Implementation of rendering system in Rust

Eduard Lavuš

Faculty of Electrical Engineering

Czech Technical University in Prague

Motivation and aim

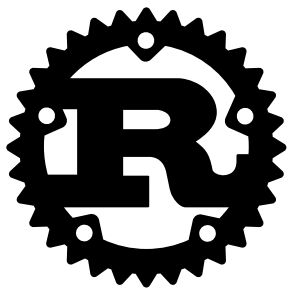
- Motivation
  - Abandoned open-source projects with similar aim
    - Vulkano and partially gfx-rs
  - Rust safety features exploration
- Aim
  - Designing and implementing flexible and transparent high-level Vulkan API wrapper
  - Comparing design to previous attempts and measuring development and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

Rust and Vulkan

Vulkan API	C++	Rust
Fast	Fast	Fast
Flexible	Flexible	Flexible
Unsafe	Unsafe	Safe
Developer unfriendly	Developer hard	Developer friendly

C++ has a lot of legacy.

Use Rust to make Vulkan safer and more friendly.

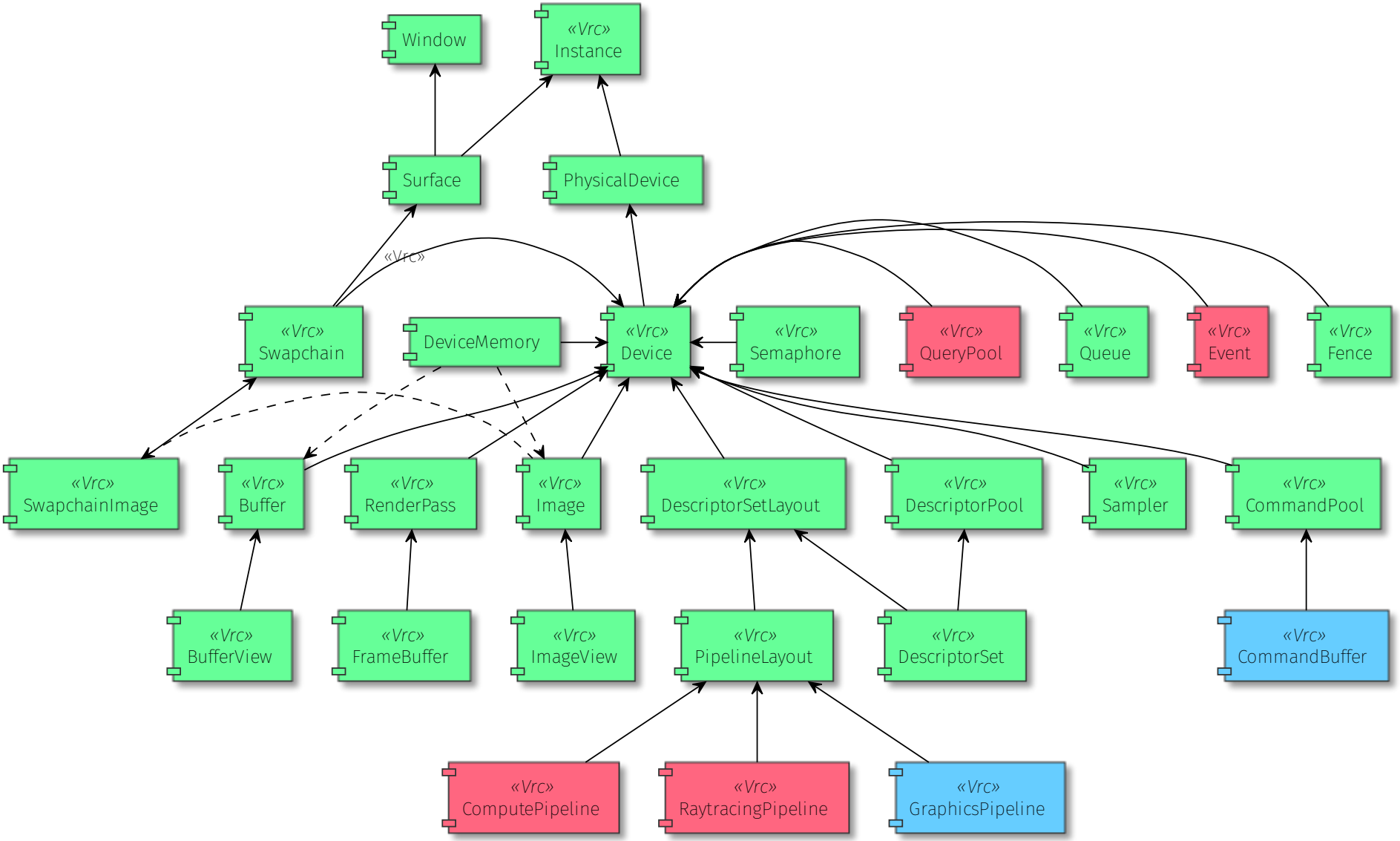


Design and implementation

Project name: Vulkayes

- **Design**
  - Transparent
    - Easy to fall down to low-level
  - Minimal overhead
  - Statical safety
    - Rust references are always valid (raw pointers are rare)
    - Non-zero number types
    - Tagged union types
- **Implementation**
  - Cargo features
    - Condition compilation
    - Multi-threaded feature
  - Vrc, Deref, generics
    - Vrc type alias allows seamless feature switching
    - Deref is already familiar to Rust developers
  - Flexibility

Design and implementation

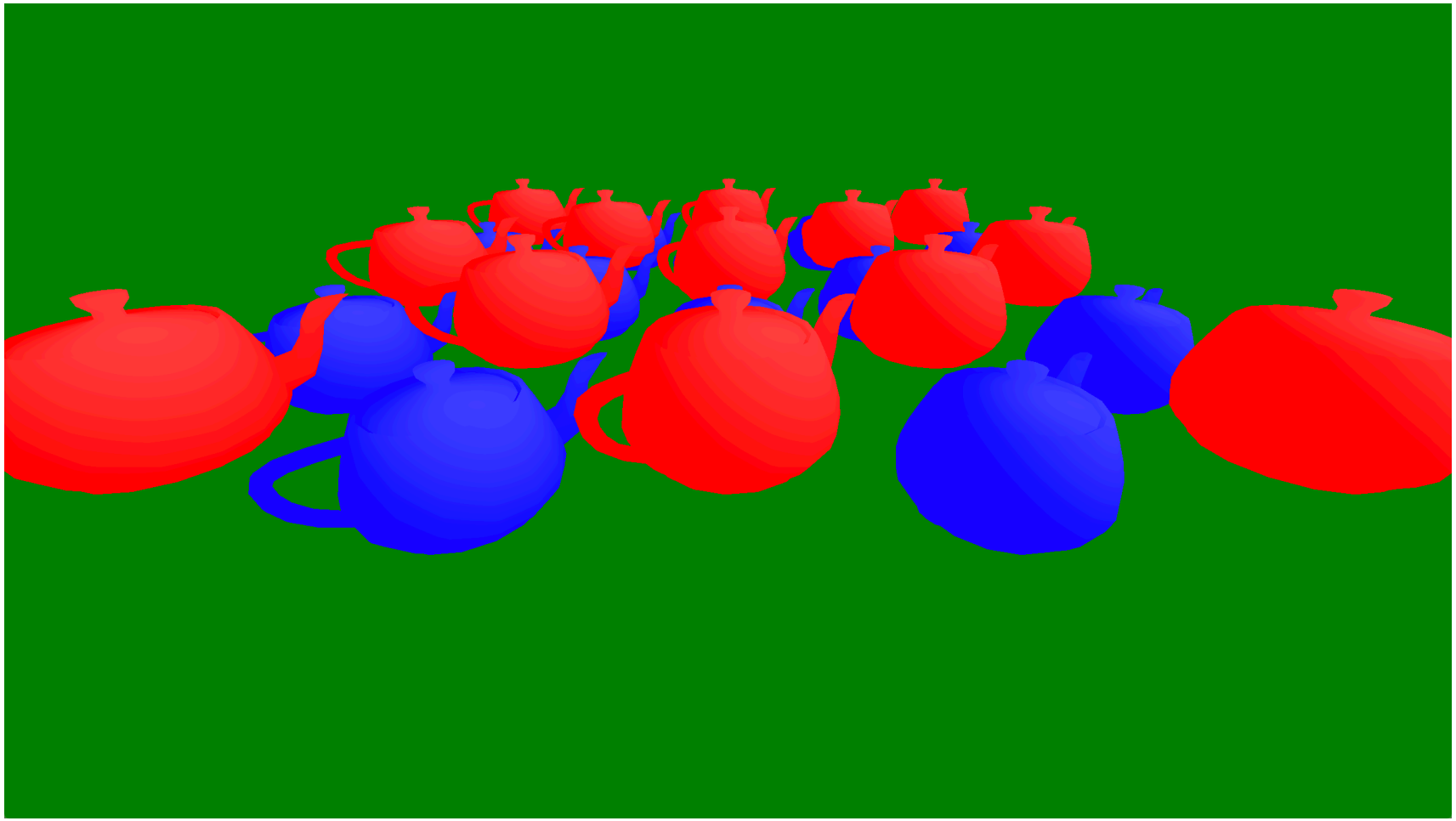


Vulkayes Object Dependency Graph, most Vulkan object are wrapped

Results and evaluation

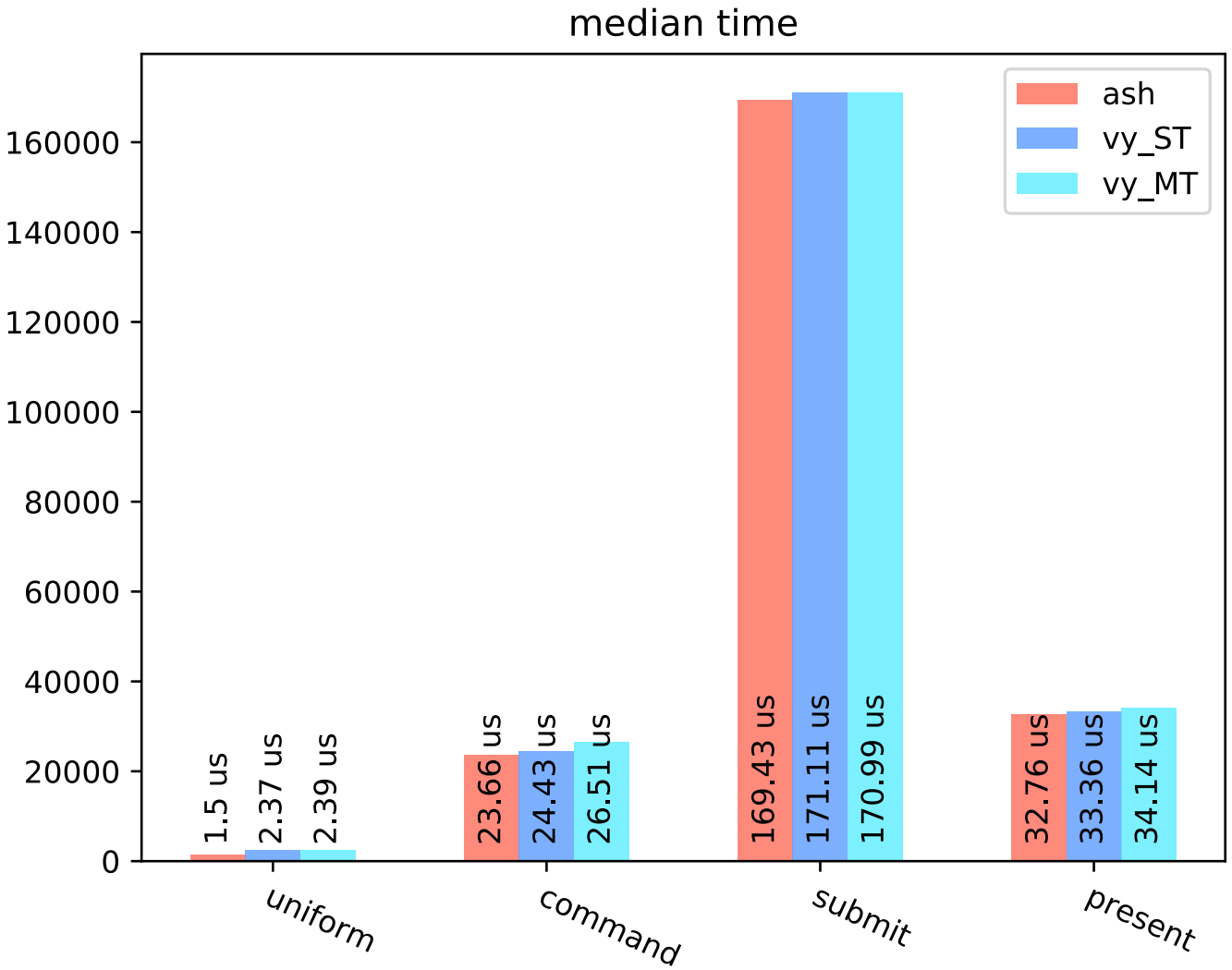
- Developer experience
  - In selected code snipptes, Vulkayes code is three times shorter than equivalent ash code
  - The benchmarking program code in Vulkayes is 33% shorter than in ash
- Performance
  - Vulkayes was evaluated against ash, the bindings to Vulkan API that are used, as a baseline
  - `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



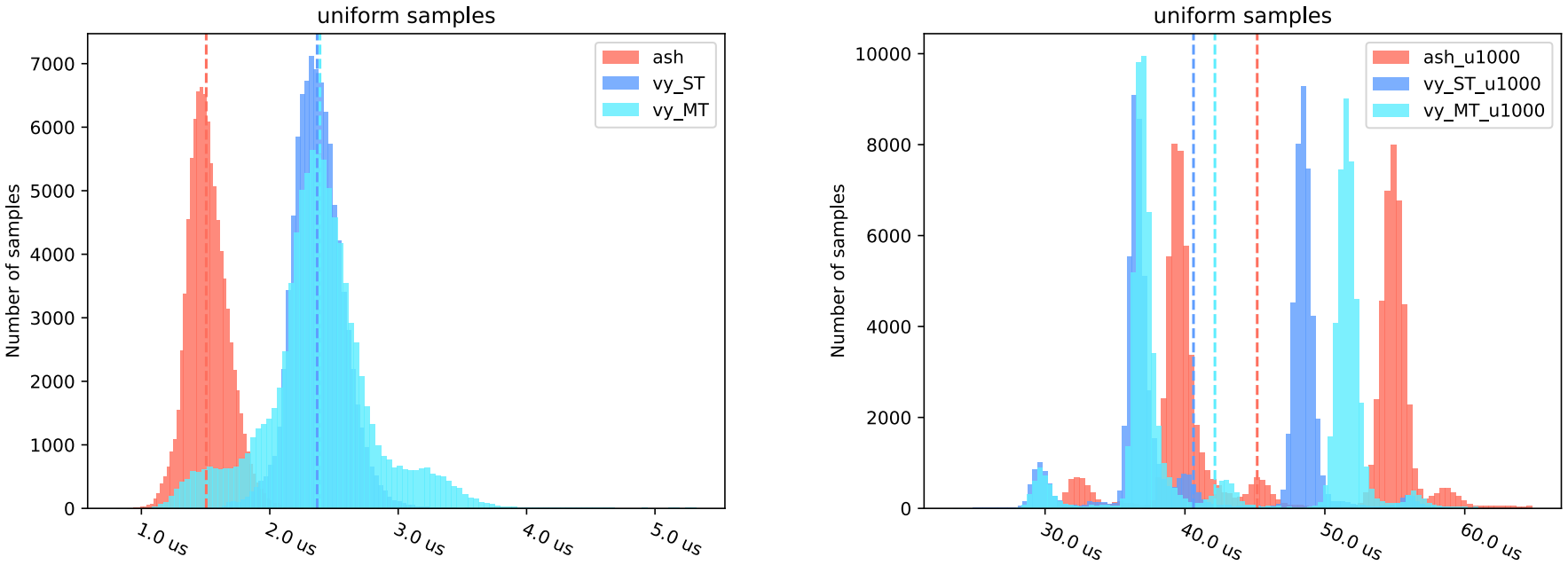
Benchmarking program output

Results and evaluation



Average median time (n = 99000)

Results and evaluation



Histogram of uniform stage of the benchmarks (n = 99000). On right: 1000 writes instead of one.

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
- Open-source, licensed under either MIT or Apache 2.0 at https://github.com/vulkayes

Unanswered questions

- What do you see as the biggest advantage of your system?
  - Transparency and developer experience
    - Fast prototyping
    - Shorter and safer code
    - Easy to work around missing parts
    - Allows very selective benchmarking

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?
  - Experience with Vulkano synchronization before starting Vulkayes
  - Tephra does something similar
  - Initial attempt based on this, but full control is too restrictive
  - Can we do better?

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?
  - Core only exposes interface for allocations
    - Integration of specific allocators is out of scope
  - However, Vulkayes provides `naive_memory_allocator` Cargo feature
  - Plan to support VMA explicitly in a separate crate (similar to `vulkayes-window`)