CSE 167 Final Project Write-up

Ray Tracer Implementation

Andrew Onozuka A16760043
University of California, San Diego

March 21st, 2025

This document describes the implementation of a ray tracer for UCSD's CSE 167: Computer Graphics, taught in Winter 2025 by Albert Chern. The project consists of seven major tasks, each contributing to the development of a basic ray tracer with features like shading, reflections, and global illumination.

Contents

Setup

```
ryoandrewonozuka@16-inch-MacBook-Pro-2021-Space-Gray-1-TB RayTracer-setup % mkdir build
cd build
cmake ..
cmake --build .
../bin/RayTracer 1 1 1
-- The CXX compiler identification is AppleClang 16.0.0.16000026
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (1.0s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/ryoandrewonozuka/Documents/GitHub/notes/ucsd/cse167/final_project/RayTracer-setup/build
[  9%] Building CXX object CMakeFiles/RayTracer.dir/main.cpp.o
[ 18%] Building CXX object CMakeFiles/RayTracer.dir/src/Camera.cpp.o
[ 27%] Building CXX object CMakeFiles/RayTracer.dir/src/RayTracer.cpp.o
[ 36%] Building CXX object CMakeFiles/RayTracer.dir/src/Scene.cpp.o
[ 45%] Building CXX object CMakeFiles/RayTracer.dir/src/geometries/GeomSphere.cpp.o
[ 54%] Building CXX object CMakeFiles/RayTracer.dir/src/geometries/GeomTriangle.cpp.o
[ 63%] Building CXX object CMakeFiles/RayTracer.dir/src/materials/GlossyMaterial.cpp.o
[ 72%] Building CXX object CMakeFiles/RayTracer.dir/src/models/Sphere.cpp.o
[ 81%] Building CXX object CMakeFiles/RayTracer.dir/src/models/Square.cpp.o
[ 90%] Building CXX object CMakeFiles/RayTracer.dir/src/models/Tetrahedron.cpp.o
[100%] Linking CXX executable /Users/ryoandrewonozuka/Documents/GitHub/notes/ucsd/cse167/final_project/RayTracer-setup/bin/RayTracer
[100%] Built target RayTracer
OpenGL Version: 4.1 Metal - 89.3

    Available commands:
      press 'H' to print this message again.
      press Esc to quit.

      Camera Controls:
      press 'O' to save a screenshot.
      press 'W'/'S' for front/back movement.
      press 'A'/'D' for left/right movement.
      press 'Q'/'E' for up/down movement.
      press the arrow keys to rotate camera.
      press 'Z'/'X' to rotate aroud view axis.
      press '-'/'+' to zoom (change of Fovy).
      press 'R' to reset camera.

      Shading Mode:
      press Spacebar to Ray Trace.
      press 'N' for Normal Shading.
      press 'P' for Debug Mode.

Rendering Progress: [==========================================] 100% Elapsed: 00s Remaining: 00s
Done! 480000 rays processed
```

Figure 1: This is what compiling in the VS Code terminal looks like.

Figure 2: Image output before any implementations.

Task 1: Ray Creation

Overview

The goal of this task was to implement a function to generate rays from the camera through each pixel. Each ray is cast from the camera's position ('eye') and directed through the center of a pixel in the image plane. The result should produce a gradient transitioning from white at the top to blue at the bottom, indicating that rays are hitting the background.

Implementation

To compute the correct direction for each ray, we follow a **perspective projection model** based on the camera parameters. The core of the implementation involves the following.

Step 1: Define the Ray Origin
Each ray originates from the camera's **eye** position:

```
// p0
ray.p0 = glm::vec3(camera.eye);
```

Step 2: Compute Pixel Coordinates in Normalized Device Space
Since the image is divided into discrete pixels, we convert pixel indices (i, j) to **normalized coordinates**

```
float x = 0.5f;
float y = 0.5f;
```

Here, $x = 0.5$ and $y = 0.5$ ensure that the ray passes through the **center of each pixel**. This will later be modified in Task 3 to introduce **anti-aliasing**.

Step 3: Compute Ray Direction in Camera Space
To correctly map pixel positions to world space, we first determine how much each pixel deviates from the center of the image. This requires a **perspective projection transformation**, which scales the coordinates using the camera's **field of view ('fovy')**:

```
// Compute scale factor based on field of view (fovy)
float scale = tan(glm::radians(camera.fovy * 0.5f));

// Convert pixel (i, j) to normalized device coordinates (NDC)
float alpha = (2.0f * (i + x) / camera.width - 1.0f) * camera.aspect * scale;
float beta = (1.0f - 2.0f * (j + y) / camera.height) * scale;
```

- α controls the **horizontal deviation** (adjusted for aspect ratio).
- β controls the **vertical deviation**.
- The scaling factor accounts for the **camera's field of view**.

Step 4: Transform to World Coordinates
To obtain the ray's direction in world space, we use the **camera's basis vectors** u, v, and w, which represent the **right, up, and forward** directions of the camera:

```
// Extract camera basis vectors from the camera matrix
vec3 u(camera.cameraMatrix[0]); // Right vector
vec3 v(camera.cameraMatrix[1]); // Up vector
vec3 w(camera.cameraMatrix[2]); // Forward vector

// Compute ray direction in world space
ray.dir = glm::normalize(alpha * u + beta * v - w);
```

This equation ensures that the ray properly transforms from **camera space** to **world space**.

Results

The expected output for this task is a gradient transitioning from white at the top to blue at the bottom. This confirms that the ray direction is correctly computed, as rays that do not hit any objects return a **background color** interpolated based on their **y-axis direction**.

Figure 3: Gradient from white at the top to blue at the bottom, representing background rays.

Task 2: Ray Intersections

Task 2.1: Ray-Sphere Intersection

The intersection between a ray and a sphere is determined using the quadratic equation:

$$||\mathbf{ro} + t\mathbf{rd} - \mathbf{c}||^2 = R^2$$

where:

- **ro** is the ray origin.
- **rd** is the ray direction.
- **c** is the sphere center.
- R is the sphere radius.

Expanding this equation results in a quadratic formula:

$$at^2 + bt + c = 0$$

where:

$$\begin{aligned} a &= \mathbf{rd} \cdot \mathbf{rd} \\ b &= 2\mathbf{rd} \cdot (\mathbf{ro} - \mathbf{c}) \\ c &= (\mathbf{ro} - \mathbf{c}) \cdot (\mathbf{ro} - \mathbf{c}) - R^2 \end{aligned}$$

Solving for t using the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If the discriminant $(b^2 - 4ac)$ is negative, no real intersections exist.

Implementation

The following code snippet shows our implementation of ray-sphere intersection:

```
// Compute quadratic equation coefficients
vec3 oc = ro - center;
float a = dot(rd, rd);
float b = 2.0f * dot(oc, rd);
float c = dot(oc, oc) - radius * radius;

// Compute discriminant
float discriminant = b * b - 4 * a * c;

// If discriminant is negative, no intersection
if (discriminant < 0) {
    return intersections;
}

// Compute intersection points
float sqrt_disc = sqrt(discriminant);
float t1 = (-b - sqrt_disc) / (2.0f * a);
float t2 = (-b + sqrt_disc) / (2.0f * a);

// Store valid intersections (t > 0)
if (t1 > 0) {
    vec3 point = ro + t1 * rd;
    vec3 normal = normalize(point - center);
```

```
        intersections.push_back({t1, point, normal, this, nullptr});
    }
    if (t2 > 0) {
        vec3 point = ro + t2 * rd;
        vec3 normal = normalize(point - center);
        intersections.push_back({t2, point, normal, this, nullptr});
    }
```

Task 2.1 Results

The implementation was tested in our ray tracer. When correctly implemented, the ray should detect a sphere and render it with normal shading.

Figure 4: Ray intersection test with a sphere and box.

Task 2.2: Ray-Triangle Intersection

The intersection between a ray and a triangle is determined using the **Möller–Trumbore algorithm**, which is an efficient method for checking if a ray intersects a triangle. The algorithm avoids computing the plane equation explicitly and instead solves for the intersection point using barycentric coordinates. Given a triangle defined by three vertices V_0, V_1, V_2, the ray equation is:

$$\mathbf{r}(t) = \mathbf{ro} + t\mathbf{rd}$$

where:

- **ro** is the ray origin.
- **rd** is the ray direction.
- t is the intersection distance.

The intersection test is performed using:

$$t = \frac{(V_0 - \mathbf{ro}) \cdot \mathbf{n}}{\mathbf{rd} \cdot \mathbf{n}}$$

where $\mathbf{n}$ is the normal of the triangle. The barycentric coordinates are then used to check if the intersection lies inside the triangle.

Implementation

The following code snippet shows our implementation of ray-triangle intersection:

```
// Extract triangle vertices
vec3 v0 = vertices[0];
vec3 v1 = vertices[1];
vec3 v2 = vertices[2];

// Compute triangle edges
vec3 edge1 = v1 - v0;
vec3 edge2 = v2 - v0;

// Compute determinant using cross product
vec3 h = cross(ray.dir, edge2);
float det = dot(edge1, h);

// If determinant is near zero, ray is parallel to triangle (no intersection)
if (abs(det) < 1e-6) return intersections;

float inv_det = 1.0f / det;

// Compute barycentric coordinates
vec3 s = ray.p0 - v0;
float u = dot(s, h) * inv_det;
if (u < 0.0f || u > 1.0f) return intersections;

vec3 q = cross(s, edge1);
float v = dot(ray.dir, q) * inv_det;
if (v < 0.0f || u + v > 1.0f) return intersections;

// Compute intersection distance t
float t = dot(edge2, q) * inv_det;
if (t <= 0) return intersections; // Intersection behind camera

// Compute intersection point
vec3 point = ray.p0 + t * ray.dir;

// Compute normal (flat shading)
vec3 normal = normalize(cross(edge1, edge2));

// Store intersection
intersections.push_back({t, point, normal, this, nullptr});
```

Task 2.2 Results

The implementation was tested in our ray tracer. When correctly implemented, the ray should detect the walls of the Cornell box, which are composed of triangles.

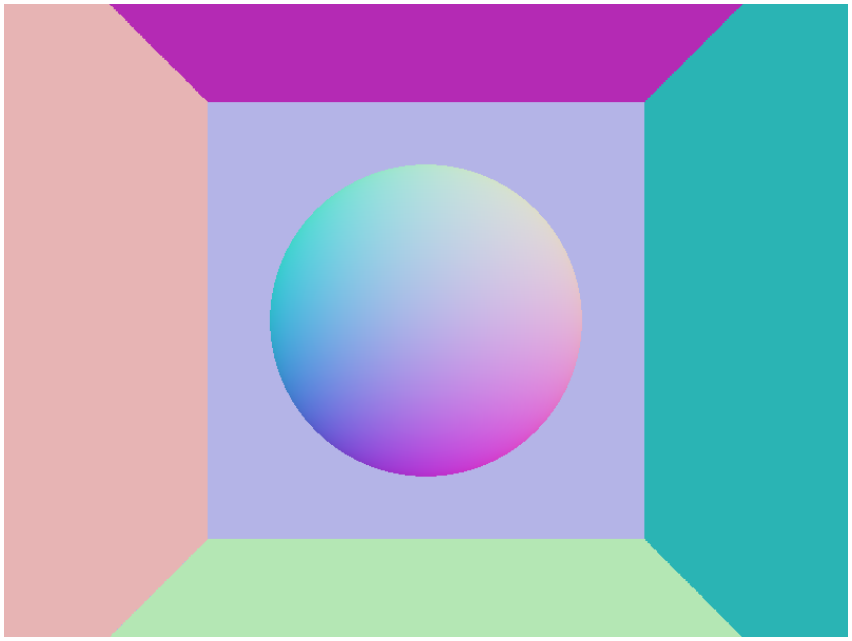

Figure 5: Ray intersection test with a Cornell box.

Task 3: Anti-Aliasing

Overview

To reduce jagged edges in the rendered image, we implemented **stochastic sampling**, also known as **supersampling anti-aliasing (SSAA)**. Instead of casting a single ray through the center of each pixel, we generate multiple rays **randomly sampled within the pixel** and average their contributions. This technique produces a smoother image by capturing subpixel variations.

Implementation

To implement stochastic sampling, we modified the function:

```
Ray RayTracer::ray_thru_pixel(int i, int j) {
    // Randomly sample x and y within the pixel
    float x = glm::linearRand(0.0f, 1.0f);
    float y = glm::linearRand(0.0f, 1.0f);

    // Compute scale factor based on field of view
    float scale = tan(glm::radians(camera.fovy * 0.5f));

    // Convert pixel (i, j) to normalized device coordinates (NDC)
    float alpha = (2.0f * (i + x) / camera.width - 1.0f) * camera.aspect * scale;
    float beta = (1.0f - 2.0f * (j + y) / camera.height) * scale;

    // Compute ray direction in world space
    vec3 u(camera.cameraMatrix[0]);
    vec3 v(camera.cameraMatrix[1]);
    vec3 w(camera.cameraMatrix[2]);

    Ray ray;
    ray.p0 = camera.eye;
    ray.dir = glm::normalize(alpha * u + beta * v - w);

    return ray;
}
```

This function ensures that each **call to 'ray_thru_pixel' generates a unique random sample**, allowing **smooth edge transitions** when multiple samples per pixel are used.

Results

The following images demonstrate the effect of increasing **samples per pixel (spp)**:

Figure 6: Single sample per pixel (spp = 1).

Figure 7: 10 samples per pixel (spp = 10).

Figure 8: 100 samples per pixel (spp = 100).

As seen above, increasing the number of samples per pixel **significantly smooths the edges**, reducing jagged artifacts.

Task 4: Shading

Overview

We implemented **single-bounce Phong-like shading**, which improves realism by simulating **direct light interaction** with surfaces. The shading accounts for:

- **Diffuse reflection** (Lambertian shading)
- **Specular highlights** (Phong shading)

Implementation

The shading model is computed using:

$$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

Where:

- I_{ambient} is a constant ambient term.
- $I_{\text{diffuse}} = C_{\text{diffuse}} \cdot E \cdot \max(0, N \cdot L)$ models diffuse reflection.
- $I_{\text{specular}} = C_{\text{specular}} \cdot E \cdot (R \cdot V)^{\alpha}$ models specular highlights.

The **Phong shading model** was implemented by modifying 'get_direct_lighting()':

```
float cos_theta = glm::max(dot(intersection.normal, shadow_ray.dir), 0.0f);

vec3 diffuse_color = vec3(1.0f); // Default white if material is missing
if (intersection.model && intersection.model->material) {
    std::shared_ptr<GlossyMaterial> glossy_material =
        std::dynamic_pointer_cast<GlossyMaterial>(intersection.model->
            material);
    if (glossy_material) {
        diffuse_color = glossy_material->diffuse;
    }
}
```

Results

The following images compare the scene **before and after implementing shading**:

Figure 9: Scene before shading implementation (flat colors).

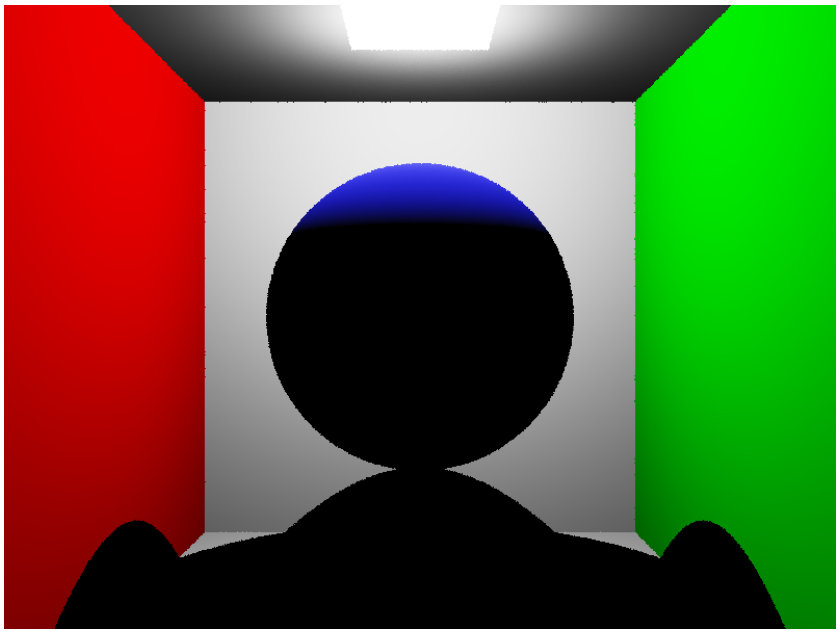

Figure 10: Scene after shading implementation (realistic lighting).

With shading implemented, the objects now **receive realistic lighting based on their orientation relative to light sources**.

Task 5: Soft Shadows

Overview

We improved the realism of shadows by implementing **soft shadows**. Rather than casting rays to a single point on the light source (which results in unnaturally sharp shadows), we now **sample randomly across the area of the light source**. This creates realistic penumbra effects, where shadows become blurrier further from occluders.

Implementation

To generate soft shadows from our square area light, we implemented **uniform random sampling** over the surface of the light using the following:

$$p = c + u \cdot s \cdot \vec{t} + v \cdot s \cdot \vec{b}, \quad u, v \in \left[-\frac{1}{2}, \frac{1}{2}\right]$$

Here:

- c: center of the square light
- s: side length
- $\vec{t}, \vec{b}$: tangent and bitangent vectors
- u, v: sampled uniformly from $[-0.5, 0.5]$

The following code implements this logic in `Square::get_surface_point()`:

```
vec3 Square::get_surface_point() {
    float u = linearRand(-0.5f, 0.5f);
    float v = linearRand(-0.5f, 0.5f);

    vec3 samplePoint = center + u * side_len * tangent + v * side_len *
        bitangent;

    // transform to world space
    return vec3(transformation_matrix * vec4(samplePoint, 1.0f));
}
```

Results

After implementing soft shadow sampling, we observed blurrier shadow boundaries, especially at higher samples-per-pixel (spp) values. At low spp, noise is present due to the random sampling, but this can be reduced with more rays per pixel. Soft shadows greatly enhance realism by simulating partial occlusion and contact shadow gradients.

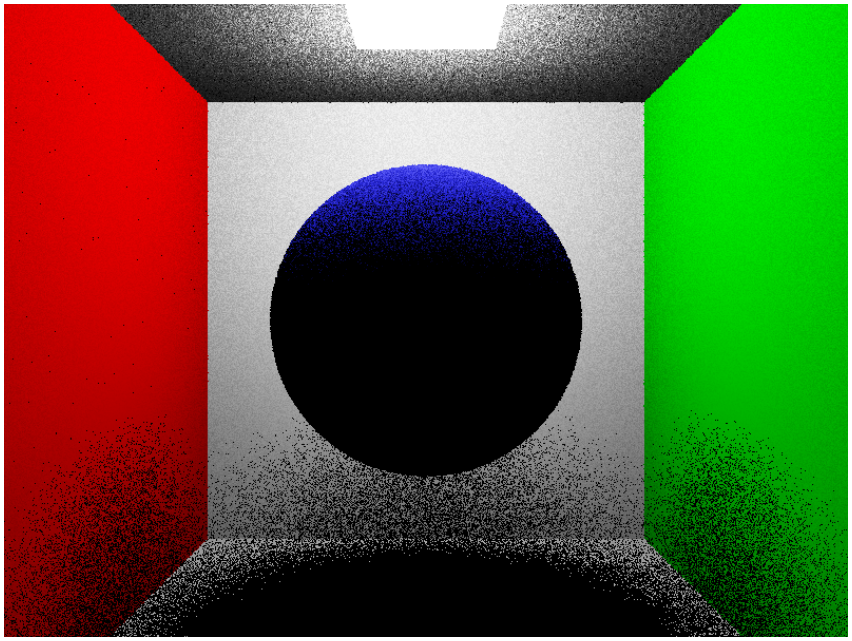

Figure 11: Soft shadows with uniform sampling of 1.

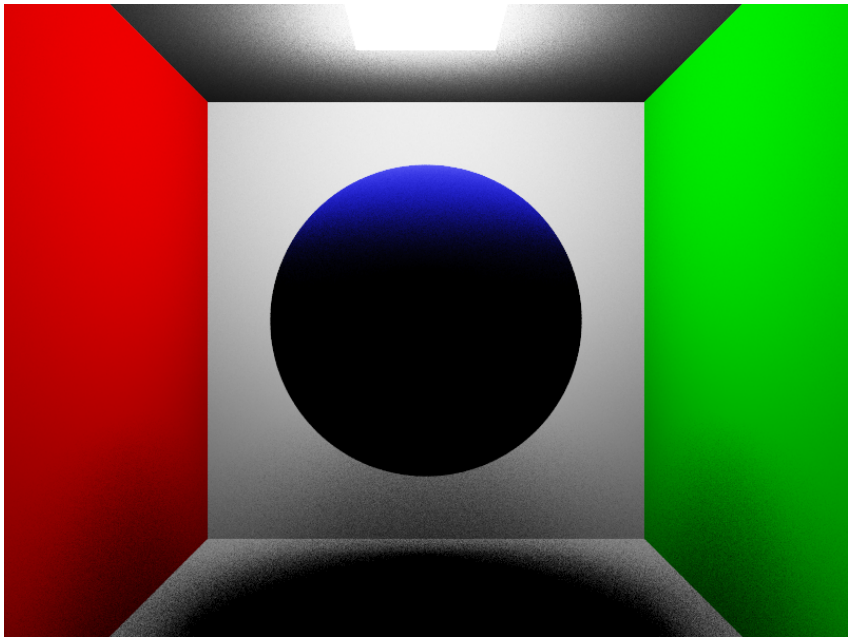

Figure 12: Soft shadows with uniform sampling of 50.

Task 6: Multiple Ray Bounces

Task 6.1: Diffuse Bounces (Cosine-Weighted Sampling)

In this task, we implemented diffuse-only bouncing using **cosine-weighted hemisphere sampling** to simulate realistic indirect light scattering off rough surfaces. For each diffuse bounce, the radiance is updated using the Lambertian BRDF scaled by the material's diffuse color and a cosine factor.

```
float s = linearRand(0.0f, 1.0f);
float t = linearRand(0.0f, 1.0f);
float u = 2.0f * M_PI * s;
float v = sqrt(1.0f - t);
vec3 hemisphere_sample = vec3(v * cos(u), sqrt(t), v * sin(u));
vec3 new_dir = align_with_normal(hemisphere_sample, normal);
```

`sample_ray_and_update_radiance()` uses these samples to spawn the next bounce ray with an updated working radiance (`W_wip`).

Task 6.1 Results

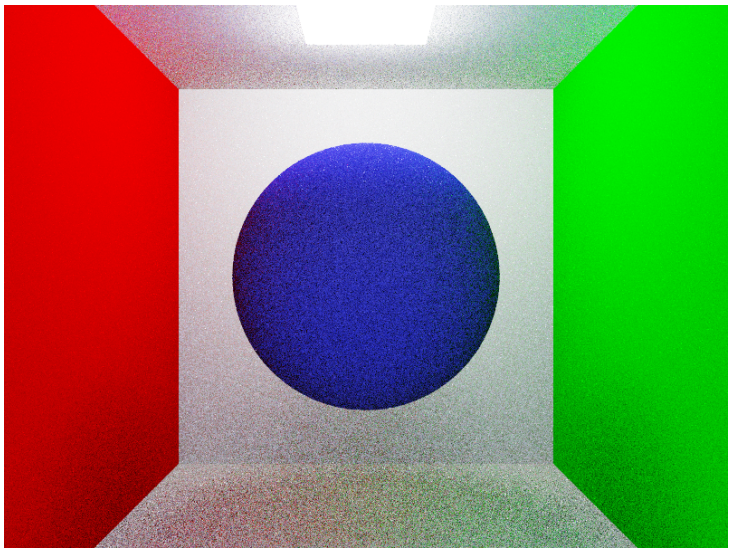

Figure 13: Task 6.1, 10 spp, 3 bounces.

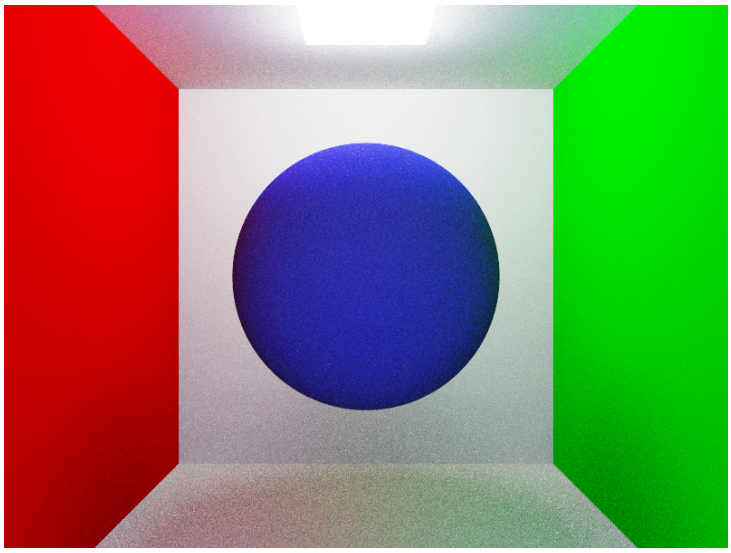

Figure 14: Task 6.1, 50 spp, 3 bounces.

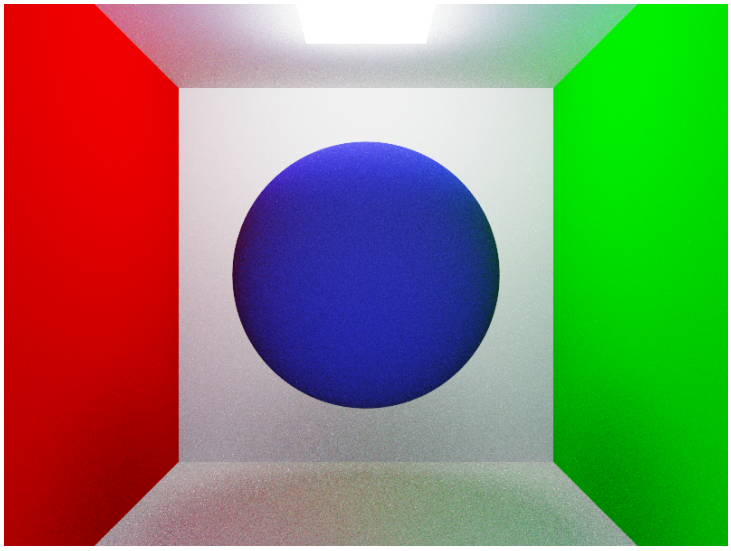

Figure 15: Task 6.1, 75 spp, 3 bounces.

Task 6.2: Specular Bounces (Mirror Reflection)

We extended our implementation to support perfect mirror-like reflections. If a material's shininess $\sigma = 1$, only mirror bounces are allowed. We calculate the reflection direction using:

```
vec3 reflection_dir = reflect(ray.dir, normal);
```

The radiance is updated using the specular reflectance of the surface. For hybrid materials, diffuse and specular bounces are selected stochastically based on shininess.

Task 6.2 Results

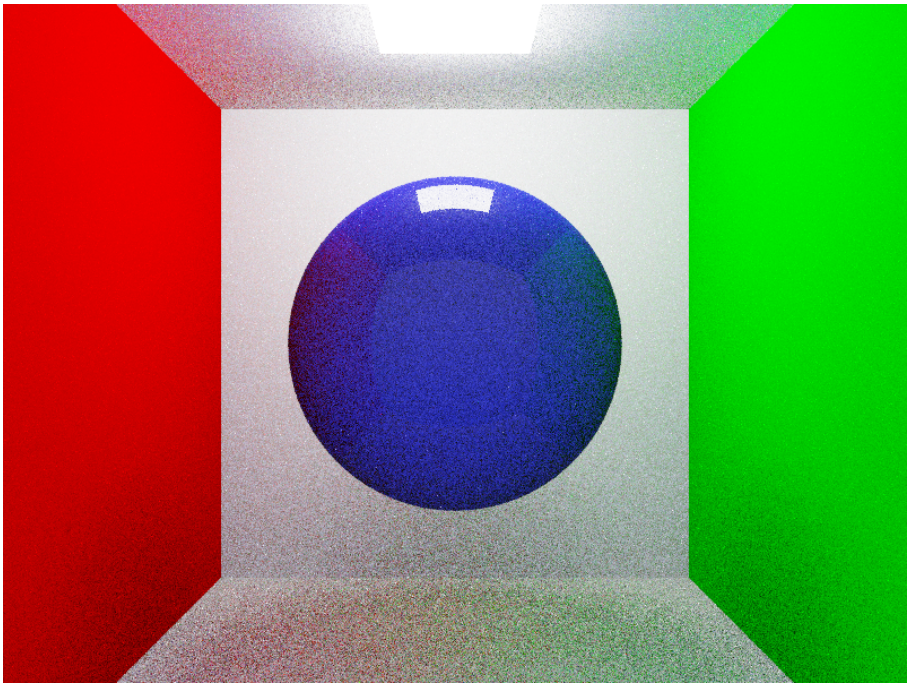

Figure 16: Task 6.2, 10 spp, 3 bounces.

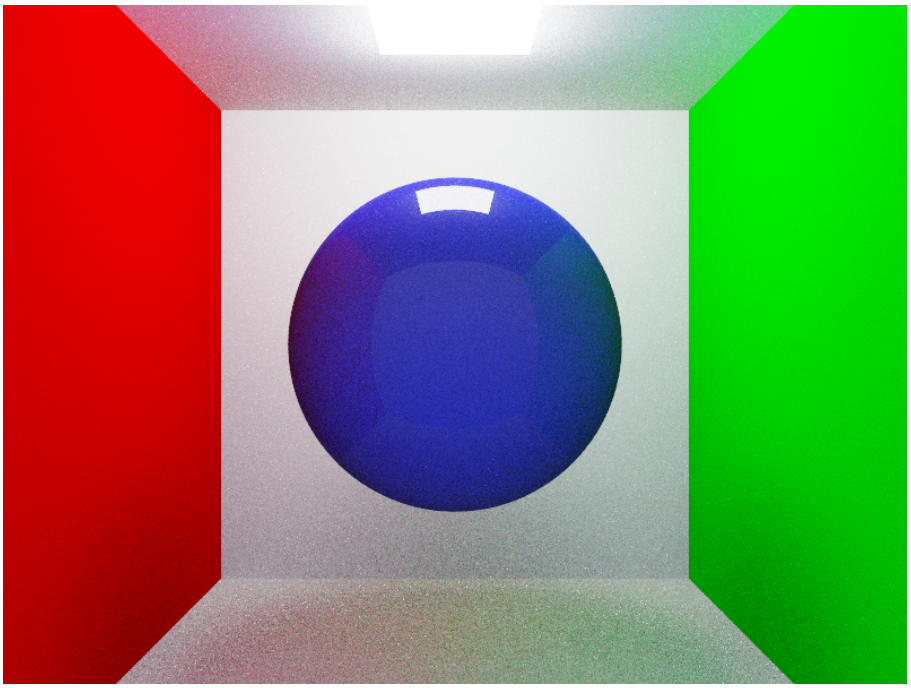

Figure 17: Task 6.2, 50 spp, 3 bounces.

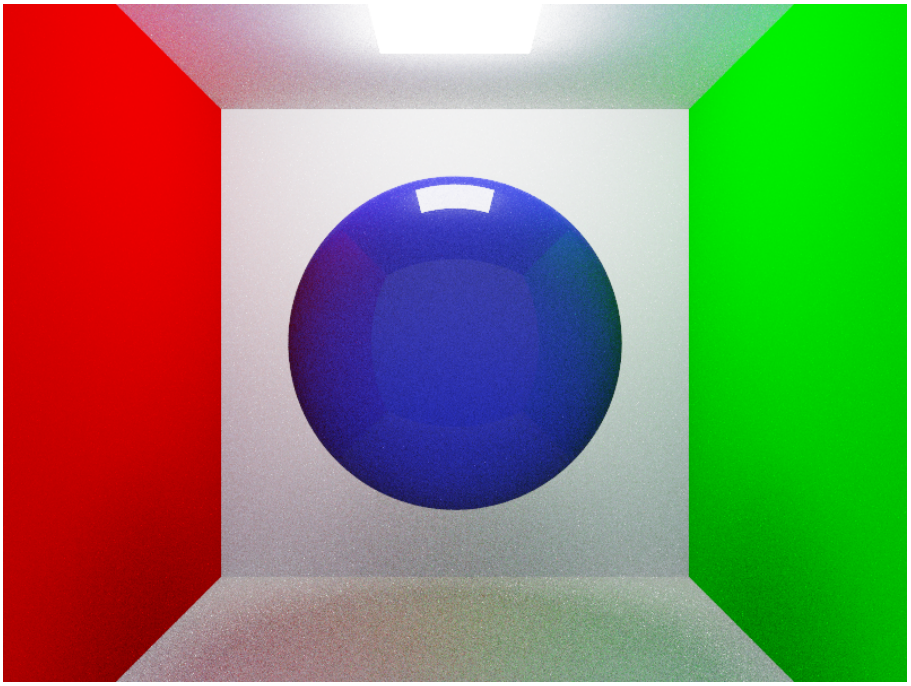

Figure 18: Task 6.2, 75 spp, 3 bounces.

Task 7: Do Something New

Overview

For our extension task, we implemented the following enhancements to our ray tracer:

- **Russian Roulette Path Termination** for improved performance.
- **Support for New Mesh Scenes: Teapot and Bunny**.
- **Mirror Material for the Bunny (Reflective Obj Rendering)**.

additions improve both rendering realism and ray tracing efficiency, while also demonstrating flexibility in scene composition.

Russian Roulette Termination

To improve efficiency, we implemented **Russian Roulette path termination**. After a minimum number of bounces, rays are randomly terminated with a fixed probability λ, and surviving rays are scaled to maintain unbiased estimation. We introduced a **minimum bounce threshold** to ensure rays always bounce a certain number of times before being considered for termination. This preserves critical indirect lighting detail and avoids premature termination of rays that contribute meaningfully to the final image.

```
// Russian Roulette with minimum bounce safeguard
const float termination_prob = 0.2f;
const int guaranteed_bounces = 2;

if (ray.n_bounces >= guaranteed_bounces) {
    float r = linearRand(0.0f, 1.0f);
    if (r < termination_prob) {
        ray.isWip = false; // Kill ray
        return ray;
    } else {
        ray.W_wip /= (1.0f - termination_prob); // Scale surviving rays
    }
}
```

This allowed us to increase the maximum bounce count without introducing bias or excessive computational cost. The tradeoff is increased image noise, which can be reduced with higher spp.

Custom Scenes: Teapot and Bunny

We created additional test scenes using '.obj' models:

- **Teapot-in-Box Scene**: A reflective blue teapot rendered within a Cornell-style box.
- **Bunny-in-Box Scene**: A Stanford bunny rendered as a mirror surface within the same environment.

To integrate the '.obj' models, we extended the scene system and added new scene IDs to the scene initializer. Each mesh was scaled, rotated, and centered for proper alignment in the scene. We also applied different materials to test glossiness and pure reflection.

Results

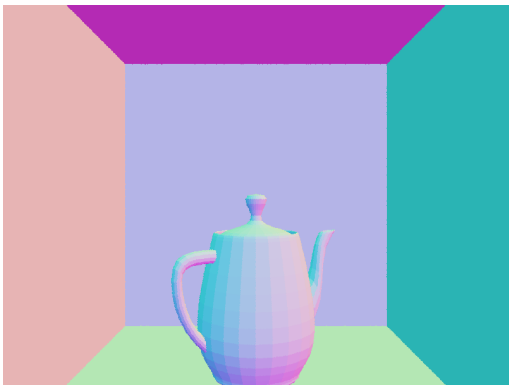

Figure 19: Custom teapot scene before ray tracing

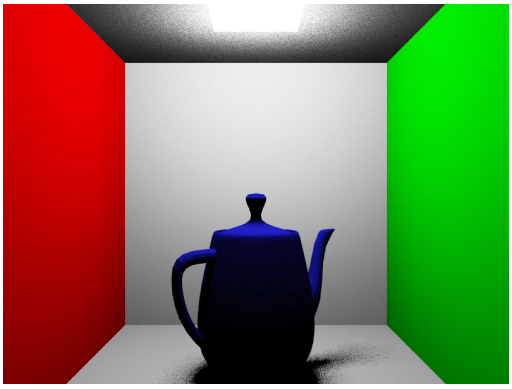

Figure 20: Custom teapot scene with 10 spp, no bounces.

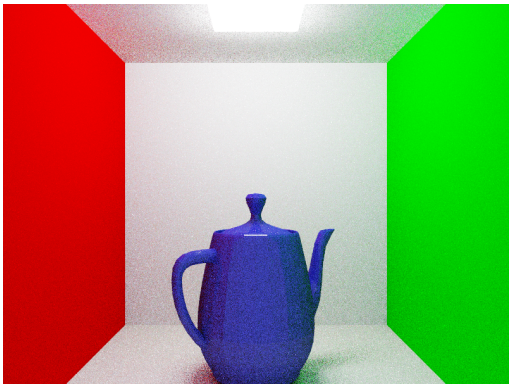

Figure 21: Custom teapot scene with glossy reflection.

Figure 22: Custom bunny scene before ray tracing.

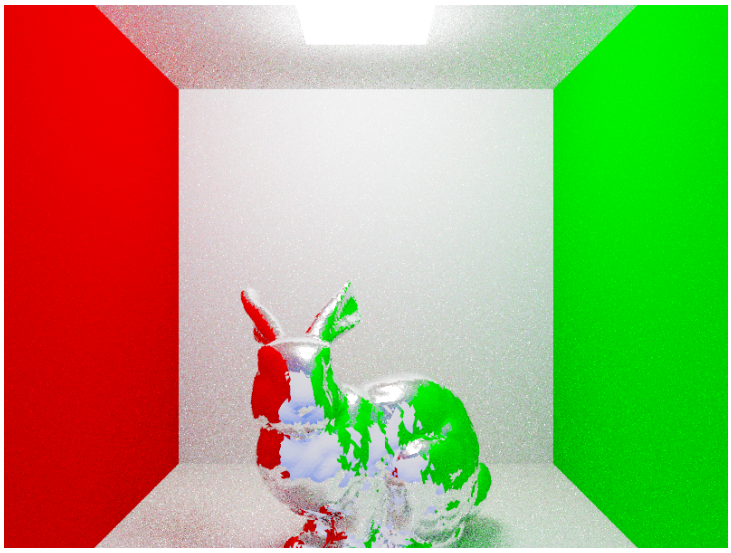

Figure 23: Custom bunny scene rendered with a mirror-like material.

Conclusion

This task demonstrated how easily our framework could be extended with new features. Russian Roulette added performance scalability, while the new mesh scenes showcased flexibility in scene composition and reflective rendering. These additions also proved useful for stress-testing our ray tracer with complex geometry.

Results and Discussion

Final Rendered Images

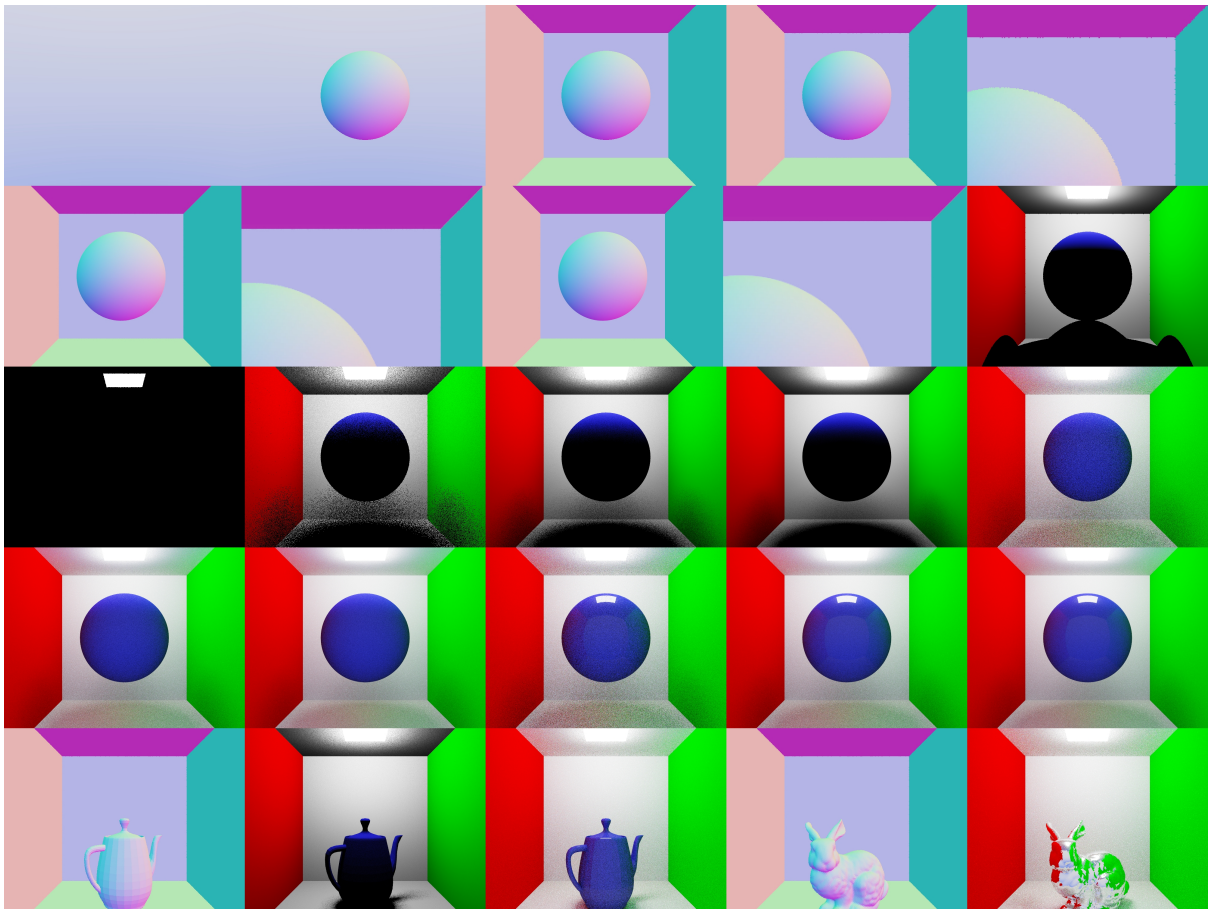

Figure 24: Compilation of all of our rendered scenes with reflections, soft shadows, and indirect lighting.

Challenges and Learning Outcomes

Throughout the project, we encountered several implementation challenges. Debugging issues related to normalized vectors and cosine-weighted sampling proved particularly tricky during the development of glossy reflections. Understanding how light should reflect across surfaces required careful handling of vector math and dot products.

Loading and rendering `.obj` files for complex meshes like the bunny and teapot also introduced unexpected bugs. Ensuring that triangle geometry, normals, and transformations were correctly processed took considerable effort. Additionally, as our scenes became more complex with added features such as soft shadows and multi-bounce reflections, rendering times increased significantly, which made testing and iteration more time-consuming.

Despite these challenges, the project was an incredibly rewarding experience. It provided hands-on insight into the foundations of physically-based rendering. While there is room for further optimization—particularly in our Task 7 extensions—we are proud of the progress made and the quality of our final rendered results.

Conclusion

This project implemented a fully functional ray tracer supporting realistic lighting through diffuse and specular shading, soft shadows, and recursive ray bounces. We added multiple physically-based rendering features including glossy materials, mirror-like reflections, and Russian Roulette path termination to improve performance while maintaining quality.

We also experimented with complex geometry such as teapots and bunnies, showcasing our tracer's ability to handle triangle meshes in addition to basic primitives. The final renders demonstrate our ray tracer's ability to simulate global illumination effects with high visual fidelity.

Future improvements could include bounding volume hierarchy (BVH) acceleration to reduce render times, adaptive sampling for noise reduction, and support for additional materials such as refraction or textured surfaces.