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

Thanky.

Declaration

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

.....

Abstract

English abstract

Keywords: key word here

Abstrakt

Slovenský abstrakt

Kľúčové slová: zámok kľúč hrad

Preklad názvu: Implementácia zobrazovacieho systému v jazyku Rust

Contents

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.

Talk about Vulkan API and why it is great

Talk about what the project aims to achieve in short and long term, mention Rust

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."[2]

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[2].

2.2 gfx-hal

gfx-hal or graphics hardware abstraction layer[3] 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.

2.3 Vulkano

Vulkano[4] 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 Vulkano's `ImmutableImage` (which is intended as one-time write image abstraction, e.g. for textures in games).

2.4 Tephra

TODO: How to correctly cite this?

Tephra 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.

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 unreasonably to 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[5]. 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:

Ownership Rust implements a very simple but powerful ownership model. Values are always moveable. 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. Ownership has serious consequences: the owner has to clean up. Values that have non-trivial destructors should run those destructors at some point.

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.

In contrast, Rust statically prevents use of moved-out variables. Once you move a value out of a variable, 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. 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 also 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 rade conditions. A reference is created, then the original referred value is destroyed 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 taked from it. 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.

3.1.1 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.

3.2 Object lifetime management

Objects in Vulkan have certain lifetime dependencies - some objects must outlive others - displayed in the diagram[TODO Figure number thing]. 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.

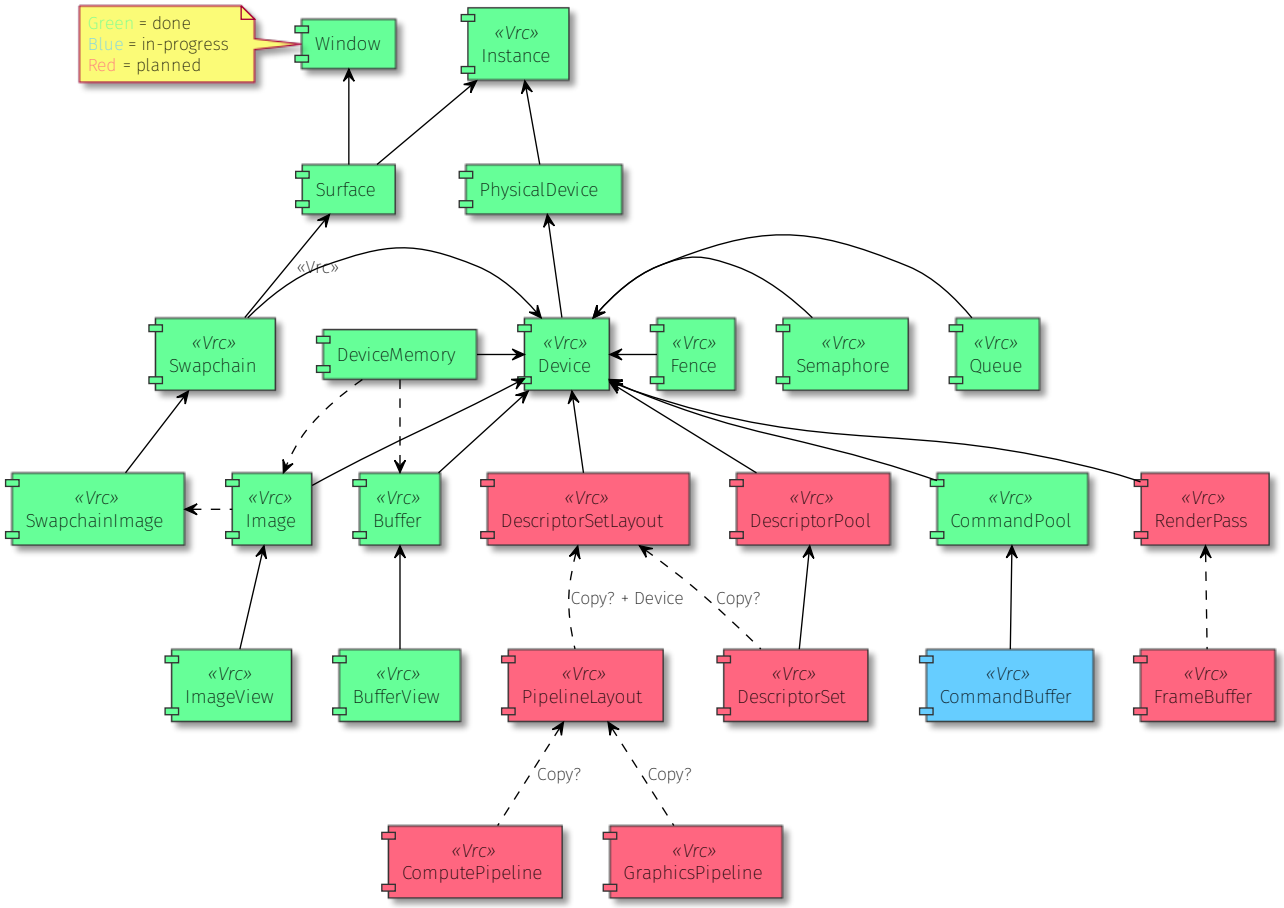

Figure 3.1: Object Dependency Graph

3.3 Memory management

Talk about how device memory management is done through user-supplied memory allocator

3.4 Synchronization

Talk about how some objects are internally synchronized

Talk about how GPU synchronization is left for future work

3.5 Validations

Talk about how only implicit validation are guaranteed, but some explicit validations are implicitly handled by api design and type system

3.6 Windows

Talk about how windows are handle, what is a surface and a swapchain and how they are supported

4 Implementation

4.1 Cargo features

Talk about what cargo features are available and why they are beneficial

4.1.1 Vrc, Vutex etc.

Talk about Vrc and Vutex aliases

4.2 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 comform 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.

4.2.1 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 `Holder`s 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.

4.2.2 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 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.

4.2.3 Generics in Vulkayes

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.

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 (about 128 to 256 MB) 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. 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 my mapping the memory into host memory using Vulkan provided mechanism and then writing to it as if it was normal host memory.

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`.

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.

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.

// TODO: Reference to the benchmark code

4.2.4 Deref

Talk about Deref trait usage

4.2.5 User code a.k.a. `dyn FnMut`

Talk about usage of `dyn`

4.3 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 three 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>,
    // Other fields
}
impl Swapchain {
    pub fn new(
        surface: Surface,
        // Other parameters
    ) -> Vrc<Self> {
        Vrc::new(
            Swapchain {
                surface: Vrc::new(surface),
                // Other fields
            }
        )
    }

    pub fn recreate(
        self: &Vrc<Self>,
        // Other parameters
```

```
    ) -> Vrc<Self> {
        Vrc::new(
            Swapchain {
                surface: self.surface.clone(),
                // Other fields
            }
        )
    }

    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.

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.

Original examples repository from `ash` crate has 1820 lines of Rust code. Current examples repository adapted to Vulkayes has 1485 lines of Rust code. This is a difference of 335 lines of Rust code.

Below is an example of the code with same functionality from the original examples and from the current ones. The code after is two times shorter than the original code while exposing the same functionality and providing static validation guarantees.

// TODO: Revisit this after some benchmarks

Before:

```
let vertex_input_buffer_info = vk::BufferCreateInfo {
    size: std::mem::size_of_val(&vertices) as u64,
    usage: vk::BufferUsageFlags::VERTEX_BUFFER,
    sharing_mode: vk::SharingMode::EXCLUSIVE,
    ..Default::default()
};
let vertex_input_buffer = base
    .device
    .create_buffer(&vertex_input_buffer_info, None)
    .unwrap();
let vertex_input_buffer_memory_req = base
    .device
    .get_buffer_memory_requirements(vertex_input_buffer);
let vertex_input_buffer_memory_index = find_memorytype_index(
    &vertex_input_buffer_memory_req,
    &base.device_memory_properties,
    vk::MemoryPropertyFlags::HOST_VISIBLE | vk::MemoryPropertyFlags::HOST_COHERENT,
)
.expect("Unable to find suitable memorytype for the vertex buffer.");

let vertex_buffer_allocate_info = vk::MemoryAllocateInfo {
    allocation_size: vertex_input_buffer_memory_req.size,
    memory_type_index: vertex_input_buffer_memory_index,
    ..Default::default()
};
let vertex_input_buffer_memory = base
    .device
    .allocate_memory(&vertex_buffer_allocate_info, None)
    .unwrap();

let vert_ptr = base
```

```
    .device
    .map_memory(
        vertex_input_buffer_memory,
        0,
        vertex_input_buffer_memory_req.size,
        vk::MemoryMapFlags::empty(),
    )
    .unwrap();
let mut slice = Align::new(
    vert_ptr,
    align_of::<Vertex>() as u64,
    vertex_input_buffer_memory_req.size,
);
slice.copy_from_slice(&vertices);
base.device.unmap_memory(vertex_input_buffer_memory);
base.device
    .bind_buffer_memory(vertex_input_buffer, vertex_input_buffer_memory, 0)
    .unwrap();
```

After:

```
let vertex_buffer = {
    let buffer = Buffer::new(
        base.device.clone(),
        std::num::NonZeroU64::new(std::mem::size_of_val(&vertices) as u64).unwrap(),
        vk::BufferUsageFlags::VERTEX_BUFFER,
        base.present_queue.deref().into(),
        buffer::params::AllocatorParams::Some {
            allocator: &base.device_memory_allocator,
            requirements: vk::MemoryPropertyFlags::HOST_VISIBLE
                | vk::MemoryPropertyFlags::HOST_COHERENT
        },
        Default::default()
    )
    .expect("Could not create index buffer");

    let memory = buffer.memory().unwrap();
    memory
        .map_memory_with(|mut access| {
            access.write_slice(&vertices, 0, Default::default());
            MappingAccessResult::Unmap
        })
        .expect("could not map memory");

    buffer
};
```

Talk about benchmarks

5.2 Scene 1

5.3 Scene 2

5.4 Scene 3

6 Conclusion

Conclude

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

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

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

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