Bachelor's Thesis

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Implementation of rendering system in Rust

Eduard Lavuš

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: Open Informatics
Subfield: Computer Games and Graphics
Date: 2020-05-22

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Lavuš** Jméno: **Eduard** Osobní číslo: **474497**

Fakulta/ústav: **Fakulta elektrotechnická**

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

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Implementace zobrazovacího systému v jazyce Rust

Název bakalářské práce anglicky:

Implementation of rendering system in Rust

Pokyny pro vypracování:

Proveďte rešerši metod používaných pro zobrazování virtuálních scén v současných herních enginech.
Vyberte důležitou podmnožinu zmapovaných metod a implementujte ji ve vlastním zobrazovacím systému založeném na jazyce Rust. Pro implementaci využijte 3D rozhraní Vulkan. V práci rozeberte výhody a nevýhody jazyka Rust ve srovnání s jazykem C++ pro implementaci dané úlohy.
Vytvořte demonstrační aplikaci, která ukáže možnosti vytvořeného systému na nejméně třech scénách různé složitosti. Součástí aplikace bude vyhodnocení rychlosti zobrazování (benchmark) a identifikace úzkých hrdel výpočtu.

Seznam doporučené literatury:

[1] Jason Gregory. Game Engine Architecture (3rd edition). CRC Press, 2018.
[2] Tomas Akenine-Moller et al. Real-Time Rendering (4th edition). CRC Press, 2018.
[3] Lagarde, S., and C. D. Rousiers. 'Moving frostbite to physically based rendering.' SIGGRAPH 2014 Conference, Vancouver. 2014.
[4] Daniel Šimek. Rozšiřitelný zobrazovací řetězec založený na odloženém stínování. Diplomá práce ČVUT FEL, 2013.
[5] Tomáš Dřínovský. Nepřímé osvětlení pomocí trasování kuželů. Diplomá práce ČVUT FEL, 2013.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

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

Datum zadání bakalářské práce: **11.02.2020** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **30.09.2021**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to give thanks to doc. Ing. Jiří Bittner, Ph.D. for his guidance and valuable insights. I would also like to thank the open-source community around Rust for making their previous work on this subject openly licensed and available to learn from.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

.....

Abstract

Vulkan is today the de-facto standard for open cross-platform real-time graphics rendering. It allows the developers to leverage the performance of GPUs on many platforms thanks to its uniform design and brings many improvements when compared to the older OpenGL API. Contrary to its predecessor, however, Vulkan is much more low-level and requires more work from the developer. This thesis presents a Rust crate (library) with core wrapper types and a high-level API design for creating and managing Vulkan objects safely in concurrent manner, with minimal impact on performance. Advantages of Rust for this specific task are discussed and the performance and verbosity in comparison to the ash crate (library), which provides raw bindings to the C Vulkan API, is evaluated.

Keywords: Vulkan, Vulkayes, graphics API, GPU, computer graphics, Vulkan API abstraction

Abstrakt

Vulkan je dnes de-facto štandardom pre otvorené mnoho-platformové vykresľovanie grafiky v reálnom čase. Vývojári môžu vďaka Vulkanu naplno využiť silu grafických kariet na mnohých platformách vďaka jeho jednotnému dizajnu. Vulkan naviac prináša mnoho vylepšení v porovnaní so starším OpenGL. Vulkan je však zameraný na omnoho nižší level ako OpenGL a preto vyžaduje viac práce na strane vývojára. Táto práca predkladá knižnicu v jazyku Rust so základnými typmi a vysokoúrovňovým návrhom na vytváranie a spracovávanie objektov Vulkanu bezpečne vo viac vláknovom prostredí a s minimálnym dopadom na rýchlosť. Diskutované sú aj výhody jazyka Rust pre túto konkrétnu úlohu. Naviac je porovnaná rýchlosť a veľkosť kódu voči Rustovej knižnici ash, ktorá poskytuje priame prepojenia do Vulkan API napísaného v jazyku C.

Kľúčové slová: Vulkan, Vulkayes, grafické API, GPU, počítačová grafika, abstrakcia Vulkan API

Preklad názvu: Implementácia zobrazovacieho systému v jazyku Rust

Contents

1 Introduction **1**
- 1.1 Vulkan API overview . . . 1
- 1.2 Vulkan API architecture . . . 2
  - 1.2.1 Execution model . . . 2
  - 1.2.2 Object model . . . 3
  - 1.2.3 Application structure . . . 3
  - 1.2.4 Complexity . . . 5

2 Related work **7**
- 2.1 V-EZ . . . 7
- 2.2 gfx-hal . . . 7
- 2.3 Vulkano . . . 8
- 2.4 Tephra . . . 8
- 2.5 Summary . . . 9

3 Design **11**
- 3.1 Rust . . . 11
  - 3.1.1 Ownership . . . 11
  - 3.1.2 Safety and speed . . . 12
  - 3.1.3 Cargo . . . 13
  - 3.1.4 Generics . . . 13
- 3.2 Object lifetime management . . . 15
- 3.3 Memory management . . . 15
- 3.4 Synchronization and validations . . . 16

4 Implementation **17**
- 4.1 Bindings . . . 17
- 4.2 Cargo features . . . 17
- 4.3 Generics . . . 18
  - 4.3.1 Device memory allocator generics . . . 18
  - 4.3.2 Mappable memory generics . . . 19
  - 4.3.3 Image view generics . . . 19
- 4.4 Abstraction . . . 20
  - 4.4.1 Reference counting . . . 20
  - 4.4.2 Type aliases . . . 21
  - 4.4.3 Deref . . . 21
- 4.5 Swapchain recreate . . . 21
- 4.6 Windowing . . . 23

5 Evaluation **25**
- 5.1 User code . . . 25
- 5.2 Benchmark . . . 26
  - 5.2.1 Stages . . . 26
  - 5.2.2 Results . . . 28
- 5.3 Safety . . . 35

6 Conclusion **37**

Bibliography **39**

Contents of the included CD **41**

1 Introduction

Since its release in 2016, Vulkan API[1] has been gaining traction as a go-to API for high-performance realtime 3D applications across all platforms. The main reason for this, apart from being cross-platform, is that Vulkan is designed as to be low-level, close to metal and with minimal overhead. This, in contrast to Khornos' older API OpenGL, leaves most of the overhead, but also complexity, to the user of the API. The user can then make decisions on where to sacrifice performance for added usability or vice versa.

This project aims to design a flexible, usable and performant wrapper on top of Vulkan API in the Rust language. It aims to provide statically upholdable invariants that are easy to break in C language. It aims to add minimal required overhead to ensure basic memory safety that is the core concept of the Rust language. The name is a play on the Rust library the project is inspired by, the Vulka**no**[2] library.

1.1 Vulkan API overview

Vulkan API, originally released in 2016[3], is a specification of an open API for high-efficiency, cross-platform access to graphics and compute on modern GPUs.

It is designed to minimize the overhead between the user application and the hardware device. Vulkan achieves this by staying low level and explicitly requiring all relevant state to be referenced by the user application, minimizing required lookups and orchestration on the driver side. This allows the user application to optimize for their specific usecase instead of relying on the driver to guess the correct strategy. However, it requires much more complexity from the user application and is much harder to master than OpenGL.

One of the reasons for Vulkans popularity is that it was designed in an intense collaboration between leading hardware, game engine and platform vendors[3]. This resulted in a lot of vendors having zero-day support for the specification in their drivers and software and it being immediatelly adopted as a native rendering solution on many platforms.

The openness of Vulkan also goes hand-in-hand its cross-platform capabilities. Vulkan is available on all three major desktop platforms (Linux, macOS, Windows) and both major smartphone platforms (Android, iOS), but also on many smaller and embedded platforms. This allows applications to easily target multiple platforms with minimal variance in the rendering code. It also prevents vendor locks as seen with DirectX or Metal APIs. Lastly, it allows the community of both professionals and hobbyists to participate in the standard itself and improve it.

One of the first mainstream games supporting Vulkan was Dota 2[4] developed by Valve, the founding company behind LunarG. LunarG is a company that specializes in developing Vulkan SDK and increasing Vulkan support[5]. Support has also quickly been added to game engines such as Unity, Unreal or Godot, allowing its power to be presented to bigger and bigger audiences.

Khronos Group, the industry consortium responsible for Vulkan API, has been continuously improving the API and releasing updates. The API is currently on version 1.2[6], which brough important updates that have been requested by the community. This proves that Vulkan aims

to improve alongside the industry and provide support and improvements into the forseeable future.

1.2 Vulkan API architecture

Vulkan is designed to be very explicit about communicating intentions and possibly expensive operations between the implementation and application. The entry point into Vulkan is the instance object, which is created by calling the `vkCreateInstace` function. This function has to be dynamically loaded, since Vulkan may be linked dynamically instead of statically. The instance object serves as the parent of all other Vulkan objects in given context and its lifetime should encapsulate the whole application.

1.2.1 Execution model

Execution model of Vulkan specifies how to initialize, prepare and execute actions on Vulkan-capable hardware. Given an instance object the application can enumerate physical devices connected to the system. These Vulkan objects represent the hardware objects supported by the local Vulkan implementation instance. Typically, they represent the GPUs (both integrated and discrete) connected to the system.

The application can create a logical device object for each physical device. This is one of the first signs on how multi-gpu paralellism works in Vulkan. Objects created from a specific Vulkan device are device-private, but Vulkan specifies a way to export objects from one device to another. Each logical device exposes so-called queues, which process work independently of each other. This represents the single-gpu parallelism in Vulkan.

Queues in Vulkan are partitioned into queue families. Each family contains queues which are compatible with each other and can seamlessly execute identical workloads. Queues not belonging to one family may not be able to execute identical workloads, the capabilities of queue families can be queried from the device.

Device memory is allocated using logical device as a parent. Device memory is always visible to the device and can be either physically located in the device memory or in the host memory. The memory can also additionaly be visible and mappable to the host memory. Devices advertise supported memory types as heaps with its types exposed as bit flags. The device can advertise many heaps, but some devices, notably integrated ones, often advertise a single multi-purpose heap for all device allocations.

Once the application has initialized the instance object, allocated memory and prepared workloads into command buffers, the work is submitted onto queues requested along with creation of logical devices. When the work is submitted, control returns to the host application and and work is asynchronously executed on the device until completion. There is no implicit way to check workload completion nor are there guarantees between submission order and task completion. Even within a specific device queue, some work may interleave and execute out of order (within some coherency constraints).

To synchronize between the host and device, between two devices or even within the device itself the synchronization primitives have to be used explicitly. Fences can be used to synchronize between the host and the device while semaphores can be used to synchronize between device operations. All of this is the responsibility of the application, but can result in great performance if used correctly.

1.2.2 Object model

Entities in Vulkan are represented as opaque objects and are handled through handles. Handles are either dipatchable (e.g. pointers) or non-dispatchable (e.g. integers). Dispatchable handles are guaranteed to be unique while non-dispatchable handles are fully opaque up to the value of the handle. The only guarantee is the binary interface (the size) of the handle.

There are parent-child relationships between certain objects and this structure forms a partial ordering on both their initialization and their destruction. Some objects are destroyed implicitly (when their parent object is destroyed) and some objects have to be destroyed explicitily. There are exceptions where the child object does not need to be destroyed before its parent but must not be used after its parent is destroyed. Vulkayes observes these relationship using reference counting as described in sec. 3.2.

1.2.3 Application structure

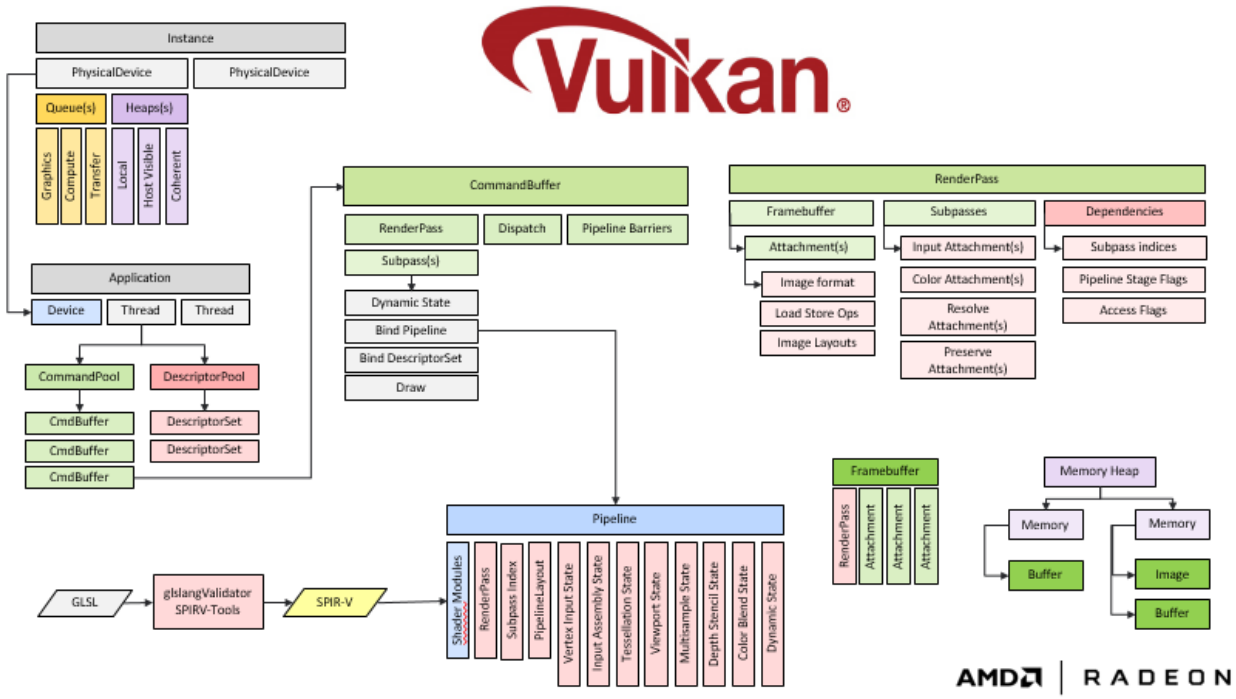

Figure 1.1: Overwiew of Vulkan API objects and basic data flow.[7]

The high level structure of Vulkan is that the user application creates an instance and chooses one or more physical devices. Queues on these devices can be split into graphics, compute, transfer and sparse categories. Some queues may support multiple properties. The application will create the queues as needed together with the device.

After device creation, the application is expected to describe as much state as it can beforehand. The application needs to create the render pass and within it describe all attachments, subpasses and dependencies of those subpasses. This early definition allows the implementation to transform this description into internal performance-oriented representation that is specific to the device. Similar process happens with the descriptor set layouts and pipeline layouts, where the application describes the requested features and settings of the descriptor/pipeline and then can allocate these objects based on those descriptions.

Assuming that the application is going to render to the screen a surface needs to be created along with a swapchain. Creating surface is platform-dependent process since it requires a

specific extension for the given platform and a platform-specific window handle. After the surface is created, the application creates a swapchain which takes care of presenting images onto the surface. The swapchain is platform-agnostic (from the application perspective) but is also implemented as an extension since not all platform necessairly need or support display surfaces or swapchains.

Images and buffers are another requirement in the process. The memory for both images and buffers is allocated and bound separately. This allows the application to use custom allocators and/or (with specific extensions) to create sparsely-bound images. Images additionaly specify a layout of their memory. This layout type can either be linear, and thus freely accesible from host, or optimal, and thus its structure is unspecified. Since most of the time images are not accessed from the host and are instead uploaded using staging buffers, most images are recommended to use the optimal layout type for performance. Not many image formats support the linear type.

The optimal layout type specifies multiple layouts as an enumeration. This enumeration always contains the `GENERAL` layout, which can be used in any context but may be least performant, and other additional layouts that may be used in specific contexts to potentially improve the performance. For example, there is a specific layout for transfer operations which is optimal for all copy and blit operations but cannot be used in attachment context. Additional complexity is added by the fact that the image can have multiple layouts at once in form of subranges. If an image is mipmapped or is an array image, each array layer and each mipmap level can potentially have different layout. The application is required to keep track of the current image layout because it is required to specify the layout, *at the time of execution*, in most of the commands that work with images.

To use images and buffers as attachments, the application needs to create views. A view object is a view into a specific subrange (mipmap levels and array layers) of an image or a specific range (offset and size) of a buffer. Views contain additional metadata that is used within the operation they are passed into. This metadata includes the subrange size, component mapping (e.g. mapping RGB to BGR) and, if an extension is enabled, different (but compatible) format for accesing the image data.

At some point, the application also needs to create memory pools. There are two types of memory pools: the descriptor pool and the command pool. These pools both serve the same general purpose, but they have different usage requirements. The importance of memory pools is that some allocations, namely the allocation of descriptor sets and command buffers, happen very frequently. System memory allocation can have considerable overhead and should be done infrequently. Pools solve this issue by allocating system memory separately from the resource memory. Allocating and freeing from the same pool will produce much less overhead than allocating and freeing from the system. Descriptor pools additionaly require a list of descriptor set layouts for which they can allocate, while command pools grow not only with initial command buffer allocation but also with each recorded command.

Shader modules are developed and compiled into SPIRV independently of the Vulkan API. The application should retrieve its shader code and pass it to Vulkan shader module creation function to create shader module object. Shader modules can later be bound to pipelines.

The next step is actualization of previously described resources. Framebuffers represent an actualization of the render pass description attachment list. Framebuffers are used in render pass commands to bind already prepared render passes with current attachments, since attachment may and often do change frame-by-frame. Descriptor sets represent an actualization of the shader interface which is “described” by the shader modules. Pipelines represent

an actualization of pipeline layouts. Most notably, pipelines describe the connection between shader modules and descriptor sets.

To render a singular frame not much beyond the device `waitIdle` method is need. However, since most applications will want to render continuously, synchronization is needed not only between frames but also between acquiring and using presentable images from the swapchain. Fences and semaphores need to be created by the application.

After preparing all the necessary resources, descriptions and actualization, the work to be done needs to be recorded into a command buffer. Command buffer is allocated from the command pool and then a few types of commands are recorded into it. Some commands may perform work of more than one type.

First type is state setting type. These commands alter the current state of the command buffer at the time of the command execution. These commands are generally not reordered during execution because they create an implicit happens-before relationship with commands after it that use that same state. These commands will bind the actualizations to the current context, such as the current framebuffers, pipeline and descriptors.

Second type is action type. These commands peform actions on resources, perform reads and writes and produce observable results. In general, these commands are most likely to be reordered and executed in parallel to improve performance. These commands include clearing, copying, blitting, drawing and computing.

Last type is synchronization type. These commands explicitly define relationships and constrains of the reodering of other commands. For example, to avoid race conditions between writing to an image and subsequently reading from it a pipeline barrier must be inserted using a synchronization command.

Finally, after a command buffer is recorded, it may be submitted for execution on a selected device queue. This queue must support the command types used within the command buffer. All resources used by the command buffer must be kept alive for at least as long as the execution lasts and all images must have the correct layouts at the time of execution of the specific command inside the command buffer that layout is defined in. All this has to be done manually by the application.

1.2.4 Complexity

As can be seen from the previous section, complexity of Vulkan applications is much greater than of its predecessor OpenGL. All this complexity gives opportunity to the application to fully express exactly when it does and doesn't need and thus be as performant as possible. However, adhering to the strict rules imposed by the API can be challenging for programmers and can be improved upon by using modern programming technologies and concepts.

Most notably, keeping track of object lifetimes can become tedious and error prone, which is why many higher-level abstractions over Vulkan tend to focus on it. Vulkayes uses reference counting and children always link to their parent so that the parent can never be destroyed before its children. Another common source of errors in applications is synchronization. Synchronization is a very complex topic even in non-Vulkan programming and can be hard to grasp and properly implement, much more so when the requirements are as dynamic as in Vulkan. This is why many wrapppers tend to offer some kind of synchronization solution.

2 Related work

There are already many libraries aiming to provide similar abstraction over Vulkan. Some of the most prominent and closest to this work are mentioned below.

2.1 V-EZ

"V-EZ is an open source, cross-platform (Windows and Linux) wrapper intended to alleviate the inherent complexity and application responsibility of using the Vulkan API. V-EZ attempts to bridge the gap between traditional graphics APIs and Vulkan by providing similar semantics to Vulkan while lowering the barrier to entry and providing an easier to use API."[7]

This ease of use does come at a price, however. The design of V-EZ leaves no room for the user to properly express their intent at critical points of execution. This leads to unnecessary slowdows and hashmap lookups which outweight most of the benefits gained by simplified API.

Last commit to V-EZ was on 2018-10-05[7].

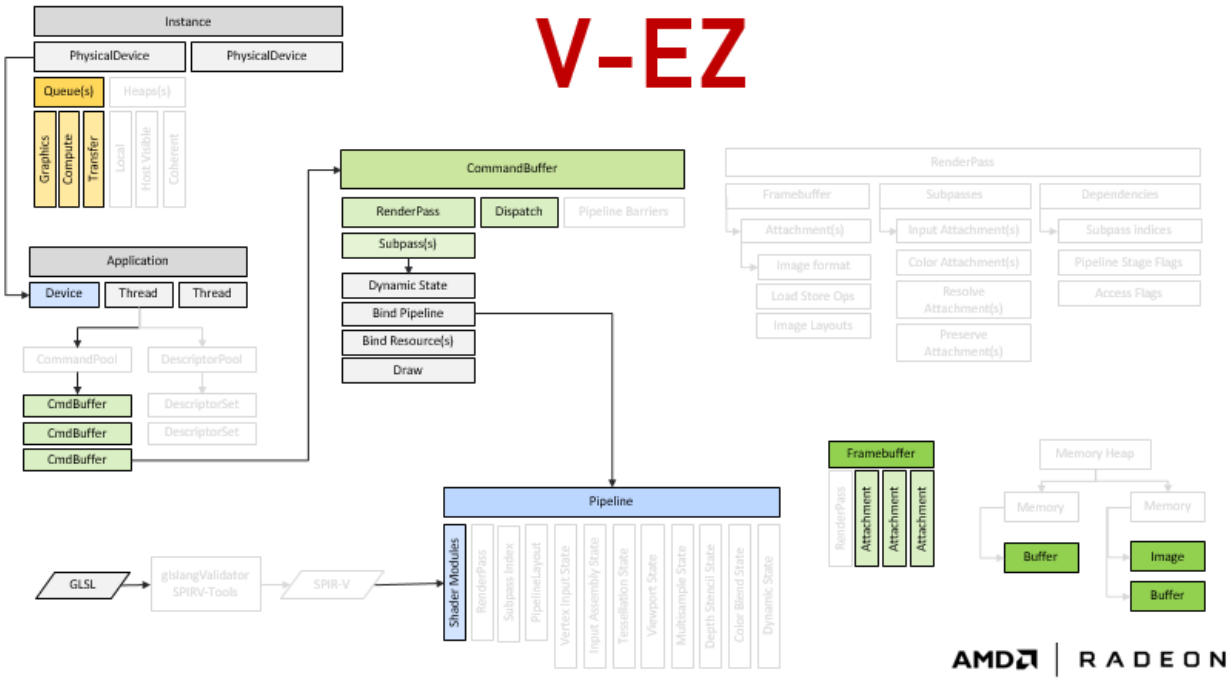

Figure 2.1: V-EZ greatly reduces the number of objects the user has to care about. Everything else is taken care of behind the scenes.[7]

2.2 gfx-hal

gfx-hal or graphics hardware abstraction layer[8] is a project aimed at abstracting graphics computations not only from hardware, but also from low-level APIs like Vulkan or OpenGL. It is, in a sense, lower level than Vulkayes aims to be. The abstraction over multiple APIs,

while very useful for most common usages, can hurt usability in niche cases where a specific extension or feature is only available in one API.

In contrast, Vulkayes aims to provide a *transparent* abstraction over Vulkan API. This allows users to use any features available to them by the API even if the abstraction doesn't implement it directly.

Figure 2.2: gfx-hal creates an abstraction layer over all mainstream graphics APIs.[8]

2.3 Vulkano

Vulkano[2] aims to provide complete validation and synchronization guarantees for the user. This proved to be too limiting and the original developer eventually left the project. Since then, not much work has been done.

Vulkayes originally started as a fork of Vulkano, however, over time, it grew into a rewrite because of many questionale design choices taken in Vulkano. Vulkano makes havy use of dynamic dispatch, which impacts performance. Its API also promises thorough validation checks, however at the expense of API flexibility, which makes it less likely to be widely adopted. For example, it is still impossible to upload mipmaps to Vulkanos `ImmutableImage` (which is intended as one-time write image abstraction, e.g. for textures in games).

Figure 2.3: Vulkano logo.[2]

2.4 Tephra

Tephra[9] is a very recent work with very similar aims to Vulkayes. It can the though of as a C++ version of Vulkayes. It takes a fresh look at the existing solutions and comes up with a transparent and flexible API for handling Vulkan.

However, many of the design considerations taken in Tephra revolve around safety and sanity of C++ language itself. This is of questionable importance and puts unnecessary strain on the

library designer. Overall, most of the well designed concepts in Tephra have to be weighted againts the unfriendliness of the language.

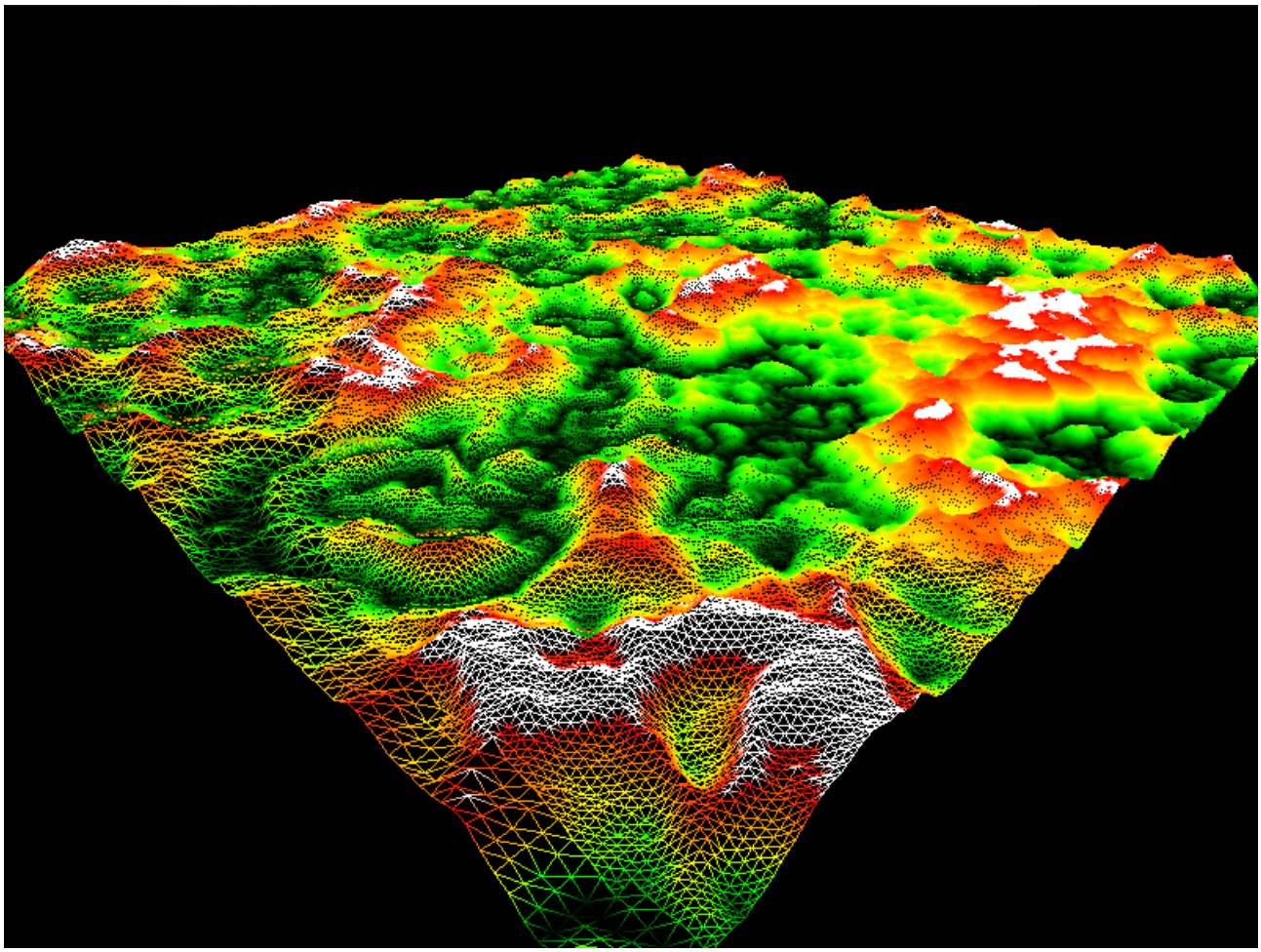

Figure 2.4: Screenshot of one of the benchmarks for Tephra.[9]

2.5 Summary

Table 2.1: *Related work summary*

Library	Status	Language	Goals
V-EZ	Abandoned	C++	Usability over performance
gfx-hal	Active	Rust	Hardware abstractions
Vulkano	Abandoned	Rust	Safety over performance
Tephra	Unknown	C++	Performance and usability
Vulkayes	Active	Rust	Performance, usability and increased safety

In summary, many projects aiming at simplifying the Vulkan API have been either abandoned or are too broad in scope to consider them production-ready. In the end, Tephra comes out as the closest and most practically usable work. However, the C++ language is itself a complex and hard to master system that places many requirements on the user of the library.

In contrast, Rust, and by extension Vulkayes, aims to offload as much off the user as possible without unnecessary and performance-reducing restrictions.

3 Design

The API was designed to fullfil three goals:

1. Be transparent - The API must allow falling back to pure Vulkan if a certain feature is not supported or implemented in the API.
2. Be fast - The API must carefully manage abstraction costs and minimize overhead.
3. Be flexible - The API must be easy to use in different contexts. It must not force the user to unreasonably change their code to fit the API.

3.1 Rust

Where performance is critical, programmers often fall back to the "classical" languages such as C and C++. These languages, however, are often burdened by legacy, backwards compatiblity and outdated design concepts.

C is a very simple and fast language. However, programming industry has changed quite a lot since its first appearance 48 years ago[10]. Concepts common at the time in programming, such as easy low-level memory access and easy mapping to machine instruction, are hardly transferrable to todays high-level requirements of programming.

C++ attempted to extend C with a useful standard library of data types, algorithms and other features. This made C++ a much better candidate at creating complex performance-critical applications. However, stemming from C, it still caries the burden of past decisions. Writing sound code often requires the programmer to be *more* expressive and pay more attention to intricacies of the language. This comes at an expense in code quality, readability and sometimes programmer sanity.

The Rust programming language became a natural choice for this project because goals 2. and 3. are already core concepts of the language itself. Unlike C, it has extensive standard library and was designed for high-level programming. Unlike C++, higher code safety requires *less* work from the programmer. That is, safety is enforced by the language features in form of statical analysis.

3.1.1 Ownership

Rust implements a very simple but powerful ownership model. You cannot prevent the compiler from moving your value. However, the language is smart about this. Moving a value does not just create a bitwise copy, it also moves the ownership which has serious consequences: the owner has to clean up. Values that have non-trivial destructors should run those destructors at some point. Indeed, the memory move semantics in Rust are simply an implementation detail of its higher-level abstraction of movement of ownership.

In C++ the only difference between a copy and a move is that the new value has a chance to take apart the old value. For example, for heap allocated types, this means the new value will take the heap memory (pointer) from the old value. The destructor, however, is still run for both the values, as it if was simply copied. Ownership in C++ is only conceptual, the language itself does not understand it nor enforce it. This moves the burden of reasoning on the the user.

In contrast, Rust statically prevents use of moved-out variables. Once you move a value out of a variable (moving the ownership somewhere else), that variable now acts as if it was uninitialized, it cannot be used anymore and its destructor is not called. The destructor is only called for the "new" value once it goes out of scope, it is the responsibility of the new owner to clean up. Moreover, this move is often optimizable by the compiler and thus is almost or entirely free.

Borrow checker The Rust borrow checker tracks borrowed values. A value is borrowed when a reference to it is created. A reference can either be immutable or mutable. There can only ever be one mutable reference and it cannot coexist with any immutable references. This completely prevents all read-write race conditions *statically*.

Borrow checking also prevents problems such as use-after-free or iterator invalidation. These problems can be considered single-thread race conditions. A reference is created, then the original referred value is destroyed or moved and then the reference is used (to read or write). Such a reference is called dangling. Rust statically prevents the existence of dangling references. When a value is borrowed, it must outlive any references taken from it. That is, an owner can lend the value to someone, but it must then keep the value in its place for as long as the borrow is valid, it cannot be moved to a new location nor dropped. This is done using lifetimes.

Lifetimes Lifetimes are how Rust tracks borrows. Each borrow (a reference) has a lifetime associated with it. The borrow cannot be used for longer than that. For example, if a value is created in a certain scope then a reference to it cannot escape that scope since it could lead to use-after-free. Additionally, programmers can use these lifetimes too, as generic arguments, to express concepts like borrowing subfields or narrowing array views.

There is one lifetime that is always available, the `'static` lifetime. This lifetime is special in that it expresses the concept of *always valid*. References with this lifetime can be used anywhere in the program, at any time, because they are known to always outlive the program itself. For example, taking a reference to static data (data compiled into the binary executable) creates a static reference that can be then freely used inside the application.

3.1.2 Safety and speed

Of course, some of the lowest-level code cannot be created in this somewhat restricted environment. The abstraction has to be built somehow. This is where **unsafe** Rust comes in. Instead of specifying additional safety features, Rust programmers have to explicitly ask to disable existing features. Code blocks marked `unsafe` are free to work with dangling pointers, have data races or cause other unsoundness, just like C++ normally does.

The implementation of the Rust standard library has empirically proven that the system truly only needs unsafe blocks few and far between. Indeed, only the most basic building blocks have to rely on unsafe operations, while all the other parts can just rely on the soundness of these simple code snippets that can easily be checked and verified over and over by quanta of programmers to ensure they truly are sound. This safety system reduces possible failures to a few narrow blocks of code, instead of leaving the programmer with having to find the bug in all of their code.

All of this is done at compile time and thus has no runtime cost. All code is as fast as the same C++ code would be, but safe.

3.1.3 Cargo

Cargo[11] is Rusts package manager. It takes care of indexing and retrieving dependencies, compiling them and publishing libraries and binaries to the registry. Cargo also takes care of project configuration. In C/C++ codebases it is common to either invoke the compiler directly, or to use build tools such as `make` or `CMake`. Cargo is similar to those build tools, but it is a component of Rust ecosystem and is targeted at Rust only.

Being a part of Rust itself, cargo is able to provide lots of useful abstraction over the rust compiler. The configuration file `Cargo.toml` is filled with useful project information such as the project name, author and short description. The file also contains technical information, like the targeted language edition, compiler and optimization flags, all of the dependencies (and how/where to look for them) and project features. Platform-specific configuration is also possible.

Features defined in `Cargo.toml` are project-unique strings. These strings can then be used from within the codebase to conditionally compile part of the code, similar to C preprocessor `#ifdef` statements. Contrary to the C preprocessor, however, these strings are defined in one central place and can even define dependency chains, so that certain features might require other features or additional dependencies. This is often used when developing on top of platform-dependent code to provide uniform interface to the user. It is also used in Vulkayes, as mentioned in sec. 4.2.

3.1.4 Generics

Generics are a very powerful tool in programming. They help avoid a common problem in libraries: “What if my object doesn’t cover all usecases”. Generics provide a way for the library user to specify their own object with their own implementation and it only has to conform to some predefined bounds. In Rust, this is done by specifying trait bounds:

```
trait BoundTrait {
    fn required_method(&self) -> u32;
}

fn generic_function<P: BoundTrait>(generic_parameter: P) -> u32 {
    generic_parameter.required_method()
}
```

In this code snippet, the `P` parameter of the `generic_function` is generic. The user can then do this:

```
struct Foo;
impl BoundTrait for Foo {
    fn required_method(&self) -> u32 {
        0
    }
}

struct Bar(u32);
impl BoundTrait for Foo {
    fn required_method(&self) -> u32 {
        self.0
```

```
    }
}
```

Now both the `Foo` struct and the `Bar` implement the trait `BoundTrait` and can be used to call `generic_function`:

```
let foo = Foo;
generic_function(foo);

let bar = Bar(1);
generic_function(bar);
```

This usage is zero-cost because the functions are monomophised at the compile time for each calling type.

Storing generic parameters Using generic parameters is one thing, but storing them is harder. Generic parameters can have different sizes that are not known at the definition time:

```
struct Holder<B: BoundTrait> {
    item: B
}

let a = Holder { item: Foo };
let b = Holder { item: Bar(1) };
```

In this snippet, it is unknown at the definition time how big the `Holder` struct will be in memory. Instead, it is decided at the use time. That is, the variable `a` possibly takes less space on the stack than the variable `b`. The size of a type is a function of its fields, if the field is generic, it can't be known up front.

Generic parameters are a part of the type. Two `Holders` with different generic parameters cannot be stored together in an uniform collection (like `Vec`). The only way to achieve that is by using dynamic dispatch.

Dynamic generics Dynamically dispatched generics can be used to mix and match different implementations of traits in the same place. It works by taking a pointer to the generic parameter and then "forgetting" the type of that parameter, only remembering the bounds. In Rust, this is handled by trait objects in the form of `dyn BoundTrait`. This is an unsized (size isn't known at compile time) type and it cannot be stored directly on the stack or in uniform collections either. It needs to be behind some kind of pointer, whether it be a reference, `Box`, `Rc/Arc` or a raw pointer. This pointer will be a so-called "fat" pointer.

For example, to store any kind of `BoundTrait` implementor in a `Vec`, it can be written like this:

```
let a = Foo;
let b = Bar(1);

let vec: Vec<Box<dyn BoundTrait>> = vec![
    Box::new(a) as Box<dyn BoundTrait>,
    Box::new(b) as Box<dyn BoundTrait>
];
```

The downside of this is both the allocation of heap memory and the access speed. Accessing

methods on the object has to go through one more level of indirection than normally and also prevents certain powerful compiler optimizations. Thus is it undesirable to use dynamic dispatch when it is not necessary.

3.2 Object lifetime management

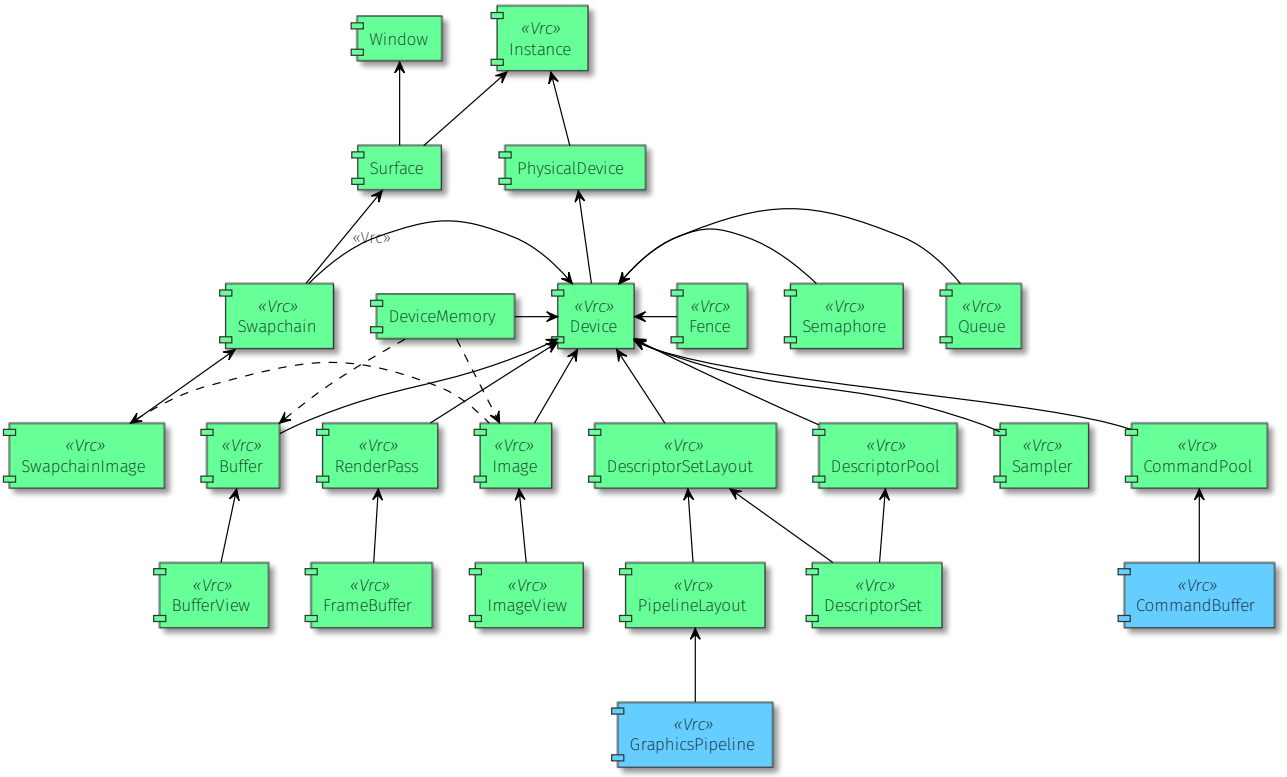

Figure 3.1: Object Dependency Graph of Vulkayes

Objects in Vulkan have certain lifetime dependencies - some objects must outlive others - displayed in fig. 3.1. Some dependencies are simpler and always apply, others are more complex and conditional. Most of these dependencies in Vulkayes are handled using reference counting. Reference counting is a programming concept where data is shared among multiple actors using some kind of reference (pointer). The pointed-to memory, apart from storing the data object itself, also stores a count of existing references to that memory. This provides an easy way to clean up resources when they are no longer used, all automatically at runtime, with overhead only during the creation and destruction of the resource itself, not during usage. The Rust safety system also prevents the pointed-to memory to be freed or otherwise deinitialized, ensuring safety.

Rust also differentiates between normal reference counted value and atomically counted reference counted value. The former is called `Rc` while the latter is called `Arc`. This is used in Vulkayes, as described in more detail in the sec. 4.4.2.

3.3 Memory management

There are two types of memory in Vulkan. Host memory - the memory accesible only to the CPU, and device memory - the memory accesible to the device. Device memory might be

host-mappable, meaning it can be accessed by the CPU if it is explicitely mapped, similar to the C `mmap` function.

Host memory in Vulkan is managed by the implementation, but Vulkan exposes a way to intercept this process by allowing the application to provide its own allocation callbacks. These callbacks are called whenever the Vulkan implementation wishes to allocate, reallocate or free memory and can be used to handle allocation in a custom manner. Vulkan specification recommends using these callbacks only for debugging purposes or in specific cases, not in general, as they would not impact the performance in any meaningful way.

Device memory, however, is a bigger topic in Vulkan. Applications are expected to allocate and manage memory themselves. Vulkan only recommends that allocation should happen in 128 to 256 MB chunks at a time to reduce the overhead. This means Vulkayes needs to provide its own way to integrate user-defined device memory allocation, as described in more detail in the sec. 4.3.1.

3.4 Synchronization and validations

Vulkan leaves almost all CPU synchronization to the user. Explicit synchronization requirements are described in the specification and Vulkan objects are not reentrant. The user application has to take care of all the synchronization requirements as to not cause a data race. Vulkayes solves this in two ways. When used normally, no synchronization is done and everything is as performant as it can be. Secondly, it provides a multi-thread feature (sec. 4.2) where mutexes are used and proper synchronization is ensured.

Validations in Vulkan are generalization of synchronization requirements. Validations specify not only how to prevent data races, but also how to prevent other undefined behaviors. Vulkan validation requirements tend to be very long, dense and hard to parse, leading to an increased chance of breaking them. Vulkayes aims to alleviate this somewhat by guaranteeing at least the most common and statically solvable validations to be fulfilled.

Last topic of synchronization is GPU synchronization. This encompasses synchronization of resource usage in command buffers executed on the GPU queues as well as the synchronization between CPU and GPU. This kind of synchronization is very important, but it is a complex topic on its own and is left to be added to Vulkayes as a separate project.

4 Implementation

4.1 Bindings

Vulkan API is an interface specified in the C programming language. C language is the de-facto standard in cross-language APIs. This means the system of bindings is available in almost any practical language, including Rust. Vulkayes relies on the ash[12] crate to provide these binding and some syntactic sugar on top. This library uses the Vulkan API Registry, canonical machine-readable definition of the API, to generate bindings from Rust to C automatically.

4.2 Cargo features

An important part of any flexible project is to give the user as much control as possible, so the library will fit their usecase. One way to achieve this in Rust are cargo features already mentioned in sec. 3.1.3.

The most important features defined and exposed in Vulkayes are described below.

naive_device_allocator This feature conditionally compiles a very naive device memory allocator into the project. Device memory allocation is a complex topic and applications are required to provide their own allocators to fit their own needs. One popular allocator is the Vulkan Memory Allocator[13], but it is a big dependency that might not be easily accessible for certain usecases. Vulkayes supports integration with VMA (and other allocators) seamlessly, but also provides the naive allocator as a simple no-dependecy alterative for quick prototyping and debugging.

multi_thread One of the biggest selling points of Vulkan are its multi-threading capabilities. Since the user is in charge of synchronizing the resources, they can design their application to fit their needs. Single-threaded applications require no synchronization, while multi-threaded applications should allow for the full power of multi-threading to be leveraged.

Safe Rust statically prevents data races using the built-in `Send` and `Sync` traits. These traits are automatically implemented (or not implemented) by the compiler to mark types as "capable of being sent between threads" and "capable of being borrowed across threads" respectively. The user is free to unsafely implement these traits back if the compiler decides to not implement them, provided that the user takes the burden of preventing data races upon themselves.

By default, object wrappers in Vulkayes are not `Send` nor `Sync`, simply because they use the `Rc` type, which is a shared pointer wrapper type that uses non-atomic loads and writes to count. By turning this feature on, all usages of `Rc` across the crate are switched to `Arc`, which is atomically counted and thus implementes `Send` and `Sync` safely.

Additionaly, single-threaded Vulkayes replaces use of mutexes with simple wrapper types that emulate the mutex API, but do not implement `Send/Sync` and do not do any synchronization. This makes the API of both single-threaded and multi-threaded Vulkayes uniform. The main reason for this feature is performance, since atomic operations and synchronization is costly compared to non-atomic counterparts.

runtime_implicit_validations Vulkayes aims to increase safety of Vulkan calls as much as possible without any performance impact. The idea is to always guarantee that the implicit

validations defined in Vulkan spec are fulfilled and the explicit validations are only fulfilled when they can be easily derived from the existing API design.

This proved to not be always possible, so a small portion (tbl. 5.5) of implicit validations requires some runtime checking to ensure their fulfillment. These validation, producing runtime overhead, are conditionaly compiled using this feature to ensure that the user can always opt-out to achieve greater performance.

4.3 Generics

Generics are used in key places across Vulkayes. One example are device memory allocators, another would be image views. They are described in detail below.

4.3.1 Device memory allocator generics

Device memory allocators have one of the biggest impact on performance of Vulkan. There is no default memory allocator in Vulkan. Instead, memory has to be allocated manually from the device. That operation, however, can be slow. That is why it is recommended by the Vulkan specification to allocate memory in bigger chunks at once and then distribute and reuse the memory as best as possible in the user code.

For Vulkayes, this means it is required to support user-defined allocators. This is the perfect usecase for generics. An image, which needs some kind of memory backing to operate, has a simplified constructor like this:

```
trait DeviceMemoryAllocation {
    // Allocation trait methods
}

trait DeviceMemoryAllocator {
    type Allocation: DeviceMemoryAllocation;

    fn allocate(&self) -> Self::Allocation;
}

struct Image {
    // Image fields
    memory: ??
}
impl Image {
    pub fn new<A: DeviceMemoryAllocator>(
        // Other fields
        memory_allocator: &A
    ) -> Self {
        // Initialization code
    }
}
```

The `memory_allocator` parameter can be any user-defined type that implement the `DeviceMemoryAllocator` trait (and thus is capable of distributing memory given some requirements). However, given the requirements of Vulkan specification, we need to

ensure that the memory outlives all usages of the image. This implies we need to store some kind of handle to the allocated memory, which can be any type implementing `DeviceMemoryAllocation` (as can be seen in the `DeviceMemoryAllocator` traits associated type `Allocation`).

Storing this memory thus has the same implications as mentioned above. We could make the `Image` struct generic over the memory it stores. This would however mean that the memory generic parameter would have to be present on anything that can possibly store the image, including swapchain images, image views, command buffers and so on. This could prevent us in the future from creating a command buffer and recording into it operations on images with possibly different memory allocations (for example, because one is a sparse image and the other is fully-backed).

Since this is very limiting, the memory inside an image can be stored using dynamic generics. So the `??` in the above code snippet would be replaced with `Box<dyn DeviceMemoryAllocation>`.

This would be ideal for images, where the memory does not need to be accessed until it is to be deallocated (barring linearly tiled images). For buffers, however, this is a common use case. Buffers are often used as staging or uniform. Data is uploaded into a buffer from the host and then copied using device operations into an image backed by fast device-local memory. The upload of data is done by mapping the memory into host memory using Vulkan provided mechanism and then writing to it as if it was normal host memory.

4.3.2 Mappable memory generics

Some use cases for mapped memory are performance-critical. For example, vertex animating data is done by continuously changing vertex buffer data according to the animation properties. This means the mapped memory has to be accessed every frame. This is where dynamic dispatch cost would be substantial, it is best to avoid it.

One of the ways to avoid this cost is to simply push it back. There are only 3 places where the generics are truly needed:

- The memory map function
- The memory unmap function
- The cleanup function

No other place of the memory handling needs custom user coding. This means it is enough to store 3 generic user-provided functions. In Rust, this can be done using the `Fn` family of traits. For example, instead of `Box<dyn DeviceMemoryAllocation>` for the cleanup function we will use `Box<dyn FnOnce(&Vrc<Device>, vk::DeviceMemory, vk::DeviceSize, NonZeroU64)>` inside a concrete `struct DeviceMemoryAllocation`. The cleanup function can be simply `FnOnce`, which can only ever be called once, while the map and unmap functions might need to be called multiple times and have to be `FnMut`.

The performance of this solution is measured in more detail in sec. 4.3.1.

4.3.3 Image view generics

Image views are another object in Vulkano that has to deal with generics. Image view can wrap any type that can “act like” an image and create a view into some kind of subrange. This can be expressed using the `ImageTrait` like so:

```
struct ImageView {
    // Image view fields
    image: ??
}
impl ImageView {
    pub fn new<I: ImageTrait>(
        image: I
    ) -> Self {
        // Initialization code
    }
}
```

As mentioned above, this is very limiting because of the generic parameter. Unlike the above case, however, the image field needs to be accessed considerably more often.

The following table shows a benchmark of so-called mixed dispatch, where an `enum` is used to provide common possible values for a given generic type and the last variant, which is the only one truly generic, is provided as a `Box<dyn Trait>` to allow using dynamic dispatch where the set of provided types is not extensive enough.

Table 4.1: Benchmark of so-called mixed dispatch enums, where enum variants house common types and the last variant houses a boxed dynamically dispatched one to cover other usecases.

benchmark	avg. black box	avg. no black box
Enum::Foo	499.01 ps	251.31 ps
Enum::Bar	499.47 ps	252.67 ps
Enum::Dyn	1.3018 ns	1.2512 ns
Foo	499.36 ps	260.76 ps
Bar	499.03 ps	252.18 ps
Qux	313.34 ps	250.41 ps
dyn Qux	1.5104 ns	1.5028 ns

As can be seen from the table, accessing a value through a dynamic dispatch is at least twice as slow as accessing it through static dispatch, and this is with optimizations prevented by using the concept of a black box from the Rust stdlib.

Non-black boxed benchmarks show that the optimizations provided by the compiler for statically dispatched values can further reduce the overhead of static dispatch, while the dynamic dispatch stays mostly the same.

This cones as an alternative to normal generic to avoid generic parameter plague and was chosen as an acceptable way to treat image type dispatch in Vulkayes.

4.4 Abstraction

4.4.1 Reference counting

Reference counting is used for two purposes. First, the most obvious one, is shared usage. Most of the objects in Vulkan stem from the Device object and operations on these objects

require access to the device and device pointers. Second is lifetime dependecy. Objects in Vulkan have a defined partial ordering on their destruction order, that is, the device must pretty much outlive all its children. This is achieved as a consequence of shared pointer usage, since all children of the device keep a link to the device, the last child to be dropped drops the device (unless the user is holding the device pointer as well, in which case it is still alive).

4.4.2 Type aliases

Vulkayes makes use of project-wide type aliases to make transitioning some of the cargo features seamless.

`Vrc` One of the most important type aliases which resolves to either `type Vrc<T> = Arc<T>` or `type Vrc<T> = Rc<T>` depending on whether the multi-threaded feature is enabled or not. This type alias is used practically everywhere, since most types (as seen in fig. 3.1 and discussed in sec. 4.4.1) are wrapped in reference counted pointer types.

4.4.3 Deref

This trait comes from the Rust standard library and is a specially known trait to the compiler. It is intended to be implemented for smart pointer types as a way to uniquely claim that a type Bar is really just a wrapper around type Foo. This fits nicely with the notion of smart wrappers in Vulkayes. For example the `Image` object:

```
pub struct Image {
    image: vk::Image,
    // fields omitted
}
impl Deref for Image {
    type Target = vk::Image;

    fn deref(&self) -> &Self::Target {
        &self.image
    }
}
```

This way, the `Image` object claims that under the hood it is simply the `vk::Image` handle but with some added information and utility on top. The `Deref` trait itself defines an associated type `Target` and a `deref` method. These two things together provide complete information about what type the `Image` object derefs to and an ability to borrow it as that type.

This is used heavily across Vulkayes for all smart wrappers around Vulkan handles. Additionally, Vulkayes makes heavy use of Rust macros-by-example system to implement most important traits (such as `Eq`, `Hash` and `Ord`) on each of the smart wrapper objects. The `impl_common_handle_traits` macro saves over 500 lines of repetitive code across the Vulkayes crate.

4.5 Swapchain recreate

Swapchain is an object in Vulkan that facilitates image presentation onto surfaces. Surfaces are an abstraction over regions of the physical display, intended mainly for windowing systems and compositors. A swapchain is created for a combination of a surface and a device.

Requirement for our Swapchain object are:

1. Only one swapchain can exist for one surface.
2. Allow user to retrieve the surface when the swapchain is no longer in use.
3. Allow user to recreate the swapchain, transferring the ownership of the surface to the new instance, retiring the old swapchain.
4. Keep retired swapchain alive until all its acquired images are not longer in use.

Satisfying all the conditions as they are is not trivial, mainly because the the first two conditions lead to the requirement of dropping the swapchain once the surface is moved out of it, however, the fourth condition requires us to keep it alive. This can also create problems where for some reason the retired swapchain outlives the active one. In such cases, the surface can happen to be dropped before the retired swapchain, which is incorrect.

To satisfy all 4 conditions, we first have to rewrite them into terms that can be expressed in the language.

1. The creation of a swapchain requires full ownership of the surface, thus our constructor has to take surface by value.
2. The swapchain has to have a method that consumes the swapchain and returns the surface by value.
3. The new, recreated swapchain has to take the old swapchain by value and extract the surface from it using method from 2.
4. The swapchain has to be reference counted to outlive all its images.

Now it is much clearer why these requirements are hard to satisfy - 4. requires that the swapchain reference counted and its lifetime is guarded dynamically, however, 2. and 3. require for the lifetime of the swapchain to end immediatelly rather than sometime in the future. We need to rewrite the requirements to work with reference counting.

Adapting 2. is implementationally trivial. We must rely on the user to first drop all outstanding shared pointers except for one and then use that one to retrieve the swapchain back as an owned value.

Adapting 3. however, is much harder to implement as we can't expect the user to wait until all outstanding operations on the current swapchain are done until creating a new one since that would limit the functionality too much. Instead, we need to make sure that the surface is alive for the longer of the two lifetimes. This is exactly what reference counting does. By reference counting the surface inside a swapchain but still requiring an owned value for swapchain creation, we can make sure that no two active swapchains are ever created for one surface while still leaving the possibility of retrieving the surface after all but one of the shared pointers are dropped.

The resulting API thus looks like this:

```
pub struct Swapchain {
    surface: Vrc<Surface>,
    // fields omitted
}
impl Swapchain {
    pub fn new(
        surface: Surface,
        // parameters omitted
    ) -> Vrc<Self> {
        Vrc::new(
```

```
            Swapchain {
                surface: Vrc::new(surface),
                // fields omitted
            }
        )
    }

    pub fn recreate(
        self: &Vrc<Self>,
        // fields omitted
    ) -> Vrc<Self> {
        Vrc::new(
            Swapchain {
                surface: self.surface.clone(),
                // fields omitted
            }
        )
    }

    pub fn surface(&self) -> &Vrc<Self> {
        &self.surface
    }
}
```

This satisfies all the rules:

1. We cannot retrieve the surface back from the swapchain without destroying the shared pointer, which dynamically ensures there are no other instances.
2. The swapchain returns a reference to the reference counted surface, which can be destroyed to gain the surface after dropping all swapchains and swapchain images in the same way as above.
3. Both the new and the old swapchain contain a reference to the surface and thus will keep it alive for as long as is needed.
4. Swapchain is reference counted and can be kept alive by the images.

4.6 Windowing

Vulkan handles windowing by providing abstraction over native windows on different platforms using extensions. Each supported platform has a specific extension that can be used to construct a Vulkan handle to a surface, which is an object abstracting over the native surface. Some platforms, notably macOS and iOS, have additional requirements on the creation process of the window.

Vulkayes provides abstraction over this in a separate crate called `vulkayes-window`. This crate contains three tiers of code. First tier is raw Vulkan creation method for each platform. This code is platform specific and highly unsafe. Second tier are implementation specific creation methods, which abstract over platform differences using the windowing library implementation, but still require unsafe code for the ash objects.

The third and most important tier are the implementation specific Vulkayes creation methods. These methods are *safe* and provide full abstraction over the platform and ash, returning

Vulkayes wrapper types ready to be used safely. These methods are the main point of the `vulkayes-window` crate, but the other tiers are provided for flexibility and transparency reasons.

Currently supported implementations are:

- `winit` - The most popular fully-featured windowing library in Rust ecosystem. Provides almost full abstraction over platform windows.
- `minifb` - One of the simplest and easiest to use windowing libraries. Provides the minimal needed abstraction to quicky and easily create and draw on window surfaces.
- `raw_window_handle` - A library providing common types intended for all Rust windowing libraries. Both `winit` and `minifb` use this library and `vulkayes-window` supports all libraries that can be glued through this library.

5 Evaluation

5.1 User code

One of the main concerns when designing a library is the user code. How the user code will look like, if it will be readable and comfortable to write.

```
let (vertex_buffer, vertex_buffer_memory) = {
    let create_info = vk::BufferCreateInfo {
        size: std::mem::size_of_val(
            &data::VERTICES
        ) as vk::DeviceSize,
        usage: vk::BufferUsageFlags::VERTEX_BUFFER,
        sharing_mode: vk::SharingMode::EXCLUSIVE,
        ..Default::default()
    };
    let buffer = unsafe {
        device
            .create_buffer(&create_info, None)
            .expect("Could not create vertex buffer")
    };

    let memory_req = unsafe {
        device.get_buffer_memory_requirements(buffer)
    };
    let memory_index = memory::find_memory_type_index(
        &memory_req,
        &device_memory_properties,
        vk::MemoryPropertyFlags::HOST_VISIBLE
            | vk::MemoryPropertyFlags::HOST_COHERENT
    )
    .expect("Unable to find suitable memory type");

    let allocate_info = vk::MemoryAllocateInfo {
        allocation_size: memory_req.size,
        memory_type_index: memory_index,
        ..Default::default()
    };
    let memory = unsafe {
        device
            .allocate_memory(&allocate_info, None)
            .expect("Could not allocate memory")
    };
    unsafe {
        device
            .bind_buffer_memory(buffer, memory, 0)
            .expect("Could not bind memory");
    }

    (buffer, memory)
};
```

```
let vertex_buffer = {
    Buffer::new(
        device.clone(),
        std::num::NonZeroU64::new(
            std::mem::size_of_val(&data::VERTICES) as u64
        ).unwrap(),
        vk::BufferUsageFlags::VERTEX_BUFFER,
        SharingMode::from(queue.as_ref()),
        BufferAllocatorParams::Some {
            allocator: &device_memory_allocator,
            requirements: vk::MemoryPropertyFlags::HOST_VISIBLE
                | vk::MemoryPropertyFlags::HOST_COHERENT
        },
        Default::default()
    )
    .expect("Could not create vertex buffer")
};
```

Code sample 5.1: *An example of the code with same functionality from the original examples (left) and from the current ones (right). The code after is three times shorter than the original code while exposing the same functionality and providing static validation guarantees.*

Overall, the code for benchmarking the spinning teapots written in pure `ash` has 1400 lines of Rust code. The code with Vulkayes with same semantics and even improved static validation guarantess has 942 lines. This is a difference of 458 lines of code. These numbers are clear indicators of the improvement in developer experience by using correctly designed wrappers.

Another interesting code example is the uniform buffer usage:

```
unsafe {
    *uniform_buffer_memory_ptr = frame_state;
}
let flush_ranges = [
    vk::MappedMemoryRange::builder()
        .memory(uniform_buffer_memory)
        .size(
            std::mem::size_of::<data::UniformData>()
                as vk::DeviceSize
        )
    .build()
];
unsafe {
    device
        .flush_mapped_memory_ranges(&flush_ranges)
        .expect("Could not flush uniform data memory");
}
```

```
uniform_buffer
    .memory().unwrap()
    .map_memory_with(|mut mem| {
        mem.write_slice(&[frame_state], 0, Default::default());
        mem.flush().expect("Could not flush uniform data memory");
        MappingAccessResult::Continue
    })
    .unwrap();
```

Code sample 5.2: *The ash code (left) is twice as long and in some cases possibly even unsafe. Vulkayes API guarantees proper locking and borrowing, provides simplified way to flush the memory and prevents unaligned writes which on some platforms might cause hard errors and abort the process. The checking for correctness, however, does have some runtime cost. One of the guarantees of safe Rust is memory safety and Vulkayes is targeting safe Rust. That is why the write slice method call above does more than just write to a pointer. There is logic to check the align of the pointer and make sure all writes are either properly aligned, or an unaligned instruction is used.*

5.2 Benchmark

A benchmark of ash vs Vulkayes was designed to compare the speed of Vulkayes against a "control sample" of ash. This benchmark measures several stages of a common rendering loop between ash and Vulkayes. Since Vulkayes is mostly safety and usability wrapper, not much runtime overhead is added, at least not in the critical hot-paths used in rendering loops. Some specific areas, however, such as memory mapping and writing need special handling to ensure safety, as mentioned in sec. 5.1.

The benchmark results have three separate columns. `ash` is the control sample/baseline measurement. `vy_ST` is the single-threaded Vulkayes while `vy_MT` is the multi-threaded feature of Vulkayes.

5.2.1 Stages

The rendering loop was split into 8 stages:

preinit In this stage frame specific variables are calculated, such as the data dependent on the elapsed time. This stage represents the user logic that is not being benchmarked.

acquire In this stage the present image is acquired from Vulkan implementation. This stage is heavily dependent on the Vulkan implementation and is not being benchmarked.

uniform In this stage uniform data specific for the frame is written into device visible mapped memory and flushed. Some absolute overhead is expected because Vulkayes does checks to ensure memory safety.

command In this stage command buffer is recorded by binding necessary state and submitting draw commands for each teapot, along with push constants. Minimal overhead is

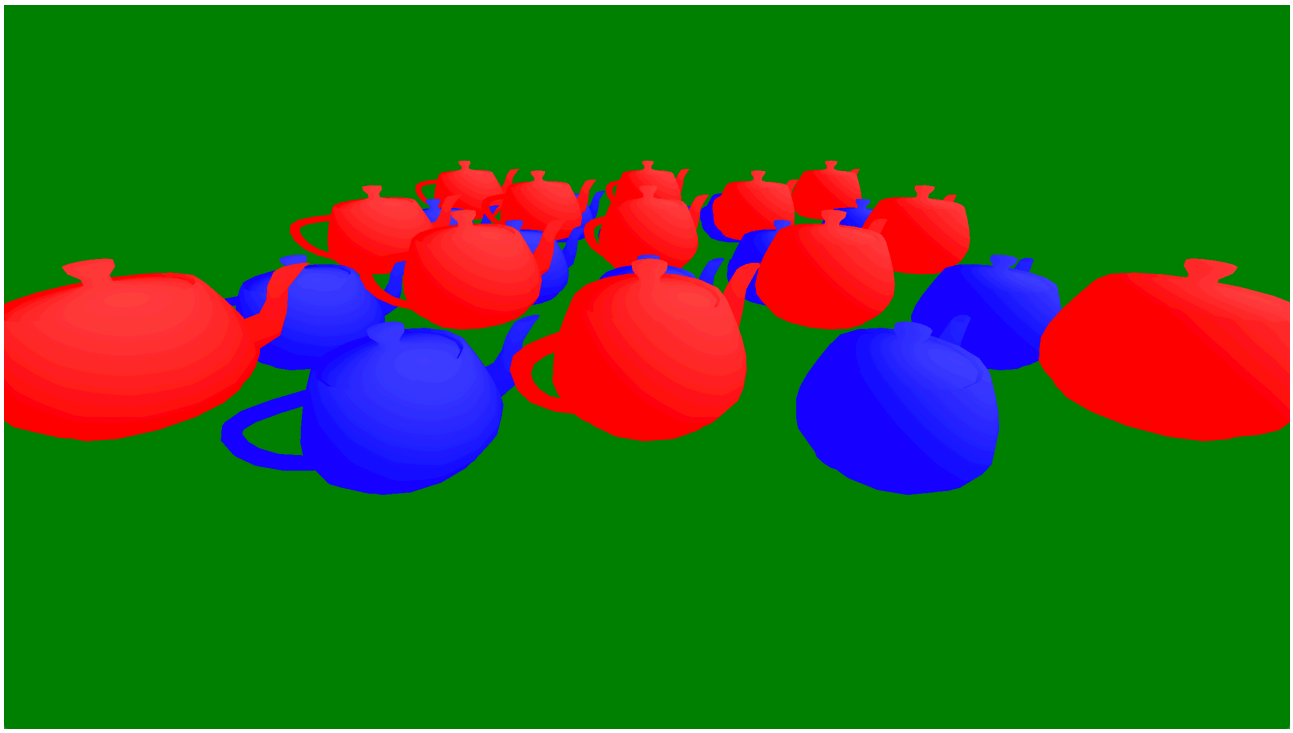

Figure 5.1: The benchmark consists of 25 non-instanced teapots with each having 531 vertices and normals and 3072 indices. Teapots at even positions are controlled by taking the $sin(time)$ while odd positions are using $-sin(time)$. Color and spin of the teapot is computed using the harmonic function. Color and world matrix are uploaded using push constants while view and projection matrices are uploaded using uniform buffers. View and projection matrices don't change but are uploaded anyway to bench their speed. There is a very simple lighting model with hard-coded directional light in the fragment shader.

expected as only one mutex needs to be locked.

submit In this stage the previously recorded command buffer is submitted for execution to Vulkan. This operation is costly on its own, but only minimal overhead is expected.

present In this stage the acquired image is submitted for presentation and a happens-before relationship is estabilished using semaphores and fences so that submitted execution is guaranteed to finish before presentation begins. Again, overhead of a mutex is expected.

wait In this stage the application waits on the presentation fence. This ensures that all measurements done inside one loop are correctly assigned to that loop. In real life applications, this wait should not happen and each frame should asynchronously finish in the background while the user logic computes data for the next frame (akin to the preinit stage). This stage represents the cost of the submitted operations on the GPU, but might also invoke some implementation-dependent synchronization the application is not aware of. Timings of this stage are thus not considered.

update In this stage the update function is called on the window and all outstanding windowing events are handled. This stage represents the update of the windowing system events and a window redraw request and is not being benchmarked.

5.2.2 Results

The benchmarks were run on three hardware and software configurations, note that only the relevant stages are present:

Table 5.1: *Average median time (n = 99000): macOS 10.15.3 (19D76), Quad-Core Intel Core i5, Intel Iris Plus Graphics 655, Vulkan 1.2.135*

Stage	ash	vy_ST	vy_MT
uniform	1.5 us	2.37 us (157%)	2.39 us (159%)
command	23.66 us	24.43 us (103%)	26.51 us (112%)
submit	169.43 us	171.11 us (101%)	170.99 us (101%)
present	32.76 us	33.36 us (102%)	34.14 us (104%)

Table 5.2: *Average median time (n = 99000): Linux 5.4.35_1, Intel i5-7300HQ, Intel HD Graphics 630, Vulkan v1.2.137*

Stage	ash	vy_ST	vy_MT
uniform	717.0 ns	1.53 us (214%)	1.38 us (193%)
command	39.37 us	40.03 us (102%)	39.78 us (101%)
submit	39.57 us	37.36 us (94%)	38.64 us (98%)
present	25.34 us	26.07 us (103%)	26.45 us (104%)

Table 5.3: *Average median time (n = 99000): Linux 5.4.35_1, Intel i5-7300HQ, NVIDIA GeForce GTX 1050 Mobile, Vulkan v1.2.137*

Stage	ash	vy_ST	vy_MT
uniform	1.42 us	2.15 us (152%)	2.24 us (158%)

Stage	ash	vy_ST	vy_MT
command	28.51 us	27.99 us (98%)	28.8 us (101%)
submit	13.15 us	13.76 us (105%)	14.38 us (109%)
present	27.08 us	26.63 us (98%)	27.29 us (101%)

As can be seen, all three tested systems exhibit similar trends. The command stage is on par with pure ash benchmark, the only possible overhead is one mutex lock, which will only have an effect on multi-thread feature in the worst case.

The submit stage also closely follows the ash baseline. This stage potentially locks great number of mutexes, so could be a potential performance bottleneck on the multi-thread feature. However, the intention of an explicit submit operation in Vulkan API is to reduce overhead of submitting smaller batches of work in favor of bigger ones, where the overhead is less noticeable. Thus, for real life applications where the command buffer size will be much bigger, it is expected to be manageable.

The present stage, similaliry, does not exhibit any noticeable slowdown. The reasoning is the same as for the submit stage. Additionaly, the present stage may also include the vertical synchronization delay if enabled, and will thus shadow smaller overhead factors such as locking mutexes.

Finally, the uniform stage exhibits the most interesting results. The accesses performed in Vulkayes are 1.5 to 2 times as slow as when performed by ash. This seems like a lot, but it is important to mention that the absolute difference between the median points is in range of 1 micro second and the overhead is of constant nature.

Furthermore, tbl. 5.4 demonstrates doing 1000 writes into the mapped memory instead of 1 each frame. In fact, Vulkayes is even sligtly faster in this case because it decides on the most efficient strategy for the write, which becomes efficient with larger number of writes.

Table 5.4: *Average median time (n = 99000): macOS 10.15.3 (19D76), Quad-Core Intel Core i5, Intel Iris Plus Graphics 655, Vulkan 1.2.135*

Stage	ash_u1000	vy_ST_u1000	vy_MT_u1000
uniform	45.16 us	40.61 us (90%)	42.14 us (93%)

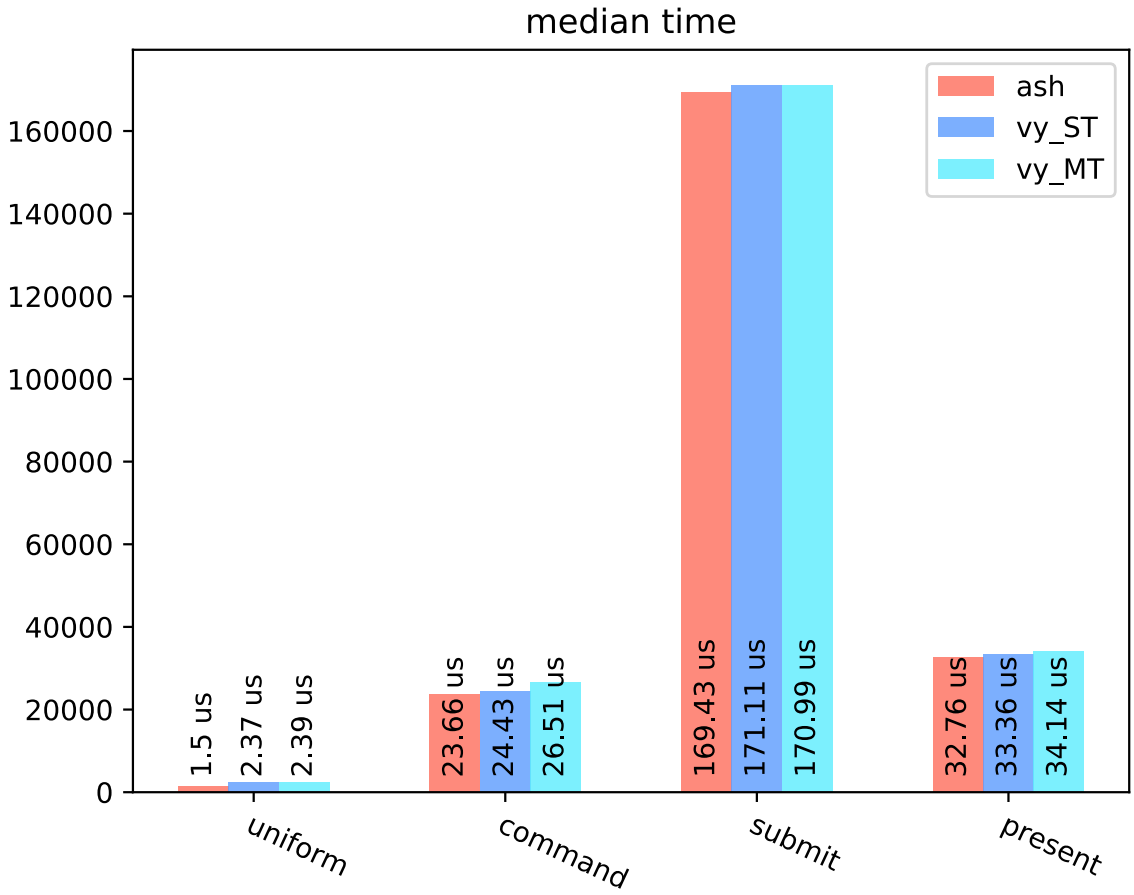

Figure 5.2: *Average median time (n = 99000): macOS 10.15.3 (19D76), Quad-Core Intel Core i5, Intel Iris Plus Graphics 655, Vulkan 1.2.135*

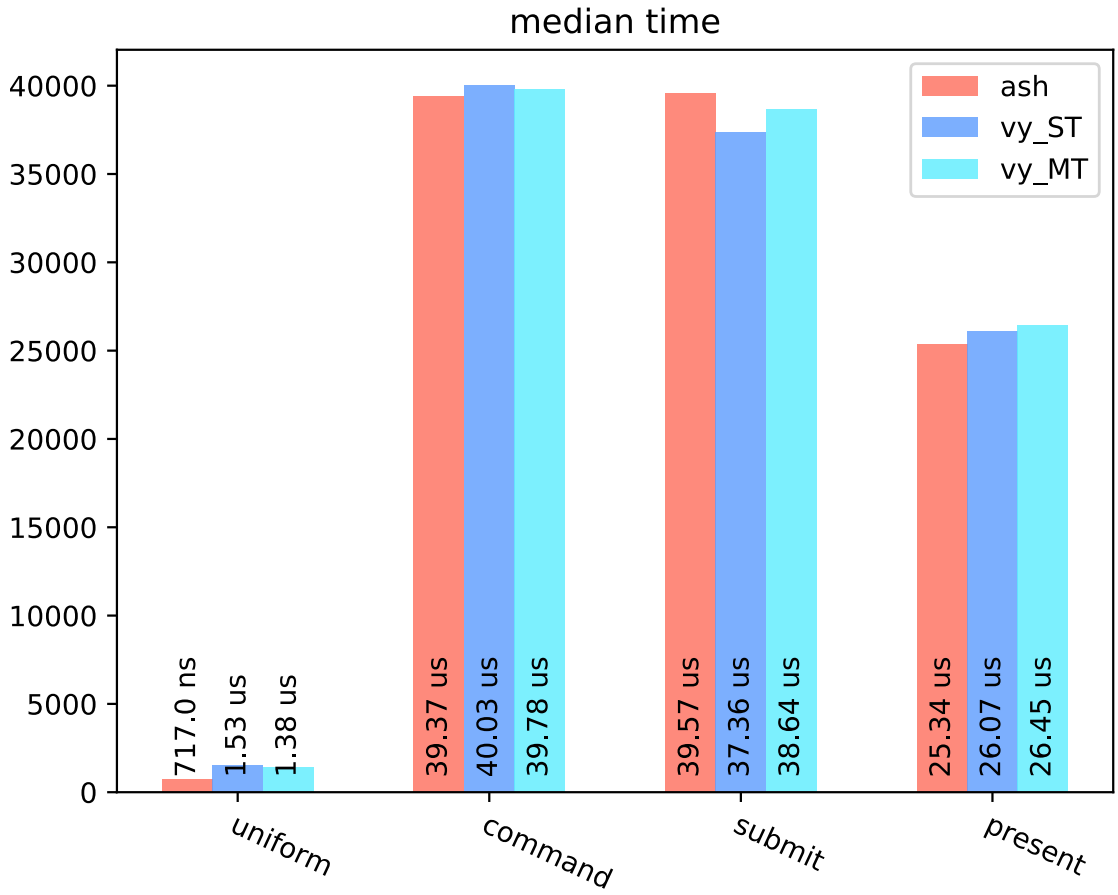

Figure 5.3: *Average median time (n = 99000): Linux 5.4.35_1, Intel i5-7300HQ, Intel HD Graphics 630, Vulkan v1.2.137*

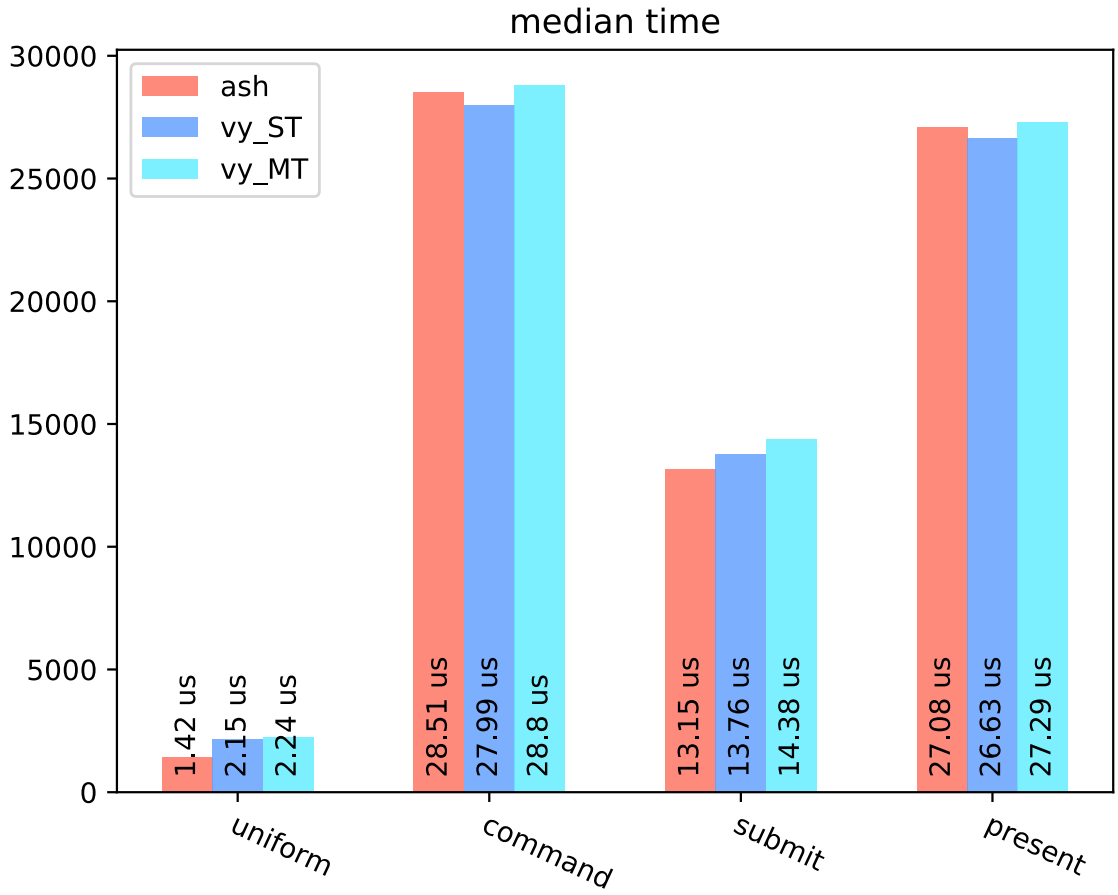

Figure 5.4: *Average median time (n = 99000): Linux 5.4.35_1, Intel i5-7300HQ, NVIDIA GeForce GTX 1050 Mobile, Vulkan v1.2.137*

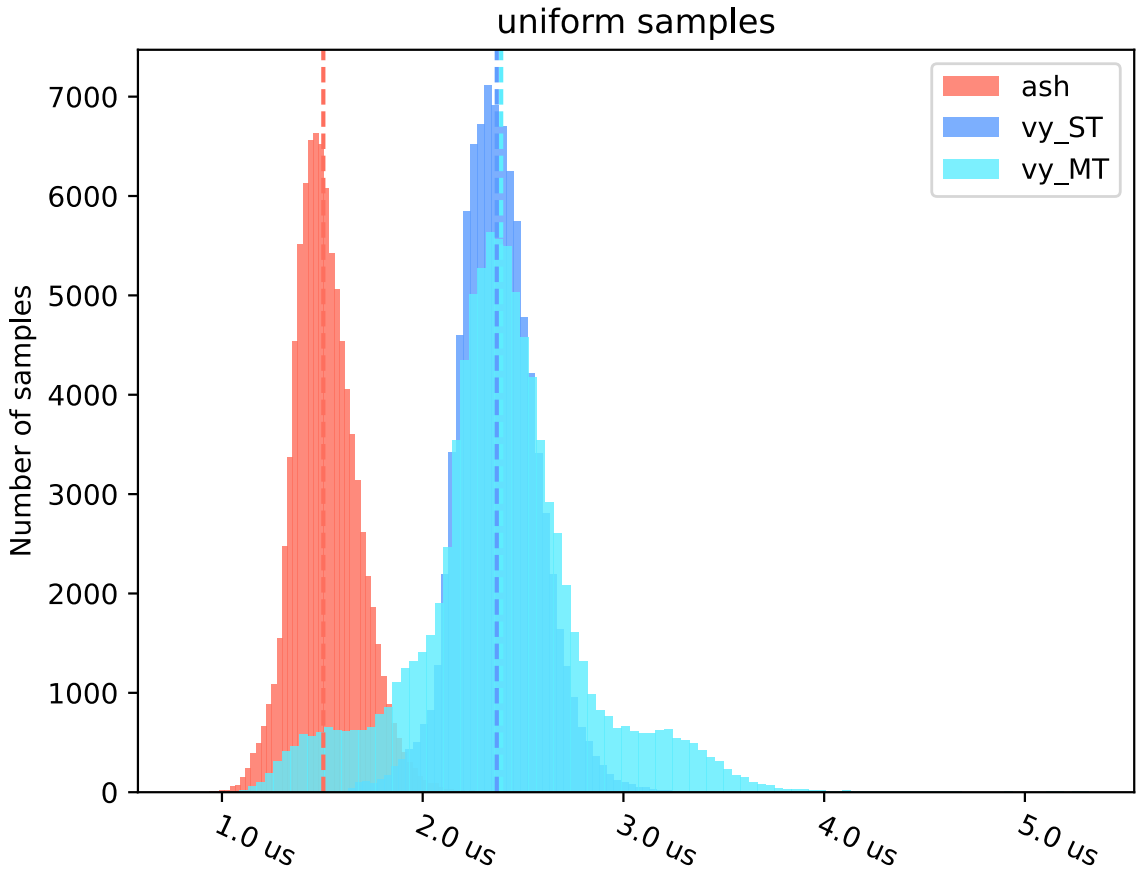

Figure 5.5: *Histogram of uniform stage of the benchmarks (n = 99000). It is clear that ash is faster than both single- and multi-threaded Vulkayes. However, the overhead is constant.*

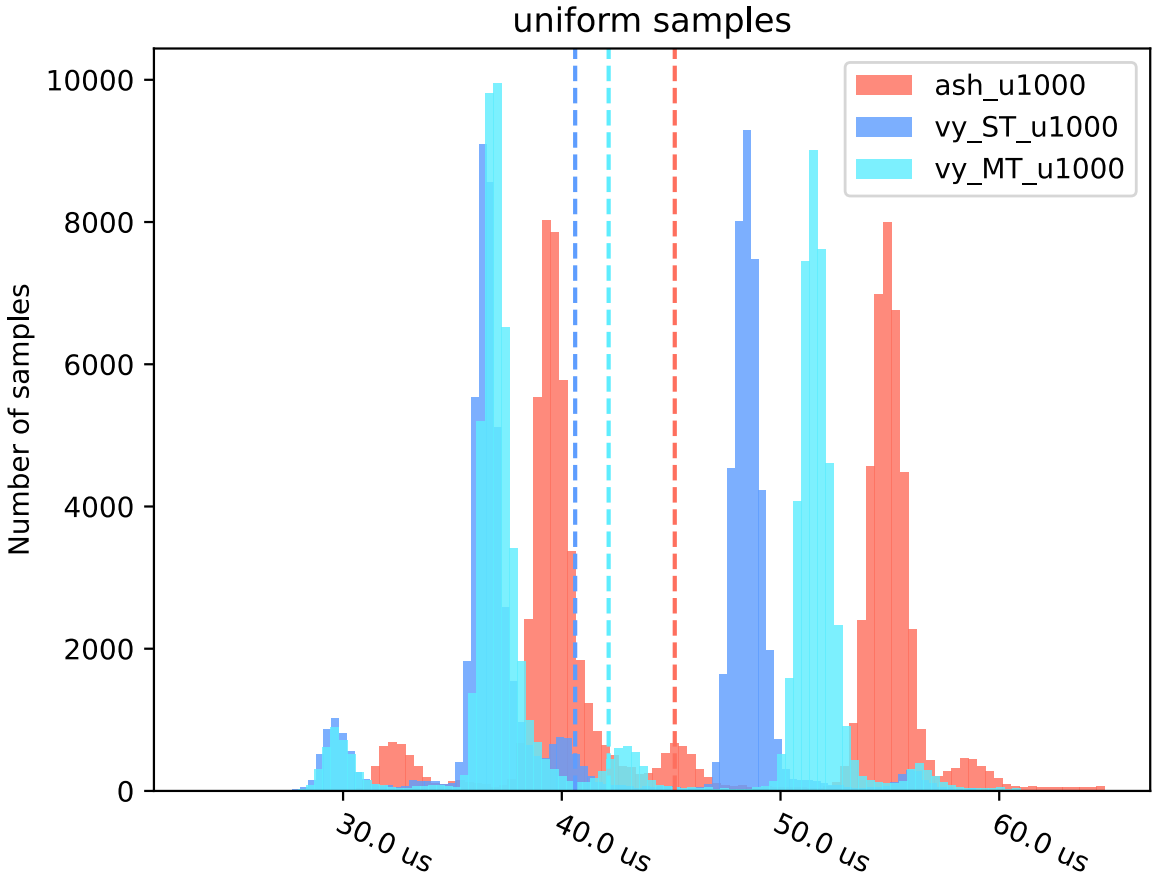

Figure 5.6: *Histogram of uniform stage of the benchmarks (n = 99000) with 1000 writes instead of 1. The overhead displayed in previous bench is overshadowed by the gains of proper writing strategy.*

5.3 Safety

One of the main goals of Vulkayes is increasing safety. This mainly includes memory safety. Vulkayes, being a safe wrapper, provides safe abstraction in the types is wraps in both the Rust way and the Vulkan API way.

Table 5.5: *Vulkan API validations status in the project.*

Category	Statically solved	Dynamically solved	Left to user	Total
Implicit	317	28	2	347
Creation	91	0	314	405
Usage	29	3	122	154
Total	437	31	438	906

In tbl. 5.5 it can be seen that the goal was achieved almost perfectly. Only two implicit validations are left to the user. This decision wasn't made lightly, but it was chosen as the most sensible one given the current limitations of the stable version of language. A small number of implicit validations couldn't be solved statically. These validations are instead checked at runtime, but only conditionally under the `runtime_implicit_validations` Cargo feature. All other implicit validations were successfully solved statically. More details about the specific validations can be found on the included CD.

Additionally, a significant amount of explicit validations, categorized under creation and usage, have been solved statically as a consequence of the natural API design and/or the implicit validations. Overall this means increased safety for the user of the API at no runtime cost.

6 Conclusion

The core Vulkayes library is successful at reducing the complexity of creating and using Vulkan types, as well as correctly destroying them at appropriate times and checking basic safety requirements. Benchmarks show that this added complexity is mostly compile-time and scales well into the runtime where applicable. Additionaly, safety is guaranteed at a certain level that should provide the user of the API with certain amount of confidence that their application will not segfault. Overall, the Vulkayes project is a good step towards a flexible and transparent Vulkan API in the Rust ecosystem, learning from previous mistakes and designs.

However, there still remains a lot of work to be done to create an API with a application design advantage as well. Designing synchronization in Vulkan by hand is error prone due to high complexity and Vulkayes should be extended with user-friendly API that is capable of lifting the burden off the user onto the implementation, prefferably mostly at compile time. Declarative synchronization definition API and other improvements to Vulkayes are left for future work.

Bibliography

[1] “Vulkan® 1.2.136 - A Specification.” [Online]. Available: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html. [Accessed: 05-Apr-2020]

[2] “Vulkano.” [Online]. Available: https://github.com/vulkano-rs/vulkano. [Accessed: 10-Apr-2020]

[3] “Khronos Releases Vulkan 1.0 Specification.” [Online]. Available: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification. [Accessed: 29-Apr-2020]

[4] “Dota 2 Update - May 23rd 2016.” [Online]. Available: https://store.steampowered.com/news/22000/. [Accessed: 05-May-2020]

[5] “LunarG.” [Online]. Available: https://www.lunarg.com/. [Accessed: 06-May-2020]

[6] “Khronos Releases Vulkan 1.2 Specification.” [Online]. Available: https://www.khronos.org/news/press/khronos-group-releases-vulkan-1.2. [Accessed: 29-Apr-2020]

[7] “V-EZ.” [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ. [Accessed: 10-Apr-2020]

[8] “gfx-hal.” [Online]. Available: https://github.com/gfx-rs/gfx. [Accessed: 10-Apr-2020]

[9] R. Galajda, “Designing a modern high-level graphics API,” Czech Technical University in Prague. Computing and Information Centre., 2020.

[10] “The Development of the C Language.” [Online]. Available: https://www.bell-labs.com/usr/dmr/www/chist.html. [Accessed: 10-Apr-2020]

[11] “The Cargo Book.” [Online]. Available: https://doc.rust-lang.org/cargo/. [Accessed: 30-Apr-2020]

[12] “ash.” [Online]. Available: https://github.com/MaikKlein/ash. [Accessed: 29-Apr-2020]

[13] “Vulkan Memory Allocator.” [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator. [Accessed: 30-Apr-2020]

Contents of the included CD

The CD contains a copy of the vulkayes-thesis repository.

Table 1: *Structure of the repository*

Directory	Description
codes	Code snipptes comparing selected features of Rust and C++ and Rust code snippet benchmarks.
documents	Documents containing topics covered in the final thesis and the thesis itself.
assets	Images, diagrams and other assets used in documents and thesis.
scripts	Scripts for building the thesis.
pdfs	Generated output pdfs.

More information can be found in the `README.md` file inside the repository.