The Future of Makie: Raytracing and Beyond

Simon Danisch

Author of Makie

Image: render of Breeze.jl BOMEX data with RayMakie

Why Raytracing in Makie?

Raytracing unlocks better 3D scientific visualization, all within the simple, familiar Makie API.

Global Illumination

Realistic light bouncing.

Volumetric Rendering

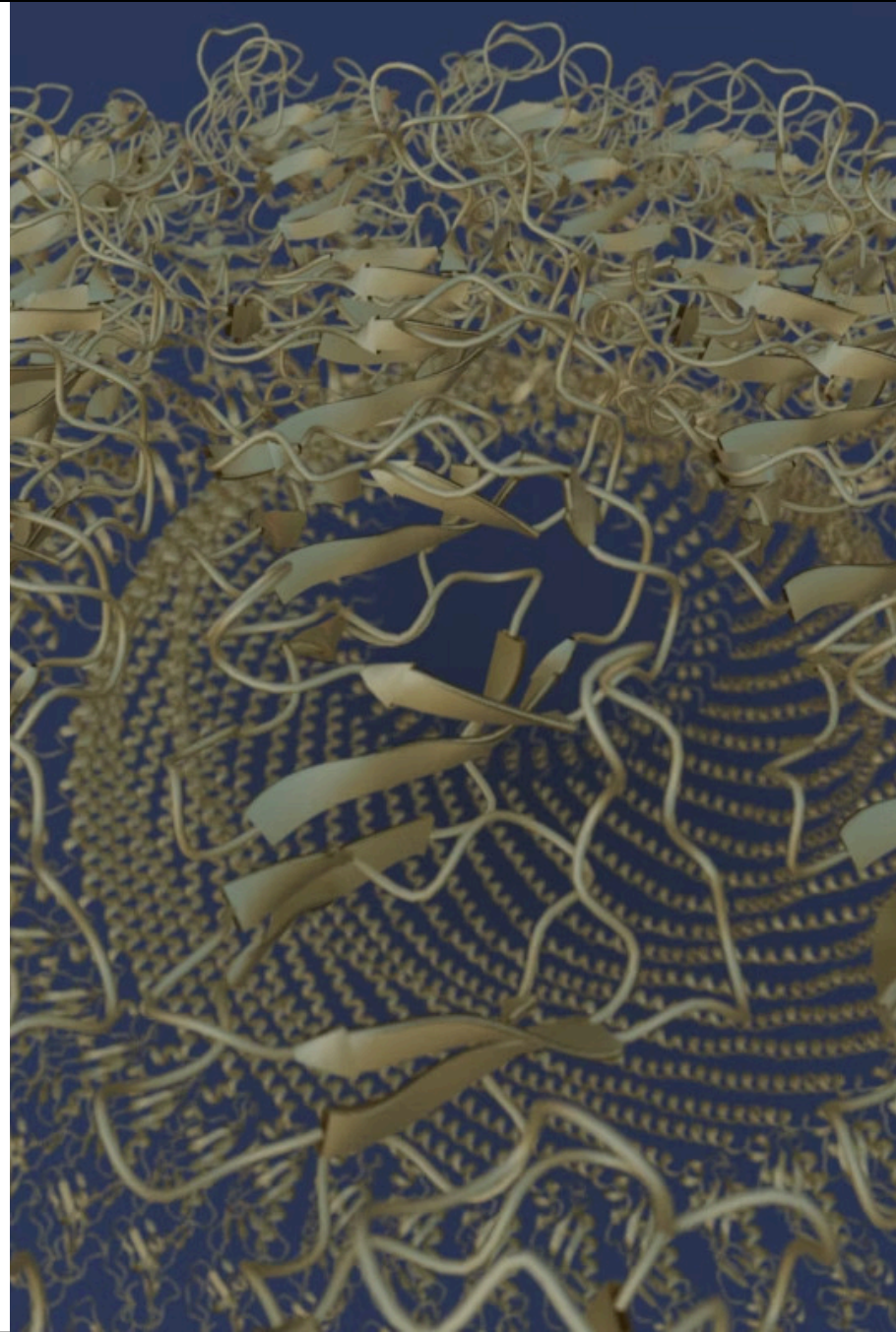
Smoke, clouds, fluids.

Real Materials

Glass, metals, scattering.

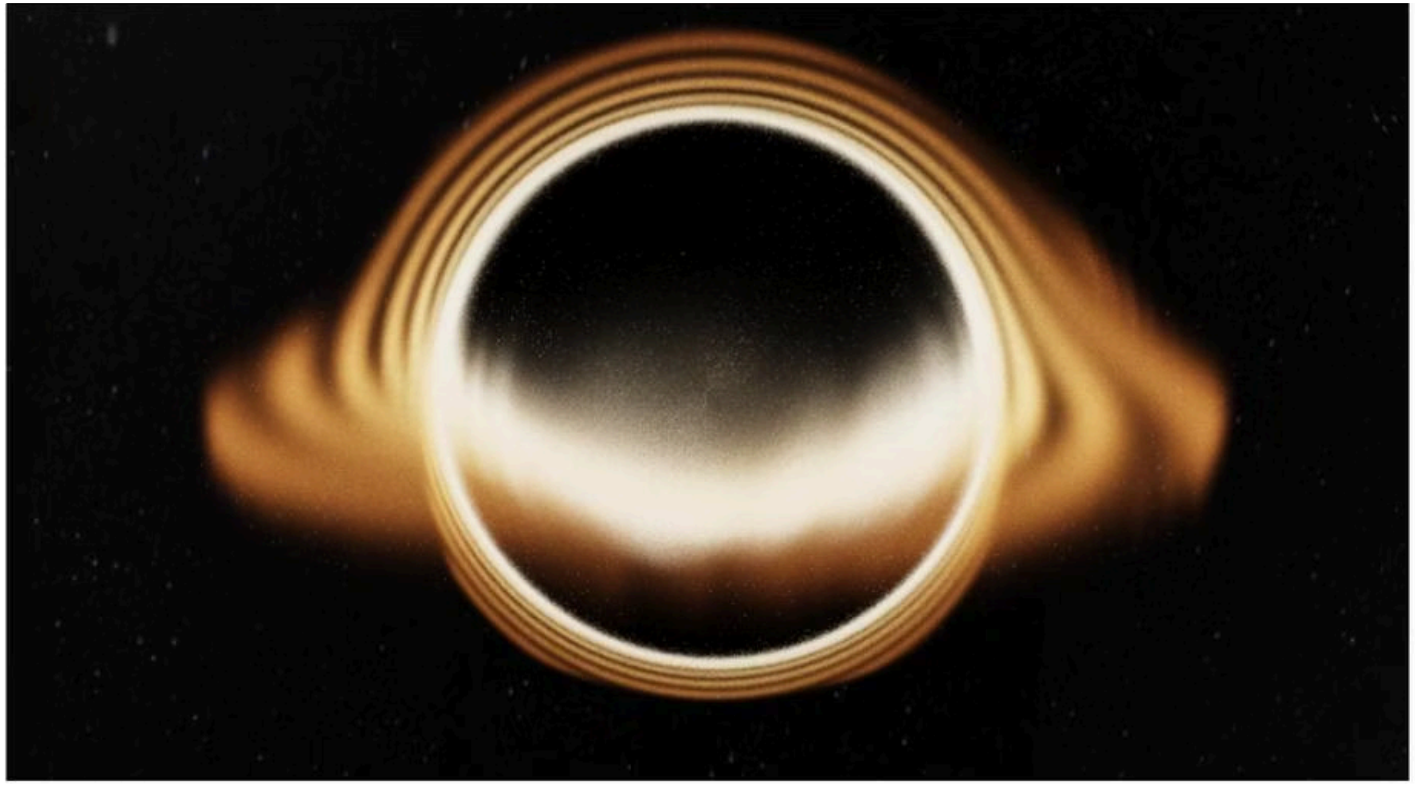
Scientific Quality

With physical based spectral rendering.



RayMakie + Hikari

RayMakie is a Makie backend built on **Hikari**, a spectral path tracer ported from pbrt-v4's `volpath`.



Same API

Most packages doing 3d plots in Makie can raytrace their visualization without any code change

Spectral Rendering

Light and materials use full spectra, enabling physically accurate materials and light types.

Cross-GPU

Runs on CUDA.jl, AMDGPU.jl and Metal.jl.

Hikari can render black-hole light bending in pure Julia.

My dream: Full Julia based GPU graphics stack

✅ GPU Compute

KernelAbstractions.jl + CUDA/AMDGPU work great.

❌ Graphics Shaders

No vertex/fragment shaders in Julia, stuck with GLSL.

❌ Hardware RT

None of the current julia GPU backends can support hardware acceleration for raytracing

We needed a backend that extends Julia's GPU story from compute into graphics and ray tracing.

Lava.jl - Julia on Vulkan

A new GPU backend: Julia code compiles to SPIR-V via a custom LLVM IR emitter - written entirely in Julia.
github.com/SimonDanisch/Lava.jl

1

Compute Backend

Drop-in for CUDA.jl / AMDGPU.jl via GPUArrays.jl + KernelAbstractions.jl

2

Graphics Shaders

Vertex/fragment/geometry/tesselation shaders written in plain Julia, no GLSL, no macros

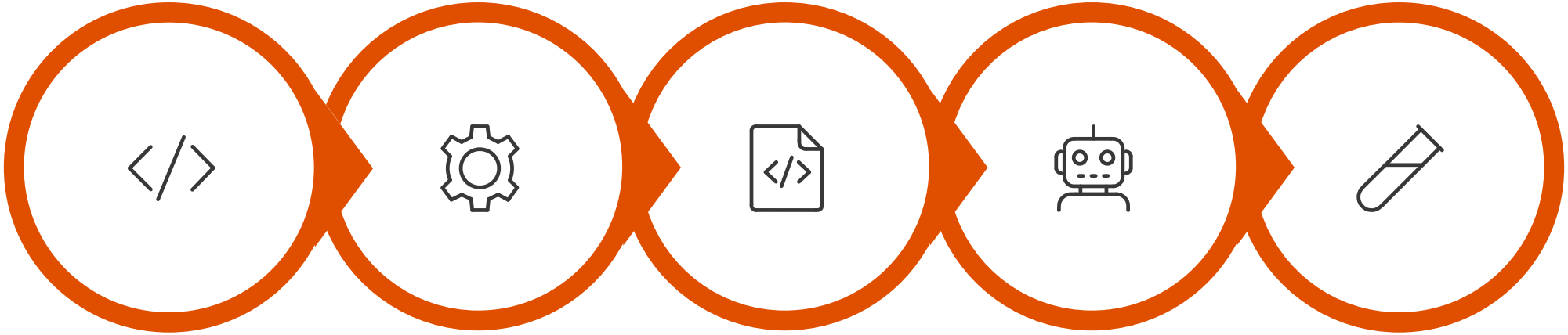
3

Hardware Ray Tracing

Real `VK_KHR_ray_tracing_pipeline` - hardware BVH traversal

Cross-platform: AMD, NVIDIA, Intel, Apple (MoltenVK), software (lavapipe). One unified `LavaArray` buffer across all shader stages - zero copies.

How It Works



Julia Function

GPUCompiler.jl

LLVM IR

SPIR-V Emitter

VkShaderModule

`llc` not needed.

Current Status

✅ What Works

~99.9% GPUArrays Tests

2,634+ passing

Full KernelAbstractions

@kernel code works

Hikari Path Tracer

Wavefront path tracer runs on Lava

⚠️ Limitations vs CUDA/AMDGPU

No BLAS / FFTW / Sparse

These libraries are not ported yet

Compilation Errors

Some packages still hit edge cases

Cross-Platform Maturing

Multi-vendor support is still evolving

Unstable Graphics APIs

Graphics and ray-tracing APIs are not stable

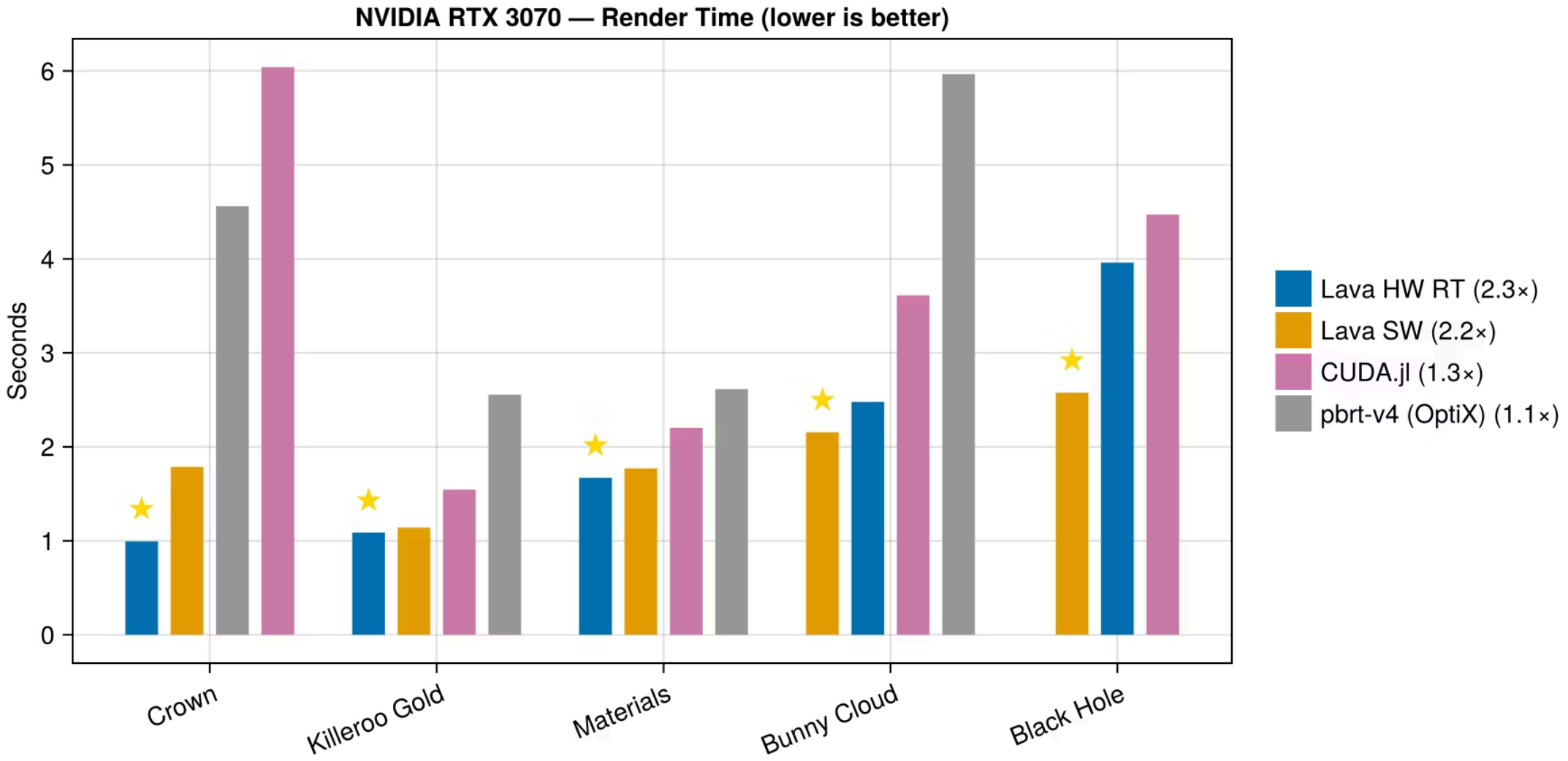
Julia-Written Graphics Shaders: Hello Triangle

```
using Lava, GLFW, GeometryBasics
function my_vertex(colors::LavaDeviceArray{Vec4f,1})
    pos = ((0f0, -0.5f0), (-0.5f0, 0.5f0), (0.5f0, 0.5f0))
    set_position!(Vec4f(pos[vertex_index()]..., 0f0, 1f0))
    gfx_output(0, colors[vertex_index()])
end
my_fragment() = gfx_output(0, gfx_input(Vec4f, 0))
colors = LavaArray(Vec4f[(1, 0, 0, 1), (0, 1, 0, 1), (0, 0, 1, 1)])
pipeline = Rasterizer(vertex=my_vertex, fragment=my_fragment)
win = RenderWindow(800, 600; title="Hello Triangle")
while isopen(win)
    GLFW.PollEvents(); acquire_next_image!(win)
    draw!(pipeline, WindowTarget(win), 3; args=(colors,), tt_vertex=Tuple{LavaDeviceArray{Vec4f,1}})
    present_frame!(win)
end
```

Plain Julia functions as vertex and fragment shaders. No GLSL, no macros, no external toolchain.

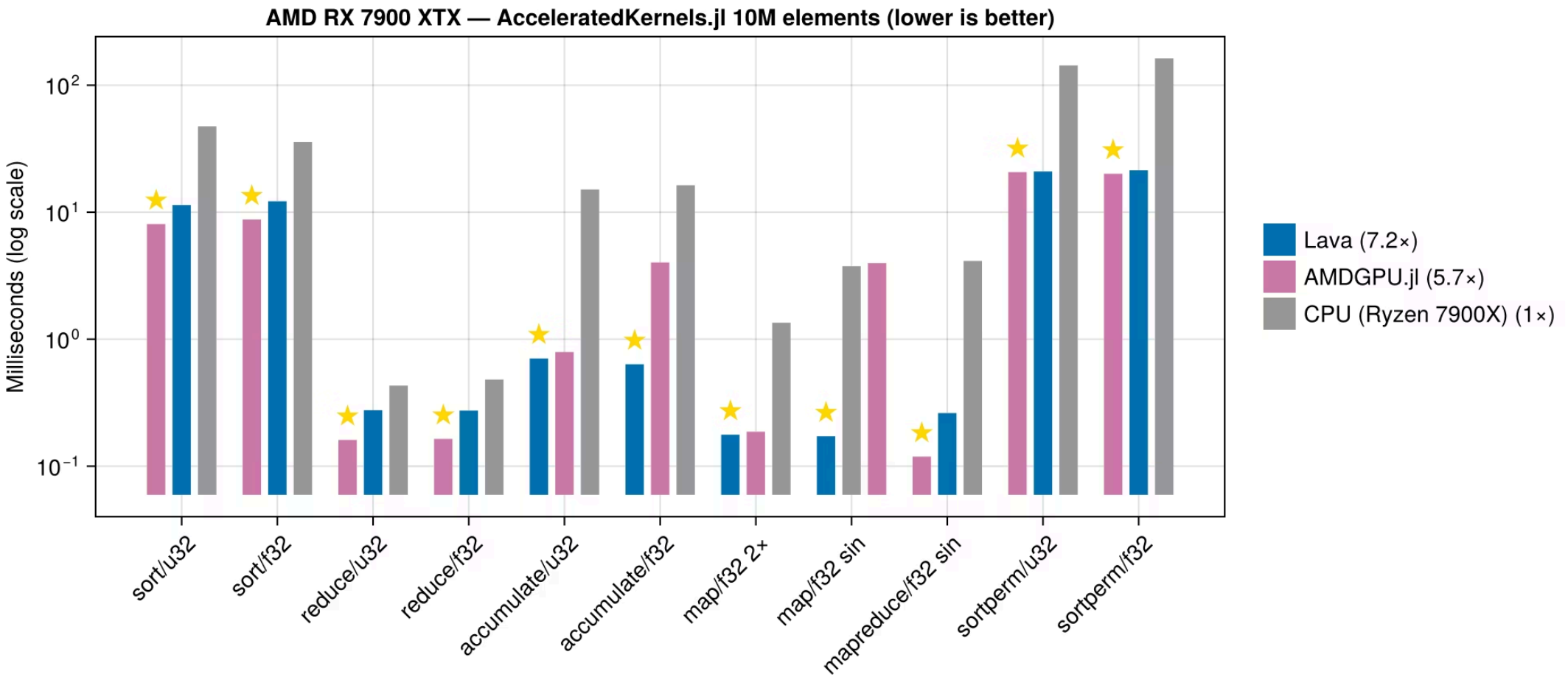
Benchmarks: Raytracing Performance

Lava HW RT 2.3x, SW 2.2x faster than CUDA.jl and original pbrt-v4 with NVIDIA Optix hardware acceleration



Benchmarks: AcceleratedKernels.jl

Lava leads or matches AMDGPU.jl across all operations



Building Lava.jl with Claude Code

✅ How It Was Built

~90% written by claude code

Mostly developped overnight on autopilot

16,000 Lines

Compiler, runtime, and array infrastructure

Best case for AI

Full GPUArrays test suite + Hikari tests scenes

⚠️ The Pitfalls

Regression rollercoaster

Fixes on one platform silently break another.

Tests help, but...

They make iteration cycles significantly slower.

Code Quality

The codebase is likely pretty messy.

Thoughts on Agentic Coding

Momentum Over Perfection

Having something to build excitement around the technology matters more than having perfect code from the start.

What Even Is a Messy Codebase?

When you can write a huge and complex package from scratch in a couple of days, the bar for 'clean code' needs rethinking.

Debugging

Claude Code with Julia execution is surprisingly effective at finding and fixing bugs

Technical Debt?

With agentic coding and good test coverage, I don't expect to debug Lava.jl by hand going forward.

What's Next

1. **VulkanMakie**
   Full Makie rendering backend with pure Julia vertex/fragment shaders replacing GLMakie
2. **Stable Multi-Vendor Support**
   Continuous CI across NVIDIA, AMD, Intel, Apple MoltenVK and lavapipe to ensure reliable cross-platform behavior
3. **RayMakie + VulkanMakie Integration**
   Seamless interop between ray tracing and rasterization, shared buffers, unified scene graph, and tight compute integration
4. **WebGPU**
   SPIR-V → Naga → WGSL: Julia GPU code running in the browser

Try it now: `]add https://github.com/SimonDanisch/Lava.jl` Works on any GPU with Vulkan support.

What's Next: Compute + Visualization

Lava as a unified GPU backend is uniquely positioned for simulations that run on the GPU and have complex visualization demands on top.

GPU Simulations

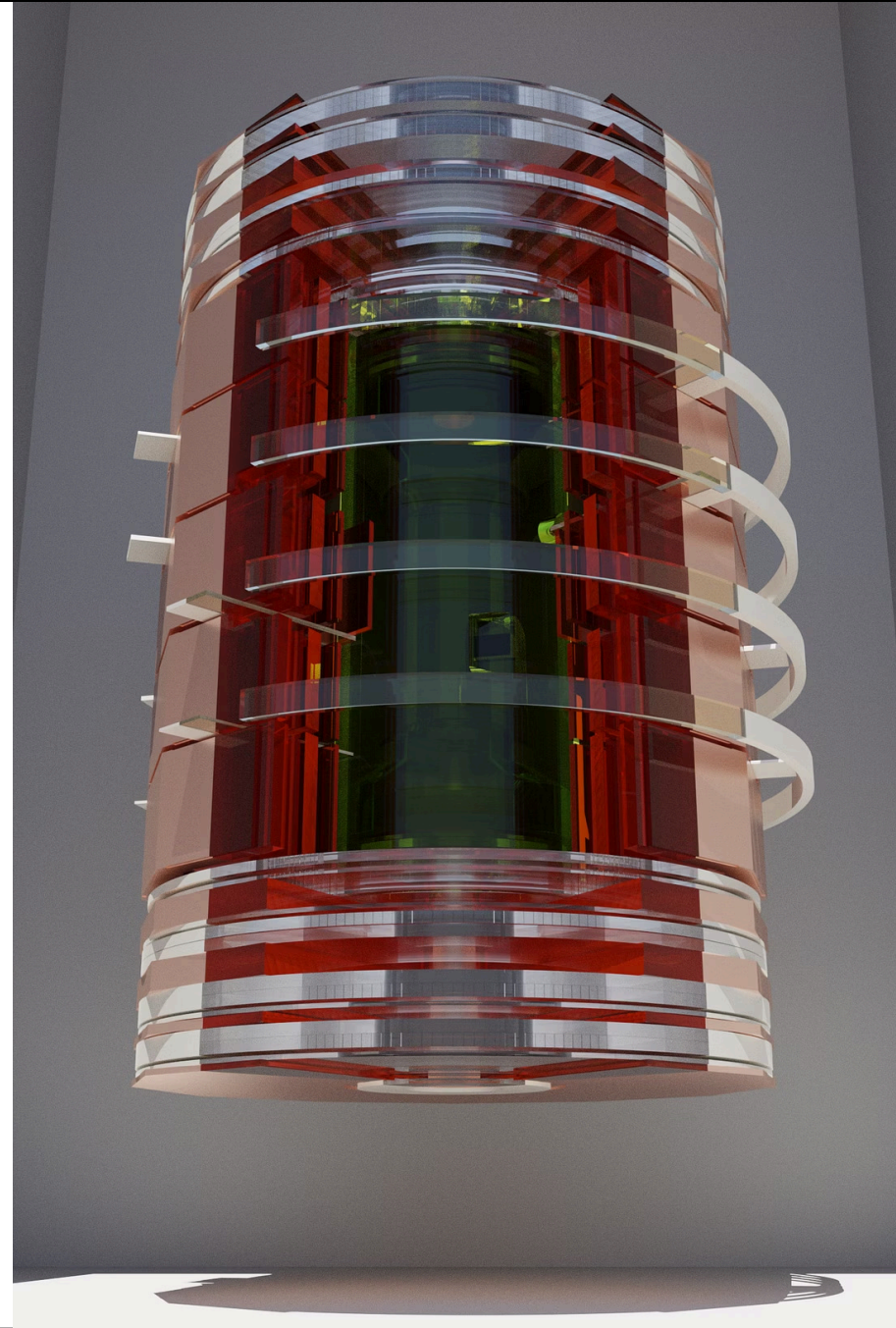
Run fluid, particle, or physics simulations entirely on the GPU with KernelAbstractions and Lava.

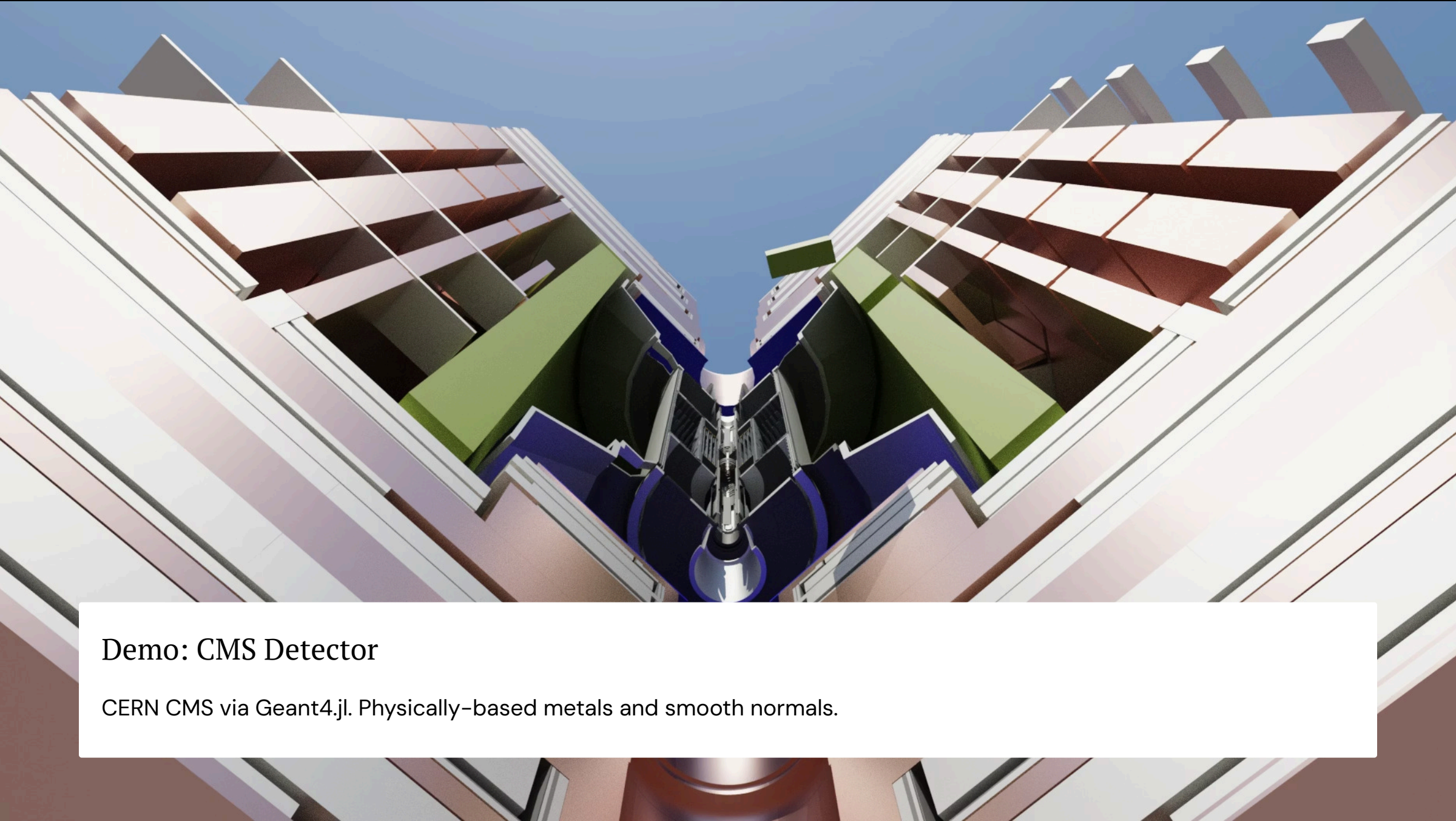
Zero-Copy Visualization

Simulation data stays on the GPU, no round-trips to CPU before rendering, works with both RayMakie and VulkanMakie.

Julia Callbacks for Pre/Post Processing

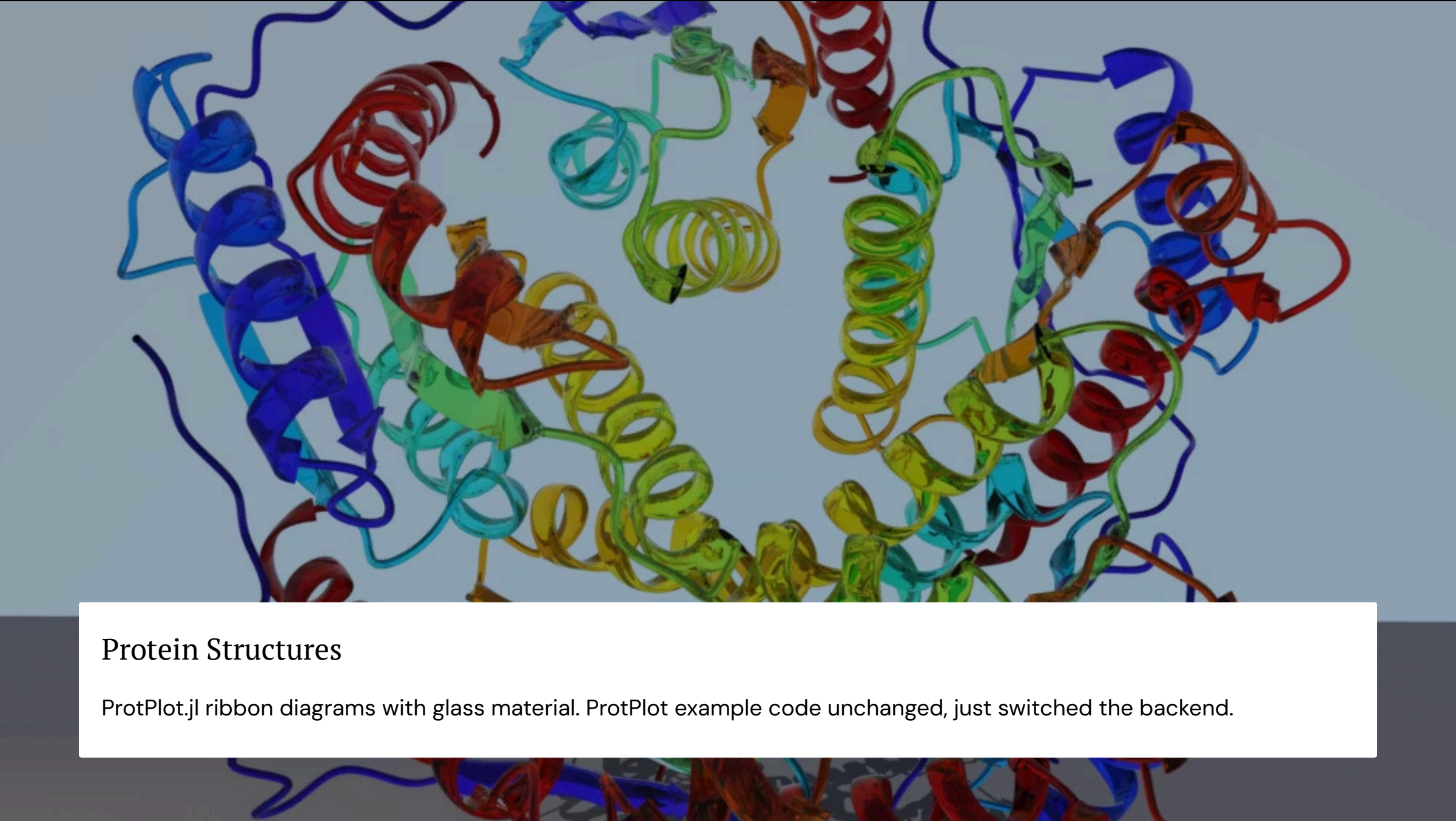
Use Julia functions as GPU callbacks for on-GPU pre- and post-processing, tightly integrated with the rendering pipeline.





Demo: CMS Detector

CERN CMS via Geant4.jl. Physically-based metals and smooth normals.



Protein Structures

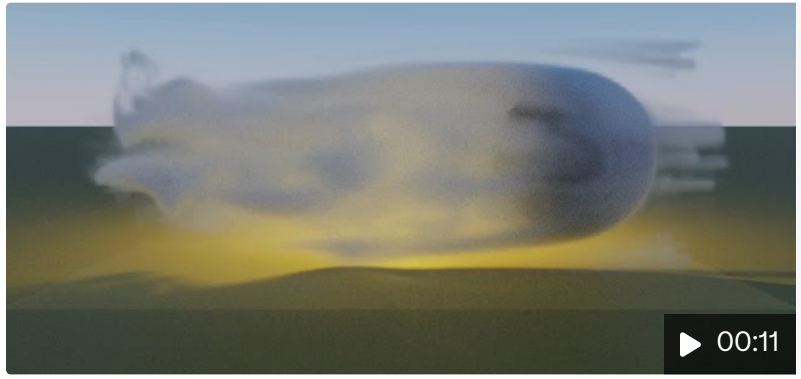
ProtPlot.jl ribbon diagrams with glass material. ProtPlot example code unchanged, just switched the backend.

Volumetric Clouds

Stanford bunny via NanoVDB, a sparse volume serialization format, which can be loaded directly from disc to GPU

WaterLily.jl Raytraced

WaterLily.jl fluid simulation rendered with RayMakie + Lava.jl.

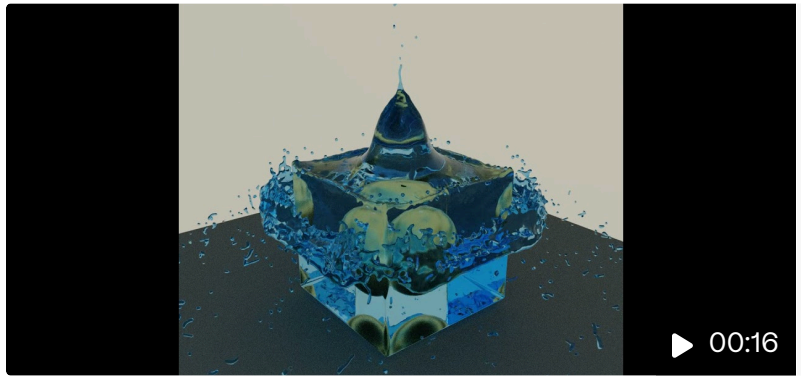


YouTube

Waterlily.jl simulation rendered with Makie raytracing backend

Water Splash (TrixiParticles.jl)

TrixiParticles.jl fluid sim with glass material (IOR 1.33) – realistic refraction and caustics.



YouTube

TrixiParticles.jl water animation with Makie ray tracing backend.

Thank You

Simon Danisch

Author of Makie, Bonito and GPUArrays

LinkedIn: www.linkedin.com/in/simondanisch

Mastodon: https://mas.to/@sdanisch

Bluesky: https://bsky.app/profile/simi.bsky.social