CGR Raytracer Coursework Report

System: macOS 25.0.0, clang-17.0.0

1 Implemented Features

1.1 Module 1: Foundation (100% Complete)

1.1.1 Vector Mathematics Library (File: `Code/vector.h`)

Generic template-based vector class: `Vec<N>` for arbitrary dimensions with specialized `Vec<3>` including x/y/z and r/g/b accessors, full operator overloading, dot/cross products, length, normalization. AI extended Vec3 to generic Vec; manually added color accessors.

1.1.2 Camera & Coordinate Transforms (Files: `Code/models.h`, `Code/image_utils.h`)

Camera struct with location, gaze, up, focal length, sensor size, resolution. `image_to_world_coordinates()` performs Pixel→NDC→Camera→World transformation with orthonormal basis.

1.1.3 PPM Image I/O (File: `Code/image.h`)

Reader/writer for ASCII (P3) and binary (P6) PPM formats with comment handling, byte-to-double conversion (0-255 → 0.0-1.0), clamping, and error handling. Modified AI code from `vector<vector<char>>` to `Vec<Vec<Pixel>>`.

1.1.4 Scene Parser (Files: `Code/scene_parser.cpp`, `Code/scene_utils.h`)

Parses custom text format with `load_scene()` supporting cameras, lights, and primitives (Spheres, Cubes, Planes, Toruses, Cylinders) using `std::istringstream`.

1.1.5 Blender Exporter (File: `Blend/Export.py`)

Python script exporting Blender scenes to custom format with object type detection, camera transformations, and matrix conversions. Enhanced to check `obj.name` and `obj.data.name` with automatic directory creation.

1.1.6 Build System (File: `Code/Makefile`)

Standard g++ Makefile with dependencies and clean target.

1.2 Module 2: Raytracing (100% Complete)

> *Changes from the Previous Module:*
> 1. *The transformation information for spheres was not incorporated into the previous module, so this had to be modified and added.*
> 2. *Added a new constructor for the Image class for empty images.*

1.2.1 Ray Shape Intersection (Files: `Code/shapes.h`, `Code/shapes.cpp`)

Implemented intersection tests for three primitive types using a transformation-based approach.

- **Sphere**: Transforms the ray to object space where the sphere always has unit radius (r=1) at the origin, solves the quadratic equation $at^2+2bt+c=0$ using the discriminant, and finds the nearest valid intersection.
- **Cube**: Uses the AABB slab method, testing the ray against three pairs of parallel planes (-0.5 to 0.5 per axis), tracks near/far t values, and determines the hit face by comparing intersection point coordinates with a tolerance threshold.
- **Plane**: Computes the normal via cross product of edge vectors from the first three points, calculates the intersection using the plane equation, and validates that the hit is within the rectangular bounds defined by all points.
- All methods transform the results back to world space.

1.2.2 Transformation System (Files: `Code/transform.h`, `Code/transform.cpp`)

Implemented 4x4 transformation matrices with the `Mat4` struct supporting multiplication and point/direction/normal transforms. The `Transform` class manages bidirectional object ↔ world conversions by storing both forward and inverse matrices. Factory methods `from_trs()` and `from_trs_nonuniform()` compose Scale→Rotate→Translate matrices. Matrix inversion is performed via Gaussian elimination with pivoting. Normal transformation uses the transpose of the inverse $\left(M^{-1}\right)^T$. Bounding box transformation converts all 8 corners to world space and recomputes axis-aligned bounds.

1.2.3 Hit Record Structure (File: `Code/hit_record.h`)

A data structure that stores ray intersection results: `intersection_point` (position), `normal` (surface normal), `t` (ray parameter distance), and `front_face` (boolean for outside/inside hit detection).

1.2.4 BVH Implementation (Files: `Code/bvh.h`, `Code/bvh.cpp`)

- A spatial acceleration structure using a binary tree with axis-aligned bounding boxes. Each `BVHNode` is either internal (with left/right children) or a leaf (containing object indices).
- Build process: computes the bounding box for the object set, splits at the median along the longest axis (X/Y/Z), and recurses until MAX_LEAF_SIZE or MAX_DEPTH is reached.
- Traversal tests ray-box intersection first; skips a subtree if missed, and early-exits from a leaf if no objects are hit. Maintains closest_t to find the nearest intersection.
- Object indexing: [0..n_spheres-1] = spheres, [n_spheres..n_spheres+n_cubes-1] = cubes, [remaining] = planes.

1.2.5 Rendering Pipeline (Files: `Code/raytracer.h`, `Code/raytracer.cpp`)

- `generate_camera_ray()` : Transforms from Pixel → NDC → Sensor → World space.
- `intersect_scene()` : Brute-force O(n) approach that tests all primitives.
- `intersect_bvh()` : Recursive BVH traversal with bounding box culling for accelerated intersection testing.

- `ray_color()` : Simple shading that maps surface normals to RGB colors (normal.xyz ⟶ 0.5*(normal+1)).
- `render_scene()` and `render_scene_bvh()` : Main rendering loops that generate rays per pixel and track intersection test counts for performance comparison.

1.3 Module 3: Whitted-Style Raytracing (100% Complete)

> *Changes from the Previous Module:*
> 1. *Extended HitRecord to include UV coordinates for texture mapping.*
> 2. *Added Material struct with comprehensive properties: diffuse, specular, ambient, shininess, reflectivity, transparency, and refractive index.*
> 3. *Updated scene parser to parse material properties and texture files from ASCII scene files.*

1.3.1 Whitted-Style Recursive Raytracing (Files: `Code/raytracer.cpp` , `Code/raytracer.h`)

The rendering pipeline implements the full Whitted-style raytracing algorithm with recursive reflection and refraction.

Ray Generation & Scene Traversal

- Primary rays generated via `generate_camera_ray()` performing Pixel→NDC→Sensor→World transformation
- `ray_color()` and `ray_color_bvh()` handle recursive ray tracing with maximum depth of 5 bounces
- Intersection testing uses both brute-force (`intersect_scene()`) and BVH-accelerated (`intersect_scene_bvh()`) methods

Shading Model: Composite BRDF The `blinn_phong_shading()` function implements a physically-based composite BRDF combining:

- **Diffuse component f_d**: Lambertian BRDF
- **Specular component f_s**: Blinn-Phong BRDF
- **Final shading equation**: $L_{\text{out}} = (f_d + f_s) \cdot L_{\text{in}} \cdot (N \cdot L)$ per light source
- **Ambient term**: $L_{\text{ambient}} = k_a \cdot 0.3$ for global illumination approximation

Lighting & Shadows

- Point lights with inverse-square falloff: $L_{\text{in}} = I \cdot \frac{\text{color}}{d^2}$
- Hard shadows via shadow rays tested with `is_in_shadow_bvh()`
- Shadow rays offset by 0.001 units along light direction to prevent self-intersection
- Multiple light sources accumulated additively

Recursive Ray Tracing

- **Reflection**: Perfect mirror reflection computed via $R = I - 2(I \cdot N)N$, weighted by material reflectivity k_r
- **Refraction**: Snell's law implementation with $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$, handling total internal reflection, weighted by transparency k_t
- **Color blending**: $\text{Color} = w_{\text{local}} \cdot L_{\text{local}} + k_r \cdot L_{\text{refl}} + k_t \cdot L_{\text{refr}}$ where $w_{\text{local}} = 1 - k_r - k_t$
- Refracted rays tinted with material diffuse color and brightness-boosted (2×) to compensate for dark backgrounds

Gamma Correction & Output

- Gamma correction with $\gamma = 2.2$: $\text{Color}_{\text{out}} = \text{Color}_{\text{linear}}^{\frac{1}{2.2}}$

1.3.2 Antialiasing (Files: `Code/raytracer.cpp`)

Implemented stochastic supersampling antialiasing to reduce jagged edges and improve image quality.

Multi-Sampling Strategy

- `render_scene_bvh_antialiased()` implement multi-sampling with configurable samples per pixel
- Random jittering within each pixel: offset sampled uniformly in [0, 1] via `random_double()`
- Ray generation with subpixel offsets: $\frac{x+r_x}{\text{width}}$ and $\frac{y+r_y}{\text{height}}$ where r_x, r_y Uniform(0, 1)

Sample Accumulation

- Colors accumulated across all samples: $\text{Color}_{\text{pixel}} = \frac{1}{N} \sum_{i=1}^{N} \text{Color}_i$

1.3.3 *Texture Mapping (Files: `Code/texture.h`, `Code/texture.cpp`, `Code/shapes.cpp`, `Code/models.h`)*

Implemented UV texture mapping with procedural coordinate generation for all primitive types.

UV Coordinate Generation The intersection routines in `shapes.cpp` populate `HitRecord.u` and `HitRecord.v` with surface parameterization for each respective shape.

Texture Manager System `TextureManager` class provides centralized texture loading and sampling.

Texture Sampling `sample(filename, u, v)` performs bilinear texture lookup:

- UV wrapping: $u' = u - \lfloor u \rfloor$ and $v' = v - \lfloor v \rfloor$ to repeat textures
- Pixel coordinate conversion: $x = u' \times (\text{width} - 1)$, $y = v' \times (\text{height} - 1)$

Material Integration In `blinn_phong_shading()`:

- Base color source: texture color if `material.has_texture`, otherwise `material.diffuse_color`
- Texture modulation: $\text{Color}_{\text{final}} = \text{Color}_{\text{texture}} \times \text{Color}_{\text{material}}$ allowing material-based tinting

1.4 Test Results

1.4.1 Module 1

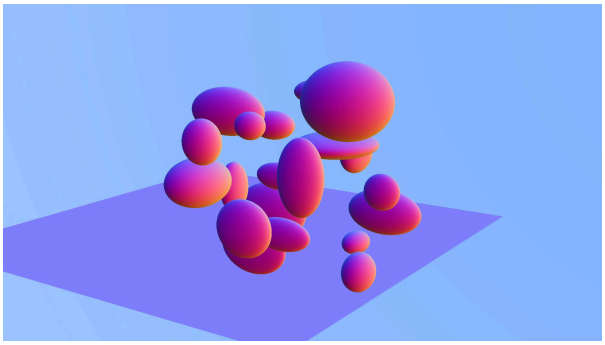
Successfully loads scenes (`ASCII/def_scene.txt`, `Blend/*.blend`), reads PPM images, generates/exports ray directions. Camera transforms validated via Blender visualization.

1.4.2 Module 2

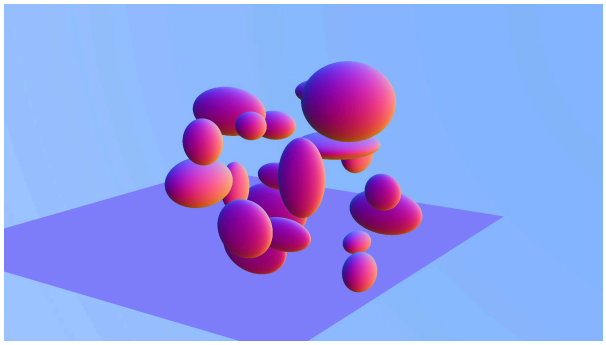
The BVH acceleration structure achieves an 8.96× speedup in render time (2.83s ⟶ 0.32s) while reducing intersection tests by 11.7× (21.0 ⟶ 1.79 tests per ray). Both methods produce visually identical results, with 707K ray hits out of 2.07M rays cast.

Metric	Brute Force	BVH
Resolution	1920×1080	1920×1080
Total Rays	2,073,600	2,073,600
Total Time	2,831 ms	316 ms
Ray Hits	707,259 (34.1%)	707,258 (34.1%)
Intersection Tests	43,545,600	3,708,325
Tests per Ray	21.0	1.79
Speedup	1.0×	**8.96×**
Test Reduction	—	**11.7×**

Table 1: Performance comparison between brute force and BVH rendering methods



Output of the brute force render.

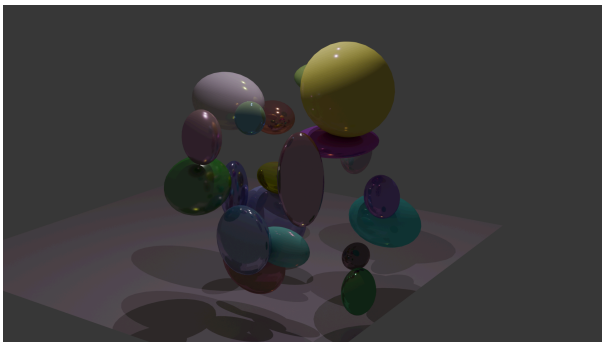


Output of the BVH render.

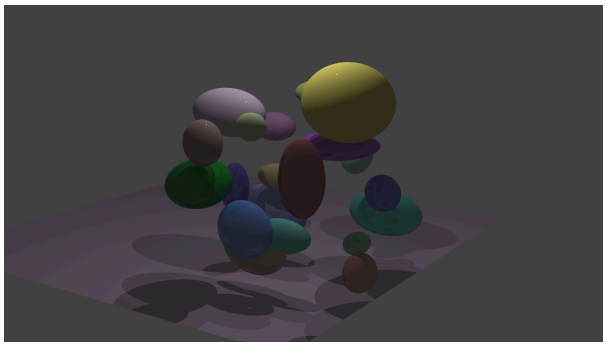
Figure 1: Comparison of render outputs using the Test1.blend file.

1.4.3 Module 3

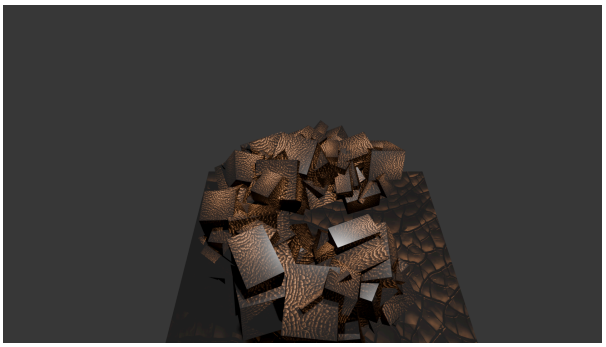
Here are the results from the ray-tracer for the 3 test scene.



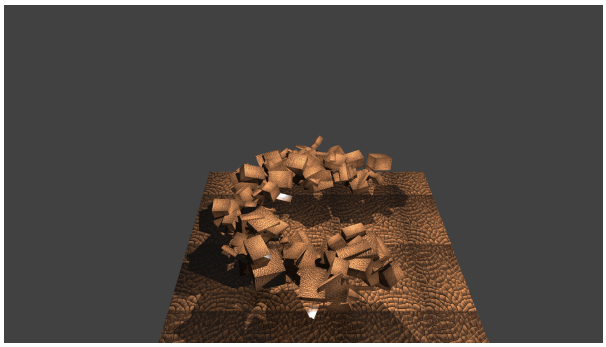
Expected Output of Test1.blend



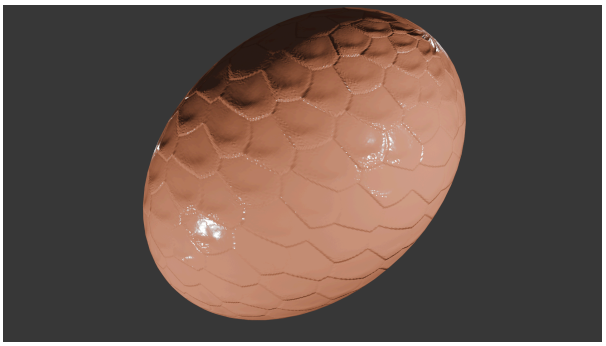
Rendered Output of Test1.blend



Expected Output of Test2.blend



Rendered Output of Test2.blend



Expected Output of Test3.blend

Rendered Output of Test2.blend

Figure 2: Comparison of render outputs using the test files.

2 Module Completion Status

Module/Topic	%	Description
Module 1	**100**	**Foundation**
Vector Math	100	Generic Vec with operations
Camera Class	100	Camera model + transformations
Coordinate Transforms	100	Pixel⟶NDC⟶Camera⟶World
PPM I/O	100	P3/P6 reader/writer
Scene Parser	100	All object types supported
Blender Exporter	100	Python export script
Build System	100	Makefile
Module 2	**100**	**Ray Tracing**
Sphere Intersection	100	Using ray and sphere equations
Plane Intersection	100	Points ⟶ Plane Eqn (+ Ray Eqn) ⟶ Point of intersection
Cube Intersection	100	AABB method for cube intersection
Transformation support for shapes	100	Transformation system using 4x4 matrices
Bounding Box support	100	Every shape can produce a bounding box
BVH Implementation	100	BVHNode hierarchical structure with traversal
Rudimentary Rendering	100	Using BVH and sequential approaches to render objects in the scene
Module 3	**100**	**Whitted-Style Raytracing**
Whitted-style raytracing	100	Recursive reflection and refraction with Blinn-Phong shading
Composite BRDF	100	Lambertian diffuse + Blinn-Phong specular components
Shadow rays	100	Hard shadows with shadow ray testing
Material system	100	Full material properties: diffuse, specular, ambient, reflectivity, transparency, IOR
Antialiasing	100	Stochastic supersampling with configurable samples per pixel
Texture mapping	100	UV coordinate generation for all primitives with TextureManager

Table 2: Completion Overview

3 Timeliness Bonus Justification

All Module 1 features have been implemented and checkpointed. Testing was performed using the Lab 2 ray visualization script in Blender with ray-to-cylinder conversion.
All Module 2 features have been implemented, and the details of their implementation are documented. An empirical test was performed using both the BVH and brute force techniques to verify the output of the raytracer.
All Module 3 features have been 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, 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.