bufferIndex, int offset, int count) {
     subCommandBuffers[bufferIndex]->record(*renderPass,
     [&](IRecordingSubCommandBuffer& subRec) {
      subRec.setVertexBuffer(*vertexBuffer);
      subRec.setIndexBuffer(*indexBuffer);
      for(auto i = offset; i < offset + count; ++i){
          subRec.setDynamicIndex("model_matrix", i);
          subRec.drawIndexed(36);
    }
  });
}
subCommandBufferFutures.push_back(tp.enqueue(func, 0, 0, 1000));
subCommandBufferFutures.push_back(tp.enqueue(func, 1, 1000, 1000));

for(auto& future : subCommandBufferFutures){
    future.wait();
}
```

Listing 3.17: Constructing sub command buffers in parallel with helper ThreadPool.

Chapter 4

Implementation

This chapter will describe the implementation of a few selected aspects of PapaGo. These have been chosen, as they demonstrate, how we have solved some of the biggest challenges with simplifying the use of Vulkan.

The implementation was written in Microsoft Visual Studio C++14, and here we present code making up version 0.0 of PapaGo, which can be found in [11]. This is the version of the API, which was also used for our user evaluations, as will be presented in Chapter 5. To get the API ready for user evaluations, the implementation was not cleaned up completely, with some dead code appearing in the code base. In the interest of authenticity, we have however decided to present code as is, with some updated formatting and added comments. To make the code readable, we have kept to a rather strict naming standard. Thus all field variables on objects start with *m_* and field variables, which are of Vulkan types, have their name beginning with *m_vk*.

4.1 Interfaces and Dynamic Link Library

To enhance usability of PapaGo, we wanted to give developers a clean interface to code up against. However as we began implementing the API, a lot of time was spent on controlling how developers could interact with API objects. Initially methods and variables, which needed to be inaccessible, were set as being private, but this caused problems, when objects in the implementation needed to use the fields of each other. A temporary fix to this was to make the objects friends, but with time this resulted in many objects being strongly coupled together.

Responding to this problem, we opted to create a layer of indirection on top of the implementation in the form of interface classes. These classes function as interfaces in languages like C# and Java, but they are implemented as regular abstract classes in our C++ implementation, as the language lacks this feature. Each class in the underlying implementation inherits from an abstract interface class, which can be interacted with by the developer. *RenderPass* inherits from *IRenderPass*, *Device* from *IDevice* and so on. By restricting developer access to these classes, we do not have to worry about hiding away details further in the implementation.

The PapaGo project is configured to build to a Dynamic Link Library(DLL), which

developers must include in their projects to gain access to the aforementioned interface classes. This library includes both the headers and implementation of PapaGo, as well as Vulkan headers used by our implementation. In this manner, the developers need not include Vulkan headers in their project, and gives us the option in the future to build the project, so that it can be used with another backend like Direct3D 12.

4.2 Memory Management

One of our design requirements for PapaGo was that resource handling should be explicit, yet intuitive. In our design we have focused mostly on making it easier to use resources used by shaders. However, when using Vulkan we experienced issues in handling all API objects. This is because, using the regular Vulkan header, there is no option for automatic memory management of Vulkan objects. This puts the burden of allocating and freeing memory on the developer and increases the risk of memory leaks.

Memory management is an issue that is faced in most programming domains. Most languages have their own set of features for handling resources, whether this is by garbage collecting in languages such as C# and Java, or manually handling the allocating and freeing of memory in C and C++ (before C++11). C++11 introduced another way to handle resources with stack allocated handles which points to the actual data via a pointer. This handle then automatically deletes its data when it is deconstructed. The advice given by Stroustrup [52], the creator of C++, is to avoid raw pointers and instead use stack allocated handles such as *unique_ptr* to manage the life time of resources.

We are aware that Vulkan comes with a similar kind of resource management in its C++ bindings package: Vulkan-Hpp. This package contains interfaces for automatic resource management in the form of the class *UniqueHandler*. These handlers are recognized by their prefix; "Unique". Similar to *unique_ptr*, they can not be copied, only moved, and they clean up all used resources when destroyed.

In PapaGo, we have implemented a similar feature, using the *unique_ptr* as recommended to contain pointers to public interfaces. Using standard classes like this communicates to experienced C++11 programmers, how this resource is managed.

Figure 4.1 shows how the *IBufferResource*, as an example, is wrapped in a *unique_ptr*. The underlying implementation of *BufferResource* also includes *unique_ptr* wrappers around the Vulkan objects, which it contains. Because of this setup, if *IBufferResource* goes out of scope, it will be destroyed along with the Vulkan objects contained within it.

4.3 Resource Handling

In order to make the handling of shader resource more explicit, yet intuitive, we needed to create our own resource classes. These hide many of the details of allocating and uploading to the different types of shader resources in Vulkan.

All resources inherit from the abstract *Resource* class, within the implementation, but their actual type will either be a *BufferResource* or an *ImageResource*. Vertex, index and uniform buffers are *BufferResource* objects, while all types of textures are *ImageResource* objects.

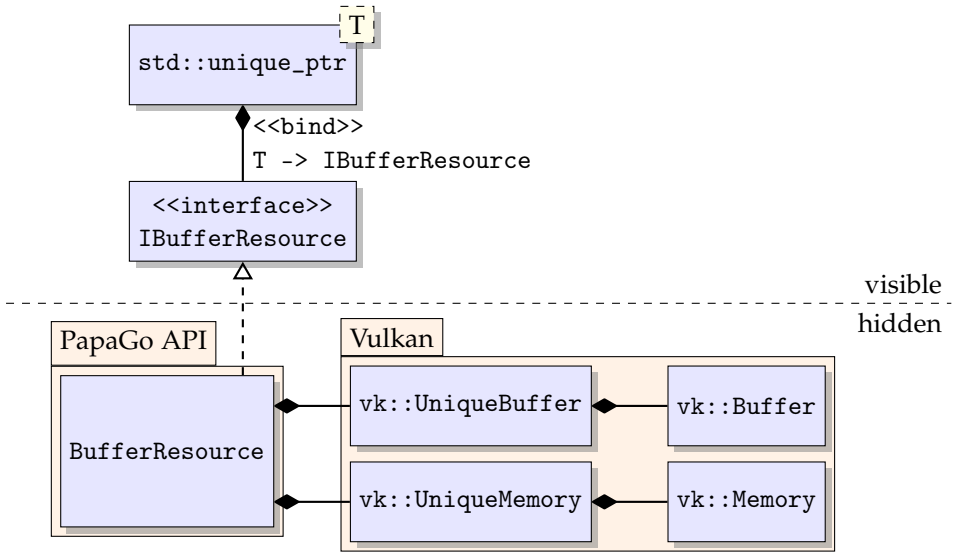

Figure 4.1: Class diagram showing an example of the relationship of resources in the PapaGo API and how they connect into Vulkan. Note that anything below the dashed line is hidden from the client when using the API.

```
Resource::Resource(
  const vk::PhysicalDevice& physicalDevice,
  const vk::UniqueDevice& device,
  vk::MemoryPropertyFlags flags,
  vk::MemoryRequirements memoryRequirements)
    : m_vkDevice(device)
    , m_size(memoryRequirements.size)
{
  // get appropriate memory type index on device
  auto memoryType = findMemoryType(physicalDevice, memoryRequirements.memoryTypeBits, flags);

  // allocate GPU memory
  m_vkMemory = device->allocateMemoryUnique(
    vk::MemoryAllocateInfo()
      .setAllocationSize(memoryRequirements.size)
      .setMemoryTypeIndex(memoryType)
  );
}
```

Listing 4.1: Implementation of the *Resource* constructor.

When one of the two aforementioned classes is instantiated, the constructor of their parent class *Resource* is called. The implementation of this can be seen in Listing 4.1. The primary function of this method is to allocate memory for the resource on the GPU.

The constructor takes four parameters, which includes both a physical and a logical Vulkan device object, which PapaGo's *Device* object wraps around. The third parameter, *flags*, is a bitmask, where each bit indicates a property of the memory that needs to be allocated. This could for instance be, whether the memory should be visible from the CPU or not. The forth and final parameter, *memoryRequirements*, is a struct, which indicates requirements for the memory being allocated. This includes the size of memory to allocate, allignment between buffer elements and a bitmask indicating the type of memory

supported. The *findMemoryType* function called returns the index of the first memory type supported by the device, which lives up to the flags given as input to the constructor. The memory type index is then used to allocate memory on the GPU of a certain size.

In the following, we describe the implementation, which is used to construct *BufferResource* and *ImageResource* objects.

4.3.1 BufferResource

Most objects in PapaGo are created through an instance of the *Device* class, including multiple different *BufferResource* objects. As their handling is very similar, we here present a single example of how the creation of a uniform buffer object occurs.

Listing 4.2, shows the implementation of *Device::createUniformBuffer*, which takes the size in bytes needed for the uniform buffer as input.

```
std::unique_ptr<IBufferResource> Device::createUniformBuffer(size_t size)
{
  return BufferResource::createBufferResource(
    m_vkPhysicalDevice, m_vkDevice,
    size,
    vk::BufferUsageFlagBits::eUniformBuffer,
    vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent);
}
```

Listing 4.2: Implementation of the *Device::createUniformBuffer* method.

This then enacts a call to the static *BufferResource::createBufferResource* method. The input given indicates that the type of Vulkan buffer to create is a uniform buffer. In addition the buffer should be visible to the CPU with a memory map on memory local to the CPU, which is to be kept coherent with memory on the GPU.

```
std::unique_ptr<BufferResource> BufferResource::createBufferResource(
  vk::PhysicalDevice physicalDevice, const vk::UniqueDevice& device,
  size_t size,
  vk::BufferUsageFlags usageFlags, vk::MemoryPropertyFlags memoryFlags,
  BufferResourceElementType type /* defaults to BufferResourceElementType::eChar*/)
{
    // Create buffer
  auto bufferCreateInfo = vk::BufferCreateInfo()
    .setSize(size)
    .setUsage(usageFlags | vk::BufferUsageFlagBits::eTransferDst);
  auto vkBuffer = device->createBufferUnique(bufferCreateInfo);

    // Extract memory requirements
  auto memoryRequirements = device->getBufferMemoryRequirements(*vkBuffer);

    // Create BufferResource
  return std::make_unique<BufferResource>(device, physicalDevice,
                                          std::move(vkBuffer),
                                          memoryFlags, memoryRequirements,
                                          size);
}
```

Listing 4.3: Implementation of the static *BufferResource::createBufferResource* method.

The implementation of the method called can be seen in Listing 4.3. This creates a Vulkan buffer, which can have data transferred to it from the CPU. Without this flag set,

it may be faster for the GPU to read the data, but it would not be possible to upload data to it directly from host. Memory requirements are then extracted, which contains information on what kind of memory is needed to support the buffer. The buffer in itself has no storage, but can be seen as a view into memory.

With the buffer created, the *BufferResource* contructor is called, which can be seen in Listing 4.4. This first calls the constructor of the parent *Resource* class, as already shown in Listing 4.1.

After the call to parent, the *m_vkInfo* field variable of type *vkDescriptorBufferInfo* is set. This struct is used later to bind the resource as a paremeter to a Vulkan descriptor set. As the buffer only contains one element, the offset is 0 and the range is the size of the buffer.

Having set this field up, the buffer is finally bound to memory, so that it has storage to be uploaded to. In this case every buffer has its own small bit of memory, however it could have been more performant to have several buffers share the same piece of memory.

```
BufferResource::BufferResource(
  const vk::UniqueDevice& device, const vk::PhysicalDevice& physicalDevice,
  vk::UniqueBuffer&& buffer,
  vk::MemoryPropertyFlags memoryFlags, vk::MemoryRequirements memoryRequirements,
  size_t  range,
  const BufferResourceElementType type /* defaults to BufferResourceElementType::eChar*/)
    : Resource(physicalDevice, device, memoryFlags, memoryRequirements)
    , m_vkBuffer(std::move(buffer)), m_elementType(type)
{
  // Configure vkDescriptorBufferInfo object
  m_vkInfo.setBuffer(*m_vkBuffer)
    .setOffset(0)
    .setRange(range);

  // Bind buffer to storage
  device->bindBufferMemory(*m_vkBuffer, *m_vkMemory, 0);
}
```

Listing 4.4: Implementation of the *BufferResource* constructor.

With the *BufferResource* created, developers can upload to it before using it. This is done through the templated *IBufferResource::upload* method, which can be seen in Listing 4.5. The input to the method is a vector of templated elements, which are cast to raw bytes within the method. The vector of chars is then sent onto *BufferResource::internalUpload* as seen in Listing 4.6.

```
void IBufferResource::upload(const std::vector<T>& data) {
  std::vector<char> buffer(sizeof(T)*data.size());
  memcpy(buffer.data(), data.data(), buffer.size());
  internalUpload(buffer);
}
```

Listing 4.5: Method supporting upload to buffers.

```
void BufferResource::internalUpload(const std::vector<char>& data) {
  auto mappedMemory = m_vkDevice->mapMemory(*m_vkMemory, 0, data.size());
  memcpy(mappedMemory, data.data(), data.size());
  m_vkDevice->unmapMemory(*m_vkMemory);
}
```

Listing 4.6: Methods supporting upload to buffers.

This shows how memory local to the CPU is mapped to the buffer's memory on the GPU. The input is then transferred to this map, which is then flushed to the GPU. At this point the data has been uploaded, and it is now available to the GPU. Data within a buffer can also be downloaded, which has a similar implementation to upload.

4.3.2 ImageResource

Image resources are similar to buffer resources in their implementation, but requires a few extra steps. While buffer resources can be read from GPU by a host-visible staging buffer, an image needs to reside in a non-host-visible device buffer. This means that image data needs to uploaded to a staging buffer using a *memcpy* and then transferred to a device buffer using a Vulkan command buffer executed on a queue.

A further difference between buffer resources and image resources is, that images needs to be transitioned between different layouts before they can be used in various ways, e.g. an image needs to be in layout *vk::ImageLayout::ePresentSrcKHR* before it can be presented on screen, and in layout *vk::ImageLayout::eTransferSrcOptimal* before it can be downloaded.

```
1  std::unique_ptr<IImageResource> Device::createTexture2D(
2      size_t width,
3      size_t height,
4      Format format)
5  {
6    vk::Extent3D extent = { uint32_t(width), uint32_t(height), 1 };
7    vk::ImageCreateInfo info;
8    info.setImageType(vk::ImageType::e2D)
9      .setExtent(extent)
10     .setFormat(to_vulkan_format(format))
11     .setInitialLayout(vk::ImageLayout::eUndefined)
12     .setMipLevels(1)
13     .setArrayLayers(1)
14     .setUsage(vk::ImageUsageFlagBits::eTransferDst |
15               vk::ImageUsageFlagBits::eSampled |
16               vk::ImageUsageFlagBits::eColorAttachment);
17
18   auto image = m_vkDevice->createImage(info);
19   auto memoryRequirements = m_vkDevice->getImageMemoryRequirements(image);
20   return std::make_unique<ImageResource>( image,
21                                           *this,
22                                           vk::ImageAspectFlagBits::eColor,
23                                           to_vulkan_format(format),
24                                           extent,
25                                           memoryRequirements);
26 }
```

Listing 4.7: Creating a 2D texture through *Device*.

Listing 4.7 shows one of the *ImageResource* creation methods on *Device*. This method creates a resource representing a 2D texture (line 8), which can be used as a color buffer in a framebuffer (last flag bit on line 16). Line 11 shows initial *vkImageLayout* the image is in after creation. This value must always be *vk::ImageLayout::eUndefined* or *vk::ImageLayout::ePreinitialized* as per the Vulkan specification.

```
1  // Allocates memory to the image and creates an image view to the provided image
2  ImageResource::ImageResource(vk::Image& image, const Device& device,
3    vk::ImageAspectFlags aspectFlags, vk::Format format,
4    vk::Extent3D extent,vk::MemoryRequirements memoryRequirements)
5    : Resource( device.m_vkPhysicalDevice, device.m_vkDevice,
6        vk::MemoryPropertyFlagBits::eDeviceLocal | vk::MemoryPropertyFlagBits::eHostVisible,
7        memoryRequirements)
8    , m_vkImage(image), m_format(format), m_vkExtent(extent)
9    , m_device(device), m_vkAspectFlags(aspectFlags)
10 {
11   m_vkDevice->bindImageMemory(m_vkImage, *m_vkMemory, 0);
12
13   createImageView(m_vkDevice, aspectFlags); // sets m_vkImageView
14
15   vk::CommandBufferBeginInfo info = {};
16   info.setFlags(vk::CommandBufferUsageFlagBits::eSimultaneousUse);
17   m_device.m_internalCommandBuffer->begin(info);
18
19   if (aspectFlags & vk::ImageAspectFlagBits::eDepth) {
20     transition<vk::ImageLayout::eUndefined, vk::ImageLayout::eGeneral>(
21         m_device.m_internalCommandBuffer);
22   }
23   else if (aspectFlags & vk::ImageAspectFlagBits::eColor) {
24     transition<vk::ImageLayout::eUndefined, vk::ImageLayout::eGeneral>(
25         m_device.m_internalCommandBuffer);
26   }
27
28   m_device.m_internalCommandBuffer->end();
29
30   vk::SubmitInfo submitInfo = {};
31   submitInfo.setCommandBufferCount(1)
32     .setPCommandBuffers(&*m_device.m_internalCommandBuffer);
33   m_device.m_vkInternalQueue.submit(submitInfo, vk::Fence());
34   m_device.m_vkInternalQueue.waitIdle();
35
36   m_device.m_internalCommandBuffer->reset(vk::CommandBufferResetFlagBits::eReleaseResources);
37
38   auto clearVector = std::vector<char>(memoryRequirements.size);
39   auto clearFloats = std::vector<float>(m_vkExtent.width * m_vkExtent.height, 1.0f);
40   auto clearStencil = std::vector<unsigned char>(memoryRequirements.size, 1);
41
42   if (aspectFlags & vk::ImageAspectFlagBits::eDepth) {
43     memcpy(clearVector.data(), clearFloats.data(), clearFloats.size() * sizeof(float));
44
45     memcpy(clearVector.data() + (clearFloats.size() * sizeof(float)),
46         clearStencil.data(),
47         clearVector.size() - clearFloats.size() * sizeof(float) - 1);
48   }
49
50   upload(clearVector);
51 }
```

Listing 4.8: The implementation of the *ImageResource* constructor

Listing 4.8 shows the *ImageResource* constructor called in Listing 4.7 at line 20. In line 13, memory and view are set up with a buffer resource. Lines 19 – 26 then transition the image from *vk::ImageLayout::eUndefined* to *vk::ImageLayout::eGeneral*.

The if-checks on lines 19 and 23 are legacy from before we saved *m_vkAspectFlags* as a

member field, and we provided this information as parameters to the *ImageResource::transition* method. Lines 38 – 50 uploads initial clear data to the memory.

4.4 GLSL Shader Analysis

To make parameter binding in PapaGo intuitive, we need to have the API analyze the shaders given to it at runtime. There are two overall types of parameters given to a shaders: vertex buffer input and uniforms. How they are handled will be described in sections below.

Originally we wanted this system of analysis to be more complex, handling all the different ways a parameter could be bound in a shader. This was scaled back because of time constraints, so we instead focused on matching binding syntax, which we have been using ourselves in our previous work. Analysis of the shader code occurs through the use of regular expressions, but initially we wanted to write our own GLSL parser to create an abstract syntax tree over shaders. This would also make it easier to do error checking, but also edit shader code, such as adding binding and input locations. However after creating a simple GLSL parser, we found this endeavor to be too time consuming, so we chose to use regular expressions instead.

4.4.1 Vertex Buffer Input Parameters

Listing 4.9 shows how input from the vertex buffer may be defined in a vertex shader. The example shows that at location 0 there is a vector of three elements, which contains a vertex position, and at location 1 there is a vector of two elements, which contains texture coordinates. If this input is read from the same vertex buffer, *pos* will be at an offset 0 from the beginning of a vertex element, while *uv* will be at an offset 12 from the beginning of an element, as *pos* has a size of 12 bytes. As mentioned in Section 3.2.4, Vulkan requires developers to themselves indicate how data from vertex buffers should be read into these parameters. However, by having PapaGo analyze shader code it becomes possible to set this up automatically, as long as all the data is included in the same vertex buffer.

```
1 // Vertex buffer input
2 layout(location = 0) in vec3 pos; //offset 0
3 layout(location = 1) in vec2 uv;  //offset 12
```

Listing 4.9: Example of vertex buffer input parameters.

The method *Parser::setShaderInput*, as shown in Listing 4.10, is called during the construction of an *IVertexShader* object. This is done to set the *m_input* field of the shader, which is a vector containing the format and offset of an input parameter. The format defines how data is laid out in the parameter, and it depends on the type of the input.

Line 7 in Listing 4.10 defines a regular expression, which matches a line in shader code, which contains vertex input. In line 15 – 26 this regular expression is used to create an iterator, which is used to iterate over all matches in the given code. The location of the match is then used as an index into *m_input*, which is then given an offset of 0 and the format.

Having looped over all matches, the offset is calculated for each parameter. This could not be done in the first loop, as the parameters are not necessarily defined in code in the same order as their location.

```
#define ITERATE(collection) std::begin(collection), std::end(collection)
#define REGEX_NUMBER "[0-9]+"
#define REGEX_NAME "[a-zA-Z_][a-zA-Z0-9_]*"

void Parser::setShaderInput(VertexShader & shader, const std::string & source)
{
  static const auto regex = std::regex(".*layout\\s*\\(location\\s*=\\s*("
                                        REGEX_NUMBER ")\\s*\\)\\s+in\\s+("
                                        REGEX_NAME ")\\s+("
                                        REGEX_NAME ");");

  std::sregex_iterator iterator(ITERATE(source), regex);
  for (auto i = iterator; i != std::sregex_iterator(); ++i) {
    auto match = *i;
    auto location = std::stoi(match[1]);
    auto format = string_type_to_format(match[2].str());
    auto name = match[3].str();

    if (shader.m_input.size() <= location) {
      shader.m_input.resize(location+1);
    }
    shader.m_input[location] = { 0, format };
  }

  // Calculate offsets. Can't do it in loop above, as allocation order could be mixed.
  auto offset = 0u;
  for (auto& input : shader.m_input) {
    input.offset = offset;
    offset += input.getFormatSize();
  }
}
```

Listing 4.10: The method which extracts location information from shaders.

4.4.2 Uniform parameters

Listing 4.11 shows some of the ways in which uniform parameters can be defined within all types of shaders.

```
// Interface block with single element
layout(binding = 0) uniform InstanceUniformBuffer {
  mat4 model_matrix; //offset 0
} modelUniform;

// Interface block with multiple elements.
// !!Not supported!!
layout(binding = 1) uniform UniformBuffer {
  mat4 view_matrix //offset 0
  mat4 projection_matrix; //offset 64
} viewProjUniform;

// Combined image-sampler binding
layout(binding = 2) uniform sampler2D sam;
```

Listing 4.11: Different types of shader parameters bindings.

Lines 1 – 4 shows an interface block, which an *IUniformBufferResource* or *IDynamicUniformBufferResource* in PapaGo can be mapped to. It lies at binding location 0 and its element *model_matrix* resides at an offset of 0 bytes from the binding location.

Lines 6 – 11 show an interface block at binding location 1, which has two elements. If the uniform buffer used as input is tightly packed, that is it has no padding as is otherwise required in Vulkan, then the first element has an offset of 0 bytes, while the second has an offset of 64 bytes. This is because the first element is a *mat4* with 16 floats. If the buffer is not packed, but uses padding as required by Vulkan for reasons of performance, then the offsets are determined by the hardware. From our experience the hardware states that each element should be made up of blocks of 256 bytes, meaning that *projection_matrix* would have an offset of 256. While we only have support for interface blocks with a single element in PapaGo, we do analyze each element of the blocks and calculate an offset in preparation for an extension. However, at the moment the offset calculated expects uniform buffers to be packed, rather than padded as Vulkan requires. This has however not led to any issues, as these offsets are not used yet.

Finally lines 13 – 14 show a free uniform outside of an interface block. Regular uniforms, which get data from uniform buffers, need to be in interface blocks, but not combined image-samplers, which represent a texture. This particular sampler uniform resides at binding location 2.

Listing 4.12 shows the method *Parser::setShaderUniforms*, which is called during creation of an *IShader* object. It identifies uniform declarations within the given shader and stores them in a map *m_bindings* on the *IShader* object. This maps names of uniforms to a struct containing binding number, offset and type of a given uniform. Lines 3 – 33 are used to match uniforms defined within an interface block and lines 36 – 54 match free uniforms. The usage of regular expressions and how matches are iterated is very similar to Listing 4.10.

Extracting data from interface blocks, first the interface blocks are matched with one regular expression, and then each uniform element within the block is matched and given an entry in *m_bindings*. Unlike in Listing 4.10, the offset of each element can be calculated in the first loop, as their sequence in the block defines their exact placement in the underlying uniform buffer.

Extracting data from free uniforms is done in a similar way, however a check is made to see what the type of the uniform is. We now realize that this check is not needed as uniform buffer types are not allowed to reside freely, when writing GLSL for Vulkan. They need to be confined within interface blocks.

4.5 Render Pass

In PapaGo, the *RenderPass* class encompasses the functionality of Vulkan's pipeline state object, render pass and descriptor set, and therefore contains instances of these objects. This section will discuss how a render pass creation is implemented as well as how parameter bindings to the object occur.

```
void Parser::setShaderUniforms(Shader & shader, const std::string & source){
    /* MATCHING WITH INTERFACE BLOCK */
    static const auto blockRegex = std::regex(  "layout\\s*\\(binding\\s*=\\s*("
                REGEX_NUMBER ")\\s*\\)\\s*uniform\\s+"
                REGEX_NAME "\\s*\\{([^\\}]*)\\}\\s*" REGEX_NAME "\\s*;");

    // Iterating over matches
    auto iterator = std::sregex_iterator(ITERATE(source),blockRegex);
    for(; iterator != std::sregex_iterator(); ++iterator) {
        auto match = *iterator;
        uint32_t binding = std::stoi(match[1].str());
        auto body = match[2].str();

        auto offset = 0u;
        static const auto body_regex = std::regex("(" REGEX_NAME ")\\s+(" REGEX_NAME ");");

        auto body_iterator = std::sregex_iterator(ITERATE(body), body_regex);
        for (body_iterator; body_iterator != std::sregex_iterator(); ++body_iterator) {
          auto body_match = *body_iterator;
          auto type = body_match[1];
          auto name = body_match[2];

          vk::DescriptorType descriptorType;
          auto typeByteSize = 0u;

          descriptorType = vk::DescriptorType::eUniformBuffer;
          typeByteSize = string_type_to_size(type);

            // Create (name, parameter) map
          shader.m_bindings.insert({ name, { binding, offset, descriptorType } });
          offset += typeByteSize;
        }
    }

    /* MATCHING WITH FREE UNIFORMS */
    static const auto regex = std::regex("layout\\s*\\(binding\\s*=\\s*("
                REGEX_NUMBER ")\\s*\\)\\s*uniform\\s+("
                REGEX_NAME ")\\s+(" REGEX_NAME ")\\s*;");

    // Iterating over matches
    auto iterator = std::sregex_iterator(ITERATE(source), regex);
    for (; iterator != std::sregex_iterator(); ++iterator) {
        auto match = *iterator;
        uint32_t binding = std::stoi(match[1]);
        auto type = match[2];
        auto name = match[3];

        vk::DescriptorType descriptorType = type == std::string("sampler2D")
          ? vk::DescriptorType::eCombinedImageSampler
          : vk::DescriptorType::eUniformBuffer;

        // Create (name, parameter) map
        shader.m_bindings.insert({ name,{ binding, 0, descriptorType } });
    }
}
```

Listing 4.12: The method extracting uniform information from the shader. *REGEX_NAME* and *REGEX_NUMBER* are the same as in Listing 4.10.

4.5.1 Pipeline creation and Caching

To create a *RenderPass* the developer calls *IDevice::createRenderPass*. This can be seen in Listing 4.13, which presents the overload of the method expecting to use a depth buffer. While the parameter name indicates a combined depth/stencil buffer, the stencil part has not been properly implemented yet in PapaGo. The method shows how a call to *Device::createVkRenderpass* creates a basic vulkan render pass object, which is then sent along to the constructor of our *RenderPass* class along with the other parameters.

```
std::unique_ptr<IRenderPass> Device::createRenderPass(
    IShaderProgram & program,
    uint32_t width, uint32_t height,
    Format colorFormat,
    Format depthStencilFormat)
{
  auto& innerProgram = dynamic_cast<ShaderProgram&>(program);
  auto renderPasses = std::vector<vk::UniqueRenderPass>();
  auto vkPass = createVkRenderpass(to_vulkan_format(colorFormat), to_vulkan_format(
    ↪ depthStencilFormat));
  return std::make_unique<RenderPass>(
    m_vkDevice,
    vkPass,
    static_cast<ShaderProgram&>(program),
    vk::Extent2D{ width, height },
    GetDepthStencilFlags(to_vulkan_format(depthStencilFormat)));
}
```

Listing 4.13: Method creating render pass.

Listing 4.14 shows the constructor called. Only a single method is called, *cacheNewPipeline*, using a bitmask as a parameter. When building a *RenderPass* object, we are not sure if the uniform buffers in the shaders given are regular uniform buffers or dynamic uniform buffers. The problem is that this needs to be decided in order to create the underlying pipeline state object.

```
RenderPass::RenderPass(
  const vk::UniqueDevice& device,
  vk::UniqueRenderPass& vkRenderPass,
  const ShaderProgram& program,
  const vk::Extent2D& extent,
  DepthStencilFlags depthStencilFlags)
  : m_shaderProgram(program)
  , m_vkDevice(device)
  , m_vkRenderPass(std::move(vkRenderPass))
  , m_depthStencilFlags(depthStencilFlags)
  , m_vkExtent(extent)
{
  cacheNewPipeline(0x00);
}
```

Listing 4.14: The *RenderPass* constructor.

To let developers bind to parameters as they please, we have decided to create a caching system. Thus each time shader parameters need to be bound, a new pipeline state object is created and cached for use, if there is not already a compatible pipeline in the cache.

The mask given to *RenderPass::cacheNewPipeline* indicates one way of binding to the uniform buffers. In the mask, a bit at position x indicates the status of the uniform at

binding x. If the bit is 0, then a non-dynamic uniform buffer is bound, and 1 if a dynamic buffer is bound.

When the *RenderPass* is created, the first pipeline cached assumes all uniform buffers bound to be non-dynamic.

Listing 4.15 shows the caching method. In line 3, we access the bindings on a *ShaderProgram* object, created as described in Section 4.4.

```
1  void RenderPass::cacheNewPipeline(uint64_t bindingMask)
2  {
3    auto bindings = m_shaderProgram.getUniqueUniformBindings();
4
5    setupDescriptorSet( m_vkDevice,
6                        m_shaderProgram.m_vertexShader,
7                        m_shaderProgram.m_fragmentShader,
8                        bindingMask);
9
10     // Defined programmable stages of pipeline
11   vk::PipelineShaderStageCreateInfo shaderStages[] = {
12     m_shaderProgram.m_vkVertexStageCreateInfo,
13     m_shaderProgram.m_vkFragmentStageCreateInfo
14   };
15
16
17   vk::PipelineVertexInputStateCreateInfo vertexInputInfo;
18   auto attributeDescription = getAttributeDescriptions();
19   vk::VertexInputBindingDescription bindingDescription;
20
21   if (!m_shaderProgram.m_vertexShader.m_input.empty()) {
22     bindingDescription = getBindingDescription();
23
24     vertexInputInfo.setVertexBindingDescriptionCount(1)
25       .setPVertexBindingDescriptions(&bindingDescription)
26       .setVertexAttributeDescriptionCount(attributeDescription.size())
27       .setPVertexAttributeDescriptions(attributeDescription.data());
28   }
29   else {
30     vertexInputInfo.setVertexBindingDescriptionCount(0)
31       .setPVertexBindingDescriptions(nullptr)
32       .setVertexAttributeDescriptionCount(0)
33       .setPVertexAttributeDescriptions(nullptr);
34   }
35
36   /* Omitted; creating objects for setting fixed function state */
37     /*...*/
```

Listing 4.15: Cache new pipeline part 1.

In Line 5, there is a call to *RenderPass::setupDescriptorSet*, which uses the binding mask to define a descriptor set, descriptor pool and descriptor set layout. This will be discussed in depth in Section 4.5.3.

Lines 11 – 14 shows how the programmable pipeline stages, which the pipeline state object is built around, are defined.

Lines 17 – 34 create a *vk::PipelineVertexInputStateCreateInfo* object, which defines how to read attributes from vertex buffers. If the vertex shader requires vertex input, then we use the functions *getAttributeDescriptions* and *getBindingDescription* to define the above object. These will be described in detail in Section 4.5.2. If the shader takes no vertex input, an

empty object is created.

In Listing 4.16 lines 3 – 9 creates the pipeline layout, which defines how uniform parameters can be bound to the pipeline. This is defined from the single descriptor set layout, which was setup in *setupDescriptorSet*. The layout is cached along, which is also the case with the pipeline state object, which is created in lines 12 – 27.

From this point on, the *RenderPass* object has been constructed and can be used to record commands and bind uniform parameters.

```
1     /*...*/
2     // Create Pipeline layout from descriptor set layout
3   vk::PipelineLayoutCreateInfo pipelineLayoutInfo;
4   if (m_vkDescriptorSetLayouts[bindingMask]) {
5     pipelineLayoutInfo.setSetLayoutCount(1)
6       .setPSetLayouts(&m_vkDescriptorSetLayouts[bindingMask].get());
7   }
8   m_vkPipelineLayouts[bindingMask] = m_vkDevice->createPipelineLayoutUnique(
9       pipelineLayoutInfo);
10
11    // Create vulkan pipeline and cache it
12    vk::GraphicsPipelineCreateInfo pipelineCreateInfo = {};
13    pipelineCreateInfo.setStageCount(2)
14      .setPStages(shaderStages)
15      .setPVertexInputState(&vertexInputInfo)
16      .setPInputAssemblyState(&inputAssembly)
17      .setPViewportState(&viewportState)
18      .setPRasterizationState(&rasterizer)
19      .setPColorBlendState(&colorBlending)
20      .setRenderPass(m_vkRenderPass.get())
21      .setLayout(m_vkPipelineLayouts[bindingMask].get())
22      .setPMultisampleState(&multisampleCreateInfo)
23      .setPDepthStencilState(depthCreateInfo);
24
25  m_vkGraphicsPipelines[bindingMask] = m_vkDevice->createGraphicsPipelineUnique(
26      vk::PipelineCache(),
27      pipelineCreateInfo);
28 }
```

Listing 4.16: Cache new pipeline part 2.

4.5.2 Bind Vertex Data

In Section 4.4, we show how bindings for a vertex buffer were calculated based on a vertex shader. These were used in the creation of a pipeline in Listing 4.15, to define how a vertex buffer used in conjunction with the pipeline should be read from. This requires a *vk::VertexInputAttributeDescription* object to be made for each attribute in the vertex buffer. This is done using the *getAttributeDescriptions* function, which is shown in Listing 4.17. This shows how the location, format and offset, which were calculated previously, are used to create the descriptor objects.

Listing 4.18 then shows how to create the *getBindingDescription*. This object stores information about the bind location, which is set to 0 here, we only support one vertex buffer to be bound at one time. The stride, equal to the size of each vertex element, is also stored in the struct, along with the input rate.

Using these structs to create a pipeline state object in Listing 4.15, any vertex buffer used with the created *RenderPass* object must live up to the defined structure.

```
std::vector<vk::VertexInputAttributeDescription> RenderPass::getAttributeDescriptions()
{
  auto inputCount = m_shaderProgram.m_vertexShader.m_input.size();
  auto attributeDescriptions = std::vector<vk::VertexInputAttributeDescription>(inputCount);

  for (auto i = 0; i < inputCount; ++i) {
    auto input = m_shaderProgram.m_vertexShader.m_input[i];
    attributeDescriptions[i].setBinding(0)
      .setLocation(i)
      .setFormat(input.format)
      .setOffset(input.offset);
  }

  return attributeDescriptions;
}
```

Listing 4.17: The method getAttributeDescriptions function called in Listing 4.15.

```
vk::VertexInputBindingDescription RenderPass::getBindingDescription()
{
  auto& inputs = m_shaderProgram.m_vertexShader.m_input;
  auto& lastInput = inputs.back();
  vk::VertexInputBindingDescription bindingDescription = {};
  bindingDescription.binding = 0;
  bindingDescription.stride = lastInput.offset + lastInput.getFormatSize();
  bindingDescription.inputRate = vk::VertexInputRate::eVertex; //VK_VERTEX_INPUT_RATE_VERTEX

  return bindingDescription;
}
```

Listing 4.18: The method getBindingDescription function called in Listing 4.15.

4.5.3 Bind Uniform Parameters

When creating our pipeline, we need to set up a descriptor set, descriptor pool and descriptor set layout. This was done in Listing 4.15 by a call to *RenderPass::setupDescriptorSet*, which is defined in Listing 4.19.

Lines 5 – 19 shows how *vkDescriptorSetLayoutBindings* objects are defined for each binding in the submitted vertex shader. Note that the binding mask submitted is used to determine, if a uniform buffer object is dynamic or not. For the sake of brevity, we do not show the similar process of creating *vkDescriptorSetLayoutBindings* from bindings on the fragment shader. Lines 23 – 28 shows that if bindings exists, then a descriptor set layout is created and cached. In the lines 32 – 44 a descriptor pool is created for the descriptor set, which is created and cached in lines 40 – 53. Note that the binding system is kept simple here, with all bindings being placed on the same parameter block and supported by the same piece of memory.

```
void RenderPass::setupDescriptorSet(const vk::UniqueDevice& device,
    const VertexShader& vertexShader, const FragmentShader& fragmentShader,
    uint64_t bindingMask){
  std::vector<vk::DescriptorSetLayoutBinding> vkBindings;
  auto vertexBindings = vertexShader.getBindings();
  for (size_t i = 0; i < vertexBindings.size(); ++i) {
    auto& vertexBinding = vertexBindings[i];
    vk::DescriptorType type = vertexBinding.type;
    auto bindingValue = vertexBinding.binding;
    if (type == vk::DescriptorType::eUniformBuffer) {
      type = (bindingMask & (1 << bindingValue))
          ? vk::DescriptorType::eUniformBufferDynamic : vk::DescriptorType::eUniformBuffer;
    }
    vk::DescriptorSetLayoutBinding binding = {};
    binding.setBinding(bindingValue)
      .setDescriptorCount(1).setDescriptorType(type)
      .setStageFlags(vk::ShaderStageFlagBits::eVertex);
    vkBindings.emplace_back(binding);
  }

    /* Creating bindings from fragment shader omitted, as it is similar to above */

  if (!vkBindings.empty()) {
    vk::DescriptorSetLayoutCreateInfo layoutCreateInfo = {};
    layoutCreateInfo.setBindingCount(vkBindings.size()).setPBindings(vkBindings.data());

    // Create Descriptor Set Layout
    m_vkDescriptorSetLayouts[bindingMask] = device->createDescriptorSetLayoutUnique(
        layoutCreateInfo);

    //Create Descriptor Pool:
    auto poolSizes = std::vector<vk::DescriptorPoolSize>(vkBindings.size());

    for (auto i = 0; i < vkBindings.size(); ++i) {
      auto& vkBinding = vkBindings[i];
      poolSizes[i].setDescriptorCount(1).setType(vkBinding.descriptorType);
    }

    vk::DescriptorPoolCreateInfo poolCreateInfo = {};
    poolCreateInfo.setPoolSizeCount(poolSizes.size())
      .setPPoolSizes(poolSizes.data()).setMaxSets(1)
      .setFlags(vk::DescriptorPoolCreateFlagBits::eFreeDescriptorSet);

    m_vkDescriptorPools[bindingMask] = device->createDescriptorPoolUnique(
        poolCreateInfo);

    //Create Descriptor Set:
    vk::DescriptorSetAllocateInfo allocateInfo = {};
    allocateInfo.setDescriptorPool(*m_vkDescriptorPools[bindingMask])
      .setDescriptorSetCount(1)
      .setPSetLayouts(&m_vkDescriptorSetLayouts[bindingMask].get());

    m_vkDescriptorSets[bindingMask] = device->allocateDescriptorSetsUnique(
        allocateInfo)[0];
  }
}
```

Listing 4.19: Creating and caching objects for parameter binding.

Listing 4.20 shows how parameters are bound to a *RenderPass* object using the *RenderPass::bindResource* method, which is presented in Listing 4.20. For this example, we use the overload of the method taking a uniform buffer resource.

In line 4 the location of the uniform binding is gotten by a name given as parameter, and in line 5 the mask indicating the current configuration used by the *RenderPass* is also accessed. In line 8, the bit corresponding to the binding being set is set to 0. Using the overload with a dynamic buffer, this would be set to 1.

In lines 10 – 14 a new pipeline is cached based on the mask, if it does not already exist.

Lines 18 – 28 show the creation of a *vkWriteDescriptorSet*, which is used to update one descriptor of the current set, so that it points to the new resource.

In Lines 30 – 46, if the mask used has been updated, a vector of *vkCopyDescriptorSet* structs is created. Each of these structs correspond to one of the uniforms, which is not set to a new value, but still needs to be copied into the new descriptor set. In line 47 the new descriptor is written to the descriptor set, and the old ones are copied into the set. Finally in line 48, the alignment of the binding is set to 0, as only one element is contained within the uniform buffer resource with an offset of 0 bytes.

With this method finished, the commands executed with the corresponding *RenderPass* will use the new uniform buffer resource, which has just been set.

4.6 Synchronization

Since explicit GPU/CPU and GPU/GPU synchronization is a core feature for Vulkan, this must be handled by the implementation of PapaGo. This is done by utilizing semaphores to handle the synchronization between the two Vulkan queues, *GraphicsQueue::m_vkGraphicsQueue* and *GraphicsQueue::m_vkPresentQueue*, and by executing pipeline barrier commands to transition image resources.

4.6.1 Semaphores

PapaGo's *GraphicsQueue* class contains two Vulkan queue member fields; one for command submission *GraphicsQueue::m_vkGraphicsQueue*, and one for presentation *GraphicsQueue::m_vkPresentQueue*. While these two queue objects may represent the same physical queue, PapaGo always needs to make sure that the developer does not try to present a framebuffer, which still has commands being executed on it.

```
void RenderPass::bindResource(const std::string & name, IBufferResource &buffer)
{
  auto& innerBuffer = dynamic_cast<BufferResource&>(buffer);
  auto binding = getBinding(name);
  auto oldMask = m_descriptorSetKeyMask;

  //update mask so this binding bit is set to 0:
  m_descriptorSetKeyMask &= (~0x00 & (0x0 << binding));

  //if we have not cached a descriptor set
  if (m_vkDescriptorSets.find(m_descriptorSetKeyMask) == m_vkDescriptorSets.end())
  {
    cacheNewPipeline(m_descriptorSetKeyMask);
  }

  auto& descriptorSet = m_vkDescriptorSets[m_descriptorSetKeyMask];

  vk::DescriptorBufferInfo info = innerBuffer.m_vkInfo;
  info.setOffset(m_shaderProgram.getOffset(name));

  std::vector<vk::CopyDescriptorSet> descriptorSetCopys;

  auto writeDescriptorSet = vk::WriteDescriptorSet()
    .setDstSet(*descriptorSet)
    .setDstBinding(binding)
    .setDescriptorType(vk::DescriptorType::eUniformBuffer)
    .setDescriptorCount(1)
    .setPBufferInfo(&info);

  // only copy "old" binding information if we have just created a new descriptor set:
  if (oldMask != m_descriptorSetKeyMask) {
    for (auto& bindingAligment : m_bindingAlignment)
    {
      auto otherBinding = bindingAligment.first;
      if (otherBinding != binding) {
        auto copyDescriptorSet = vk::CopyDescriptorSet()
          .setDescriptorCount(1)
          .setDstBinding(otherBinding)
          .setDstSet(*descriptorSet)
          .setSrcBinding(otherBinding)
          .setSrcSet(*m_vkDescriptorSets[oldMask]);

        descriptorSetCopys.push_back(copyDescriptorSet);
      }
    }
  }
  m_vkDevice->updateDescriptorSets({ writeDescriptorSet }, descriptorSetCopys);
  m_bindingAlignment[binding] = 0;
}
```

Listing 4.20: Method extracting uniform information from the shader.

Listing 4.21 shows the *GraphicsQueue::present()* method. Line 3 makes sure that the queue is not in use when trying to present. Lines 5 – 22 extracts the *ImageResource* objects, which have been submitted to the *GraphicsQueue* (acting as the *m_vkGraphicsQueue*), and transitions them to *vk::ImageLayout::ePresentSrcKHR*. The method *transitionImageResources* (Lines 18 and 39) is a helper method defined in *GraphicsQueue*, which calls the appropriate transition function on all the supplied *ImageResource* objects. Lines 24 – 34 sets up the semaphore (*m_vkRenderFinishSemaphore*), adds it to the *presentInfo* object, and presents the *m_vkPresentQueue* object. Line 29 indicates, that the presentation should wait for the given semaphore to be signalled. Line 37 waits on the present queue (like line 3). This is done because lines 39 – 43 transitions the images back to *vk::ImageLayout::eGeneral*, which may only be done once the presentation is complete. Line 45 clears the set of *Resource* pointers used in lines 5 – 22. The final lines advance the inner framebuffer index of the swap chain.

4.6.2 Pipeline Barriers

Vulkan requires image resources to be in an appropriate layout before they can be used. The transition from one layout to another can be accomplished through recording a pipeline-barrier on a command buffer. Since both the layout an image transitions from and to needs to be given, PapaGo needs to keep track of the layout a given image is in at all times. To accomplish this, PapaGo makes sure that images always are in the layout *vk::ImageLayout::eGeneral*. If another layout is required by an event (e.g. the developer wants to download the image resource), then the image is transitioned from *vk::ImageLayout::eGeneral* to whatever is needed, and after the event requiring a specific layout has completed, then the image is transitioned back to *vk::ImageLayout::eGeneral*.

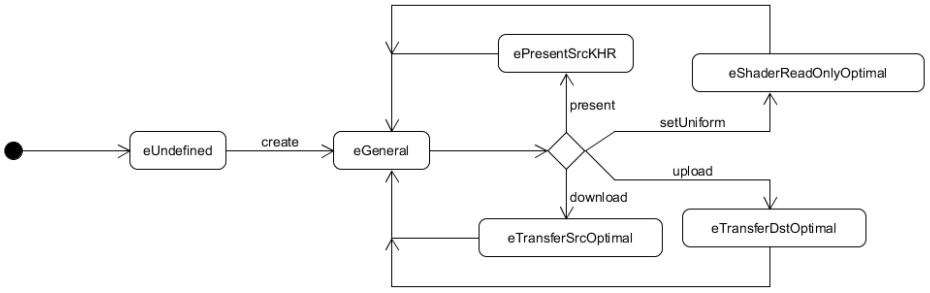

Figure 4.2: Flow of user initiated image transitions

Figure 4.2 describes which events influences the layout of an image. As can be seen, *eGeneral* is the central layout. At all events an *ImageResource* start in *eGeneral* (after the initial transition at creation), then transition to whatever is needed, and then reverts back to *eGeneral*. Apart from these transitions, there are also some implicit transitions which happens within a render pass, but the initial layout before the render pass is entered is always *eGeneral*, and the render pass always reverts the image to *eGeneral* once it is done.

```
void GraphicsQueue::present()
{
  m_vkPresentQueue.waitIdle();

  std::set<ImageResource*> imageResources;
  imageResources.emplace(
      &m_swapChain.m_colorResources[m_swapChain.m_currentFramebufferIndex]
  );

  // Find image resources from the submitted resources
  for (auto& resource : m_submittedResources) {
    if (typeid(*resource) == typeid(ImageResource)) {
      auto imageResource = reinterpret_cast<ImageResource*>(resource);
      imageResources.emplace(imageResource);
    }
  }

  transitionImageResources<vk::ImageLayout::eGeneral, vk::ImageLayout::ePresentSrcKHR>(
    m_device.m_internalCommandBuffer,
    m_device.m_vkInternalQueue,
    imageResources
  );

  std::vector<vk::Semaphore> semaphores = { *m_vkRenderFinishSemaphore };
  std::vector<vk::SwapchainKHR> swapchains = { static_cast<vk::SwapchainKHR>(m_swapChain) };

  vk::PresentInfoKHR presentInfo = {};
  presentInfo.setWaitSemaphoreCount(semaphores.size())
    .setPWaitSemaphores(semaphores.data())
    .setSwapchainCount(1)
    .setPSwapchains(swapchains.data())
    .setPImageIndices(&m_swapChain.m_currentFramebufferIndex);

  auto res = m_vkPresentQueue.presentKHR(presentInfo);

  //TODO: better way to know when ImageResources can safely be transitioned back to eGeneral
  m_vkPresentQueue.waitIdle();

  transitionImageResources<vk::ImageLayout::ePresentSrcKHR, vk::ImageLayout::eGeneral>(
    m_device.m_internalCommandBuffer,
    m_device.m_vkInternalQueue,
    imageResources
  );

  m_submittedResources.clear();

  auto nextFramebuffer = m_device.m_vkDevice->acquireNextImageKHR(
      static_cast<vk::SwapchainKHR>(m_swapChain),
      0,
      *m_vkImageAvailableSemaphore,
      vk::Fence());
  m_swapChain.m_currentFramebufferIndex = nextFramebuffer.value;
}
```

Listing 4.21: The *GraphicsQueue::present()* method.

```
1  template<>
2  inline void ImageResource::transition<vk::ImageLayout::eUndefined, vk::ImageLayout::eGeneral>(
3    const CommandBuffer& commandBuffer,
4    vk::AccessFlags srcAccessFlags,
5    vk::AccessFlags dstAccessFlags)
6  {
7    auto imageMemeoryBarrier = vk::ImageMemoryBarrier(
8      srcAccessFlags, //srcAccessMask
9      dstAccessFlags, //dstAccessMask
10     vk::ImageLayout::eUndefined, //oldLayout
11     vk::ImageLayout::eGeneral, //newLayout
12     VK_QUEUE_FAMILY_IGNORED,  //srcQueueFamilyIndex
13     VK_QUEUE_FAMILY_IGNORED,  //dstQueueFamliyIndex
14     m_vkImage, //image
15     { m_vkAspectFlags, 0, 1, 0, 1 } //subresourceRange
16   );
17
18   commandBuffer->pipelineBarrier(
19     vk::PipelineStageFlagBits::eTopOfPipe, //srcStageMask
20     vk::PipelineStageFlagBits::eBottomOfPipe, //dstStageMask
21     vk::DependencyFlags(), //dependencyFlags
22     {}, //memoryBarriers
23     {}, //bufferMemoryBarriers
24     { imageMemeoryBarrier } //imageMemoryBarriers
25   );
26 }
```

Listing 4.22: An example of a transition method.

Listing 4.22 shows an example of an image transition template specialization, which transitions from *vk::ImageLayout::eUndefined* to *vk::ImageLayout::eGeneral*. *srcAccessMask* and *dstAccessMask* are defaulted *vk::AccessFlags()*. First, an *vk::ImageMemoryBarrier* object is created (line 7), and then a command is recorded to insert a pipeline barrier (line 18). Because a pipeline barrier is used, instead of transitioning the image immediately, we can let the driver decide exactly when the transitions take place. Lines 19 – 20 describe the stages, between which the transition needs to occur.

Chapter 5

API Evaluation

In this chapter, we describe, how we evaluate the programmability of PapaGo by performing experiments, where participants need to solve programming tasks with the API. Afterwards we analyze the data gathered from participants, followed by a conclusion upon our findings. The version of PapaGo analyzed in this chapter can be found at [11].

Previously, APIs have been evaluated through the use of informal principles like readability, simplicity and reliability, which are either evaluated by the API designers themselves or an outside expert. While this is an inexpensive approach, it does not always discover all programmability problems. This is in part because designers and experts may be blind to issues experienced by users, as they use the API. Therefore we now see an increase in empiric API evaluation, where design flaws are discovered by observing how participating users use an API to complete programming tasks.

We see two types of user-based API evaluation methods in the literature: Those evaluating APIs individually and those comparing API features. In our previous work [10, 8], we used the API Walkthrough method [47] and Cognitive Dimensions method [17] to evaluate Direct3D 12 and Vulkan individually. After having made individual evaluations, we were able to compare the APIs.

Alternatively we could have used the methods described in [21] and [22]. [21] compares the usage of factories and regular object construction in API design, while [22] compares how static and dynamic typing is affected by the availability of API documentation. These studies both use a between subject approach, which means that participants are split into groups, with each group being given one of the tools tested to solve a handful of programming tasks. This is opposed to a within subject approach, where the same participant has to complete each task with every tool being tested.

We have considered using one of these approaches to compare PapaGo to Vulkan, yet as Vulkan functions at a lower level of abstraction, such a comparison would not be fair. Alternatively a comparison of OpenGL could be made, but if we use a between subject approach, we would need to double the number of participants, as well as finding participants with equal knowledge of PapaGo as OpenGL. Instead we could do a within subject study, but this risks our results being altered thanks to carryover effect.

In light of this, we decide to use a similar approach to our earlier work, focusing on evaluating PapaGo by itself. However, last time API Walkthrough was used as Direct3D 12 and Vulkan were large and complex APIs, which were expensive to test with a task-based

approach. As PapaGo is a simpler API, we choose to gather data through experiments, where we observe users, as they use the tool for solving programming tasks.

While we want to analyze our API with the Cognitive Dimensions method, the basic method of conducting experiments as presented by Clarke and Becker [17] is rather vauge. Instead, we choose to perform experiments using the discount evaluation method by Kurtev, Christensen, and Thomsen [31], as described in the following section.

5.1 Experiments

For conducting our task-based user experiments, we use the Discount method for programming language evaluation [31]. While this has been developed for evaluation of programming languages with no implemented compiler, we expect that it will also work for evaluating an implemented API. This is because, while the method is well-defined, it is also generic enough to fit many use cases.

Our main reason for picking this method is that it is rather inexpensive, while still allowing us to discover many issues within the API. In its most basic form, the method requires five participants to perform a handful of programming tasks, while being observed. It follows a nine-step procedure, which is described as follows:

1. Create Tasks.
2. Create a sample sheet.
3. Estimate the task duration.
4. Prepare setup.
5. Gather participants.
6. Start the experiment.
7. Keep the participants talking.
8. Interview the participant.
9. Analyze data.

During our execution of the method we **Create Tasks** from the use cases, which were used to design the API in Section 3.1.

Our tasks are defined as follows:

1. Draw a triangle.
2. Draw several textured cubes.
3. Draw several textured cubes with parallel command submission.
4. Draw a shaded cube using shadow mapping to cast shadow on other objects.

The first task is created to see, how the participant can get a very basic graphical program up and running. While this is not one of the original three use cases, it allows our participants to get familiar with the API by initially solving a simpler task. The second task tests some of the more advanced features of the API including resource management and parameter binding. The third task tests, if participants can understand and make use of parallel command submission. The fourth and final task tests, how participants can create a lighting system using shadow mapping, which requires several render passes to be strung together. The third and forth tasks are rather advanced, so we do not expect every participant to complete them. However the first and second tasks cover most of the APIs functionality, so this is not seen as a big issue.

As we want to examine the usage of our API exclusively, we write the shaders beforehand and ask the participants to only write host-code.

Next we **Create a sample Sheet**, which contains explanations of core concepts as well as code examples. It was originally included in the method to help participants complete tasks without getting feedback when no compiler was available. While our API can give feedback, we still think that it is too a complex to be used without supporting material. Therefore we write a sample sheet in the form of a wiki-site, found at [13]. This is made accessible to participants a few days before the experiment occurs, allowing them to get an overall idea of the API. We hope this will keep participants from "freezing" at the beginning of the experiment, as reported in the case of the the original method.

Having defined the sample sheet we **Estimate task duration** for each task. This is difficult to set as our participants vary greatly in abilities, but we ourselves are able to get up several textured cubes in under an hour. Doubling this, and adding time for looking at the more advanced tasks and interviews, we set each session to take three hours, with the expectation that not all tasks will be completed.

To **Prepare setup**, we single out one of our own laptops for performing the tests. We could allow participants to use their own computers, but this would add to setup time. Participants are made to write code in Microsoft Visual Studio. We decide not to record the experiments with video and audio, as this requires the setup of a lot of recording equipment, and we do not assume it to be necessary. Instead we opt for having one dedicated note taker.

When **Gathering participants**, we were looking for developers experienced in doing graphics programming with OpenGL or Direct3D 11, as these APIs display the level of abstraction, which we were aiming towards with PapaGo. As coding is done in C++, participants also needed a basic knowledge of this language. While the method calls for five participants, finding graphics programmers local to us proved to be difficult. Therefore we opt for using three participants in total, which are described in the following:

- Participant A is a Computer Science student doing his bachelor's degree. He has some experience with C++ and has programmed a bit in OpenGL at both school and in his spare time.
- Participant B is a full time programmer, who is very familiar with C++. He has used both Direct3D 9, 10 and 11 as well as OpenGL, and he has worked in the video game industry for a couple of years.
- Participant C is a student doing a master's degree in Vision, Graphics and Interactive Systems. He has some familiarity with C++ and has been taught OpenGL during a

computer graphics course. Later he has also used Direct3D 11, but he has focused mostly on shader programming by letting the Unity game engine handle host code.

As the participants have different levels of experience, we hope that this will lead a diverse array of problems to be discovered.

When we **Start the experiment**, we conduct a small interview with the participant to give us an exact idea of who they are and what their level of graphics programming experience is. The interview questions can be found in Appendix B.1. One member of our group acts as a main facilitator, explaining tasks to the participant, answering questions and discussing tasks solutions.

The facilitator must also **Keep the participant talking** as they code, so that notes can be taken about their thought process. Not all participants got to finish each task, in Table 5.1 we have recorded which task each participant completed, and how much time it took. No participants got to complete task 4, but we allowed participant B to look at our solution to the task, so that we could get some feedback. When participant B was writing a solution to task 2, he experienced a bug in the API, which took about 20 minutes for the facilitators to locate. This has not been counted towards his completion time in Table 5.1.

After the experiment is finished, we **Interview the participant**, asking them about their thoughts on the API. As we are interested in analyzing our data in terms of the Cognitive Dimensions method, we make sure to ask questions in terms of each of the 12 dimensions. The questions used for the interview can be found in Appendix B.2.

In the original method, they **Analyze Data** by identifying programmability problems and prioritizing them as cosmetic, serious or critical. While we also identify problems, we instead use the extended rating scale for severity found in [42], as it allows problems to be rated with a higher granularity ,and each rating is more formally defined. This will be described in the following section.

The notes taken during each session can be found in [9]

	Task 1	Task 2	Task 3	Task 4
Participant A	107	(38)	–	–
Participant B	51	54	16	(5)
Participant C	72	61	13	–

Table 5.1: Minutes taken by participants to complete each task. Note that Participant A completed task 2 partially, only rendering a single textured cube. In addition Participant B only got to look at a possible solution to Task 4 for a few minutes, which we provided.

5.2 Usability Problems

Having conducted our three sessions, we initially identify problems discovered within each test session. Here we also include problems discovered in the documentation, as we assume it to be crucial for use of the API. The problems are later combined into a total list of problems discovered across all test sessions. These problems are then rated by severity using the 0 to 4 rating scale described in [42]. Each level of the scale is defined as follows:

0. I don't agree that this is a usability problem at all

1. Cosmetic problem only: need not be fixed unless extra time is available on project
2. Minor usability problem: fixing this should be given low priority
3. Major usability problem: important to fix, so should be given high priority
4. Usability catastrophe: imperative to fix this before product can be released

Severity is rated according to three factors: The frequency of the problem, the negative impact it has on usability, and its persistence after being encountered for the first time. In the group we kept these factors in mind, as we examined each problem, which was then rated by its severity. The list of rated problems can be found in Appendix C.

To get a better overview over the severity ratings, we have made a bar chart in Figure 5.1. From this we can tell that most of the problems discovered are either cosmetic or minor, which we regard as a positive. However, we do have a small handful of major and catastrophic usability problems, which will have to be fixed. How we go about updating the API to solve these problems is described later in Section 5.4. We hope that by solving some of these major issues, some of the smaller issues will also disappear. For instance, by updating the wiki documentation to solve problem #36, we may also improve the explanation of certain API elements. This would solve issues like #2 and #5, where participants had difficulty understanding new concepts in part because of lacking documentation. Having now identified several usability problems within our API, we go about analyzing PapaGo in terms of the Cognitive Dimensions for APIs. This is described in the next section.

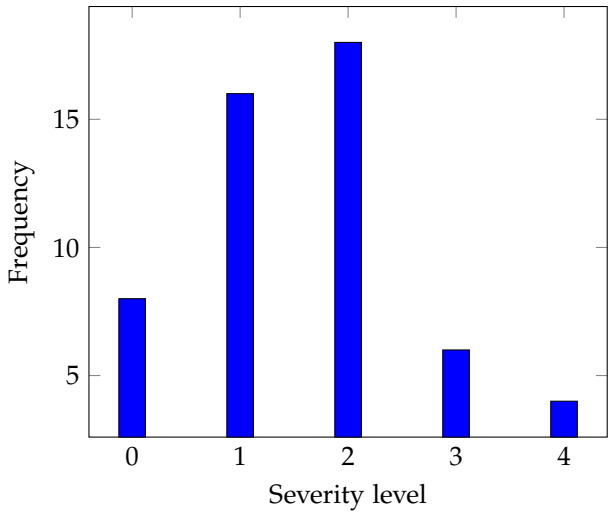

Figure 5.1: Bar chart over severity of problems presented in Appendix C

5.3 Cognitive Dimensions

The Cognitive Dimensions method, as described in [17], is used to analyze multiple aspects of an API. As opposed to discovering usability problems, this method gives a holistic view of the API including both negative and positive aspects. Using the method, the API is rated in terms of 12 dimensions, which are described in Table 5.2. The rating of each dimension is described in prose, as this is a qualitative method. We support each rating given by using the notes taken during the test sessions as evidence. By choosing this method, we ensure that we analyze and evaluate PapaGo from different angles.

In the following we go through each dimension.

Dimension	Description
Abstraction Level	What are the minimum and maximum levels of abstraction exposed by the API, and what are the minimum and maximum levels usable by a targeted developer.
Learning Style	What are the learning requirements posed by the API, and what are the learning styles available to a targeted developer.
Working Framework	What is the size of the conceptual chunk needed to work effectively.
Work-Step Unit	How much of a programming task must/can be completed in a single step.
Progressive Evaluation	To what extent can partially completed code be executed to obtain feedback on code behavior?
Premature Commitment	To what extent does a developer have to make decisions before all the needed information is available?
Penetrability	How does the API facilitate exploration, analysis, and understanding of its components, and how does a targeted developer go about retrieving what is needed?
API Elaboration	To what extent must the API be adapted to meet the needs of a targeted developer?
API Viscosity	What are the barriers to change inherent in the API, and how much effort does a targeted developer need to expend to make a change?
Consistency	Once part of an API is learned, how much of the rest of it can be inferred?
Role Expressiveness	How apparent is the relationship between each component and the program as a whole?
Domain Correspondence	How clearly do the API components map to the domain? Are there any special tricks?

Table 5.2: The 12 cognitive dimensions with descriptions. [17]

Abstraction Level

Looking at abstraction within the API, we focus on the extremes, the highest and lowest level of abstraction. Each feature here has been labelled with the participant, who brought up its level of abstraction.

- High:
  - Automatic coupling of vertex attributes and shaders (A).
  - Binding through uniform name (C).
  - Swap chain index automatically updates at present (B, C).
- Low:
  - Having API objects wrapped in unique pointers (A, C).
  - Command submission (A, C).
  - Resource handling (A, B, C).

Learning Style

Regarding learning requirements, the participants had to learn the following new concepts to use the API. We have labeled each concept with the participant, who needed to learn it.

- Swap Chain (A, C).
- Command Submission (A, B, C).
- Render Pass (A, B).
- Graphics Queue (A, B, C).
- Dynamic Uniform Buffers (A, B, C).

In terms of learning styles, we expect the API to be used in a top down manner and provided documentation through a wiki. The non-professional participants A and C used this resource, learning in a top-down manner, however participant B was able to work in a bottom-up manner, exploring the API on his own through experiments.

Working Framework

All participants experienced that they needed to consider a handful of objects to set up the program initially (*IDevice*, *ISurface*, *IShader* and *IShaderProgram*). However after this point the working framework is generally low, as only a few objects need to be considered when performing most regular tasks. Participant B commented on how it was sometimes difficult to find out, where certain objects need to be used. For instance, it was not clear after creating an *IRenderPass* object that it needed to be used to record commands into an *ICommandBuffer*. This required him to search through most of the API, which as a result expanded the working framework.

Work-Step Unit

In general, we do not experience that any of our tasks given to our participants can be executed in a single step.

From witnessing the participants, we believe that we can equate the time spent on a task with the amount of work steps to code a solution. Thus by looking at Table 5.1, we can see that the time taken to write a simple program producing a triangle, is about the

same as expanding it to the more complex case of drawing multiple texture cube. This suggests that while the initial setup takes some work-steps, the later changes to the code require less work. Especially the process of parallelizing command construction seems to be low. This was also agreed upon by the participants themselves.

Progressive Evaluation

Participant B experienced a bug in the API, which required facilitators to intervene. In response to this participant B commented on the lack of a logging system, which could print errors to the console. In the APIs current state there is minimal error handling with only a few safety checks being made in the underlying implementation. Participant C also experienced an incorrect error message. The lack of a solid system for handling errors ultimately hurts progressive evaluation of the API, as users do not get the feedback required to support experimentation.

Premature Commitment

Participant C commented that he had to create command buffers, before knowing how they would be used later on. We also witnessed participant B changing the default settings of the features and extensions of an *IDevice*. Activating features and extensions in this manner requires the developer to make some choices in how the GPU should be used later on. Except for these two examples, there seemed to be no issue with premature commitment.

Penetrability

Participant B felt that the API was easy to get into thanks to our use of common names within the field of graphics programming. As a result he almost never had to look up information in the documentation. Both B and C thought there was a good flow between objects, as by looking at parameters in constructors for certain objects, they could figure out what objects they needed to construct beforehand. For instance, to create an *IDevice* object, they saw that they needed to create an *ISurface* object. Participant C however commented on the lack of cross referencing between pages in our documentation, which hampered discoverability.

API Elaboration

Participant A said that he regarded the setting up of shaders and render passes as boiler-plate code, indicating that he may want to write some helper functionality. Otherwise both participant A and C were positive towards the API and did not think of much that could be elaborated upon. Participant B thought that elaboration would be difficult, as the API is heavily tied to Windows and modern C++. He wanted the API to be more portable, so that it could be used on other systems so wrappers for other languages could be written.

API Viscosity

As mentioned previously, it seems that once a base program has been written it becomes easy to make changes in the code. All participants experienced this and thus we determine the viscosity of the API to be rather low.

Consistency

All participants experienced that most other API features could be inferred after drawing a triangle. This may somewhat be attributed to the small size of the API. However participant A and B did have to spend some time, figuring out that resources needed to be bound to an *IRenderPass*. Because of the few issues we experienced, we determine that the API has good consistency. An example of this is that almost every object is created from an *IDevice* object, which the participants found natural after creating the first few objects.

Role Expressiveness

In regards to common elements within graphics programming like shaders and resources, participants had no issue in figuring out their purpose. However participant B noted that some of the newer concepts like command buffers, graphics queues and render passes got mixed up in his head. Participants A and C especially seemed to have issues in figuring out what the purpose of commands were. However after completing the tasks, all stated that the relationships between objects made sense.

Domain Correspondence

Participant A and C had difficulty commenting on domain correspondence in regards to terms, which they were unfamiliar with, such as swap chain and commands. However they liked the use of terms like shader program, which they recognized from OpenGL and Direct3D. Participant B was already somewhat familiar with command submission, but had not encountered sub command buffers or graphics queue before. He thought that *IRenderPass* should be called an *IPipelineStateObject*, as this was a term he was familiar with. Interestingly participant C liked the use of the *IRenderPass* name, but agreed with participant B that the *IDynamicUniformBuffer* should get another name. This is because dynamic may indicate that the allocated memory of the buffer can be resized at runtime.

5.4 Solutions to Usability Problems

Here we describe updates to our API design and wiki, which have been made to solve some of the more severe problems described in Appendix C. The updated, and as of now final, version of PapaGo can be found at [12].

5.4.1 Problem #22: Rendering Without a Swap Chain

Participant B noted that even when utilizing the off-screen rendering capabilities of PapaGo, the swap chain extension needed to be enabled despite the developer not intending to present the results. This is because the constructing method for creating an *IGraphicsQueue* requires a swap chain as input.

In order to fix this problem, the swap chain was removed as an input parameter when creating a graphics queue, and instead given as an input parameter when calling the method *IGraphicsQueue::present*.

However this meant that we did not know, when an image on the swap chain was ready to be presented. To overcome this, fences were used to communicate between the GPU and CPU, as to known when a image was ready to present. Each image within a swap chain has a fence.

5.4.2 Problem #35: Dynamic Binding of Textures

Participant B pointed out that it is not possible to bind new textures to shaders on the same command buffer.

Since textures are bound to a render pass, and render passes are used to record command buffers (which override previous recordings whenever *ICommandBuffer::record* is called), all new instances of these objects would be needed for the relatively simple task of setting a new texture in the same render pass.

```
/* setup code omitted */

std::vector<ParameterBinding> bindings;
  bindings.reserve(3);
  bindings.emplace_back( "view_projection_matrix", viewUniformBuffer.get());
  bindings.emplace_back( "model_matrix", instanceUniformBuffer.get());
  bindings.emplace_back( "sam", texture1.get(), sampler.get());
  auto paramBlock = device->createParameterBlock(*renderPass, bindings);

  bindings.clear();
  bindings.emplace_back("view_projection_matrix", viewUniformBuffer.get());
  bindings.emplace_back("model_matrix", instanceUniformBuffer.get());
  bindings.emplace_back("sam", texture2.get(), sampler.get());
  auto paramBlock2 = device->createParameterBlock(*renderPass, bindings);

/* ... */

ThreadPool tp(4);
for (int t = 0; t < 4; ++t) {
  tp.enqueue([&](int t) {
    subCommandBuffers[t]->record(*renderPass, [&](IRecordingSubCommandBuffer& cmdBuf) {
      auto& block = (t % 2 == 0) ? paramBlock : paramBlock2;
      cmdBuf.setParameterBlock(*block);
      cmdBuf.setVertexBuffer(*vertexBuffer);
      cmdBuf.setIndexBuffer(*indexBuffer);

      for (int i = t * 250; i < t * 250 + 250; ++i) {
        cmdBuf.setDynamicIndex(*block, "model_matrix", i);
        cmdBuf.drawIndexed(36);
      }
    });
  }, t).wait();
}

/* ... */
```

Listing 5.1: Example of multiple parameter block usage in one render pass.

To combat this, *IParameterBlock* and *ParameterBinding* were implemented. The role of the *ParameterBinding* is to couple the name of a uniform variable in a shader to a resource. The *IParameterBlock* represents a *vk::DescriptorSet*. We chose this name, as it is a more general term of the concept. Now, instead of binding resources to the *RenderPass* before recording commands, resources are bound to *IParameterBlock*s, and a *IRecordingSubCommandBuffer::setParameterBlock* method has been added.

Listing 5.1 shows the creation of two parameter blocks at line 8 and 14, each with a unique texture bound it. In line 22 – 23, we choose the parameter block and set a parameter block. In this example, four tasks are enqueued on the thread pool, and every other of them will use *paramBlock* and the rest will use *paramBlock2*.

5.4.3 Problem #36: Improving the Documentation

All participants criticized the documentation for being lacking and not providing enough information. An attempt was made to fix this by adding more cross reference links in each wiki page, so that developers would have easier means of navigating through the PapaGo wiki. This updated wiki can be found at [12]

To work on this problem further would require to add more information about PapaGo into the wiki. This could be in the form of in-depth explanations of classes, or the inclusion of additional code examples. Additionally, the relationship between the classes in PapaGo could be explained using a class diagram.

5.5 Discount Method for API Evalaution

Our original intent was to adapt the discount method for programming language evaluation [31]. However various parts of it were replaced with other methodologies. We no longer consider it an adaptation of the discount method for programming language evaluation but deem it a new method, which we have named The Discount Method for API Evaluation. The method goes as follows:

1. **Create tasks**

   Create the tasks that the test participants will attempt to solve.

2. **Create Documentation**

   The participant should have access to documentation describing how the API works and how to use it. The documentation should be made available to the participants a few days before the test session, it should also be available to the participant during the test session.

3. **Estimate task duration**

   Estimate the amount of time it will take to solve the tasks.

4. **Prepare Setup**

   Prepare location, hardware, interview questionnaire, and recording equipment. If recording equipment is not used, have a dedicated note taker write notes for the entire test session.

5. **Gather participants**

   Find participants which are the target audience for the API, preferably with varying levels of experience to find a broader range of issues.

6. **Start the experiment**

   Before starting the code session, the facilitator should hold a small interview, to estimate what level of experience the participant is at. Once the pre-code interview is done, the code session of the experiment can begin.

7. **Keep the participants talking**

   The participant should be encouraged by the facilitator to always use the think-out-loud-protocol.

8. **Interview the participant**

   After the participant has solved the tasks or if the time for the experiment has run out, the post code interview can begin. It is used to ask the participant, what their first impressions were of the API, and if they would use it in other projects.

   Additionally, the interview is used to let the participant answer a questionnaire through dialogue, based on the cognitive dimensions from [17].

9. **Analyze data**

   Use the data gathered from the test session and interview to fill out cognitive dimensions [17] in combination with, what was observed during the test session. Afterwards the issues detected should be prioritized by using the severity ratings for usability problems provided in [42].

Chapter 6

Application Examples

With PapaGo having been upgraded in response to our usability evaluation, this chapter is dedicated to showing programs developed using the API. These implementations are written using the finalized version of PapaGo and can be found here [12].

6.1 3D Grid of Cubes

This example aims to recreate the scene used for benchmarking in [8, 10]. In this scene, a grid of variable number of $N \times N \times N$ cubes are drawn with the camera placed so that the grid fits in the viewport. An example of the application output can be seen in Figure 6.1. All listings presented in this section are from the same PapaGo program. The vertex shader coded against can be found in Listing 3.12 and the fragment shader can be found in Listing 3.13.

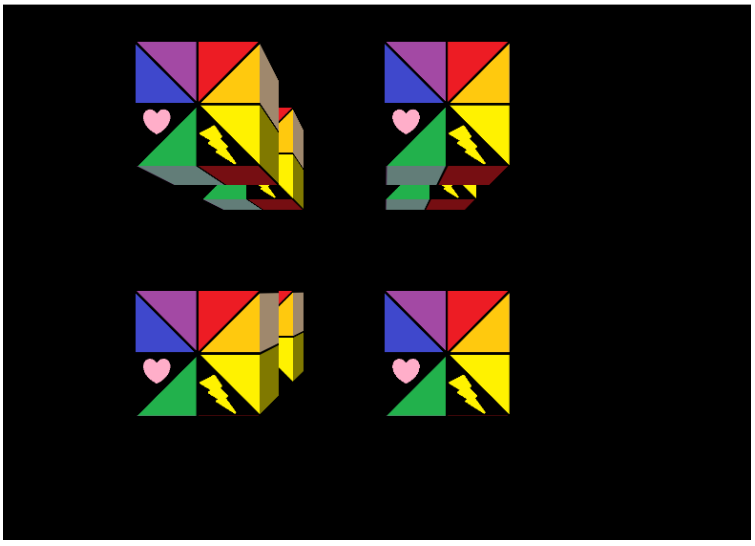

Figure 6.1: Example output from the program described in this section, where $N = 2$.

6.1.1 Resource setup

To upload resources to the GPU, we first need to know which GPU to use through an *IDevice* object, and which *ISurface* to render to. The resulting setup code can be seen in Listing 6.1. Lines 2 – 7 sets up the data needed. The scene set up in line 7 is a collection of objects to be rendered, including all their information such as transformation matrices. lines 9 – 16 creates references to the graphics card and surface to render to. The resources such and vertex-, index- and uniform buffers, textures and samplers are then created in lines 18 – 30.

```
1  int main(){
2      int texW, texH;
3      auto texPixels = /* read texture pixels and output width and height to texW and texH */;
4      std::vector<CubeVertex> cubeVertices{ /* Vertex definitions */ };
5      std::vector<uint16_t> cubeIndices{ /* Indicies */ };
6      auto hwnd = StartWindow(800, 600);
7      auto scene = /* get scene information */;
8
9      auto surface = ISurface::createWin32Surface( 800, 600, hwnd);
10     IDevice::Features features{};
11     features.samplerAnisotropy = true;
12     IDevice::Extensions extensions{};
13     extensions.swapchain = true;
14     extensions.samplerMirrorClampToEdge = false;
15     auto devices = IDevice::enumerateDevices(*surface, features, extensions);
16     auto& device = devices[0];
17
18     auto vertexBuffer = device->createVertexBuffer(cubeVertices);
19     auto indexBuffer = device->createIndexBuffer(cubeIndices);
20     auto texture = device->createTexture2D(texW, texH, Format::eR8G8B8A8Unorm);
21     auto sampler = device->createTextureSampler2D(
22         Filter::eLinear,
23         Filter::eLinear,
24         TextureWrapMode::eRepeat,
25         TextureWrapMode::eRepeat);
26     auto vpMatrix = device->createUniformBuffer(sizeof(glm::mat4));
27     auto mMatrices = device->createDynamicUniformBuffer(
28         sizeof(glm::mat4),
29         scene.renderObjects().size());
30
31     /* ... */
32 }
```

Listing 6.1: Creation of resources for use in this example.

Note that the uniform buffers and texture are not yet filled with data; they are only instantiated with a given size. Only the vertex- and index buffers are filled with data when they are created. The uniform buffers and textures are uploaded to later, which can be seen in Listing 6.2.

```
int main(){
    /* resource creation omitted */

    texture->upload(texPixels);

    vpMatrix->upload(/* calculate view-projection matrix */);

    std::vector<glm::mat4> mMatricesData;
    //dynamic uniform buffer:
    for (auto ro : scene.renderObjects())
    {
        mMatricesData.push_back(/* calculate model matrix for ro */);
    }

    mMatrices->upload(mvpMatricesData);

    /* ... */
}
```

Listing 6.2: Filling the texture and uniformbuffers with data.

6.1.2 Render Pass Setup

Before commands for rendering can be recorded, some additional objects need to be created. This is shown in Listing 6.3. A swap chain is created in the lines 3, it contains the framebuffers which should be rendered to.

Lines 7 – 9 show the creation of a shader program, which is then used to create a render pass in line 10.

Lines 22 – 28 is used to define parameter bindings between uniform variable names and resource, which are then packed together in a parameter block. Note the strings given in lines 24 – 26; these are the names of the uniforms in the shaders.

```
int main(){
  /* resource setup omitted */
  auto swapchain = device->createSwapChain(Format::eR8G8B8A8Unorm, Format::eD32Sfloat,
                      3, IDevice::PresentMode::eMailbox);
  auto parser = Parser(/* Path to glslangValidator.exe */);

  auto vertexShader = parser.compileVertexShader(readFile("shaders/shader.vert"), "main");
  auto fragmentShader = parser.compileFragmentShader(readFile("shaders/shader.frag"), "main");
  auto shaderProgram = device->createShaderProgram(*vertexShader, *fragmentShader);
  auto renderpass = device->createRenderPass(*shaderProgram, surface->getWidth(),
    surface->getHeight(), swapchain->getFormat(), Format::eD32Sfloat );

  std::vector<ParameterBinding> bindings {
    {"view_projection_matrix", vpMatrices.get()},
    {"model_matrix", mMatrices.get()},
    {"sam", texture.get(), sampler.get()}
  };
  auto parameterBlock = device->createParameterBlock(*renderpass, bindings);
  /* ... */
}
```

Listing 6.3: Setting up the render pass and the related data such as shaders. This is also where the buffers are bound to the names in the shaders.

6.1.3 Command construction and Submission

Next, we want to tell the GPU to use the newly uploaded data for draw calls. This can be seen in Listing 6.4. Line 3 creates the *ISubCommandBuffer* object, to which we wish to record.

```
1  int main(){
2    /* resource and render pass setup omitted */
3    auto subCommandBuffer = device->createSubCommandBuffer();
4    subCommandBuffer.record(*renderpass, [&](IRecordingSubCommandBuffer& rcmd) {
5      rcmd.setVertexBuffer(*vertexBuffer);
6      rcmd.setIndexBuffer(*indexBuffer);
7      rcmd.setParameterBlock(*parameterBlock);
8
9      for(auto i = 0u; i < scene.renderObjects().size(); ++i)
10     {
11       rcmd.setDynamicIndex(*parameterBlock, "model", i);
12       rcmd.drawIndexed(cubeIndices.size());
13     }
14   }
15   auto graphicsQueue device->createGraphicsQueue();
16
17   while(running){
18     if(/* Win32 Message is available */){
19       /* Handle Win32 message */
20     }
21     else {
22       commandBuffer->record(*renderpass, *swapchain, [&](IRecordingCommandBuffer& rcmd) {
23         rcmd.clearColorBuffer(0.0f, 0.0f, 0.0f, 1.0f);
24         rcmd.clearDepthBuffer(1.0f);
25         rcmd.execute({*subCommandBuffer});
26       });
27       graphicsQueue->submitCommands({ *commandBuffer });
28       graphicsQueue->present(*swapchain);
29     }
30   }
31 }
```

Listing 6.4: Recording into the different command buffers and submitting the result to the graphics queue.

Lines 4 – 7 begins recording to the *ISubCommandBuffer* object and sets the vertex buffer, index buffer, and parameter block to the objects created in Listing 6.3.

Lines 9 – 13 renders all objects in the scene, each with their own model matrix.

Lines 22 – 26 records to the *ICommandBuffer* object each frame. Here, the color and depth buffer are cleared, and the commands recorded in the *ISubCommandBuffer* object are executed.

The remaining lines submits the *ICommandBuffer* object and presents render target to screen using the *IGraphicsQueue* object.

This example assumes that the scene is static, which means that no object are ever created, destroyed, moved, rotated, or scaled. If the scene was dynamic, the program would update the *ISubCommandBuffer* objects in the while loop.

6.2 Shadow mapping

This example shows an implementation of a shadow mapping lighting model. As the name suggests, it is a way of creating shadows in a scene. Unlike a simple lighting model like phong lighting, shadow mapping takes all other objects in the scene into account. Therefore objects in the scene can cast shadows on other objects.

To do this, multiple render passes are needed. The idea is to first render the entire scene from the light sources point of view into an off screen buffer framebuffer, which contain a depth buffer. Then the texture, which is bound as the depth buffer of the first render pass, is given as input to the next render pass as a texture uniform. In the second render pass, the scene is rendered from the camera's point of view. Each fragment visible to the camera then gets the position, it would have had from the light source's point of view, calculated. If the z-coordinate (depth) of this position is lower than the corresponding value in the depth buffer from the first render pass, then this fragment must be in shadow.

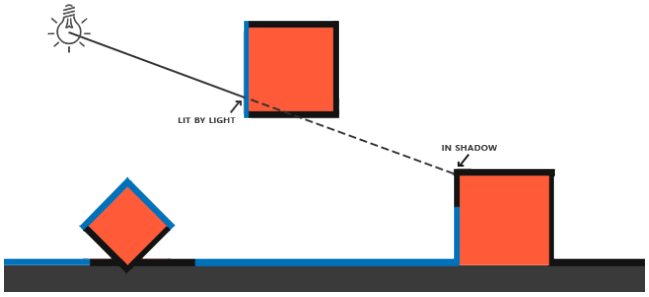

Figure 6.2: Illustration of the concept of shadow mapping from [54].

If the scene in Figure 6.2 were to be rendered from the light's point of view, then the length of the fully drawn line from the light would be the value in the depth buffer for the fragment where the line meets the box (marked with *LIT BY LIGHT*).

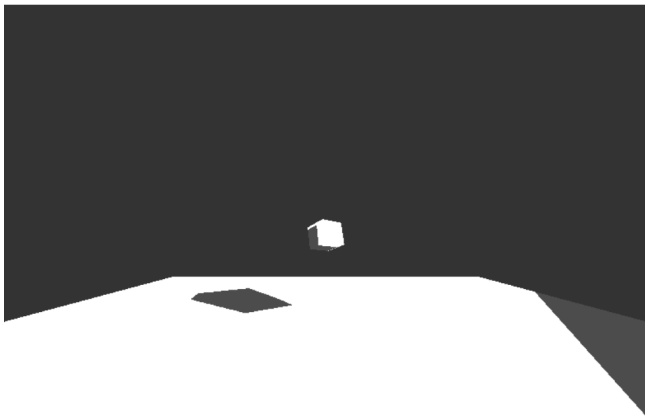

Figure 6.3: Example of shadow mapping rendered with PapaGo.

The following sections of code produces the result seen in Figure 6.3. Note how the shadow of the cube is hitting both the cube itself and the ground below it. The code presented is part of a continuous program, and shaders coded up against can be found in Appendix D.

6.2.1 Resource setup

Listing 6.5 shows the code which sets up the needed resources: Lines 5 – 25 creates two combined view and projection matrices (*vpMatCam* and *vpMatLight*). Note that both matrices were created using the same projection matrix, so both the camera and the light is going to have perspective. In Listing 6.6 Lines 2 – 8 creates two model matrices; one for the ground and one for the floating cube (*modelMatGround* and *modelMatCube*, respectively). Lines 10 – 14 puts these matrices into two separate dynamic uniform buffers; one for model matrices and one for view/projection matrices. The model matrix buffer has the ground model matrix at index 0, and the cube model matrix at index 1. The view/projection matrix buffer has the camera matrix at index 0, and the light matrix at index 1. Lines 16 – 29 creates the textures (one color and one depth) the first pass will render to, plus a sampler which will be used by the second render pass to sample from these textures. Lines 31 – 34 sets up a texture to be used for the models in the second render pass, and uploads pixel data to it.

```
1  void main()
2  {
3      /* Creation of window, surface, device, parser, vertices, and indices omitted...*/
4
5    auto viewMatCam = glm::lookAt(
6      glm::vec3{ 0.0f, 0.0f, 10.0f },     //<-- world position
7      glm::vec3{ 0.0f, 0.0f, 0.0f },      //<-- target
8      glm::vec3{ 0.0f, 1.0f, 0.0f }       //<-- up
9    );
10
11   glm::vec3 lightPos = {0.0f, -4.0f, 2.0f};
12   auto viewMatLight = glm::lookAt(
13     lightPos,
14     glm::vec3{0.0f, 0.0f, 0.0f},
15     glm::vec3{0.0f, 1.0f, 0.0f}
16   );
17
18   auto projMat = glm::perspective(
19     glm::radians(90.0f),                                  //<-- FOV
20     float(surface->getWidth()) / surface->getHeight(),   //<-- aspect ratio
21     0.1f,                                                 //<-- near plane
22     20.0f);                                               //<-- far plane
23
24   auto vpMatCam = projMat * viewMatCam;
25   auto vpMatLight = projMat * viewMatLight;
26   /*...*/
```

Listing 6.5: Resource setup part 1, setting up the two view-projection matricies.

```
    /*...*/
  auto modelMatGround =
      glm::translate(glm::vec3{ 0.0f, 4.0f, 0.0f })
      * glm::scale(glm::vec3{ 25.0f, 1.0f, 25.0f });

  auto modelMatCube =
      glm::translate(glm::vec3{0.0f, 0.0f, 0.0f})
      * glm::rotate(glm::radians(45.0f), glm::vec3{ 0.5f, 1.0f, 0.0f });

  auto vpDynUniform = device->createDynamicUniformBuffer(sizeof(glm::mat4), 2);
  vpDynUniform->upload<glm::mat4>({ vpMatCam, vpMatLight });

  auto modelDynUniform = device->createDynamicUniformBuffer(sizeof(glm::mat4), 2);
  modelDynUniform->upload<glm::mat4>({ modelMatGround, modelMatCube });

  auto colTarget = device->createTexture2D(
      surface->getWidth(),
      surface->getHeight(),
      Format::eR8G8B8A8Unorm
  );
  auto colTargetDepth = device->createDepthTexture2D(
      surface->getWidth(),
      surface->getHeight(),
      Format::eD32Sfloat
  );
  auto colSampler = device->createTextureSampler2D(
      Filter::eLinear, Filter::eLinear,
      TextureWrapMode::eRepeat, TextureWrapMode::eRepeat
  );

  int texW, texH;
  std::vector<char> pixelData = /* Load pixels and store width and height in texW and texH */
  auto tex = device->createTexture2D(texW, texH, Format::eR8G8B8A8Unorm);
  tex->upload(pixelData);
  /* ... */
}
```

Listing 6.6: Resource setup part 2, setting up the rest of the resources.

6.2.2 Render Pass Setup

In Listing 6.7 the lines 4 – 19 sets up the render pass object and parameter block object to be used for the first pass (the one recording depth data from the light sources point of view). Lines 21 – 36 creates the command buffer and sub command buffer which will be used to record the first render pass, and records the sub command buffer. In Listing 6.8 Lines 9 – 27 sets up the render pass object and parameter block object to be used for the second render pass. Note that two separate parameters shares the same buffer (*view_projection* and *shadow_view_projection*). Lines 29 – 45 creates the command buffer and sub command buffer, which will be used to record the second render pass (the one from the cameras point of view). Note the two bindings, which share a buffer, are simultaneously bound to different dynamic indices (lines 35 – 36). The final line creates a graphics queue object.

```
1  void main()
2  {
3  /*previous setup omitted*/
4    auto colorVert = parser.compileVertexShader(readFile("shaders/clipCoord.vert"), "main");
5    auto colorFrag = parser.compileFragmentShader(readFile("shaders/clipCoord.frag"), "main");
6    auto colorProgram = device->createShaderProgram(*colorVert, *colorFrag);
7    auto colorPass = device->createRenderPass(
8        *colorProgram,
9        colTarget->getWidth(),
10       colTarget->getHeight(),
11       colTarget->getFormat(),
12       colTargetDepth->getFormat()
13     );
14
15   std::vector<ParameterBinding> colorParams;
16   colorParams.emplace_back("vp", vpDynUniform.get());
17   colorParams.emplace_back("m", modelDynUniform.get());
18
19   auto colorParamBlock = device->createParameterBlock(*colorPass, colorParams);
20
21   auto colorCmd = device->createCommandBuffer();
22   auto colorSubCmd = device->createSubCommandBuffer();
23   colorSubCmd->record(*colorPass, [&](IRecordingSubCommandBuffer& rcmd) {
24     rcmd.setParameterBlock(*colorParamBlock);
25     rcmd.setVertexBuffer(*vertices);
26     rcmd.setIndexBuffer(*indices);
27     rcmd.setDynamicIndex(*colorParamBlock, "vp", 1);  //<-- use light view
28
29     //ground:
30     rcmd.setDynamicIndex(*colorParamBlock, "m", 0);
31     rcmd.drawIndexed(indexCount);
32
33     //cube:
34     rcmd.setDynamicIndex(*colorParamBlock, "m", 1);
35     rcmd.drawIndexed(indexCount);
36   });
37   /*...*/
```

Listing 6.7: First render pass setup.

```
    /*...*/
  auto swapchain = device->createSwapChain(
      Format::eB8G8R8A8Unorm,     //<-- color buffer format
      Format::eD32Sfloat,         //<--- depth buffer format
      3,
      IDevice::PresentMode::eMailbox
    );

  auto shadowVert = parser.compileVertexShader(readFile("shaders/shadow.vert"), "main");
  auto shadowFrag = parser.compileFragmentShader(readFile("shaders/shadow.frag"), "main");
  auto shadowProgram = device->createShaderProgram(*shadowVert, *shadowFrag);
  auto shadowPass = device->createRenderPass(
      *shadowProgram,
      swapchain->getWidth(),
      swapchain->getHeight(),
      swapchain->getFormat(),
      Format::eD32Sfloat      //<-- depth buffer format (same as swap chain)
    );

  std::vector<ParameterBinding> shadowParams;
  shadowParams.emplace_back("view_projection", vpDynUniform.get());
  shadowParams.emplace_back("shadow_view_projection", vpDynUniform.get());
  shadowParams.emplace_back("model", modelDynUniform.get());
  shadowParams.emplace_back("tex", tex.get(), colSampler.get());
  shadowParams.emplace_back("shadow_map", colTargetDepth.get(), colSampler.get());

  auto shadowParamBlock = device->createParameterBlock(*shadowPass, shadowParams);

  auto shadowCmd = device->createCommandBuffer();
  auto shadowSubCmd = device->createSubCommandBuffer();
  shadowSubCmd->record(*shadowPass, [&](IRecordingSubCommandBuffer& rcmd) {
    rcmd.setParameterBlock(*shadowParamBlock);
    rcmd.setVertexBuffer(*cube.vertex_buffer);
    rcmd.setIndexBuffer(*cube.index_buffer);
    rcmd.setDynamicIndex(*shadowParamBlock, "view_projection", 0);  //<-- camera view
    rcmd.setDynamicIndex(*shadowParamBlock, "shadow_view_projection", 1); //<-- light view

    //ground:
    rcmd.setDynamicIndex(*shadowParamBlock, "model", 0);
    rcmd.drawIndexed(cube.index_count);

    //cube:
    rcmd.setDynamicIndex(*shadowParamBlock, "model", 1);
    rcmd.drawIndexed(cube.index_count);
  });

  auto graphicsQueue = device->createGraphicsQueue();
  /*...*/
}
```

Listing 6.8: Second render pass setup

6.2.3 Render Loop

In Listing 6.9 the lines 10 – 19 moves the light source in a circular motion around the world origin. Lines 21 – 36 records the two primary command buffers, one for each render pass. The first command buffer (*colorCmd*) must be re-recorded each frame because the light moves. If the light movement is removed, then the recording of *colorCmd* could be done outside the render loop. The second command buffer (*shadowCmd*) must be recorded each frame, as it records to the swap chain, which contains three framebuffers (line 5 in Listing 6.8), but only one framebuffer is recorded to at a time. The remaining code submits the two command buffers to the graphics queue (in correct order) and presents the swap chain.

```
1  void main()
2  {
3      /*previous setup omitted*/
4
5      run = true;
6      while (run)
7      {
8          /*frame timing and window message pump handling omitted*/
9
10         lightPos =
11             glm::vec3(glm::rotate(0.01f, glm::vec3{0.0f, 0.0f, 1.0f})
12             * glm::vec4(lightPos, 1.0f));
13
14     viewMatLight = glm::lookAt(
15       lightPos,
16       glm::vec3{ 0.0f, 0.0f, 0.0f },
17       glm::vec3{ 0.0f, 1.0f, 0.0f }
18     );
19     vpDynUniform->upload<glm::mat4>({ projMat * viewMatLight }, 1);
20
21     colorCmd->record(
22         *colorPass,
23         *colTarget,
24         *colTargetDepth,
25         [&](IRecordingCommandBuffer& rcmd) {
26         rcmd.clearColorBuffer(1.0f, 0.0f, 1.0f, 1.0f);
27         rcmd.clearDepthBuffer(1.0f);
28         rcmd.execute({ *colorSubCmd });
29             }
30       );
31
32     shadowCmd->record(*shadowPass, *swapchain, [&](IRecordingCommandBuffer& rcmd) {
33       rcmd.clearColorBuffer(0.2f, 0.2f, 0.2f, 1.0f);
34       rcmd.clearDepthBuffer(1.0f);
35       rcmd.execute({ *shadowSubCmd });
36     });
37
38     graphicsQueue->submitCommands({ *colorCmd, *shadowCmd });
39     graphicsQueue->present(*swapchain);
40     }
41 }
```

Listing 6.9: Shadow map render loop example

Chapter 7

Performance Benchmark

Having evaluated the usability of PapaGo, a performance benchmark is also executed. While our main goal has been one of increasing usability, developers will not adopt the tool, if the cost in performance is too high. In this chapter, we describe our performance benchmark process and the results gathered from it.

7.1 Benchmark Setup

To benchmark performance, we use an extended version of the application described in Section 6.1. It is extended so that it may render models other than cubes, and commands can be recorded in parallel with different amounts of threads. This makes the application equivalent with the benchmarking application used in our earlier work with Vulkan [10, 8]

Thus it possible to compare the performance of PapaGo with the Vulkan performance data presented in [8].

In this section, we discuss the different configurations under which this application is tested, as well as the data collected from tests.

7.1.1 Test Configurations

Different configurations are used when running the tests. A configuration is given by the tuple $(modelType, threadCount, gridDimension, hardware)$.

Model type refers to the model used for each element in the 3D grid rendered. The model rendered can be either a cube consisting of 12 triangles or a human skull consisting of 9537 triangles. This allows us to switch between a more CPU-bound case with cubes and a more GPU-bound case with skulls.

Thread count indicates the number of threads used to construct commands for submission, and grid dimension indicates the side length of the 3D grid in terms of elements.

Hardware determines, which hardware setup the application is run on. To make the results gathered easier to generalize, we use two different hardware setups, each with a GPU from one of the two big vendors: AMD and NVIDIA. The first system has an AMD FX-8320 CPU and a AMD Sapphire Radeon R9 280 GPU, the second has an Intel Core i7-4770k CPU and a GeForce GTX 1060 GPU. These will from now on be referred to as AMD and NVIDIA respectively, and their detailed specification can be seen in Table 7.1. Notice that while the two systems have similar CPUs, the NVIDIA system has a more powerful GPU.

		AMD	**NVIDIA**
CPU	Clock Rate (GHz)	3.5	3.5
	# Logical Threads	8	8
	RAM (GB)	8	8
GPU	Clock Rate (MHz)	850	1506
	VRAM (GB)	3	3
	# Shading Units	1152	1792
	# Texture Mapping Units	72	112

Table 7.1: Specification of the two hardware setups used for the benchmarks.

7.1.2 Collecting Data

When running a configuration, there is the question of what data to record. As we are interested in performance, we look into the speed at which frames are produced and presented on screen by the application.

Two metrics exists for measuring this, frame rate and frame time. Frame rate is the frames per second produced, and frame time is the time it takes to produce a single frame. If frame time is measured in milliseconds the relationship between the two metrics is given as:

$$\text{frame time} = \frac{1000}{\text{frame rate}} \tag{7.1}$$

From this it can be seen, that when the frame rate is high, small differences in frame rate have a small impact on the frame time. For instance, 60 FPS is equivalent with 16.67 ms per frame, while 59 is equivalent with 16.95 ms per frame. However at low frame rates, a small change makes a huge difference. Given 2 FPS, which is equivalent to 500 ms, and 3 FPS, equivalent to 333 ms, the difference is much greater.

Thus, when performing benchmarks at high loads, accuracy is lost by using frames per second as a metric. Therefore we choose frame time as the metric recorded. In order to minimize probe effect, the frame time is recorded only once each second, using the frame time of the last frame presented.

The result of each test configuration is the mean frame time recorded. However the nature of running our application on a system with multiple background processes means that data points will scatter around the true mean in some manner.

We ran a single pilot test on the configuration $(cubes, 1, 30, AMD)$, running it for 300 seconds, which yielded a mean μ of 49.09 and a standard deviation σ of 2.32. Plotting the bell curve defined by these two variables along with the frequency distribution over the data points yields the graph in Figure 7.1.

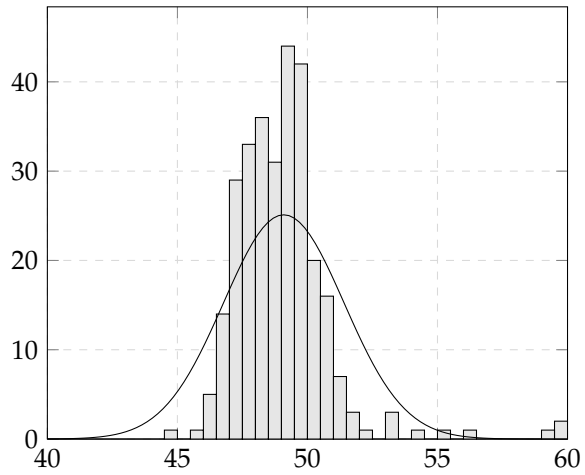

Figure 7.1: Bell curve fitted over frequencies of data recorded during pilot test.

While not an exact match, this shows the data gathered to approximately fit with a normal distribution.

Knowing that the data behaves in this way, it is possible to calculate with some certainty, the number of samples to collect from each configuration, so that we may calculate the true mean frame time. Finding the number of samples is important, as we want to ensure that we get enough data, while not spending too many resources on testing.

The formula, as can be found in [15], defining the number of samples to collect is given as:

$$n = (\frac{Z_{\alpha/2} * \sigma}{E})^2 \tag{7.2}$$

Where n is the number of samples, Z is the Z-table function, α is the alpha level, σ is the standard deviation and E is the error margin.

Alpha level $alpha$ indicates the probability that this estimation procedure will not find the true value of our frame time, given the error margin E. It is related to the confidence level, which is the probability that the process will find the true value. If our confidence level is 95%, then α is given as:

$$\alpha = (1 - 0.95) = 0.05 \tag{7.3}$$

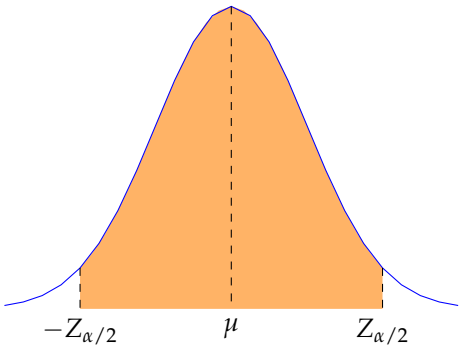

Figure 7.2: Normal distribution, showing the points on the x axis yielded by the Z-table function.

The Z-table function calculates the number of standard deviations the input lies away from the mean given a standard normal distribution. This is defined as having a mean μ of 0 and a standard deviation σ of 1. In the formula, $\alpha/2$ is used as input to the Z-table function to find the distance to each of the two tails of the bell curve, which can be seen on Figure 7.2.

If we want a confidence level of 95% then the Z value calculated is:

$$Z_{0.05/2} = 1.96 \tag{7.4}$$

This means that the area of the distribution between -1.96σ and 1.96σ retain 95% of the area under the curve.

The error margin E signifies, how big an error we can accept to either side of the true mean. So an error of $E = 0.5$ milliseconds with a confidence level of 95% would mean that there is a 95% chance that the measurement is true within an error of 0.5 milliseconds.

Using the standard deviation calculated during the pilot test, $\sigma = 2.32$ milliseconds, along with a 95% confidence level and an error margin $E = 0.5$ milliseconds, the number of samples is calculated as:

$$n = (\frac{Z_{\alpha/2} * \sigma}{E})^2 = (\frac{Z_{0.05/2} * 2.32}{0.5})^2 = (\frac{1.96 * 2.32}{0.5})^2 = 82.71 samples \tag{7.5}$$

Rounding up, this becomes 83 samples.

Because the benchmarks run on systems with background processes, which are difficult to manage, performance of the same configuration may differ depending on when it is run. To get around this, we run the same configuration five times, collecting 30 data points for each run. After completion, we then disregard the two worst performing runs, and calculate the mean frame time of the three best runs. This allows us to ignore the odd run, where a background process significantly impacts performance.

We have also noticed that the first data point collected during a run is often a big outlier. Therefore we choose to disregard this point.

This means that the mean frame time of a configuration is computed from the 29 last data points recorded from the three best runs out of five runs, yielding 87 samples in total.

7.2 Tests and Results

In this section, we describe the exact tests performed using the test application and their results. The two types of tests performed are the same as found in [10, 8]. The scripts used to perform the tests, and the raw data collected, can be found at [9]. The results presented in this section will be discussed further in Section 8.3, where a comparison with Vulkan performance is also brought up.

7.2.1 Increasing Threads

One of the design requirements for PapaGo was that *The API should be able to speed up CPU-bound applications*. We have attempted to support this by allowing for multithreaded command construction, similar to Vulkan. Therefore we want to see how the number of threads impact the performance of the test application.

The configurations tested are defined as $(hardware, modelType, threadCount, 30)$. That is, all variables are varied except for the grid dimension, which stays at 30. 30 was chosen from a few pilot tests, as this is the grid dimension, where the application runs at about 60 frames per second on both hardware setups. 60 frames per second is significant, as this is the goal most professionally made games strive for.

The number of threads tested was 1 – 16, as both hardware setups have 8 cores on their CPUs. We do not expect to see a great difference in performance once all cores are active, but for the sake of certainty this number is doubled from 8 to 16.

Results

Figure 7.3 shows the results of increasing the number of threads, when rendering cubes with both hardware setups.

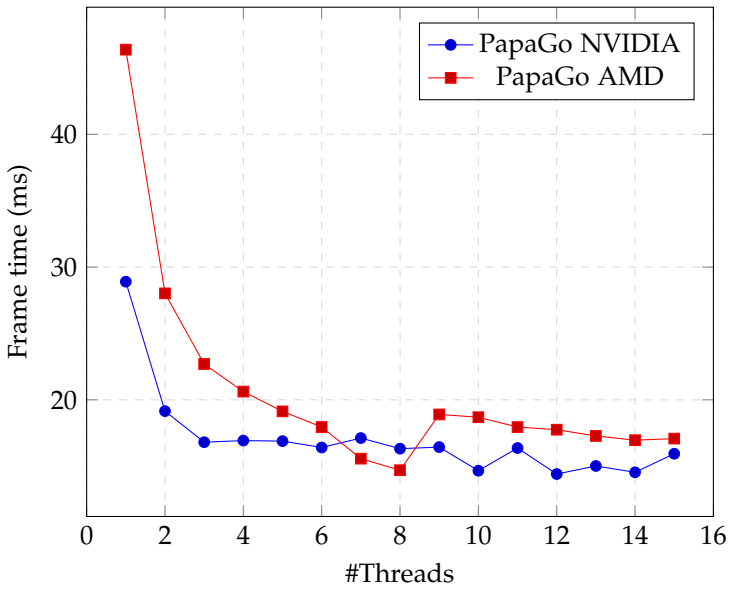

Figure 7.3: Frame time for variable amount of thread, when rendering a 30x30x30 grid of cubes.

In both cases, the first part of the plots can be fitted with a polynomial function. This means that there is a polynomial decrease in frame time, when adding the first few threads. Therefore multithreaded command construction has a big impact in this more CPU-bound case.

However, after the first few threads they stop having an impact. In the case of NVIDIA this happens already after three threads. With AMD, performance increases until reaching the number of cores on its CPU, which is eight. After this point, adding more threads negatively affects performance on AMD, while it boosts the performance slightly on NVIDIA.

Comparing the two plots, AMD gets the most benefit of multithreading, and it even outperforms NVIDIA at seven and eight threads. However for the most part, NVIDIA outperforms AMD to a smaller degree, by approximately 5 milliseconds.

Figure 7.4 shows the results of increasing the number of threads, when rendering skulls on both hardware setups.

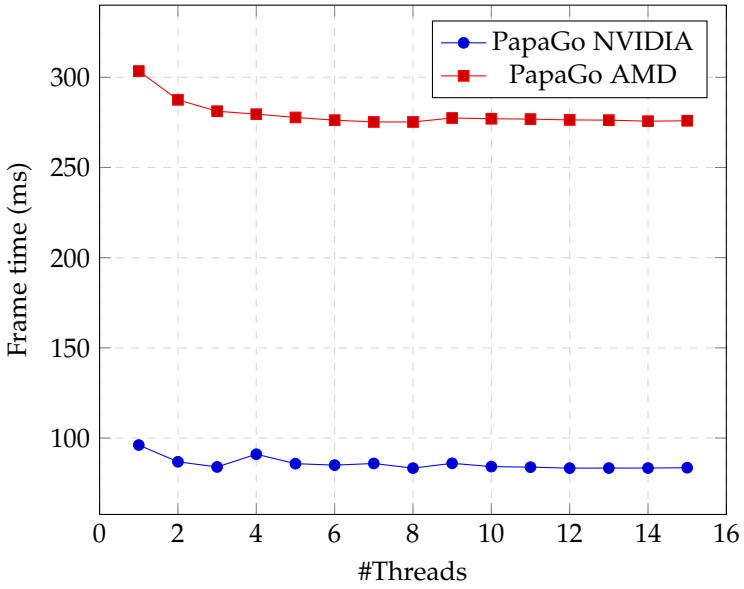

Figure 7.4: Frame time for variable amount of thread, when rendering a 30x30x30 grid of skulls.

By rendering skulls instead of cubes, more work needs to be done on the GPU per draw call, which makes the application more GPU-bound Again the beginning of both plots have a polynomial decrease in frame-time, but with a much lower impact than in the case with cubes. In addition, the application runs about three times faster on NVIDIA. This is expected, as the the NVIDIA system has a more powerful GPU, and when the application is GPU-bound, the power of the GPU sets the performance. In the CPU-bound case, the performance is defined more by the speed of the CPU and the application's host code. This can be seen across the results shown in this section.

Having conducted these test, we have found the optimal amount of threads to use for each combination of hardware and model. This is depicted in Table 7.2.

	AMD	NVIDIA
Cubes	7	12
Skulls	7	15

Table 7.2: Amount of threads for each combination of hardware and model that gave the best performance in our tests.

7.2.2 Increasing Load

In addition to knowing how the number of threads affect performance, we also want to know how the number of objects to render impacts the performance of the application under optimal conditions.

Configurations are defined as $(hardware, modelType, optimalThreads, gridDimension)$. Here the optimal amount of threads is defined in Table 7.2. The variable *gridDimension* is made to increase from 5 to 50 with 5 step increments. This is chosen as $5 \times 5 \times 5$ models is a very light load, while $50 \times 50 \times 50$ model is well above yielding anything close to 60 frames per second.

Results

Figure 7.5 shows how the load changes performance when rendering cubes. In both cases this shows a linear relationship between load and frame time. However the NVIDIA plot follows this trend more closely than in the AMD case. The plots follow each other very closely, but AMD has a slightly better performance at higher loads, up to approximately 5 milliseconds.

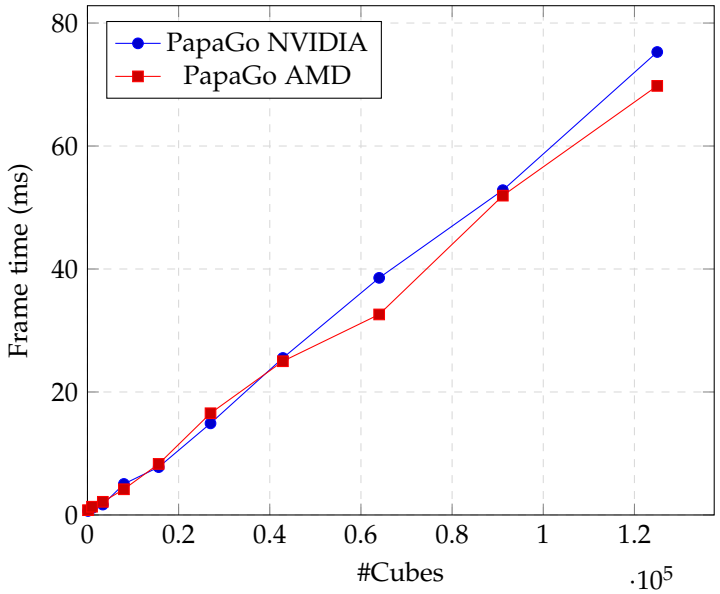

Figure 7.5: Frame time for variable amount of cubes, when rendering a grid of cubes with optimal number of threads.

Figure 7.6 shows how load changes performance when rendering skulls. The relationship between load and frame time is still linear, however the last point on the AMD plot breaks with this. This may indicate that at this load, the amount of data causes the AMD GPU to experience more cache misses. Again we notice that when the application is GPU-bound, then the faster NVIDIA GPU performs the best, with it performing 4 times faster in the case where the grid has dimensions $50 \times 50 \times 50$.

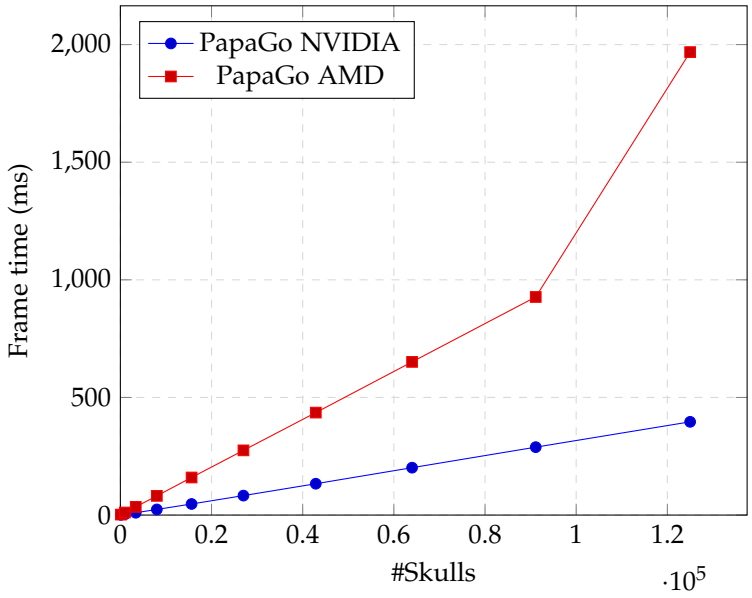

Figure 7.6: Frame time for variable amount of skulls, when rendering a grid of skulls with optimal number of threads.

Chapter 8

Discussion

This chapter discusses the design process for PapaGo, as well as how it was evaluated on usability and performance. In light of the discussions presented here, the report will be concluded upon in the next chapter.

8.1 Design Process

To design PapaGo we defined our own design process by using the guidelines described in [5] as a baseline. Critiquing [5], we find that its nine guidelines are a mix of processes and advice. We believe that this makes the use of the guidelines less clear, and we therefore suggest splitting them into two sections. The first section should include only processes:

1. Define user requirements.
2. Write use cases for the API.
3. Define the API.
4. Have your peers review the API.
5. Write examples with the API.

The second section should include only advice:

1. Take inspiration from similar APIs.
2. Ready it for extension.
3. Do not make the API public before the time is right.
4. Leave out functionality, which may not be necessary.

Looking at the processes, it may be noted that the guidelines suggest that an API should be defined using a waterfall approach. In this manner, the specification of the API is described in full before any implementation occurs. Yet [5] also hints at some circularity within the process, with steps two and five involving a similar task.

By following these processes, we defined our API specification in full and began implementing it. However during implementation, we kept having to redesign the specification. This was either, because there were difficulties in mapping PapaGo down to Vulkan, or because we discovered edge cases, which had not been considered in the original design. This lead us to adopt a rather ad hoc process of redesigning parts of the API when necessary.

While we could have avoided some of these problems by spending more time on the original design, we believe that an overall iterative approach would have been more beneficial. This would involve switching between design and implementation, which lets the two processes inform each other. As to not have implementation details dominate the design, we argue that more time has to be spent on the design in the first iterations. Yet, as the process goes on, the specification should become more static and more time should be spent on the implementation and finally performance.

During the design part of each iteration, new insights may be gathered through peer reviews. Yet, we find that user evaluations lead to the discovery of more issues, because participants have access to a partially implemented API. Therefore it would be interesting to include user evaluations within the iterative design process. However user evaluations are expensive and can only be used, when the implementation has reached a certain point. Thus they should be used sparingly and during later iterations of the overall process.

8.2 User Evaluation

To evaluate the design of PapaGo, we performed three user evaluation sessions, as described in Section 5.1. Setting up and executing these sessions, we used an altered version of [31].

8.2.1 Test Sessions

This section discusses how tests were conducted with our three participants: A, B and C.

Participant A and C were both part of the intended user group for PapaGo, and because of their similar level of skill, they discovered some of the same usability problems. Participant B in comparison was not a part of this group, and with his experience he ended up finding other problems. Because of his expertise, he also focused on some technical details in his comments, which regarded performance and portability of the API. For instance, he commented on how the virtual function calls into the API could decrease performance. While this was not related to the usability of PapaGo, it was important information, which we wish to have had earlier on in the implementation process.

Were we to redo the process, participant B would have been consulted as a technical expert during the design and implementation of the API. The usability of the final version of PapaGo would then be evaluated by participant A and C, as they are part of our intended user group.

In [31], it was mentioned that participants had a tendency to freeze up during sessions. We hoped to decrease the likelihood of this occurring by providing API documentation to the participants ahead of time, so that they had a while to get familiar with the system. While participant B never froze up, A and C both experienced bouts of nervousness, even with documentation available beforehand and during the sessions. It seems that test ses-

sions are very unnatural to participants, with participant A noting explicitly that coding would be easier without someone looking over his shoulder. During the sessions, we attempted to alleviate tension by providing hints and encouraging discussion between the participant and the facilitator.

It would be interesting to formalize, how the facilitator can interact with the participant, in order to let the session run smoothly. We had no formalized method, meaning that interactions were controlled by the style of the given facilitator, with some being more guiding than others.

To decrease tension further, it may have helped to decrease the number of people present at the sessions. As we were not recording the sessions, we assumed it best to have every author experience the reactions of the participant. Yet, with four people paying attention to the participant at once, the latter will have felt some pressure. Recording the session may therefore have been better, but being recorded can also be stressful to the participant.

8.2.2 Evaluation Results

Evaluating upon PapaGo, all participants ended up having a positive impression of the API, though they did not want to use it for any serious work in its current state. We do not see this as a great issue, as our goal has not been to provide a polished end product, but evaluate our design ideas and prove that the primary features of Vulkan are valuable.

In this section, we will evaluate the results of our test sessions, both in terms of our design requirements, and then compared to our old evaluation of Vulkan from [10].

Compared to our design requirements

Our API requirements are as follows:

1. The API should be able to speed up CPU-bound applications.
2. Parameter binding should be intuitive.
3. Resource handling should be explicit but intuitive.
4. The API should be stateless.

To speed up CPU-bound applications, PapaGo allows for *ISubCommandBuffers* to be constructed in parallel. None of the participants were previously familiar with using GPU commands directly, but they all learned to use them after a while. Only participants B and C got to parallelize their programs with the aid of a thread pool. In both cases, the participants were able to quickly make their programs run in parallel, which allowed for the speed up of CPU-bound applications.

With respect to parameter binding, participant B and C needed some time to figure out, what names in the shader to bind to. It was not immediately obvious that binding should occur to names inside of interface blocks, rather than the names of the blocks themselves. However after realizing how binding to names occurred, there seemed to be no issue. For all participants there was some confusion, as to why parameters need to be bound through *IRenderPass* objects. Participant B also noted that this setup did not allow for textures to be changed through commands, so this action would require a new render

pass to be executed. In response to this, we reintroduced the parameter block concept from Vulkan into PapaGo, so that all parameters can be bound through commands. While this adds more complexity to PapaGo through a new class, the amount of flexibility added outweighs the loss of abstraction.

Participant B stated that he liked the explicit handling of resources in PapaGo, as it gave him more control over the application. All participants were able to create and upload data to resources with no problem. However the naming of *IDynamicBufferResource* left its purpose ambiguous. While PapaGo supports the same type of explicit resource handling as in Vulkan, it does not require the developer to do any pointer arithmetic or calculate any padding. Thus we see that explicit resource handling does not present a problem, as long as it is kept simple.

In PapaGo the entire render state is wrapped in an *IRenderPass* object, which must be used for recording commands. Making state explicit caused no great confusion for participants, who had otherwise been used to the implicit state of OpenGL. However the naming of the class caused some issues. Participant A was not familiar with the term, and participant B figured it to be a way of structuring commands rather than a state. Participant B stated that he wanted the state refereed to as a pipeline state object. However Participant C found the naming of *IRenderPass* to be natural.

In total, we find that we live up to the requirements set up in Chapter 3. However there were some problems with, how some concepts were named.

Compared to Previous Vulkan Evaluation

Comparing our Cognitive Dimensions analysis of Vulkan, found in [10], and the same analysis applied upon PapaGo in Section 5.3, it seems that PapaGo is more usable than Vulkan.

In terms of **abstraction level**, developers using PapaGo need not worry about complexities like pointer arithmetic or image layout transitions as is the case in Vulkan. Many of the objects in Vulkan have been abstracted away or have been combined in PapaGo, such as in the case *IRenderPass*, which contains both a Vulkan render pass and pipeline state object.

The **learning style** of both APIs is top-down, but while participant A and C made use of documentation during our sessions, participant B could program without the extra help. This indicates that PapaGo can also be used in a bottom up manner, if the developer has enough knowledge of the field.

On the surface, both PapaGo and Vulkan have a large **working framework**, as many objects, including resources, commands and render state, must be considered at the same time. However our participants did not seem to find the working framework to be large, perhaps because of the high level of abstraction.

The overall **work-step unit** is also similar. For instance, it takes the same number of overall steps to add a new uniform to a shader. This involves an update of the shader, the creation of a new render state and resource, followed up by a parameter binding. However, even with this number of steps in PapaGo, it did not seem to be a large problem to our participants. Thus, we find that the downside to Vulkan in this regard was not the number of overall steps to complete, but the number of sub-tasks for each step, which have been mostly abstracted away in PapaGo.

Vulkan is better in terms of **progressive evaluation**, as it has a full fledged error handling system. On the other hand, the current version of PapaGo does have a logging system implemented, but only a few error checks and messages have been added to it.

In both APIs developers must make **premature commitments**, such as enabling certain features on their device. It is also best practice in both APIs, to create as many objects as possible before the render loop.

Penetrability of PapaGo was experienced as rather high with Participant B even being able to use it without documentation. All participants felt that the order of object initialization had a natural flow, even if they did have to use the documentation sometimes. This is different from Vulkan, where we ourselves had a lot of issues getting into the API, spending our first week to create a triangle. Using PapaGo, participants were able to do this as only a part of our three hour test sessions.

Using Vulkan, we saw it necessary to **elaborate** on the API, abstracting away some of the detail presented with our own functions. Our participants saw no big reason to do this in the case of PapaGo, because of its high level of abstraction.

Looking at **viscosity**, PapaGo scores low, as it was easy for participants to make changes to their program after drawing their first triangle on screen. However in Vulkan, we had to spend a lot of time when going beyond drawing the first triangle.

Both PapaGo and Vulkan seem to be pretty **consistent** in their design, with PapaGo having the advantage of being a much smaller API.

Both APIs experience issues with **role expressiveness**. As an example with Vulkan, the naming of the different objects used for parameter binding caused confusion. In PapaGo the names of classes like *IRenderPass* and *IDynamicUniformBuffer* were not expressive enough. However with the lower number of classes to handle, our participants seemed to be less burdened by these problems with naming.

Both APIs use common terminology and have good **domain correspondence**. However we did find that participants A and C were not familiar with some more advanced terms like swap chain. However, we assume that it is better to stick with this terminology than creating our own. This assumption was backed by participant B.

8.3 Benchmark Results

In this section, we discuss the results of benchmarking PapaGo in comparison to Vulkan, as well as how performance may be improved.

8.3.1 Comparing Performance with Vulkan

In [8], we performed the same manner of benchmarking as in Chapter 7. This allows us to compare results, however this may not be fair to PapaGo, which by design has some overhead. It would be difficult to have PapaGo run as fast as Vulkan, so the Vulkan performance should be seen as an ideal. A more fair comparison would perhaps be in regards to OpenGL, which we pose as a possible future work.

Figure 8.1 shows how performance is impacted by rendering cubes with an increasing number of threads. This shows that the application developed using PapaGo runs slower than the equivalent application written in Vulkan. Looking at the optimal number of threads when running the application on either NVIDIA or AMD, the frame time is about 3 times bigger, when using PapaGo. However, the frame time is improved to a much larger degree by the addition of multiple threads in the PapaGo case. For instance, 8 threads in PapaGo on AMD decrease the frame time from about 50 milliseconds to 15, while this is about 6 to 4 milliseconds when running Vulkan on AMD.

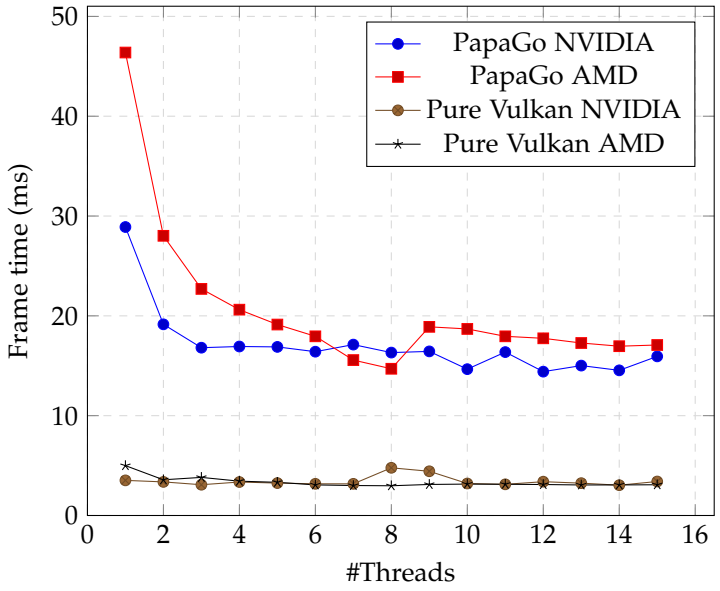

Figure 8.1: Results of variable amount of threads, when rendering a 30x30x30 grid of cubes in both Vulkan and PapaGo.

Figure 8.2 depicts how performance is impacted by the more GPU-bound case, where skulls are rendered with an increasing number of threads. This shows that the number of threads does not impact performance, when running Vulkan, but it does have a small impact with PapaGo. This indicates that, while the application is GPU-bound in Vulkan, the overhead introduced by PapaGo means that the application is still CPU-bound. Thus, multithreading has some impact.

Interestingly, while the frame time for running PapaGo on AMD as compared to Vulkan is double, the performance of PapaGo and Vulkan are about the same on NVIDIA. Thus, in this latter case the overhead of the API does not impact performance, instead the GPU sets the speed. We are not quite sure why PapaGo and Vulkan do not have similar performance.

In addition to test threading behavior, we also looked into how an increased load impacts performance.

Figure 8.3 shows how performance is impacted by increasing the number of cubes to draw on screen. All cases show a linear relationship between the number of cubes to draw and the frame time. We also see how PapaGo is outperformed to a larger degree by Vulkan, as the number of objects to draw is increased. For instance, at $45 \times 45 \times 45$ PapaGo on AMD runs around 50 milliseconds while Vulkan runs at about 15. Interestingly,

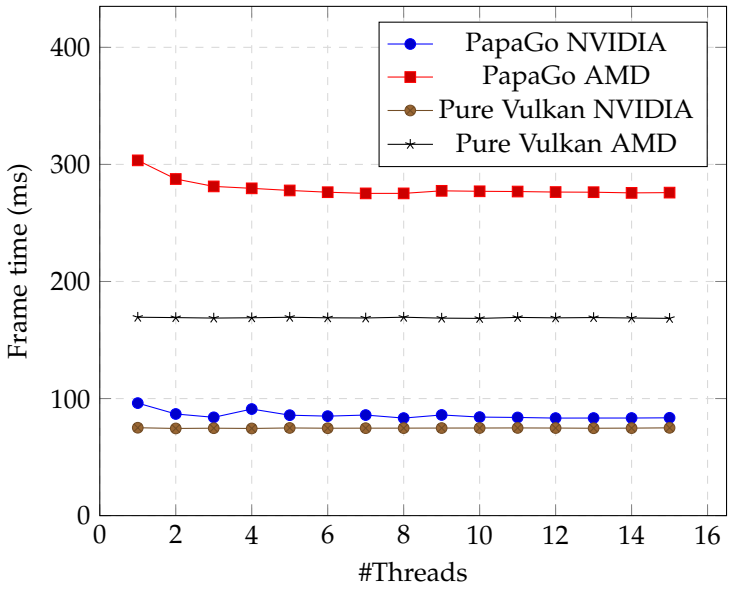

Figure 8.2: Results of variable amount of threads, when rendering a 30x30x30 grid of skulls in both Vulkan and PapaGo.

Vulkan on AMD runs much faster than Vulkan on NVIDIA at higher loads, however the performance of the two machines is much more similar in the PapaGo case. It seems some part of the implementation of PapaGo has diminished Vulkan's advantage on AMD.

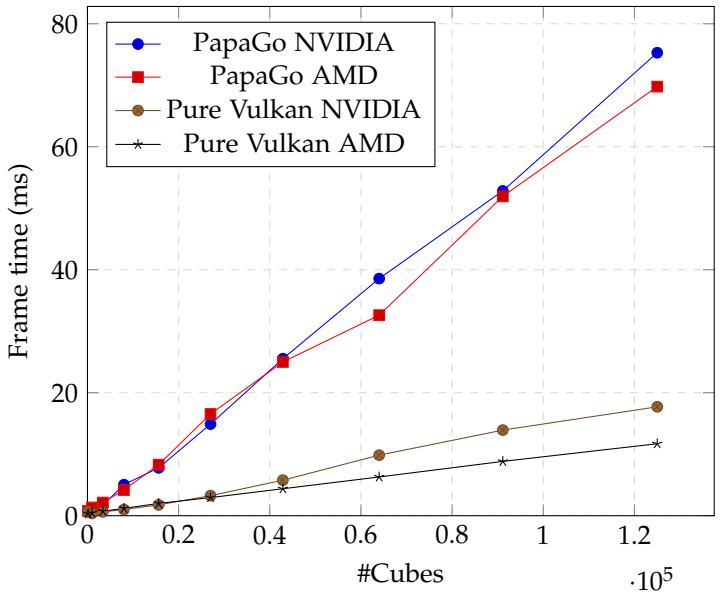

Figure 8.3: Results of increasing variable amount of cubes, when rendering a grid of cubes with optimal number of threads.

Figure 8.4 shows performance data for the test, where the number of skulls to draw is increased. Here we see that the difference in performance for PapaGo and Vulkan on AMD is not as great as in Figure 8.3. For instance, in the case where $45 \times 45 \times 45$ cubes are rendered, the frame time of PapaGo is three times larger than when running Vulkan on AMD, but it is only a third larger when rendering $45 \times 45 \times 45$ skulls.

However, the outlier present in the PapaGo plots does not appear in the Vulkan case. This may suggest an error in the benchmarking process, or perhaps the PapaGo implementation creates more cache misses on the GPU. As in Figure 8.2, the plots for PapaGo and Vulkan do not differ much.

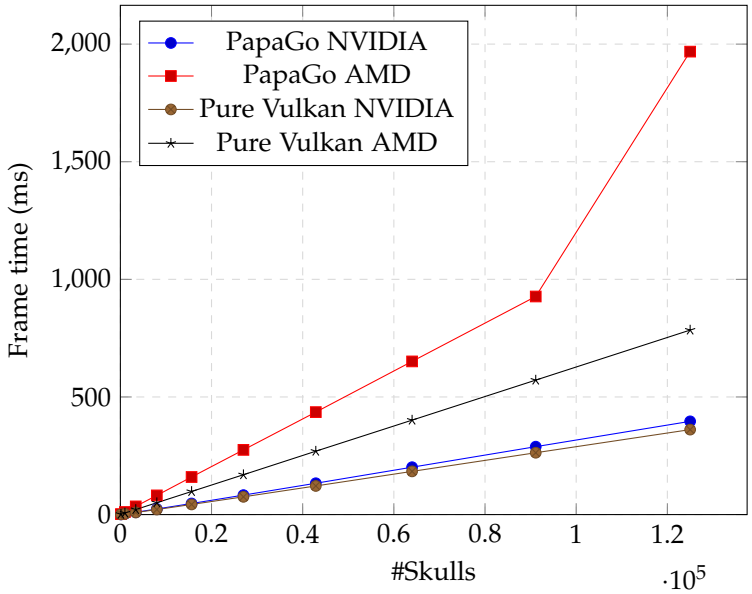

Figure 8.4: Results of increasing variable amount of skulls, when rendering a grid of skulls with optimal number of threads.

In total, the performance of PapaGo is impacted more by the addition of threads than when using Vulkan. However Vulkan always outperforms PapaGo in the CPU-bound case. In the GPU-bound case, Vulkan performs best on AMD, while the two APIs run at the same speed on NVIDIA.

8.3.2 Optimizing the Implementation

In light of comparing the performance of PapaGo and Vulkan, it is clear that PapaGo introduces some overhead on the CPU. We believe that much of the overhead is due to the following performance issues:

- The virtual functions used to hide complexity from the end user.
- Dynamic libraries are slower than static libraries.
- The way that we transition images to a standard layout regardless of its use.
- The clearing of the framebuffer is done through the command buffer instead of directly in the render pass.
- One memory chunk is allocated for each resource.

All of these points need to be looked into in order to speed up PapaGo. Increasing the performance of the API is crucial, if it is going to be used for serious projects.

In PapaGo, we hide the implementation from developers by having all implemented API classes inherit from abstract classes. This means that each call into the API is a virtual function call. **Virtual functions** are implemented in C++ as a v-table, which contains pointers to the actual implementations. This adds a layer of indirection when calling functions, which slows down performance. In addition the compiler does not know which function implementation is called at compile-time, which means that it cannot optimize code in the same manner as with non-virtual functions.

The main reason for compiling PapaGo to a **dynamic library** over a static library is that it allows us to change parts of the PapaGo API without requiring the developer to recompile their code. The performance loss will be mainly in the initialization phase, where all the symbols needs to be read from the dll. During the render loop of the program, where performance has been measured, most method calls are done through the virtual methods on classes instead of through the dll interface.

When handling Vulkan's image resources in our implementation, we make sure to **transition all images back to a known layout** after use. In Vulkan all images have a certain layout at a given point in time. Some layouts are more performant for some operations, and in some case they are even required. For instance, an image needs to be in one specific layout in order to be written to disk, another to be used as a render target and yet another to be a uniform texture. As we are never quite sure what operation has just been applied to an image, we always make sure to transition it back to the common *eGeneral* layout. This slows down performance, as our image transitions need to wait for all execution on the GPU to finish before changing the layout. To speed up performance, we must find away to not transfer back to *eGeneral* as frequently.

Because we wanted to add control instead of removing it, we wanted to allow the developer to **clear the frame- and depthbuffer through the command buffers**. However this is not the recommended approach when using Vulkan. Instead, it is recommended to use the render pass to clear the buffers, when the render target is loaded for use, which an almost free operation. This change could be implemented without much hassle, by allowing the developer to designate that a render target should be cleared automatically.

In PapaGo, the **allocation of memory is handled by the API**, and currently one Vulkan buffer and one area of device memory is created for each resource. This is sub-optimal as this causes more CPU overhead and cache misses, than if all resources were allocated on a single buffer with a single area of corresponding memory [28]. In this case, offsets would be used to access the different pieces of memory, which correspond to the different. resources.

8.4 Threats to Validity

In this section, we discuss threats to validity concerning both the benchmark of PapaGo and usability evaluation of the API. This will concern both internal validity, whether we can draw conclusions from the data, and external validity, whether the results of our work can be generalized outside of our tests.

8.4.1 Benchmark

The internal validity of our benchmark process is somewhat hampered, as the test application was run on a Windows system with some background processes popping up while data was recorded. Thus the application would slow down for periods, making the data unrepresentative of the application's true performance. We combated this in part by turning off Windows update, a common source of this behavior, but some outlier data was still produced. However by collecting many data points, over multiple runs, the variance of the data is lowered, and we come closer to calculating the true mean frame rate of each configuration.

On the issue of how much data to collect, the number of samples was calculated with the assumption that the data points follow some kind of normal distribution. We back this up by doing a pilot test and fitting the data to a bell curve, as shown in Section 7.1.2. While this shows the data to fit somewhat with a bell curve, we do fear that because of background processes increasing frame time, the data points are more likely to lay above the true mean rather than below.

There is also a problem, in that the number of samples computed for one configuration was used for all test configurations. This may not have been sound, as by looking at the individual data points, it seems that there is a greater variance when running the application at higher loads. Therefore it would have been better to have calculated the number of samples for each configuration. This would however have been rather time consuming, so an alternative would have been to make a dynamic recording system. This would keep collecting data until the calculated mean does not change within some given bounds.

During tests the name of the window in which the application output is displayed is updated at each second. This is not a cheap operation, and excluding it, we would most likely see an increase in performance. However this was also used when collecting performance data from Vulkan, so the data sets are still comparable.

There is also the issue whether our comparison of PapaGo and Vulkan is fair. It could be argued that as PapaGo should be used as an alternative to OpenGL, the comparison should occur between these APIs. In the CPU-bound case we also expect PapaGo to fare better against OpenGL than Vulkan. However we see Vulkan performance as an ideal, which PapaGo would reach if it introduced no overhead. Thus it is a good way to gauge the amount of overhead introduced by our API.

In terms of external validity, we cannot generalize the performance of PapaGo to professionally made games, as they often use different techniques to decrease CPU-boundness such as instancing. Thus the test application would not rely on only multithreading in an actual use case. However it does give us a gauge, of how much multithreading can help a CPU-bound application in a worst case.

8.4.2 Usability Evaluation

Internal validity of our usability evaluation is under pressure, as the participants became somewhat stressed during the evaluation. Especially Participant A and C had a tendency to freeze up or stop talking, as a lot of attention was pointed their way. This bared us from truly examining, how they would solve the tasks on their own. Alternatively we could have them do the tasks by themselves, but this would have hampered our ability to collect data and help them when necessary.

We must also admit that as time became short around the third task, where participants had to parallelize their program, facilitators became more eager to guide them than on the other tasks, to ensure that they finished. While they were still allowed to think for themselves, comparatively they were helped more in this case than the previous two. Thus the parallelization process may not be as straight forward, as we have otherwise reported.

There is also the issue of the documentation provided, which was not quite finished, when the participants received it beforehand. Afterwards it was completed before any test sessions began, but the difference in documentation may have caused some confusion to participants.

Looking at external validity, the use cases completed by the participants were rather simple, so participants never got to try out advanced features like using multiple render passes. Thus we can only conclude upon the usability, when the developer uses the core features of the API, and we do not know how well it works in serious graphics development. However as shown in Chapter 6, advanced applications like shadow mapping can be developed with PapaGo with no major problems introduced by the API itself.

Chapter 9

Conclusion

In this chapter, we conclude upon our efforts as showcased in this report. Afterwards we will discuss our future works.

At the beginning of this work, we posed the following problem statement:

How may we implement a stateless API on top of Vulkan, which removes unneeded complexity and supports easy parameter binding through runtime analysis of GLSL shaders? How may we investigate the programmability for this new API targeted at developers new to graphics programming?

From this we began an experimental process of designing an API, named PapaGo, in order to answer the statement. Through this process we set up four specific design requirements for the API, which were given as follows:

1. The API should be able to speed up CPU-bound applications
2. Parameter binding should be intuitive
3. Resource handling should be explicit but intuitive
4. The API should be stateless

An API specification was created to live up to these requirements, which in large part was based on Vulkan, but abstracted away many of its complexities. After having a finalized design for the API, it was implemented on top of Vulkan.

To evaluate whether the implemented API lived up to the requirements, we designed our own API evaluation method, as a variant of [31]. This method was originally developed for evaluating programming languages even before a compiler is implemented by having participants complete well defined programming tasks under supervision. Different from [31], the sample sheet given to participants was swapped with full API documentation, which was made available before the test sessions occurred. When analyzing data collected during sessions, we rate each usability problem discovered according to the scale defined in [42]. The data is further analyzed in terms of the 12 cognitive dimensions for API evaluation [17]. We present this method as part of our contributions of this report.

We used the above method on three participants, who were tasked with completing some basic tasks with PapaGo. From this we can conclude upon the API requirements.

In terms of having the API help with CPU-bound applications, we allowed *ISubCommand* buffers to be recorded to in parallel and provided a thread pool class to help the parallelization process along. While none of the participants were familiar with using explicit GPU commands, they quickly learned how to use them, and they were able to turn their serial programs parallel.

Parameter binding through GLSL shader analysis was in part intuitive, as this allowed participants to bind resources to parameters through names rather than locations. However they had some issues in recognizing what names to bind to, and the fact that parameter bindings were placed on the render state-encompassing *IRenderPass* object was surprising.

Looking at explicit resource handling, where resources like buffers and textures are handled through objects, the participants enjoyed the amount of control they had over their resources. Here they did not have to worry much about pointer arithmetic or memory padding. However they did have an issue in recognizing the purpose of *IDynamicUniformBuffer* resource objects.

Finally the API was made stateless by wrapping render state in the *IRenderPass* class rather than keeping it implicit as in OpenGL. While the participants had not experienced this before, they had no great issue in handling the concrete state.

In total, we can conclude that we lived up to the requirements stated from a design perspective.

Through the API evaluation process, we also identified some usability problems with PapaGo with four of them being severe. These involved developers not being able to switch between texture resources using commands, lack of cross-references in the API documentation and a *ISwapChain* object being required for off-screen rendering.

While speed was not the main issue of our efforts, we have tested the benchmarked performance of PapaGo and its implementation. This was done by constructing the same application and performing the same tests as in our earlier work [10, 8]. Two hardware setups were used in the tests, while they had similar CPUs, the first setup had an AMD GPU and the other a more powerful NVIDIA GPU.

The benchmark showed that in the more CPU-bound case, where many simple cubes are rendered, additional threads for command construction improve performance, especially on AMD. Yet, this impact is lessened in the GPU-bound case, where many complex models are rendered to screen.

Looking at how performance was affected by an increase in objects to draw, we found that the two hardware setups performed in the same manner in the more CPU-bound case. But testing out the GPU-bound case, the setup with the more powerful NVIDIA GPU outperformed AMD with a frame time, which was three times lower.

Comparing to results from our earlier work [10, 8], we see that the benchmark of PapaGo follows the same patterns as when benchmarking Vulkan. However in the CPU-bound case, PapaGo performs with worse than the equivalent Vulkan application. For instance, in the case where $45 \times 45 \times 45$ cubes are rendered, PapaGo on AMD runs at around 50 milliseconds while Vulkan runs at about 15 milliseconds. However this difference is much smaller in the GPU-bound case, where in the case with $45 \times 45 \times 45$ skulls, the frame time of PapaGo is only about a third more than Vulkan.

While this proves that our API includes some overhead, we have presented ways in which the implementation may be optimized. In addition, from our own experience it looks like the speed at which an application can be developed from scratch in PapaGo, when compared with Vulkan, can be a difference of magnitude.

In conclusion, the contribution of this report is a new graphics API, PapaGo, which we have evaluated on both usability and performance. The API has been found very usable and with good future potential, however future efforts must be put into speeding up its performance. In addition we have developed a new method for evaluating the usability of APIs.

9.1 Future work

With the report concluded, we dedicate the rest of this chapter to discuss possible future work in regard to our current efforts.

9.1.1 Improvements to PapaGo

During our own work with PapaGo and through the user evaluation process, we have noted some possible improvements, which could be made to the API.

Firstly, the API could be extended with new features, which are already present in Vulkan. Here we could allow developers to edit the fixed-function stages of the pipeline, or perhaps include other shader types like geometry, tessellation and compute shaders. This would add greatly to the PapaGo's flexibility. Especially the addition of compute shaders is interesting, as this would allow for the API to be used for both graphics and GPGPU programming.

With more time, we would also like to build upon the error logging system of PapaGo. During evaluation, we found that not enough errors were caught by the API, which made it more difficult to program in. We have begun to handle some errors, but more time would be needed to cover all cases in the code base.

For this project we have focused on providing a working API. To reach this goal, portability of PapaGo was of least concern, meaning that it is only available on Windows as of now. In the future, we would like to increase the portability of the API, so that it may run on other platforms like Linux and MacOS.

Lastly, as the API introduces some overhead onto Vulkan, we would like to expend some efforts to lower this. We do not assume that it will be possible to remove all overhead, however we already know of some parts of the implementation, which could be optimized.

9.1.2 Compare PapaGo with OpenGL

In this report, we compared PapaGo's performance with that of pure Vulkan, which showed that there are some problems with overhead. However it could be argued that, as PapaGo is at an abstraction level similar to OpenGL, a comparison between PapaGo and OpenGL would be more fair.

As opposed to Vulkan, we should be able to create some use cases, where PapaGo outperforms OpenGL. This would for instance be a situation, where a lot of the commands executed on the GPU can be reused between frames. Additionally, PapaGo should allow for faster parameter binding and state switches in comparison to OpenGL.

9.1.3 Discount Method for API Usability Evaluation

When setting up our user evaluations, we ended up creating our own method of evaluating APIs containing elements from [31], [17], and [42]. While this was an unexpected contribution of our work, we did experience that this method yielded useful results, allowing us to get a better understanding of PapaGo. In the future, we would like to formalize this method, so that it may be used by others. We would also like to use it to evaluate other APIs, as this would give use an insight into both the method and the APIs evaluated.

Bibliography

[1] AMD. *GPUOpen-LibrariesAndSDKs/Anvil*. URL: https://github.com/GPUOpen-LibrariesAndSDKs/Anvil/ (visited on 06/07/2018).

[2] Chad Anthony Austin and Dirk Reiners. "Renaissance: A functional shading language". PhD thesis. Iowa State University, 2005.

[3] Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

[4] Axel Blackert. *Evaluation of multi-threading in Vulkan*. 2016.

[5] Jasmin Blanchette. "The little manual of API design". In: *Trolltech, Nokia* (2008).

[6] Joshua Bloch. "How to Design a Good API and Why It Matters". Google Tech Talks. 2007.

[7] Antoine Bossard. "High-Performance Graphics in Racket with DirectX". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2017, pp. 814–825.

[8] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. "Comparing Direct3D 12 and Vulkan onPerformance and Programmability". in preperation for publication. 2018.

[9] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *data*. URL: https://github.com/dpw105f18/data (visited on 06/07/2018).

[10] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *Direct3D 12 vs. Vulkan: Comparing Low-level Graphics APIs on Performance and Programmability*. Aalborg University.

[11] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *PapaGo v0.0*. URL: https://github.com/dpw105f18/papago-api/releases/tag/v0.0-user_test (visited on 06/07/2018).

[12] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *PapaGo v0.1*. URL: https://github.com/dpw105f18/papago-api/releases/tag/v0.1-alpha (visited on 06/07/2018).

[13] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *PapaGoUsabilityTestWiki*. URL: https://github.com/dpw105f18/PapaGoUsabilityTestWiki (visited on 06/07/2018).

[14] Jens Breitbart. "CuPP-A framework for easy CUDA integration". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–8.

[15] Stan Brown. *How Big a Sample Do I Need?* 2013. URL: https://brownmath.com/stat/sampsiz.htm (visited on 06/07/2018).

[16] Nicholas J Chornay. "An OpenGL Backend for Halide". PhD thesis. Massachusetts Institute of Technology, 2013.

[17] Steven Clarke and Curtis Becker. "Using the cognitive dimensions framework to evaluate the usability of a class library". In: *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*. 2003.

[18] John M Daughtry, Umer Farooq, Jeffrey Stylos, and Brad A Myers. "API usability: CHI'2009 special interest group meeting". In: *CHI'09 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2009, pp. 2771–2774.

[19] Simon Dobersberger. "Reducing Driver Overhead in OpenGL, Direct3D and Mantle". In: *University of Applied Sciences Technikum Wien* (2015), pp. 2–32.

[20] Conal Elliott. "Programming graphics processors functionally". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM. 2004, pp. 45–56.

[21] Brian Ellis, Jeffrey Stylos, and Brad Myers. "The factory pattern in API design: A usability evaluation". In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 302–312.

[22] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. "How do api documentation and static typing affect api usability?" In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 632–642.

[23] Epic Games. *Unreal Engine 4 Documentation*. 2018. URL: https://docs.unrealengine.com/latest/INT/ (visited on 06/07/2018).

[24] Khronos Group. *Khronos OpenGL® Registry*. URL: https://www.khronos.org/registry/OpenGL/index_gl.php.

[25] Khronos Group. *OpenGL Shading Language*. URL: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

[26] Yong He, Tim Foley, and Kayvon Fatahalian. "A system for rapid exploration of shader optimization choices". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), p. 112.

[27] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. "Shader components: modular and high performance shader development". In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), p. 100.

[28] Chris Hebert. *Vulkan Memory Management*. 2016. URL: https://developer.nvidia.com/vulkan-memory-management (visited on 06/07/2018).

[29] Joshua Kinney. *3ds Max UV Mapping Fundamentals*. URL: https://www.pluralsight.com/courses/3ds-max-uv-mapping-fundamentals (visited on 06/07/2018).

[30] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174.

[31] Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. "Discount method for programming language evaluation". In: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM. 2016, pp. 1–8.

[32] Bryan Langley. *GPUs in the task manager*. 2017. URL: https://blogs.msdn.microsoft.com/directx/2017/07/21/gpus-in-the-task-manager/ (visited on 06/07/2018).

[33] *Learning Vulkan*. 1st ed. 35 Livery Street, Birmingham, United Kingdom: Packt Publishing Ltd., 2016. ISBN: 978-1-78646-980-9.

[34] Marc Levoy. *Standford Bunny*. 1994.

[35] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. "Cg: A system for programming graphics hardware in a C-like language". In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 896–907.

[36] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. "Shader algebra". In: *ACM Transactions on Graphics (TOG)* 23.3 (2004), pp. 787–795.

[37] Michael D McCool, Zheng Qin, and Tiberiu S Popa. "Shader metaprogramming". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association. 2002, pp. 57–68.

[38] Micosoft. *Direct3D*. URL: https://msdn.microsoft.com/da-dk/library/windows/desktop/hh309466(v=vs.85).aspx (visited on 06/07/2018).

[39] Microsoft. *HLSL*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561\%28v=vs.85\%29.aspx?f=255&MSPPError=-2147217396.

[40] Brad A Myers and Jeffrey Stylos. "Improving API usability". In: *Communications of the ACM* 59.6 (2016), pp. 62–69.

[41] John Nickolls and William J Dally. "The GPU computing era". In: *IEEE micro* 30.2 (2010).

[42] Jakob Nielsen. "Severity ratings for usability problems". In: *Papers and Essays* 54 (1995), pp. 1–2.

[43] Nvidia. *nvpro-pipeline/VkHLF*. URL: https://github.com/nvpro-pipeline/VkHLF (visited on 06/07/2018).

[44] Nathaniel Nystrom, Derek White, and Kishen Das. "Firepile: run-time compilation for GPUs in scala". In: *ACM SIGPLAN Notices*. Vol. 47. 3. ACM. 2011, pp. 107–116.

[45] Mikael Olofsson. *Direct3D 11 vs 12: A Performance Comparison Using Basic Geometry*. 2016.

[46] OpenSceneGraph. *The OpenSceneGraph Project Website*. 2017. URL: http://www.openscenegraph.org/ (visited on 06/07/2018).

[47] Portia O'Callaghan. *The api walkthrough method*. 2010.

[48] Zheng Qin. "An embedded shading language". MA thesis. University of Waterloo, 2004.

[49] Ofer Rosenberg. *Introduction to GPU Architecture*. 14/11/2011. URL: http://haifux.org/lectures/267/Introduction-to-GPUs.pdf.

[50] Adrian Sampson. "Let's Fix OpenGL". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[51] Andrew Stromme, Ryan Carlson, and Tia Newhall. "Chestnut: A Gpu programming language for non-experts". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2012, pp. 156–167.

[52] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842.

[53] Moisés Viñas, Zeki Bozkus, and Basilio B Fraguela. "Exploiting heterogeneous parallelism with the Heterogeneous Programming Library". In: *Journal of Parallel and Distributed Computing* 73.12 (2013), pp. 1627–1638.

[54] Joey de Vries. *Learn OpenGL*. URL: https://learnopengl.com (visited on 06/07/2018).

[55] Wikipedia. *Direct3D*. 2018. URL: https://en.wikipedia.org/wiki/Direct3D (visited on 06/07/2018).

[56] Wikipedia. *Mantle (API)*. 2018. URL: https://en.wikipedia.org/wiki/Mantle_(API) (visited on 06/07/2018).

[57] Wikipedia. *Metal (API)*. 2018. URL: https://en.wikipedia.org/wiki/Metal_(API) (visited on 06/07/2018).

[58] Wikipedia. *Texture Mapping Unit*. URL: https://en.wikipedia.org/wiki/Texture_mapping_unit (visited on 06/07/2018).

[59] Wikipedia. *Vulkan (API)*. 2018. URL: https://en.wikipedia.org/wiki/Vulkan_(API) (visited on 06/07/2018).

Appendix A

Initial Use Cases

The following listing are the ones we showed to Martin Kraus as part of our Peer Review to get feedback for our initial design. The shaders coded against are not present in this chapter, however they can be found in [9].

A.1 Drawing several cubes

```
class Object{
  mat4 model;
  texture tex;
  vec3 positions[];
  vec3 color[];
  vec3 uv[];
  vec3 indices[];
}

std::vector<Object> objects = // Initializer list of objects to draw
auto surface = // Get surface reference
mat4 view = // view matrix
mat4 projection = // projection matrix

auto mainPass = RenderPass("main.vert", "main.frag");
Renderer rend = Renderer(surface, 3, FIFO_BUFFERING);

Sampler sampler();
sampler.SetMagFilter(NEAREST);
sampler.SetMinFilter(NEAREST);
sampler.SetTextureWrapS(CLAMP_TO_EDGE);
sampler.SetTextureWrapT(CLAMP_TO_EDGE);

while(true) {
  rend.SetRenderpasses({mainPass});

  auto cbuf1 = PrimaryCommandBuffer(mainPass);
  cbuf1.setPrimitiveTopology(TRIANGLES);
  cbuf1.clearFrameBuffer(0.0f, 0.0f, 0.0f, 1.0f);
  cbuf1.clearDepthBuffer();

  auto sBuf1 = SecondaryCommandBuffer();
  sBuf1.setUniform("projection", projection);
  sBuf1.setUniform("view", view);

  for(auto& obj : objects){
    sBuf1.setUniform("model", obj.model);
    sBuf1.setUniform("texture", obj.tex, sampler);
    sBuf1.setInput("inPosition", obj.position);
    sBuf1.setInput("inColor", obj.color);
    sBuf1.setInput("inTexCoord", obj.uv);
    sBuf1.setIndexBuffer(obj.indices);
    sBuf1.drawInstanced(obj.indices.size(), 1, 0, 0);
  }
  cbuf1.attatchBuffers({sBuf1});
  mainPass.setCommands({cbuf1});
  rend.renderToScreen(mainPass.output["outColor"]);
}
```

Listing A.1: Initial use case for drawing cubes.

A.2 Drawing several cubes in parallel

```
class Object() {
  mat4 model;
  texture tex;
  vec3 positions[];
  vec3 color[];
  vec3 uv[];
  vec3 indices[];
}

ThreadPool thread_pool(5);
std::vector<Object> objects = // Initializer list of objects to draw
auto surface = // Get surface reference
mat4 view = // view matrix
mat4 projection = // projection matrix
auto mainPass = RenderPass("main.vert", "main.frag");
Renderer rend = Renderer(surface, 3, FIFO_BUFFERING);
Sampler sampler();
sampler.SetMagFilter(NEAREST); sampler.SetMinFilter(NEAREST);
sampler.SetTextureWrapS(CLAMP_TO_EDGE); sampler.SetTextureWrapT(CLAMP_TO_EDGE);

while(true) {
  rend.SetRenderpasses({mainPass});

  auto cbuf1 = PrimaryCommandBuffer(mainPass);
  cbuf1.setPrimitiveTopology(TRIANGLES);
  cbuf1.clearFrameBuffer(0.0f, 0.0f, 0.0f, 1.0f);
  cbuf1.clearDepthBuffer();
  auto sBuf1 = SecondaryCommandBuffer();
  sBuf1.setUniform("projection", projection);
  sBuf1.setUniform("view", view);

  for(auto i = 0; i < thread_pool.thread_count(); ++i) {
     sBufs[i] = SecondaryCommandBuffer(mainPass);
  }

  thread_pool.for_each(objects, [&sBufs](Object& object, size_t thread_index){
  auto& buf = sBufs[thread_index];
  buf.setUniform("model", obj.model);
    buf.setUniform("texture", obj.tex, sampler);
    buf.setVertexBuffer(obj);
    buf.setIndexBuffer(obj.indices);
    buf.drawInstanced(obj.indices.size(), 1, 0, 0);
  });
  thread_pool.wait();
  cbuf1.attatchBuffers({sBuf1});
  mainPass.setCommands({cbuf1});
  rend.renderToScreen(mainPass.output["outColor"]);
}
```

Listing A.2: Initial use case for drawing cubes in parallel.

A.3 Shading a cube with shadow mapping

```
class object(){
  mat4 depthMVP; mat4 MVP; mat4 depthBiasMVP;
  vec3 positions[]; vec3 color[]; vec3 indices[];
}

std::vector<Object> objects = // Initializer list of objects to draw
auto surface = // Get surface reference

auto shadowPass = RenderPass("shadow.vert", "shadow.frag");
auto mainPass = RenderPass("main.vert", "main.frag");
Renderer rend =  Renderer(surface, 3, FIFO_BUFFERING);

Sampler sampler();
sampler.SetMagFilter(NEAREST); sampler.SetMinFilter(NEAREST);
sampler.SetTextureWrapS(CLAMP_TO_EDGE); sampler.SetTextureWrapT(CLAMP_TO_EDGE);

while(true) {
  rend.SetRenderpasses({shadowPass, mainPass});

  auto cbuf1 = PrimaryCommandBuffer(shadowPass);
  cbuf1.setPrimitiveTopology(TRIANGLES);
  cbuf1.clearFrameBuffer(0.0f, 0.0f, 0.0f, 1.0f);
  cbuf1.clearDepthBuffer();

  auto sBuf1 = SecondaryCommandBuffer();
  for(auto& obj : objects) {
    sBuf1.setUniform("depthMVP", obj.depthMVP);
    sBuf1.setInput("position", obj.position);
    sBuf1.setIndexBuffer(obj.indices);
    sBuf1.drawInstanced(obj.indices.size(), 1, 0, 0);
  }
  cbuf1.attatchBuffers({sBuf1});
  shadowPass.setCommands({cbuf1});

  cbuf2 = PrimaryCommandBuffer(mainPass);
  cbuf2.setPrimitiveTopology(TRIANGLES);
  cbuf2.clearFrameBuffer(0.0f, 0.0f, 0.0f, 1.0f);
  cbuf2.clearDepthBuffer();
  cBuf2.setUniform("shadowMap", shadowPass.output["fragmentdepth"], sampler);

  auto sBuf2 = SecondaryCommandBuffer();
  for(auto& obj : objects) {
    sBuf2.setUniform("MVP", obj.MVP);
    sBuf2.setUniform("depthBiasMVP", obj.depthBiasMVP);
    sBuf2.setInput("position", obj.position);
    sBuf2.setInput("color", obj.color);
    sBuf2.indexBuffer(obj.indices);
    sBuf2.drawInstanced(obj.indices.size(), 1, 0, 0);
  }
  cbuf2.attatchBuffers({sBuf2});
  mainPass.setCommands({cBuf2});

  rend.renderToScreen(mainPass.output["outColor"]);
}
```

Listing A.3: Initial use case for shadow mapping

Appendix B

Interview Questions

This chapter contains questions asked to our participants before and after executing tasks using PapaGo.

B.1 Questions before executing tasks

- Pre-test interview
- What education do you have?
- What work experience do you have in programming?
- Which programming languages have you worked in?
- How much experience do you have in graphics programming?
  - Which APIs have you utilized?
    - Any experience with Vulkan or Direct3D 12?

B.2 Questions after executing tasks

- What did you think of the tasks?
  - Level of difficulty?
- What's your first impression of the PapaGo API?
- Would you like to keep using PapaGo or use another API?
- What was good in PapaGo?
- What should have been better in PapaGo?
- Cognitive Dimensions Questionnaire
  - Abstraction level
    - Did the API provide adequate control? Too much?
    - What was easiest to do?
    - What was most difficult?
  - Learning style
    - Would you be able to use this API without documentation?
    - Do you think you could learn it by exploration alone?

    * What was the most useful resource to learn the API?
- Working framework
    * Did you have to think about too many different elements at once?
    * How many elements did you feel you needed to keep in mind during coding? Which elements?
- Work-step unit
    * How much work did you have to do in order to enact a change in the program?
    * Was this a fitting amount of work?
- Progressive evaluation
    * Did you feel that you could execute the application during coding in order to evaluate your progress?
    * What did you think of the feedback given through the API?
- Premature commitment
    * What did you think of the dependency between API objects?
    * Did you ever have to make a decision before all information was available?
- Penetrability
    * How easy was it to explorer the API features?
    * How easy was the API to understand?
    * How did you go about retrieving what you needed to solve the tasks?
- API Elaboration
    * Any features, which were missing?
    * How would you expand upon the API?
- API Viscosity
    * How difficult was it to make changes to the system, when parts of it were already coded?
- Consistency
    * After drawing a triangle, could you infer the remaining features of the API? How?
- Role Expressiveness
    * Was it ever difficult to tell what the role of different classes and methods were? Which?
- Domain Correspondence
    * Were you previously familiar with the terminology of the API (commands, command buffers, queues etc.)?
    * Does the naming of these elements make sense?
    * Would you like to change any of the names? Which and why?

Appendix C

Defined Usability Problems

This chapter contains all the identified usability problems of PapaGo along with their severity rating.

ID	Description	Participants	Severity
1	Entry point for test setup took a while to discover.	A, C	0
2	IDevice was not immediatly reconizable as the central class.	A, C	2
3	Code examples from wiki were needed to instantiate ISurface and IDevice using ISurface:createWin32Surface and IDevice::enumerateDevices respectivly.	A, C	2
4	Setting up shaders for use required more steps than expected.	A,C	2
5	How to make draw calls, through commands, was not immediatly obivous.	A,C	2
6	Wrapping all created objects in unique pointers created issues when passing objects around the code.	A,B,C	2
7	Parser::CompileVertexShader takes a string containing GLSL source code as trhough its "source" parameter, but participant expected to input the path of a file containing the code..	A	1
8	It was unexpected that Parser::CompileVertexShader takes a string with the name of the shader entry point through its "entryPoint" parameter.	A	1
9	It was surprising that vertex, index and uniform buffers were all IBufferResources.	A	0
10	The usage of IShaderProgram after its construction was not clear.	A, C	2
11	The role of IRenderPass was not immediatly obivous.	A, B	3
12	It was possible to pick inappropriate formats for color and depth/stencil buffers.	A, B	2
13	It was not obivous that resources needed to be bound to shaders through an IRenderPass object.	A, B	3
14	It takes some time to become familiar with the concept of commands for enacting GPU instructions.	A, C	2
15	The term swap chain as used for multi-buffering was not recognizable.	A, C	2
16	The difference between ICommandBuffer and ISubCommandBuffer was not immediatly obivous.	A	1
17	The relationship between command buffers and recorders are not obvious.	A	1
18	Using a C++ lambda expression for recording command buffers was surprising.	A, B, C	1
19	The documentation in the wiki was necessary to proceed.	A, C	1
20	The usage of modern C++ features and the standard library hampers portability.	B	0
21	Shaders can only be compiled at runtime.	B	0
22	The swap chain extension on IDevice could be disabled, while an ISwapchain is required to create an IGraphicsQueue.	B	4

23	It was surprising that the "frameCount" parameter of IDevice::CreateSwapChain was of the type size_t rather than a more restrictive enum.	B	1
24	The meaning of the entry ‘eMailbox’ in the IDevice::Presentmode enum was not clear.	A, B, C	2
25	At first it is thought that images can be presented directly from an ISwapChain rather than through commands.	B,C	1
26	The usage of generics in the API can lead to a larger executable size.	B	0
27	When creating an IRenderPass, there was no option for offsetting the viewport rendered to.	B	1
28	It is possible to record command buffers, where the buffer formats of the IRenderPass and ISwapChain used do not match. This leads to a crash.	B	2
29	The difference between a IUniformBuffer and a IDynamicUniformBuffer was not obivous. It was thought that dynamic meant that the buffer could be resized at runtime.	B, C	3
30	It was unexpected that an ICommandBuffer, rendering to an ISwapChain, had to be recorded in the render loop, as to have the swap chain swap its buffers.	B,C	3
31	A lack of error messages meant that participants would get stuck without outside help.	A, B, C	4
32	It was not obivous that the API automatically pads memory between elements in IDynamicUniformBuffer objects.	B	0
33	IBufferResources cannot be resized at runtime.	B	0
34	An interface block in a GLSL shader may only contain one entry.	B	3
35	It is not possible to bind new textures to shaders within the same command buffer.	B	4
36	The wiki documentation was lacking explanations of core concepts like commands, and contained no cross-referencing between pages.	A,B, C	4
37	By using C++ lambda expressions for recording commands, it was expected that parallelization of command construction occured automatically.	B	1
38	Compute shaders were not avaliable.	B, C	2
39	It was not obivous, where some objects, like IShaderProgram, needed to be used as input parameters.	A, B, C	2
40	Participant attemped to use IParser statically for shader compilation.	B	0
41	The relationship between the "name" parameter of IRenderPass::BindResource and the name of a shader uniform was not immediately obvious.	B,C	2
42	The only backend avlaiable for the API is Vulkan.	B	0
43	The role of an IParser instance as a GLSL compiler was not obious.	C	1
44	The purpose of the ISurface class was not obivous, resulting in participants using window dimensions rather than surface dimensions to define their render targets.	A, B, C	1
45	The usage of the IGraphicsQueue was not immediatly obivous.	A, B, C	2
46	The PresentMode enum was difficult to find, as it was located on the IDevice class rather than in global scope.	C	1
47	The IRecordingCommandBuffer::execute method could be seen as executing sub commands rather than recording them.	C	1
48	The name of the ISampler class could be mistaken for a combined image sampler, as found in GLSL.	C	1
49	When drawing multiple objects, it was not obvious that a model matrix needed to be in a dynamic uniform buffer, while the view projection matrix needed to be in a regular uniform buffer.	C	1
50	The purpose of IRecordingSubCommandBuffer ::setDynamicIndex was not immediately obvious.	B, C	3

51	3D textures are not supported.	C	2
52	Some buffer resources, like vertex buffers, have data uploaded to them at creation, while other buffers, like uniform buffers, do not.	A	2

Table C.1: List of usability problems found during our test sessions. Each problem is rated by its severity.

Appendix D

Shadow Map Shaders

This chapter contains shader code, used for implementing the shadow mapping example in Section 6.2.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 pos;
layout(location = 1) in vec2 uv;

layout(binding = 0) uniform VpMatrix {
  mat4 vp;
} view_proj;

layout(binding = 1) uniform ModelMatrix {
  mat4 m;
} model;

layout(location = 0) out vec4 out_pos;

void main() {
  gl_Position = view_proj.vp * model.m * vec4(pos, 1.0f);
  out_pos = gl_Position;
}
```

Listing D.1: Vertex shader used to make shadow mapping, pass one.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec4 pos;

layout(location = 0) out vec4 color;

void main() {
  color = vec4((pos.xyz / pos.w) * 0.5f + 0.5f, 1.0f);
}
```

Listing D.2: Fragment shader used to make shadow mapping, pass one.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 pos;
layout(location = 1) in vec2 uv;

layout(binding = 0) uniform UniformBufferClass{
    mat4 view_projection;
} ubo;

layout(binding = 2) uniform SomeName{
  mat4 shadow_view_projection;
} sn;

layout(binding = 1) uniform UniformInstanceClass{
    mat4 model;
} uio;

layout(location = 0) out vec2 texture_coord;
layout(location = 1) out vec4 shadow_coord;

void main(){
    gl_Position = ubo.view_projection * uio.model * vec4(pos, 1.0);
    texture_coord = uv;
    shadow_coord = sn.shadow_view_projection * uio.model * vec4(pos, 1.0);
}
```

Listing D.3: Vertex shader used to make shadow mapping, pass two.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 2) uniform SomeName{
  mat4 shadow_view_projection;
} sn;

layout(binding = 3) uniform sampler2D tex;
layout(binding = 4) uniform sampler2D shadow_map;

layout(location = 0) in vec2 texture_coord;
layout(location = 1) in vec4 model_pos;

layout(location = 0) out vec4 color;

float bias = 0.005f; //ori: 0.005

void main(){

  vec2 shadow_uv_coord = (model_pos.xy / model_pos.w) * 0.5f + 0.5f;
  vec4 shadow = texture(shadow_map, shadow_uv_coord);
    float visibility = shadow.r < (model_pos.z / model_pos.w) - bias
        ? 0.3 : 1.0;
    color = texture(tex, texture_coord) * visibility;
}
```

Listing D.4: Fragment shader used to make shadow mapping, pass two.