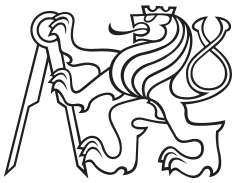
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Designing a modern high-level graphics API

Roman Galajda

Supervisor: Ing. Jaroslav Sloup
Field of study: Open Informatics
Subfield: Computer Graphics
January 2020

Acknowledgements

I would like to give thanks to Ing. Jaroslav Sloup for his supportive attitude and valuable insights. A special thanks goes to Bohemia Interactive Simulations k.s. for sponsoring the development of this project and to The Khronos Group, Inc. for introducing the Vulkan API, without which this thesis wouldn't have been possible.

Declaration

I declare that all parts of this thesis have been written by myself and that I mentioned all sources of information and the literature that has been used, in accordance with the guideline for adhering to ethical principles for elaborating an academic final thesis.

In Prague, 6. January 2020

.....

Abstract

The introduction of Vulkan [13], a low-level graphics API, in 2016 has presented an opportunity for developers to leverage the rendering and compute capabilities of modern graphics cards in a high performance, cross-platform manner. However, its use comes at significantly higher development costs compared to its high-level predecessor, OpenGL. This thesis presents an interface which aims to reduce those costs while maintaining the advantages of Vulkan, striking a balance in the level of abstraction it exposes. Building upon Vulkan to send commands to the graphics device, its goal is to allow the user to express intent without superfluous details and yet not stand in the way between the code and the hardware when high performance and control are needed. This interface has been designed, implemented and evaluated against both OpenGL and Vulkan in terms of performance, memory use and verbosity of code.

Keywords: Tephra, Vulkan, Computer Graphics, Graphics API, GPU

Supervisor: Ing. Jaroslav Sloup

Abstrakt

Příchod grafického, nízko-úrovňového grafického rozhraní Vulkan [13] v roce 2016 přinesl pro developery příležitost využít vykreslovací a výpočetní schopnosti dnešních grafických karet na mnoha platformách a s vysokým výkonem. Pro jeho použití je ovšem třeba vynaložit značně větší úsilí, než pro jeho předchůdce OpenGL. Tato práce předkládá rozhraní, které zjednodušuje vývoj v porovnání s rozhraním Vulkan, pro které je nadstavbou, ale zároveň se snaží zachovávat jeho výhody. Jeho cíl je umožnit uživatelům vyjádřit svůj záměr bez přebytečných detailů, ale tak, aby nestálo v cestě mezi kódem a grafickým zařízením když je vysoký výkon a kontrola za potřebí. Toto rozhraní bylo navrženo, naimplementováno a porovnáno oproti OpenGL a Vulkan z hlediska výkonu, paměťových nároků a velikosti kódu.

Klíčová slova: Tephra, Vulkan, Počítačová grafika, grafické API, GPU

Překlad názvu: Návrh moderního vysoko-úrovňového grafického API

Contents

1 Introduction 1

2 Related work 3

2.1 Benefits of Vulkan over OpenGL . 3

2.2 Benefits of high-level abstraction libraries around Vulkan 4

2.3 A look at existing libraries 5

3 Interface design 7

3.1 General concepts 7

3.1.1 Object lifetime 8

3.1.2 Handling of arrays 9

3.1.3 Threading behavior 10

3.1.4 Validation and debugging . . . 10

3.2 Initialization 11

3.3 Shader interface and descriptor layout 13

3.4 Pipelines 16

3.5 Render passes 18

3.6 Memory management 22

3.7 Images and buffers 24

3.8 Descriptor sets 25

3.9 Execution model in Vulkan 26

3.9.1 Synchronization 27

3.9.2 Frame graphs 29

3.10 Jobs 30

3.10.1 Command lists 31

3.10.2 Transient resources and aliasing 32

3.10.3 Job lifetime and synchronization 33

3.10.4 Job examples 34

3.11 Surface and swapchain 36

4 Implementation details 37

4.1 Jobs and job-local resources 39

4.1.1 Job-local resource allocation . 40

4.1.2 Preinitialized buffer allocation 41

4.1.3 Descriptor set allocation 42

4.2 Automatic synchronization 42

4.2.1 Buffer access maps 43

4.2.2 Image access maps 44

4.2.3 Queue state 45

4.2.4 Barrier list 45

4.2.5 Job compilation and submission 46

4.2.6 Render pass synchronization . 47

5 Evaluation **49**

5.1 The tests 49

5.2 The results 52

5.3 Synchronization analysis 56

6 Conclusion **59**

A Bibliography **61**

B Contents of the included CD **63**

C Project Specification **65**

Chapter 1

Introduction

While consumer graphics hardware and the demands put on it have been evolving significantly over the past 25 years, both growing more and more complex, the graphics APIs had a hard time catching up without breaking compatibility and abandoning their previous design philosophies. Over time, GPU drivers have accumulated additional layers that through various heuristics and guesswork attempted to translate the high-level commands as received through the aging APIs into performant low-level instructions for the modern GPU architectures. This involuntary level of indirection makes it difficult for the user to see what is really happening, causes non-obvious performance issues and requires the user to learn what the "fast paths" are in the API for modern architectures. Even then the interface itself might be the limiting factor, due to the lack of multithreading support and other features.

This has recently changed, with the introduction of the Vulkan API [13], allowing the user to communicate with the underlying hardware much more directly, without the driver and oddities of the API getting in the way. While the performance has improved and the interface is smaller compared to the previous APIs, being this low level has its disadvantages, too. The implementation of even the simplest example is very verbose, especially if it needs to work well on all platforms. Several areas previously handled by the driver now practically require the user to write an abstraction for themselves, rather than using the API directly, such as for memory allocation. Various concepts like synchronization and image layout transitions are difficult to grasp and easy to make errors in, but are also possible to be automated, given enough information from the user.

While this thesis presents a brief introduction of the relevant concepts in Vulkan, a quick overview of the main features of the Vulkan API can be found at `https://renderdoc.org/vulkan-in-30-minutes.html`.

This project aims to design a high-level graphics and computing API with modern GPU architectures and use cases in mind, while avoiding the disadvantages of the existing APIs. This is only possible thanks to Vulkan providing a performant low-level interface that can be built upon freely. It is not meant to replace Vulkan for uses where its explicitness and level of control are paramount, neither is it supposed to replace OpenGL, where fast prototyping and an already established community outweigh its downfalls. Instead, its aim is to fill the hole somewhere in the middle, for the projects that require a high performance solution, but whose developers wish to avoid the steep learning curve and large time investment associated with the Vulkan API. Since this library stems from Vulkan and wishes to make its concepts accessible to a wider audience, it has been aptly named "Tephra," the geological name for "rock fragments and particles ejected by a volcanic eruption."

Chapter 2

Related work

Several graphics libraries that form an abstraction layer around Vulkan have been written already. To be able to compare between them and point out their advantages and shortcomings, we first need to consider the reasons why we might want to use an abstraction library over Vulkan in the first place, rather than use the API itself. On the other hand of the spectrum, we should also recognize the benefits Vulkan has over an API like OpenGL. An ideal abstraction would preserve all of Vulkan's benefits while resolving all the issues it currently has.

2.1 Benefits of Vulkan over OpenGL

Vulkan, as a low level API, resides closer to the hardware, exposing its functionality more directly and with less abstractions than previous interfaces. This means the driver implementations are simple and thin, generally offering lower CPU overhead. Due to the interface being designed in a way that provides the driver all the information it needs to know, the drivers can avoid relying on various heuristics to guess what the user's intents are and what they will do in the future. Thanks to explicit synchronization and defining constructs ahead of time, less work needs to be done by the driver during actual rendering. It features an optional validation layer that can be turned off when the product is shipped, saving the implementation from checking whether all arguments passed to the API calls are correct at that time. The explicit API design makes it more obvious what the hardware is actually doing and what operations are slow. There are no hidden copies or synchronization

points. In some ways it is also more straightforward than OpenGL. Things can generally only be accomplished one way. For example, there are only 4 drawing functions in Vulkan, as opposed to 19 in OpenGL.

The API is designed with multithreading support in mind, not having monolithic objects that keep state and restrict parallelism where it's important, notably for pipeline compilation and preparing command buffers, both of which are time intensive and easily parallelizable. Before shaders get compiled in pipeline state objects, they need to be pre-compiled into the SPIR-V bytecode, which is easier and faster to parse than human readable GLSL code. Pipeline state is created ahead of time and explicitly. This prevents sudden framerate drops when previously unseen shaders and pipeline state combinations get used for rendering. In OpenGL, they have to be automatically compiled by the driver just in time before the rendering of the frame takes place.

Explicit memory management means related resources can reside in the same memory allocation. Images and buffers with non-overlapping usage can even share the same memory space, leading to a reduced memory footprint. The user of the API also knows how much GPU memory is used by the application. OpenGL sometimes duplicates resources to maintain the illusion of an immediate API. Thanks to explicit management of resources like command buffers, synchronization primitives and descriptor sets, their instances can be repurposed once not in use without an unnecessary destroy-create cycle. An increasing number of GPU architectures use tiled rendering, the advantages of which can be leveraged through render passes, allowing large reductions in memory bandwidth in such architectures through pixel-local storage.

2.2 Benefits of high-level abstraction libraries around Vulkan

There are several reasons why using an abstraction library over Vulkan is beneficial. Vulkan is a very verbose API and even simple demos often require more than a thousand lines of code. Some of its concepts are also hard to learn and understand, as evidenced by the frequent questions asked in the community. For those reasons, for anything more complex than demos and simple applications, it has become very common and even mandatory to implement abstractions over some parts of Vulkan. Most notably, the handling of synchronization, inserting barriers and image layout transitions where necessary. All of those require knowledge of previous and upcoming commands as well as deep understanding of the synchronization principles in

Vulkan. Using it directly is hard to get correct and error prone to maintain. Suballocating memory from larger chunks is needed for performance reasons, which often leads to using a general purpose heap allocator. Wrappers and handlers around the Vulkan API as a whole have its merit as well, reducing the amount of code that needs to be written, simplifying debugging, enabling easier platform support and ensuring the proper lifetime of Vulkan handles.

As is often said, no size fits all. Every project would benefit from writing its own abstraction around Vulkan aimed specifically for its own use of the API. That is not time efficient in practice, however. Many projects are better off using a general abstraction over Vulkan than using Vulkan directly, even if it does not suit its needs perfectly. Several such abstraction libraries already exist at varying levels of scale, focus and stages of development.

2.3 A look at existing libraries

V-EZ [3] is an abstraction library around Vulkan developed by AMD, significantly simplified in an effort to make Vulkan more accessible. Many parts of Vulkan are completely hidden, serving as a good starting point for people to learn the basics of the API and has its use for simple applications. It is, however, still a C API and requires explicit resource lifetime management without RAII concepts. The simplification doesn't come without sacrifices either. Descriptor sets and pipelines are created on the fly, potentially leading to unexpected delays during rendering. Image layout transitions and barriers are not optimal due to missing ways for the user to communicate future accesses to resources. Various resource pools are abstracted away, causing unexpected locking in multithreaded code. It exposes no concept of render passes, resulting in reduced performance on tiled renderers. It is also automatically resolving multisampled images, rather than letting the user pick the right moment for it. Overall, this library loses many of the performance benefits of Vulkan, while the deceptively simplified API still leaves some non-obvious responsibilities to the user. It also doesn't ask the user for useful information, leading to less than optimal execution on the device as well. The value was put more on ease of use, willing to sacrifice performance to achieve it.

DiligentEngine [6] provides a low level abstraction around all the main graphics APIs with a focus on the modern, low level APIs. Even then it had to make several sacrifices to be able to do so, targeting the lowest common denominator of multiple graphics APIs. This manifests mostly as a loss of control, specifically over how and where resources are created and where

workloads are executed. For example asynchronous compute doesn't seem to be implicitly supported. Due to the differences in shader interface between APIs, you can't layout descriptor sets and constant buffer blocks arbitrarily. Pipeline compilation can only be single threaded. Similarly to V-EZ, it does not expose render passes and does not ask the user enough information about future usage to insert barriers efficiently. Its automatic barrier insertion can be selectively disabled in exchange for being able to record commands in parallel, since resource state is tracked globally, so one has to choose one or the other. DiligentEngine is a well made and useful library, especially when also targeting platforms that do not support Vulkan. When the user can afford to focus only on platforms that support Vulkan as an API, there is still room for improvement.

Several other libraries exist, which aim to make using Vulkan easier without providing any high level abstractions. Vulkan-Hpp [12], Anvil [1] and Vpp [9] are examples of such. They are low level APIs on their own, building upon Vulkan only through thin layers. Where this is desirable and high levels of control is required, these libraries provide at least some comfort over using Vulkan directly, with very little compromises. However, the scope of this project is to design a high level API, so it is not comparable.

There are many other libraries in the Vulkan ecosystem, but they mostly focus on individual areas of Vulkan, rather than providing a wrapper around it as a whole and therefore weren't included here. Only two libraries mentioned provide a significant enough abstraction around Vulkan to alleviate most of its issues with usability. In some areas, however, they overshoot in their design, sacrificing some of the benefits Vulkan has over other APIs in the first place.

Chapter 3

Interface design

3.1 General concepts

As is apparent from the previous chapter, there is still a place for an API that does not sacrifice performance or control for the sake of ease of use, while still providing a rich, but unobstructive set of abstractions specific to Vulkan. This chapter will provide an overview of such a design. Some general design practices and goals need to be laid out, first.

There are certain aspects any abstract programming interface should have. Its design should be intuitive, separated into logical parts that can be reasoned about individually. The most obvious path towards a goal should also be the correct path, and the correct path should be easy to use. Its components need to be consistent and easy to discover. It should be well documented, but the user shouldn't need to look up its documentation at every step to get things done. It should allow for the common tasks to be done easily and without any friction, but it should be able to handle most of the rare edge cases as well.

For an abstraction library around Vulkan in particular, a strong focus needs to be put on performance. Some small sacrifices may be made there, but only if other aspects of the API benefit from it enough and if the performance lost is in areas that are not used often. It shouldn't incorporate too thick abstractions and heuristics that make it non-obvious for somebody familiar with graphics programming to know what is happening behind the scenes.

This explicitness is valuable for debugging and making it clear what operations are expensive on the GPU.

The library must have high levels of interoperability with Vulkan itself. That means accepting Vulkan objects as well as exposing them in its own abstractions. This is necessary both because of the rich ecosystem of existing libraries and tools around Vulkan that the user may wish to use, as well as their own code, but also because of the many extensions available, not all of which may be implemented in Tephra.

3.1.1 Object lifetime

One of the important design decisions relates to object lifetime. In Vulkan, all objects are created and destroyed explicitly. Since Tephra is a C++ API, it uses RAII concepts for managing the lifetime of resources and ensuring they get automatically freed after their handler's destructors are called. A relevant design aspect is that once a workload has been submitted to the device, it should be difficult to accidentally affect it afterwards. This is not true for Vulkan in general, where many objects are required to be kept around while the workload is executing. Such a design is not desirable when the destruction of objects is implicit as in RAII. Therefore, Tephra ensures that the destruction of Vulkan objects is deferred until it is safe to do so. This is done by maintaining a list of objects to be released or destroyed for every submitted workload after it and all previous workloads finish executing. This guarantees that all already submitted workloads are not affected. The resources are garbage collected through an explicit call or at various points during the API usage.

Already submitted workloads can be unintentionally affected by other ways than hasty destruction of Vulkan objects. Data can be visible by both the host and the device at the same time, creating hazards. This is especially dangerous for developers used to OpenGL, where, for example, creating one uniform buffer and overwriting its contents every frame is common, despite the device still accessing the old contents of the previous frame. OpenGL handles this by silently creating multiple uniform buffers. Doing the same in Tephra would violate explicitness, so it instead encourages the use of transient resources whose lifetime is limited to a single workload, making the user request every "fresh" buffer. To the user it may then look like a new uniform buffer is being created every frame, but instead the old buffers are reused once they are no longer being accessed by the device. This can be handled efficiently through the job system described in later chapters. It improves upon what OpenGL does by making this creation explicit, even though the

buffers are still managed by the implementation.

3.1.2 Handling of arrays

Due to the effort in Vulkan to minimize API calls and context switches, many functions accept and return arrays of values. As a general C API, Vulkan exposes this by asking the user for pointer and count pairs, while functions that return arrays are expected to be called multiple times, first returning the number of elements to be returned and then filling an array prepared by the user.

A C++ API can do better and, for example, allow the user to pass in a temporary array through the means of an initializer list, but also easily pass existing arrays of various form. Overloading the functions is one possibility, but for functions that accept multiple arrays this leads to combinatorial explosion of overloads. Since the array often needs to be contiguous anyway, a simple class that represents a contiguous view of an array is provided, `ArrayView`. It can be constructed from a pointer and count pair, or converted by the global `view`() function from iterator pairs, standard library vectors or C arrays. Such a general purpose array view must not be constructable from temporary arrays such as initializer lists, however, because it does not own the data and merely points to it. To enable temporary arrays to be passed to functions, a second array view class is provided, `ArrayParameter`. This one is expected to have a temporary storage and unlike `ArrayView` is directly constructible from initializer lists, as well as from regular `ArrayViews`. Where a temporary storage is not permissible, like as members of structs, Tephra asks for an `ArrayView` instead. For convenience, `rangeView`() function is also defined, creating a view of only a specific range of the passed array and `viewOne`(), creating a single element view out of a reference.

Listing 3.1: ArrayView usage example

```
tp::Pipeline* compiledPipelines[2] = { &pipelineA, &pipelineB };
device->compileComputePipelines(
    { &pipelineSetupA, &pipelineSetupB },
    pipelineCache,
    tp::view(compiledPipelines)
);
```

3.1.3 Threading behavior

Another aspect of a modern graphics API is supporting multithreaded usage. There are several key areas that require multithreading to be efficient: Building pipelines, which involves compiling shaders, and recording commands for the GPU. Both need to support parallelization well enough. While designing such an API, every object should take one of the following stances: Either it can be used from multiple threads at the same time, in which case the object is said to be internally synchronized, or it is left to the user to make sure only a single thread accesses the object at one time in a way that modifies its state. Then it is externally synchronized. In Vulkan, all objects with the exception of the pipeline cache are externally synchronized.

In Tephra, the `Application`, `Device` and `PipelineCache` objects are internally synchronized, the rest externally. The reasoning for this decision is that in a multithreaded application, all threads need to interface with these objects and can't just create their own instances, unlike, for example, various resource pools, which are not thread safe and is up to the application to decide whether to synchronize this access or simply make sure only one thread accesses each instance at a time.

Every job and command list can only be built by one thread at a time. The parallelism is mainly achieved by being able to execute multiple command lists within a job, each recorded by a different thread. Building jobs can be parallelized as well, but that may somewhat limit the usefulness of jobs if they are unnecessarily split up. Submitting jobs can only be done in parallel if they are being submitted to different queues.

3.1.4 Validation and debugging

Validation and debugging is well designed in Vulkan. By default, very little error checking and input validation is performed, but optionally, validation layers can be inserted in between the API front and the driver that check for correct API usages and provide debug information. Such information should be retained and still make sense to the user, while being augmented with the library's own valid usage checks. For that purpose, many Tephra function implementations come with toggleable validation checks and together with the debug system provide the user with information what function call caused the debug message to be displayed. Additionally, almost every object can be assigned with a name to identify objects of the same type between each

other. The naming is also transmitted to Vulkan objects through its own debug naming extension that can then be used by 3D graphics debuggers such as RenderDoc [5].

3.2 Initialization

In Vulkan, several steps are necessary before any meaningful work can be done with it. The first catch comes right away: The Vulkan header provides static pointers to the API functions defined by the loader. The loader then dispatches the calls to the underlying instance and device implementations of the driver. Because there can be multiple devices and versions present, the loader must choose the correct implementation to use on every API call. To avoid this overhead, Vulkan provides a capability to load the function pointers dynamically, specific to the actual driver and device that will be used. This is, however, less convenient to the user.

The next step is to create a `VkInstance` object, used to store any kind of application-wide state. During creation, this provides an opportunity for the user to specify a list of layers. Layers in Vulkan are able to hook up to the API calls the application makes and provide new functionality, like capturing screenshots, or offer debugging utilities and usage validation. The application also specifies its name and version, allowing the driver developers to detect it and inject application dependent optimizations.

An instance object then provides a way to query the list of Vulkan supporting physical devices present in the environment. Various properties of each device are exposed to the application, like a list of supported extensions, features, formats, memory properties and more. The role of the application at this stage is to choose one or multiple physical devices, along with the extensions, features and queues that are to be enabled. The `VkDevice` object, created out of a single physical device or a group of compatible devices, then represents a logical device and will be used for the rest of the API calls.

Finally, the `VKSwapchainKHR` is an optional object, a part of an extension rather than the core API. It is nevertheless essential for presenting images to the screen. It provides a way for the OS specific surface to provide GPU images to the application to draw to and facilitates presenting them to the surface. Rather than a single `swapBuffers`() function, Vulkan splits it into two separate operations: `vkAcquireNextImageKHR`(), which waits for a swapchain image to become available for drawing a new frame. `vkQueuePresent`() then

submits the image back to the swapchain, ready to be displayed on the screen. The number of images in the swapchain is controllable by the user, as well as the presentation mode.

In previous APIs, instance, device and swapchain objects have been abstracted into a single Context object. That approach, however, does not allow the user to select a specific device from multiple available ones, or using more devices at once at all, unless they are able to be linked into a single device group. Compute only workloads also don't require a swapchain or a window to render to. Creating a windowless context in OpenGL can be difficult. For that reason keeping the context split into instance, device and swapchain objects is beneficial. The extra steps needed are only a one time cost at initialization. Even still, querying the extensions, features, formats, queues and memory types of all physical devices and choosing the right one that meets all requirements can take a lot of work if it is to be done properly, so there is room for abstraction there.

Besides exposing interface for querying the capabilities of every physical device directly, Tephra introduces the `DeviceProfile` object. This object contains a collection of all capabilities and features a device can have. Specified by the user, it can represent the application's requirements for the kind of device it is supposed to run on. In the library it is used in several ways. A physical device can be queried to support the specified `DeviceProfile`, offering a simple way to find a device that meets the minimum requirements. The `DeviceProfile` object is then used for constructing a `Device` object from the chosen `PhysicalDevice`, with the library enabling only the extensions and features present in the profile object. Finally, in debug mode, it can provide validation that the application does not exceed the capabilities declared in the profile, even though the actual device supports them. This makes it easier to write applications that support many different devices and are able to recover from missing capabilities as early as possible.

Listing 3.2: Initialization usage example

```
// Debug report handler logs debug messages from Vulkan and Tephra
auto debugHandler = tp::StandardReportHandler(std::cout);

auto appSetup = tp::ApplicationSetup(
    { "Example app", tp::Version(0, 1, 0) },
    &debugHandler
);
auto app = tp::Application::createApplication(appSetup);

// Set required device capabilities
tp::DeviceProfile profile {};
profile.extensions.add({
    tp::DeviceExtension::GraphicsJobs });
```

```
// Pick the first device that meets the requirements
const tp::PhysicalDevice* physicalDevice = nullptr;
for (auto& device : app->getPhysicalDevices()) {
    if (device.supportsProfile(profile)) {
        physicalDevice = &device;
        break;
    }
}
if (physicalDevice == nullptr) {
    std::cerr << "No graphics device is present" << std::endl;
    exit(1);
}

// Add any optional capabilities
if (physicalDevice->features.contains(
        tp::DeviceFeature::GeometryShader)) {
    profile.features.add(tp::DeviceFeature::GeometryShader);
}

auto device = app->createDevice(*physicalDevice, profile);
```

3.3 Shader interface and descriptor layout

Vulkan accepts shader programs only in the SPIR-V intermediate representation format. Compilation of GLSL and HLSL programs into SPIR-V is handled by third party utilities (glslangValidator) separate from the graphics API itself. At runtime, SPIR-V shader modules are then used to build pipeline objects. A pipeline object can consist of multiple shader modules, each for one of the different programmable pipeline stages, such as vertex, fragment or compute stages. They are further described in the following chapter.

An important aspect of an application is interfacing host code with shaders executed on the device. Images and buffers need to be bound to become accessible from within the shader. Unlike in OpenGL, binding resources to symbols is not done through their name in Vulkan, but instead they need to be assigned a binding number in the shader code explicitly. The resources are then bound through descriptors to those numbers. But because binding them individually would incur too much overhead and does not translate well to actual hardware, they are bound at once in descriptor sets, created ahead of time.

Before that, however, the application needs to provide the descriptor set

layout, defining the interface through which resources are bound as descriptor sets. It is a collection of descriptor bindings, containing the descriptor type, binding number within the set, array count and a set of shader stages that can access it. Descriptor set layouts then need to be provided alongside the shader modules when building a pipeline, and the same descriptor set layout then needs to be used for creating the descriptor sets that finally assign resources to descriptors.

The available descriptor types in Vulkan, and by extension in Tephra, are:

- `Sampler` describes the parameters for sampling a sampled image, such as the filtering used.
- `SampledImage` is a readonly image that can be accessed in combination with a sampler.
- `CombinedImageSampler` is a sampled image and a sampler bound together in a single descriptor. It is provided by Vulkan for GLSL convenience and it is said that some platforms may benefit from using combined image samplers.
- `StorageImage` is a read/write image that is addressed directly with load, store and atomic operations instead of through a sampler.
- `UniformBuffer` is a read-only buffer of limited size. Used for efficiently passing uniformly accessed, constant data to shaders.
- `StorageBuffer` is a general purpose read/write buffer that offers load, store and atomic operations to arbitrarily structured data.
- `UniformTexelBuffer` and `StorageTexelBuffer` are general purpose read-only and read/write buffers accessed through an associated format in the same way as for an image. In Tephra, `UniformTexelBuffer` is renamed to `TexelBuffer` to avoid confusion, since it doesn't behave like a formatted `UniformBuffer`, as the name would otherwise suggest.
- `InputAttachment` is used for framebuffer-local read operations. Explained in detail in a later chapter.

For buffers, the starting offset and size can be provided when creating a descriptor set. This allows the user to combine data of similar purpose into large buffers and bind individual parts of them, a practice that is encouraged in Vulkan. Additionally, uniform and storage buffers can be marked as dynamic, allowing the offset and size of the binding to be passed dynamically at a time when the descriptor set is being bound, rather than during its creation.

Multiple descriptor sets can be bound at the same time and they are identified between each other by their set number. The order of descriptor sets is hierarchical. When a descriptor set gets bound to a set number, the descriptor set currently bound to that number or higher becomes unbound. Similarly, when a pipeline created with the same descriptor set layouts as the

currently bound descriptor sets up to some set number, then all descriptor sets above that number get unbound. The intent of this is to have descriptors that change often between draws in a set with a high set number, but descriptors that are common for all draws in a frame would have a low set number, with all the pipelines used compatible with it. This allows such descriptor sets to be bound much less frequently, improving performance.

This presents somewhat of a shift in the way shader interface is handled. In OpenGL, it is common practice to rely on shader code reflection, letting the application adapt to changes in the shader code. It does not matter when different shaders have a different interface, but in Vulkan, it is important for optimal performance to have shaders share the same interface, defined in the application through descriptor set layouts. In fact, Vulkan offers no reflection capabilities for the layout of descriptors itself, but there exist third party libraries like SPIRV-Cross that provide it, as it is necessary for some use cases. Ideally, however, to avoid bugs with descriptors and other interface mismatching, they would be defined in separate files and used both by the application and shader modules. Such a library is beyond the scope of this project, though.

Besides uniform buffers, per-draw uniform data can also be passed to the shaders through push constants. These enable to pass limited amount of information to the shader at the time the draw call is being made, without storing them in an additional buffer and binding descriptor sets. For constructing a pipeline, only the size and offset into a user provided pointer need to be provided, as well as the set of stages they can be accessed in.

Passing additional data to shaders through buffers or push constants can be tricky. The layout needs to follow the std140 and std430 layout standards, respectively. Because of these requirements, one cannot just define an identical looking structure in C++ code and expect the binary interface to match the memory layout in the shader. Therefore providing a simple compile-time interface and auto-layout functionality for std140 and std430 would be useful. The user can then define the shader structure with identical layout in C++, decorated with some additional macros and types, and then be able to use it directly for writing data to uniform buffer memory. Such a compile-time layout definition would be the fastest way to write aligned data, but the data structure might not always be known during the compilation. Additional runtime functions for querying the alignment of individual types could be provided as well for that purpose.

Listing 3.3: Pipeline layout example

```
const tp::DescriptorBinding firstSetBindings[] = {
    // Global uniform buffer in set #0, binding #0
    { 0, tp::DescriptorType::UniformBuffer,
      tp::ShaderStage::Vertex | tp::ShaderStage::Fragment },
    // Shadow map in set #0, binding #1
    { 1, tp::DescriptorType::SampledImage,
      tp::ShaderStage::Fragment } };
const tp::DescriptorBinding secondSetBindings[] = {
    // Material specific texture, set #1, binding #0
    { 0, tp::DescriptorType::SampledImage,
      tp::ShaderStage::Fragment } };
// Object specific push constant data
tp::PushConstantRange pushConstants = {
    tp::ShaderStage::Vertex, 0, sizeof(glm::mat4x4) };

auto firstSetLayout = device->createDescriptorSetLayout(
    0, tp::view(firstSetBindings));
auto secondSetLayout = device->createDescriptorSetLayout(
    1, tp::view(secondSetBindings));
auto pipelineLayout = device->createGraphicsPipelineLayout(
    { &firstSetLayout, &secondSetLayout },
    { pushConstants });
```

3.4 Pipelines

What was previously managed as a global state in OpenGL is now contained in a pipeline state object that is created ahead of time and then bound as needed. In Tephra, the pipeline state object is composed of:

- Shader modules for all active programmable pipeline stages
- Reference to a pipeline layout containing the layout of descriptor sets and push constants
- The layout of vertex input bindings
- The render pass and subpass the pipeline will be executing in. They contain information about the format and sample count of the framebuffer attachments that the shader will be rendering into and pixel local storage layout for tiled architectures.
- Rasterization primitive (points/lines/triangles) and input geometry topology (triangle list/strip/fan)
- Number of viewports in a multi-viewport setup
- Depth and face orientation culling, stencil culling
- Multisampling and alpha-to-coverage
- Blending state

All of this information should be possible for the user to reason about ahead of the actual rendering and define it as a monolithic state. This allows the driver to perform any optimizations needed for that particular setup. Some parts of the pipeline state can be also set dynamically, like the blending constants and stencil reference values. In fact, Tephra leverages this in the background. Vulkan, by default, asks for the full viewport and scissor test states that depend on the resolution of the framebuffer. This can hardly be predicted ahead of time without recompiling pipelines every time the window size changes. Luckily, Vulkan allows these to be set dynamically, leaving only the number of viewports to be specified when creating a pipeline.

Another major simplification Tephra makes here is in the render pass. As will be described in the following chapter, render passes also specify the synchronization and image layout transitions, which depend on what commands are executed before and after the render pass. This is also difficult to predict ahead of time. Tephra leverages Vulkan's concept of compatible render passes that allows a pipeline to be used with a different, but compatible render pass from the one it was created with. It only asks of the user of only the compatible subset at this time, called `RenderPassLayout`.

Even in Vulkan, creating pipelines is slow. It often involves compiling the shader stages and potentially baking pipeline state into the shader instructions. For that purpose, Vulkan provides pipeline cache objects for speeding up the pipeline compilation phase and even allows saving this opaque, driver defined data structure to disk, so it can be read back on the next application run. Pipeline caches are one of the few thread-safe objects in Vulkan, so that they can be used for pipeline compilation in multiple threads. Additionally, to let the driver reuse resources between pipelines as much as possible, they are compiled in batches.

Creating pipeline objects can get especially verbose in Vulkan due to all the state that is required to be set. Tephra simplifies this by introducing sensible defaults that leave most features, like geometry shaders, multisampling, blending and others disabled, unless explicitly requested. All of the state that is required for the pipeline and that has no sensible defaults has to be provided in the constructor, so that the setup is always valid. Compilation happens by the virtue of `compileGraphicsPipelines`() and `compileComputePipelines`() device functions that take an array of pipelines to compile and a pipeline cache object to use. The compilation happens in a single thread, but the user is allowed and encouraged to split the work into multiple threads, each calling these functions for its own batch.

Listing 3.4: Pipeline setup example

```
tp::ShaderStageSetup vShaderSetup = { &vertexModule, "main" };
tp::ShaderStageSetup fShaderSetup = { &fragmentModule, "main" };

tp::VertexInputAttribute inputAttributes[] = {
    // Vertex attribute #0 - Position, 16 bytes
    { 0, tp::Format::COL128_R32G32B32A32_SFLOAT, 0 },
    // Vertex attribute #1 - Normal, 8 bytes
    { 1, tp::Format::COL64_R16G16B16A16_SFLOAT, 16 } };
tp::VertexInputBinding vertexInput = {
    // Vertex binding #0, 16 + 8 byte stride
    0, tp::view(inputAttributes), 24, tp::VertexInputRate::Vertex };

tp::GraphicsPipelineSetup pipelineSetup = {
    &pipelineLayout, &renderPassLayout, 0,
    { vertexInput },
    vShaderSetup, fShaderSetup };
pipelineSetup.enableFaceCulling(false, true);
pipelineSetup.enableMultisampling(tp::MultisampleLevel::x4);
pipelineSetup.enableDepthTest(tp::CompareOp::LessOrEqual);
// Enable alpha blending
tp::AttachmentBlendState blendState = {
    tp::BlendState(
        tp::BlendFactor::SrcAlpha,
        tp::BlendFactor::OneMinusSrcAlpha),
    tp::BlendState::noBlend() };
pipelineSetup.setColorAttachmentBlendStates({ blendState });

tp::Pipeline pipeline;
device->compileGraphicsPipelines(
    { &pipelineSetup }, &pipelineCache, { &pipeline });
```

3.5 Render passes

In recent years, there has been an increasing number of GPU architectures leveraging the tiled rendering method [11], in which the rendering viewport is subdivided by a regular grid into tiles. Each tile is then rendered separately with the geometry that intersects it. This is unlike traditional rendering processes, where the entire image is rendered at once. The advantages of this stem from reducing the storage required for the render process. This can allow the GPU to use the much faster on-chip buffers for storing intermediate results, rather than communicating to memory, reducing bandwidth usage and improving performance. Products that use tiled rendering include Nvidia GPUs since the Maxwell architecture and many mobile and embedded GPUs from manufacturers such as ARM (Mali), Qualcomm (Adreno) and Imagina-

tion Technologies (PowerVR 5/6/7 series). All of these support Vulkan as a graphics API.

Current graphics APIs with the exception of Vulkan and OpenGL ES, are however not well fitted for driving tiled architectures. The application might want to implement deferred rendering and draw various attributes of the scene geometry to temporary buffers in one pass, followed by one or more passes that, for each pixel, read these attributes and calculate lighting for the pixel. On a tiled architecture, it would be possible to store the attributes in an on-chip memory, sending data from one pass to another without involving external memory. This is only possible if each pixel in the later pass only accesses data from a pixel in the same tile that was written in a previous pass. The problem is that the application cannot communicate that combining such passes together is possible, leaving it up to guesswork for the driver. Worse still, combining passes may require the shaders used in them to be compiled differently and that may bring stalls or reduced performance if it has to be done at runtime. OpenGL ES handles this problem simply by providing an extension that allows the user to write specialized shaders that write to and read from what's called pixel local storage that persists in between compatible passes. The downside is exactly that - the use of pixel local storage requires its own shader as well as significant changes to the rendering code, increasing development efforts to support both styles of rendering architectures efficiently. Vulkan, despite being a much lower level API, resolves this with a relatively high-level abstraction. The user creates RenderPass objects that mainly provide the following information to the implementation:

- The format and sample count of each framebuffer attachment (render target image) used.
- A number of subpasses, each of them containing information about the indices of attachment used as depth/color/input images for the subpass.
- How the subpasses depend on each other.

Each subpass represents a single rendering pass that reads and writes to a subset of the given attachment images. The dependencies between subpasses inside of a single render pass then let the driver know how the passes depend on each other and how the rendering is going to go ahead of time, during pipeline creation. This makes it possible to defer the appropriate layout of pixel local storage and let the driver implementation compile shaders to be as efficient for tiled architectures as possible. Even other architectures may benefit from the additional information that was previously known to the driver only after the rendering takes place.

During rendering, render passes are used through the self-explaining commands `VkCmdBeginRenderPass`, `VkCmdNextSubpass` and `VkCmdEndRenderPass`. The

first of which assigns matching resources to the attachments and begins the recording of the commands belonging to the first subpass of the render pass. All pipelines bound at this point must be compiled for this subpass or one compatible with it. The following commands simply advance to the next subpass in the render pass or exit the recording altogether.

Since render passes are themselves quite high-level constructs, Tephra only simplifies their use in several ways. The first is by utilizing the Vulkan concept of compatible render passes, where one render pass can be substituted by another if they only differ in a certain subset of attributes. Letting the user only specify information outside of this subset removes the need to define the load and store operations as well as external dependencies, which are often difficult for the user to predict ahead of time. This also decreases the number of pipelines that need to be created that would only differ in those details. The compatible subset of render passes can then be used both to construct pipeline layouts and actual render passes at runtime. It is, therefore, appropriate to call it `RenderPassLayout`.

Another way the complexity can be reduced is by realizing that the most commonly used render passes only have a single subpass and the ones that have multiple mostly only depend on each other through the framebuffer attachments. These dependencies can be easily deduced by the API and the user only needs to specify which subpasses depend on each other, rather than filling out a long structure describing the details of the dependency. For advanced use cases, where different subpasses or parts of the pipeline depend on each other through non-attachment resources such as storage images and buffers, the user can supply additional dependencies manually.

The use of render passes during rendering is not too different in Tephra. Together with the `RenderPassLayout`, the user specifies which images to assign to each attachment, along with the load and store operations. Unlike the stateful begin and end render pass commands, however, the user submits render lists that contain the draw commands for each subpass. These lists allow for efficient recording of commands with minimal overhead. That means, while it is often not necessary, that the user may sometimes need to provide information about what resources will be used in the render pass so the library can synchronize it properly. This will all be talked about in more detail later. The important aspect is that similar concept applies to compute operations as well. Tephra, unlike Vulkan, also has compute passes that accept compute lists in a similar fashion as render passes accept render lists. The difference is that in a compute pass, there are no attachments and there is only one subpass.

Listing 3.5: RenderPassLayout setup example with multisampling

```
const tp::AttachmentDescription attachments[] = {
    { tp::Format::DEPTH32_D32_SFLOAT, tp::MultisampleLevel::x4 },
    { tp::Format::COL32_R16G16_SFLOAT, tp::MultisampleLevel::x4 },
    // Resolve attachment
    { tp::Format::COL32_R16G16_SFLOAT, tp::MultisampleLevel::x1 } };

const tp::AttachmentBinding bindings[] = {
    { tp::AttachmentBindPoint::DepthStencil(), 0 },
    { tp::AttachmentBindPoint::Color(0), 1 },
    // Resolve attachment gets filled with resolved samples
    // from the 0th color attachment
    { tp::AttachmentBindPoint::ResolveFromColor(0), 2 } };

// Single subpass RenderPass
tp::SubpassLayout subpass{ tp::view(bindings), {} };
tp::RenderPassLayout renderPass = device->createRenderPassLayout(
    tp::view(attachments), { subpass });
```

Listing 3.6: RenderPassLayout setup example for deferred rendering

```
const tp::AttachmentDescription attachments[] = {
    // G-Buffer attachments
    { tp::Format::DEPTH32_D32_SFLOAT },
    { tp::Format::COL32_R8G8B8A8_SRGB },
    { tp::Format::COL32_R16G16_SFLOAT },
    // Lighting buffer
    { tp::Format::COL64_R16G16B16A16_SFLOAT } };

const tp::AttachmentBinding gBufferBindings[] = {
    { tp::AttachmentBindPoint::DepthStencil(), 0 },
    { tp::AttachmentBindPoint::Color(0), 1 },
    { tp::AttachmentBindPoint::Color(1), 2 } };

const tp::AttachmentBinding lightingBindings[] = {
    // 0th framebuffer attachment bound both as a readonly
    // depth&stencil attachment, but also as an input
    // attachment that can be read from in the shader
    { tp::AttachmentBindPoint::DepthStencil(true), 0 },
    { tp::AttachmentBindPoint::Input(0), 0 },
    { tp::AttachmentBindPoint::Input(1), 1 },
    { tp::AttachmentBindPoint::Input(2), 2 },
    { tp::AttachmentBindPoint::Color(0), 3 } };

tp::SubpassLayout gBufferSubpass{
    tp::view(gBufferBindings), {} };
tp::SubpassDependency gBufferDependency{ 0 };
tp::SubpassLayout lightingSubpass{
    tp::view(lightingBindings),
    // Depends on the G-Buffer pass
```

```
    tp::viewOne(gBufferDependency) };
tp::RenderPassLayout renderPass = device->createRenderPassLayout(
    tp::view(attachments), { gBufferSubpass, lightingSubpass });
```

3.6 Memory management

Where previous APIs have handled memory allocation on the GPU automatically, Vulkan leaves everything in the hands of the user. While it can be as easy as allocating a separate block of memory for every resource, doing so is not efficient, because the memory is allocated directly through the OS, which can be an expensive operation. In fact, the recommended size of an allocation is in the range of 128-256 MB. This makes it almost a requirement for the user to implement their own sub-allocation strategy or use an existing library such as the Vulkan Memory Allocator [2] developed by AMD. Tephra also makes use of this library internally, however it provides its own memory allocation interface.

One way resources can be categorized is by their lifetime. Long lived (also called persistent) resources such as textures, vertex buffers and other data, are allocated once and live throughout many frames and work submissions. Short lived resources, on the other hand, are only required as temporary storage for intermediate results and are often immediately consumed with their data discarded. Examples would be framebuffer images and staging buffers for data transfers between the CPU and GPU. This is an important distinction in Tephra. Short lived resources are mostly managed by the job system described in a later chapter and the user does not have to allocate them manually.

Vulkan exposes up to 11 different memory types to allocate from. In practice, however, they can be grouped into 5 main types:

- Device local memory for efficient read/write access by the GPU, but most likely inaccessible to the CPU. Most resources will probably live here. This type is always available.
- Host memory without CPU caching for fast writes by the CPU. This memory is still accessible by the GPU, but at a slower pace. Their main purpose is to host staging buffers that temporarily store data to be copied to GPU memory. It can also be used for reasonably fast direct access by the GPU, but only if it's read once and can fit into the GPU cache. This type is always available.

- Host memory with CPU caching for fast reads by the CPU. Ideal for moving data from GPU to CPU, however GPU reads from this memory might be slower than from the previous type.
- Device local memory types that are also visible by the CPU, with or without caching. This type supersedes the previous ones in performance, but on desktop cards it might not exist or be limited in size. On integrated GPUs and other devices with unified memory architecture, memory of this type is prevalent, on the other hand.

If choosing the appropriate memory type wasn't tricky enough, resources of different type may only be able to reside in specific memory types. This makes code highly platform dependent, not just for the choice of memory type itself, but depending on the memory location a transfer between host and device might or might not need a staging buffer and an extra copy. The selection of a memory type can be both automatic and explicit, if the user specifies a progression of memory types that should be considered for that resource, starting from the most preferred types. The library then iterates over that sequence until it finds a valid memory location for that resource or reaches the end, throwing a memory allocation error. The memory type the resource ended up allocated in can be queried afterwards.

Because there are several common use cases, Tephra offers several predefined memory progressions that should cover most use cases:

- `Device` - Only device local memory will be allocated, otherwise memory allocation error is thrown. Device-local memory is required when allocating images and should be used for resources accessed by the GPU often enough that not being in device local memory should be avoided for performance reasons. Progression: `DeviceLocal` → `DeviceLocalHostVisible` → `DeviceLocalHostCached`
- `Host` - Used for resources that should live in host memory. Meant for large data that is being read by the GPU infrequently and shouldn't be wasting the potentially limited device local, host visible memory. This is the best progression for staging buffers to then copy data from to device local memory. Progression: `HostCached` → `HostVisible`
- `UploadStream` - Used for resources that are written to by the CPU often enough that a copy from a staging buffer should be avoided. Progression: `DeviceLocalHostVisible` → `DeviceLocalHostCached` → `HostVisible` → `HostCached`
- `ReadbackStream` - Used for reading back data from the GPU without a staging copy. Progression: `DeviceLocalHostCached` → `HostCached` → `DeviceLocalHostVisible` → `HostVisible`

3.7 Images and buffers

There are two main types of resources that can be created in Vulkan. Buffers host only one dimensional data, while images are also able to support 2D and 3D data with filtering and mipmapping as well as cubemaps and image arrays. Images store data in a given format, while buffers can store data arbitrarily and do not have a format associated with them, unless accessed by a buffer view or as a vertex buffer. While buffers are mostly used for providing uniform data to shaders or describing geometry with vertex and index buffers, images store the textures that are sampled from or framebuffer attachments that are rendered into. All buffers can be created in either host or device memory, but images can only be created in device memory and cannot be directly accessed by the host.

There are several stages of resource creation in Vulkan. First, the resource itself needs to be created according to the selected size and format and intended usage. Then, its memory requirements need to be queried and the resource bound to a memory location that matches those requirements. Separating resource creation and memory binding in this way helps the user choose the right memory in Vulkan, but it becomes less useful when the process of selecting the memory is automated. Other than that, resource creation is straightforward enough in Vulkan and mostly unchanged here. Like with other objects in Tephra, the lifetime of resources is extended when they are destroyed so that all already submitted workloads using it finish successfully.

During creation, the user must specify how the resources are going to be used. Images that are meant to be render targets might be allocated differently from sampled textures. Additionally, in Vulkan, images have a state associated with them, their layout. The layout further limits how the image can currently be accessed, but is able to be changed at runtime when needed through an image layout transition operation, which is a part of synchronization barriers. In practice, this might mean that, for example, a render target gets decompressed when its transitioned from being a render target to a texture to be sampled. Tephra resolves image layouts automatically, as will be described in later chapters.

Image resources are usually not accessed directly, but through `ImageView` objects. These select a range of the image's subresources (array elements, mip levels, depth/stencil aspects) and optionally interpret 2D image arrays as sides of a cubemap, or 3D images as an array of 2D slices. They can also remap the components or interpret the data as a different, but compatible

format. `BufferView` objects offer the same thing for a buffer, representing a contiguous range of its contents, optionally interpreted with a given format.

Listing 3.7: Image setup example

```
// Create a cubemap image as an array of 6 2D layers
tp::ImageSetup imageSetup {
    tp::ImageType::Image2DCubeCompatible,
    tp::ImageUsage::ColorAttachment | tp::ImageUsage::SampledImage,
    tp::Format::COL32_R8G8B8A8_SRGB,
    tp::Extent3D(256, 256),
    9, 6 };
auto image = device->allocateImage(
    imageSetup, tp::MemoryPreference::Device);

// Create the cubemap view that can be used in shaders
tp::ImageViewSetup viewSetup {
    tp::ImageViewType::ViewCube,
    image->getWholeSubresourceRange() };
auto cubeView = image->createView(viewSetup);
```

3.8 Descriptor sets

Descriptor sets were mentioned in one of the previous chapters, but it hasn't been described how they are actually created and used. In Vulkan, they aren't created individually, but instead are allocated from descriptor set pools for performance reasons. However, they are fixed in size, so the user needs to know the maximum number of various types of descriptors that will be allocated from it ahead of time, an information that is rarely known. Tephra's descriptor set pools instead automatically grow as needed, by creating additional Vulkan pools. As with other pools in Tephra, their behavior is configurable to fit the application's needs.

In Vulkan, descriptor sets are mutable. They are allocated in an uninitialized state and can be written to any amount of times. However, because they are updated from the host and used by the device, care must be taken to not update a descriptor set that is still in use. Exposing this would be in violation of the principle that no host operations should affect workloads that have already been submitted to the device. The intent of mutable descriptor sets is reusing resources from past workloads instead of allocating new descriptor sets every time. In Tephra, resources with a short lifetime are managed by the job system as will be discussed in later chapters. As a consequence, descriptor sets are immutable once created and they are immediately initialized with

data during their creation, since they can be assumed to have a longer lifetime.

There is one use case where mutable descriptor sets are necessary and that is the "bind everything" method. It relies on binding one very large descriptor set with descriptor arrays that contain every resource that will be accessed during the workload. This avoids the overhead of binding descriptor sets between draw calls as this large descriptor set can only be bound once. Rebuilding such a large set from scratch every time would be inefficient in that case. Since this is still only practical in Vulkan with the use of a fairly recent extension, supporting this use case is not a priority yet.

Listing 3.8: Descriptor set example

```
// Create default descriptor pool
auto descriptorPool = device->createDescriptorPool({});

// Allocate descriptor set for the layout defined
// in previous chapters.
tp::Descriptor descriptors[] {
    { uniformBuffer->getDefaultView() },
    { shadowmap->getDefaultView() }
};
tp::DescriptorSetSetup setup {
    firstSetLayout, tp::view(descriptors) };
tp::DescriptorSet firstSet;
descriptorPool->allocateDescriptorSets(
    { setup }, tp::viewOne(firstSet));
// ...
list.cmdBindDescriptorSets(
    pipelineLayout, { firstSet, secondSet });
```

3.9 Execution model in Vulkan

In previous generation APIs, commands are generally submitted one at a time and there are guarantees that the following command will be able to see the results of all commands submitted previously, so no synchronization by the user is needed. In Vulkan, however, commands are first recorded into a command buffer and submitted in bulk later into one of the device's queues. Commands and queue submissions also need to be synchronized between each other to get correct results. This is what helps Vulkan enable multithreaded use.

Each device exposes one or multiple queues split into queue families by

their capabilities. Commands submitted to different queues may be able to run in parallel on the device, but that does not have to be the case. Generally graphics commands will be sequential, but compute commands may run in parallel with each other and with graphics commands. That is only useful if one of the commands does not have full occupancy of the GPU's resources, otherwise it can actually degrade performance. Therefore, it should only be used together with careful profiling. The more useful command-wide parallelism is provided by queues that only support transfer operations. Those are generally DMA units able to copy memory asynchronously with other commands, but not necessarily as fast as the more capable queues.

Because queue families and the queues in them vary heavily from device to device, in Tephra, queues are specified by the intent of the user. There will always be a queue family that supports all available operations on device and is exposed as the main queue. Additional queues are exposed as a number of compute only and transfer only queues. They are guaranteed to support their respective operations, but they might not actually come from a different queue family and might even be the same as the main queue. This eases the effort in writing multi-platform code. The Vulkan queue actually used can be queried by the API or directly provided instead.

In Vulkan, commands are recorded into a command buffer instead of being submitted directly. There are primary and secondary command buffers. Only primary ones can be submitted to a queue, while secondary ones can only be executed from a primary command buffer. Render pass commands can only be used in a primary command buffer, but secondary command buffers can be executed inside of a render pass, inheriting it. Command buffers are allocated from a command buffer pool in Vulkan and can be potentially reused. Reusing command buffers is generally of limited usefulness, however, and you are expected to re-record them every frame.

3.9.1 Synchronization

Synchronization between commands also needs to be handled manually in Vulkan by explicitly defining dependencies between two sets of commands. There are two kinds of dependencies. An execution dependency guarantees commands from the first set will be performed before commands from the second set. A memory dependency builds upon that, guaranteeing that any memory operations performed by the first set will be visible by commands from the second set. In GPUs, this presumably corresponds with cache and pipeline flushes.

There is another way to characterize dependencies by the operations performed by the two sets and how they translate to execution and memory dependencies:

- Read-after-read dependency - Not actually a dependency, can happen in any order.
- Read-after-write dependency - Translates to both an execution and a memory dependency.
- Write-after-read dependency - Translates to an execution dependency only. The read operation needs to finish before the memory can be overwritten.
- Write-after-write dependency - Translates to both an execution and a memory dependency. Just execution dependency is not enough, because it alone does not guarantee the order in which write operations happen.

Dependencies are expressed in multiple ways in Vulkan. Pipeline barrier commands are the most common way, defining an execution and optionally a memory dependency between all commands that come before it and all commands that come after. They can also express an image layout transition. Semaphores are used to synchronize between queue submissions to different queues, forming both an execution and a memory dependency. Fences can be used to add a dependency between the device and the host. They can be waited on by the host or their state can be checked. The most general form of dependency expression are events. Events can be both signalled and queried by the host, and the device can also signal them and wait on them, down to the level of individual commands, but they cannot be used for synchronization between different queues.

A recent extension to Vulkan introduces another synchronization primitive - Timelines [8]. The issue with semaphores is that they are single use only and require a one-to-one correspondence. That is, exactly one queue submission signals a semaphore and exactly one submission waits on it. This can get rather inconvenient, especially for modular application designs that don't always have full knowledge about who the producer or consumer of their work is. Timelines solve this by keeping track of a single, monotonically increasing integer value. A queue submission or the CPU can both increment this value and wait until it reaches or surpasses a given value. There are also no restrictions to how many queue submissions can signal or wait on a timeline. It is easy to understand and convenient enough that Tephra exposes timelines in its API. As a KHR Vulkan extension, it is expected that most if not all drivers will support it, as it will eventually be merged into the core Vulkan API.

3.9.2 Frame graphs

The challenging part of Vulkan's synchronization is not only its complexity, but also the way it's exposed. Pipeline barriers synchronize two sets of commands. This is troublesome as it requires knowledge about what was executed before and what will be executed later. Keeping track of it manually is error prone and making a system like this modular is difficult. Improper synchronization results in either incorrect results or reduced performance. Overreaching or unnecessary pipeline barriers are the leading cause of poor performance over high-level APIs.

One solution for a modular synchronization system are frame graphs as described in a talk by Yuriy O'Donnell [10] and implemented in a third party library for Vulkan [4]. In short, they split up the rendering into individual passes, effectively extending subpasses in Vulkan's render passes. They take a data oriented approach towards the passes. Each pass is a standalone block, using virtual resource handles as inputs and outputs and defines which operations are going to be performed inside of it. Dependencies are then not expressed between passes themselves, but between resources. An input of one pass is either an output of some other pass or an imported permanent resource external to the frame graph. Transient resources, which exist only within the frame graph's scope, can also be created.

The user describes this frame graph ahead of time, giving the library high-level knowledge of the render process, allowing it to insert synchronization and assign memory to the virtual resources. Granite's render graph reorders the passes to minimize the number of barriers, while Frostbite's frame graphs follow the order they are defined in. An important part of both is aliasing of transient resources. Since they have limited lifetime which is known in advance, resources with disjoint lifetime can be aliased to the same memory. This can result in significant savings of memory usage. After the frame graph description is built by the user and compiled by the library, callbacks populate it with the actual device commands.

The downside of frame graphs is that such a description of the rendering process can be too involved and may lead to overspecification, providing more information than what is actually needed. It forces the user to split up code into passes according to synchronization concerns, rather than into logical blocks. Each pass is required to use a different set of virtual resources, which then need to be tied together. Finally, since the user only defines dependencies between passes, the actual order in which they are submitted can be arbitrary, for passes that do not depend on each other. The most efficient order is hard to determine and may depend on factors not easily

known to the API implementation and therefore should be left up to the user.

3.10 Jobs

Tephra presents its own system for submitting commands with correct synchronization to the GPU. Jobs represent units of execution that the user wants to submit at the same time to the device - a workload. That might correspond to an entire frame in a rendering application, or even just a copy operation for uploading a resource. Jobs solve the same problem as frame graphs described in the previous chapter, but instead of the user defining a graph structure with passes and dependencies between them, commands are instead recorded into a flat list, similarly to all other graphics APIs. The dependencies are inferred based on the accesses the commands recorded into it make. However, the key difference from previous graphics APIs is that this automatic synchronization is limited in scope. The user is asked for additional information that is needed for correct synchronization but that cannot be easily inferred from the job alone, like how the resource is going to be used inside command lists and in shaders, or to synchronize jobs submitted to different queues.

Figuring out accesses for copy commands and other work that does not involve programmable shaders is easy enough. But not only do shaders access resources through descriptor sets, making it difficult to directly track, most importantly they are unpredictable. It can be quite common that a resource is bound to a descriptor set, but isn't actually used by the shader. It could be determined, albeit not easily, whether a resource is used statically through shader program reflection, but next to impossible if its usage is determined dynamically, like by a value read from a buffer. So therefore Tephra requires the user to specify what accesses are being made during shader passes.

It would be tedious to specify all accesses on every draw command, though. Considering the common case of rendering the geometry of the scene through many draw calls into the same render target, several observations can be made. Most of the accesses, except for the render target, are read only. Many of them are textures and various vertex or index buffers that have been written to once and read the same way since then. A good way to exploit that is to allow the user to specify, after the resource has been written to, that it will be accessed as read-only in a given way. In Tephra, this is done through an `cmdExportResource`() command. A pipeline barrier can be inserted at that point to synchronize against a combination of future read accesses right away and any such reads will not need to be communicated to the API, since they

have already been handled. However, if the resource needs to be modified again or read in another way, that needs to be specified by the user. Any new unexpected accesses invalidate the effects of the previous export operation.

Additionally, where its necessary to specify accesses inside a shader, rather than doing it more or less identically for every draw call, it can be specified on the level of render and compute passes, collectively called command passes. In the uncommon case where commands within a command pass need explicit synchronization between each other, the user can either insert a pipeline barrier manually or split the commands into multiple passes.

The accesses recorded into the job also define the image layouts an image should be in at any point, making the insertion of image layout transition fairly straightforward. Vulkan includes an undefined image layout, which can be transitioned from whenever the contents of the resource don't need to be preserved, instead of its actual layout. This is exposed as a `cmdDiscardImageContents`() command, which makes Tephra internally override the current image subresource region's layout to the undefined one.

3.10.1 Command lists

The automatic synchronization that the job offers is really only useful for the high-level commands, rather than for the individual draw calls that often don't require any explicit synchronization between each other. Those commands also make up the bulk of all commands that are submitted, in most cases. For that reason Tephra provides command lists that record commands directly to a Vulkan command buffer with minimal overhead. They can also be recorded in parallel, allowing the recording of a job to be multithreaded.

The commands available in command lists are limited. This is both because of them being implemented as secondary command buffers as well as the fact that high-level commands like copy operations benefit greatly from automatic synchronization and the command overhead for them is very rarely a bottleneck. Also, because being inside or outside a Vulkan's render pass further restricts the commands available, command lists in Tephra are split into `RenderList` and `ComputeList`, each supporting only the relevant commands. Furthermore, rather than providing two different ways to do the same thing, along with a way to describe shader accesses for individual commands in a job, all of these compute and render commands can only be used in their respective command lists, not in jobs.

cmdExportResource	Job
cmdDiscardImageContents	Job
cmdCopy∗, cmdClear∗, cmdFill∗	Job
cmdBlitImage	Job
cmdResolveImage	Job
cmdExecuteComputePass	Job
cmdExecuteRenderPass	Job
cmdDispatch∗	CommandList
cmdDraw∗	RenderList
cmdClearAttachments	RenderList
cmdBindIndexBuffer	RenderList
cmdBindVertexBuffers	RenderList
cmdSet∗ (pipeline state)	RenderList
cmdBindPipeline	RenderList, CommandList
cmdBindDescriptorSets	RenderList, CommandList
cmdPushConstants	RenderList, CommandList
cmdPipelineBarrier	RenderList, CommandList
cmdResetQueryPool	Job
cmdCopyQueryPoolResults	Job
cmdBeginQuery, cmdEndQuery	Job, RenderList, CommandList
cmdWriteTimestamp	Job, RenderList, CommandList

Table 3.1: Availability of commands in Tephra

3.10.2 Transient resources and aliasing

Jobs also benefit from using transient resources, which are accessible only during the time the job is executing. This allows for more efficient allocation strategies, since all of these job-local resources will be released at the same time. It also makes it possible to automatically reuse memory if similar non-overlapping jobs are submitted after each other, e.g. once per frame. Transient images and buffers are requested by a Job call that takes an image or buffer setup structure instead of an already created resource. Not only images and buffers can be transient, but descriptor sets also often have a lifetime limited to a single job, since they can contain transient resources.

Similarly to frame graphs, it is possible to determine the first and last use of all resources within the job, allowing for memory aliasing of transient resources in the same way. Since commands are not being reordered in any way, the aliasing layout can be determined by walking through the recorded commands at job build time, allocating memory for the resource just in time for its first use and freeing it after the last. Extra synchronization may be needed between two different uses of the same region of memory.

Because the first and last use of transient, job-local resources can only be known after the job has finished recording, the actual allocation of resources has to happen after that point. That means that a resource view is returned that does not actually view an actual resource yet. It will still need to be accepted by the recorded commands and get assigned to the actual resource after the job has been recorded.

One special case are host initialized, job-local buffers. These buffers have their lifetime extended, starting not at the point the job is executing, but once it has finished recording. This is useful for uploading data from the CPU to the GPU, where the host can write into this job-local buffer before the job is submitted. The extended lifetime is necessary to ensure the memory is not being used by any previous jobs at the time the data is being written by the host. This requires a different allocation strategy as well.

3.10.3 Job lifetime and synchronization

Jobs are created from a `JobResourcePool` object that manages all of its resources. It assumes that jobs created from it will execute on the same queue and therefore in order. This allows it to assign the same memory to each job created from it as long as the resulting memory dependencies are handled. Pools don't free unused resources on their own, but unlike other pools, it is not a very viable strategy to free up those resources simply by recreating it in this case. The reason is that doing so makes a spike in memory usage, where old resources still have to be kept around and cannot be reused by the new `JobResourcePool`. Therefore, it provides a `trim`() method, that frees up resources that haven't been used since the last N calls of `trim`(), where N is an integer passed to the method. This is intended to be called before or after every frame or job submission, continuously freeing only those resources that are no longer in use.

`Job` objects go through several stages in their lifetime. They are created from a `JobResourcePool` object in a recording state, ready to accept commands to be executed. In this state they can be enqueued to a device queue, at which point the user is giving up access to the job and can no longer record commands into it. This is when all of the job-local resources are allocated and ready to be used. Preinitialized buffers can be mapped and written to and command lists can be recorded into compute and render passes. The user can then submit all of the jobs that have been enqueued into a queue as a single workload. After this point modifying any resources used by the job is an undefined operation, but they can be safely freed.

Jobs need to be synchronized between each other. Memory dependencies are implicit through access to resources. Execution dependencies between jobs submitted to the same queue are implicit as well on the level of Vulkan. Execution dependencies between jobs in different queues need to be specified by the user through timeline semaphores. One job should signal the semaphore by incrementing its value and another job that depends on it can wait until a specified value is reached. The second job will wait for the first by the virtue of an execution barrier even after it's been enqueued and submitted. The host can also directly wait on or query the execution of submitted workloads, using the handle returned by the `submitQueuedJobs()` call.

3.10.4 Job examples

Listing 3.9: Job to upload data through staging buffer example

```
auto jobPool = device->createJobResourcePool(
    tp::QueueType::AsyncTransfer, {});

// Upload data to the vertex buffer through a preinitialized
// staging buffer
auto stagingJob = jobPool->createJob();
tp::BufferSetup stagingBufferSetup{
    vertexBuffer->getSize(),
    tp::BufferUsage::TransferSrc};
tp::BufferView stagingBuffer = stagingJob->acquireLocalBuffer(
    stagingBufferSetup, tp::MemoryPreference::Host, true);

stagingJob->cmdCopyBuffer(stagingBuffer,
    vertexBuffer->getDefaultView(),
    { tp::BufferCopyInfo(0, 0, vertexBuffer->getSize()) });

// Export resource for future usage
stagingJob->cmdExportResource(
    vertexBuffer->getDefaultView(), tp::ReadAccess::DrawVertex);
// Enqueueing the job initializes its resources
device->enqueueJob(
    tp::QueueType::AsyncTransfer, std::move(stagingJob));

// Copy data to the staging buffer
{
    auto mappedMemory = stagingBuffer.mapForHostAccess(
        tp::MemoryAccess::WriteOnly);
    std::copy(
        vertexData.begin(), vertexData.end(),
        mappedMemory.get<glm::vec4*>());
}
```

```
// Submit the job for execution now
device->submitQueuedJobs(tp::QueueType::AsyncTransfer);
```

Listing 3.10: Job with a render pass drawing an object

```
auto job = jobPool->createJob();
// Render pass writes color to swapchain image
tp::FramebufferAttachment framebuffers[2] {
    tp::FramebufferAttachment(
        swapchainImage->getDefaultView(),
        tp::AttachmentLoadOp::Clear,
        tp::AttachmentStoreOp::Store,
        tp::ClearValue::ColorFloat(0.0f, 0.0f, 0.0f, 0.0f)),
    tp::FramebufferAttachment(
        depthBuffer->getDefaultView(),
        tp::AttachmentLoadOp::Clear,
        tp::AttachmentStoreOp::DontCare,
        tp::ClearValue::DepthStencil(1.0f, 0))
};

tp::RenderPassSetup renderPassSetup{
    renderPassLayout, tp::view(framebuffers), {}, {}};
tp::RenderPass* renderPass = job->cmdExecuteRenderPass(
    renderPassSetup);

// Record the render pass commands inline. The function will run
// when the job is submitted.
renderPass->recordInline(0, [this](tp::RenderList& renderList) {
    tp::Rect2D viewportRect = tp::Rect2D({ 0, 0 }, {
        swapchainImage->getExtent().width,
        swapchainImage->getExtent().height });
    renderList.cmdSetViewport({ tp::Viewport(viewportRect) });
    renderList.cmdSetScissor({ viewportRect });
    renderList.cmdBindGraphicsPipeline(pipeline);
    renderList.cmdBindDescriptorSets(
        pipelineLayout, { &descriptorSet });

    renderList.cmdBindVertexBuffers(
        { vertexBuffer->getDefaultView() });
    renderList.cmdDraw(vertexCount);
});

job->cmdExportForPresent(swapchain, frameIndex);
device->enqueueJob(tp::QueueType::Main, std::move(job));
device->submitQueuedJobs(tp::QueueType::Main);
```

3.11 Surface and swapchain

The final piece of the puzzle in a graphics API is displaying the rendering results on the screen. The display target can be a physical display in its entirety or just a window, owned by the window manager of the operating system. It is called a surface in Vulkan and is acquired through platform specific means. Physical devices also aren't guaranteed to support presenting to every surface through all of their queues.

From a surface, a swapchain object can be created that manages the process of continuously presenting frames to the surface. It allows some level of control by letting the user specify the presentation mode and number of swapchain images that will be created. These affect whether the presentation operation will be tied to the refresh rate of the monitor, if screen tearing can be observed and the display latency of the rendered images. The best settings depend on the kind of application and how much it values low latency, power usage and screen tearing.

The user does not own the swapchain images, but can use them after they are acquired. Acquiring the next image in the swapchain can block, depending on the swapchain settings and implementation. The acquire operation alone does not guarantee that the image can be used immediately, but it can signal a fence or a semaphore, so that commands that depend on it being accessible can be submitted as soon as possible. The queue submission that renders the current frame then signals another semaphore, which a separate present operation waits on. This operation finally queues the image to be displayed on the surface. Therefore unlike other APIs, the interaction with the swapchain is split into separate acquire and present operations, allowing better control over the rendering process by the application.

The platform specific part of creating a presentable surface is better left to dedicated third party libraries like GLFW. Instead, Tephra only abstracts around the Vulkan swapchain, expecting an already created Vulkan surface object on creation. It also offers an easy way to query surface support of a physical device. The Tephra swapchain object handles the necessary synchronization more or less automatically, issuing the necessary semaphores with only an extra bit of information from the user about the particular queue that an acquired image will be used in.

Chapter 4

Implementation details

The implementation of the interface is designed to be easily extendable and modular. It spreads to over 100 files and 200 classes. Heavyweight objects like `Device` are implemented as containers of specialized components, handling a specific part of the functionality. The `Device` container, for example, is composed of a logical device that wraps around Vulkan's logical device, a physical device storing the properties of the logical device, a memory allocator wrapper around VMA, a workload tracker for tracking submitted workloads and issuing callbacks when they finish, a deferred destructor ensuring resources get freed only when its safe to, a global command buffer pool and a synchronization state for each queue. The container itself only implements getters for the components and has very little functionality of its own.

This detail is completely hidden from the user of the API. Most classes of the interface use static polymorphism to dispatch calls to their implementation class that inherits from the interface class. There is a layer of dispatch methods that do nothing more than issue calls to the implementation class. This presents a good place to add validation to check whether the API is being used correctly, failing early and in an obvious manner. All API calls that can issue debug messages or throw an exception are wrapped in a debug context that is able to log the name of the function and the class where the warning or error took place. This is useful when they originate from Vulkan, making it easier to see what Tephra calls caused it without the use of breakpoints. Most class implementations in Tephra contain a `DebugTarget` object that references the user provided debug handler, the type and name of the object and the type and name of the parent object it was created from. All of these features can be disabled through compile flags, leaving minimal

to no overhead when not needed.

Listing 4.1: Example of a dispatch method implementing Device interface

```
Workload Device::submitQueuedJobs(DeviceQueue queue) {
    auto deviceImpl = static_cast<DeviceContainer*>(this);
    TEPHRA_DEBUG_SET_CONTEXT(
        deviceImpl->getDebugTarget(),
        "submitQueuedJobs(" + queue.getName() + ")"
    );

    QueueState* queueState = deviceImpl->getQueueState(queue);

    #ifdef TEPHRA_ENABLE_DEBUG_TEPHRA_VALIDATION
        if (queueState == nullptr) {
            reportValidationError(
                "'queue' is an invalid DeviceQueue handle."
            );
        }
    #endif

    return queueState->submitQueuedJobs();
}
```

To prevent freeing of Vulkan objects before it's legal to do so, most Vulkan handles are wrapped in a handle lifeguard object. When this object is destroyed, it either releases the handle directly or queues it for deferred destruction, depending on the type of the object. The deferred destructor keeps a queue of handles to release of each type. When a new workload is submitted, the current queue of handles is stored to be released once the workload and all workloads submitted before it finish executing. Resource demanding objects like buffers and images are an exception. Those keep track of the last workload they've been actually used in to minimize the wait time until destruction. The implementation does not actively wait for the fences associated with each workload to notice they are done. Instead, the workload tracker's `update()` method is called during key moments, like when a job gets enqueued or before memory allocations. It can also be triggered manually through `Device->updateWorkloadCallbacks()`.

4.1 Jobs and job-local resources

Jobs need to be able to record commands into its own format to be then later recorded into a Vulkan command buffer once all the resources used are created and known. For efficiency, the memory for commands is allocated from a list of large blocks of memory that the commands fill in a linear fashion. When a command is to be recorded, a `CommandMetadata` structure is stored at the next available space, containing the type of the command and pointer to `CommandMetadata` of the next command, initially set to `nullptr`. The last command's pointer is updated to the new one, forming a linked list of commands. The command is then able to record the details of the command to a command type specific structure into the immediately following memory in the buffer. Variable sized arrays that some commands accept as parameters can also be stored within the buffer.

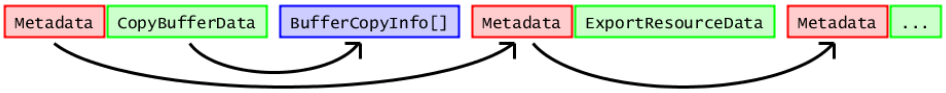


Figure 4.1: Data layout of recorded job commands in linear memory

Job-local resources need to be used in commands before they are even created. This requires an extra class that implements most of the `Buffer` or `Image` interfaces, redirecting the calls to the underlying resource when it's present, or throwing an error when it's not. It needs to be able to hold onto the parameters for creating the underlying resource until the moment it can be created. The API returns the same resource view for both persistent and job-local resources, so that this is handled transparently to the user. Resource views therefore store a flag that says what implementation they are a view of and the dispatch function dynamically chooses which implementation to use.

Resource views often depend on the corresponding Vulkan view handle, which can also only be created after the underlying resource has been assigned. Because the underlying resources are meant to be reusable between multiple jobs, creating identical views of them for every job would be wasteful. Therefore, persistent resource implementations that are used as underlying resources contain a cache of the Vulkan view handles created so far, reusing them whenever possible. Care must be taken to ensure all of the offsets are correct due to the amount of indirections. Resource views can refer to a range of a job-local resource, which in turn refers to a range of its underlying persistent resource, kept by the `JobResourcePool`.

4.1.1 Job-local resource allocation

To reduce the number of created resources and enable better reuse of memory between jobs, job-local resources are suballocated from larger persistent resources. This is especially the case for buffers. When the user allocates several smaller job-local buffers of the same type and memory preference, only a single underlying buffer is created to host all of them. This buffer is then reused in the following jobs. A large part of these suballocators is the automatic aliasing of memory. When two buffers are known not to be used at the same time, they can be assigned to the same range of the underlying buffer, reducing the memory footprint. During the time commands are recorded, the implementation keeps track of the indices of the first and last commands each resource was used in.

Once the job-local buffer allocator has a list of buffers to allocate, it can sort them into groups by compatibility, depending on the usage and memory preference. Once sorted, these buffers need to be assigned to a list of existing underlying buffers, each of a fixed size. Any leftover buffers will be accommodated by a single new persistent buffer. For the aliasing suballocator, we can arrive at the following simplified formulation. Each buffer of index $i = 1, ..., n$ to be allocated can be represented by $b_i \in \mathbb{N}; e_i \in \mathbb{N}; b_i \leq e_i$ being the beginning and end of its usage respectively, and $s_i \in \mathbb{N}$ as its size in memory. The goal is to then find $o_i \in \mathbb{N}$ representing the offset in memory that the buffer will be assigned to, such that no two buffers with overlapping usage also overlap in memory. Given those requirements, the total size of the underlying buffer $S = \max_{1 \leq i \leq n} (o_i + s_i)$ is to be minimized.

A simple algorithm for finding an approximate solution to the problem in a reasonable time was found. It is a greedy incremental algorithm that assigns the best possible spot to each buffer one at a time. For each such buffer, it iterates over all already assigned buffers for ones with overlapping usage. Starting with $o_i = 0$, an already assigned buffer then splits the viable offset search space into two halves, one to the left and one to the right. If the assigned buffers are sorted by their starting offset, then we are able to tell whether the current buffer will fit into the space to the left, that is, $o_i + s_i \leq o_j$, since we know all buffers with a smaller offset have already been visited. In that case the buffer can be assigned to an offset o_i. Otherwise, we set $o_i = o_j + s_j$ where j is the index of the already assigned buffer and continue with the offset search space $[o_i, \infty)$.

This algorithm gives the best results when the input buffers are sorted by their size, largest first. This is because smaller buffers are more likely to fit to the left of the already assigned, and therefore larger, ones. When

working with an underlying buffer of a fixed size, we also check whether a new buffer fits to the right. If it doesn't, then it cannot be assigned at all. Additional slight complexity comes from memory alignment requirements, where the offset needs to be a multiple of some amount of bytes, depending on the usage.

For job-local images this reuse is greatly limited. A backing image is compatible with a requested one only if they share the same type, usage, extent, mip level count, sample level count, creation flags and format compatibility class. Even then, opportunities for reuse can arise between various render targets. The allocator is able to assign multiple single layer images into a backing image with multiple layers using the same suballocator as job-local buffers, except operating on layers rather than byte offsets. There is a possibility in the future to reuse the underlying memory by creating multiple otherwise incompatible images that alias to the same memory range. However, many platforms observe lower performance when images used as render targets do not have their own dedicated memory allocation.

4.1.2 Preinitialized buffer allocation

Because the preinitialized buffers is not limited to the execution of the job, but extends before it, it cannot be allocated using the existing job-local resource allocator. If multiple consecutive jobs are submitted, each of them having their own preinitialized resources, they can't occupy the same region of memory, unlike other job-local resources. Because jobs created from the same resource pool will execute sequentially, this calls for a ring buffer allocator. One issue is that we do not know the required size of the ring buffer ahead of time and it needs to be able to grow with usage requirements.

This growable ring buffer was implemented using a collection of fixed size ring buffers for storage. When a new allocation is to be made, the last ring buffer that has been allocated from is checked if it has enough free space. If it doesn't, other buffers are checked in a circular fashion. In a way it becomes a ring buffer of ring buffers. The one scenario where the space isn't used optimally is when the head offset is near the end of the buffer, but the new allocation does not fit and has to be allocated at the beginning of the buffer. Then the space at the end is treated as used even though a smaller allocation could fit. The impact of this has been minimized by sorting the allocations by size in an increasing order, similarly to how other job-local buffers are sorted in a decreasing one.

4.1.3 Descriptor set allocation

Similarly to other resources, job-local descriptor sets are created after the job is enqueued, once all job-local resources it refers to are allocated. `DescriptorSet` objects act like views of resources in the way that they are just pointers to real handles stored in a `DescriptorPool`. Job-local descriptor sets simply point to a handle that will get assigned a handle at a later point. The job-local descriptor set allocator does nothing more than stores the descriptor setups and allocates all of the descriptor sets at once using the job resource pool's descriptor pool.

The persistent descriptor pool allocator itself is a little more interesting. Since, unlike Vulkan's descriptor pools, it needs to be able to hold arbitrary amount of descriptors of each type, it must be able to grow as needed, as well as reuse previously freed descriptor sets. When new descriptor sets are to be allocated, a list of free descriptor sets of the same layout are checked first and reused by updating their contents. For the leftover sets, a new Vulkan descriptor pool is created to fit them. Descriptor sets are updated through Vulkan update templates, which offer a more efficient way to do so, directly from the descriptor setups provided by the user, so that the amount of copying of data is minimized.

4.2 Automatic synchronization

Synchronization in Vulkan can be solved by keeping a track of accesses to memory. A dependency can then be formed between a new access and all previous accesses of overlapping memory regions. These accesses can either be inferred directly from the commands submitted to the job, or from the ones specified by the user through export operations or in command passes. Once a dependency is identified, a good way to insert a barrier needs to be found to synchronize that dependency.

The first implementation that worked only on buffers was inserting global memory barriers, which synchronized two types of accesses against all of memory. This is said to have less overhead than using many resource specific buffer and image barriers. However, once images were introduced, that was no longer possible, because image layout transitions require image barriers and are limited to a particular subresource range of the image.

The current implementation uses only resource specific memory barriers for synchronization. It is split into two parts according to the process described in the first paragraph. There is an access map, which for a single resource maintains a map of past accesses, using them to define a dependency. The barrier list then serves to synchronize these dependencies, searching through the already inserted barriers for an optimal spot to synchronize the dependency.

4.2.1 Buffer access maps

Several observations can be made to help the access map find dependencies efficiently. Let's consider a simplified case where all accesses happen on the same subresource range. A new access only forms a dependency to all previous accesses since (and including) the last write access. This is thanks to dependency chaining. The last write access must have already had a dependency formed to all previous accesses. Therefore, a dependency to this last write access also implies a similar dependency against all previous accesses. This is true for read only accesses as well, but an execution only dependency against it only implies execution dependency against the previous accesses, which is not enough to synchronize against a previous write access.

So, this means that the synchronisation state for such an indivisible subresource range can be fully represented by storing the last write access as well as the mask of all read only accesses that happened since. An additional read access that is not yet included in the mask forms an execution and memory dependency against the last write access. A new write access forms an execution and memory dependency against the last write access and an execution dependency against the read accesses. It can then overwrite the write access and reset the read accesses.

Extending this for buffers where the subresource range can be any contiguous region in the buffer means implementing a range map, where each point in the buffer is mapped to at most one entry, describing the current synchronization state for that point. An efficient implementation of range maps can be a simple sorted list. The entry that a given point belongs to can then be found in $\mathcal{O}(\log n)$ time. Finding all intersecting ranges for a new range then involves looking up the range containing its two end points. The intersecting ranges are all the ranges between those two in the sorted list.

The process of inserting a new access differs from whether the access is read only or not. If it is a write access, it replaces all other accesses that are fully

covered by it in the access map. Partially covered accesses are clipped. This ensures that after insertion each point is still only covered by a single access entry. Read accesses would ideally split any existing accesses and extend only the range they actually cover, but that could quickly lead to too many entries in the access map. Instead, the implementation simply extends the read masks of all intersecting access ranges, so that read accesses have very little overhead. This can sometimes lead to false write-after-read dependencies, but it should be rare enough to be worth the savings in storage and performance.

4.2.2 Image access maps

Image access maps are similar to their buffer counterparts. The accesses also store the current layout of the image subresource. Image layout transition, when needed, is defined to be a memory operation in Vulkan. Therefore an otherwise read-only access will need to be synchronized as if it is also a write access if the dependency it forms includes a layout transition. Additional complexity stems from the fact that image subresource ranges are defined by two independent values, the array layers and mip levels. Since the number of mip levels is limited (less than 16), the array layers are used for purposes of the range map. Intersection tests still consider the mip levels of the access, however.

This means a single layer in the resource can be covered by multiple subresource ranges, even if they do not overlap in their mip levels. This is not problematic for the sorted list implementation as long as all of the overlapping ranges are identical in the range of array layers they cover. To reduce the number of image subresource ranges that need to be stored, the mip levels are stored as a bit mask, lifting the usual requirement of the range being contiguous for mip levels. Upon processing a new access, the algorithm first finds all potentially intersecting ranges through the range map and filters out all of those that do not intersect in the mip levels.

Any resulting dependencies by read-only accesses that require a layout transition are handled separately. Layout transition happens before the actual access, as a write operation. It therefore "promotes" the dependency to previous accesses as if the destination access was a write access. This layout transition is then recorded as a special write access followed by the original read access. The rest is handled similarly to the buffer access map.

4.2.3 Queue state

These access maps need to persist in between jobs, so they have to be stored in some way. In Vulkan, resources are owned by a single queue family that has exclusive access to it unless transitioned. Tephra extends this concept by letting individual queues own resources, with the transition operations being exports. This allows resource accesses to be stored on a per-queue basis, allowing multiple threads to submit commands to different queues in parallel without the need for any locks, as long as they don't use the same subresource ranges at the same time. The queue state is not only responsible for keeping access maps for each resource, but also handles the queueing and compilation of jobs as a whole as well as communication with other queues through timeline semaphores and export operations.

4.2.4 Barrier list

The second part of the process is translating the dependencies found by the access map into barriers. Dependencies operate over a contiguous range, the intersection of the source and destination accesses. A buffer or image memory barrier can be used in a pipeline barrier to synchronize that dependency. The goal here is to minimize the number of memory barriers and pipeline barriers, while synchronizing the dependency as late as possible.

This is achieved by keeping a simple list of barriers to be inserted into the command stream. When a new dependency is to be synchronized, it needs to know what the index of the first barrier to be reused is. This is stored in the access map entry as the index of the barrier immediately following the last write or read access. The barrier list is then iterated over from that index, looking for the optimal barrier.

The ideal candidate is a pipeline barrier that already has a memory barrier that covers the new dependency. Since the access map has knowledge of the relevant memory accesses and how they have been synchronized, the access entry can also store the reference to this specific memory barrier. This is useful when multiple different read accesses follow a write access. They can then reuse the same memory barrier to synchronize both of the read accesses at once.

Where a new memory barrier needs to be inserted and has multiple existing pipeline barriers to choose from, it should choose one that already satisfies

its execution dependency, prioritizing the last one if there are more. Such a pipeline barrier only needs to be extended by a memory barrier, without introducing any new sync points. If no pipeline barrier has the right execution dependency, the latest existing pipeline barrier is extended to cover it. Whether this is better than inserting a new barrier at a later point is arguable. On one hand it serves to minimize the amount of barriers, but it might insert a dependency sooner than it is really needed. Further research and profiling would be beneficial.

4.2.5 Job compilation and submission

We can now piece together the whole process of submitting a list of enqueued jobs. This is better shown with the following pseudocode:

Listing 4.2: Pseudocode of the process of submitting a list of enqueued jobs

```
QueueSyncState& syncState = getSyncState(queue);
for (auto job : queuedJobs) {
    VkCommandBuffer cmdBuffer = commandPool->acquireCommandBuffer();

    // Update sync state and form barriers
    BarrierList barriers;
    for (auto command : job.commands) {
        std::vector<Accesses> accesses = identifyAccesses(command);

        // First synchronize with previous accesses
        for (auto access : accesses) {
            AccessMap& accessMap = syncState[access.resource];
            syncState.synchronizeAccess(access, &barriers);
        }

        // Then insert the accesses for the following commands
        // to synchronize against.
        for (auto access : accesses) {
            AccessMap& accessMap = syncState[access.resource];
            syncState.insertAccess(access);
        }
    }

    // Record the Vulkan command buffer
    for (auto command : job.commands) {
        while (barriers.front().commandIndex <= command.index) {
            recordBarrier(barriers.front(), &cmdBuffer);
            barriers.pop_front();
        }

        // Convert Job command buffer to Vulkan's, resolving
```

```
        // resource views to actual Vulkan handles
        recordCommand(command, &cmdBuffer);
    }

    submitInfo.append(cmdBuffer);
}
Workload workloadHandle = deviceImpl->submit(queue, submitInfo);
```

This is skipping over details like handling export commands, which insert "fake" read accesses right before the next point where exported resources can be used outside of the job. That is, before command pass execution and at the end of the job. `DiscardImageContents` commands set the current layout of all overlapping accesses in the access map with an undefined layout. The cached barrier references inside the affect map also need to be reset after a job submission.

4.2.6 Render pass synchronization

Some synchronization already needs to be resolved at the time the Tephra render pass layout is created. Vulkan expects an explicit description of dependencies between subpasses, even if that dependency is formed by the attachments themselves. The actual stage and access masks as well as the image layouts are therefore inferred by the library according to the information about the usage of attachments during each subpass. It follows rules described by Vulkan specification as for what layouts are permitted when a single attachment is used as, for example both an input and a read only depth attachment at the same time.

There are two aspects in render passes that differentiate it from other commands from the perspective of the job compiler. Both stem from the fact that in Vulkan, render passes themselves do some amount of synchronization. Individual subpasses can use the attachments differently and can even perform image layout transitions. Therefore, its attachment accesses have to be synchronized as two sets of accesses. The first set is the first usage of each attachment in the render pass and it is what past accesses need to be synchronized against. The second set are the last usages, which is what future accesses will depend on.

The other aspect is that a render pass can act as a barrier with its external dependencies. They, however, behave only like global memory barriers for non-attachment images, so there will be scenarios where regular pipeline barriers

will need to be inserted alongside the render pass anyway. For now, Tephra simply ignores this aspect, leaving external dependencies empty and relying only on the more powerful pipeline barriers. However, in the future adding the support for leveraging external subpass dependencies where possible may provide some benefit, since the driver might be able to implement them more efficiently.

Chapter 5

Evaluation

It is difficult to objectively measure many qualities of an interface, like ease of use, consistency and how intuitive its design is. Only time will tell how well it integrates into existing engines and how quickly users without prior experience are able to become familiar with the library and its best practices. One aspect that can be analyzed is the number of lines of code needed to implement the same program as compared to different APIs. Even though it is a very rough measure, it should be telling of how much information the interface asks of its user and therefore roughly how "low level" it is. Something as simple as rendering a triangle can be expressed in a few tens of lines in OpenGL, but the same example can get close to a thousand lines of code in Vulkan.

What can be measured directly is the performance. The overhead as compared to using Vulkan directly should be as small as possible. A small memory footprint is welcome, too. Finally, we can take a closer look at the synchronization primitives automatically inserted by the library and discuss how close they are to an optimal setup.

5.1 The tests

For the sake of comparison, the same demos need to be implemented in OpenGL, Vulkan and Tephra with identical results. A third party test suite [7] comparing the performance of OpenGL and Vulkan-Hpp [12] was used as

the basis. It was then extended to include a Tephra implementation for a three way comparison. Some additional modifications to the test scenes were also made to further focus on specific areas of the APIs.

The first scene is a static scene made up of a number of randomly colored low-poly balls. It is meant to highlight the overhead of the API calls. For that reason the demo does not use instancing. The low triangle count and small size on the screen ensures the performance will be limited by how fast the CPU can generate the commands. The scene is first rendered with a small number of balls (1000) to measure the flat overhead of each API when drawing a mostly empty scene. The second test draws the scene with a million balls, such that the vast majority of the time will be spent on the individual draw calls in all APIs. This test is run both with a single threaded and multithreaded implementation for Vulkan and Tephra. OpenGL The original test suite has a multithreaded OpenGL implementation as well, however the threading is only applied to the position updates, which doesn't run in this static version.

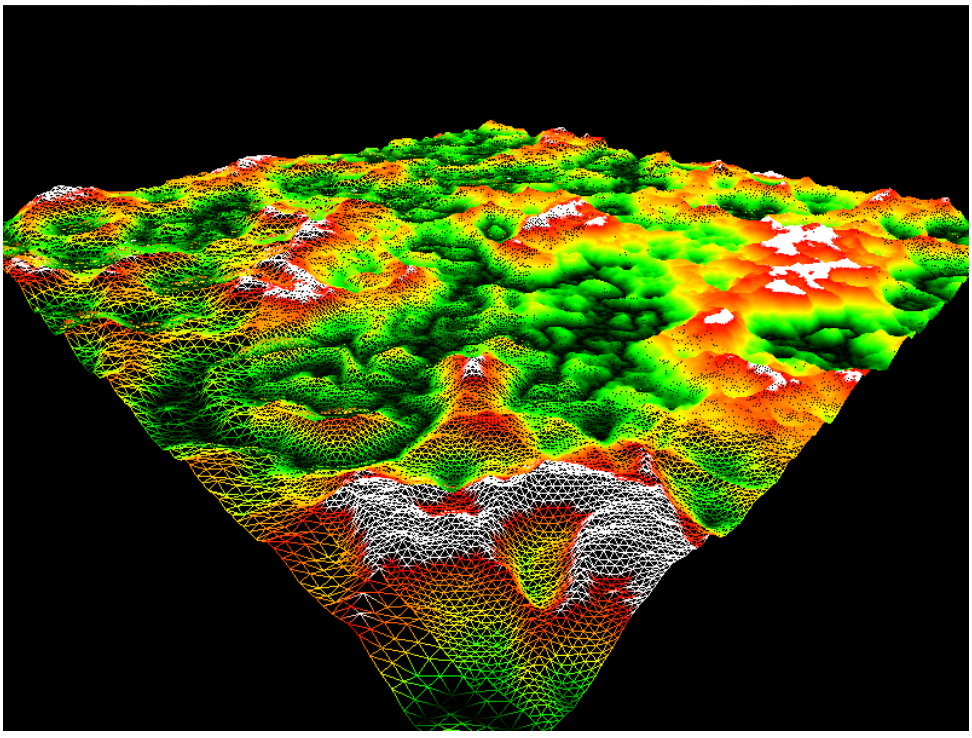


Figure 5.1: Screenshot of scene #2

The second scene draws dynamic geometry of a terrain with an adaptive level of detail. It's rendered as a wireframe with each fragment colored according to its height. This is scene is largely CPU bound as well due to each segment being a separate draw call. The traversal of the quad tree has a considerable cost as well, irrespective of the API used. In the sample, Vulkan has a multithreaded version for this scene, but we did not notice a significant

enough improvement on any platform.

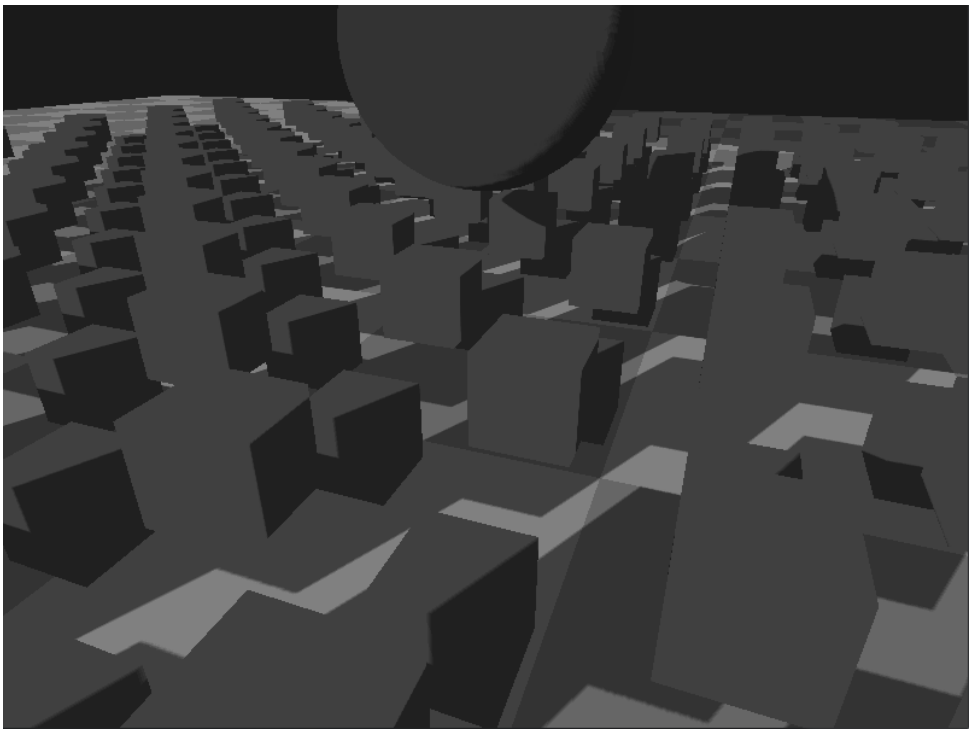


Figure 5.2: Screenshot of scene #3

The third and final scene is a more complex render, closer to a real use case. It first renders a shadowmap of the scene made up of a few hundred boxes and a sphere, which is then used for shading during the draw pass from the perspective of a camera rotating around the scene. This is mostly a GPU bound scene and therefore tests the efficiency of barriers and render passes. It is used for two tests, one with a shadowmap resolution of 4096x4096 pixels and the other with 1024x1024 pixels.

For the performance tests, each of them was run for a period of 60 seconds to capture the average frame time. The first second of data was discarded to stabilize the frame rate after initialization. The initial results of the first scene showed Tephra 10-15% faster than Vulkan-Hpp. Short investigation revealed that was because the implementation in the original sample uses static pointers for the API functions, whereas Tephra always retrieves them dynamically for the particular device they will be used on, reducing overhead. For the sake of fairness, the Vulkan-Hpp implementation was extended to use dynamic pointers as well. Besides that, it is expected that the performance overhead of Vulkan-Hpp is very low compared to using the Vulkan API directly. All of the tests were ran on three different computers with details described in the following table:

Name	Operating system GB of memory	GPU CPU [# of cores / threads]
Nvidia laptop	Windows 7 8 GB	Nvidia GT 650M Intel Core i5-4200M @ 3.1 Ghz [2/4]
AMD desktop	Windows 7 16 GB	AMD Radeon RX 580 AMD Ryzen 5 1600 @ 3.5 Ghz [6/12]
Nvidia desktop	Windows 10 32 GB	NVIDIA RTX 2080 Intel Core 15-5600K @ 3.7 Ghz [6/6]

Table 5.1: Tested platforms

5.2 The results

	Vulkan-Hpp	OpenGL	Tephra
Scene #1 with 1 000 objects, single threaded			
Nvidia laptop	0.599 ms	0.840 ms (140%)	0.671 ms (112%)
AMD desktop	0.223 ms	0.827 ms (371%)	0.251 ms (113%)
Nvidia desktop	0.268 ms	0.321 ms (120%)	0.288 ms (107%)
Scene #1 with 1 000 000 objects, single threaded			
Nvidia laptop	73.6 ms	228 ms (310%)	73.9 ms (100%)
AMD desktop	136 ms	665 ms (489%)	149 ms (110%)
Nvidia desktop	43.9 ms	163 ms (371%)	25.8 ms (59%)
Scene #1 with 1 000 000 objects, multithreaded			
Nvidia laptop	73.5 ms	N/A	73.6 ms (100%)
AMD desktop	151 ms	N/A	151 ms (100%)
Nvidia desktop	11.0 ms	N/A	11.2 ms (102%)
Scene #2			
Nvidia laptop	8.41 ms	11.9 ms (141%)	8.47 ms (101%)
AMD desktop	11.7 ms	25.6 ms (219%)	11.7 ms (100%)
Nvidia desktop	7.13 ms	8.99 ms (126%)	7.08 ms (99%)
Scene #3 with a 4096x4096 shadowmap			
Nvidia laptop	8.97 ms	4.02 ms (45%)	8.57 ms (96%)
AMD desktop	1.58 ms	2.60 ms (165%)	0.839 ms (53%)
Nvidia desktop	0.611 ms	0.563 ms (92%)	0.627 ms (103%)
Scene #3 with a 1024x1024 shadowmap			
Nvidia laptop	3.34 ms	2.43 ms (73%)	2.81 ms (84%)
AMD desktop	0.854 ms	2.54 ms (297%)	0.618 ms (72%)
Nvidia desktop	0.444 ms	0.565 ms (127%)	0.475 ms (107%)

Table 5.2: Frame times in milliseconds for each test, platform and API

As was mentioned, the first test that draws 1 000 static objects in scene #1 reveals the API overhead for each frame, independent on the number of draw calls. Tephra is here consistently around 10% slower than Vulkan on all platforms. Caching the render pass and framebuffer objects so that they are not recreated every frame may help in this area. Moving on to a million draw calls, the per-draw call overhead starts to dominate. Tephra's performance in this test is rather inconsistent on different computers, showing even an unexpected significant speedup over Vulkan in one case. What is consistent, however, is the much higher overhead of OpenGL, with the scene taking 3 to 5 times longer to render.

The same scene with multithreading removes these inconsistencies, making Tephra just as fast as Vulkan in this test. Of note is the fact that Vulkan does not benefit from multithreading on Windows 7, making only one of the tested systems show improved frame times over the previous test.

Next up is the second scene, with Tephra's performance staying close to that of Vulkan, while OpenGL trails behind. Another observation unrelated to the APIs is that the CPU cost of draw calls is higher for AMD drivers, since the tests show lower frame rates on these tests than on the Nvidia laptop, despite the AMD system's CPU being more powerful.

The third scene shows a scenario where the GPU is more likely to be the bottleneck. Here the results are once again inconsistent. It seems that OpenGL is limited by the CPU instead on two of the platforms, where reducing the resolution of the shadowmap has little effect. Tephra, however, tends to perform better than the Vulkan implementation, especially on the AMD system. This can be because of synchronization being set up differently, as will be investigated in the following section, or due to the vertex data being suballocated from a single Vulkan buffer, rather than one for each render object.

	Vulkan-Hpp	OpenGL	Tephra
Scene #1 1 000 objects	56.3 MB	93.9 MB (167%)	61.6 MB (109%)
Scene #1 1 000 000 objects	358.7 MB	95.4 MB (27%)	465.2 MB (130%)
Scene #2	232.9 MB	259.6 MB (111%)	298.5 MB (128%)
Scene #3 4096x4096 shadowmap	59.1 MB	93.6 MB (158%)	65.3 MB (110%)
Scene #3 1024x1024 shadowmap	59.3 MB	73.9 MB (125%)	65.3 MB (110%)
Average	100%	119%	117%

Table 5.3: Process CPU memory usage

	Vulkan-Hpp	OpenGL	Tephra
Scene #1 1 000 objects	13.3 MB	26.4 MB (198%)	13.6 MB (102%)
Scene #1 1 000 000 objects	13.8 MB	26.4 MB (191%)	14.1 MB (102%)
Scene #2	61.2 MB	72.3 MB (118%)	61.3 MB (100%)
Scene #3 4096x4096 shadowmap	85.5 MB	91.4 MB (107%)	83.2 MB (97%)
Scene #3 1024x1024 shadowmap	24.6 MB	30.5 MB (124%)	22.3 MB (91%)
Average	100%	148%	98%

Table 5.4: Dedicated GPU memory usage

Both CPU and GPU memory usage of the process was also measured for all scenes on the same system. Several interesting observations can be made about the differences between how the Vulkan and OpenGL drivers use memory, but of more importance is the overhead of Tephra compared to Vulkan. For the purpose of these tests, the Vulkan Memory Allocator for Tephra was disallowed from pre-allocating large blocks, so that the GPU memory usage of the process as reported by the Windows Process Explorer is more accurate. In the results, Tephra ended up balancing between 110% and 130% of the CPU memory usage of the Vulkan-Hpp implementation. Its GPU memory usage stayed very similar, except for the last scene, where Tephra has a slight lead. This is most likely due to the suballocation of vertex data from a single buffer, therefore caused by a difference in the scene implementation rather than the API itself.

	Vulkan-Hpp	OpenGL	Tephra
Common setup	180	76 (42%)	97 (54%)
Scene #1	130	31 (24%)	61 (47%)
Scene #1 multithreaded	163	N/A	77 (47%)
Scene #2	140	35 (25%)	80 (57%)
Scene #3	312	71 (23%)	135 (43%)
Average	100%	28%	49%

Table 5.5: Logical lines of code for each scene and API

Another measured aspect is the number of lines of code needed for the implementations of the test scenes for each API, as a rough measure of how high level they are. Using Vulkan-Hpp over the C Vulkan API arguably makes for a better comparison against Tephra, since they are both C++ interfaces leveraging modern language features to reduce the amount of boilerplate code, making the major difference in line count the result of how much information is being asked of the user.

To account for the differences in coding styles, the measured metric is the number of logical lines of code, rather than the actual number of lines of source code. Only the lines of code that directly interface with the API are counted, formatted to a maximum of 120 characters per line. Lines without code, class and function definitions, non-API function calls, timing and error handling code are all discarded and are not a part of the measured code count. This is all to get the best measure of how much code is needed to work with each of the interfaces.

The common setup code includes initialization, creating the context and a window for OpenGL and the instance, device, surface and swapchain for Vulkan and Tephra. This setup is shared among all of the scenes. Looking at the results, both OpenGL and Tephra require about half as much code for initialization as Vulkan-Hpp. The scenes themselves have a lot less code for OpenGL at about a quarter of what Vulkan-Hpp needs, while Tephra still hovers around the 50% mark. Calculating a weighted average of these results nets OpenGL 28% and Tephra 49% of the Vulkan-Hpp amount of code. If the setup code is included in each of the scenes as if they were standalone applications, it gives us 33% for OpenGL and 50% for Tephra.

5.3 Synchronization analysis

Analysing the results of the automatic synchronization is rather tricky. Vulkan's validation layers should issue a warning when something is amiss, but they don't have close to full coverage. Even then, there are no warnings for unnecessary barriers. Evaluating the results depends on the knowledge and understanding of what should be the optimal way of synchronization for each scenario, knowledge that I don't believe I fully possess due to the many caveats.

The correctness of the automatic synchronization help confirm several tests that are a part of Tephra. They show example scenarios with consideration of how pipeline barriers should optimally be placed and confirms that is how the library inserts them. Here is one such example along with comments explaining the purpose of each command. The test checks whether exactly two pipeline barriers were inserted as specified.

Listing 5.1: Synchronization test confirming no needless barriers get inserted

```
job->cmdFillBuffer(bufferA, 0, bufferSize, ...);
// Test that only one read after write barrier gets inserted
// for multiple non-overlapping reads (Barrier #1)
job->cmdCopyBuffer(bufferA, bufferB, {...});
job->cmdCopyBuffer(bufferA, bufferB, {...});
// Test read after read non-dependency
job->cmdCopyBuffer(bufferA, bufferC, {...});
// Test non-overlapping regions non-dependency
job->cmdFillBuffer(bufferC, bufferSize / 2, bufferSize / 4, ...);
job->cmdFillBuffer(bufferC, 3 * bufferSize / 4, bufferSize / 4, ...);
// Do put write after read execution barrier here (Barrier #2)
job->cmdFillBuffer(bufferA, 0, bufferSize / 2, ...);
// Augment the previous barrier with another write after read memory
// barrier without inserting a new one
job->cmdCopyBuffer(bufferA, bufferA, {...});
// Also augment the previous barrier to synchronize a copy between
// two unrelated buffers
job->cmdCopyBuffer(bufferB, bufferC, {...});
```

The third scene in the comparison suite can also be investigated. The Vulkan-Hpp implementation of the scene as done by the author does not synchronize optimally. Unlike Tephra, it uses external subpass dependencies for synchronization, however they are overreaching and each render pass is

flushing the entire pipeline, which may be the cause of its reduced performance on some platforms. Additionally, it still inserts a manual pipeline barrier in between the render passes despite the dependency being able to be expressed by the external subpass dependencies.

Tephra handles it entirely through manual pipeline barriers. The first barrier comes before the shadowmap pass. It synchronizes the previous frame's shader read of the shadowmap to this frame's depth write. It also inserts a similar one that acts on the depth image of the render pass here rather than in the following barriers because this barrier already forms the appropriate execution barrier. Doing it later could stall the pipeline more. The second barrier lies in between the render passes and ensures the shadowmap becomes visible from the shader. It also transitions the swapchain image to a layout optimal for color attachments after it's been presenting on the screen. The final barrier after the render pass simply does the opposite, preparing the image for presentation.

An optimal synchronization setup would rely on external render pass dependencies here as it should be possible to do so without using any pipeline barriers. However, the actual dependencies expressed would most likely stay the same as with the pipeline barriers, so the benefit depends only on how much more efficient this style of synchronization is. This part deserves more testing in the future.

Chapter 6

Conclusion

In this thesis, it has been shown that there is still a place for a modern, high-level graphics API. Key design elements have been established along with the requirements and qualities such an interface should have. This design has been realized in detail to create a general-purpose graphics API called "Tephra" and then implemented using the low level Vulkan API to communicate with the GPU.

This implementation underwent testing and comparison against an established high-level graphics API, OpenGL, as well as Vulkan itself through a thin abstraction library. All of the tests implemented using Tephra required half as much code compared to the low-level Vulkan library, while only incurring less than 10% of performance overhead. It also maintains other advantages of Vulkan besides the low draw call overhead, such as the ability to scale well with multithreading and more control than what OpenGL provides. At the same time it does not require the user to learn the rules of synchronization and pipeline barriers like Vulkan does. This can lead to a Tephra implementation showing a higher performance than a hand written Vulkan implementation, as was the case in one of the test scenes.

There are still improvements that can be made, however, both to the automatic synchronization as well as yet unimplemented features. This is an ongoing effort with the goal of producing a polished, well tested and feature complete graphics API that is able to present itself as a serious candidate for the graphics API of choice for many applications, bridging the gap between the aging last generation APIs and their difficult-to-master successors. We feel this work has been successful in approaching that goal.

Appendix A

Bibliography

[1] Advanced Micro Devices. Anvil. https://github.com/GPUOpen-LibrariesAndSDKs/Anvil, 2016.

[2] Advanced Micro Devices. Vulkan Memory Allocator. https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator, 2017.

[3] Advanced Micro Devices. V-EZ. https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ, 2018.

[4] Hans-Kristian Arntzen. Render graphs and Vulkan - a deep dive. http://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive, Aug 2017.

[5] Karlsson Baldur. RenderDoc. https://renderdoc.org, 2014.

[6] Diligent Graphics. Diligent Engine. http://diligentgraphics.com/diligent-engine, 2016.

[7] Damian Dyńdo. OpenGL vs Vulkan. https://github.com/RippeR37/GL_vs_VK, 2017.

[8] James Jones. Siggraph. https://www.khronos.org/assets/uploads/developers/library/2019-siggraph/Vulkan-04-Timeline-Semaphore-SIGGRAPH-Jul19.pdf, Jul 2019.

[9] Jan Nyorain. Vpp. https://github.com/nyorain/vpp, 2016.

[10] Yuriy O'Donnell. FrameGraph: Extensible Rendering Architecture in Frostbite. https://www.gdcvault.com/play/1024612, Mar 2017.

[11] Rys Sommefeldt. A look at the PowerVR graphics architecture: Tile-based rendering. http://www.imgtec.com/blog/a-look-at-the-powervr-graphics-architecture-tile-based-rendering/, Apr 2015.

[12] The Khronos® Group. Vulkan-Hpp. https://github.com/KhronosGroup/Vulkan-Hpp, 2016.

[13] The Khronos® Vulkan Working Group. Vulkan® 1.1.130 - A Specification, 2016.

Appendix B

Contents of the included CD

1. **readme.txt -** The readme file containing further details for using the CD contents.
2. **Tephra.sln -** The Microsoft Visual Studio C++ solution file for the Tephra library.
3. **Tephra/interface/ -** The Tephra interface files to be included.
4. **Tephra/impl/ -** The Tephra implementation files.
5. **Tephra/documentation/ -** The Doxygen source and html documentation.
6. **Tephra/examples/ -** An example project using Tephra.
7. **Tephra/tests/ -** Integration test project.
8. **GL_vs_VK/ -** A comparison suite between OpenGL and Vulkan extended to test Tephra as well.
9. **GL_vs_VK/run_tephra.bat -** A batch file that runs the prebuilt x64 binary with Tephra implementation.
10. **Thesis/src/ -** Latex source files for the thesis.
11. **Thesis/Tephra.pdf -** This thesis in PDF.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Galajda** Jméno: **Roman** Osobní číslo: **434967**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh moderního vysoko-úrovňového grafického API

Název diplomové práce anglicky:

Designing a modern high-level graphics API

Pokyny pro vypracování:

Design a modern high-level graphics API that improves upon OpenGL and existing high-level interfaces [2-4] in performance or ease of use while avoiding their disadvantages. The design should consider multi-threaded use of the API, as well as leveraging the multiple asynchronous queues that modern graphics cards provide for submitting commands. It should re-evaluate concepts used in existing high-level graphics APIs and how they fit modern desktop and mobile GPU architectures.
Implement the proposed design using the Vulkan API [1] as the back-end for interfacing with GPUs at a low level. Create several example programs, each using the new API, modern OpenGL and Vulkan, and compare the implementations to each other with respect to their CPU and GPU performance and memory usage as well as the number of lines of code needed.
New API is not meant to replace Vulkan for uses where its explicitness and level of control are paramount, neither is it supposed to replace OpenGL, where fast prototyping and an already established community outweigh its downfalls. Instead, its aim is to fill the hole somewhere in the middle, for the projects that require a high performance solution, but whose developers wish to avoid the steep learning curve and large time investment associated with the Vulkan API.

Seznam doporučené literatury:

[1] The Khronos Vulkan Working Group: Vulkan 1.1.99 - A Specification (with all registered Vulkan extensions). February 2019.
[2] Yuriy O'Donnell: FrameGraph: Extensible Rendering Architecture in Frostbite. Game Developers Conference, March 2017.
[3] Diligent Engine - A Modern Cross-Platform Low-Level 3D Graphics Library. January 2019, https://github.com/DiligentGraphics/DiligentEngine
[4] AMD: V-EZ Vulkan wrapper library. Version 1.1.0, August 2018. https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Sloup, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2019** Termín odevzdání diplomové práce: **07.01.2020**

Platnost zadání diplomové práce: **20.09.2020**

Ing. Jaroslav Sloup podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

