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

Related work

Talk about V-EZ, Vulkano, gfx-hal, mention tephra

Design

Something and more

Rust

Talk about why Rust was chosen, include cpp vs Rust examples

Drop

The term `drop` refers to what is often called `destructor` in OOP languages. It is a piece of code that runs at most once exactly before the object is destroyed. In Rust, a user-provided `drop` code can be provided by implementing the `Drop`[2] trait for your type:

```
pub trait Drop {
    fn drop(&mut self);
}
```

Structures in Rust are dropped recursively, in this order:

1. The `Drop` implementation (if any) is run for the outer type
2. Each field of the struct is recursively dropped starting from the 1. point
3. The object itself is destroyed by the language implementation

It is obvious that the only way to influence this process is by implementing the `Drop` trait.

Partial dropping

In the case of the `Surface<Window>` structure, it was needed to allow the user to retrieve the owned `Window` generic parameter if the user desired to work with the window after the library was done with it. We would like a method that looks like `pub fn drop_without_window(self) -> Window;`. The structure owns the window because we need to uphold an invariant that the window always outlives the surface.

First attempt At first, the structure looked like:

```
pub struct Surface<Window> {
    window: Window,
    loader: ash::extensions::khr::Surface,
```

```
    surface: ash::vk::SurfaceKHR
}
impl<W> Surface<W> {
    pub fn drop_without_window(self) -> Window {
        self.window // this will not compile because moving window out of self is a compi

        // here the `self` parameter would be implicitly dropped, but it is now
        // in an inconsistent state that the compiler cannot reason about because
        // of non-trivial drop implementation
    }
}
impl<W> Drop for Surface<W> {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
```

However, manually implementing `Drop` for a type prevents destructuring of that type. That is the implementation of `drop_without_window` will not compile because that would prevent the `Drop` implementation of surface from running. This is a problem because this prevents us from extracting the `window` field out of `surface`, simply because we cannot prove to the compiler in a **safe** manner that the drop code does not depend on `window` field being valid and not moved out of.

Second attempt Second solution was to move the fields with custom `drop` code into a separate inner struct:

```
struct InnerSurface {
    loader: ash::extensions::khr::Surface,
    surface: ash::vk::SurfaceKHR
}
impl Drop for InnerSurface {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
pub struct Surface<Window> {
    // wrong order of fields, window will be dropped before inner
    window: Window,
    inner: InnerSurface
}
impl<W> Surface<W> {
    pub fn drop_without_window(self) -> Window {
```

```
        self.window

        // Implicitly drop self.inner at the end of this scope, window is returned
    }
}
```

This code compiles and works. However, the `unsafe` block in the `drop` implemnetation has non-trivial invariants that need to be uphelp and they are not: `window` needs to outlive `inner`.

Drop order of fields in structs in Rust is defined to be in the order of declaration, which is sometimes non-obivous and needs good documentation so that nobody accidentally moves struct fields around. The example above declares `window` field before `inner` field which would result in the wrong drop order and cause problems.

This solution is viable, however, relying on the drop order in Rust has been slightly controversial, as it clashes with the notion that the declaration order of fields in struct does not imply their memory layout. Indeed, while drop order has been stabilized in [3], it is still recommended for clarity to use `std::mem::ManuallyDrop` when something non-trivial is happening with drop.

Third attempt Third and final attempt was inspired by the recommendation in the documentation of `ManuallyDrop`[4]. The code looks like this:

```
struct Surface<Window> {
    window: Option<Window>,
    loader: ash::extensions::khr::Surface,
    surface: ash::vk::SurfaceKHR
}
impl<W> Surface<W> {
    pub fn drop_without_window(mut self) -> Window {
        self.window.take().unwrap()
        // This will never panic as there is no way to create instance of Surface
        // without window set as Some. However, if you somehow do manage to create
        // such instance *without undefined behavior*, no undefined behavior will occur.
        // The compiler should also be able to reason that the value of window will
        // never be `None` and optimize the branch out.

        // here self is still in valid state and is implicitly dropped in full
    }
}
impl<W> Drop for Surface<W> {
    fn drop(&mut self) {
        unsafe {
            self.loader.destroy_surface(self.surface, None);
        }
    }
}
```

This code upholds all invariants, does not require additional `unsafe` code and makes it obvi-

ous that `window` is not a normal field but something with special logic. In the end, this code is not only safest of all the alternatives, but also the easiest to implement.

In C++

For comparison, this problem in C++ be much harder to solve correctly. Consider the following two programs:

```
int main() {
    std::cout << ">> Moving out" << std::endl;
    {
        Window original(1);
        std::cout << "original " << original.a << std::endl << std::endl;

        Surface<Window> surface(std::move(original));
        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl << std::endl;

        Window moved = surface.destroy_without_window();

        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl;
        std::cout << "moved out " << moved.a << std::endl << std::endl;
    }

    std::cout << std::endl << ">> Not moving out" << std::endl;
    {
        Window original(2);
        std::cout << "original " << original.a << std::endl << std::endl;

        Surface<Window> surface(std::move(original));
        std::cout << "original " << original.a << std::endl;
        std::cout << "in surface " << surface.window.a << std::endl << std::endl;
    }
}
```

```
fn main() {
    println!(">> Moving out");
    {
        let original = Window::new(1);
        println!("original {:?}\n", original);

        let surface = Surface::new(original);
        // println!("original {:?}", original); // Compiler error, original was moved
        println!("in surface {:?}\n", surface.window);

        let moved = surface.destroy_without_window();
        // println!("original {:?}", original); // Compiler error, original was moved
```

```
        // println!("in surface {:?}", surface.window); // Compiler error, surface was mo
        println!("moved out {:?}\n", moved);
    }

    println!("\n>> Not moving out");
    {
        let original = Window::new(2);
        println!("original {:?}\n", original);

        let surface = Surface::new(original);
        println!("in surface {:?}\n", surface.window);
    }
}
```

Note: Full implementation of both these programs is available in the appendix.

In the Rust version, the compiler provides move semantics, protects us from ever using a value that was moved and the program behaves as expected. The `surface` is destroyed exacly when `destroy_without_window` is called.

In contrast, the C++ version requires us to implement explicit move constructor. The moved value is nothing but an instance of the original class with some marker value inside that tells us not to really destroy it in the destructor, but it still runs destructors for all the moved instances. Additionally, the surface is not destroyed in the line that cleary says `destroy_without_window`, it is destroyed when it goes out of scope at the end of the block. All of this places great strain on the programmer, who is much more error prone, instead of the compiler.

Object lifetime management

Talk about how object lifetime is managed, maybe compare to tephra, talk about cargo features toggling

Synchronization

Talk about how some objects are internally synchronized

Talk about how GPU synchronization is left for future work

Validations

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

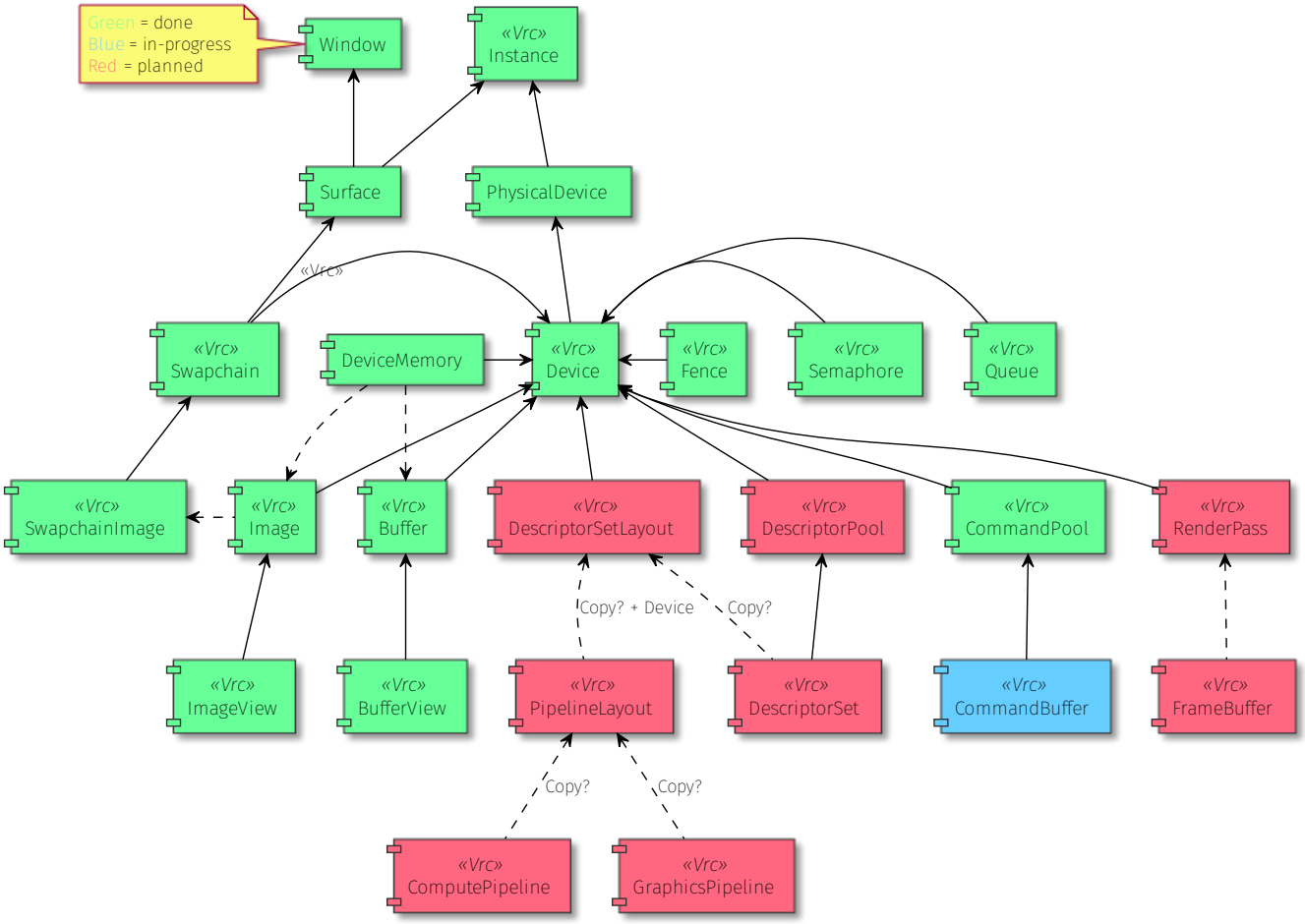

Figure 1: Object Dependency Graph

Memory management

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

Windows

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

Implementation

Cargo features

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

Vrc, Vutex etc.

Talk about Vrc and Vutex aliases

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.

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.

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.

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

Deref

Talk about Deref trait usage

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

Talk about usage of `dyn`

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.

Evaluation

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

Scene 1

Scene 2

Scene 3

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] “Drop.” [Online]. Available: https://doc.rust-lang.org/std/ops/trait.Drop.html. [Accessed: 03-Mar-2020]

[3] “RFC-1857 Stabilize Drop Order.” [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/1857-stabilize-drop-order.md. [Accessed: 03-Mar-2020]

[4] “ManuallyDrop.” [Online]. Available: https://doc.rust-lang.org/std/mem/struct.ManuallyDrop.html. [Accessed: 03-Mar-2020]