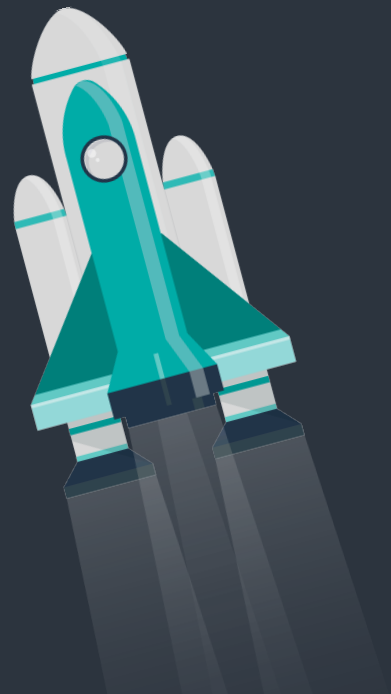
Build a Ray Tracer with Rust and wgpu

Niklas Korz

1. Introduction
2. Basics of wgpu
3. Tracing rays
4. Bringing it to the web
5. Wrap-up



About Me

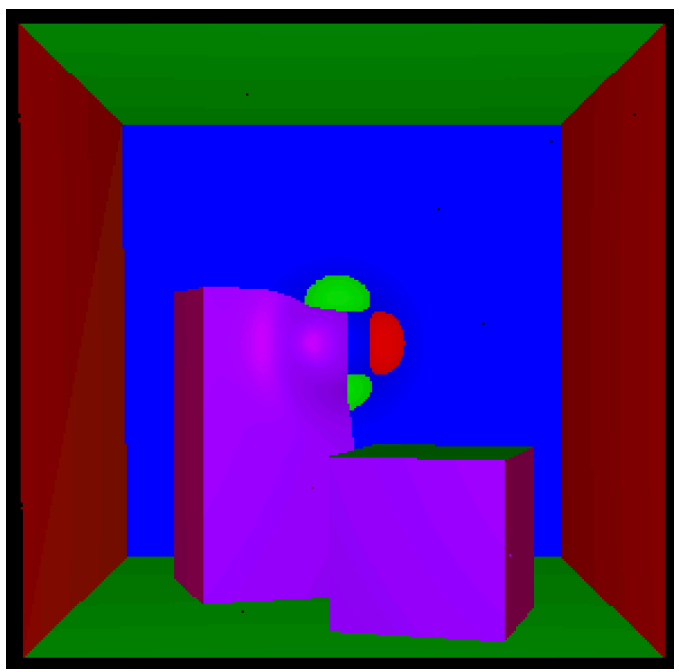
- Co-founder and tech lead at **alugha.com**
- Meetup organizer: “Nix Your Bugs & Rust Your Engines”
- Likes:
  - computer graphics
  - interactive media
  - pen & paper roleplaying games
- Website: **https://korz.dev**
- Mastodon: **@niklaskorz@rheinneckar.social**

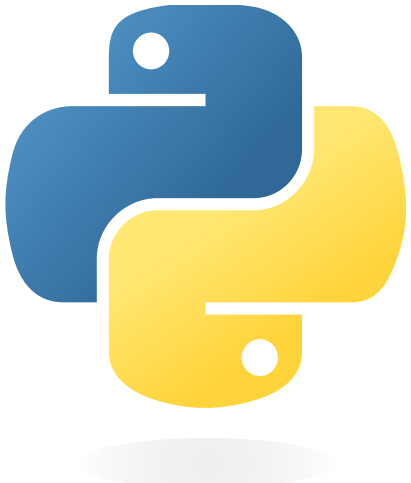
Goals for today

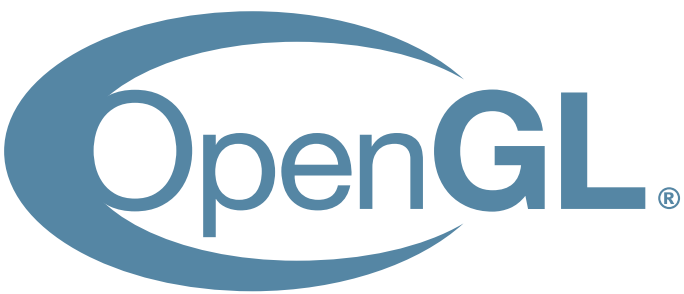
- Introduction to wgpu APIs
- Shader programming with WGSL
- Basic knowledge about ray tracing
- *Foundation* for implementing your own ray tracer
  - 3h isn’t much time
  - let’s see how far we come

Workshop background

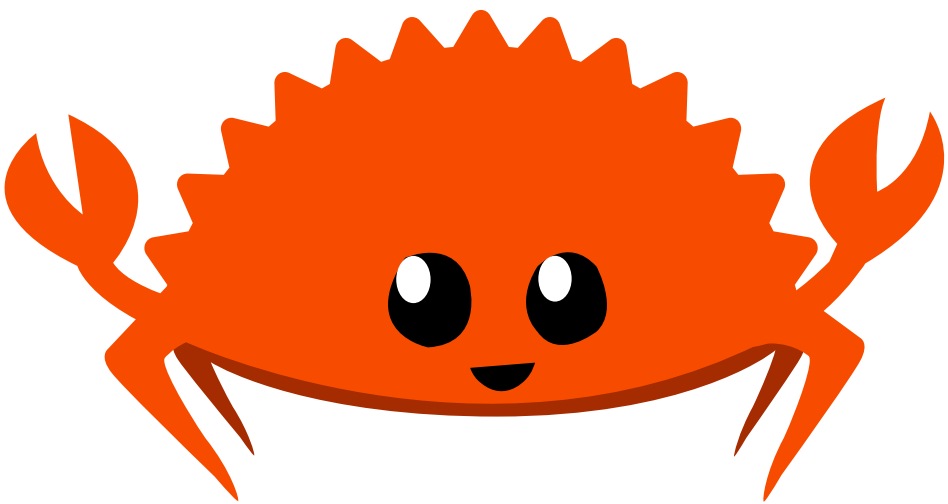
- Talk at RustFest Zürich: “Interactive Exploration of Nonlinear Ray Casting with Rust and wgpu”
- Research project during Master of Science studies at Heidelberg University
- Continuous refraction of light





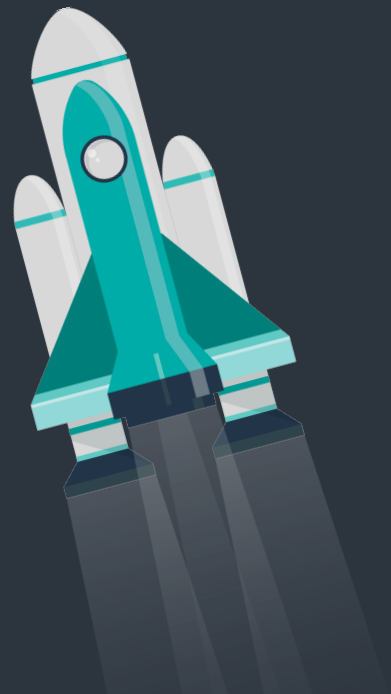

OpenGL®


Vulkan®

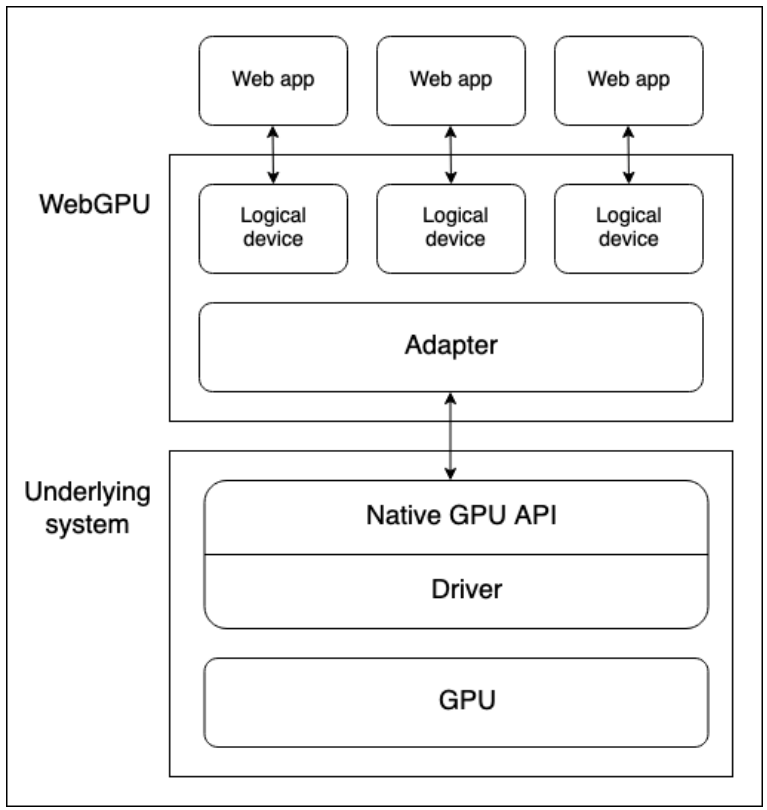



gpu

1. Introduction
2. **Basics of wgpu**
3. Tracing rays
4. Bringing it to the web
5. Wrap-up



WebGPU



by Mozilla Contributors (CC-BY-SA)

- New graphics standard by W3C
- Successor to WebGL
- Based on family of modern graphics APIs
  - Explicit management of state and resources
  - No global state machine
  - But: provides abstractions for safer and easier usage
- Supports compute shaders!

OpenGL/WebGL

- Global state machine
- C-like shading language (GLSL)
- No compute shaders in WebGL
- Wide compatibility (**ANGLE**)
  - OpenGL ES on top of DirectX 11
  - …and Vulkan, Metal

WebGPU

- Explicit resource management
- Shading language mix of TS / Rust
- Compute shaders (on the web!)
- Limited hardware / browser support
  - old Intel GPUs *were* problematic
  - compatibility improved with DX12
  - fallback backend using OpenGL

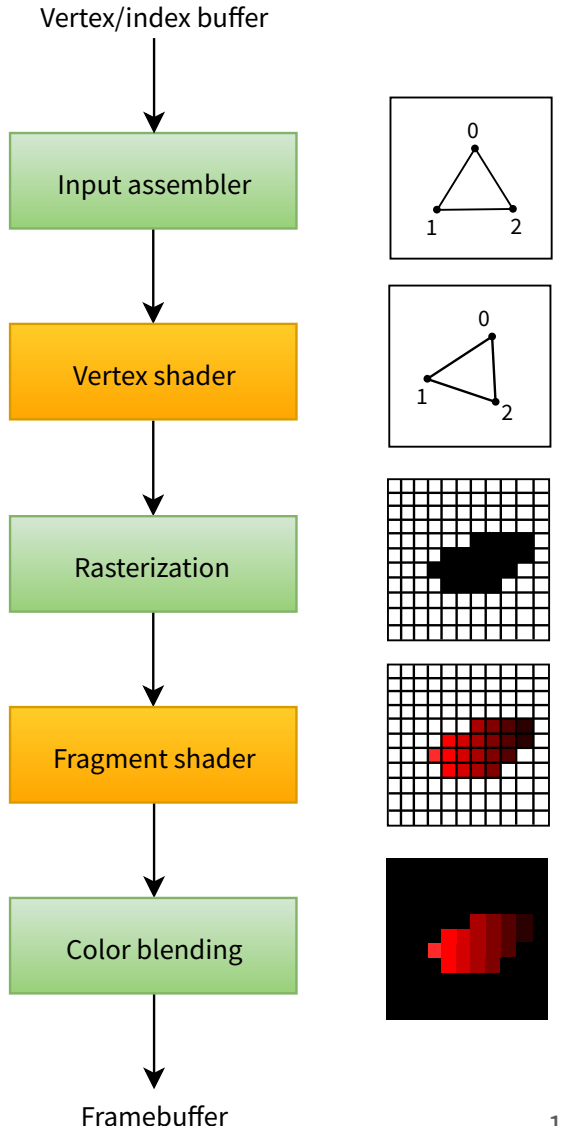
WebGPU Shading Language

```
@vertex
fn vert_main() -> @builtin(position) vec4<f32> {
  return vec4<f32>(0.0, 0.0, 0.0, 1.0);
}

@fragment
fn frag_main(
  @builtin(position) coord_in: vec4<f32>
) -> @location(0) vec4<f32> {
  return vec4<f32>(
    coord_in.x, coord_in.y, 0.0, 1.0);
}
```

- New textual language
  - Considers target limitations
  - Focus on validation
- Multiple targets
  - SPIR-V for Vulkan
  - HLSL for DirectX 12
  - MSL for Metal

The Graphics Pipeline

- Inputs for GPU:
  - vertex array (coordinates)
  - index array
  - anything else (uniforms)
- Some stages are programmable
  - vertex shader
  - fragment shader
  - compute shader



Graphics Pipeline
by Alexander Overvoorde
(CC-BY-SA)

Explicit Management of State and Resources

- API context: `wgpu::Instance`
  - configures and checks native graphics API backends
- Physical GPU: `wgpu::Adapter`
  - must be able to target display of our window
- Logical GPU: `wgpu::Device`
  - enforces feature restrictions, memory limits, …
- Presentation target: `wgpu::Surface`
  - can be a native window or an HTML canvas

Explicit Management of State and Resources

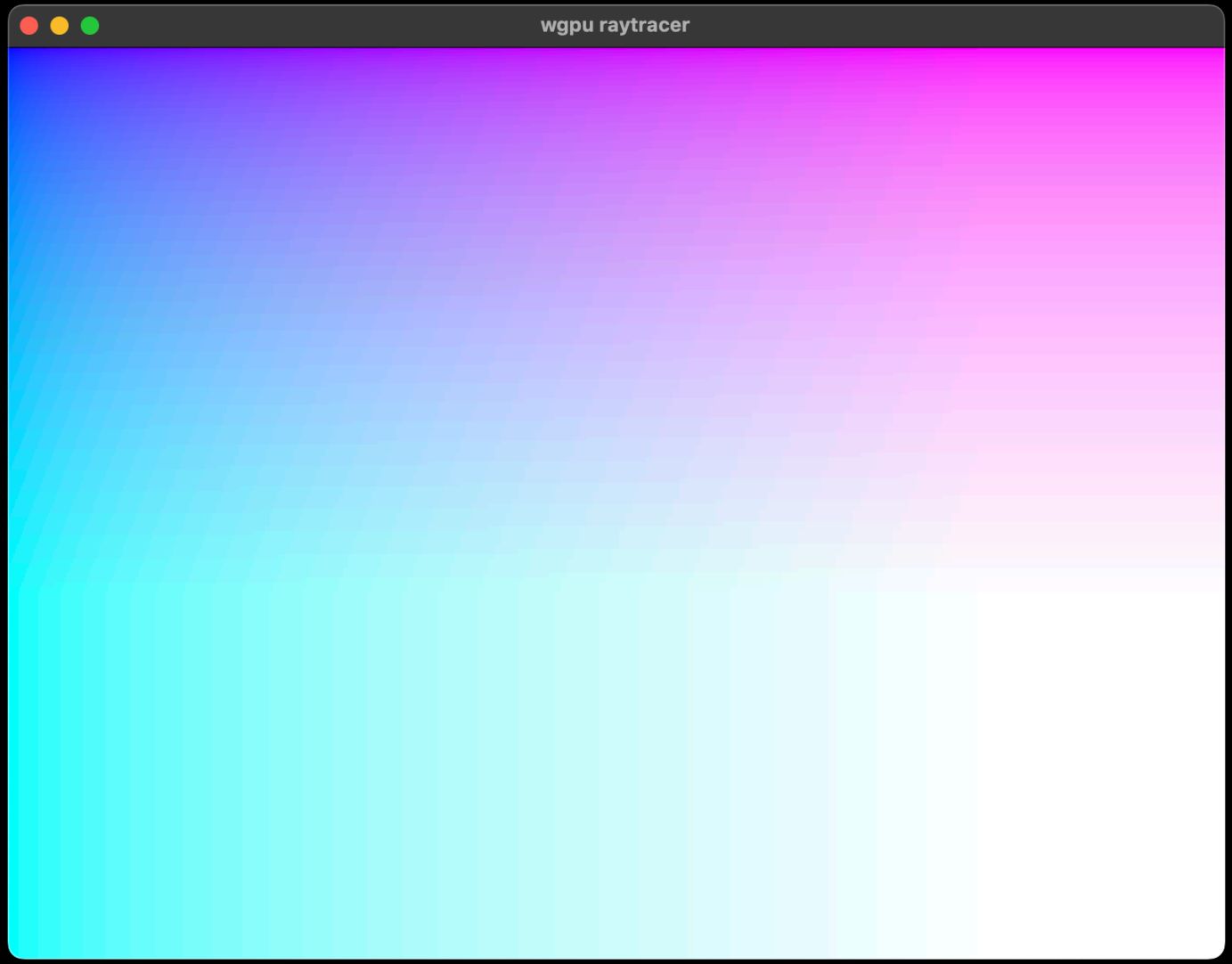
- Pipeline configuration: `wgpu::RenderPipelineDescriptor`
  - How are vertices processed?
  - How are primitives assembled?
  - How are fragments (pixels) blended?
- Encoding commands for the GPU:
  - `wgpu::CommandEncoder` : bring commands into GPU-digestible format
  - `wgpu::RenderPass` : describe what will be drawn in this frame

IDE support

- Visual Studio Code:

  https://github.com/PolyMeilex/vscode-wgsl
- JetBrains / RustRover:

  https://plugins.jetbrains.com/plugin/18110-wgsl-support
- Some more or less working plugins for Neovim and Zed


wgpu raytracer

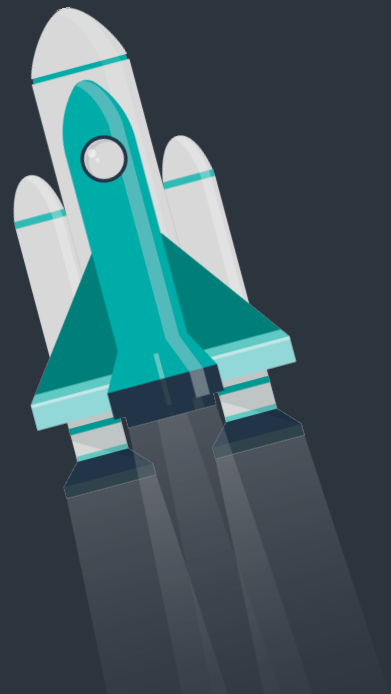
Exercises

1. Clone repository: https://github.com/niklaskorz/rustlab2024-wgpu
2. Ensure the project builds and runs on your machine: `cargo run`
3. Setup your graphics pipeline: instructions in `src/application.rs`
4. Bring colors to your window: instructions in `src/application.wgsl`
5. Experimentation time: what else can you draw?
   - Built-in WGSL functions: https://webgpufundamentals.org/webgpu/lessons/webgpu-wgsl-function-reference.html

Showtime

- Share what you made!
- Explain how your solution works
- Any problems you encountered along the way?

1. Introduction
2. Basics of wgpu
3. Tracing rays
4. Bringing it to the web
5. Wrap-up

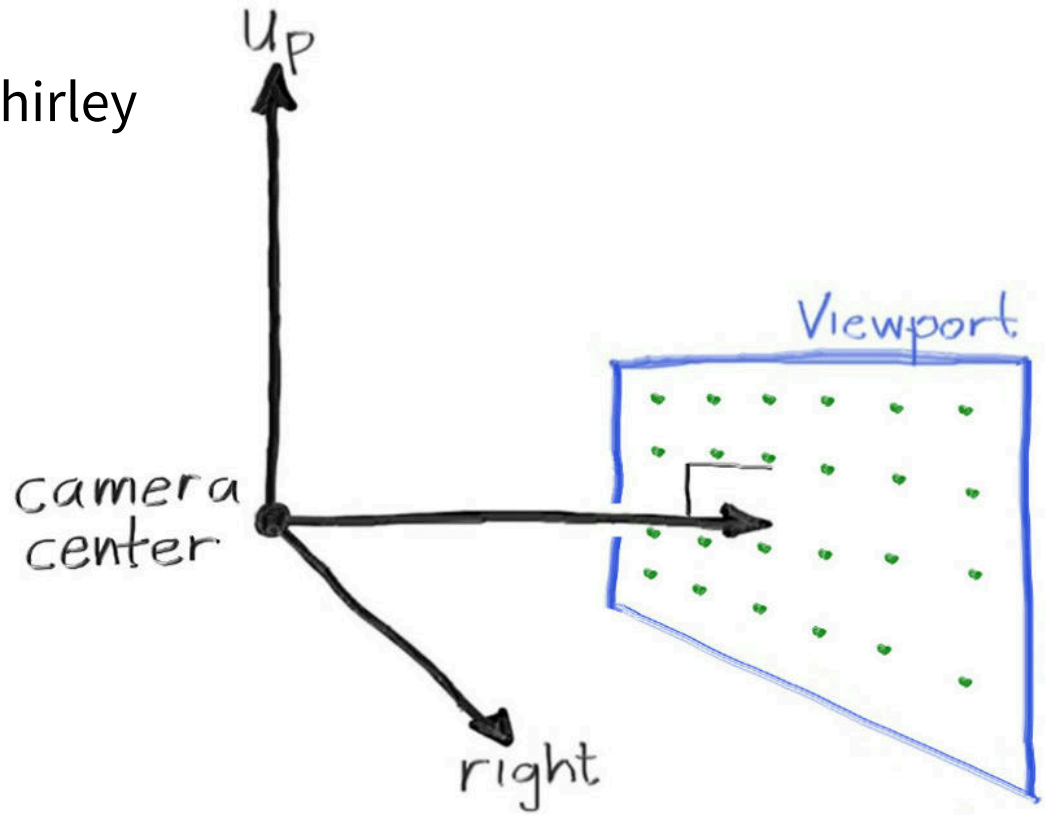


What is Ray Tracing?

- Simulates how light interacts with objects in a scene
- Rays originate from an imaginary camera through a viewport
- Can include more complex phenomena
- Ray Casting: without reflections, refractions, …

What is Ray Tracing?

Illustration by P. Shirley



Ray-Sphere intersection

$$x^2 + y^2 + z^2 = r^2$$

$$(\boldsymbol{C} - \boldsymbol{P}(t)) \cdot (\boldsymbol{C} - \boldsymbol{P}(t)) = r^2$$

$$\boldsymbol{P}(t) = \boldsymbol{Q} + t * \boldsymbol{d}$$

Solve for t:

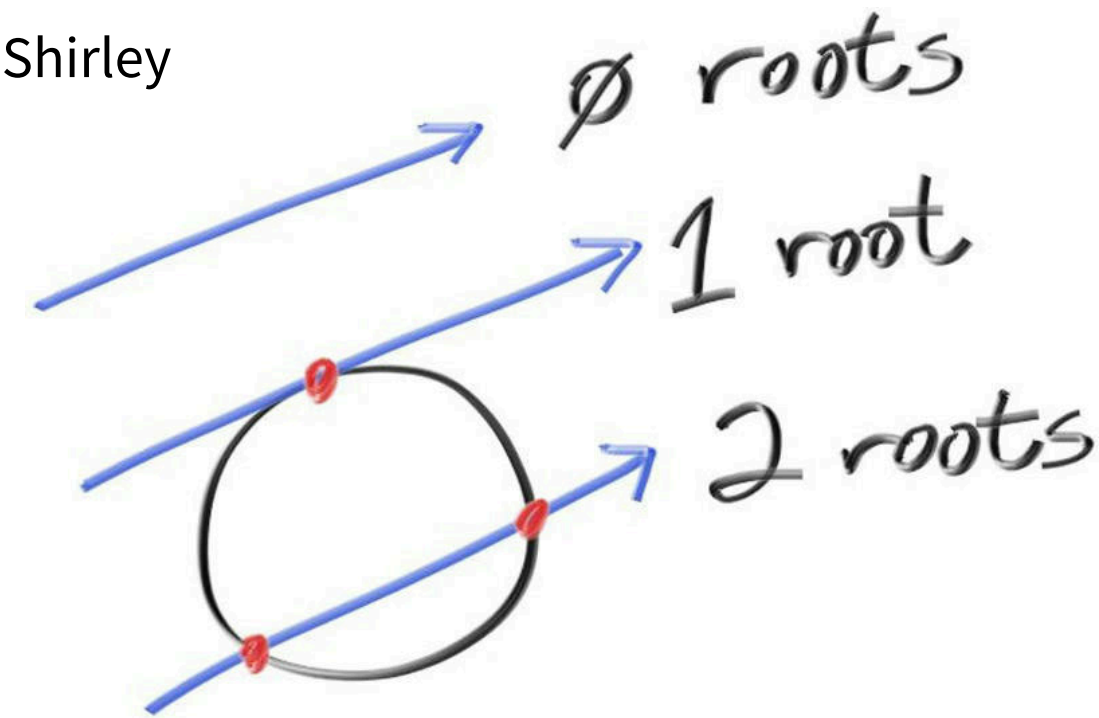
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where

$$a = \boldsymbol{d} \cdot \boldsymbol{d}, b = -2\boldsymbol{d} \cdot (\boldsymbol{Q} - \boldsymbol{C}), c = (\boldsymbol{C} - \boldsymbol{Q}) \cdot (\boldsymbol{C} - \boldsymbol{Q}) - r^2$$

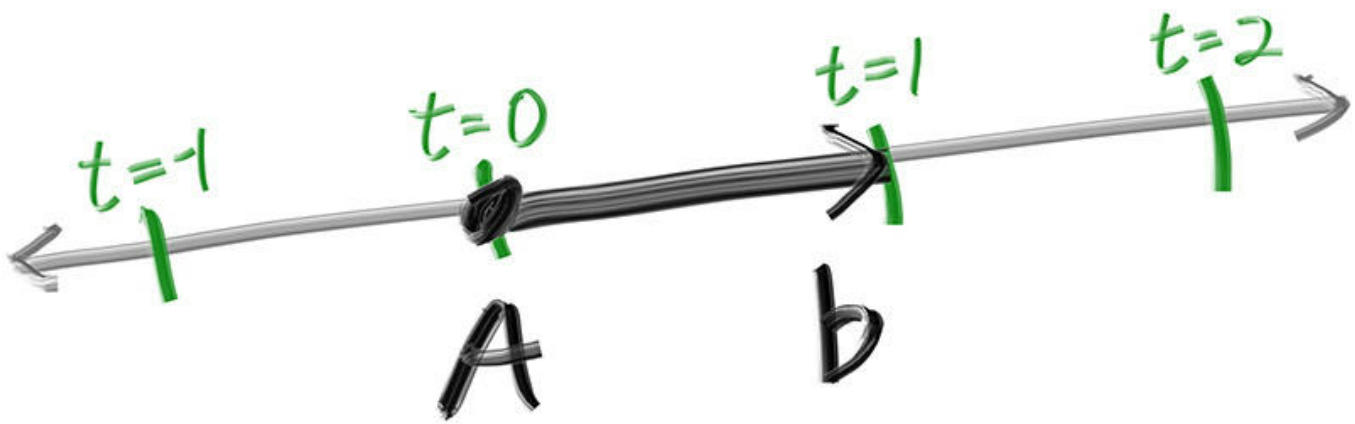
Ray-Sphere intersection

Illustration by P. Shirley



Ray-Sphere intersection

Illustration by P. Shirley



Computer shaders

- Compute shaders can perform non-rendering work
- May run longer than fragment shaders
- Work execution on GPU: workgroups (almost like threads)
- Inside a workgroup, memory can be shared

Computer shaders

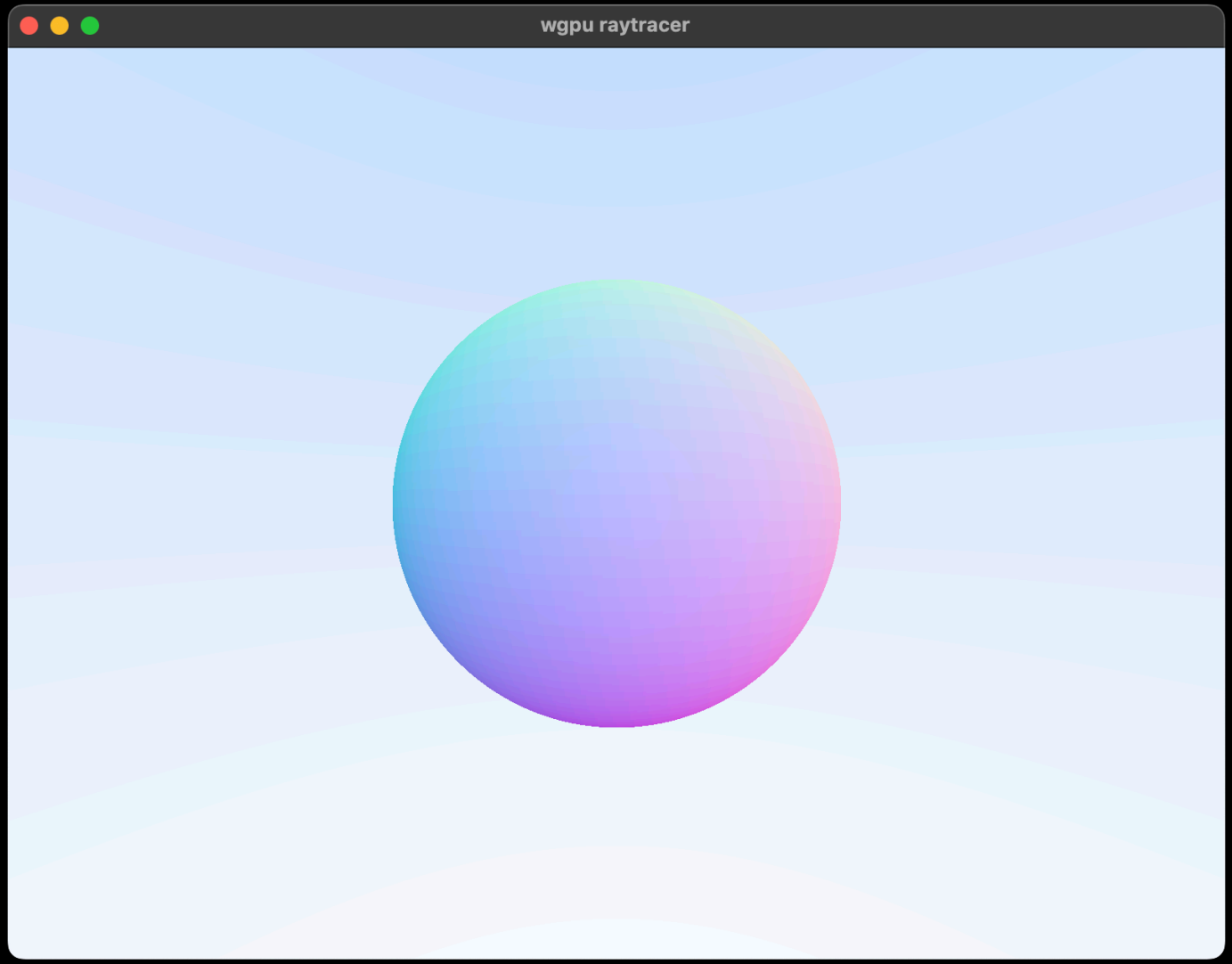
- How do we display the compute shader's output?
  - Proxy geometry
  - Fragment shader does nothing but read and display output texture
- Texture access from both shaders through bindings
  - Bind groups combine multiple bindings
  - Pipeline has to know about bindings: bind group layouts

Arcball camera

- Camera control that works similar to a snow dome or globe
- Paper by Ken Shoemake (1992)
- Preimplemented for exercises

Uniform buffers

- Sharing data between CPU and GPU
- Globally available inside your shader (through bindings)
- Faster than storage buffers, but read-only inside shader


wgpu raytracer

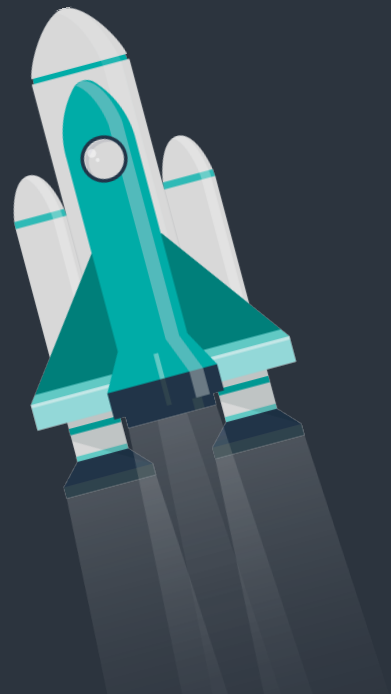
Exercises

1. Commit your solutions to not lose them and checkout branch `ch2`
2. Follow the tasks in ch2's README.md
3. Bonus exercises at the end of README.md

Showtime

- Share what you made!
- Explain how your solution works
- Any problems you encountered along the way?

1. Introduction
2. Basics of wgpu
3. Tracing rays
4. **Bringing it to the web**
5. Wrap-up



WebAssembly

- Portable binary code, usually JIT-compiled
- Stable call interface (more **powerful ABIs** are proposed)
- Sandboxed execution
- Cross-language
  - but you have to bring your own allocator

    (**unless you're using garbage collection**)

WebGPU implementation status

- https://github.com/gpuweb/gpuweb/wiki/Implementation-Status
- Chrome: enabled on Windows and macOS as of Chrome 113
- Firefox: enabled in Firefox Nightly
- Safari: enabled in Safari Technology Preview
  https://webkit.org/blog/14879/webgpu-now-available-for-testing-in-safari-technology-preview/

Compiling Rust to WebAssembly

```
rustup target add wasm32-unknown-unknown

cargo install -f wasm-bindgen-cli --version 0.2.95

RUSTFLAGS=--cfg=web_sys_unstable_apis \
  cargo build --target wasm32-unknown-unknown --release

wasm-bindgen --out-dir public \
  --web target/wasm32-unknown-unknown/release/rustlab2024-wgpu.wasm
```

Running our Rust web app

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <script type="module">
      import init from "./rustlab2024.js";
      init();
    </script>
  </body>
</html>
```

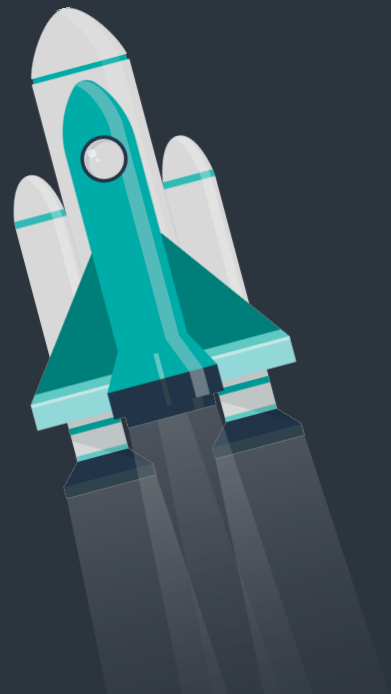
Running our Rust web app

```
cargo install simple-http-server

simple-http-server ./public

# => Running on http://localhost:8000
```

Exercises

1. Install the wasm32-unknown-unknown target for your Rust toolchain
2. Install wasm-bindgen-cli 0.2.95
3. Build the WASM binary
4. Run wasm-bindgen to generate the boilerplate
5. Run a local web server on the public/ directory
6. Try out your application in Chrome, Firefox Nightly and/or Safari TP

1. Introduction
2. Basics of wgpu
3. Tracing rays
4. Bringing it to the web
5. **Wrap-up**



Further Reading

- https://raytracing.github.io/ (also used by this workshop)
- https://nelari.us/post/weekend_raytracing_with_wgpu_1/
- https://www.scratchapixel.com/
- https://sotrh.github.io/learn-wgpu/
- https://webgpufundamentals.org/
- https://google.github.io/tour-of-wgsl/
- https://graphicscompendium.com

Questions? Feedback?

- Download: **https://dl.korz.dev/rustlab2024.pdf**
- Talk to me during the conference
- Contact me on…
  - Email: **contact@korz.dev**
  - Mastodon: **@niklaskorz@rheinneckar.social**