CGR Raytracer Coursework Report

System: macOS 25.0.0, clang-17.0.0

1 Implemented Features

1.1 Ray Object Intersection

Complete intersection testing is implemented for multiple object types, including full support for transformations.

Supported Shapes:
1. Sphere: Quadratic solver [Figure 1]
2. Cube: AABB slab method [Figure 2]
3. Plane: Bounded plane with normal-based intersection [Figure 1]
4. Cylinder: Body and cap intersection with bound checking [Figure 4]
5. Cone: Quadratic intersection with height constraints [Figure 4]
6. Torus: Quartic solver with Newton-Raphson refinement [Figure 4]

All shapes are defined as unit primitives and are transformed using TRS (Translation-Rotation-Scale) matrices.

1.2 Material System

The material system utilizes Blinn-Phong shading with physically-based extensions.
- Diffuse (Lambertian): Base color with texture support [Figure 1]
- Specular/Reflective: Blinn-Phong highlights [Figure 7]
- Glossy reflection: Importance-sampled rough reflections [Figure 6]
- Refractive/Transparent: Snell's law with Fresnel equations [Figure 6]
- Emissive: Self-illuminating materials [Figure 4]

Reflections are colored by the base texture for metallic surfaces (the color difference is visible in [Figure 2]). Additionally, an emission strength threshold is employed to bypass Blinn-Phong shading for strong emitters.

1.3 Lighting

The lighting system supports area lights, soft shadows, and physically-based falloff.

Light Types Supported:

- Point light [Figure 1]
- Area light [Figure 6]
- Spot light [Figure 6]
- Emissive materials [Figure 5]

Intensity falloff follows the inverse square law, and soft shadows are rendered using stratified sampling.

1.4 Acceleration Structure

A Bounding Volume Hierarchy (BVH) with median-split partitioning accelerates intersection tests. The BVH also supports temporal bounding boxes to facilitate motion blur [Figure 7].

1.5 Texture Mapping

Bilinear-filtered texture sampling with UV coordinates is implemented for all shapes. In-memory caching is used to eliminate redundant disk reads for textures.
UV coordinates are generated for all shapes; specifically, the torus uses toroidal coordinates (φ around the major radius, θ around the minor radius). Bilinear filtering provides smooth texture interpolation. A V-flip correction accounts for the image coordinate system, and UV clamping ensures textures are stretched rather than repeated.

1.6 Motion Blur [Figure 5]

Motion blur simulates object movement during exposure by sampling rays at random points in time.

1.6.1 Implementation Details

- Ray struct includes a time parameter [0, 1].
- Matrix-based interpolation is used for transformations (TRS decomposition + SLERP).
- Temporal bounding boxes are implemented for BVH efficiency.

1.7 Depth of Field [Figure 7]

A thin lens camera model with aperture and focal distance control is implemented. Points are randomly sampled on the lens aperture disk, and all rays pass through a specific point on the focal plane to ensure sharp focus. Uniform disk sampling produces circular bokeh effects.
CLI: `--depth-of-field <aperture> <focal_distance>`

1.8 Soft Shadows [Figure 6]

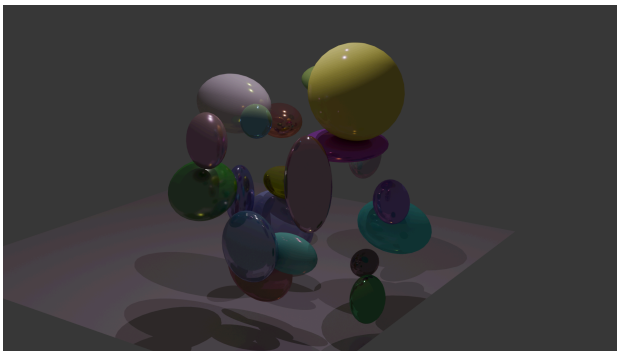
Stratified sampling of area light sources produces realistic soft shadows. Square, rectangle, and disk area lights are supported. Shadows remain transparent when casting through glass objects.
CLI: `--soft-shadows <samples>`

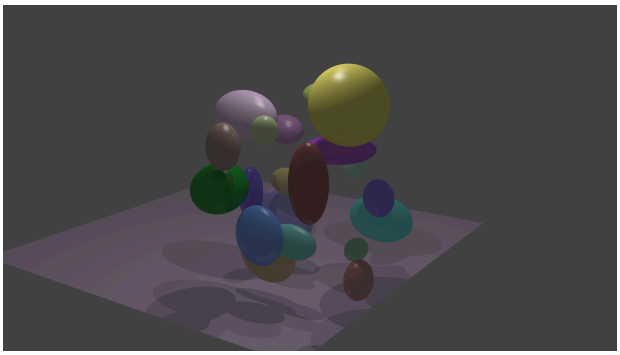
1.9 Glossy Reflections

Importance-sampled rough surface reflections use a power-cosine distribution.
CLI: `--glossy-reflection <samples>`
Glossiness values of 1.0 represent perfect mirrors, while values below 0.94 represent rough or diffuse objects. Importance sampling reduces noise, and the recursion depth is limited to 2 to prevent exponential ray explosion. Glossy reflections are colored by the base material color.

1.10 Test Results

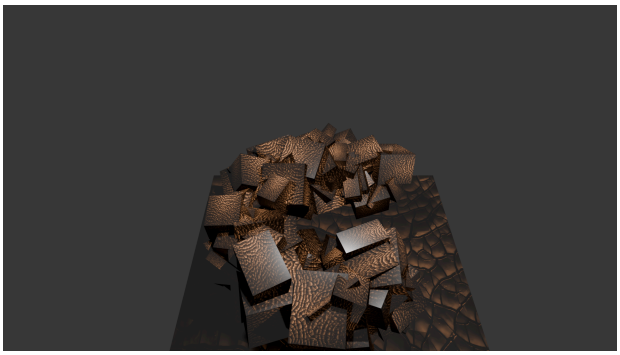
The following figures display the rendered outputs from the raytracer for all seven test scenes.

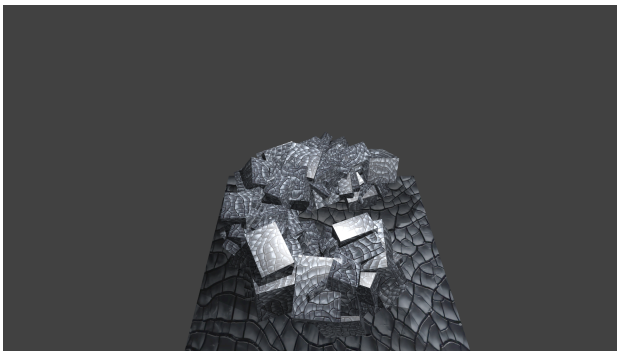




Expected Output Rendered Output

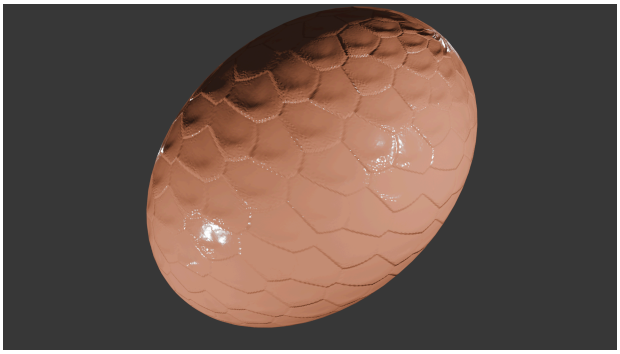
Figure 1: Comparison of render outputs for Test1.blend

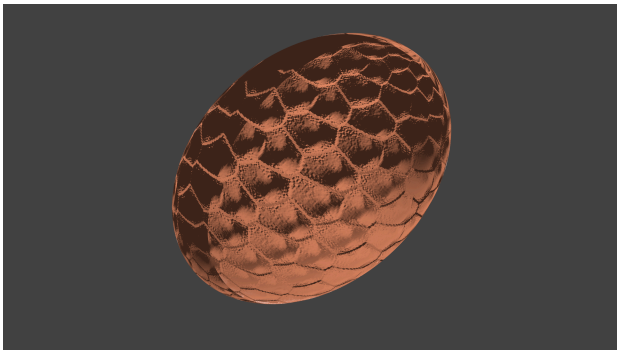




Expected Output Rendered Output

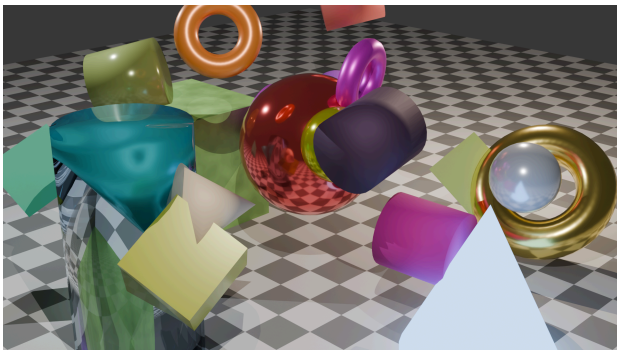
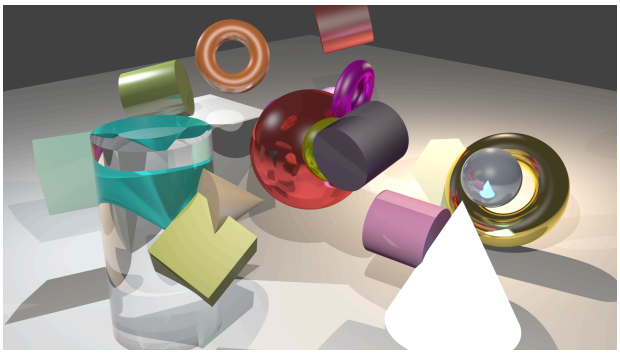
Figure 2: Comparison of render outputs for Test2.blend





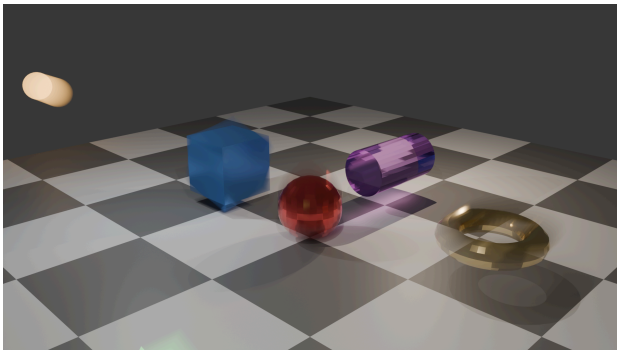
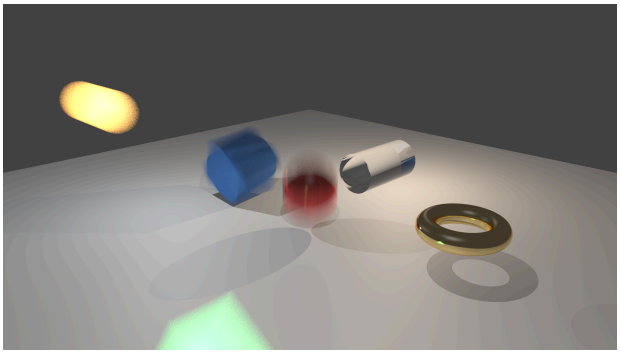
Expected Output Rendered Output

Figure 3: Comparison of render outputs for Test3.blend

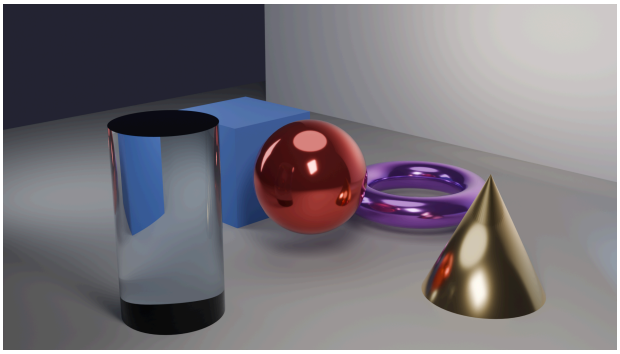
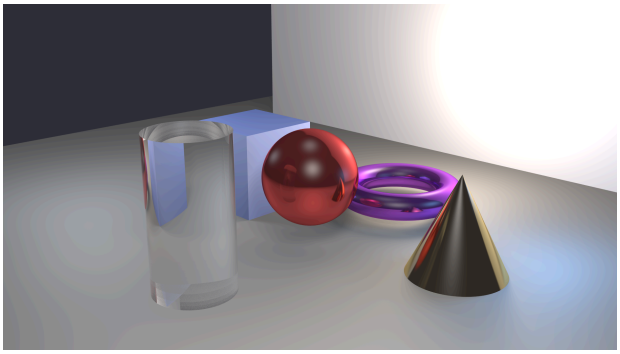
Expected Output Rendered Output

Figure 4: Comparison of render outputs for Test4.blend

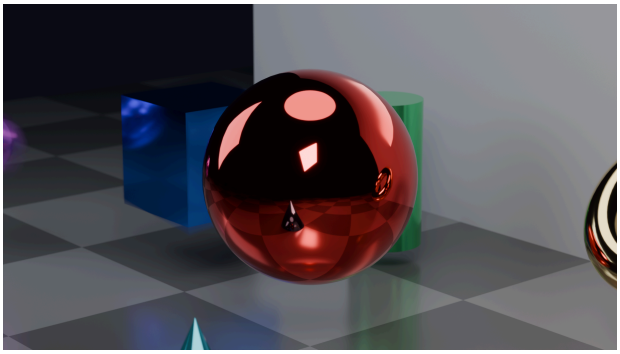
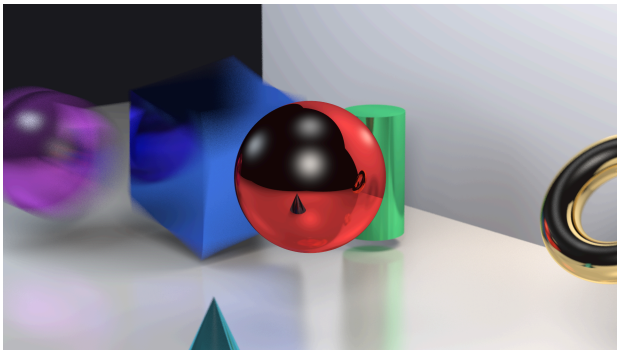
Expected Output Rendered Output

Figure 5: Comparison of render outputs for Test5.blend

Expected Output Rendered Output

Figure 6: Comparison of render outputs for Test6.blend

Expected Output Rendered Output

Figure 7: Comparison of render outputs for Test7.blend

2 Module Completion Status

Module/Topic	%	Description
Module 1	**100**	
Blender Exporter	100	Python export script
Camera Space	100	Pin-hole camera with proper transformations
Image R/W	100	PPM reader and writer
Module 2	**100**	
Ray Intersection	100	Sphere/Cube/Plane
BVH Implementation	100	BVH hierarchical structure with traversal
Module 3	**100**	
Whitted-style raytracing	100	Recursive reflection and refraction
Antialiasing	100	Multi-sample AA (configurable)
Texture mapping	100	UV mapping with bilinear filtering
Final Raytracer	**100**	
System Integration	100	CLI args, modular architecture
Distributed RT	100	Soft shadows + glossy reflection
Lens Effects	100	Motion blur + depth of field
Exceptionalism	**100**	
Ray Intersection for Complex Objects	100	Cone/Cylinder/Torus
Parallelization	100	OpenMP support
Structured Logging	100	Custom logger inspired by Zerolog
Thread-safe progress bar	100	`tqdm`-style progress bar with dynamic prediction

Table 1: Completion Overview

3 Timeliness Bonus Justification

All Module 1 features were implemented and checkpointed. Testing was performed using the Lab 2 ray visualization script in Blender with ray-to-cylinder conversion. *(17th October, 2025)*
All Module 2 features were implemented, and the details of their implementation were documented. An empirical test was performed using both BVH and brute-force techniques to verify the raytracer's output. *(31st October, 2025)*
All Module 3 features were implemented and checkpointed. The Whitted-style raytracing system includes recursive reflection/refraction (max depth 5), composite BRDF shading (Lambertian + Blinn-Phong), hard shadows via shadow rays, a full material system (diffuse, specular, ambient, reflectivity, transparency, IOR), stochastic supersampling antialiasing with configurable samples, and UV texture mapping for all primitive types with a centralized TextureManager. *(21st November, 2025)*
All Module 4 features were implemented. These include distributed raytracing with area lights (rectangular, disk) using stratified grid sampling for soft shadows, glossy reflections using power-cosine importance sampling, motion blur via temporal interpolation, and depth of field with a thin lens camera model. Additionally, a comprehensive CLI interface and multithreaded processing (using OpenMP) were implemented.

4 Exceptional Features

4.1 Torus/Cylinder/Cone Intersection [Figure 7]

The torus implementation required solving the quartic polynomial equation for ray-torus intersection, going beyond basic geometric primitives. This involved using Ferrari's method (for the quartic equation), cubic resolvent solving (for the cubic arising from Ferrari's method), and Newton-Raphson refinement (to ensure the hit point lies exactly on the surface).
While the coursework only required basic primitives (cube, sphere, plane), I chose to implement a mathematically complex surface to deepen my understanding of intersection algorithms.
In addition to the torus, I also implemented intersections for cylinders and cones, which required solving quadratic equations with additional calculus for the cone.

4.1.1 Technical Challenges

1. **Quartic Solver Instability**: Initially, I used floating-point precision, which produced NaN values. Switching to `long double` resolved this issue.
2. **Noisy Roots**: Quartic roots had errors of approximately 10^{-4}, causing visible artifacts. This was solved using Newton-Raphson refinement with 3-5 iterations.
3. **Fast-Math Incompatibility**: The `-ffast-math` compilation flag, which does not support infinites, broke the NaN checks in the quartic solver. This flag had to be removed from the compilation process.

4.2 OpenMP Multithreading with Progress Tracking

Parallel rendering with dynamic load balancing and thread-safe progress reporting was implemented. This was primarily a quality-of-life improvement, as tracking render progress was difficult, and multithreading provided approximately an 8x speed boost on my personal laptop.
Using dynamic estimated time prediction, I implemented a Python `tqdm`-style progress bar to visualize the render status. This is thread-safe and utilizes lock-free atomic counting from multiple threads to track progress. To predict the ETA correctly and prevent jitteriness, I used Exponential Moving Averages.

4.3 Structured Logger (Zerolog inspired)

I implemented a custom thread-safe structured logging system, inspired by Go's Zerolog library. The logger uses chainable method calls, e.g., `Logger::instance().Info().Str("key", "value").Msg("message")` —and utilizes RAII-based locking with mutexes to prevent interleaved outputs from parallel threads. There is also support for multiple log levels (Debug, Info, Warn, Error) via a `log_level` configuration.

5 Reflection on Coding Assistants

5.1 Usefulness

I utilized Claude Code and Gemini 3 for the coursework, and they proved highly effective. They excelled at producing boilerplate code for mathematical operations (vectors, matrix transformations) and observability features like the tqdm-inspired progress bar and the zerolog-inspired logging system. For complex algorithms (BVH builder, torus quartic solver), they provided a solid starting point, though refinement was necessary.
The most helpful aspect was providing renders to the AI assistant and asking it to identify differences in the images. This helped me discover minute details about BSDF and physically-based rendering, such as why my colors didn't match the expected renders or why the torus intersection was producing artifacts. This expedited debugging, as I knew exactly where to look for bugs.
Finally, AI was most useful for making code more modular. Until Module 3, all my code was in one directory of `.h` and `.cpp` files, with all shapes using a single file for intersection. With a plan and Claude Code's help, I refactored the code and reached a working state in approximately 2 hours (a task which otherwise would have taken close to a day).

5.2 Weaknesses

These AI systems proved poor at understanding numbers and showed significant limitations when I was debugging the torus artifact bug. Additionally, during OpenMP integration, Claude Code introduced a bug where the BVH root node was different across threads, resulting in images containing only the background color. This bug was especially difficult to debug because the BVH builder was working correctly and producing the expected output, but the BVH nodes passed to the raytracer were empty.

5.3 Example of Prompts

5.3.1 Torus Intersection

> *Implement a ray-torus intersection in C++ using surface equation. The torus has the following struct:*
>
> ```
> struct Torus : Shape {
>   Point location;
>   Point rotation;
>   Vec3 scale{1.0, 1.0, 1.0};
>   double major_radius{};
>   double minor_radius{};
> };
> ```
>
> *We need to return the closest intersection point and normal vector at the point of intersection.*

The generated code implemented the quartic equation and Ferrari's method using `float` precision. The following modifications were required:

1. Changed all `float` to `long double` for numerical stability.
2. Added Newton-Raphson refinement to achieve better tolerances.
3. Added NaN checks.

5.3.2 BVH Construction

> *Implement a BVH for ray tracing acceleration. Use AABB for bounding boxes.*

Basic BVH recursive partitioning and AABB intersection were generated. However, I had to add temporal bounding boxes for motion blur, configuration for `MAX_LEAF_SIZE` and `MAX_DEPTH` parameters, and BVH statistics for debugging.

5.3.3 OpenMP Parallelization

> *Write a thread-safe C++17 structured logger class that mimics the API style of Golang's 'zerolog'. It should use a fluent, chainable interface (e.g.,* `log.Info().Str("key", "val").Msg("message")` *). Requirements:*
>
> - *Levels: Support Debug, Info, Warn, Error.*
> - *No External Deps: Use only the C++ Standard Library.*

The result was a mostly correct logger, but the output was in JSON format (the default for zerolog), and the implementation was not thread-safe. I had to make changes to ensure thread safety and format the output as: `[TIME] [LEVEL] Message key=value key=value`.