Implementation of rendering system in Rust

Eduard Lavuš

Faculty of Electrical Engineering

Czech Technical University in Prague

2020-06-01

Motivation and aim

- Motivation
  - Abandoned open-source projects with similar aim
  - Rust safety features exploration
- Aim
  - Designing and implementing flexible and transparent high-level Vulkan API wrapper
  - Comparing design to previous attempts and measuring development and performance cost
  - Creating the “core” for future work to build upon

Structure of the thesis

- Introduction to Vulkan
- Overview of existing projects in both Rust and C++
- Design principles and Rust features
- Implementation details and difficulties
- Evaluation
  - Developer experience
  - Performance
  - Safety

Rust and Vulkan

- Rust
  - Fast
  - Flexible
  - Safe
  - Developer friendly
- Vulkan
  - Fast
  - Flexible
  - Unsafe
  - Developer unfriendly

Design and implementation

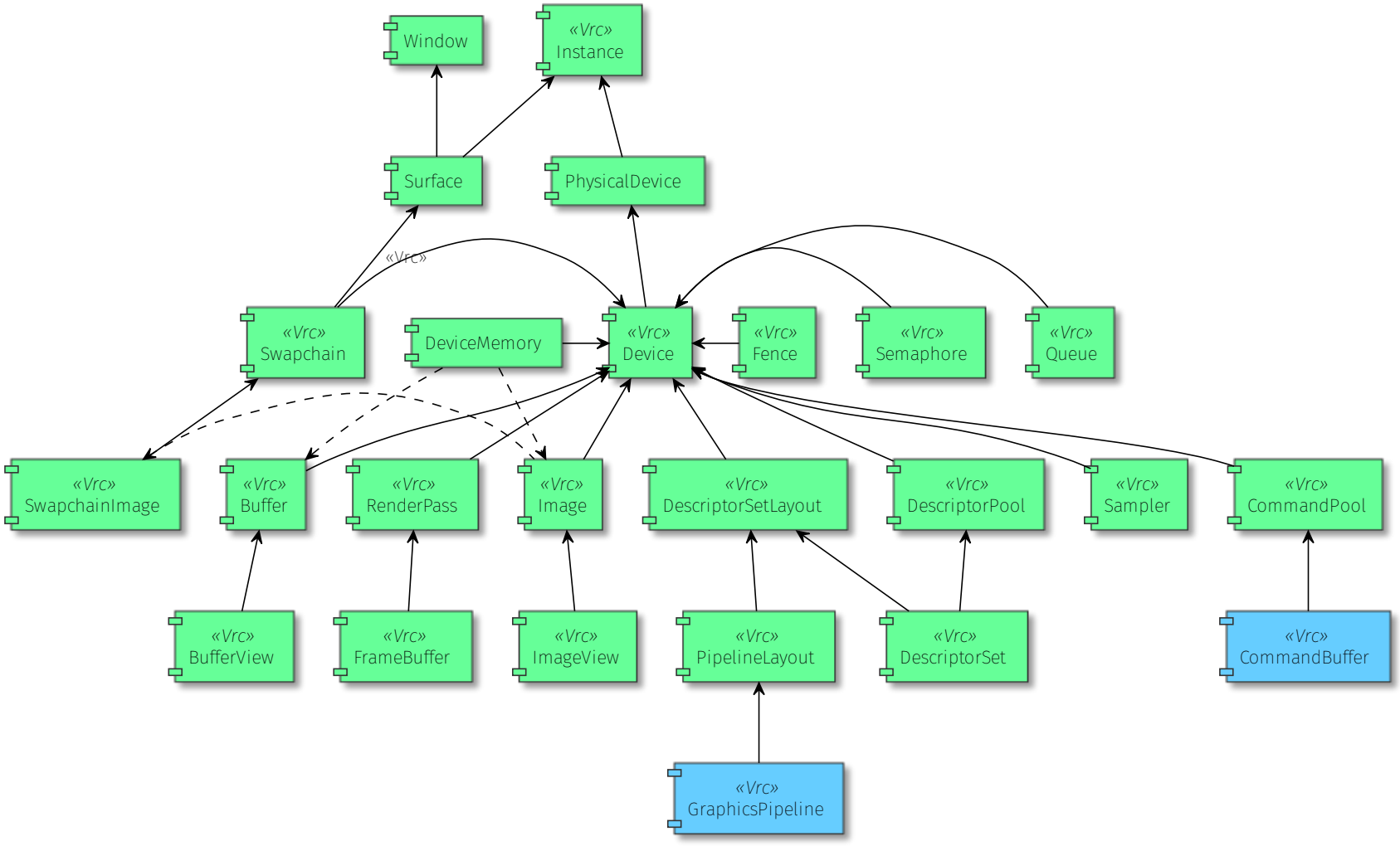

Figure 1: Object Dependency Graph of Vulkayes

Design and implementation

Project name: Vulkayes

- Design
  - Transparent
  - Minimal overhead
  - Statical safety
- Implementation
  - Cargo features
  - Vrc, Deref, generics
  - Flexibility

Results and evaluation

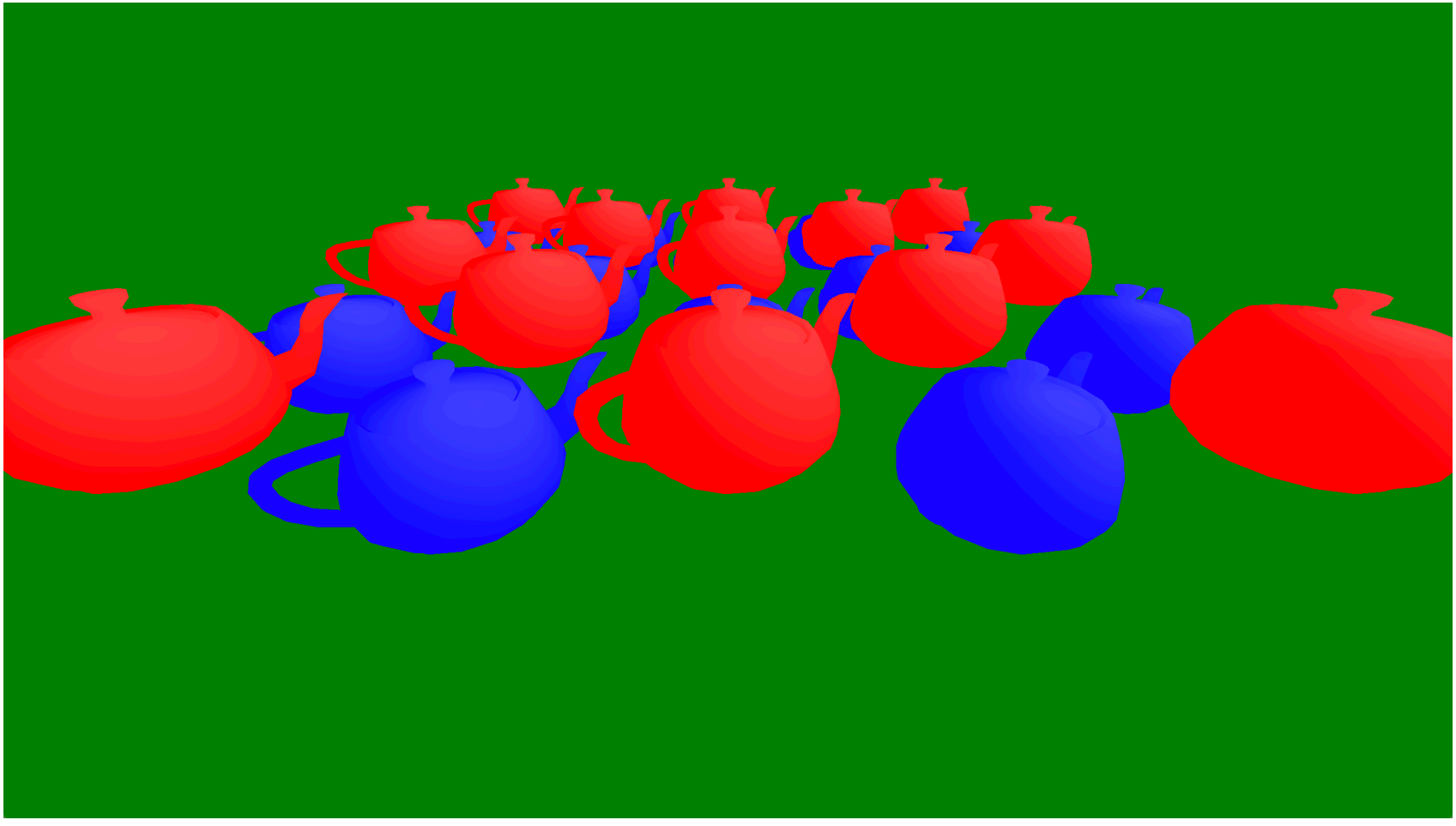

Figure 2: Benchmarking application output

Results and evaluation

- Developer experience
  - In selected code snipptes, Vulkayes code is three times shorter than equivalent ash code
  - The benchmarking program code in Vulkayes is 33% shorter than in ash
- Performance
  - Vulkayes was evaluated against ash, the bindings to Vulkan API that are used, as a baseline
  - `vy_ST` represents single-threaded version of Vulkayes
  - `vy_MT` represents Vulkayes with multi-threading enabled
- Safety
  - All but two Vulkan implicit validations were solved: 317 statically and 28 dynamically
  - 21% (120) of explicit validations were solved statically as a byproduct of good API design

Results and evaluation

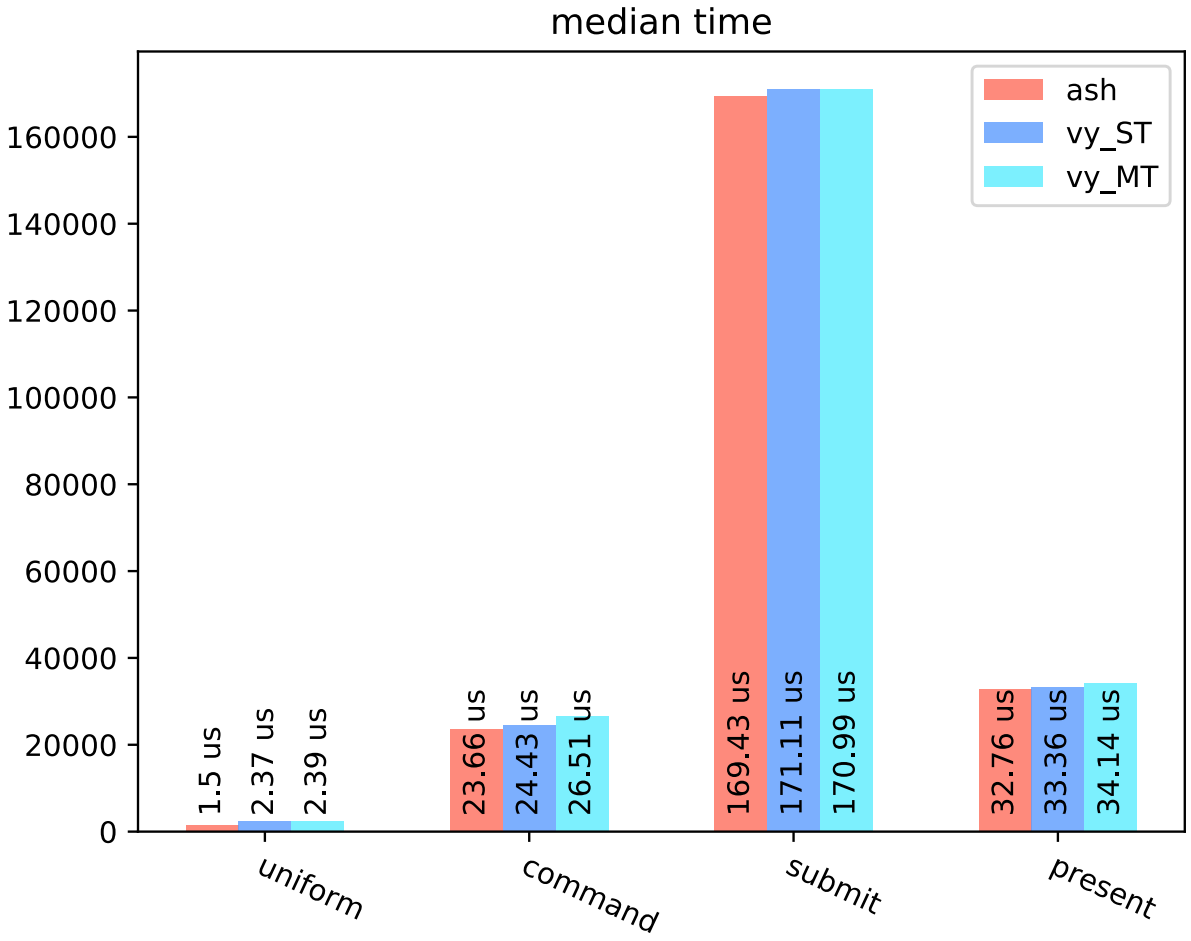

Figure 3: *Average median time (n = 99000): macOS 10.15.3 (19D76), Quad-Core Intel Core i5, Intel Iris Plus Graphics 655, Vulkan 1.2.135*

Results and evaluation

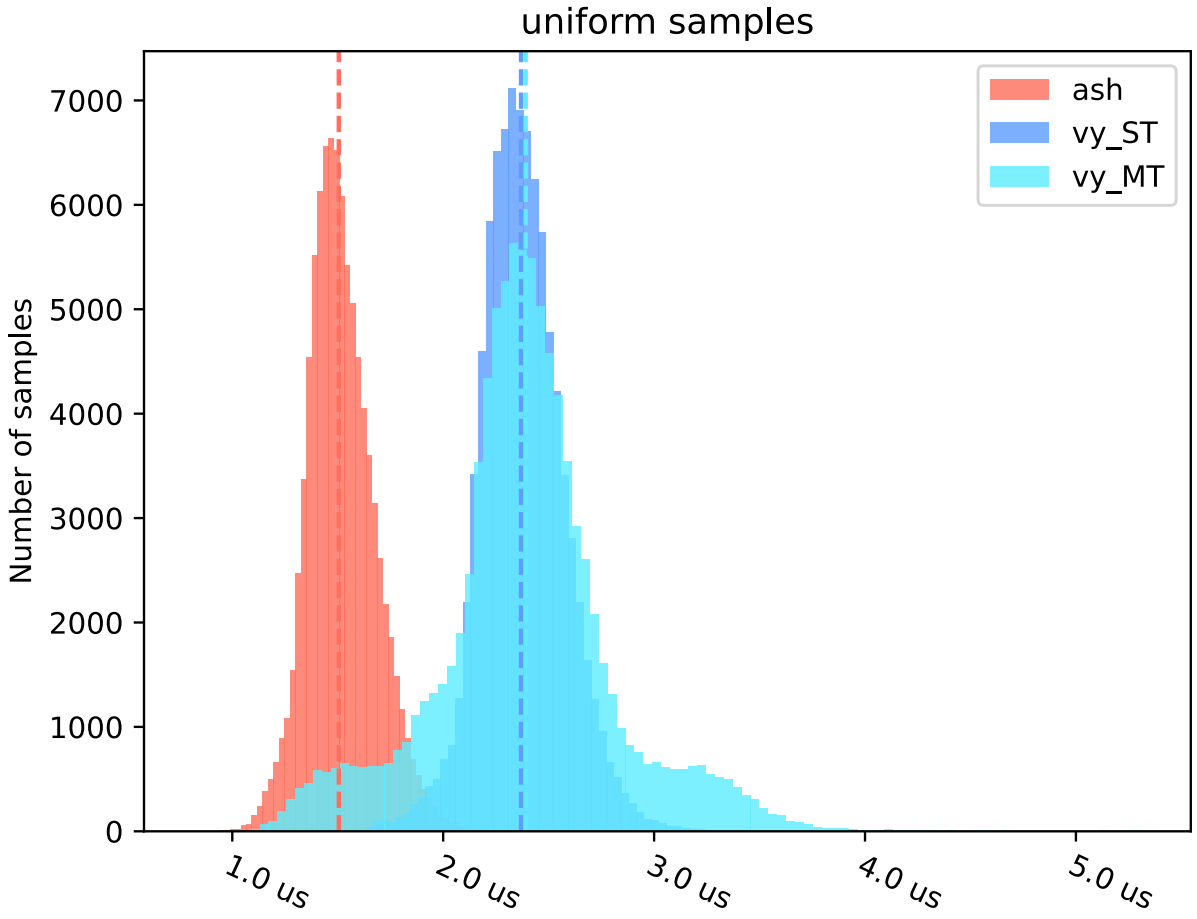

Figure 4: *Histogram of uniform stage of the benchmarks (n = 99000). It is clear that ash is faster than both single- and multi-threaded Vulkayes. However, the overhead is constant.*

Results and evaluation

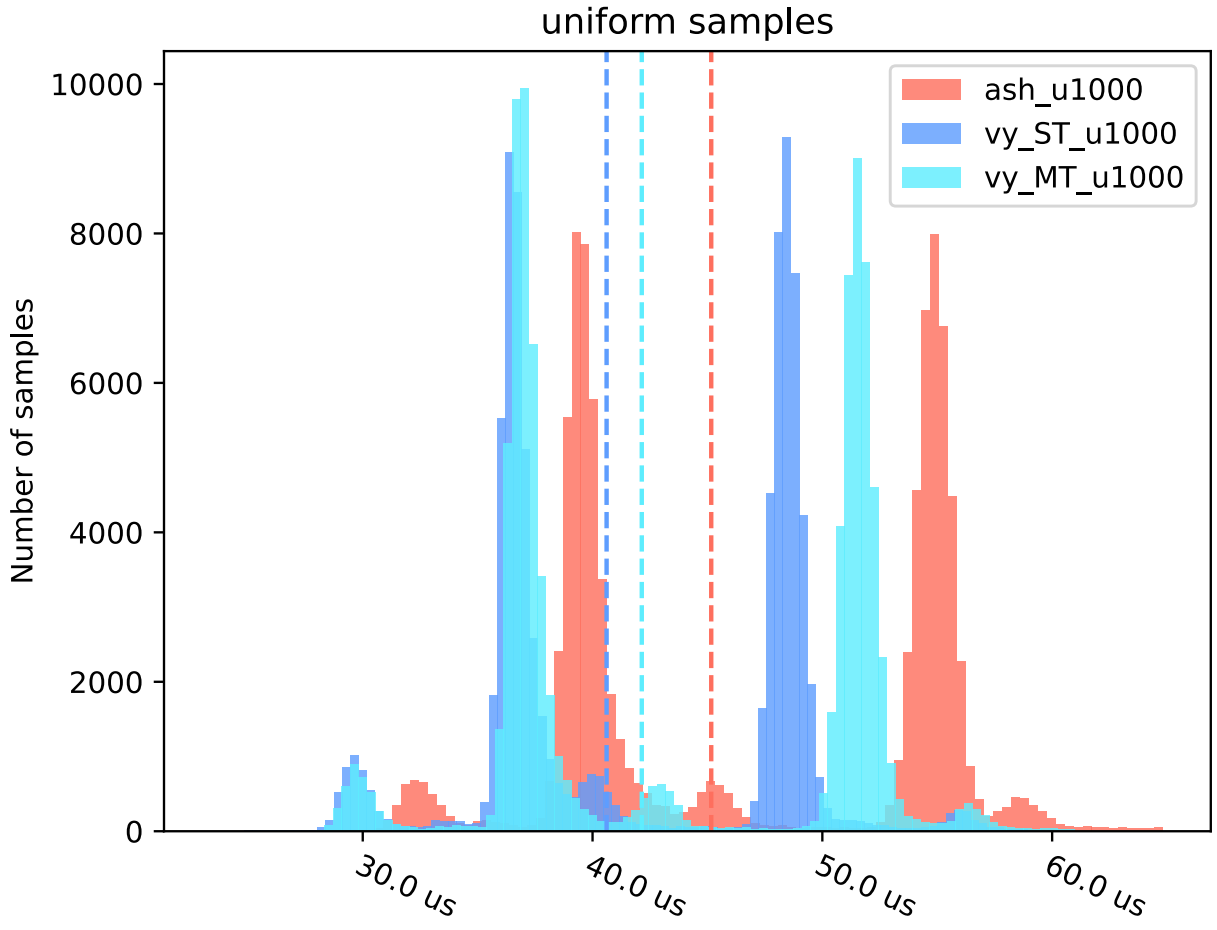

Figure 5: *Histogram of uniform stage of the benchmarks (n = 99000) with 1000 writes instead of 1. The overhead displayed in previous bench is overshadowed by the gains of proper writing strategy.*

Conclusion

- Code written using Vulkayes is shorter but still flexible
- Vulkayes performs as fast as ash
- Safety is greatly increased thanks to both Rust and API design
- Vulkayes is a good step towards a more complex modular solution

Unanswered questions

- What do you see as the biggest advantage of your system?
  - Transparency over bindings (ash). This allows Vulkayes to grow iteratively with unfinished parts being written in ash until they are implemented. It is great for prototyping and also allows for benchmarking code very selectively.

Unanswered questions

- Subsection 3.4 mentions a need to correctly synchronize CPU and GPU in respect to sharing resources. Do you have have any ideas about possible solutions?
  - I've done a lot of work on Vulkano synchronization to make it as generic as possible before starting Vulkayes. I later learned that Tephra does something quite similar. It would seem like that is a good way to start, so porting the work I did on Vulkano would be my first attempt. However, controlling the synchronization is very hard, because it requires full control over the environment (like Vulkano and Tephra do). I would like to think that we can do better over time.

Unanswered questions

- Subsection 3.3 references a recommentation from Vulkan about allocating memory in bigger chunks (e.g. 256MB). From the implementation description it is not quite clear how exactly allocation happens in Vulkayes, respectively who is responsible for effective allocation/deallocation. Is it the programmer, Rust or Vulkayes?
  - Vulkayes core (this work) only exposes interface (Rust trait and some structs) for dealing with device memory allocations. Integration of specific memory allocators is required (such as the VMA), but outside of scope of the core library. However, Vulkayes does provide a Cargo feature `naive_memory_allocator` which provides a very simple implementation of this core interface to allow quick prototyping. There is a plan to support VMA explicitly through another crate (similar to `vulkayes-window`).