Raytracer Report: CGR 2025

Student ID: s2286795

December 3, 2025

1 Implementation Summary

This report documents a modular C++ raytracer implementing Whitted-style recursion, distributed raytracing, BVH acceleration structure and many other features. The system was developed on macOS 15.5 (Apple Silicon, Arm64 architecture) using Apple Clang-17.0.0.

2 Build and Run

Only run the first two commands to build the project for now. The rest of the commands can be found in Section 5.

1. Within the folder s2286795, navigate to Code/build by running:

   ```
   cd Code/build
   ```

2. run cmake and make to build:

   ```
   cmake .. && make
   ```

3. Then, you can change the blend files in Blend folder by running:

   ```
   cmake -D BLEND_FILES="../../Blend/motion_blur.blend" .
   ```

4. And then, export the blend file to a json file in ASCII by running:

   ```
   make export
   ```

5. The last step, is to start rendering:

   ```
   ./Raytracer -input scene.json -output output.ppm -bvh -s 4 -light_sample 1
   ```

   All the generated Images appeared in this report can be found in the Output folder with its corresponding name but I also included the full commands attached for every scene mentioned in this report so that you can easily verify. *(if any of the command line provided failed to run, please email me at s2286795@ed.ac.uk)*

3 Visual Demonstration & Evaluation

3.1 Example Test Scene Comparison

Below is a side-by-side comparison of the implemented Whitted-style raytracer against the Blender Ground Truth.

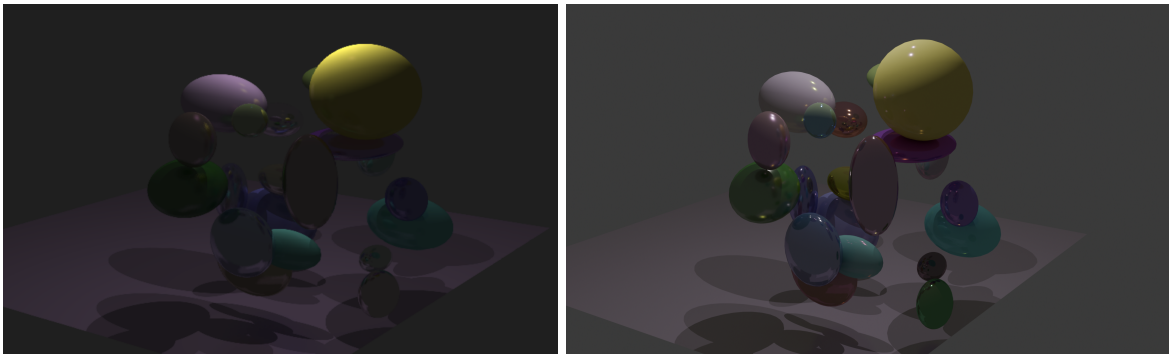

Figure 1: Test Scene 1 Left: My test scene. Right: Blender

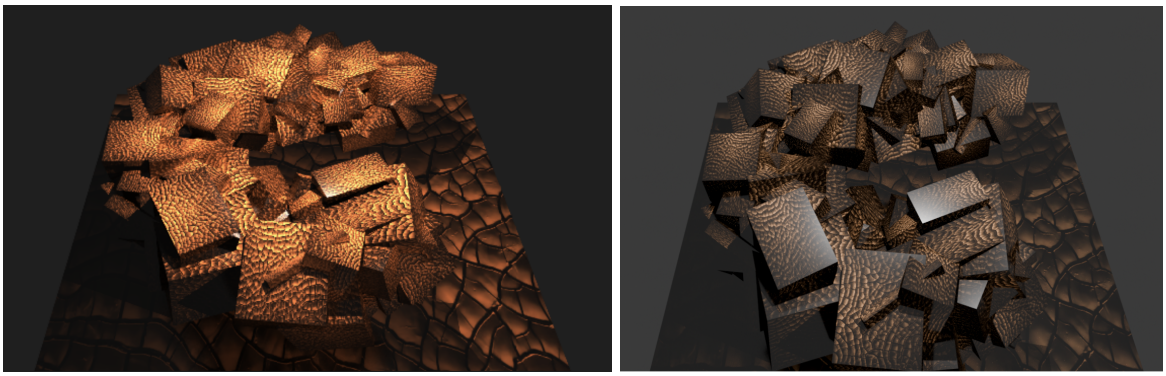

Figure 2: Test Scene 2 Left: My test scene. Right: Blender

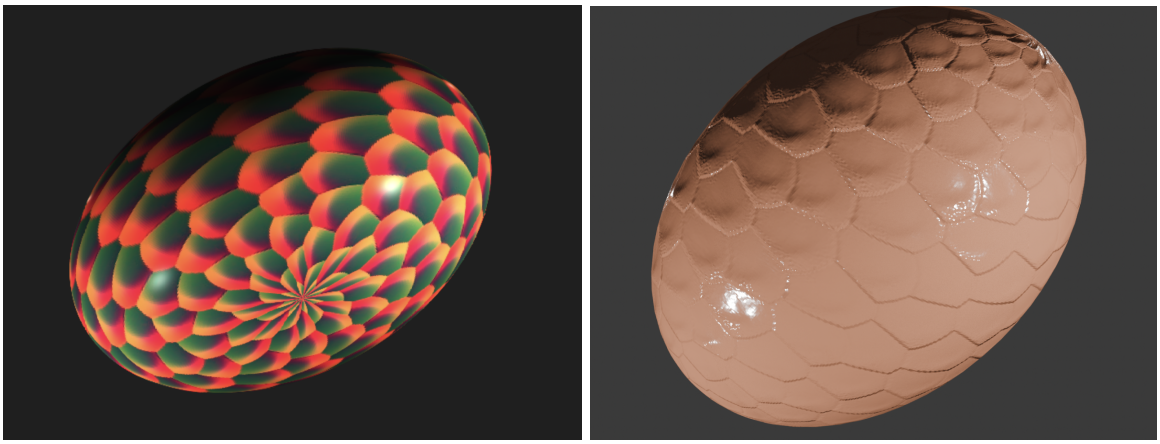

Figure 3: Test Scene 3 Left: My test scene. Right: Blender

Test Scene 1: All the shapes are geometrically correct and the reflection, materials match well Despite being darker. Regeneration command in Section 5. **[See Command 5.1]**

Test Scene 2: My rendered image is brighter but all of the geometry match perfectly. Regeneration command in Section 5. **[See Command 5.2]**

Test Scene 3: I used a different texture. Regeneration command in Section 5. **[See Command 5.3]**

3.2 Feature Showcase

1. Whitted-style Raytracing: Foundation of the raytracer. It contains features including the Camera class, Image class, Shapes class and relevant subclasses.
Example Prompt: Generate a C++ member function for translation, rotation, and scale that calculates object_to_world and its inverse, world_to_object for the given shape class (I had to check and modify the maths from AI's answer).

2. Acceleration: A Bounding Volume Hierarchy (BVH) was implemented. Table 1 demonstrates the render time improvement on a complex scene with 100 spheres in Figure 4.

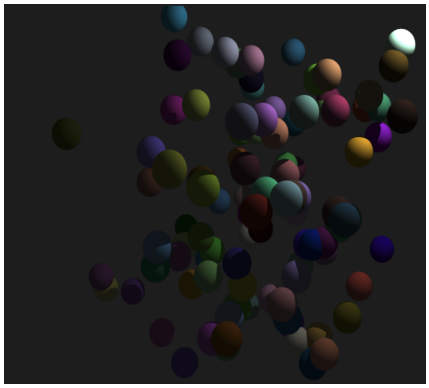

Figure 4: Blender scene with 100 spheres

Scene	Time (Linear)	Time (BVH)	Speedup Factor
Scene 1	98.07s	23.61s	4.15x

Table 1: Performance comparison demonstrating acceleration structure efficiency.

Example Prompts: This feature is entirely hand written.
Regeneration command in Section 5. **[See Command 5.4]**

3. Antialiasing (Stratified Sampling): Demonstration of edge smoothing using jittered sub-pixel sampling. When zoomed in, you can see the sawtooth-like edges and shadows on the cube generated with 1 sample per pixel. For both the 100 samples per pixel and Blender generated image, the edges are smooth.

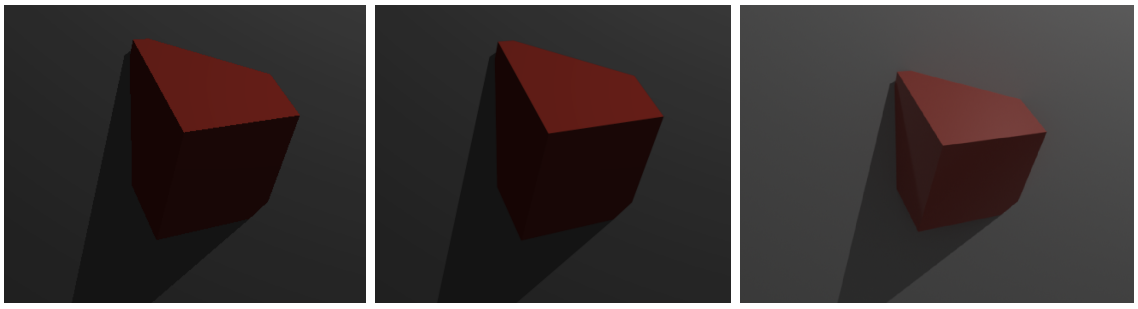

Figure 5: Left: 1 sample/pixel. Middle: 100 samples/pixel (Antialiased). Right: Blender

Example Prompt: could you refactor my code such that teh shade() function incorporates the antialiasing logic I implemented in main() to make it more modular.

Regeneration command in Section 5. **[See Command 5.5]**

4. Distributed Raytracing (Soft Shadows & Glossy Reflections): Soft shadows are implemented by modeling lights as area light sources and sampling multiple points across the light surface. When zoomed in, the effects were clearly visible in the shadows.

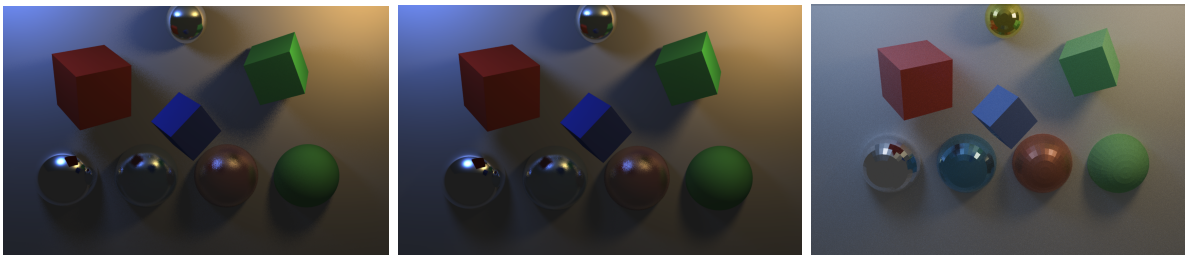

Figure 6: Soft shadows Left: 1 light sample. Middle: 16 light samples. Right: Blender

Example Prompt: Fully implemented by myself.
Regeneration command in Section 5. **[See Command 5.6]**

I implemented glossy reflections by randomly perturbing the reflection vector based on surface roughness, utilizing the stratified pixel sampling loop (controlled by the `-s` flag) to average these jittered rays into a smooth result.

Figure 7: Glossy reflection Left: 1 sample/pixel. Middle: 100 samples/pixel. Right: Blender

Example Prompt: Refactor the shade() function to implement glossy reflections by perturbing the reflection vector based on roughness for each pixel sample.
Regeneration command in Section 5. **[See Command 5.7]**

5. Lens Effect(Depth of Field & Motion Blur): Demonstration of depth of field with a focal distance set to the center sphere.

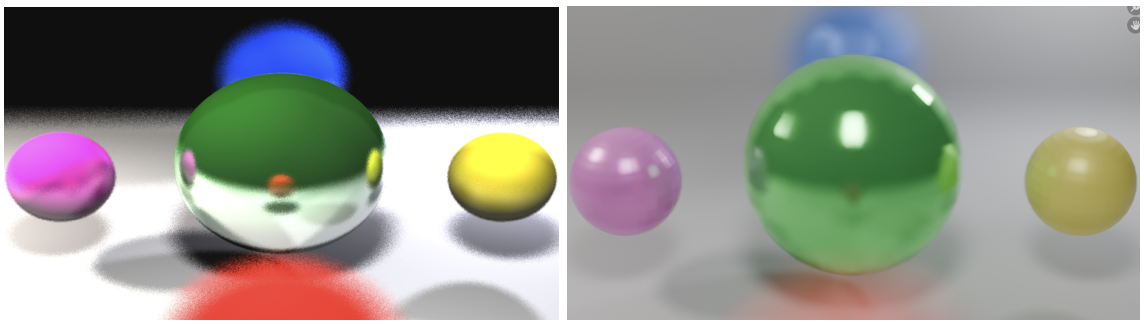

Figure 8: Depth of Field Left: My implementation. Right: Blender

Example Prompt: Update camera.cpp to support finite aperture and focal distance parameters.

Regeneration command in Section 5. **[See Command 5.8]**

Demonstration of motion blur where the blue sphere is still, orange sphere is moving fast horizontally and the yellow sphere is moving downwards slowly (There are two light sources).

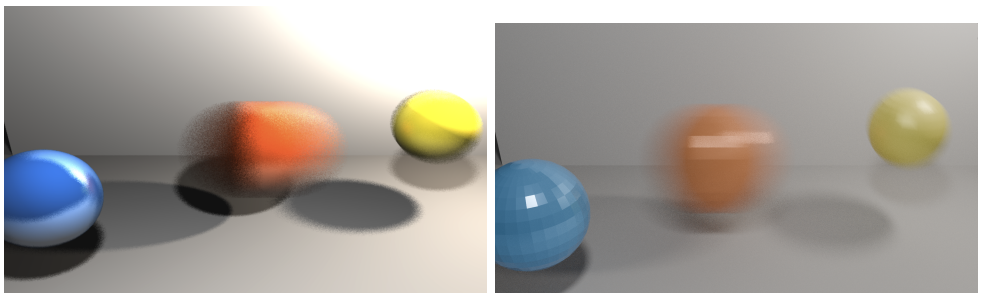

Figure 9: Motion Blur Left: My Implementaion. Right: Blender

Example Prompt: Modify the Sphere::intersection() function such that the ray is moved backwards from the velocity direction to achieve motion blur.

Regeneration command in Section 5. **[See Command 5.9]**

4 Coursework Administration

4.1 Timeliness & Deviations

4.1.1 Module 1: Camera Space & Blender Exporter

Changes: Since the checkpoint, the `Camera` class gained `aperture` and `focus_dist` parameters. The `Export.py` script was refactored to recursively "walk" backwards through material node connections.
Justification: The camera updates were essential for simulating thin-lens physics (Depth of Field). The exporter update was required to locate texture files hidden behind intermediate nodes (e.g., Mix/Math) for Module 3.

4.1.2 Module 2: Ray Intersection & Acceleration

Changes: In `shapes.cpp`, matrix logic was generalized to the base class, and the `Sphere` was refactored to intersect in object space. A `velocity` attribute was also added.
Justification: Object-space intersection allows non-uniform scaling (e.g., ovals) without distinct classes. The velocity attribute is a prerequisite for Motion Blur.

4.1.3 Module 3: Whitted-style Raytracing

Changes: The `shade` function was updated for stochastic light sampling, and `Trace` now perturbs reflection vectors based on roughness.
Justification: These modifications were necessary to upgrade the standard Whitted renderer to support Distributed Raytracing features (Soft Shadows and Glossy Reflections).

4.1.4 Argument for Timeliness Bonus

I submitted functional code for all three modules by their respective deadlines. The deviations described above were not corrections for errors, but necessary architectural extensions to support advanced "Final Raytracer" features like Motion Blur and Distributed RT. As the core specifications were met on time and subsequent changes were strictly for system expansion, the timeliness bonus is justified.

4.2 Generative AI Usage

Generative AI tools were utilized throughout the development of this raytracer, serving as an efficiency booster for boilerplate generation. The tools were particularly effective for standardizing tasks such as setting up the initial `PPM` image writer and outlining the structure for the JSON parsing logic. This accelerated the initial setup phase.

However, the utility of these assistants was heavily dependent on the precision of the prompts. I observed that vague instructions often yielded code that did not align with the specific modular architecture. For instance, when generating intersection routines, the AI initially defaulted to world-space geometric formulas, ignoring the required object-space transformation logic. To correct this, I had to construct prompts explicitly explaining the matrix implementation.

Furthermore, the generated output rarely functioned correctly without modification (Maybe my AI model isn't powerful enough). Almost every snippet required a little bit fine-tuning, integration, and debugging. I had to verify the vector math—particularly for the soft shadow sampling and glossy reflections—to ensure it adhered to the my lighting models. Ultimately, while generative AI significantly improved workflow efficiency, it could not replace a fundamental understanding of the material, serving best as a drafting tool that required human oversight.

4.3 Completion Table

Table 2: Project Completion Status

Module	**Topic**	**Marks**	**% Completed**
Module 1	Blender Exporter	1	100%
	Camera Space	4	100%
	Image R/W	1	100%
Module 2	Ray Intersection	4	100%
	Acceleration (BVH)	4	100%
Module 3	Whitted-style Raytracing	8	100%
	Antialiasing	2	100%
	Textures	2	100%
Final	System Integration	4	100%
	Distributed RT	8	100%
	Lens Effects	4	100%
Other	Report	8	100%
	Exceptionalism	10	0%
	Timeliness Bonus	20	100%
Total		**80**	**-**

5 Raytracer Command Line Examples

2.1. Example Test Scene Comparison

Command 5.1: Test Scene 1 Rendering

```
cmake -D BLEND_FILES="../../Blend/Test1.blend" .
make export
./Raytracer -input scene.json -output test1.ppm -bvh -s 5 -light_sample 16
```

Command 5.2: Test Scene 2 Rendering

```
cmake -D BLEND_FILES="../../Blend/Test2.blend" .
make export
./Raytracer -input scene.json -output test2.ppm -bvh -s 4 -light_sample 16
```

Command 5.3: Test Scene 3 Rendering

```
cmake -D BLEND_FILES="../../Blend/Test3.blend" .
make export
./Raytracer -input scene.json -output test3.ppm -bvh -s 4 -light_sample 16
```

3.2. Feature Showcase

Command 5.4: BVH Performance Verification

```
cmake -D BLEND_FILES="../../Blend/BVH.blend" .
make export
```

And then for BVH absent (Linear) and BVH present rendering:

```
./Raytracer -input scene.json -output bvh.ppm -s 4 -light_sample 1
./Raytracer -input scene.json -output bvh.ppm -bvh -s 4 -light_sample 1
```

Command 5.5: Antialiasing (1 vs 100 samples/pixel)

```
cmake -D BLEND_FILES="../../Blend/Antialiasing.blend" .
make export
```

Then, for 1 sample/pixel and 100 samples/pixel, run respectively:

```
./Raytracer -input scene.json -output antialiasing_1.ppm -bvh -s 1 -light_sample 1
./Raytracer -input scene.json -output antialiasing_100.ppm -bvh -s 10 -light_sample 1
```

Command 5.6: Distributed Raytracing (Soft Shadows)

```
cmake -D BLEND_FILES="../../Blend/Distributed.blend" .
make export
```

Then, for 1 light sample and 16 light samples, run respectively:

```
./Raytracer -input scene.json -output softshadows_1.ppm -bvh -s 3 -light_sample 1
./Raytracer -input scene.json -output softshadows_16.ppm -bvh -s 3 -light_sample 16
```

Command 5.7: Distributed Raytracing (Glossy Reflections)

```
cmake -D BLEND_FILES="../../Blend/glossy_reflection.blend" .
make export
```

Then, for 1 sample per pixel and 100 samples per pixel, run respectively:

```
./Raytracer -input scene.json -output glossy_reflection_1.ppm -bvh -s 1 -light_sample
    1
./Raytracer -input scene.json -output glossy_reflection_100.ppm -bvh -s 10 -
    light_sample 1
```

Command 5.8: Lens Effect (Depth of Field)

```
cmake -D BLEND_FILES="../../Blend/dop.blend" .
make export
./Raytracer -input scene.json -output dof.ppm -bvh -s 4 -light_sample 1
```

Command 5.9: Lens Effect (Motion Blur)

```
cmake -D BLEND_FILES="../../Blend/motion_blur.blend" .
make export
./Raytracer -input scene.json -output motion_blur.ppm -bvh -s 4 -light_sample 1
```