Modular Real-Time OpenGL Ray Tracing Engine

with Shader-Based Path Tracing and Temporal Filtering

Dumitru Zotea

University of Milan – MSc Computer Science

Real Time Graphics Programming (RTGP)

zotea.dumitru@studenti.unimi.it

December 2025

Abstract

This report details the design and implementation of a real-time ray tracing engine built on OpenGL 4.1 using a fragment-shader-based path tracing approach. The system employs a full-screen ray generation pass and supports analytic geometry as well as BVH-accelerated triangle meshes, complemented by environment lighting, one-bounce global illumination, and configurable mirror and glass materials.

Because only a small number of samples per pixel are computed each frame, the renderer integrates several temporal denoising techniques: multi-frame accumulation, motion-vector-driven temporal anti-aliasing (TAA), and a compact SVGF-style spatial filter guided by world-space GBuffer data. All components are implemented exclusively within the fixed-function OpenGL pipeline, avoiding compute shaders or hardware ray tracing extensions, and therefore remain fully portable across platforms.

An ImGui interface exposes all rendering parameters, enabling real-time inspection and experimentation. Performance is evaluated on an Apple M4 Max MacBook Pro and a Windows desktop with an NVIDIA RTX 5090 GPU. The results show that the system achieves temporally stable imagery at very low sample counts while sustaining interactive frame rates on both integrated and discrete GPUs.

Keywords: Real-Time Ray Tracing, OpenGL 4.1, Path Tracing, BVH Acceleration Structures, Temporal Anti-Aliasing, Spatiotemporal Denoising, SVGF, Global Illumination, GPU Rendering Techniques

Contents

1 Introduction

Real-time ray tracing has become increasingly relevant in modern rendering pipelines, and recent Apple Silicon GPUs (M3 and later) even provide dedicated ray tracing hardware. However, this hardware is only accessible through Metal, while macOS continues to expose OpenGL 4.1 as a legacy compatibility layer with no support for compute shaders or ray tracing extensions. This project examines how far real-time path tracing can be pushed within these constraints by implementing a complete ray tracing pipeline entirely in OpenGL 4.1. A full-screen fragment shader is used to launch rays for each pixel, ensuring compatibility across both discrete Windows GPUs and integrated Apple Silicon hardware despite the limitations of the OpenGL environment.

The renderer incorporates a broad set of rendering and denoising features:

- Analytic geometry (planes and spheres)
- BVH-accelerated triangle meshes for complex models
- Environment lighting and a configurable directional sun
- One-bounce global illumination and ambient occlusion
- Temporal accumulation and sub-pixel jitter for stochastic sampling
- Motion-vector-based temporal anti-aliasing (TAA) with luminance clamping
- A lightweight SVGF-inspired spatial filter guided by GBuffer data
- A complete ImGui designer/debug panel for parameter tuning

Since real-time path tracing typically operates at extremely low samples per pixel, the project places strong emphasis on temporal stability and denoising. Accumulation, motion-vector reprojection, and GBuffer-guided filtering work together to construct temporally stable images while maintaining interactive frame rates.

The primary goal of this work is not only to demonstrate that real-time ray tracing is feasible within the limitations of OpenGL 4.1, but also to design a modular and readable engine illustrating how modern ray tracing techniques can be adapted to a legacy graphics API.

The remainder of this report introduces the theoretical background, describes the engine architecture and rendering pipeline, details the core algorithms, and evaluates performance on both an Apple M4 Max MacBook Pro and a Windows desktop equipped with an NVIDIA RTX 5090 GPU.

2 Background

Ray tracing simulates the transport of light by tracing rays through the scene and evaluating their interactions with surfaces. Because each ray may intersect multiple primitives before contributing radiance to a pixel, the computational cost is typically much higher than that of rasterization, which projects geometry directly onto the image plane and shades visible surfaces only once.

To make real-time ray tracing feasible, modern approaches rely on several key concepts that reduce intersection cost, reuse information across frames, and suppress noise:

Acceleration structures. A bounding volume hierarchy (BVH) organizes geometry into a hierarchy of bounding boxes. This reduces the number of ray–triangle tests from linear to roughly logarithmic complexity, making mesh-based scenes practical for real-time applications.

- Rays intersect bounding boxes before testing individual triangles.
- Large portions of the scene can be skipped entirely.

Stochastic sampling and temporal reuse. Because only a few samples per pixel can be traced per frame, path-traced images remain noisy unless information is accumulated over time.

- Sub-pixel jitter distributes samples differently each frame.
- Multi-frame accumulation progressively reduces noise when the view is stable.

Spatiotemporal filtering. When the camera or objects move, accumulated samples must be re-aligned with the current frame. Real-time ray tracers combine temporal reprojection with lightweight spatial filtering to stabilize the image.

- Motion vectors reproject history samples into the current frame.
- Temporal anti-aliasing (TAA) blends new samples with reprojected history using confidence and rejection tests.
- SVGF-inspired filtering reduces residual noise through GBuffer-guided, edge-aware smoothing.

GBuffer-guided reconstruction. Modern spatiotemporal filters rely on auxiliary information stored in a GBuffer (such as position, normal, and albedo) to distinguish between edges and surfaces. These signals guide both temporal and spatial filters in preserving detail while eliminating high-frequency noise.

- Normals avoid blurring across object boundaries.
- Positions prevent smoothing across depth discontinuities.
- Variance estimates regulate filtering strength.

These concepts form the theoretical foundation of the renderer developed in this project. The following sections describe how each idea is implemented entirely within the constraints of the OpenGL 4.1 pipeline using GLSL fragment shaders and multipass rendering.

3 Development Environment and Libraries

The project was developed primarily on a 16" MacBook Pro equipped with an Apple M4 Max system-on-chip (14-core CPU, 32-core GPU) and 36 GB of unified memory, running macOS Tahoe 26.1. Although recent Apple Silicon hardware includes dedicated ray tracing units, these are only exposed through Metal. OpenGL 4.1 on macOS operates entirely through a legacy compatibility layer and does not provide access to compute shaders or hardware RT extensions. As a result, all ray tracing in this project is implemented purely through fragment shaders and multipass rendering.

To ensure cross-platform correctness and performance, the renderer is also built and tested on a Windows 11 desktop equipped with an NVIDIA RTX 5090 GPU, an AMD Ryzen 9 9900X CPU, and 32 GB of DDR5-6000 memory. This allowed direct comparison between integrated Apple Silicon graphics and a high-end discrete GPU while maintaining identical rendering logic and shader code.

Build Tools

The following tools and compilers were used throughout development:

- **CMake 3.26.4** for project configuration and build generation.
- **Apple Clang 16.0.0** as the C/C++ compiler on macOS.
- **MSVC (Visual Studio 2022)** as the compiler on Windows.

Third-Party Libraries

All external dependencies are included as Git submodules within the `libs/` directory and compiled as part of the build. The versions listed below were retrieved via `git describe` on each submodule:

- **GLFW** — `3.4-24-g509fc702` Used for window creation, input handling, and OpenGL context setup.
- **GLM** — `1.0.0-85-g2d4c4b4d` Header-only linear algebra library for vectors, matrices, and camera math.
- **Dear ImGui** — `v1.91.9b-92-gb8b040e2f` Provides the interactive debug and parameter-tuning interface.
- **Assimp** — `v5.4.3-169-gc4515f53c` Used to import mesh data before BVH construction.
- **tinyobjloader** — `v2.0.0rc13-8-g3bb554c` Lightweight OBJ loader used for additional test assets.
- **GLAD 2** — custom OpenGL loader generated through the official GLAD service.

In addition, the project uses `stb_image.h` for loading environment maps and textures. All libraries are linked statically to ensure portability and to avoid runtime dependencies beyond the platform-provided OpenGL drivers and frameworks (e.g., `Cocoa`, `IOKit`, and `CoreVideo` on macOS).

This setup provides a stable and reproducible environment for building, debugging, and benchmarking the renderer across macOS and Windows.

4 Engine Overview

The codebase is organized into a set of modular components, each responsible for a specific stage of the rendering pipeline. This structure keeps the project easy to extend, debug, and profile across both macOS and Windows.

- **Application layer** — initializes the window and OpenGL context, manages the main render loop, handles timing and frame synchronization, and coordinates all subsystems. This layer also manages startup/shutdown order to avoid unsafe destruction of GL resources.
- **Rendering layer** — encapsulates all GPU-side rendering logic, including shader compilation, framebuffer objects (FBOs), render target allocation, GBuffer generation, accumulation buffers, temporal reprojection, and the final present pass. This layer exposes high-level functions such as `renderRayPass()`, `renderTAA()`, and `renderPresent()`.
- **Scene layer** — stores geometric data and builds the acceleration structures. It handles:
  - loading triangle meshes via Assimp or tinyobjloader,
  - flattening geometry into GPU-friendly SOA buffers,
  - constructing the BVH on the CPU,
  - uploading nodes and triangles into texture buffer objects (TBOs) for GLSL access.
- **Camera and input system** — manages camera orientation, projection matrices, movement, mouse look, and key bindings. It also computes motion vectors and exposes camera-moved events to reset accumulation when needed.
- **UI layer** — built on Dear ImGui, providing real-time control of rendering parameters such as SPP, GI and AO multipliers, exposure, TAA weights, SVGF settings, material properties, debug modes, BVH visualization, and environment map intensity. This is the main interface for interacting with the renderer (Figure 1).

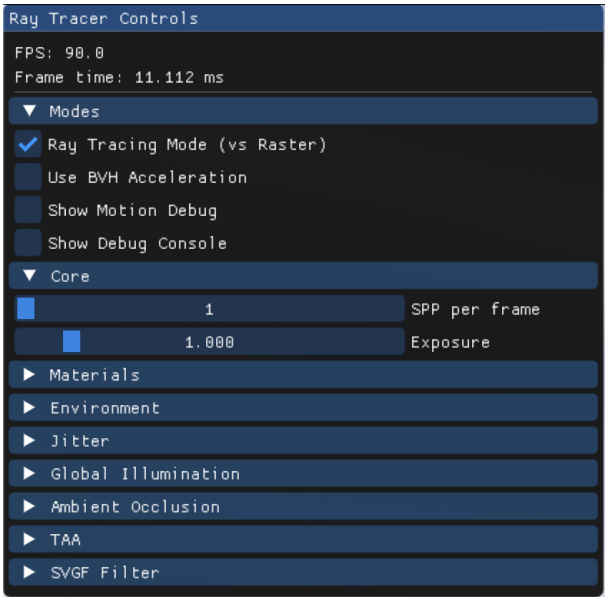

Figure 1: Interactive ImGui interface used to control sampling parameters, denoising filters, lighting settings, material properties, and debugging visualizers.

- **Shader modules** — the GLSL codebase is split into small, logically focused shader units, each responsible for a distinct part of the ray-tracing algorithm:
  - BVH traversal routines and triangle intersection logic,
  - analytic geometry (planes, spheres, infinite ground),
  - direct lighting, environment sampling, BRDF evaluation,
  - one-bounce GI and ambient occlusion,
  - temporal reprojection and TAA blending,
  - SVGF-style spatial filtering guided by GBuffer position/normal/variance.

The renderer operates in two distinct modes that share most of the infrastructure:

- **Raster mode** — a lightweight forward renderer used for debugging, visualizing BVH nodes, previewing mesh geometry, and verifying materials and camera behavior without the cost of path tracing.
- **Ray mode** — the full path-tracing pipeline, launching one or more rays per pixel and feeding the results into the accumulation, TAA, and SVGF stages for reconstruction.

Together, these modules form a compact but highly extensible architecture. Each layer can be modified independently, enabling experimentation with new sampling strategies, denoising filters, or geometric primitives without affecting the rest of the system.

5 Rendering Pipeline

Each frame proceeds through a sequence of steps that update the camera state, manage render targets, execute the ray tracing passes, and apply the final spatiotemporal reconstruction filters:

1. **Input polling and camera update.** Keyboard and mouse inputs are processed, the camera's view and projection matrices are updated, and motion information is recorded. If the camera moved, the accumulation buffer is reset to avoid reusing invalid history.

2. **Render target validation.** If the window size or resolution has changed, all affected textures (GBuffer, accumulation buffers, motion vectors, and final output targets) are reallocated to match the new viewport dimensions.

3. **Parameter upload.** All user-controlled settings exposed through ImGui—such as exposure, SPP, GI/AO toggles, TAA and SVGF weights, environment intensity, and material parameters—are uploaded to the shaders as uniforms.

4. **Ray tracing pass (when in ray mode).** A full-screen quad triggers the `rt.frag` shader, which performs:

   (a) primary ray construction with sub-pixel jitter,

   (b) analytic and BVH-accelerated intersection testing,

   (c) shading (direct light, environment, optional one-bounce GI),

   (d) GBuffer output of world-space position, normal, and motion vector,

   (e) raw radiance output for accumulation.

   This pass produces the noisy Monte Carlo estimate for the current frame along with the auxiliary data required by the reconstruction filters.

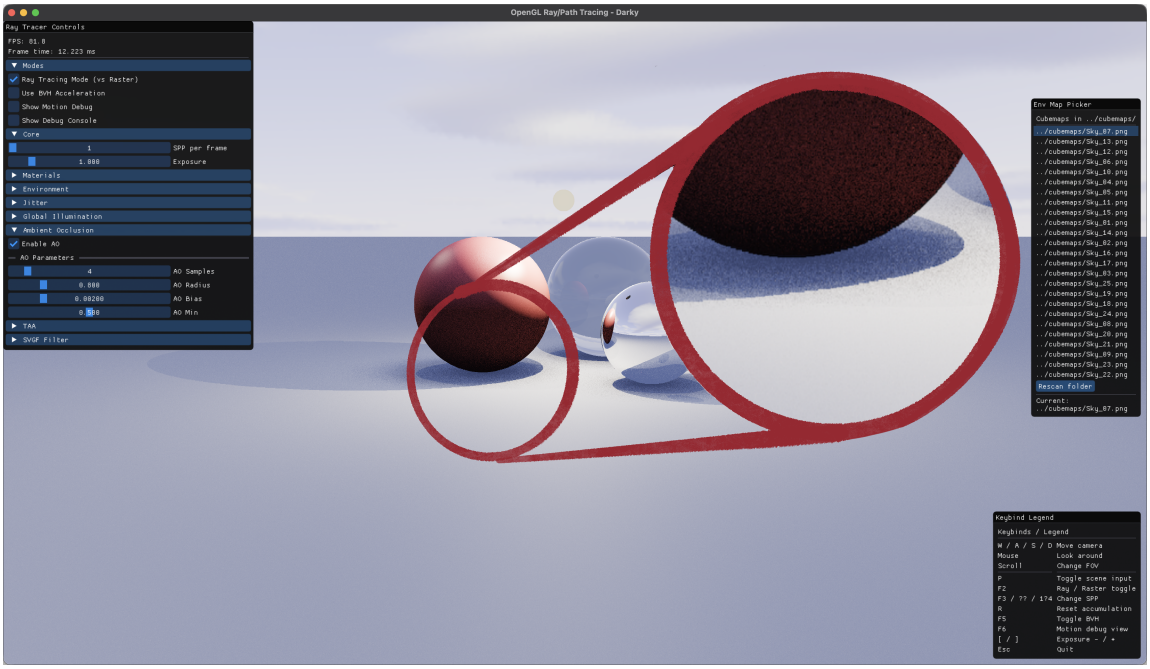

Figure 2: Raw Monte Carlo radiance output at 1 sample per pixel (SPP) produced by the ray tracing pass. The high noise level motivates the use of temporal accumulation, TAA, and SVGF filtering.

5. **Temporal resolve (TAA).** Motion vectors are used to reproject the previous frame's accumulated radiance into the current frame. A temporal anti-aliasing stage then blends the reprojected history with the new sample using luminance clamping and confidence weighting to suppress flickering and ghosting.

6. **Present pass (`rt_present.frag`).** The final shading stage performs:

   - per-pixel variance estimation from the accumulation buffer,
   - an SVGF-inspired edge-aware spatial filter guided by normals and depth,
   - tonemapping and gamma correction for display.

   This stage converts the temporally stabilized but noisy radiance estimates into the final denoised image.

7. **UI rendering.** The frame concludes with the ImGui pass, which renders all debug panels, parameter sliders, visualizers, and real-time statistics.

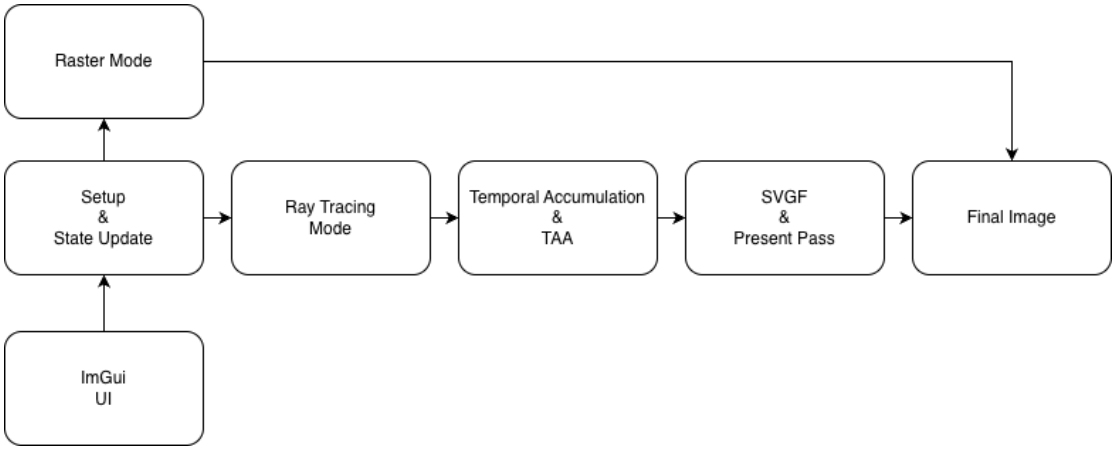

Figure 3: Overview of the multipass rendering pipeline including ray tracing, motion-vector extraction, temporal accumulation (TAA), SVGF-inspired filtering, and the final present pass.

When operating in **raster mode**, the pipeline bypasses the ray tracing and denoising stages entirely. Instead, the scene is rendered using a simple forward pass that draws analytic primitives, loaded meshes, and optional BVH debug visualizations. This mode enables rapid debugging of geometry, camera, and material settings without the cost of full path tracing.

6 Core Techniques

Fragment-based path tracing. The renderer performs all ray generation and shading inside a fragment shader. For each pixel, a primary ray is constructed from the camera origin and screen-space direction vectors derived from the view–projection matrix and field-of-view parameters. To enable temporal accumulation and reduce aliasing, a sub-pixel jitter (generated via a Halton sequence) offsets the ray origin slightly each frame. This produces a different sampling pattern per frame while remaining stable over time.

Ray–scene intersections are evaluated using:

- analytic sphere and plane intersection tests for simple primitives,
- BVH-accelerated triangle intersection for complex meshes.

The shading model includes direct lighting and an optional one-bounce indirect illumination term, allowing a coarse approximation of global illumination within the constraints of real-time performance.

BVH traversal. All BVH nodes and triangle data are flattened into arrays and uploaded to the GPU as texture buffer objects (TBOs). The GLSL traversal routine walks the hierarchy using a small, manually implemented stack. The layout of nodes is optimized for coherent access patterns across neighboring pixels, reducing branch divergence and improving GPU occupancy. The intersection shader tests bounding boxes before falling back to triangle-level intersection, significantly reducing per-pixel work.

Motion vectors and TAA. To stabilize the accumulated image during motion, each pixel stores a motion vector representing the difference between its previous-frame and current-frame NDC coordinates. These motion vectors are used to reproject history samples into the current frame. The temporal anti-aliasing (TAA) stage then applies:

- history reprojection using per-pixel motion vectors,
- luminance-based confidence weighting to reject unstable samples,
- clamping to prevent ghosting and overshooting artifacts,
- velocity-dependent blending to reduce smearing during fast motion.

This stage significantly improves temporal stability while maintaining low sample counts per frame (Figure 4).

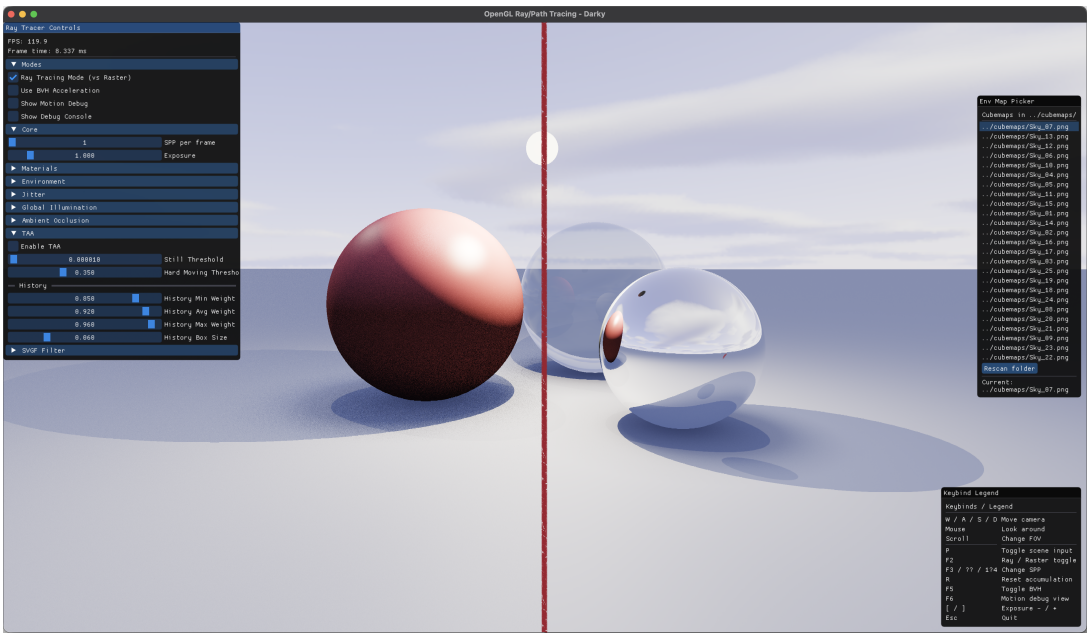

Figure 4: Effect of temporal anti-aliasing (TAA). Left: accumulated radiance without motion-vector-based reprojection, showing ghosting and instability. Right: TAA-corrected image using history reprojection and luminance clamping.

SVGF-style filtering. After temporal accumulation, residual high-frequency noise is removed using an SVGF-inspired spatial filter. The filter samples a local neighborhood around each pixel and performs edge-aware smoothing based on:

- similarity of surface normals,
- depth proximity to avoid crossing discontinuities,
- color agreement across samples,
- estimated variance in the accumulated signal.

These cues are derived from the GBuffer and the history buffer, enabling selective denoising that preserves geometric edges, lighting transitions, and fine detail while reducing Monte Carlo noise (Figure 5).

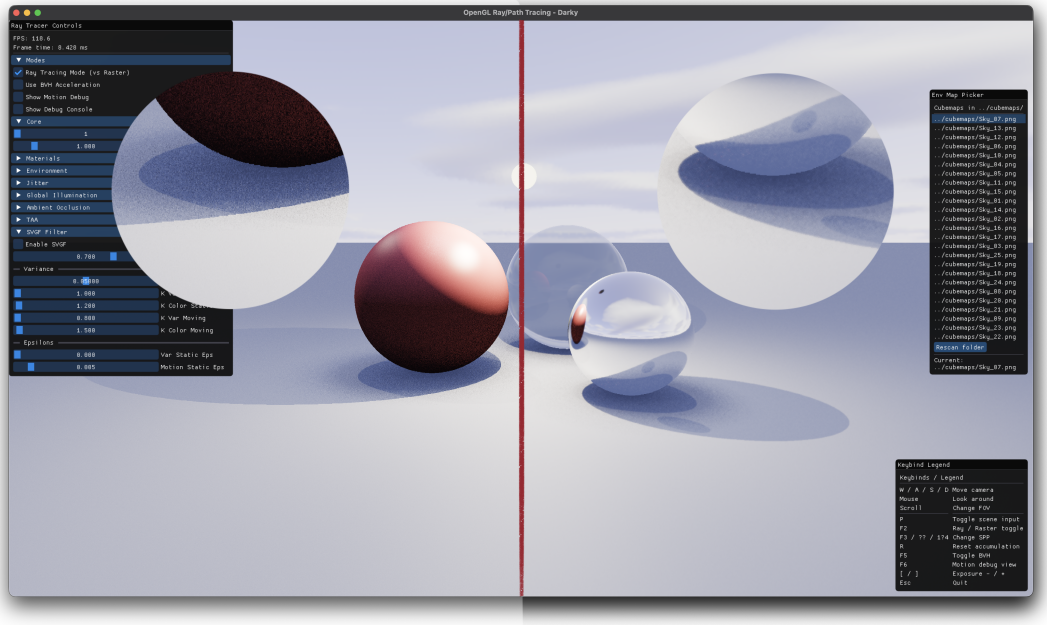

Figure 5: Comparison of the SVGF-inspired spatial filtering stage. Left: temporally accumulated but still noisy radiance. Right: denoised result produced by edge-aware filtering guided by position, normal, and variance cues.

7 Implementation Details

Render parameters. All user-adjustable settings are stored in a unified `RenderParams` struct, which acts as the central configuration block for the renderer. It includes: SPP settings, exposure, GI and AO multipliers, material properties (albedo, gloss, IOR, mirror strength), jitter toggles, TAA and SVGF weights, environment intensity, and various debug flags. At the start of each frame, this struct is uploaded to GLSL using a tightly packed uniform interface, ensuring that all shader stages operate on consistent state without redundant updates.

GBuffer. The renderer maintains a compact GBuffer containing:

- world-space normals,
- world-space positions,
- per-pixel motion vectors in NDC space.

These textures are written during the ray tracing pass and consumed in later stages. The GBuffer reveals geometric discontinuities and surface orientation changes, allowing TAA and SVGF to make edge-aware decisions during reprojection and filtering

Framebuffers. The pipeline uses multiple FBOs to isolate intermediate results and support temporal accumulation:

- **Ray tracing FBO** (MRT) — produces raw radiance, normals, positions, and motion vectors.
- **Accumulation/History FBO** — stores the temporally accumulated radiance and variance estimators used for TAA and SVGF.
- **Present FBO** — applies spatial denoising, tonemapping, and gamma correction to generate the final frame.

This separation allows each stage to operate independently and prevents feedback loops that could corrupt temporal history.

Environment lighting. The engine includes a lightweight cubemap loader capable of assembling a full cubemap from a 4×3 cross-layout texture. During rendering, environment lookups are sampled using the reflected or refracted ray direction. The ImGui interface lets the user dynamically adjust environment intensity, toggle between HDR maps, and visualize the active lighting configuration.

Platform specifics. The renderer targets OpenGL 4.1—the final version officially supported on macOS. Since macOS exposes OpenGL through a Metal-backed compatibility layer, compute shaders and hardware ray tracing extensions are unavailable. As a result, all ray traversal, shading, accumulation, and denoising logic is implemented entirely within fragment shaders. On Windows, the same OpenGL path runs unmodified on modern NVIDIA/AMD drivers, enabling direct performance comparisons between integrated Apple Silicon GPUs and high-end discrete GPUs.

8 Performance Evaluation

Performance tests were run at a resolution of **TODO insert resolution** on the following systems:

- Apple M4 Max (32-core GPU)
- NVIDIA RTX 5090 (80% power limit)

For each configuration, the camera was placed in a representative scene with both analytic primitives and BVH-loaded meshes. Frame rate was measured after the accumulation and TAA pipeline had stabilized, averaging the FPS over several seconds of continuous rendering.

8.1 Configurations tested

The following renderer configurations were evaluated:

- **Baseline** — 1 SPP, no GI/AO, no TAA, no SVGF.
- **Realistic** — 1 SPP, GI + AO + TAA enabled (no SVGF).
- **Full** — 1 SPP, GI + AO + TAA + SVGF.
- **Heavy** — 4 SPP, GI + AO + TAA + SVGF.

Tables will be filled after benchmarking.

Table 1: Performance on Apple M4 Max (32-core GPU).

Config	SPP	GI/AO	TAA/SVGF	FPS
Baseline	1	off/off	off/off	TODO
Realistic	1	on/on	on/off	TODO
Full	1	on/on	on/on	TODO
Heavy	4	on/on	on/on	TODO

Table 2: Performance on NVIDIA RTX 5090 (80% power limit).

Config	SPP	GI/AO	TAA/SVGF	FPS
Baseline	1	off/off	off/off	TODO
Realistic	1	on/on	on/off	TODO
Full	1	on/on	on/on	TODO
Heavy	4	on/on	on/on	TODO

9 Conclusion

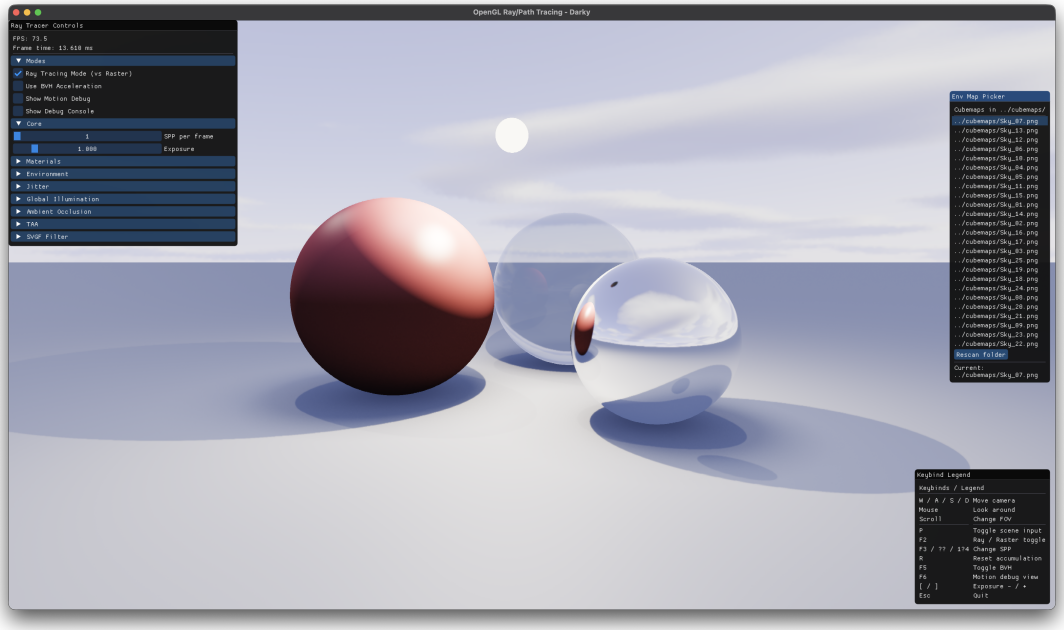

Figure 6: Final reconstructed frame after temporal accumulation, TAA, SVGF filtering, and tonemapping. The renderer maintains interactive frame rates at 1 SPP.

This project demonstrates that real-time ray tracing can be achieved within the constraints of OpenGL 4.1 using only fragment shaders and multipass rendering. Despite the absence of compute shaders or hardware ray tracing extensions, the engine combines BVH-accelerated traversal, temporal accumulation, motion-vector reprojection, and SVGF-inspired filtering to produce stable images at extremely low samples per pixel. The modular design of the codebase, together with the interactive ImGui interface, provides fine-grained control over sampling, filtering, lighting, and material parameters, making the renderer both flexible and educational.

Performance measurements across Apple Silicon and high-end discrete GPUs show that the approach scales effectively on modern hardware, maintaining interactive frame rates even with complex geometry and global illumination enabled. These results highlight the viability of fragment-shader-based ray tracing as a practical platform for experimentation and teaching, and demonstrate how modern rendering techniques can be adapted to legacy graphics APIs while still achieving visually credible real-time performance.

References

[1] G-Truc Creation. Glm: Opengl mathematics. `https://github.com/g-truc/glm`, 2025. Accessed: 2025-12-02.

[2] GLFW Development Team. Glfw: Open source library for opengl. `https://www.glfw.org/`, 2025. Accessed: 2025-12-02.

[3] Omar Cornut and Contributors. Dear imgui. `https://github.com/ocornut/imgui`, 2025. Accessed: 2025-12-02.