

# **Interactive Exploration of Nonlinear Ray Casting with Rust and wgpu**

Niklas Korz

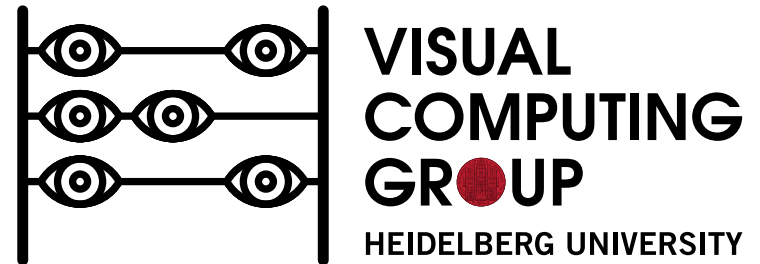
# About Me

- MSc graduate (Heidelberg University)
- Co-founder and tech lead at [alugha.com](https://alugha.com)
- Meetup organizer: “Nix Your Bugs & Rust Your Engines”
- Website: <https://korz.dev>
- Mastodon: [@niklaskorz@rheinneckar.social](https://@niklaskorz@rheinneckar.social)



# Project Origins

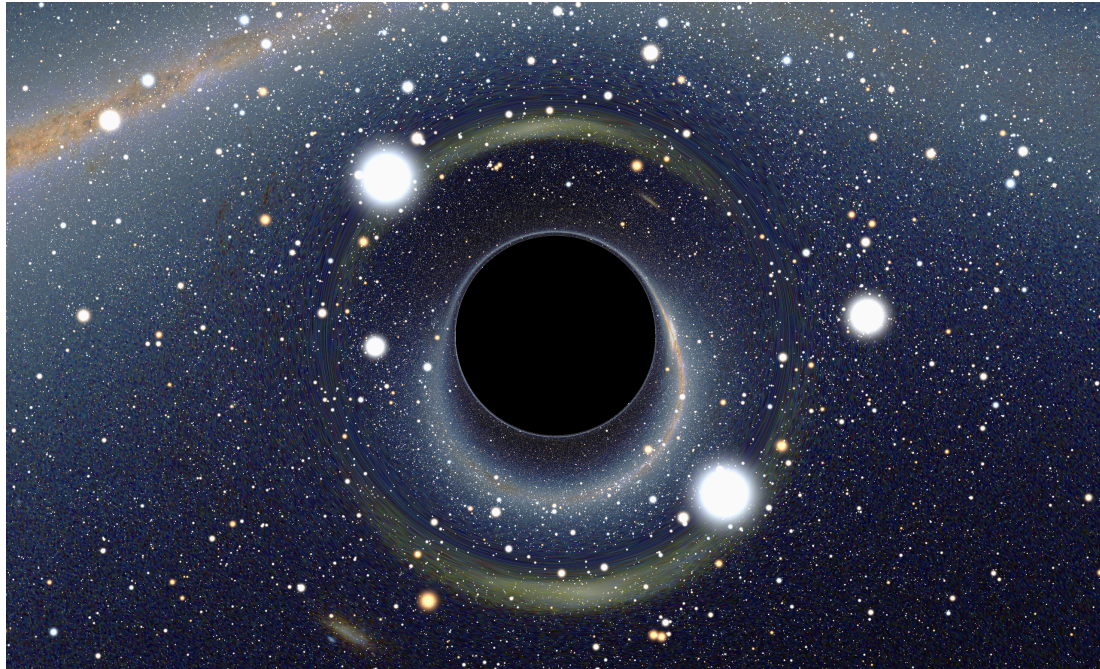
- Software project during Master of Science studies at Heidelberg University
- Duration: April 2021 - November 2021
- Visual Computing Group
- Supervisor Prof. Dr. Filip Sadlo



# Exploring and understanding visual phenomena in nonlinear scenes



Mirage in a desert (public domain)

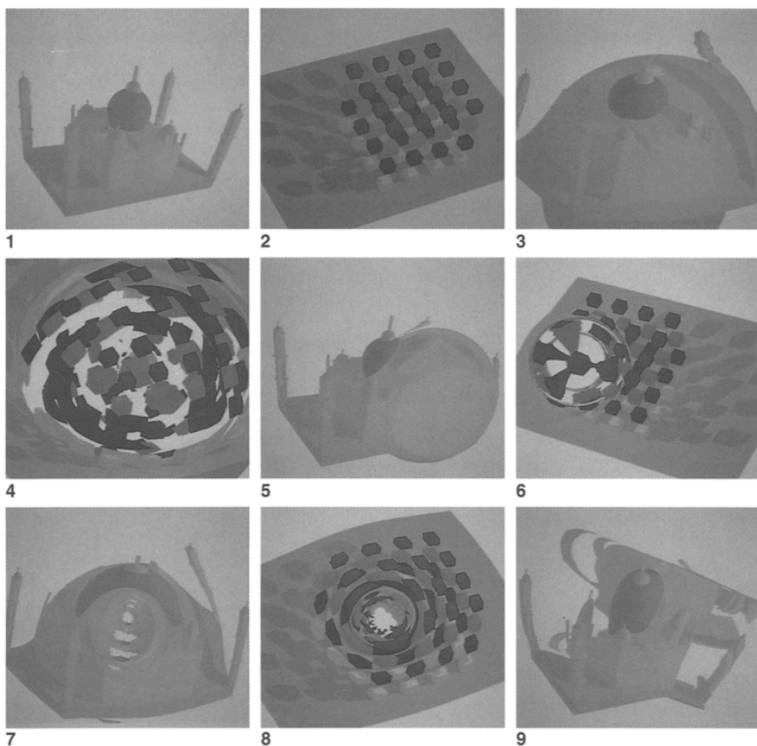


Simulated black hole by Alain Riazuelo (CC-BY-SA)



# E. Gröller. “Nonlinear Ray Tracing: Visualizing Strange Worlds”

The Visual Computer 11:5, 1995, pp. 263–274.

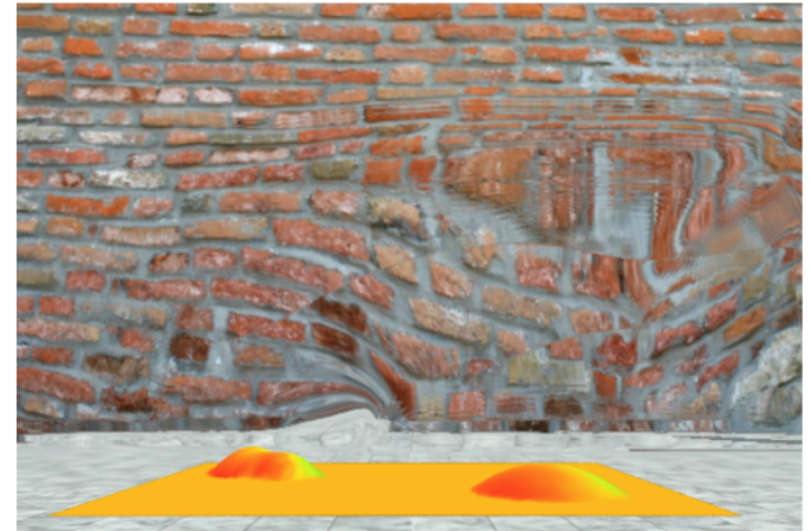


- Rendering of nonlinear scenes
  - Gravitation centers and lines
  - Chaotic dynamic systems
- Algorithms:
  - Uniform subdivision
  - Hierarchical bounding volume structure
- Not interactive

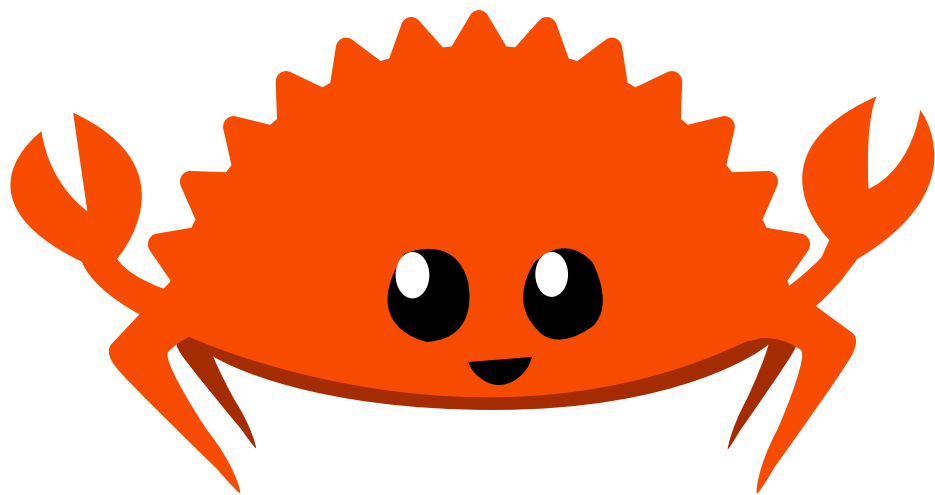
# Y. Zhao et al. “Visual Simulation of Heat Shimmering and Mirage”

IEEE Transactions on Visualization and Computer Graphics 13:1, 2007, pp. 179–189.

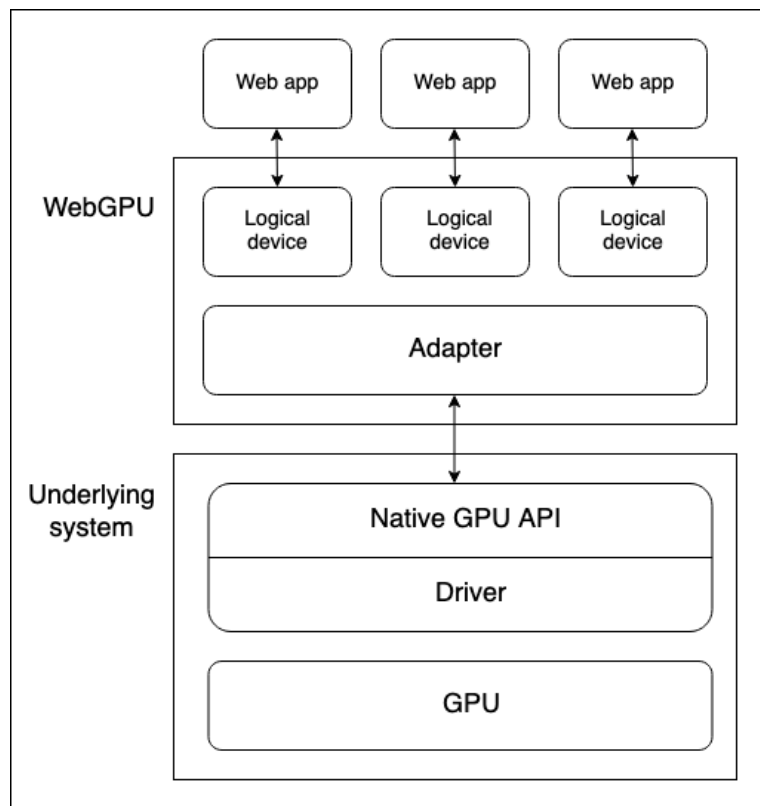
- Physically-based framework for simulating visual effects of heated air
- Rendering through ray tracing on GPU
- Iterations using small step size inside heat volume
- Air refraction computed from Snell’s law
- No alternative visualisation of ray paths



C++? JavaScript?  
OpenGL? Vulkan?



# 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!

# WebGPU Shading Language

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

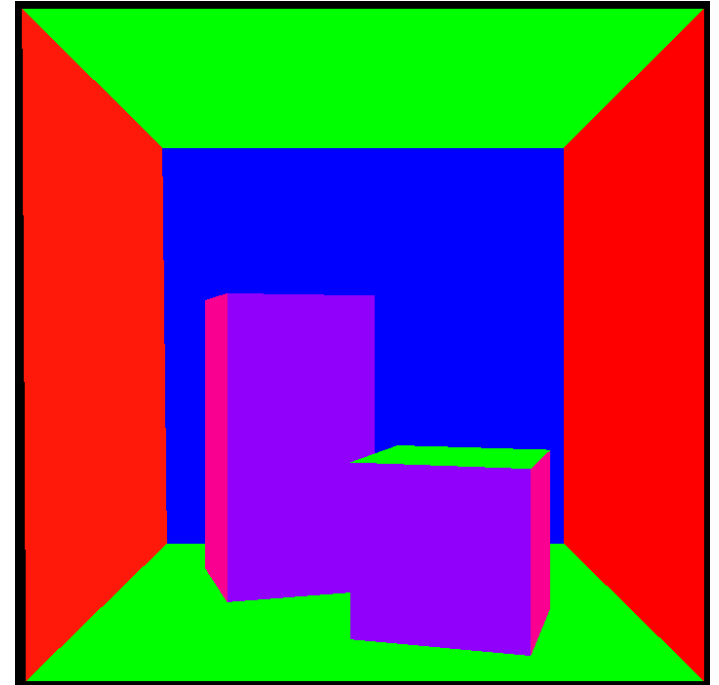
@fragment
fn frag_main(
    @builtin(position) coord_in: vec4<f32>
) -> @location(0) vec4<f32> {
    return vec4<f32>(coord_in.x, coord_in.y,
        0.0, 1.0);
}
```

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



# The Beginning

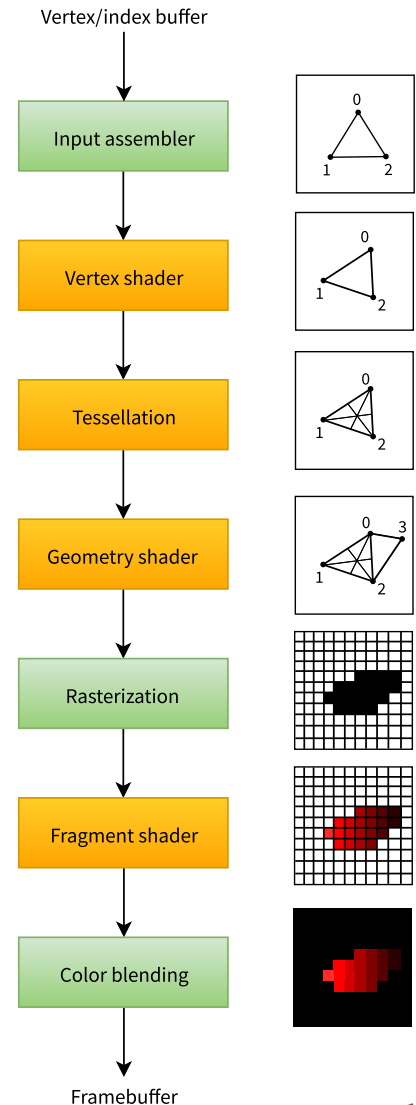
- Simple linear ray caster as foundation
- Execution: WebGPU compute shaders
  - Cast rays from camera through every pixel
  - Colors determined by intersection of rays with objects in the scene
  - Möller-Trumbore intersection algorithm



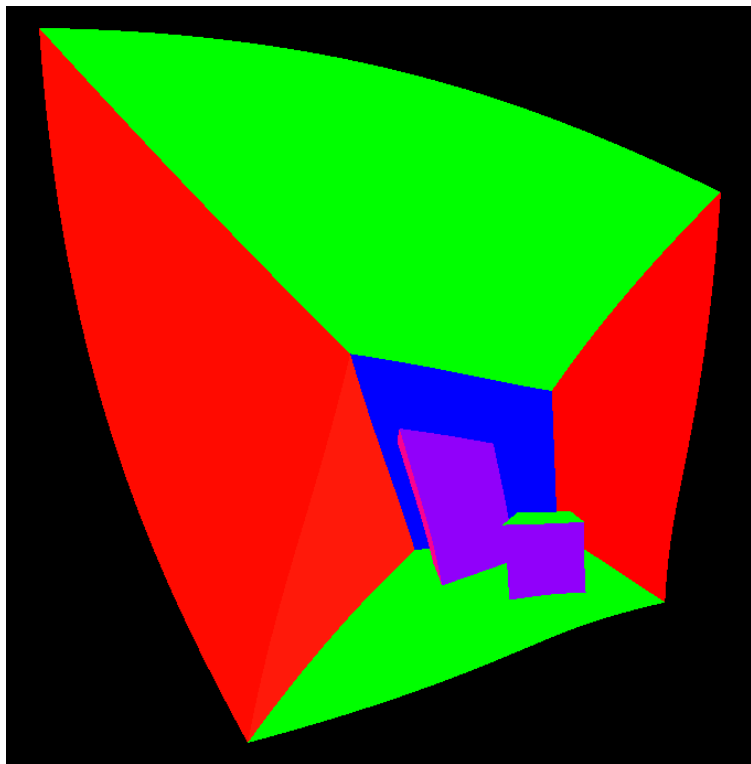
# The Beginning

- Simple linear ray caster as foundation
- Problem: can't write screen buffer from compute shaders
  - Presentation through proxy geometry
  - Vertex shader forwards proxy geometry as-is
  - Fragment shader reads colors from storage texture

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



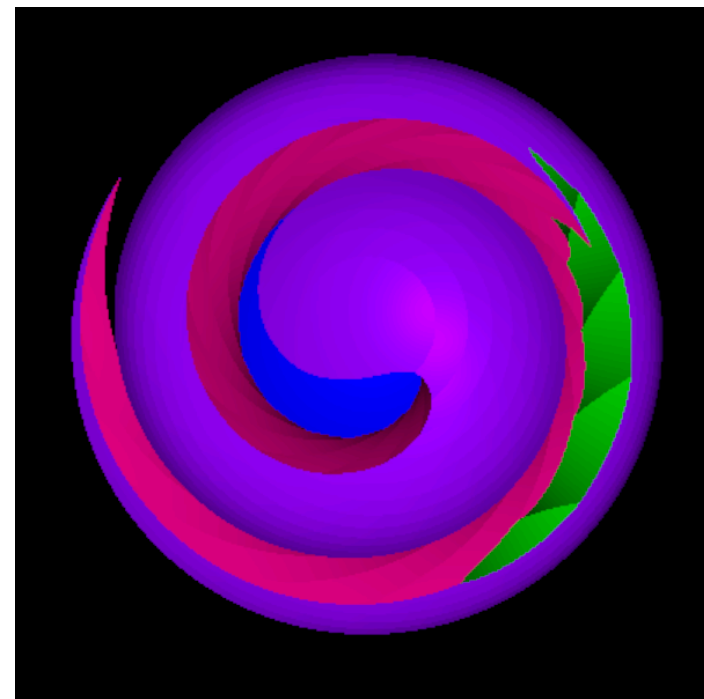
# Nonlinear Rays



- Goal: evaluate path of ray in vector field
- Iterative approach:
  - Subdivision of ray into segments
  - 4th order Runge-Kutta integration inside segment
- Intersection tests between positions
- Early ray termination if intersection found

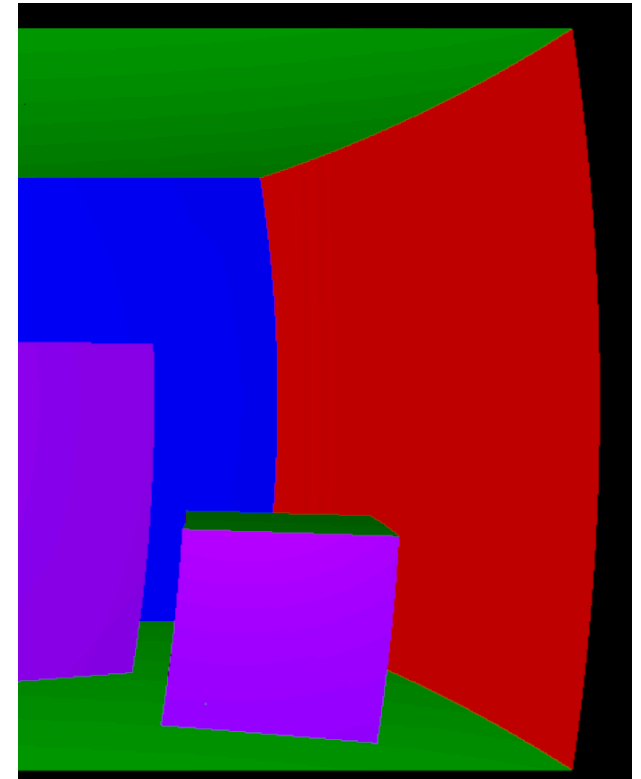
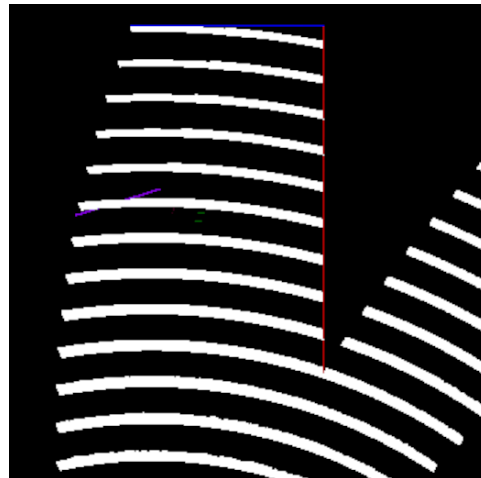
# Lighting

- Blinn-Phong illumination
  - Inaccurate but useful
  - Gives a sense of incoming ray direction
- Camera considered as “light source”
  - Incoming light direction = outgoing direction
  - Simplification for context of nonlinear rays



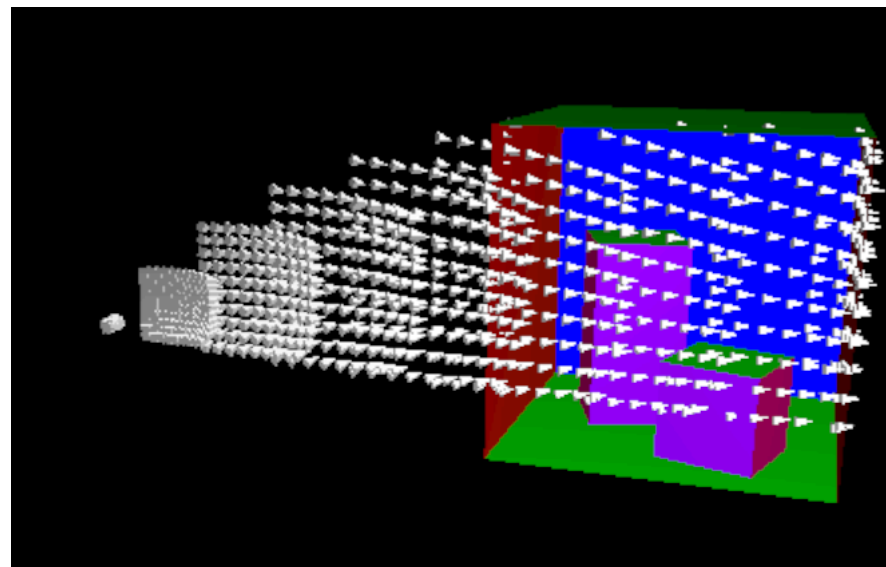
# 2D Wavefronts

- Ray cast view hard to understand by itself
- Reference view supports comprehension
- First idea: 2D wavefronts
- Works if rays mostly stay on same plane
- Unsuitable for other fields



# 3D Wavefronts

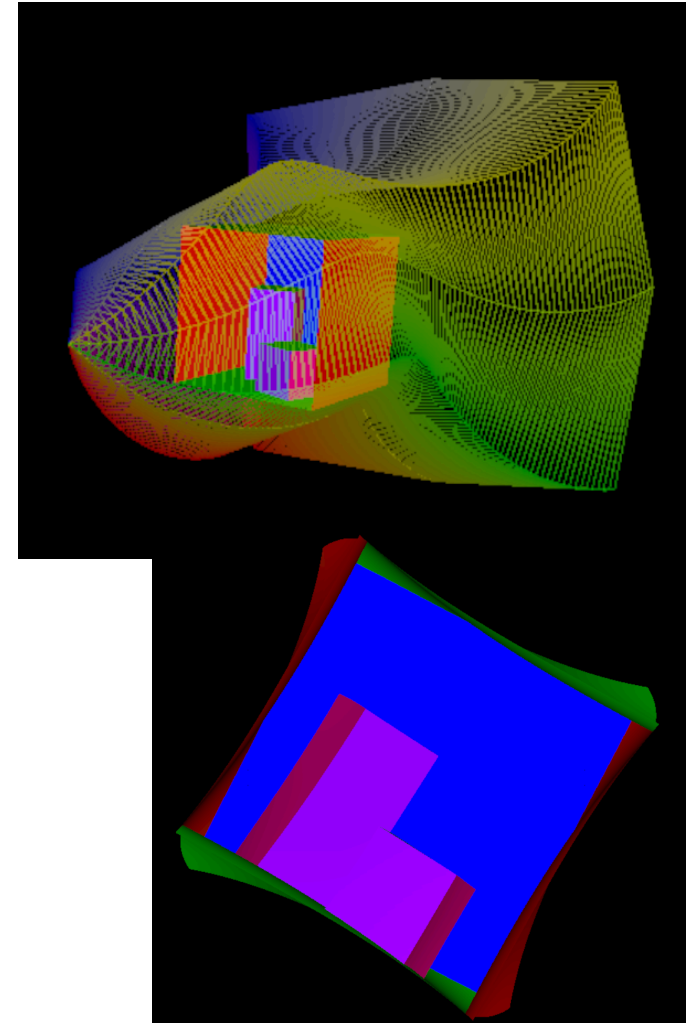
- 3D reference view with camera controls
- Arrow glyphs placed along wavefronts
- Rays sampled in compute shader
- Single mesh drawn with instancing
  - One vertex shader run per sample (position, orientation)
  - Applied as matrix transformation to glyph's vertices
- But: no connection between samples of same ray





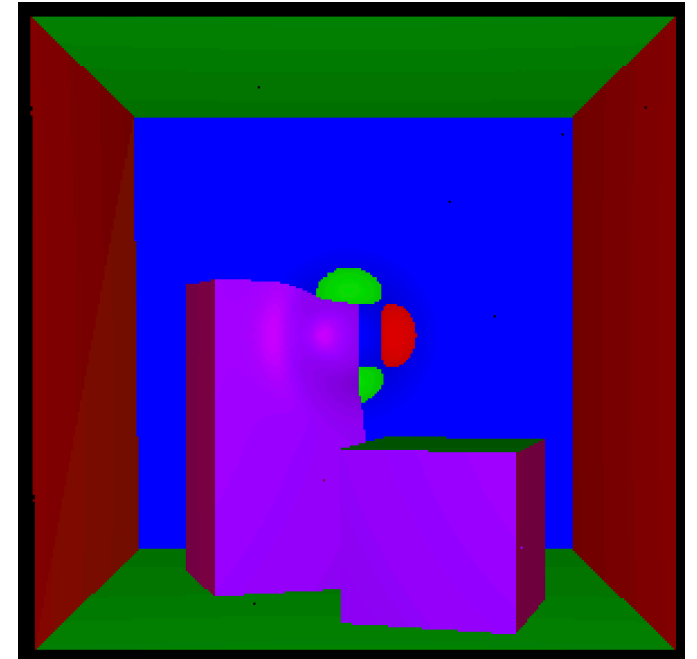
# 3D Outline Mesh

- Multicolored 3D mesh constructed
  - Sampled from eight rays on outline of camera
  - Different color per ray for distinction
- Rendered with indexed drawing
  - Index buffer precomputed once
  - Samples written into fixed-size vertex buffer
- Wireframe mode emphasizes sampled rays



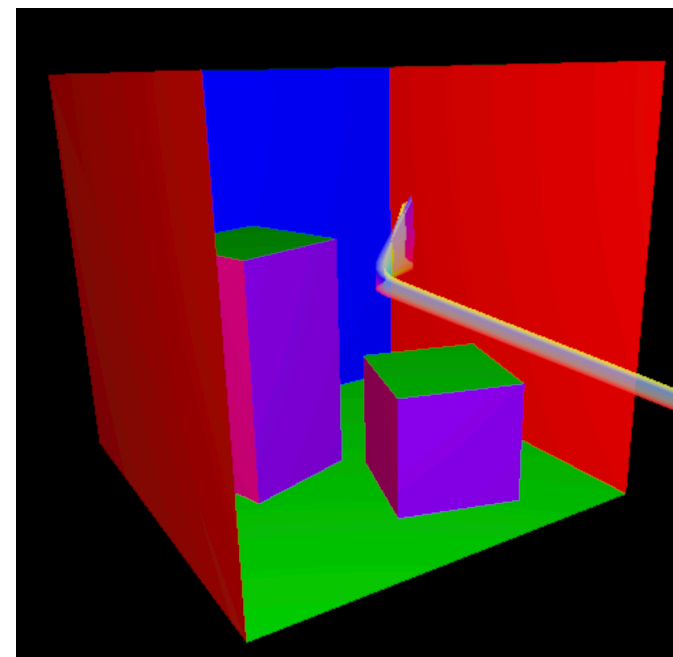
# Simulation of Mirages

- Continuous refraction in a medium (e.g., air)
- Approximated through Runge-Kutta
  - Refraction between previous and current point
  - Refraction index computed from air temperature
- Spherical and plane heat source
- Interpolate core and environmental temperature
  - Linear heat spread (unrealistic but easy)
  - Sigmoid heat spread (smoother)



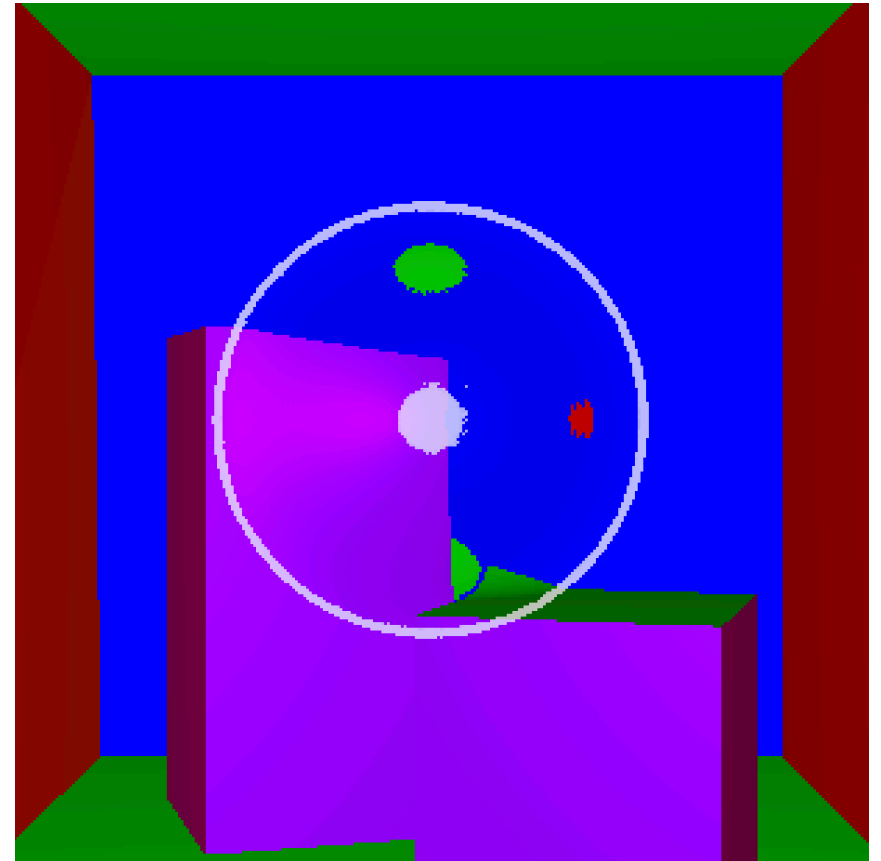
# Ray Neighborhood Outline

- 3D outline mesh may not show interesting parts of field
- User wants to inspect points of interest
- Sample rays around point chosen by user
  - Position passed to compute shader to select rays
  - Reuses outline mesh mechanism
  - Index buffer unchanged
- Shows which part of the scene is seen in mirage
- Emphasizes regional distortions



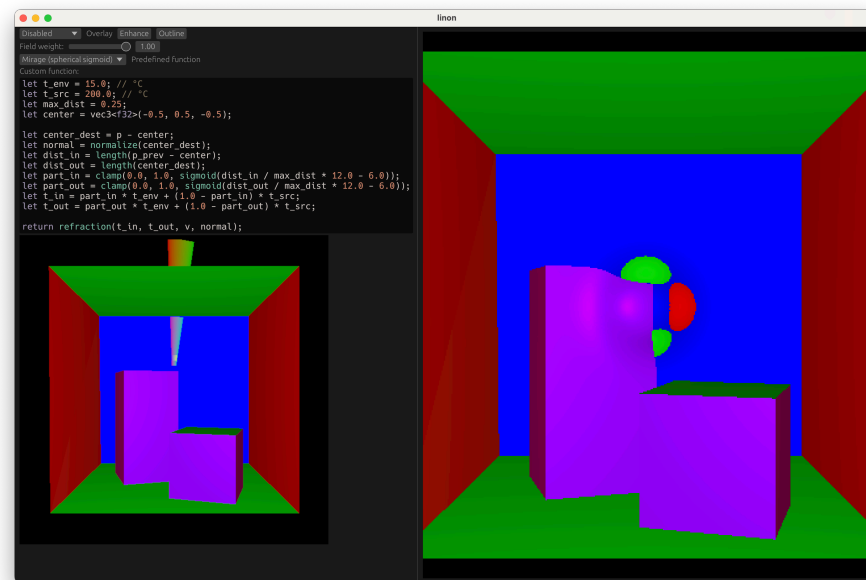
# Lyapunov Exponents Overlay

- Emphasize areas of different behavior
  - Divergence of rays around a point
  - Local gradients from central difference
- Additional compute shader for overlay
  - Ray casting first to get end positions
  - Scaled exponents are drawn on top



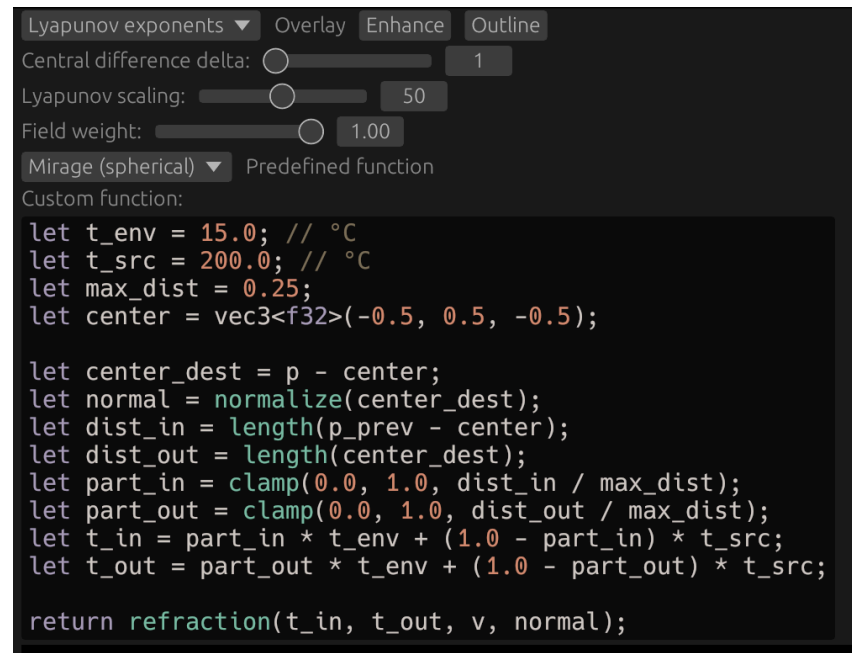
# Final Application

- Rendered linearly in reference view (left)
- Rendered non-linearly in main view (right)
  - Optional overlay for visualization of Lyapunov exponents
  - Mouse click on pixels to visualize ray neighborhood in reference view



# Final Application

- List of editable predefined functions as an entry point
- Functions can incorporate multiple parameters:
  - Previous and current ray position
  - Initial and current ray velocity
  - Time passed since creation of ray



```
let t_env = 15.0; // °C
let t_src = 200.0; // °C
let max_dist = 0.25;
let center = vec3<f32>(-0.5, 0.5, -0.5);

let center_dest = p - center;
let normal = normalize(center_dest);
let dist_in = length(p_prev - center);
let dist_out = length(center_dest);
let part_in = clamp(0.0, 1.0, dist_in / max_dist);
let part_out = clamp(0.0, 1.0, dist_out / max_dist);
let t_in = part_in * t_env + (1.0 - part_in) * t_src;
let t_out = part_out * t_env + (1.0 - part_out) * t_src;

return refraction(t_in, t_out, v, normal);
```



# Room for Improvement

- More complex scenes
  - Textures
- Performance improvements through acceleration structures
- Larger library of predefined functions and helpers as foundation for further experiments

# Questions?

- Repository: <https://github.com/niklaskorz/linon>
- Demo (Firefox Nightly & Chrome): <https://niklaskorz.github.io/linon/>
- Open Source (MIT license)
- Contact me on...
  - Email: [contact@korz.dev](mailto:contact@korz.dev)
  - Mastodon: [@niklaskorz@rheinneckar.social](https://mastodon.social/@niklaskorz)
  - Matrix