CGR 2025 Raytracer Coursework Report

Student ID: s2289730

December 4, 2025

Contents

1 Introduction

1.1 Overview

This report documents the design and implementation of a modular Whitted-style raytracer written entirely in C++, following the specification for CGR 2025. The renderer supports:

- Blender scene export via a Python script,
- a full pinhole camera model,
- ray–primitive intersection routines (sphere, cube, plane),
- an acceleration structure (BVH),
- Whitted-style recursive reflection and refraction,
- antialiasing,
- texture mapping,
- soft shadows, glossy reflections, depth of field, and motion blur,
- build/run control via command-line flags.

1.2 Marker Instructions

A condensed summary of every command needed by the marker. You can simply make run to run everything altogether. However, I recommend trying out each function indicidually.

To change the scene, change ASCII/current.json to one of the test.json files within the Blend folder.

You can use Export.py to generate the test json files if you wish to do so

Output can be found within the Output folder under output.ppm

- `make run_no_lens` (runs all effects without lens effects (recommended))
- `make` (builds)
- `./raytracer --textures / --no-textures`
- `./raytracer --aa / --no-aa`
- `./raytracer --soft-shadows / --no-soft-shadows`
- `./raytracer --glossy / --no-glossy`
- `./raytracer --dof / --no-dof`
- `./raytracer --motion-blur / --no-motion-blur`

If you don't want to read the details of the report, each image represented has a caption with instructions on how to generate it so you can just check those.

1.3 Task 1: Blender Exporter

The Blender export script gathers all relevant scene information and writes it to `export.json`, which the C++ ray tracer later consumes. Cameras, lights, spheres, cubes and planes are inspected directly from Blender's object data, and their essential properties are serialised into a simple ASCII JSON format.

1.3.1 Camera Export

For each camera, the script extracts its world position, gaze direction (derived from Blender's $-\mathbf{Z}$ axis), up vector ($\mathbf{Y}$), focal length, sensor size and the film resolution. These are computed from the camera's world transform:

$$\mathbf{gaze} = R(0, 0, -1), \qquad \mathbf{up} = R(0, 1, 0),$$

ensuring the external ray tracer reconstructs an equivalent camera model.

1.3.2 Point Lights, Spheres, Cubes, and Planes

Point lights are exported with their world–space position and radiant intensity. Spheres are identified by name and written out with their centre, scale (allowing ellipsoids in C++), and material. Cubes include translation, rotation and scale so that the ray tracer can rebuild an oriented box. Planes are exported by transforming the mesh vertices to world space and writing out the four corners of the quad.

1.3.3 Material Extraction

The exporter inspects common Blender node setups (Principled BSDF, Diffuse/Glossy stacks, Glass/Refraction nodes, and texture inputs). These are converted into a Whitted-style material with parameters

$$(k_d, k_s, \text{shininess}, \text{reflectivity}, \text{transmissive}, \text{ior}),$$

including a custom mapping from Blender roughness to a Blinn–Phong exponent:

$$\alpha = \max(2,\ \min(512,\ 2/r^2 - 2)).$$

1.4 Task 2: Camera Space Transformations

The `Camera` class loads the exported parameters, constructs an orthonormal basis and converts pixel coordinates into world–space rays. Once the JSON block is located, the camera reads its position, forward and up vectors, focal length and sensor dimensions. The basis vectors are then formed as:

$$\mathbf{f} = \frac{\mathbf{forward}}{\|\mathbf{forward}\|}, \qquad \mathbf{r} = \frac{\mathbf{f} \times \mathbf{up}}{\|\mathbf{f} \times \mathbf{up}\|}, \qquad \mathbf{u} = \mathbf{r} \times \mathbf{f}.$$

A pixel (p_x, p_y) is mapped to a position on the film plane using the sensor size and resolution, and converted into a world–space direction:

$$\mathbf{d} = f\,\mathbf{f} + x_s\,\mathbf{r} + y_s\,\mathbf{u},$$

which is then normalised. The camera position is used as the ray origin.

1.5 Task 3: Image Read/Write (PPM)

The `Image` class provides minimal PPM (`P3`) support. It reads an image by parsing the header

$$\texttt{P3} \quad W \quad H \quad \texttt{maxval},$$

then loads per-pixel RGB values into a `std::vector`. Pixels are accessed through simple getters and setters, and the entire buffer can be written back out to a PPM file in the same format. This satisfies the assignment requirement to implement image I/O without external libraries.

1.6 PPM Image I/O: Testing

To test this feature, compile and run `./raytracer` to generate `Output/output.ppm`, then open it with any PPM viewer to confirm the pixel data is correct.

2 Module 2: Ray–Primitive Intersection and Acceleration

Module 2 introduces the core intersection routines for spheres, cubes and planes, and then builds an acceleration structure to reduce the number of ray–primitive tests. All primitives derive from a common `Shape` interface that provides an `intersect()` method and a `bounds()` method, and hits are reported through a compact `Hit` record storing t, position, normal, UVs and a pointer to the shape. This keeps the raytracer modular while allowing each primitive to implement its own geometry.

2.1 Sphere and Ellipsoid Intersections

The `Sphere` class covers both uniform spheres and ellipsoids via per-axis scale, and optionally motion through a velocity field. Rays are evaluated at `ray.time`, so a sphere's centre is treated as $\mathbf{c}(t) = \mathbf{c}_0 + t\mathbf{v}$. The ray and hit point are transformed into a unit-sphere space by subtracting the time-dependent centre and scaling by the inverse axis lengths, after which the standard quadratic form solves the intersection. Normals are derived from the ellipsoid gradient and UVs come from longitude–latitude mapping. Bounding boxes expand automatically to cover a moving sphere's entire swept volume.

2.2 Plane Quads

A plane quad is defined by its four world-space vertices and a precomputed normal. The ray first intersects the infinite plane; an inside test using edge cross products ensures that the point lies within the quad. Bilinear coordinates from one vertex give stable UVs, and the bounding box is simply the AABB of the four corners.

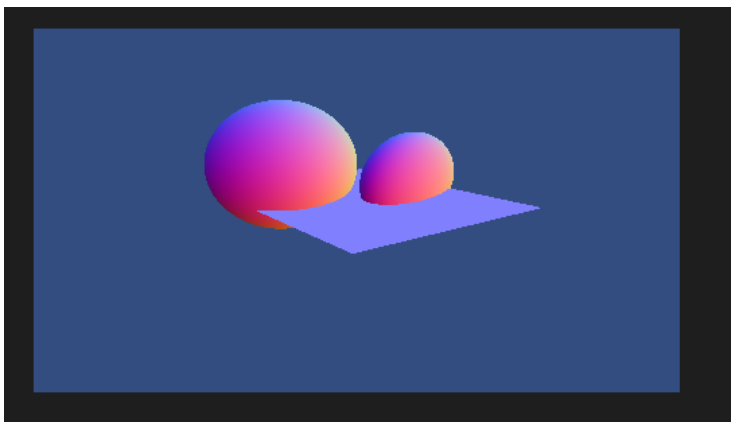

Figure 1: Example of ray intersections with spheres and a plane

2.3 Oriented Cubes

Cubes are represented as the canonical box $[-1, 1]^3$ in object space. A full TRS matrix is stored along with its inverse and inverse-transpose. The ray is transformed into object space, and intersection is solved via the slab method across the three axes. The hit normal is based on the dominant coordinate of the object-space hit point, then transformed back to world space. UVs come from remapping two coordinates to $[0, 1]$. The world-space bounding box is computed by transforming all eight canonical cube corners.

2.4 Bounding Boxes and BVH

The `AABB` class supports expansion by points or other boxes and implements a fast slab-based ray–box intersection. The BVH is stored as a flat array of nodes. During construction, each node accumulates the bounds of its primitives; small ranges form leaf nodes, while larger ranges are split along the longest axis using a median partition (`std::nth_element`). This yields a balanced hierarchy without requiring SAH.

Traversal is iterative: the root is pushed onto a small stack, each node is tested against the ray, and only intersected children are explored. Leaves perform direct ray–primitive calls, but the BVH prunes most of the scene quickly, greatly reducing work.

2.5 Testing and BVH Speedup

Two scenes were rendered twice—once with a naïve loop over all shapes and once using the BVH. Rendered images match exactly, but the BVH dramatically reduces runtime as scene complexity increases.

Scene	Brute Force (ms)	BVH (ms)	Speedup
9 shapes	544	327	1.66×
126 shapes	14003	2012	6.96×

Table 1: BVH performance improvement measured on two scenes.

These results show that even a simple median-split BVH offers substantial acceleration, especially as the number of objects grows.

3 Module 3: Whitted-Style Ray Tracing, Antialiasing and Textures

Module 3 turns the ray–geometry framework from Modules 1 and 2 into a full Whitted-style renderer. The key additions are: recursive evaluation of reflection and refraction, a Blinn–Phong shading model for direct lighting, supersampling antialiasing on the image plane, and basic texture mapping driven by the `(u,v)` coordinates computed in the intersection code. This section explains how these pieces fit together in the final raytracer.

3.1 Overall Shading Pipeline

The core of the renderer is the function `traceRay`, which returns a radiance value for a single ray. It first finds the closest intersection using either the BVH or a naïve loop over all shapes. If no shape is hit, the function returns a simple constant background colour. For a valid hit, `traceRay` looks up the material associated with the intersected `Shape` and computes three contributions:

1. Direct illumination from all point lights using a Blinn–Phong model.
2. Reflected radiance, which is added if the material has non-zero reflectivity.

3. Refracted radiance, which is added if the material is marked transmissive and has a non-trivial index of refraction.

Each recursive call carries a depth counter, and recursion is clamped at a maximum depth `kMaxDepth = 4` to avoid infinite loops in scenes with mutually reflective objects. A small offset along the surface normal (`kShadowBias`) is applied to all secondary rays so that they do not immediately self-intersect the emitting surface.

3.2 Blinn–Phong Illumination

Direct lighting is implemented in the helper function `shadeHit`. It takes the hit record, the view direction, the current ray time, the list of point lights, the material map and an optional texture pointer. For each light, the code constructs a shadow ray towards the light position (or a jittered position if soft shadows are enabled), tests for occlusion using the BVH, and skips the light if any object blocks this shadow ray.

If the light is visible, the local colour is computed using the Blinn–Phong model. The diffuse term is based on the cosine between the surface normal and the light direction, and is scaled by the material's diffuse reflectance `kd`. The specular term uses a half-vector between the light direction and the view direction, and is raised to a Blinn exponent `shininess`. Both components are modulated by an inverse-square falloff $\propto 1/r^2$ using the light's `radiant_intensity` field.

The lighting function is also responsible for combining material colours with texture samples. When textures are enabled and a valid `Texture` object is available, the shader samples the texture at the `(u,v)` coordinates stored in the hit record and treats this as an albedo. The effective diffuse colour is then the component-wise product of `kd` and this albedo. This approach preserves the material's overall intensity while letting the texture pattern modulate colour at the pixel level.

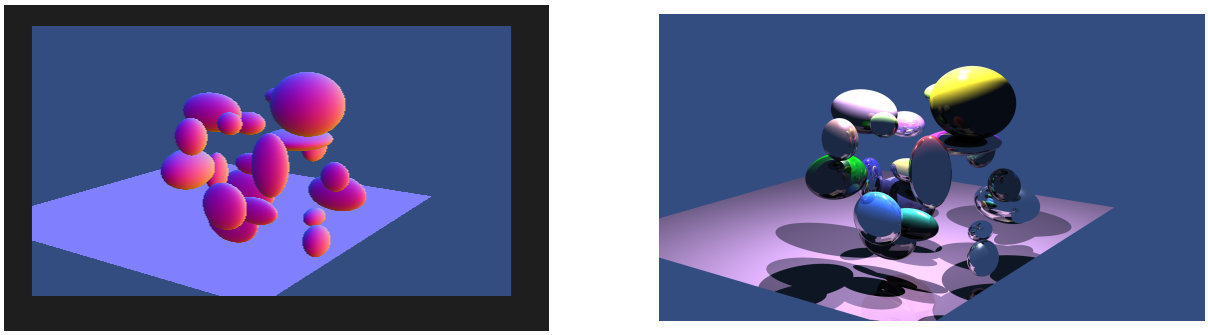

Figure 2: Comparison of a scene rendered without BP shading (left) and with BP shading (right)

3.3 Recursive Reflections and Refractions

Specular reflections and refractions are handled directly inside `traceRay`. After computing the local Blinn–Phong term, the code checks the material parameters `reflectivity` and `transmissive`. If `reflectivity` is greater than zero, it computes the ideal mirror direction using a standard reflection formula and then spawns one or more reflection rays depending on whether glossy reflections are enabled.

The reflection term is also where the "distributed ray tracing" behaviour begins to show up. For perfectly sharp mirrors, the code sends a single ray along the ideal reflection direction. When glossy reflections are enabled, the ideal direction is jittered in a small lobe using `jitter_reflection`. This helper samples a random vector inside a unit sphere, scales it by a roughness parameter and adds it to the ideal direction before normalising. The roughness is derived from the Blinn exponent: very shiny materials lead to a small deviation from the mirror direction, while low-shininess materials produce wider lobes. The average of all these jittered rays is

multiplied by the material `reflectivity` and added to the local colour, giving soft highlights and blurred reflections.

Refractions are handled in a similar spirit. The code computes the correct inside/outside configuration by comparing the incoming ray direction with the surface normal, swaps the indices of refraction if necessary, and then uses a small helper to implement Snell's law. If refraction is possible (i.e. we do not hit total internal reflection), a transmitted ray is spawned on the opposite side of the surface. A Fresnel term based on a Schlick approximation decides how much energy to allocate to reflection vs transmission. The existing local lighting and reflection contributions are then mixed with the transmitted radiance returned by the recursive call, using the material's `transmissive` value and the Fresnel coefficient.

3.4 Antialiasing on the Image Plane

Antialiasing is implemented by supersampling each pixel on a regular grid. A global toggle `g_enable_aa` controls whether the renderer uses a single ray per pixel or a multi-sample approach. When antialiasing is disabled, the code shoots exactly one ray through the centre of each pixel. When antialiasing is enabled, each pixel is subdivided into a 4×4 grid, giving sixteen sub-pixel samples (Used 2x2 grid for testing purposes as it is way faster).

For each sub-sample, the code computes a slightly offset image-plane position, uses the camera model to generate a ray direction, optionally applies depth-of-field and motion blur jitter, and then calls `traceRay`. The sixteen colours are then averaged to obtain the final pixel value. This reduces jagged edges on high-contrast boundaries and also produces smoother results when combined with motion blur and glossy reflections.

The antialiasing path is intentionally structured in parallel with the single-sample path: the same camera and shading code is used in both cases, which makes it straightforward to toggle antialiasing at runtime and compare the visual difference between aliased and supersampled images.

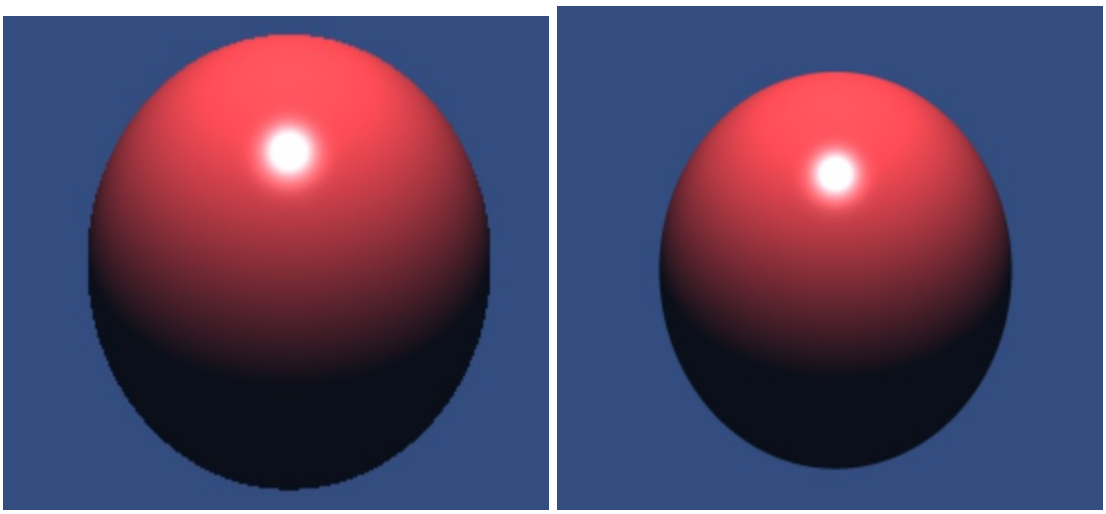

Figure 3: Comparison of a scene rendered without antialiasing (left) and with a 4x4 supersampling grid (right). test3.json [./raytracer –no-aa and ./raytracer –aa]

3.5 Texture Mapping

The texture system is encapsulated in a small `Texture` class that supports both ASCII and binary PPM files (P3 and P6). At load time the class parses the PPM header, normalises colour values to 8-bit RGB and stores them in a flat array. The `sample` method takes continuous `(u,v)` coordinates, wraps them into the range $[0, 1)$, maps them to integer pixel indices and returns a

normalised RGB triple. For simplicity, a nearest neighbour lookup is used; this is sufficient to demonstrate texture mapping for the coursework.

On the geometry side, each primitive's intersection routine fills in the `uvw` field of the hit record with its own texture coordinates. Spheres and ellipsoids use a spherical mapping derived from the unit direction to the hit point, plane quads parameterise the surface using bilinear coordinates from one corner, and cubes map each face from $[-1, 1]$ to $[0, 1]$. These UVs are then read in `shadeHit` and passed to the texture sampler whenever textures are enabled.

In the current implementation, the raytracer loads a single PPM file from the `Textures` folder and applies it to all textured objects. The material exported from Blender already contains fields for `kd`, `ks`, roughness and other properties. The PPM sample acts as an albedo map: the sampled colour is multiplied into `kd` before diffuse shading, so the same lighting code can be used for both textured and untextured objects. This could be extended straightforwardly to support distinct textures per material by parsing the optional texture filename emitted by the exporter.

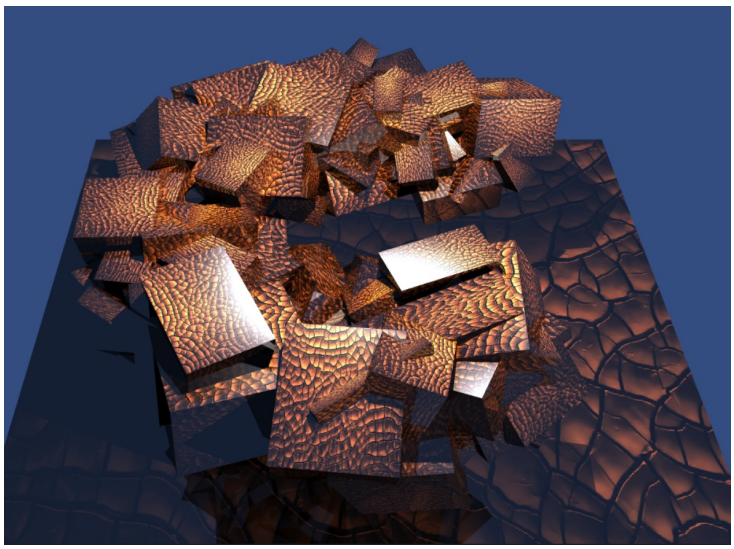

Figure 4: Texture mapped onto cubes and a plane using the shared PPM texture. test2.json [make run_no_lens]

3.6 Comparison with Blender and Testing

Because the camera parameters and scene geometry are exported directly from Blender, the projection and relative object placement in my images closely match Blender's Cycles/EEVEE renders. The main differences are in shading and global illumination: my implementation uses a simple Whitted-style model with Blinn–Phong specular highlights, no indirect diffuse, and a minimal sky colour. As a result, colours and contrast differ, but the composition (camera angle, object silhouettes and relative depth) aligns well with the Blender output.

How to reproduce and test (for the marker).

- Use the provided `.blend` file in the `Blend/` folder, run the exporter script to regenerate `export.json`, and render in Blender for reference.
- Build the raytracer via the Makefile and run it once with default settings to generate `Output/output.ppm`, which shows Whitted-style shading with antialiasing and textures enabled.
- Re-run with the flags `--no-aa` and/or `--no-textures` to obtain ablation images for antialiasing and texturing. These runs correspond to the comparison figures included in this section.

4 Final Raytracer and Distributed Effects (Module 4)

The final stage of the project ties together all previous modules into a single executable raytracer and adds several "distributed" effects: soft shadows, glossy reflections, depth-of-field and motion blur. The end result is a modular system that can be driven from the command line and from Blender via the JSON exporter, while remaining small enough to inspect and modify.

4.1 System Integration and Command-Line Control

All functionality is exposed through a single C++ program `raytracer.cpp`. At startup the code parses a set of command-line flags and stores them in global feature toggles such as `g_enable_textures`, `g_enable_soft_shadows`, `g_enable_glossy`, `g_enable_dof`, `g_enable_motion_blur` and `g_enable_aa`. Each flag (`--no-dof`, `--no-motion-blur`, `--no-soft-shadows`, `--no-glossy`, `--no-aa`, `--no-textures`) directly disables the corresponding effect without changing any other part of the pipeline. This means the same binary can be used both as a simple Whitted renderer and as a more feature-rich distributed raytracer, simply by toggling options.

Scene data is loaded from the `export.json` file produced by the Blender exporter. The camera class reads focal length, sensor size, image resolution and the exported gaze and up vectors, so the virtual camera in the raytracer matches Blender's perspective. Geometry is created from the exported spheres, cubes and planes, and each object is mapped to a `Material` structure (diffuse/specular colours, shininess, reflectivity, transmissive flag and index of refraction). Lights are parsed as point lights with an optional `radius` field, which is used to distinguish between hard and soft shadows. A BVH is always built over the set of `Shape` pointers to accelerate intersection queries for both primary and secondary rays. Before rendering, a single ray is traced through the centre pixel using the BVH to estimate an auto-focus distance; this distance is later used by the depth-of-field code as the focal plane.

4.2 Distributed Raytracing: Soft Shadows and Glossy Reflections

Soft shadows are implemented entirely in the lighting routine. Each point light carries a radius; when this radius is non-zero and soft shadows are enabled, the light behaves as a small spherical area instead of an ideal point. For every shading event the code draws several random offsets inside a unit sphere, scales them by the light radius and adds them to the nominal light position. Each offset defines a slightly different shadow ray. If a particular shadow ray is occluded, that sample contributes nothing; if it reaches the light, a Blinn–Phong contribution is added. Averaging over these samples produces penumbrae: pixels close to the geometrical shadow edge receive a mix of lit and shadowed samples, so shadows blur gradually rather than changing abruptly.

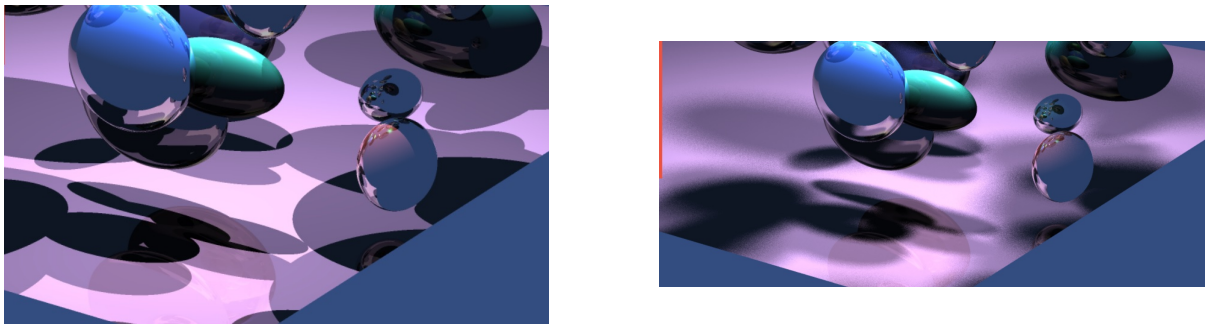

Figure 5: Comparison of a scene rendered without soft shadows (left) and with soft shadows (right). test1.json with [./ raytracer –no-soft-shadows] and [./raytracer –soft-shadows] Shadows may appear different when you run it as I have turned down kshadowsamples so it doesn't take too long for you to run it. For these results, kshadowsamples = 12

Glossy reflections use a similar idea, but in direction space. First, the ideal mirror direction is computed from the incident ray and the surface normal. Then, if glossy reflections are enabled, the code perturbs this direction with a small random vector drawn from a unit sphere, scaled by a roughness parameter, and normalises the result. Multiple such jittered directions are traced and averaged. The roughness is derived from the material's Blinn exponent: very high `shininess` values lead to a small lobe (almost perfect mirror), while lower exponents produce wider scattered reflections. This creates visibly smooth highlights on shiny spheres and more blurred reflections on rough materials, without changing the underlying Whitted recursion scheme.

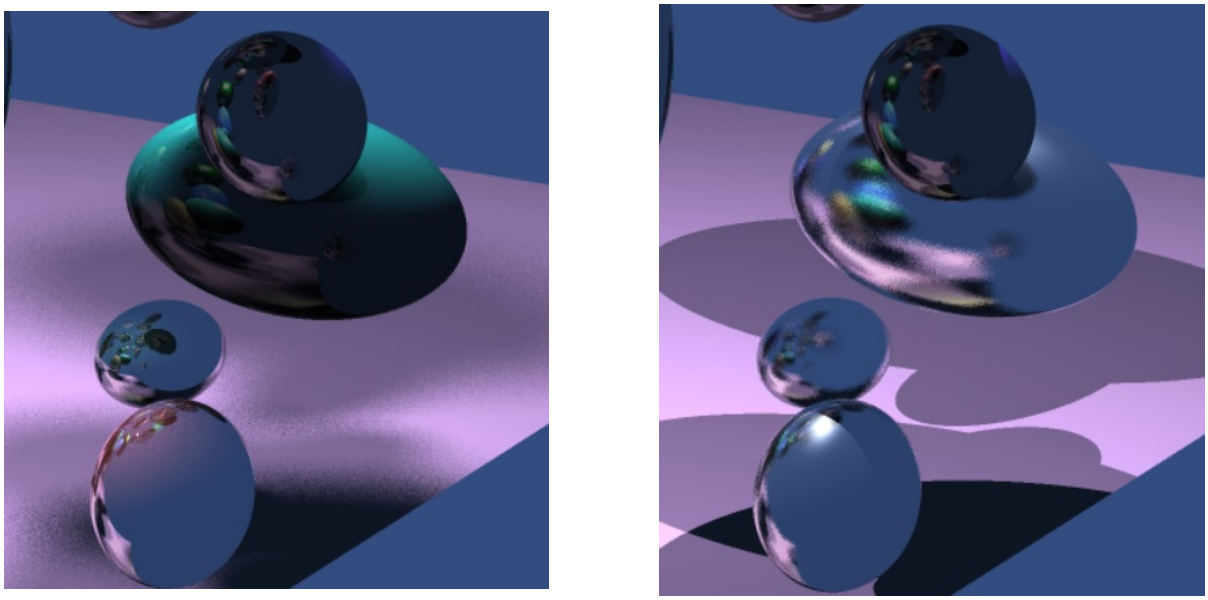

Figure 6: Comparison of a scene rendered without custom glossy reflections and with custom glossy reflections. test1.json with [./raytracer –no-glossy] and [./raytracer –glossy]

4.3 Lens Effects: Motion Blur and Depth of Field

Lens effects in the final renderer are driven by two independent random variables: the ray's *time* (for motion blur) and the ray's *origin* on a finite lens aperture (for depth of field).

Motion blur is implemented by assigning every ray a timestamp. When enabled, the renderer draws a uniform random value between `g_shutter_open` and `g_shutter_close` and stores it in `ray.time`. Dynamic primitives such as spheres use this timestamp during intersection: each object with a velocity vector interprets `ray.time` as a normalised shutter position, and its centre is evaluated as

$$c(t) = c_0 + t\,v.$$

The BVH bounds are also expanded to cover the entire swept volume between $t = 0$ and $t = 1$, ensuring that moving objects are always intersected correctly. With many temporal samples, fast-moving objects appear naturally streaked along their path, whereas static objects remain perfectly sharp. **Exceptionality Marks for DOF implementation:** Depth of field is implemented using a thin-lens model on top of the pinhole camera. For each primary sample, the renderer first computes the ideal pinhole ray using the camera transforms from Module 1 and finds where this ray meets a plane at the auto-focus distance. This point represents where the pinhole ray would intersect the focal plane. Next, the renderer samples a random point on a disk representing the camera lens using `random_in_unit_disk`, scales this point by a lens radius, and converts it into world space using the camera's right and up vectors. The primary ray then originates from this jittered lens position and points toward the focal-plane intersection. Objects lying exactly at the focal distance remain sharp, while objects nearer or farther receive slightly different sample origins per pixel and blur into circles of confusion.

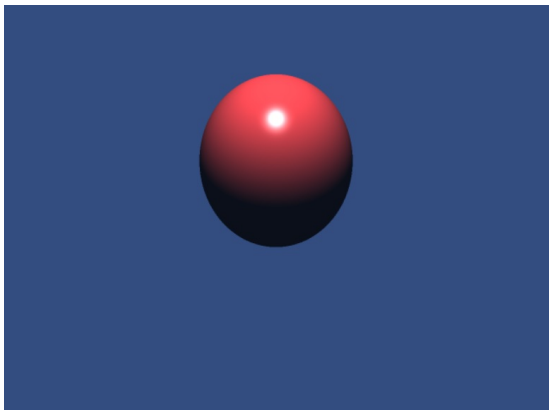

Figure 7: [./raytracer –no-motion-blur]

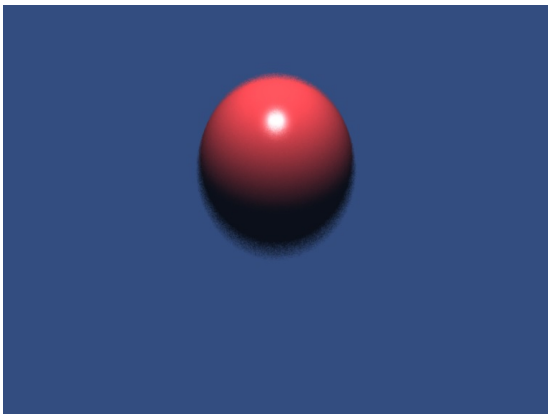

Figure 8: [./raytracer –motion-blur]

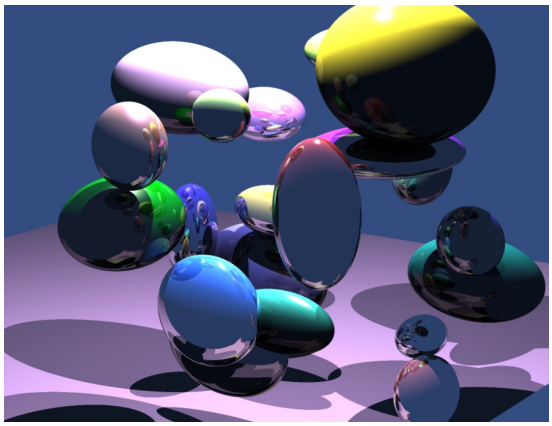

Figure 9: [./raytracer –no-dof]

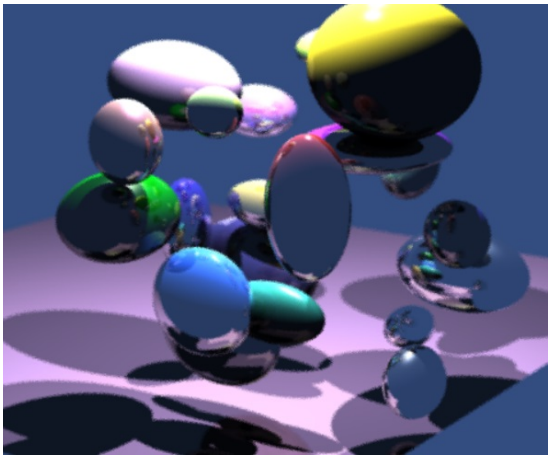

Figure 10: [./raytracer –dof]

4.4 Usage and Testing

The final executable always builds the full system but allows individual effects to be disabled for comparison. Running the program with no extra flags produces a Whitted-style image with textures, soft shadows, glossy reflections, motion blur, depth of field and antialiasing enabled. Re-running the same scene with flags such as `--no-soft-shadows`, `--no-glossy`, `--no-dof` or `--no-motion-blur` yields diagnostic images where each feature can be inspected in isolation. All outputs are written as `P3` PPM files into the `Output/` folder and are directly comparable to Blender renders of the same exported scene.

5 Evaluation and Results

5.1 Comparison with Blender

Figure 11: Comparison of a scene with my raytracer (left) and the blender render (right)

5.2 Limitations and known issues

There is an issue with black artifacts showing when DOF and motion blur is set to a high amount. This can be fixed by increasing the number of samples but also results in vastly longer render times. Not really a bug but annoying if a lot of blur is desired

6 Feature Completion Summary

6.1 Marks table and completion percentages

Topic	Max Marks	Estimated Completion (%)	Notes
Module 1: Blender exporter	1	100	
Module 1: Camera space	4	100	
Module 1: Image R/W	1	100	
Module 2: Ray intersection	4	100	
Module 2: Acceleration	4	100	
Module 3: Whitted-style	8	100	
Module 3: Antialiasing	2	100	
Module 3: Textures	2	100	
Final: System integration	4	100	
Final: Distributed RT	8	100	
Final: Lens effects	4	75	imperfect implementation
Report	8	100	
Exceptionalism	10	30	DOF implementation

Table 2: Summary of implemented topics and estimated completion.

7 Timeliness Bonus

7.1 Module 1 checkpoint

Had to do some slight changes with how cubes were represented as they slightly differed in blender and in my original implementation. I also had to make many changes to export.py to

support the changes in module 2 and 3. To test, simply run Export.py on a blender file and check camera.h and image.cpp

7.2 Module 2 checkpoint

Didn't make many changes to module 2. To test, check intersect.h for intersections and bvh.h and aabb.h for acceleration scheme

7.3 Module 3 checkpoint

Check Raytracer.cpp as most of the code is within there. Once again, just added functionality to Raytracer.cpp, didn't remove or change any existing. You can test and run module 3 in the exact same way as you tested the final raytracer. Overall, I believe I should receive full marks for timeliness bonus.

8 Use of Coding Assistants

Using a coding assistant for this assignment had both helpful and frustrating sides. On the positive end, it definitely sped things up, especially for the large or repetitive parts of the raytracer. Things like boilerplate C++ structures, JSON parsing, or setting up the BVH would have taken much longer on my own. It was also useful for explaining concepts quickly when I needed a refresher on something like jittered sampling or motion-blur timing.

But there were weaknesses too. The main issue was that the code the assistant produced didn't always work straight away. Sometimes it misunderstood what I wanted, or it wrote something that didn't fit neatly with the rest of my codebase. Because of that, I learned to be very modular and specific in how I asked for changes—usually requesting edits in small, isolated chunks rather than large rewrites. Even then, I sometimes had to fix or clean up parts myself.

There were also sections where it was simply easier to write the code on my own rather than explain it to the assistant in enough detail. Some logic was so particular to my setup that describing the exact behaviour I wanted would have taken longer than just implementing it directly.

Overall, the assistant was great for speeding up the broader structure and helping with explanations, but it wasn't a drop-in solution. I still had to understand the code, debug it, and fill in the gaps myself.