CGR 2025 Raytracer Report

Student ID: s2288598

1 Introduction

This report summarises the implementation of a modular Whitted-style raytracer written in C++ for the CGR 2025 coursework.

2 System Overview

The raytracer is built using the provided Makefile. To compile the raytracer, run

```
make raytracer
```

To render a scene with filename `scene.txt`, run

```
./raytracer -i scene.txt
```

The renderer has the following runtime options:

- `-i <file>` – input ASCII scene file.
- `-o <file>` – output PPM image.
- `-w, -h` – override output resolution.
- `-spp <int>` – antialiasing with N samples per pixel.
- `-d <int>` – maximum recursion depth.
- `-no-bvh` – disable BVH acceleration.
- `-no-shading` – disable shading.
- `--shadow-samples <int>` – soft shadows using N shadow rays.
- `--glossy-samples <int>` – glossy reflections using N rays.
- `-exposure <float>` – exposure multiplier.
- `-tonemap <mode>` – set tone mapping operator (`none`, `reinhard`, `aces`).
- `--help`.

Rendered scenes are saved to the `Output` folder, Blend files used are located in the `Blend` folder, textures used are in the `Textures` folder (both as `.png` and `.ppm`) and meshes used are in the `Meshes` folder.

3 Module 1

All components implemented in Module 1 are validated through the scenes and comparisons shown for later modules.

3.1 Blender Exporter

I implemented a Python exporter (`export.py`) that converts Blender scenes into a custom ASCII format.

3.2 Camera Space Transformations

I implemented the camera class used throughout the raytracer. A test file reads camera information for `Test1.txt`, displays the camera information, and calculates rays to the corner pixels. To run the test, run

```
./Tests/test_camera
```

3.3 Image Read and Write

I implemented an `Image` class (`image.h/cpp`) capable of reading and writing ASCII PPM (`P3`) files. A test file confirms that PPM images are correctly read, manipulated and written. To run the test, run

```
./Tests/test_image
```

4 Module 2

4.1 Ray Intersection

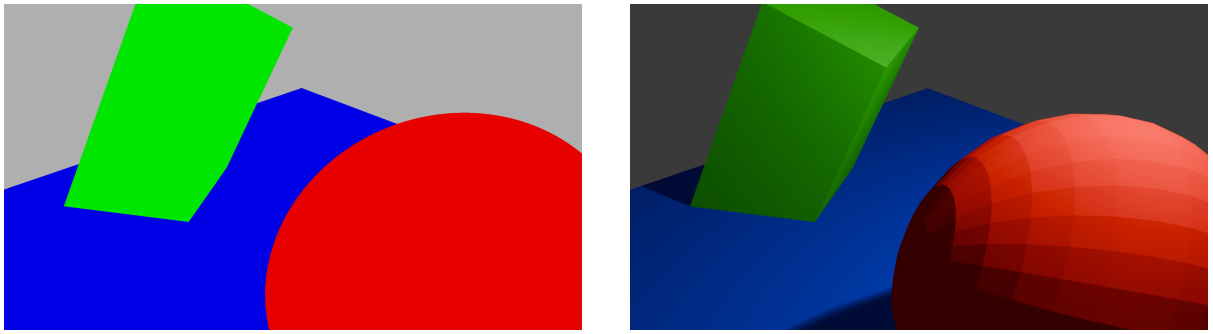
I implemented ray intersection routines for spheres, cubes, planes and triangles. To test this, I compared my render with no shading to Blender's render.

Intersection – Fig. 1

Blend file: `intersect.blend`

Raytracer commands:

```
./raytracer -i intersect.txt -o intersect.ppm -no-shading
```



(a) Raytracer output (b) Blender reference render

Figure 1: Comparison between my raytracer and Blender for the ray intersection test scene.

4.2 Acceleration Hierarchy

I implemented a BVH (Bounding Volume Hierarchy) acceleration structure, which reduces ray intersection cost from $O(n)$ to $O(\log n)$.

To test, the renderer was run twice: once with the BVH enabled and once with it disabled.

BVH – Fig. 2

Blend file: `manyspheres.blend`

Raytracer commands:

```
./raytracer -i manyspheres.txt -o manyspheres.ppm -exposure 0.1
./raytracer -i manyspheres.txt -o manyspheres.ppm -exposure 0.1 -no-bvh
```

The measured times were

Configuration	Render Time (s)
BVH enabled	4.79
BVH disabled	47.48

Turning BVH on results in an approximately **10× speedup** for this scene.

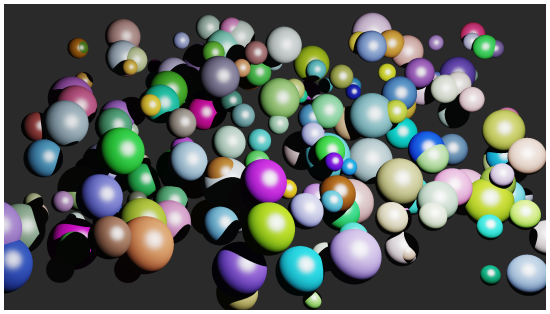


Figure 2: BVH test scene containing 200 spheres.

5 Module 3

5.1 Whitted-style Raytracing

5.1.1 Blinn–Phong Shading

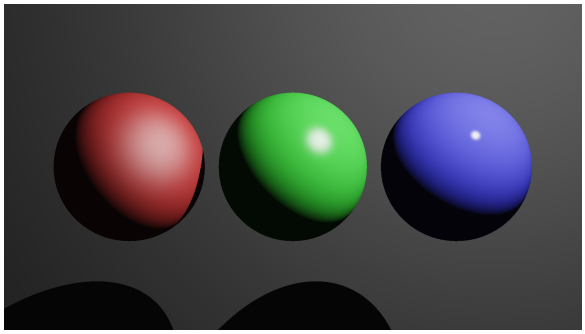
I implemented Blinn–Phong shading. To demonstrate it, I constructed a scene with 3 spheres each with different material properties: rubber (red), plastic (green) and polished (blue).

Blinn-Phong Shading – Fig. 3.

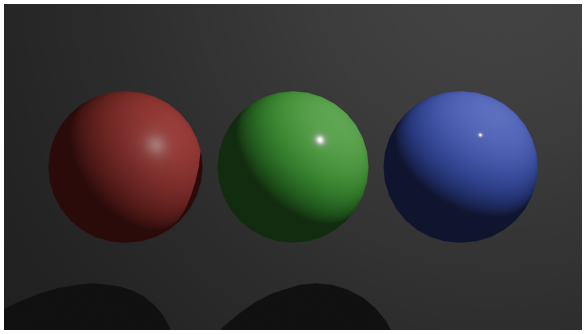
Blend file: `blinnphong.blend`

Raytracer commands:

```
./raytracer -i blinnphong.txt -o blinnphong.ppm -exposure 0.2
```



(a) Raytracer render



(b) Blender render

Figure 3: Comparison between raytracer and Blender Blinn–Phong shading.

5.1.2 Reflection

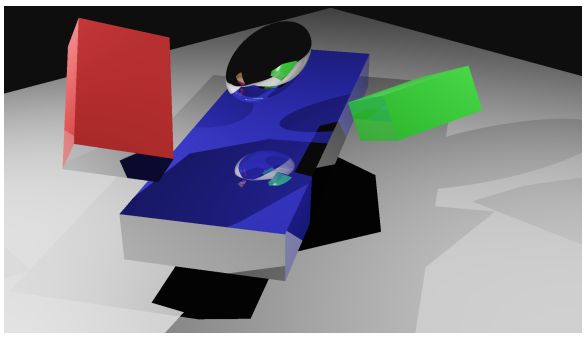
I implemented reflections. To test, I rendered a scene with various reflective objects and compared to Blender's render.

Reflection – Fig. 4

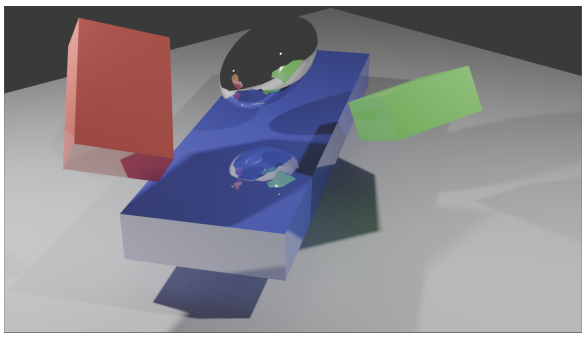
Blend file: `reflection.blend`

Raytracer commands:

```
./raytracer -i reflection.txt -o reflection.ppm -exposure 0.02
```



(a) Raytracer render



(b) Blender render

Figure 4: Comparison between raytracer and Blender reflections.

5.1.3 Refraction

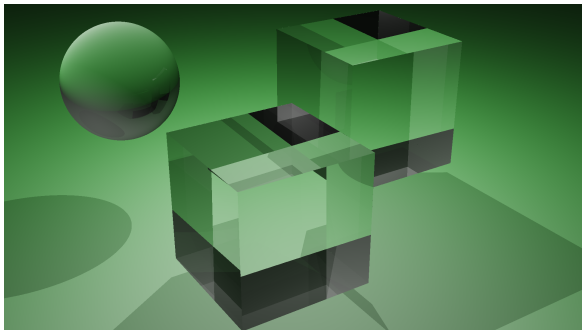
I implemented refraction. To test, I rendered a scene with glass objects and compared to Blender's render.

Refraction – Fig. 5

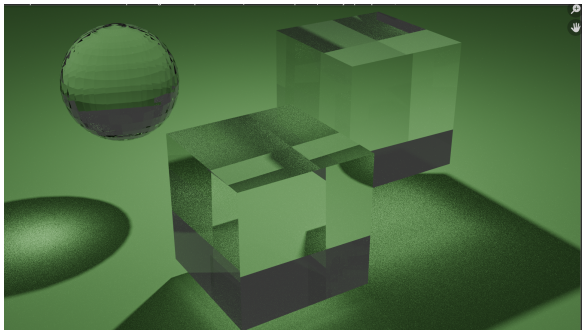
Blend file: `glass2.blend`

Raytracer commands:

```
./raytracer -i glass2.txt -o glass2.ppm -exposure 0.02 --spp 8
```



(a) Raytracer render



(b) Blender render

Figure 5: Comparison between raytracer and Blender refractions.

5.2 Antialiasing

I implemented stratified supersampling antialiasing, where each pixel is subdivided into a jittered grid of rays. Increasing the number of samples reduces jagged edges and improves image smoothness.

Antialiasing

Blend file: `spheres.blend`

Raytracer commands:

```
./raytracer -i spheres.txt -o noaa.ppm -exposure 0.1
./raytracer -i spheres.txt -o aa.ppm -exposure 0.1 -spp 8
```

(a) AA off

(b) AA on

Figure 6: Effects of antialiasing.

5.3 Textures

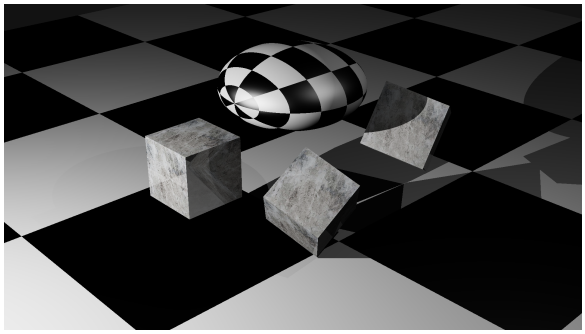
I implemented texture mapping from `.ppm` files.

Texture – Fig. 7.

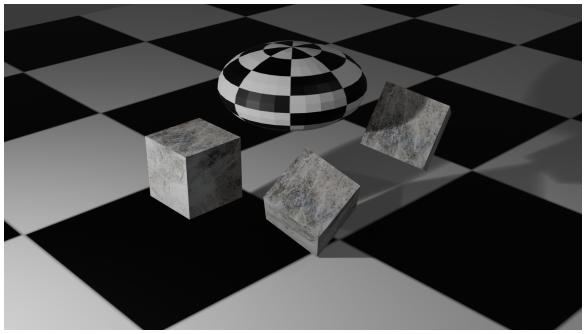
Blend file: `texture.blend`

Raytracer commands:

```
./raytracer -i texture.txt -o texture.ppm -exposure 0.03
```



(a) Raytracer render



(b) Blender render

Figure 7: Comparison between raytracer and Blender textures.

6 Final Features

6.1 System Integration

I combined all the modules in modular way and added command line arguments to toggle features. Run

```
./raytracer --help
```

to see.

6.2 Distributed Raytracing

I implemented soft shadows and glossy reflections.

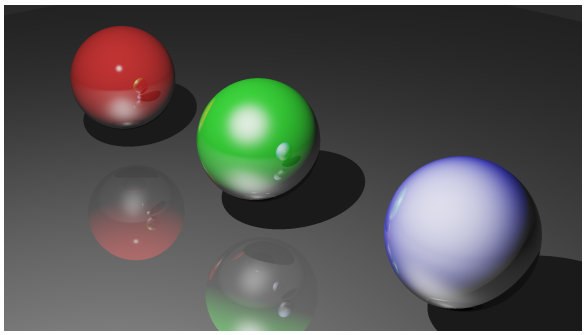
6.2.1 Soft Shadows

Soft Shadows – Fig. 8

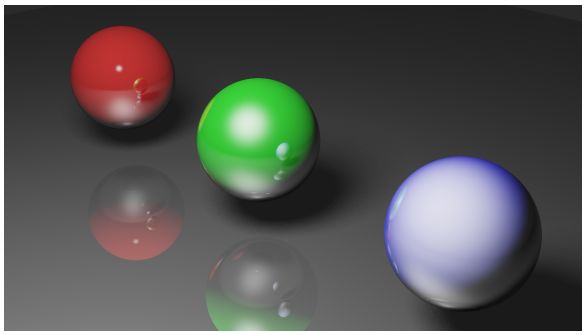
Blend file: `spheres.blend`

Raytracer commands:

```
./raytracer -i spheres.txt -o spheres_plain.ppm -exposure 0.1
./raytracer -i spheres.txt -o spheres_soft.ppm -exposure 0.1
   --shadow-samples 16
```



(a) Normal shadows



(b) Soft shadows

Figure 8: Comparison between normal and soft shadows.

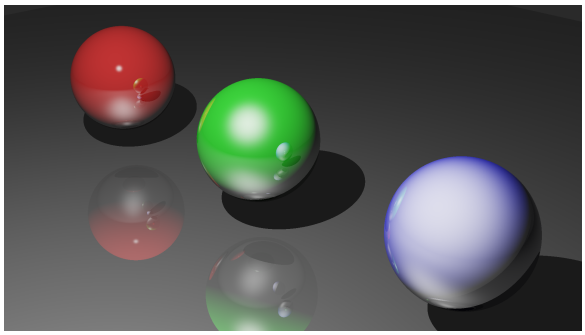
6.2.2 Glossy Reflections

Glossy Reflections – Fig. 9

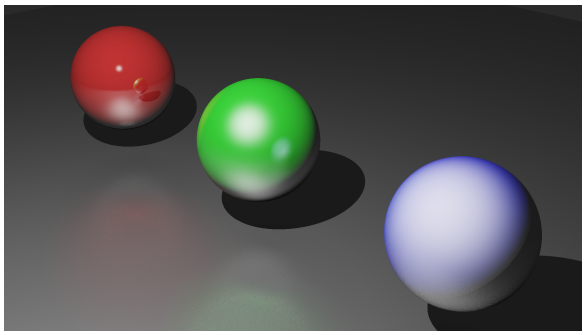
Blend file: `spheres.blend`

Raytracer commands:

```
./raytracer -i spheres.txt -o spheres_plain.ppm -exposure 0.1
./raytracer -i spheres.txt -o spheres_gloss.ppm -exposure 0.1
   --glossy-samples 16
```



(a) Normal reflections



(b) Glossy reflections

Figure 9: Comparison between normal and glossy reflections.

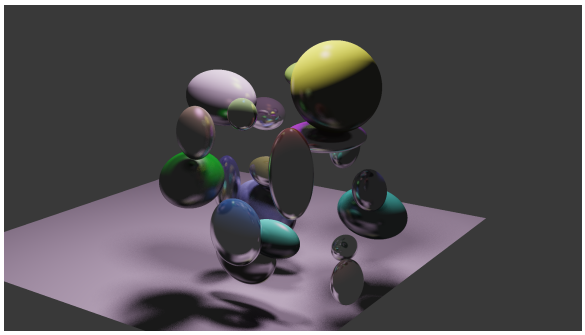
6.2.3 Both

Distributed Raytracing (Both) – Fig. 10

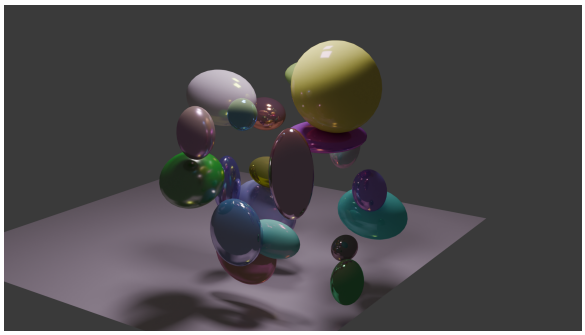
Blend file: `test1.blend`

Raytracer commands:

```
./raytracer -i test1.txt -o test1.ppm -exposure 0.15
    --shadow-samples 16 --glossy-samples 16
```



(a) Raytracer render



(b) Blender render

Figure 10: Comparison between raytracer and Blender distributed effects.

6.3 Lens Effects

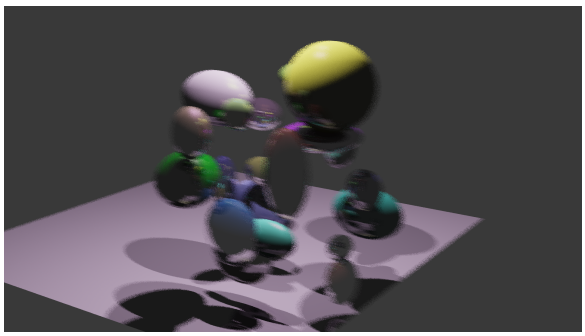
I implemented motion blur and depth of field. Motion blur is controlled by the camera's velocity vector property, and depth of field is controlled by the camera's aperture and focal distance properties.

Lens Effects – Fig. 11

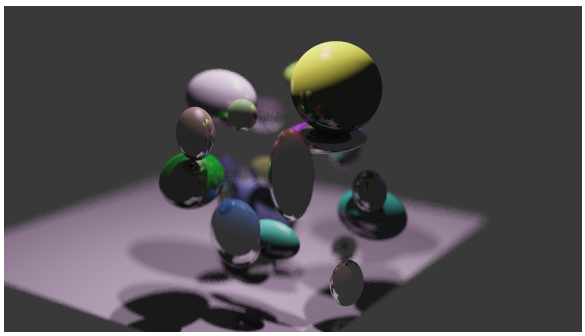
Blend files: Test1_motionblur.blend, Test1_dof.blend

Raytracer commands:

```
./raytracer -i test1_motionblur.txt -o motionblur.ppm -exposure 0.15 -spp 4
./raytracer -i test1_dof.txt -o dof.ppm -exposure 0.15 -spp 4
```



(a) Motion blur



(b) Depth of field

Figure 11: Lens effects.

7 Exceptionalism

7.1 Tone Mapping, Gamma Correction and Exposure Control

I implemented ACES and Reinhard tone mapping, gamma correction and exposure control. This gave me control over the lighting and colour of the render.

7.2 Triangle Meshes

I implemented support for rendering arbitrary triangle meshes. I did not implement exporting meshes from `.blend` files. The mesh is read from `Meshes/sculpture.obj`.

Triangle Meshes – Fig. 12

Blend file: `N/A`

Raytracer commands:

```
./raytracer -i meshtest.txt -o meshtest.ppm -exposure 0.2
```

Figure 12: Mesh composed of 7798 triangles, with marble texture.

8 Miscellaneous

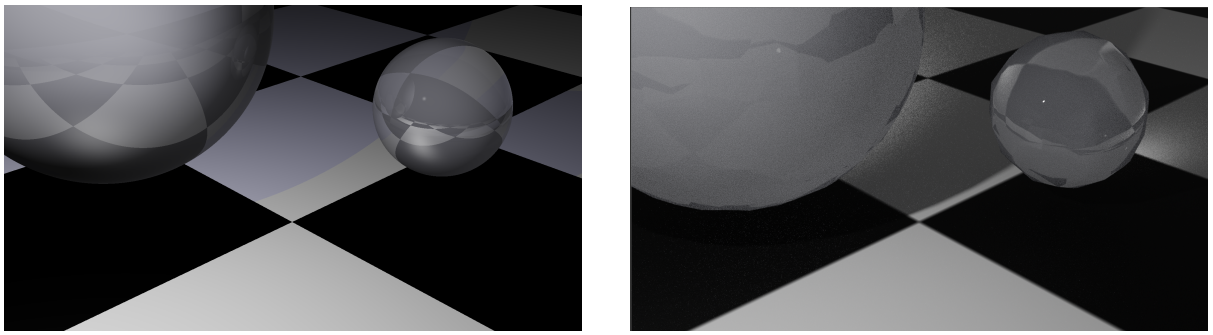
This section contains some extra scenes that combine multiple features at the same time.

Refraction – Fig. 13

Blend file: `glass.blend`

Raytracer commands:

```
./raytracer -i glass.txt -o glass.ppm -exposure 0.02
```



(a) Raytracer render (b) Blender render

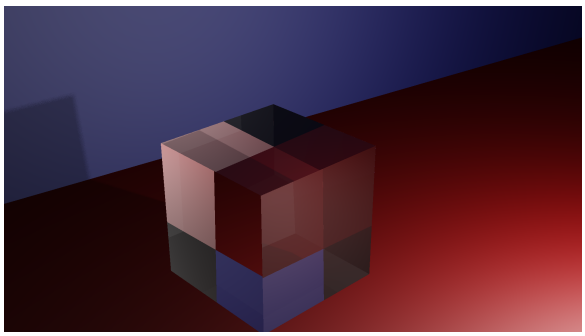
Figure 13: Comparison between raytracer and Blender refractions.

Refraction (Soft Shadows) – Fig. 14

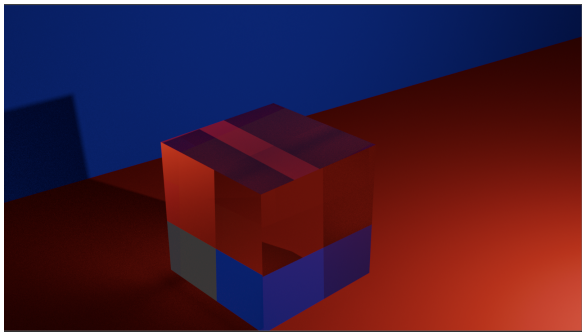
Blend file: `glass.blend`

Raytracer commands:

```
./raytracer -i glass3.txt -o glass3.ppm -exposure 0.01 --shadow-samples 4
```



(a) Raytracer render



(b) Blender render

Figure 14: Comparison between raytracer and Blender refractions with soft shadows.

9 Timeliness Bonus

For the timeliness bonus, I submitted Modules 1, 2 and 3 on time, and kept the general structure the same all throughout the assignment. Changes were only made to improve the modularity of the codebase or fix bugs.

9.1 Module 1

The most major deviation from this module is moving the vector and ray structs from `camera.h` to `maths.h`. As the number of vector and maths functions grew, I felt it was necessary to create a dedicated maths header. A slight issue with aspect ratio in `pixelToRay` was also fixed.

Other than the implementation of new features, the structure of the other files remains pretty much unchanged.

9.2 Module 2

The only major deviation between this module and the final submission is that the raytracing logic was moved from `main.cpp` to `raytracer.cpp`.

9.3 Module 3

Changes were made to `raytracer.cpp`. The code for antialiasing was changed to use stratified sampling. Shadows for transparent objects were improved, adding transmissive shadows through glass. Shadow logic was made more modular by adding a dedicated function `computeShadowFactor`.

10 Thoughts on the use of Coding Assistants

I used AI coding assistants throughout this assignment. One of their strengths is that they can save a lot of time on menial tasks: for example, generating Blender scripts, writing boilerplate code, or summarising the strengths and weaknesses of coding assistants. They can complete (smaller) chunks of this assignment in one prompt, which is very helpful under time pressure. To me, their greatest strength is using them to explain new concepts. For instance, implementing BVH would have taken me a lot longer if I had to first scour textbooks about how it works. The assistant helped me understand the idea before writing the code.

However, they also have notable limitations. It's very easy to trust generated code without understanding it. I spent hours trying to fix a bug caused by code I had copied and pasted without reading. The biggest downside is, in my opinion, that using it too much reduces how

much you understand the content. From experience, if I work through a problem myself I retain a much deeper understanding than if I ask ChatGPT to explain it to me. To me, there doesn't seem to be any way to avoid the use of coding assistants in the future, especially in take-home assignments.

11 Evaluation

Table 1: Summary of Completed Coursework Features

Feature	**Status**
Module 1	
Blender Exporter	Completed
Camera Space Transformations	Completed
Image Read/Write (PPM)	Completed
Module 2	
Ray Intersection (Sphere, Cube, Plane, Triangle)	Completed
Acceleration Structure (BVH)	Completed
Module 3	
Whitted-Style Raytracing	Completed
Antialiasing	Completed
Texture Mapping	Completed
Final Raytracer	
System Integration	Completed
Distributed Raytracing (Soft Shadows, Glossy)	Completed
Lens Effects (Motion Blur, Depth of Field)	Completed
Exceptionalism	
Tone Mapping / Gamma Correction	Completed
Triangle Mesh Rendering	Completed
Summary of Strengths and Weaknesses of AI Coding Assistants	Completed