Handling 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