Automagic Raytracer Development Report

s2693586

October 2024

Contents

1 Introduction

1.1 Project Overview

This project aims to develop a raytracer from scratch with C++. A raytracer simulates the behaviour of light rays by tracing their paths as they interact with objects within the scene to produce photorealistic images.

The primary objective of this project is to implement essential rendering features, including:

- **Blinn-Phong shading** for simulating realistic lighting.
- **Reflection and refraction** to render mirrors and transparent materials.
- **Tone mapping** to adapt high dynamic range scenes for standard display devices.
- Handling of **ray-object** intersections for primitives such as spheres, triangles, and cylinders.

Beyond these foundational features, the project implemented advanced techniques to enhance performance and realism, such as:

- **Bounding Volume Hierarchies (BVH)** for faster intersection tests.
- **Antialiasing** via pixel sampling to smooth jagged edges.
- Realistic rendering features like **depth of field and aperture**.

Note: For building and running the project, please refer to "FeatureList.txt" for instructions.

1.2 Background Mathematics

1. Intersection of a ray with an object:

$$P = O + tD$$

   Where:

   - O is the origin of the ray.
   - D is the ray's direction vector.
   - t is the distance from the ray origin to the intersection point.

2. Reflection of a ray:

$$R = 2(N \cdot L)N - L$$

   Where:

   - N is the normal vector at the intersection point.
   - L is the direction vector of the incoming ray.
   - R is the direction vector of the reflected ray.

3. Refraction of a ray:

$$T = \eta_i \frac{-L + (N \cdot L)\hat{N}}{\left\| -\eta_i L + (N \cdot L)\hat{N} \right\|} - \eta_t \hat{N}$$

   Where:

   - η_i and η_t are the refractive indices of the two materials.
   - $\hat{N}$ is the normalised surface normal vector at the intersection point.
   - L is the direction vector of the incoming ray.
   - T is the direction vector of the refracted ray.

4. colour calculation:

$$C = E + \sum_{i=1}^{n} w_i C_i$$

   Where:

   - E is the emitted colour of the surface.
   - C_i is the reflected colour contribution from the i-th light source.
   - w_i is the transmission factor of the corresponding light path.

2 Implementation

2.1 High-Level Design

The primary components of this project are divided into modules for scalability, maintainability, and debugging, each of which is responsible for a specific aspect of the raytracing process. The overview is listed as below:

1. **Core**
   - **Vector3**: A class for 3D vector mathematics including addition, subtraction, normalisation, dot product and cross product
   - **Ray**: A class to represent a ray in 3D space with an origin and a direction mainly for tracing paths from the camera to the scene, also used for light, reflection and refraction rays.
2. **Scene**:
   - **Geometry**: A base class for all objects and provides interfaces for operations like ray-object intersection, centroid computation and bounding box calculations.
   - **Sphere, Cylinder, Triangle**: Specific intersection algorithm for each shape and methods to compute bounding boxes.
   - **Material**: Save the surface properties such as diffuse, specular coefficients, reflectivity, etc.
3. **Lighting**:

- **Light**: Represents point light in the scene, including position and intensity.

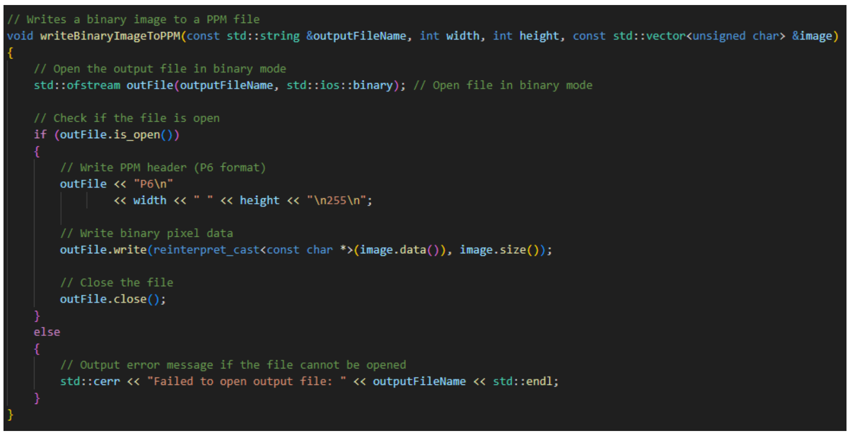
4. **Camera**:
   - **Camera**: Models the virtual camera, handles field of view, aspect ratio, depth of field, aperture, and generates rays for each pixel on the image plane.
5. **Shading**:
   - **Blinn-Phong Shading**: Shading for realistic lighting, including ambient, diffuse and specular components. Also handles reflective and refractive effects through recursive ray tracing.
   - **Binary**: Shade the pixel that hit any object red, otherwise black.
6. **Acceleration**:
   - **BVH**: Organises objects into a tree structure to optimise ray-object intersection, using Axis-Aligned Bounding Boxes to accelerate intersection queries.
7. **Input and Configuration**:
   - **JSON Reader**: Parse scene configuration from a JSON file.
8. **Rendering Pipeline**:
   - **Rendering loop**: Manages the overall rendering process, including all the above features, and supports antialiasing and tone mapping for enhanced visual realism.

2.2 Feature Implementation

Detailed descriptions of each feature:

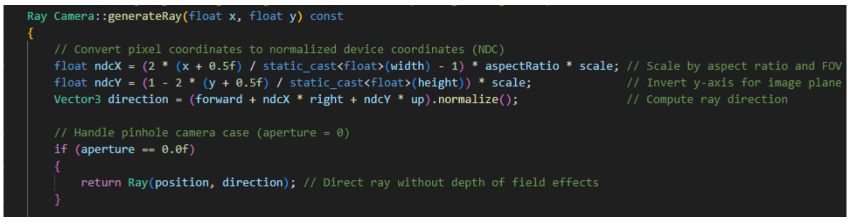
2.2.1 Basic Features

1. **Image writing**: Store the pixel data (RGB values) in a 1D buffer and ensure the data is clamped back to 0-255. Write the PPM header (P6 format) with image width, height, and maximum colour value. Then, write the pixel values in binary format to the file. See implementation in Figure 1.
2. **Virtual pin-hole camera**: Compute the forward (lookAt - position), right and up vectors to define the camera's orientation in 3D space. Determine the image plane's dimension based on the field of view, aspect ratio, and resolution, and generate rays for each pixel from the camera to the pixel position. See implementation in Figure 2.



```cpp
// Writes a binary image to a PPM file
void writeBinaryImageToPPM(const std::string &outputFileName, int width, int height, const std::vector<unsigned char> &image)
{
    // Open the output file in binary mode
    std::ofstream outFile(outputFileName, std::ios::binary); // Open file in binary mode

    // Check if the file is open
    if (outFile.is_open())
    {
        // Write PPM header (P6 format)
        outFile << "P6\n"
                << width << " " << height << "\n255\n";

        // Write binary pixel data
        outFile.write(reinterpret_cast<const char *>(image.data()), image.size());

        // Close the file
        outFile.close();
    }
    else
    {
        // Output error message if the file cannot be opened
        std::cerr << "Failed to open output file: " << outputFileName << std::endl;
    }
}
```

Figure 1: Code snippet for writing image.



```cpp
Ray Camera::generateRay(float x, float y) const
{
    // Convert pixel coordinates to normalized device coordinates (NDC)
    float ndcX = (2 * (x + 0.5f) / static_cast<float>(width) - 1) * aspectRatio * scale; // Scale by aspect ratio and FOV
    float ndcY = (1 - 2 * (y + 0.5f) / static_cast<float>(height)) * scale;             // Invert y-axis for image plane
    Vector3 direction = (forward + ndcX * right + ndcY * up).normalize();               // Compute ray direction

    // Handle pinhole camera case (aperture = 0)
    if (aperture == 0.0f)
    {
        return Ray(position, direction); // Direct ray without depth of field effects
    }
```

Figure 2: Code snippet for pin-hole camera.

3. **Intersection Test**:

- **Sphere**: Represent the sphere with its centre $\mathbf{C}$ and radius r.A ray is represented as:

$$\mathbf{R}(t) = \mathbf{O} + t \cdot \mathbf{D}$$

  where:

  - $\mathbf{O}$: Ray origin
  - $\mathbf{D}$: Ray direction

  Compute the vector $\mathbf{OC}$:

$$\mathbf{OC} = \mathbf{O} - \mathbf{C}$$

  Solve the quadratic equation:

$$a = \mathbf{D} \cdot \mathbf{D}, \quad b = 2 \cdot (\mathbf{OC} \cdot \mathbf{D}), \quad c = (\mathbf{OC} \cdot \mathbf{OC}) - r^2$$

  Compute the discriminant:

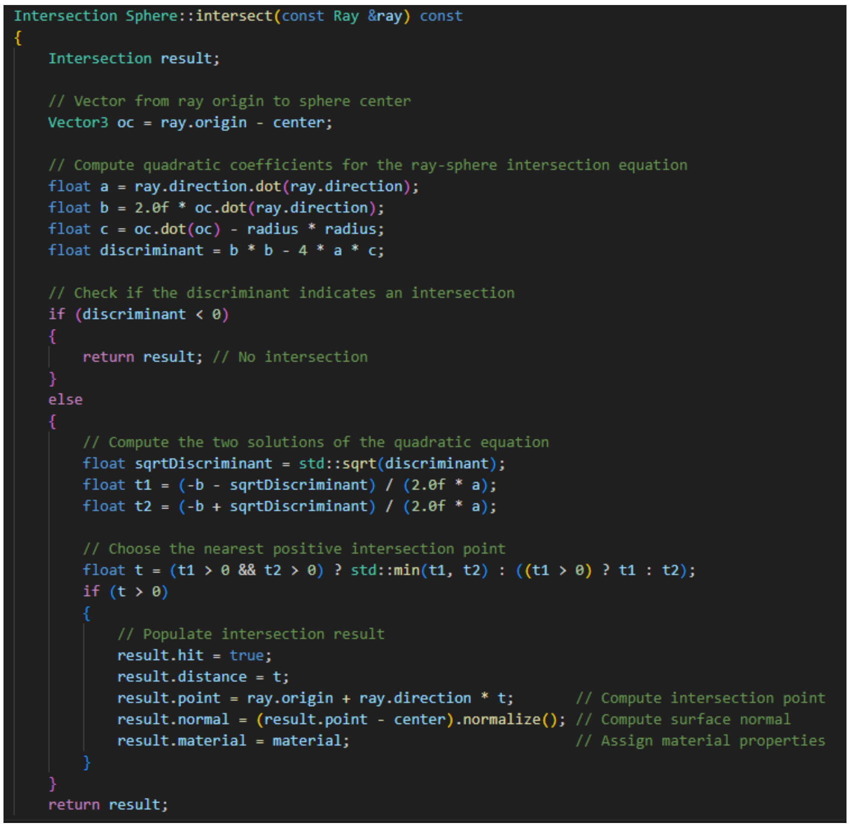
$$\text{Discriminant} = b^2 - 4 \cdot a \cdot c$$

  - If Discriminant < 0, the ray does not intersect the sphere.
  - Otherwise, compute the roots:

$$t_1 = \frac{-b - \sqrt{\text{Discriminant}}}{2a}, \quad t_2 = \frac{-b + \sqrt{\text{Discriminant}}}{2a}$$

  Determine the closest intersection point:

$$t = \min(t_1, t_2) \quad \text{if both roots are positive.}$$



```cpp
Intersection Sphere::intersect(const Ray &ray) const
{
    Intersection result;

    // Vector from ray origin to sphere center
    Vector3 oc = ray.origin - center;

    // Compute quadratic coefficients for the ray-sphere intersection equation
    float a = ray.direction.dot(ray.direction);
    float b = 2.0f * oc.dot(ray.direction);
    float c = oc.dot(oc) - radius * radius;
    float discriminant = b * b - 4 * a * c;

    // Check if the discriminant indicates an intersection
    if (discriminant < 0)
    {
        return result; // No intersection
    }
    else
    {
        // Compute the two solutions of the quadratic equation
        float sqrtDiscriminant = std::sqrt(discriminant);
        float t1 = (-b - sqrtDiscriminant) / (2.0f * a);
        float t2 = (-b + sqrtDiscriminant) / (2.0f * a);

        // Choose the nearest positive intersection point
        float t = (t1 > 0 && t2 > 0) ? std::min(t1, t2) : ((t1 > 0) ? t1 : t2);
        if (t > 0)
        {
            // Populate intersection result
            result.hit = true;
            result.distance = t;
            result.point = ray.origin + ray.direction * t;       // Compute intersection point
            result.normal = (result.point - center).normalize(); // Compute surface normal
            result.material = material;                          // Assign material properties
        }
    }
    return result;
```

Figure 3: Code snippet of intersection test for spheres.

- **Cylinder**:
  A cylinder is defined by its center $\mathbf{C}$, axis vector $\mathbf{A}$ (normalized), radius r, and height h. To determine whether a ray intersects with a finite cylinder, we follow these steps:
  First, the ray is represented as:
  $$\mathbf{R}(t) = \mathbf{O} + t \cdot \mathbf{D},$$
  where $\mathbf{O}$ is the ray origin, and $\mathbf{D}$ is the ray direction.
  The vector from the ray origin to the cylinder centre is computed as:
  $$\mathbf{OC} = \mathbf{O} - \mathbf{C}.$$
  Next, the ray direction and origin are projected onto a plane perpendicular to the cylinder's axis $\mathbf{A}$. The perpendicular components are:
  $$\mathbf{D}' = \mathbf{D} - (\mathbf{D} \cdot \mathbf{A}) \cdot \mathbf{A}, \quad \mathbf{O}' = \mathbf{OC} - (\mathbf{OC} \cdot \mathbf{A}) \cdot \mathbf{A}.$$
  The intersection of the ray with the infinite cylinder is determined by solving the quadratic equation:
  $$a = \mathbf{D}' \cdot \mathbf{D}', \quad b = 2 \cdot (\mathbf{O}' \cdot \mathbf{D}'), \quad c = \mathbf{O}' \cdot \mathbf{O}' - r^2,$$
  $$\text{Discriminant} = b^2 - 4 \cdot a \cdot c.$$

If Discriminant < 0, the ray does not intersect the cylinder. Otherwise, the two possible intersection points (roots) are:

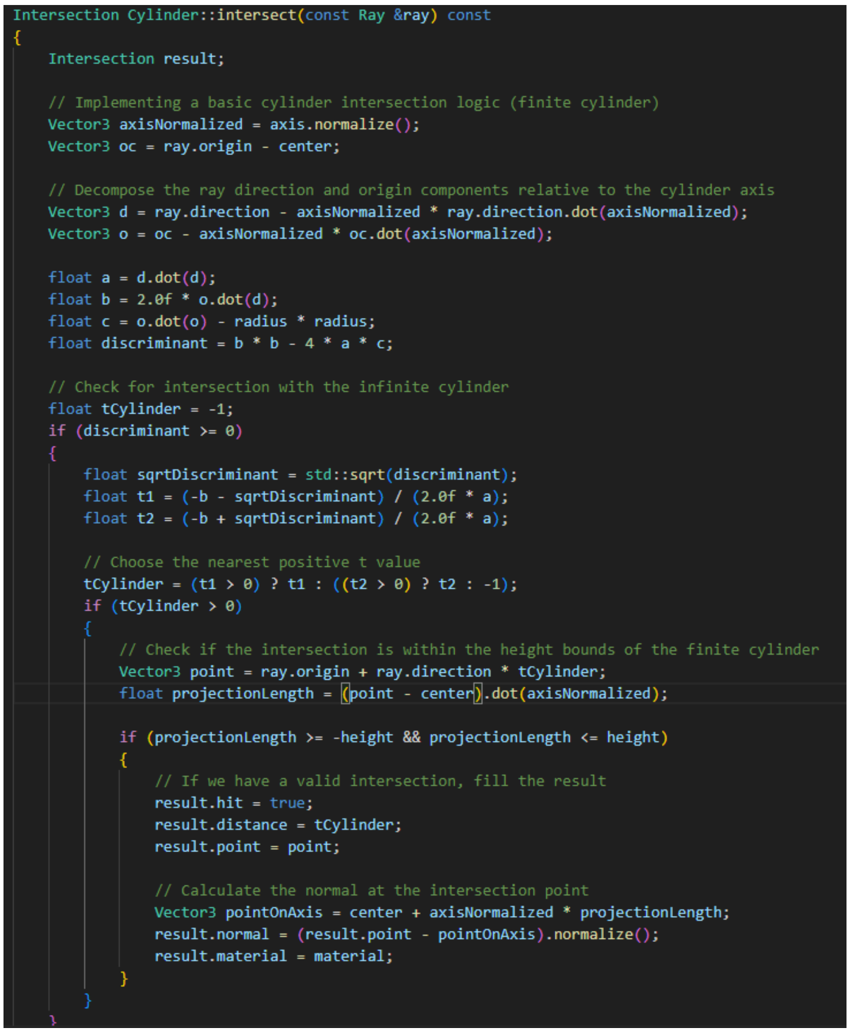
$$t_1 = \frac{-b - \sqrt{\text{Discriminant}}}{2a}, \quad t_2 = \frac{-b + \sqrt{\text{Discriminant}}}{2a}.$$

To validate that the intersection points are within the cylinder's height bounds, compute the projection length of the intersection points onto the cylinder's axis:

$$\text{Projection Length} = (\mathbf{P} - \mathbf{C}) \cdot \mathbf{A},$$

where $\mathbf{P}$ is the intersection point. The intersection is valid only if:

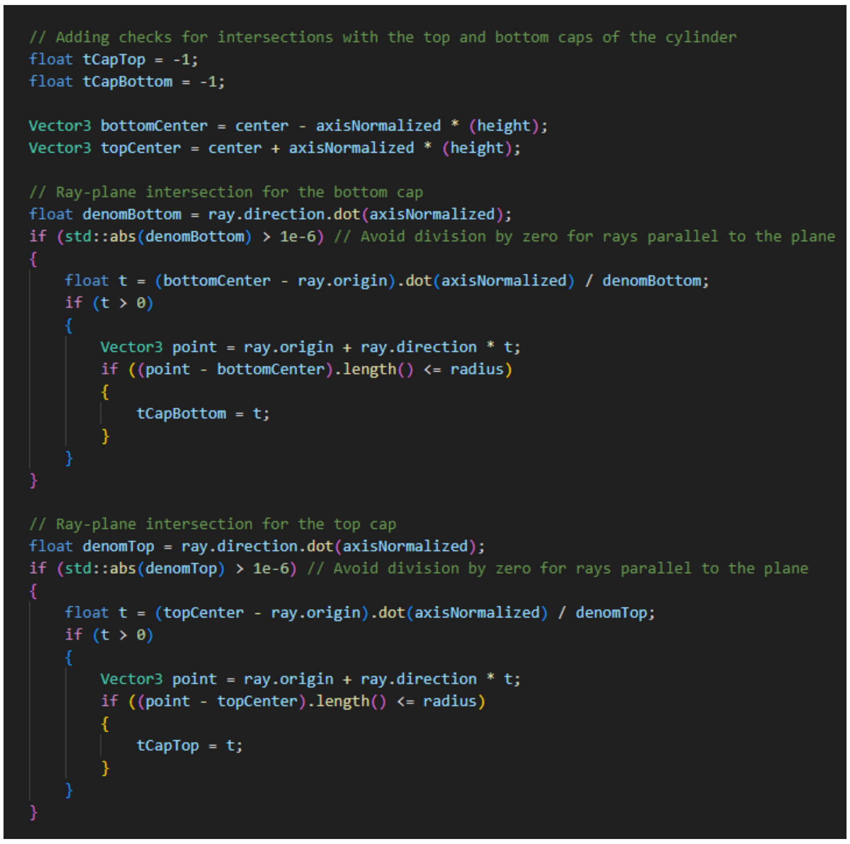
$$-h \leq \text{Projection Length} \leq h.$$



```
Intersection Cylinder::intersect(const Ray &ray) const
{
    Intersection result;

    // Implementing a basic cylinder intersection logic (finite cylinder)
    Vector3 axisNormalized = axis.normalize();
    Vector3 oc = ray.origin - center;

    // Decompose the ray direction and origin components relative to the cylinder axis
    Vector3 d = ray.direction - axisNormalized * ray.direction.dot(axisNormalized);
    Vector3 o = oc - axisNormalized * oc.dot(axisNormalized);

    float a = d.dot(d);
    float b = 2.0f * o.dot(d);
    float c = o.dot(o) - radius * radius;
    float discriminant = b * b - 4 * a * c;

    // Check for intersection with the infinite cylinder
    float tCylinder = -1;
    if (discriminant >= 0)
    {
        float sqrtDiscriminant = std::sqrt(discriminant);
        float t1 = (-b - sqrtDiscriminant) / (2.0f * a);
        float t2 = (-b + sqrtDiscriminant) / (2.0f * a);

        // Choose the nearest positive t value
        tCylinder = (t1 > 0) ? t1 : ((t2 > 0) ? t2 : -1);
        if (tCylinder > 0)
        {
            // Check if the intersection is within the height bounds of the finite cylinder
            Vector3 point = ray.origin + ray.direction * tCylinder;
            float projectionLength = (point - center).dot(axisNormalized);

            if (projectionLength >= -height && projectionLength <= height)
            {
                // If we have a valid intersection, fill the result
                result.hit = true;
                result.distance = tCylinder;
                result.point = point;

                // Calculate the normal at the intersection point
                Vector3 pointOnAxis = center + axisNormalized * projectionLength;
                result.normal = (result.point - pointOnAxis).normalize();
                result.material = material;
            }
        }
    }
```

Figure 4: Code snippet of intersection test for the body of cylinders.

Finally, check for intersections with the cylinder's top and bottom caps (treated as disks). The caps are defined by planes at $\mathbf{C} + h \cdot \mathbf{A}$ and $\mathbf{C} - h \cdot \mathbf{A}$. Ray-plane intersection tests determine if the ray intersects these disks within their radii. The closest valid intersection (cylinder body or caps) is selected. See implementation in Figure 4, 5



```cpp
// Adding checks for intersections with the top and bottom caps of the cylinder
float tCapTop = -1;
float tCapBottom = -1;

Vector3 bottomCenter = center - axisNormalized * (height);
Vector3 topCenter = center + axisNormalized * (height);

// Ray-plane intersection for the bottom cap
float denomBottom = ray.direction.dot(axisNormalized);
if (std::abs(denomBottom) > 1e-6) // Avoid division by zero for rays parallel to the plane
{
    float t = (bottomCenter - ray.origin).dot(axisNormalized) / denomBottom;
    if (t > 0)
    {
        Vector3 point = ray.origin + ray.direction * t;
        if ((point - bottomCenter).length() <= radius)
        {
            tCapBottom = t;
        }
    }
}

// Ray-plane intersection for the top cap
float denomTop = ray.direction.dot(axisNormalized);
if (std::abs(denomTop) > 1e-6) // Avoid division by zero for rays parallel to the plane
{
    float t = (topCenter - ray.origin).dot(axisNormalized) / denomTop;
    if (t > 0)
    {
        Vector3 point = ray.origin + ray.direction * t;
        if ((point - topCenter).length() <= radius)
        {
            tCapTop = t;
        }
    }
}
```

Figure 5: Code snippet of intersection test for the top and bottom of cylinders.

- **Triangle**:
  To determine whether a ray intersects a triangle, we use the Möller–Trumbore intersection algorithm. This algorithm is efficient and avoids explicitly calculating the triangle's plane equation.
  The triangle is defined by its three vertices $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$. The ray is represented as:
  $$\mathbf{R}(t) = \mathbf{O} + t \cdot \mathbf{D},$$
  where $\mathbf{O}$ is the ray origin, and $\mathbf{D}$ is the ray direction.
  First, calculate the edges of the triangle:
  $$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0, \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0.$$
  Next, compute the determinant to check if the ray and triangle are parallel:
  $$\mathbf{h} = \mathbf{D} \times \mathbf{e}_2, \quad a = \mathbf{e}_1 \cdot \mathbf{h}.$$
  If a is close to zero ($|a| < \epsilon$), the ray is parallel to the triangle, and there is no intersection.
  To continue, calculate the inverse of a and the vector from the ray origin to one of the triangle's vertices:
  $$f = \frac{1}{a}, \quad \mathbf{s} = \mathbf{O} - \mathbf{v}_0.$$

The barycentric coordinates u and v are computed to verify if the intersection point lies within the triangle:

$$u = f \cdot (\mathbf{s} \cdot \mathbf{h}).$$

If $u < 0$ or $u > 1$, the intersection lies outside the triangle.
Next, compute $\mathbf{q}$ and the second barycentric coordinate v:

$$\mathbf{q} = \mathbf{s} \times \mathbf{e}_1, \quad v = f \cdot (\mathbf{D} \cdot \mathbf{q}).$$

If $v < 0$ or $u + v > 1$, the intersection lies outside the triangle.
Finally, compute the intersection distance t:

$$t = f \cdot (\mathbf{e}_2 \cdot \mathbf{q}).$$

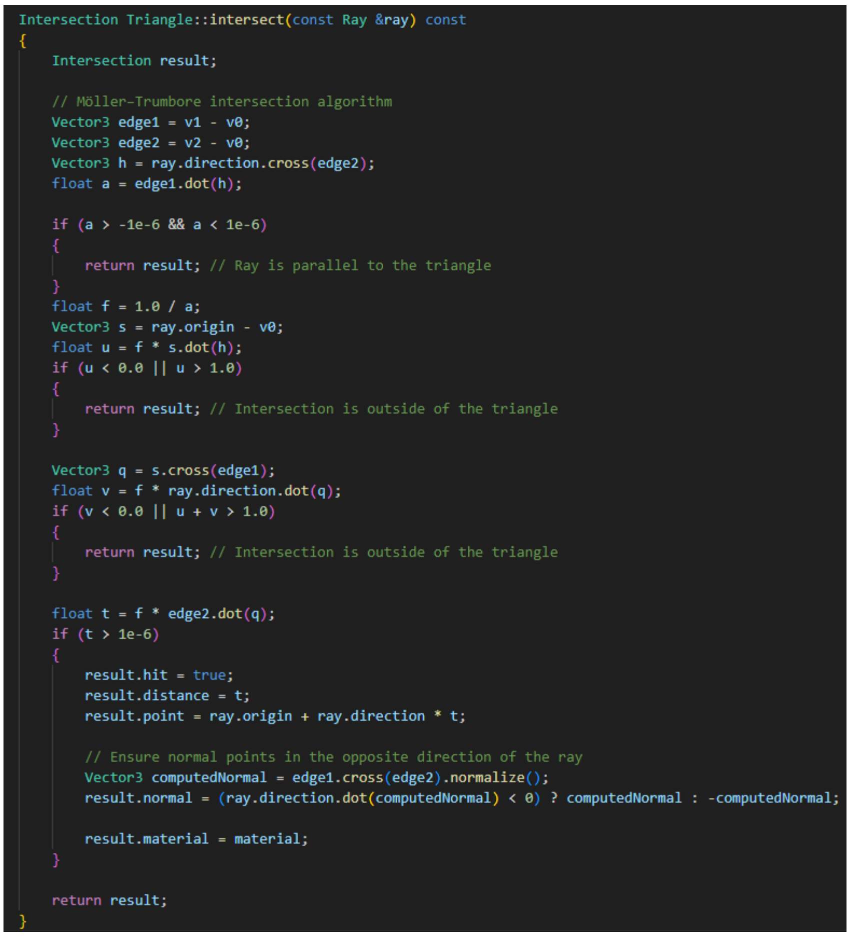
If $t > 0$, the ray intersects the triangle at:

$$\mathbf{P} = \mathbf{O} + t \cdot \mathbf{D}.$$

Ensure that the computed normal at the intersection points in the opposite direction of the ray:

$$\mathbf{N} = \mathbf{e}_1 \times \mathbf{e}_2, \quad \mathbf{N} = \begin{cases} \mathbf{N}, & \text{if } \mathbf{D} \cdot \mathbf{N} < 0, \\ -\mathbf{N}, & \text{otherwise.} \end{cases}$$

This ensures consistent orientation for shading.
The implementation of the triangle intersection test is shown in Figure 6.



```
Intersection Triangle::intersect(const Ray &ray) const
{
    Intersection result;

    // Möller-Trumbore intersection algorithm
    Vector3 edge1 = v1 - v0;
    Vector3 edge2 = v2 - v0;
    Vector3 h = ray.direction.cross(edge2);
    float a = edge1.dot(h);

    if (a > -1e-6 && a < 1e-6)
    {
        return result; // Ray is parallel to the triangle
    }
    float f = 1.0 / a;
    Vector3 s = ray.origin - v0;
    float u = f * s.dot(h);
    if (u < 0.0 || u > 1.0)
    {
        return result; // Intersection is outside of the triangle
    }

    Vector3 q = s.cross(edge1);
    float v = f * ray.direction.dot(q);
    if (v < 0.0 || u + v > 1.0)
    {
        return result; // Intersection is outside of the triangle
    }

    float t = f * edge2.dot(q);
    if (t > 1e-6)
    {
        result.hit = true;
        result.distance = t;
        result.point = ray.origin + ray.direction * t;

        // Ensure normal points in the opposite direction of the ray
        Vector3 computedNormal = edge1.cross(edge2).normalize();
        result.normal = (ray.direction.dot(computedNormal) < 0) ? computedNormal : -computedNormal;

        result.material = material;
    }

    return result;
}
```

Figure 6: Code snippet of intersection test for triangles.

4. **Blinn-Phong shading**: Blinn-Phong shading computes realistic lighting at a point on a surface. It combines ambient, diffuse, and specular lighting contributions to produce a visually pleasing result.

   The model assumes the following components:
   - A light source with intensity $\mathbf{L}$,
   - A surface normal $\mathbf{N}$,
   - A view direction $\mathbf{V}$ (towards the camera),
   - A light direction $\mathbf{L_D}$ (from the surface point to the light source).

   The shading model is described mathematically as:

$$\mathbf{C} = \mathbf{C}_{\text{ambient}} + \mathbf{C}_{\text{diffuse}} + \mathbf{C}_{\text{specular}},$$

   where $\mathbf{C}$ is the resulting colour at the surface point.

   Ambient lighting provides a base illumination to simulate light scattered in the environment:

$$\mathbf{C}_{\text{ambient}} = k_a \cdot \mathbf{C}_{\text{material}} \cdot \mathbf{C}_{\text{light}},$$

   where k_a is the ambient coefficient, $\mathbf{C}_{\text{material}}$ is the material's ambient colour, and $\mathbf{C}_{\text{light}}$ is the light's ambient colour.

   Diffuse lighting simulates light scattered equally in all directions from a surface. It is calculated as:

$$\mathbf{C}_{\text{diffuse}} = k_d \cdot \mathbf{C}_{\text{material}} \cdot \mathbf{C}_{\text{light}} \cdot \max(0, \mathbf{N} \cdot \mathbf{L_D}),$$

where k_d is the diffuse coefficient, and $\mathbf{N} \cdot \mathbf{L_D}$ is the cosine of the angle between the surface normal and the light direction.

Specular lighting simulates shiny highlights caused by light reflecting off a surface. The Blinn-Phong model uses a halfway vector $\mathbf{H}$ between the light direction and view direction:

$$\mathbf{H} = \frac{\mathbf{L_D} + \mathbf{V}}{\|\mathbf{L_D} + \mathbf{V}\|}.$$

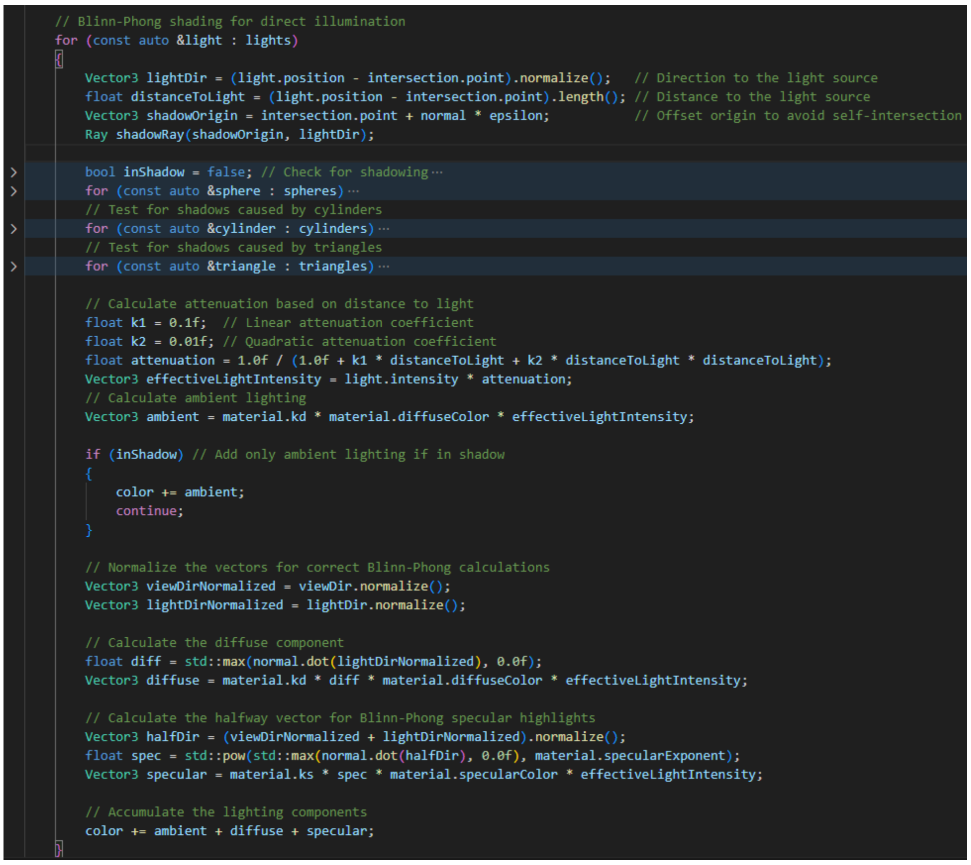
The specular component is then calculated as:

$$\mathbf{C}_{\text{specular}} = k_s \cdot \mathbf{C}_{\text{material}} \cdot \mathbf{C}_{\text{light}} \cdot \max(0, \mathbf{N} \cdot \mathbf{H})^n,$$

where k_s is the specular coefficient, n is the specular exponent controlling the shininess, and $\mathbf{N} \cdot \mathbf{H}$ is the cosine of the angle between the surface normal and the halfway vector.

The final shading computation at a surface point combines the three components:

$$\mathbf{C} = \mathbf{C}_{\text{ambient}} + \mathbf{C}_{\text{diffuse}} + \mathbf{C}_{\text{specular}}.$$



```
// Blinn-Phong shading for direct illumination
for (const auto &light : lights)
{
    Vector3 lightDir = (light.position - intersection.point).normalize();   // Direction to the light source
    float distanceToLight = (light.position - intersection.point).length(); // Distance to the light source
    Vector3 shadowOrigin = intersection.point + normal * epsilon;           // Offset origin to avoid self-intersection
    Ray shadowRay(shadowOrigin, lightDir);

    bool inShadow = false; // Check for shadowing···
    for (const auto &sphere : spheres)···
    // Test for shadows caused by cylinders
    for (const auto &cylinder : cylinders)···
    // Test for shadows caused by triangles
    for (const auto &triangle : triangles)···

    // Calculate attenuation based on distance to light
    float k1 = 0.1f;  // Linear attenuation coefficient
    float k2 = 0.01f; // Quadratic attenuation coefficient
    float attenuation = 1.0f / (1.0f + k1 * distanceToLight + k2 * distanceToLight * distanceToLight);
    Vector3 effectiveLightIntensity = light.intensity * attenuation;
    // Calculate ambient lighting
    Vector3 ambient = material.kd * material.diffuseColor * effectiveLightIntensity;

    if (inShadow) // Add only ambient lighting if in shadow
    {
        color += ambient;
        continue;
    }

    // Normalize the vectors for correct Blinn-Phong calculations
    Vector3 viewDirNormalized = viewDir.normalize();
    Vector3 lightDirNormalized = lightDir.normalize();

    // Calculate the diffuse component
    float diff = std::max(normal.dot(lightDirNormalized), 0.0f);
    Vector3 diffuse = material.kd * diff * material.diffuseColor * effectiveLightIntensity;

    // Calculate the halfway vector for Blinn-Phong specular highlights
    Vector3 halfDir = (viewDirNormalized + lightDirNormalized).normalize();
    float spec = std::pow(std::max(normal.dot(halfDir), 0.0f), material.specularExponent);
    Vector3 specular = material.ks * spec * material.specularColor * effectiveLightIntensity;

    // Accumulate the lighting components
    color += ambient + diffuse + specular;
}
```

Figure 7: Code snippet of Blinn-Phong shading.

Figure 8 shows an example of Blinn-Phong shading applied to a scene, demonstrating the ambient, diffuse, and specular contributions.

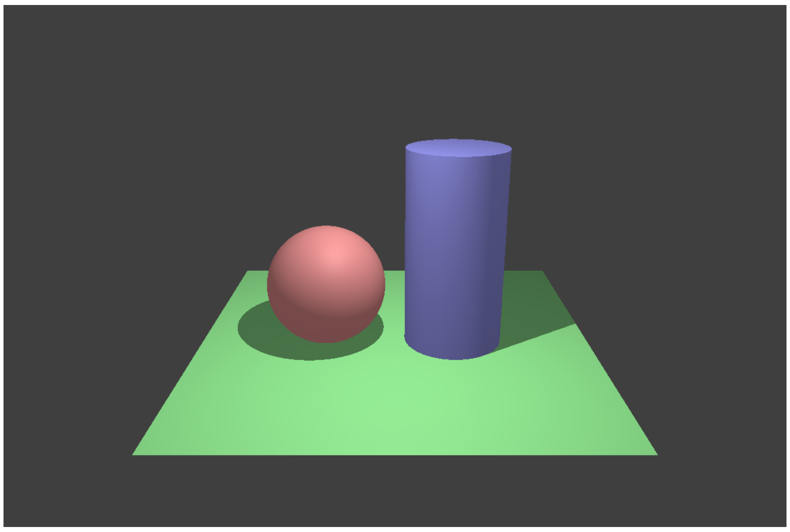


Figure 8: Example of Blinn-phong shading and shadow applied while rendering the image.

5. **Shadow:** A shadow ray $\mathbf{R}(t)$ is defined as:

$$\mathbf{R}(t) = \mathbf{P} + t \cdot \mathbf{L},$$

where:

- $\mathbf{P}$: Surface point.
- $\mathbf{L}$: Direction from $\mathbf{P}$ to the light source.
- t: Distance parameter along the ray.

The ray origin is slightly offset by ϵ along the surface normal to avoid self-intersection:

$$\mathbf{R}_{\text{origin}} = \mathbf{P} + \epsilon \cdot \mathbf{N}.$$

The implementation can be seen from Figure 7 and effects can be seen from Figure 8

6. **Tone Map**: Linear tone map is implemented based on the luminance of the input HDR colour.

Luminance Calculation: The luminance of the input HDR colour is computed to represent its perceptual brightness. It is calculated as a weighted sum of the RGB channels:

$$\text{Luminance} = 0.2126 \cdot \text{Red} + 0.7152 \cdot \text{Green} + 0.0722 \cdot \text{Blue}.$$

Tone Mapping: The luminance is adjusted using the tone mapping equation:

$$\text{ToneMappedLuminance} = \frac{\text{Luminance}}{\text{Luminance} + 1 + \epsilon},$$

where ϵ is a small value to prevent division by zero. The input colour is scaled by the ratio of the tone-mapped luminance to the original luminance:

$$\text{Mappedcolour} = \text{Inputcolour} \cdot \frac{\text{ToneMappedLuminance}}{\text{Luminance} + \epsilon}.$$

Gamma Correction: To enhance perceptual accuracy, gamma correction is applied:

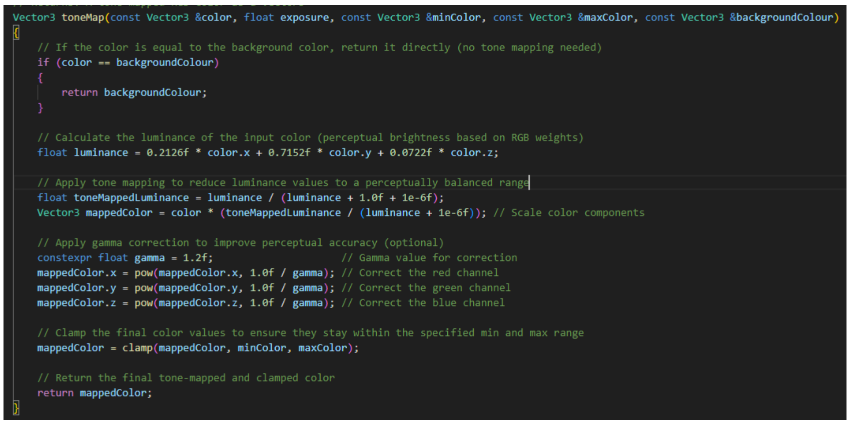
$$\text{Mappedcolour}_i = (\text{Mappedcolour}_i)^{\frac{1}{\gamma}},$$

where γ is the gamma value. In this implementation, $\gamma = 1.2$ is used.

Clamping: The final tone-mapped colour is clamped to ensure it remains within the specified minimum and maximum colour bounds:

$$\text{Clampedcolour}_i = \max(\text{Mincolour}_i, \min(\text{Mappedcolour}_i, \text{Maxcolour}_i)).$$

Background colour Handling: If the input colour matches the background colour, no tone mapping is applied, and the background colour is returned directly.



```cpp
Vector3 toneMap(const Vector3 &color, float exposure, const Vector3 &minColor, const Vector3 &maxColor, const Vector3 &backgroundColour)
{
    // If the color is equal to the background color, return it directly (no tone mapping needed)
    if (color == backgroundColour)
    {
        return backgroundColour;
    }

    // Calculate the luminance of the input color (perceptual brightness based on RGB weights)
    float luminance = 0.2126f * color.x + 0.7152f * color.y + 0.0722f * color.z;

    // Apply tone mapping to reduce luminance values to a perceptually balanced range
    float toneMappedLuminance = luminance / (luminance + 1.0f + 1e-6f);
    Vector3 mappedColor = color * (toneMappedLuminance / (luminance + 1e-6f)); // Scale color components

    // Apply gamma correction to improve perceptual accuracy (optional)
    constexpr float gamma = 1.2f;                    // Gamma value for correction
    mappedColor.x = pow(mappedColor.x, 1.0f / gamma); // Correct the red channel
    mappedColor.y = pow(mappedColor.y, 1.0f / gamma); // Correct the green channel
    mappedColor.z = pow(mappedColor.z, 1.0f / gamma); // Correct the blue channel

    // Clamp the final color values to ensure they stay within the specified min and max range
    mappedColor = clamp(mappedColor, minColor, maxColor);

    // Return the final tone-mapped and clamped color
    return mappedColor;
}
```

Figure 9: Code snippet of tone mapping.

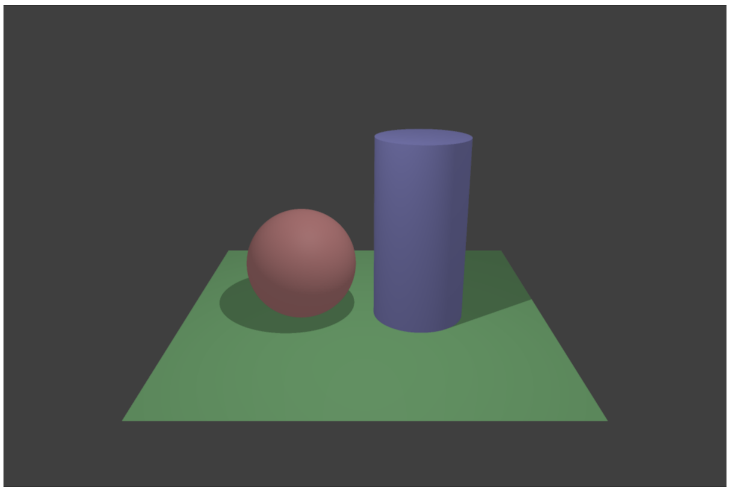


Figure 10: Effect of tone mapping on simple_phong.json.(antialiasing enabled)

By comparing Figure 7 and 10, we can see the effect of the tone map being applied to the image.

7. **Reflection**: Reflection simulates the way light bounces off surfaces, creating mirror-like effects. In ray tracing, reflections are calculated recursively by tracing secondary rays from the point of intersection.

   Mathematical Formulation: The direction of the reflected ray $\mathbf{R}$ is computed using the incident ray direction $\mathbf{I}$ and the surface normal $\mathbf{N}$:

$$\mathbf{R} = \mathbf{I} - 2 \cdot (\mathbf{I} \cdot \mathbf{N}) \cdot \mathbf{N}.$$

   Here:

   - $\mathbf{I}$: Direction of the incident ray (normalized).
   - $\mathbf{N}$: Surface normal at the intersection point (normalized).
   - $\mathbf{R}$: Direction of the reflected ray (normalized).

   Algorithm:

   (a) Compute the reflected ray direction $\mathbf{R}$ using the formula above.

   (b) Offset the ray origin slightly along $\mathbf{R}$ to avoid self-intersections:

$$\mathbf{R}_{\text{origin}} = \mathbf{P} + \epsilon \cdot \mathbf{R},$$

   where $\mathbf{P}$ is the intersection point and ϵ is a small constant.

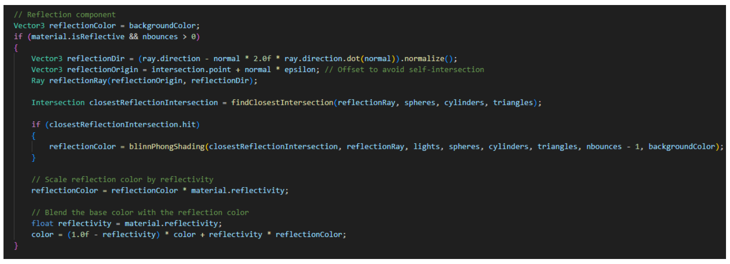
   (c) Trace the reflected ray into the scene to find the next intersection.

   (d) If an intersection is found, calculate the reflected color recursively. The recursion depth is limited to prevent infinite loops:

$$\mathbf{C}_{\text{reflected}} = \mathbf{C}_{\text{object}} + \text{reflectivity} \cdot \text{trace}(\mathbf{R}, \text{depth} - 1).$$

   (e) Blend the reflected color with the surface's base color using the material's reflectivity coefficient:

$$\mathbf{C} = (1 - \text{reflectivity}) \cdot \mathbf{C}_{\text{base}} + \text{reflectivity} \cdot \mathbf{C}_{\text{reflected}}.$$



```
// Reflection component
Vector3 reflectionColor = backgroundColor;
if (material.isReflective && nbounces > 0)
{
    Vector3 reflectionDir = (ray.direction - normal * 2.0f * ray.direction.dot(normal)).normalize();
    Vector3 reflectionOrigin = intersection.point + normal * epsilon; // Offset to avoid self-intersection
    Ray reflectionRay(reflectionOrigin, reflectionDir);

    Intersection closestReflectionIntersection = findClosestIntersection(reflectionRay, spheres, cylinders, triangles);

    if (closestReflectionIntersection.hit)
    {
        reflectionColor = blinnPhongShading(closestReflectionIntersection, reflectionRay, lights, spheres, cylinders, triangles, nbounces - 1, backgroundColor);
    }

    // Scale reflection color by reflectivity
    reflectionColor = reflectionColor * material.reflectivity;

    // Blend the base color with the reflection color
    float reflectivity = material.reflectivity;
    color = (1.0f - reflectivity) * color + reflectivity * reflectionColor;
}
```

Figure 11: Code snippet for reflection.

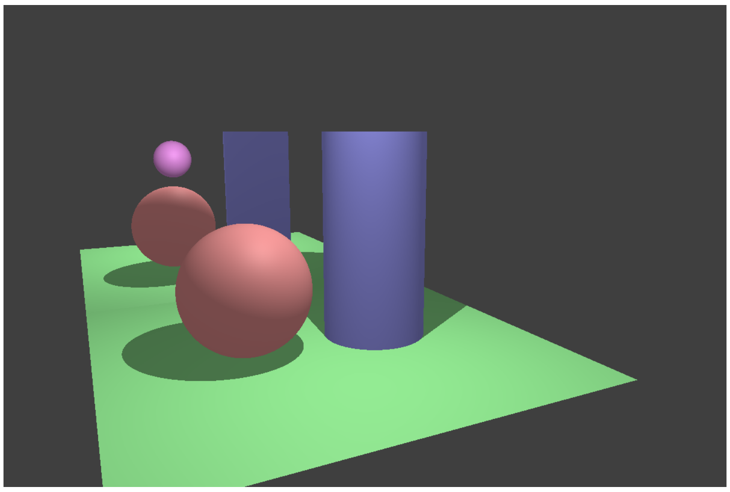


Figure 12: Effect of reflection in an image

8. **Refraction**: Refraction simulates the bending of light as it passes through a transparent surface. In ray tracing, refraction is computed using Snell's Law, which describes how light changes direction when transitioning between media with different refractive indices.

   Mathematical Formulation: The direction of the refracted ray $\mathbf{T}$ is calculated using the incident ray direction $\mathbf{I}$, the surface normal $\mathbf{N}$, and the ratio of refractive indices $\eta = \frac{\eta_i}{\eta_t}$, where:

   - η_i: Refractive index of the medium the ray is exiting.
   - η_t: Refractive index of the medium the ray is entering.

   Snell's Law for refraction is given by:

   $$\mathbf{T} = \eta \cdot \mathbf{I} + (\eta \cdot \cos\theta_i - \sqrt{1 - \eta^2 \cdot (1 - \cos^2\theta_i)}) \cdot \mathbf{N},$$

   where $\cos\theta_i = -\mathbf{I} \cdot \mathbf{N}$.

   Total Internal Reflection: If the discriminant inside the square root becomes negative, total internal reflection occurs, and no refraction is possible.

   —

   Algorithm:

   (a) Compute $\eta = \frac{\eta_i}{\eta_t}$ based on the refractive indices of the media.

   (b) Calculate $\cos\theta_i = -\mathbf{I} \cdot \mathbf{N}$. Adjust the normal $\mathbf{N}$ if the ray is exiting the surface.

   (c) Compute the refracted direction $\mathbf{T}$ using Snell's Law. If the discriminant is negative, handle total internal reflection.

   (d) Offset the refracted ray's origin slightly along $\mathbf{T}$ to avoid self-intersection:

   $$\mathbf{T}_{\text{origin}} = \mathbf{P} + \epsilon \cdot \mathbf{T}.$$

   (e) Trace the refracted ray into the scene to compute the refracted color recursively.

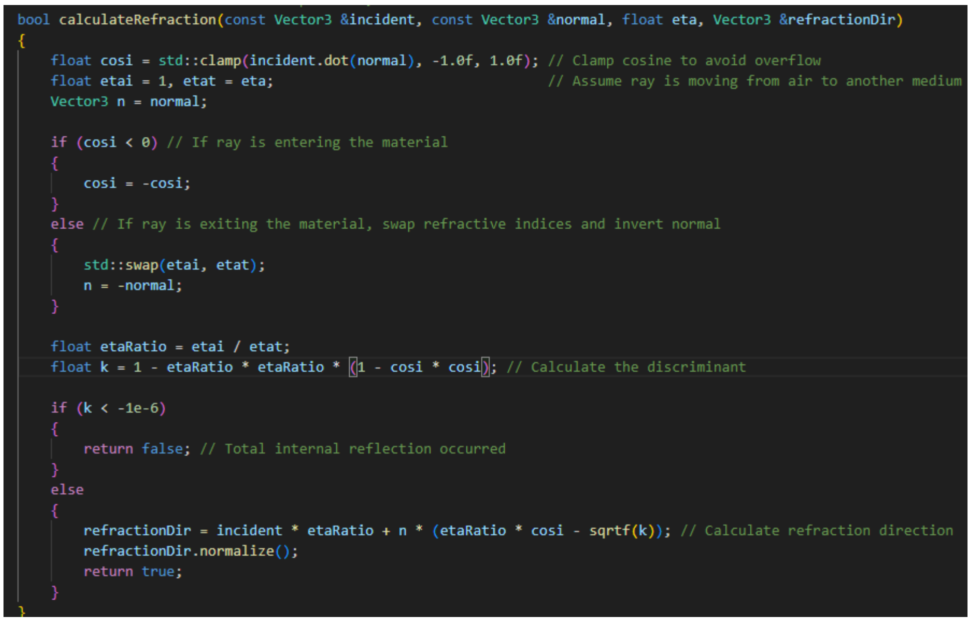
(f) Combine the refracted and reflected colors using the Fresnel reflectance factor R_f:

$$\mathbf{C} = (1 - R_f) \cdot \mathbf{C}_{\text{refracted}} + R_f \cdot \mathbf{C}_{\text{reflected}},$$

where R_f is computed using Schlick's approximation:

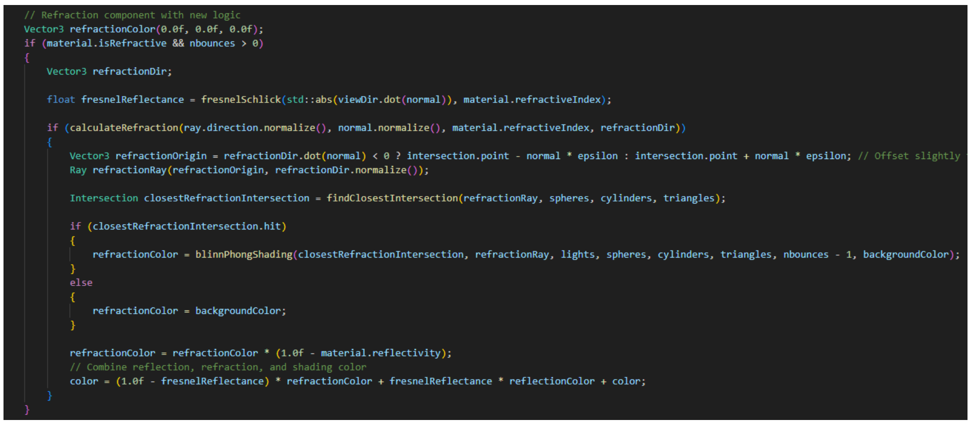
$$R_f = R_0 + (1 - R_0) \cdot (1 - \cos\theta_i)^5,$$

and $R_0 = \left(\frac{\eta_i - \eta_t}{\eta_i + \eta_t}\right)^2$.



```cpp
bool calculateRefraction(const Vector3 &incident, const Vector3 &normal, float eta, Vector3 &refractionDir)
{
    float cosi = std::clamp(incident.dot(normal), -1.0f, 1.0f); // Clamp cosine to avoid overflow
    float etai = 1, etat = eta;                                 // Assume ray is moving from air to another medium
    Vector3 n = normal;

    if (cosi < 0) // If ray is entering the material
    {
        cosi = -cosi;
    }
    else // If ray is exiting the material, swap refractive indices and invert normal
    {
        std::swap(etai, etat);
        n = -normal;
    }

    float etaRatio = etai / etat;
    float k = 1 - etaRatio * etaRatio * (1 - cosi * cosi); // Calculate the discriminant

    if (k < -1e-6)
    {
        return false; // Total internal reflection occurred
    }
    else
    {
        refractionDir = incident * etaRatio + n * (etaRatio * cosi - sqrtf(k)); // Calculate refraction direction
        refractionDir.normalize();
        return true;
    }
}
```

Figure 13: Code snippet for calculating refractions.



```cpp
// Refraction component with new logic
Vector3 refractionColor(0.0f, 0.0f, 0.0f);
if (material.isRefractive && nbounces > 0)
{
    Vector3 refractionDir;

    float fresnelReflectance = fresnelSchlick(std::abs(viewDir.dot(normal)), material.refractiveIndex);

    if (calculateRefraction(ray.direction.normalize(), normal.normalize(), material.refractiveIndex, refractionDir))
    {
        Vector3 refractionOrigin = refractionDir.dot(normal) < 0 ? intersection.point - normal * epsilon : intersection.point + normal * epsilon; // Offset slightly
        Ray refractionRay(refractionOrigin, refractionDir.normalize());

        Intersection closestRefractionIntersection = findClosestIntersection(refractionRay, spheres, cylinders, triangles);

        if (closestRefractionIntersection.hit)
        {
            refractionColor = blinnPhongShading(closestRefractionIntersection, refractionRay, lights, spheres, cylinders, triangles, nbounces - 1, backgroundColor);
        }
        else
        {
            refractionColor = backgroundColor;
        }

        refractionColor = refractionColor * (1.0f - material.reflectivity);
        // Combine reflection, refraction, and shading color
        color = (1.0f - fresnelReflectance) * refractionColor + fresnelReflectance * reflectionColor + color;
    }
}
```

Figure 14: Code snippet for applying refraction in blinn-phong rendering.

Figure 13 and 14 show the implementation of refraction in the code, and Figure 15 shows the rendered image with a refractive material.

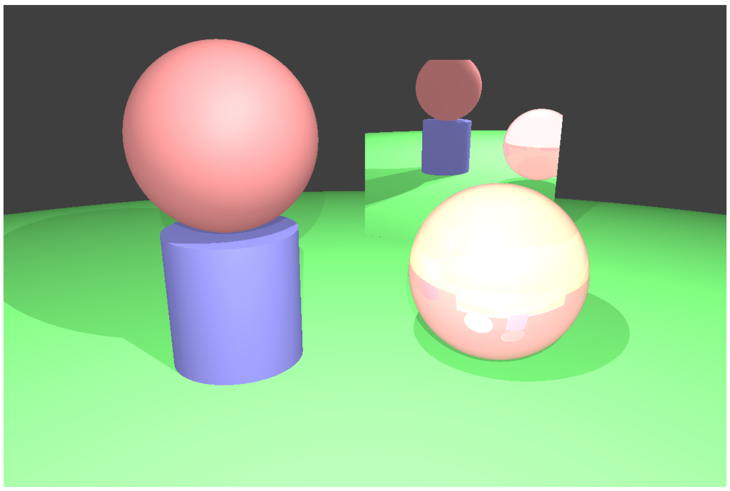


Figure 15: Effect of refraction applied.

2.2.2 Intermediate Features

1. **Bounding Volume Hierarchy(BVH)**: A Bounding Volume Hierarchy (BVH) is a tree structure used to accelerate ray-object intersection tests in ray tracing. BVH organises scene geometry into a hierarchy of bounding volumes, allowing efficient pruning of objects that do not intersect with a ray. This significantly reduces the number of intersection tests, improving rendering performance.

   Each node in the BVH represents a bounding volume that encompasses a subset of the scene's geometry:

   - **Leaf Node:** Contains a small number of geometric primitives (e.g., triangles, spheres, or cylinders).
   - **Internal Node:** Contains two or more child nodes and a bounding box that encloses all its children's volumes.

   The bounding volumes are typically axis-aligned bounding boxes (AABBs) for their computational efficiency.

   The BVH is built recursively:

   $$\text{BVHNode} = \text{Build}(\text{primitives}, \text{depth}),$$

   where:

   - Primitives are the scene's geometric objects.
   - Depth is the current level in the BVH tree.

   Steps to build the BVH:

   (a) Compute an AABB for the entire set of primitives.

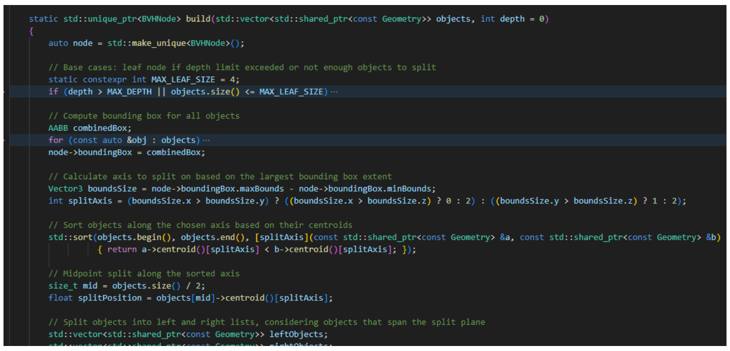
   (b) Partition the primitives into two groups based on a splitting criterion (e.g., midpoint or surface area heuristic).

   (c) Recursively build BVH nodes for each partition until a termination criterion is met (e.g., a minimum number of primitives per leaf node).

Ray traversal through the BVH efficiently determines potential intersections:

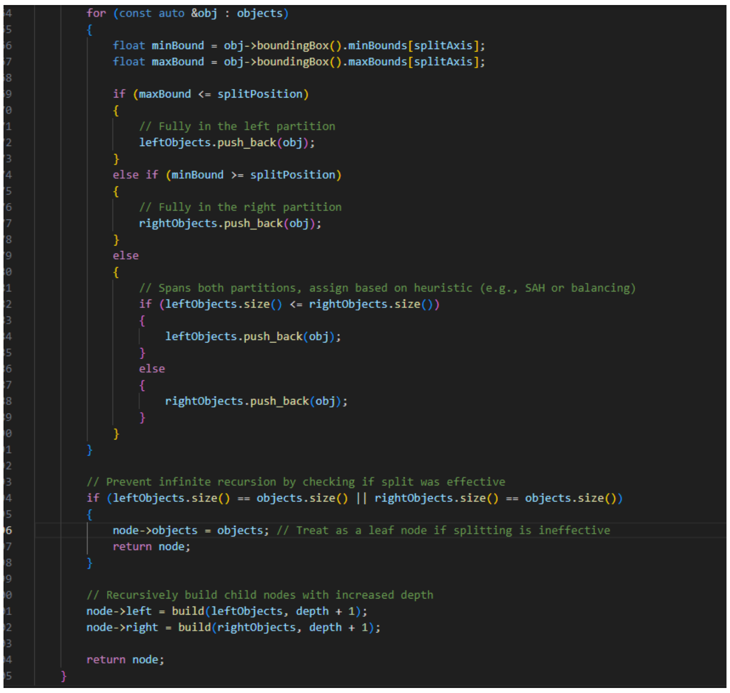
- Start at the root node of the BVH.
- Test the ray against the bounding box of the current node.
- If the ray intersects:
  - For leaf nodes, test the ray against the contained primitives.
  - For internal nodes, traverse the child nodes recursively.
- If no intersection occurs, terminate the traversal for that node.

The below figures show the implementation of BVH on top of the Blinn-Phong shading that has already been implemented. The performance of BVH will be discussed in Section 3.



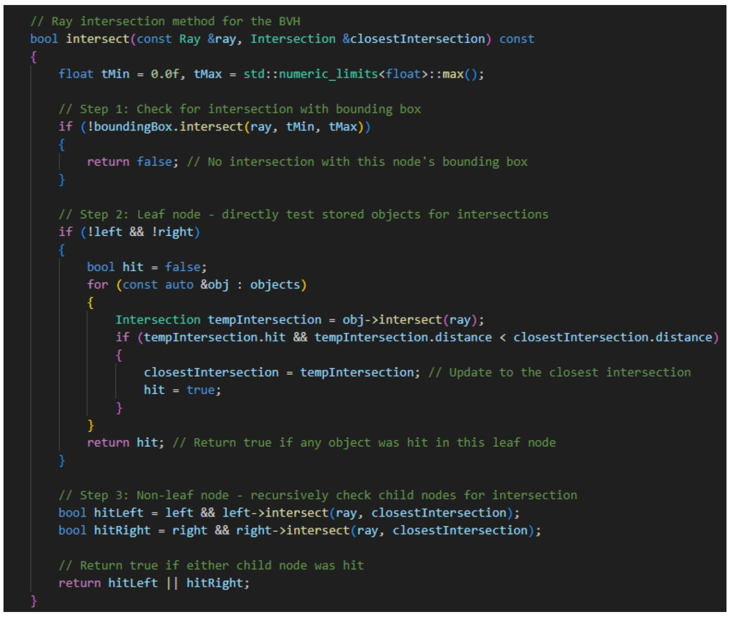
```cpp
static std::unique_ptr<BVHNode> build(std::vector<std::shared_ptr<const Geometry>> objects, int depth = 0)
{
    auto node = std::make_unique<BVHNode>();

    // Base cases: leaf node if depth limit exceeded or not enough objects to split
    static constexpr int MAX_LEAF_SIZE = 4;
    if (depth > MAX_DEPTH || objects.size() <= MAX_LEAF_SIZE) ···

    // Compute bounding box for all objects
    AABB combinedBox;
    for (const auto &obj : objects) ···
    node->boundingBox = combinedBox;

    // Calculate axis to split on based on the largest bounding box extent
    Vector3 boundsSize = node->boundingBox.maxBounds - node->boundingBox.minBounds;
    int splitAxis = (boundsSize.x > boundsSize.y) ? ((boundsSize.x > boundsSize.z) ? 0 : 2) : ((boundsSize.y > boundsSize.z) ? 1 : 2);

    // Sort objects along the chosen axis based on their centroids
    std::sort(objects.begin(), objects.end(), [splitAxis](const std::shared_ptr<const Geometry> &a, const std::shared_ptr<const Geometry> &b)
              { return a->centroid()[splitAxis] < b->centroid()[splitAxis]; });

    // Midpoint split along the sorted axis
    size_t mid = objects.size() / 2;
    float splitPosition = objects[mid]->centroid()[splitAxis];

    // Split objects into left and right lists, considering objects that span the split plane
    std::vector<std::shared_ptr<const Geometry>> leftObjects;
```

Figure 16: Part one of function for building BVH boxes.



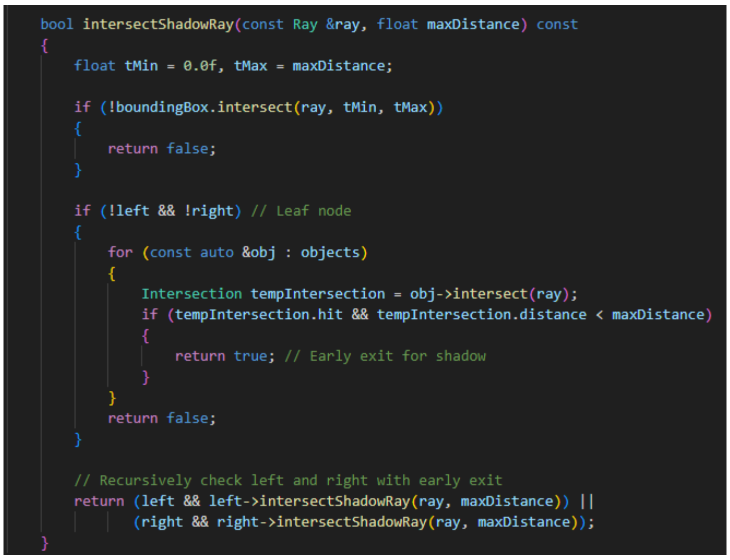
```cpp
        for (const auto &obj : objects)
        {
            float minBound = obj->boundingBox().minBounds[splitAxis];
            float maxBound = obj->boundingBox().maxBounds[splitAxis];

            if (maxBound <= splitPosition)
            {
                // Fully in the left partition
                leftObjects.push_back(obj);
            }
            else if (minBound >= splitPosition)
            {
                // Fully in the right partition
                rightObjects.push_back(obj);
            }
            else
            {
                // Spans both partitions, assign based on heuristic (e.g., SAH or balancing)
                if (leftObjects.size() <= rightObjects.size())
                {
                    leftObjects.push_back(obj);
                }
                else
                {
                    rightObjects.push_back(obj);
                }
            }
        }

        // Prevent infinite recursion by checking if split was effective
        if (leftObjects.size() == objects.size() || rightObjects.size() == objects.size())
        {
            node->objects = objects; // Treat as a leaf node if splitting is ineffective
            return node;
        }

        // Recursively build child nodes with increased depth
        node->left = build(leftObjects, depth + 1);
        node->right = build(rightObjects, depth + 1);

        return node;
    }
```

Figure 17: Part two of function for building BVH boxes.



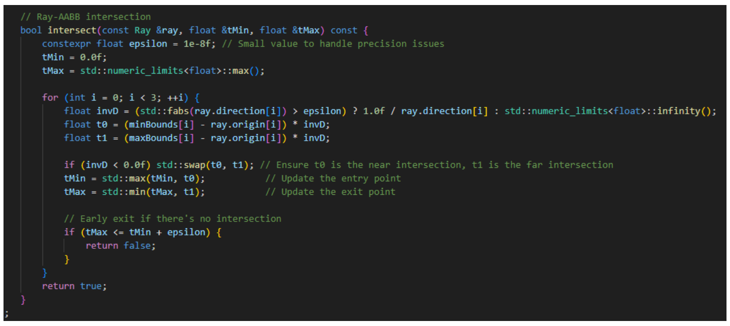
```cpp
// Ray intersection method for the BVH
bool intersect(const Ray &ray, Intersection &closestIntersection) const
{
    float tMin = 0.0f, tMax = std::numeric_limits<float>::max();

    // Step 1: Check for intersection with bounding box
    if (!boundingBox.intersect(ray, tMin, tMax))
    {
        return false; // No intersection with this node's bounding box
    }

    // Step 2: Leaf node - directly test stored objects for intersections
    if (!left && !right)
    {
        bool hit = false;
        for (const auto &obj : objects)
        {
            Intersection tempIntersection = obj->intersect(ray);
            if (tempIntersection.hit && tempIntersection.distance < closestIntersection.distance)
            {
                closestIntersection = tempIntersection; // Update to the closest intersection
                hit = true;
            }
        }
        return hit; // Return true if any object was hit in this leaf node
    }

    // Step 3: Non-leaf node - recursively check child nodes for intersection
    bool hitLeft = left && left->intersect(ray, closestIntersection);
    bool hitRight = right && right->intersect(ray, closestIntersection);

    // Return true if either child node was hit
    return hitLeft || hitRight;
}
```

Figure 18: BVH box intersection function.



```cpp
bool intersectShadowRay(const Ray &ray, float maxDistance) const
{
    float tMin = 0.0f, tMax = maxDistance;

    if (!boundingBox.intersect(ray, tMin, tMax))
    {
        return false;
    }

    if (!left && !right) // Leaf node
    {
        for (const auto &obj : objects)
        {
            Intersection tempIntersection = obj->intersect(ray);
            if (tempIntersection.hit && tempIntersection.distance < maxDistance)
            {
                return true; // Early exit for shadow
            }
        }
        return false;
    }

    // Recursively check left and right with early exit
    return (left && left->intersectShadowRay(ray, maxDistance)) ||
           (right && right->intersectShadowRay(ray, maxDistance));
}
```

Figure 19: BVH box shadow intersection function.



```cpp
// Ray-AABB intersection
bool intersect(const Ray &ray, float &tMin, float &tMax) const {
    constexpr float epsilon = 1e-8f; // Small value to handle precision issues
    tMin = 0.0f;
    tMax = std::numeric_limits<float>::max();

    for (int i = 0; i < 3; ++i) {
        float invD = (std::fabs(ray.direction[i]) > epsilon) ? 1.0f / ray.direction[i] : std::numeric_limits<float>::infinity();
        float t0 = (minBounds[i] - ray.origin[i]) * invD;
        float t1 = (maxBounds[i] - ray.origin[i]) * invD;

        if (invD < 0.0f) std::swap(t0, t1); // Ensure t0 is the near intersection, t1 is the far intersection
        tMin = std::max(tMin, t0);           // Update the entry point
        tMax = std::min(tMax, t1);           // Update the exit point

        // Early exit if there's no intersection
        if (tMax <= tMin + epsilon) {
            return false;
        }
    }
    return true;
}
;
```

Figure 20: AABB helper function for constructing BVH.

2.2.3 Advanced Features

1. **Pixel Sampling for Antialiasing**: Antialiasing is a technique used to reduce visual artefacts, such as jagged edges, that occur due to insufficient sampling of the scene. Pixel sampling achieves smoother results by averaging multiple samples taken within each pixel. This process is crucial for producing high-quality rendered images.

   (a) Divide each pixel into subregions based on the number of samples (e.g., 4×4).

   (b) Generate sample positions using one of the sampling methods (e.g., jittered sampling).

   (c) For each sample position, cast a ray into the scene and compute the color at the intersection point.

   (d) Average the colors of all samples to compute the final pixel color:

   $$\mathbf{C}_{\text{pixel}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{C}_i,$$

   where N is the total number of samples, and $\mathbf{C}_i$ is the color computed for each sample.

   An example plot of the random samples algorithm is given in Firgure 21

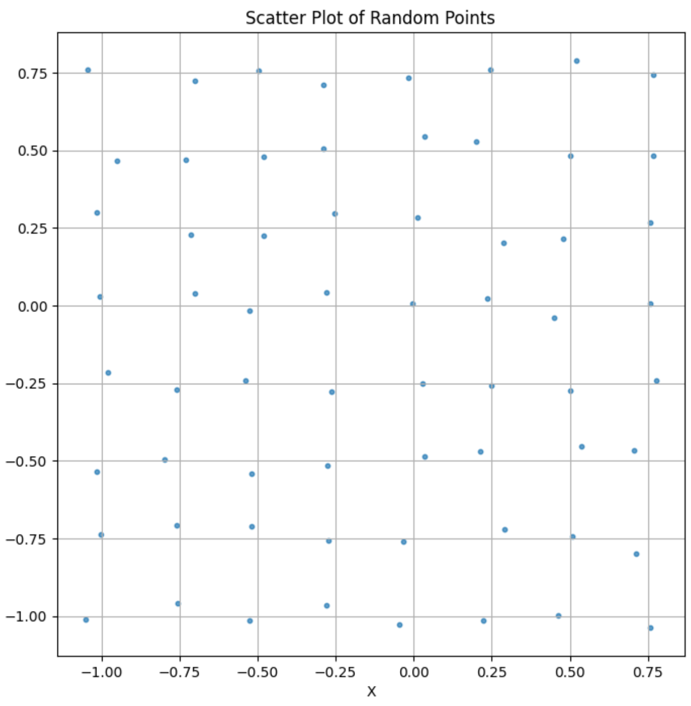
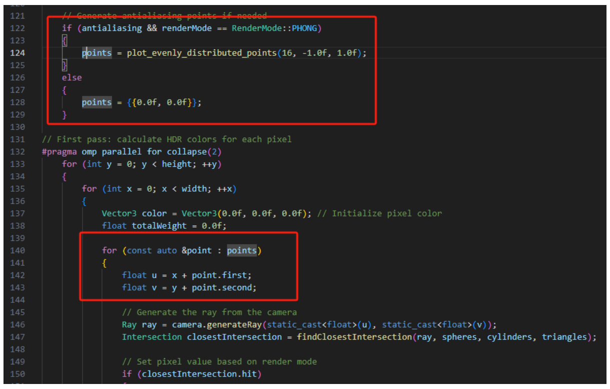


Figure 21: Plot of the randomly sampled pixel.

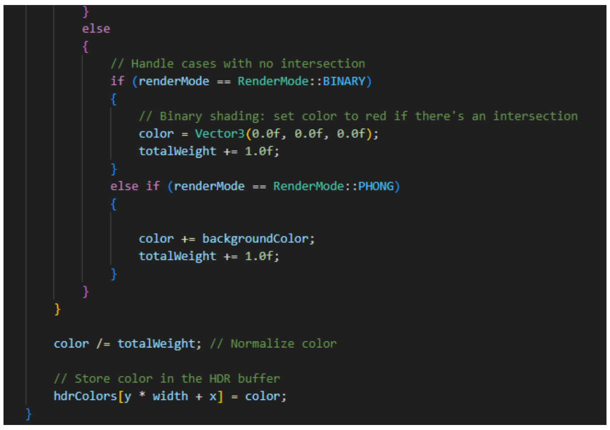
Below are the code snippets for random sampling of the pixels.

By applying 16 samples to each pixel we can achieve a fantastic result on antialiasing as shown in Figure 25 and 26



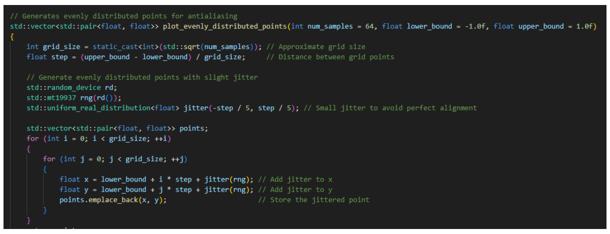
```cpp
    // Generate antialiasing points if needed
    if (antialiasing && renderMode == RenderMode::PHONG)
    {
        points = plot_evenly_distributed_points(16, -1.0f, 1.0f);
    }
    else
    {
        points = {{0.0f, 0.0f}};
    }

// First pass: calculate HDR colors for each pixel
#pragma omp parallel for collapse(2)
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            Vector3 color = Vector3(0.0f, 0.0f, 0.0f); // Initialize pixel color
            float totalWeight = 0.0f;

            for (const auto &point : points)
            {
                float u = x + point.first;
                float v = y + point.second;

                // Generate the ray from the camera
                Ray ray = camera.generateRay(static_cast<float>(u), static_cast<float>(v));
                Intersection closestIntersection = findClosestIntersection(ray, spheres, cylinders, triangles);

                // Set pixel value based on render mode
                if (closestIntersection.hit)
```

Figure 22: Apply random sampling in rendering.



```cpp
                }
                else
                {
                    // Handle cases with no intersection
                    if (renderMode == RenderMode::BINARY)
                    {
                        // Binary shading: set color to red if there's an intersection
                        color = Vector3(0.0f, 0.0f, 0.0f);
                        totalWeight += 1.0f;
                    }
                    else if (renderMode == RenderMode::PHONG)
                    {

                        color += backgroundColor;
                        totalWeight += 1.0f;
                    }
                }
            }

            color /= totalWeight; // Normalize color

            // Store color in the HDR buffer
            hdrColors[y * width + x] = color;
        }
```

Figure 23: Code snippet for calculating colour if pixel sampling is enabled.



```cpp
// Generates evenly distributed points for antialiasing
std::vector<std::pair<float, float>> plot_evenly_distributed_points(int num_samples = 64, float lower_bound = -1.0f, float upper_bound = 1.0f)
{
    int grid_size = static_cast<int>(std::sqrt(num_samples)); // Approximate grid size
    float step = (upper_bound - lower_bound) / grid_size;     // Distance between grid points

    // Generate evenly distributed points with slight jitter
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_real_distribution<float> jitter(-step / 5, step / 5); // Small jitter to avoid perfect alignment

    std::vector<std::pair<float, float>> points;
    for (int i = 0; i < grid_size; ++i)
    {
        for (int j = 0; j < grid_size; ++j)
        {
            float x = lower_bound + i * step + jitter(rng); // Add jitter to x
            float y = lower_bound + j * step + jitter(rng); // Add jitter to y
            points.emplace_back(x, y);                      // Store the jittered point
        }
    }
```

Figure 24: Code snippet for generating points for sampling the pixel.

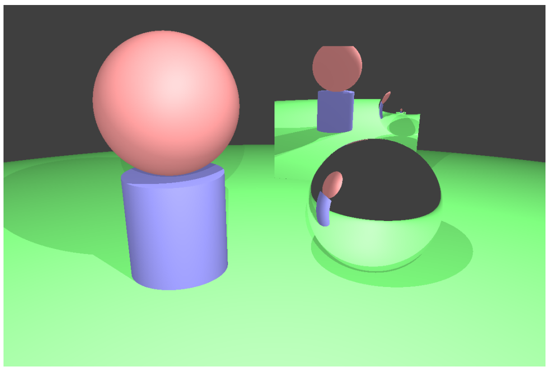


Figure 25: Scene rendered with Blinn-Phong shading without antialiasing.

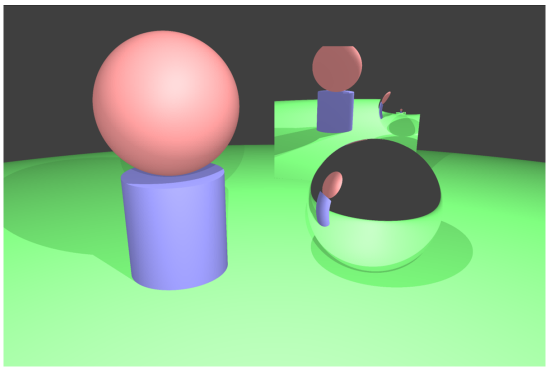


Figure 26: Scene rendered with Blinn-Phong shading with antialiasing.

2. **Lens Sampling for Depth of Field (Aperture)** Lens sampling simulates depth of field (DoF) effects by accounting for the aperture of a camera lens. In ray tracing, this involves generating rays from a finite aperture and focusing them on a specific focal plane, mimicking the behavior of real-world cameras.

   The depth of field is achieved by tracing rays through randomly sampled points on the lens aperture. These rays converge at a focal plane to produce sharp images of objects at a specific distance.

   Steps: 1. Define the focal distance d_f, the distance from the camera to the focal plane. 2. Sample random points $\mathbf{P}_a$ on the lens aperture. 3. Compute the direction of the ray from the sampled aperture point $\mathbf{P}_a$ to a point $\mathbf{P}_f$ on the focal plane.

   The direction of the sampled ray is given by:

   $$\mathbf{D}_a = \frac{(\mathbf{P}_f - \mathbf{P}_a)}{\|\mathbf{P}_f - \mathbf{P}_a\|},$$

   where:

   - $\mathbf{P}_f = \mathbf{O} + d_f \cdot \mathbf{D}$ is the intersection of the original ray with the focal plane.
   - $\mathbf{P}_a$ is a randomly sampled point on the aperture.

   Algorithm for Lens Sampling:

   (a) Generate a random sample on the aperture disk:

   $$x_a = r \cdot \cos\theta, \quad y_a = r \cdot \sin\theta,$$

   where r is a random radius and θ is a random angle within the aperture's radius.

   (b) Calculate the new ray origin as:

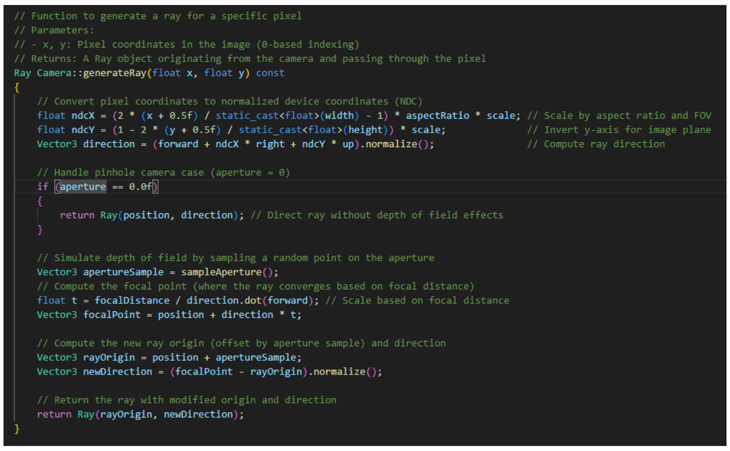
   $$\mathbf{P}_a = \mathbf{P} + x_a \cdot \mathbf{U} + y_a \cdot \mathbf{V},$$

   where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal vectors defining the lens plane.

   (c) Compute the focal point $\mathbf{P}_f$ on the focal plane.

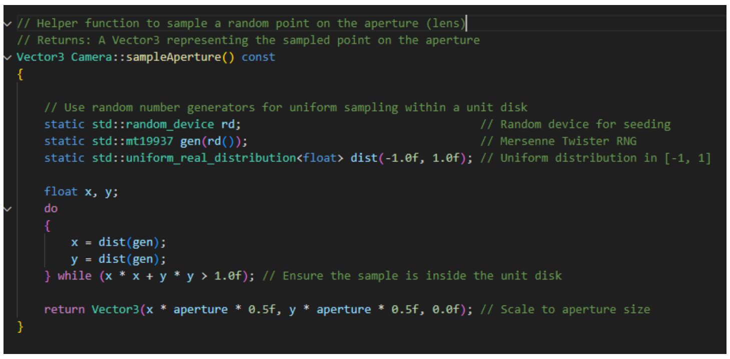
   (d) Trace the new ray from $\mathbf{P}_a$ to $\mathbf{P}_f$.

   Firgure 27 and 28 shows the code snippet of this function, and Figure 29 and 30 show the comparison of the pinhole camera and simulation of a camera with depth of field and aperture.



```cpp
// Function to generate a ray for a specific pixel
// Parameters:
// - x, y: Pixel coordinates in the image (0-based indexing)
// Returns: A Ray object originating from the camera and passing through the pixel
Ray Camera::generateRay(float x, float y) const
{
    // Convert pixel coordinates to normalized device coordinates (NDC)
    float ndcX = (2 * (x + 0.5f) / static_cast<float>(width) - 1) * aspectRatio * scale; // Scale by aspect ratio and FOV
    float ndcY = (1 - 2 * (y + 0.5f) / static_cast<float>(height)) * scale;              // Invert y-axis for image plane
    Vector3 direction = (forward + ndcX * right + ndcY * up).normalize();                // Compute ray direction

    // Handle pinhole camera case (aperture = 0)
    if (aperture == 0.0f)
    {
        return Ray(position, direction); // Direct ray without depth of field effects
    }

    // Simulate depth of field by sampling a random point on the aperture
    Vector3 apertureSample = sampleAperture();
    // Compute the focal point (where the ray converges based on focal distance)
    float t = focalDistance / direction.dot(forward); // Scale based on focal distance
    Vector3 focalPoint = position + direction * t;

    // Compute the new ray origin (offset by aperture sample) and direction
    Vector3 rayOrigin = position + apertureSample;
    Vector3 newDirection = (focalPoint - rayOrigin).normalize();

    // Return the ray with modified origin and direction
    return Ray(rayOrigin, newDirection);
}
```

Figure 27: Code snippet generating ray for lens sampling.



```cpp
// Helper function to sample a random point on the aperture (lens)
// Returns: A Vector3 representing the sampled point on the aperture
Vector3 Camera::sampleAperture() const
{

    // Use random number generators for uniform sampling within a unit disk
    static std::random_device rd;                                  // Random device for seeding
    static std::mt19937 gen(rd());                                 // Mersenne Twister RNG
    static std::uniform_real_distribution<float> dist(-1.0f, 1.0f); // Uniform distribution in [-1, 1]

    float x, y;
    do
    {
        x = dist(gen);
        y = dist(gen);
    } while (x * x + y * y > 1.0f); // Ensure the sample is inside the unit disk

    return Vector3(x * aperture * 0.5f, y * aperture * 0.5f, 0.0f); // Scale to aperture size
}
```

Figure 28: Code snippet for the helper function to sample the aperture.

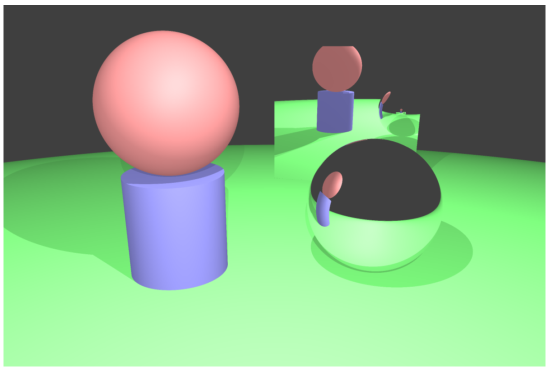


Figure 29: Scene rendered with Blinn-Phong shading with a pinhole camera.

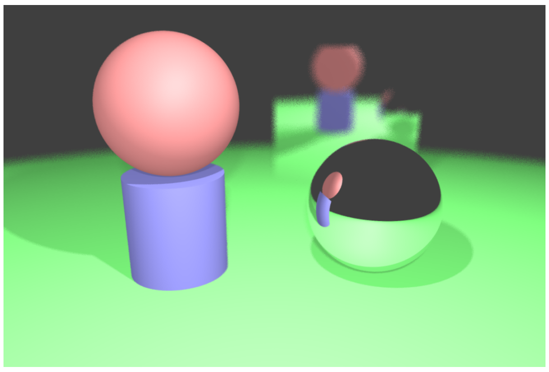
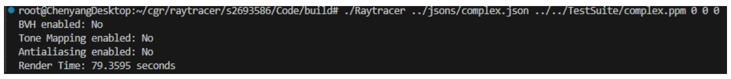


Figure 30: Scene rendered with Blinn-Phong shading with depth of field and aperture applied.

3 Evaluation and Future Works

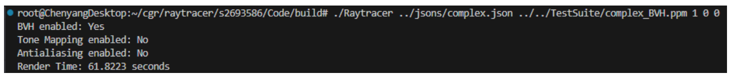
3.1 Performance Analysis

The speed-up of BCH has been tested on a more complex scene. As shown in Figure 31 and 32. We can see that with BVH enabled for this scene, the render time is 61.8 seconds, whereas without BVH, the render time is 79.4 seconds. By applying BVH, we have gained a 22.1% increase in rendering efficiency.



```
root@ChenyangDesktop:~/cgr/raytracer/s2693586/Code/build# ./Raytracer ../jsons/complex.json ../../TestSuite/complex.ppm 0 0 0
BVH enabled: No
Tone Mapping enabled: No
Antialiasing enabled: No
Render Time: 79.3595 seconds
```

Figure 31: Timed BVH disabled runtime.



```
root@ChenyangDesktop:~/cgr/raytracer/s2693586/Code/build# ./Raytracer ../jsons/complex.json ../../TestSuite/complex_BVH.ppm 1 0 0
BVH enabled: Yes
Tone Mapping enabled: No
Antialiasing enabled: No
Render Time: 61.8223 seconds
```

Figure 32: Timed BVH enabled runtime.

By applying BVH to other scenes, we have also observed that the more complex the scene(more objects), the better the BVH can speed up the rendering time.

3.2 Future works

There are a few potential areas that can be improved on top of this project:

1. **Acceleration**: Apart from BVH, we can introduce GPU-based optimisations as GPUs are extremely efficient on tasks like this and will improve rendering speed.
2. **Sampling**: Other methods like stratified and importance sampling might improve quality.
3. **Lighting**: Volumetric light can be implemented for soft shadows, which will increase the realism of the rendered image.

4 Conclusion

The development of this raytracer project has successfully demonstrated the implementation of key rendering techniques, including Blinn-Phong shading, reflection, refraction, and tone mapping, alongside optimisations such as Bounding Volume Hierarchies (BVH) and pixel sampling for antialiasing. These features, combined with advanced techniques like depth of field and aperture simulation, have enabled the creation of visually appealing and photorealistic scenes.

Throughout the project, mathematical rigour and algorithmic efficiency have been prioritised to ensure accuracy in ray-object intersections and lighting computations. Incorporating BVH, in particular, has shown significant improvements in rendering performance, as evidenced by the 22.1% speed-up in complex scenes. Pixel sampling and lens sampling have further enhanced image quality, reducing aliasing artefacts and providing realistic camera effects.

Despite the accomplishments, there are areas for further exploration. The project can benefit from additional acceleration techniques, such as GPU-based ray tracing, to reduce rendering times. Advanced sampling methods and volumetric lighting could also enhance realism and flexibility in lighting effects. These improvements can extend the versatility of the raytracer and enable its application to more complex and dynamic scenes.

This project has provided a robust foundation for understanding and implementing core and advanced raytracing techniques. While limitations exist, the outcomes achieved demonstrate the potential for further refinement and expansion, paving the way for future more sophisticated and efficient rendering systems.

Appendix: All images rendered

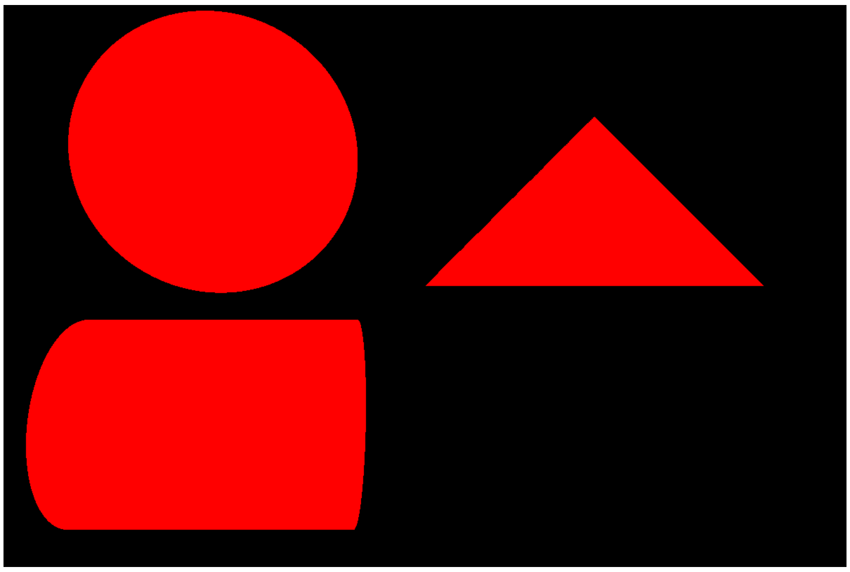


Figure 33: binary_primitives.png

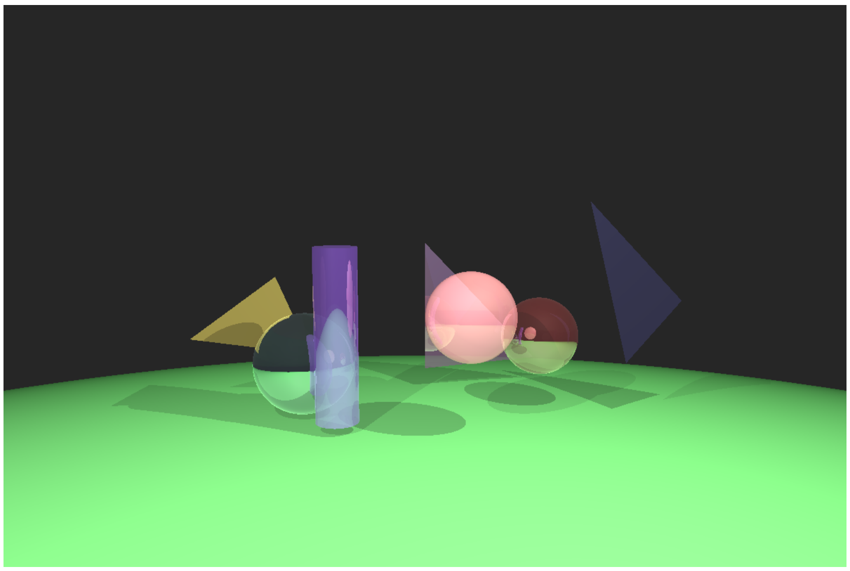


Figure 34: complex.png

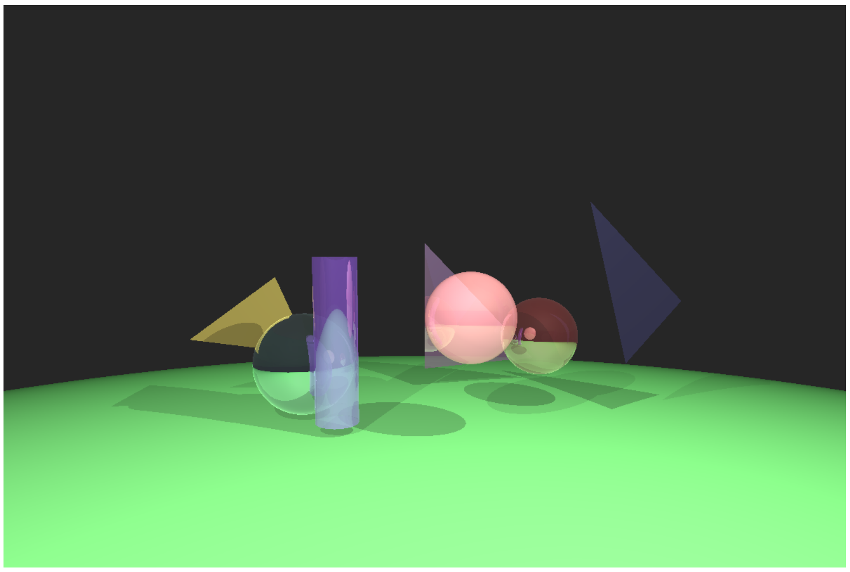


Figure 35: complex_BVH.png

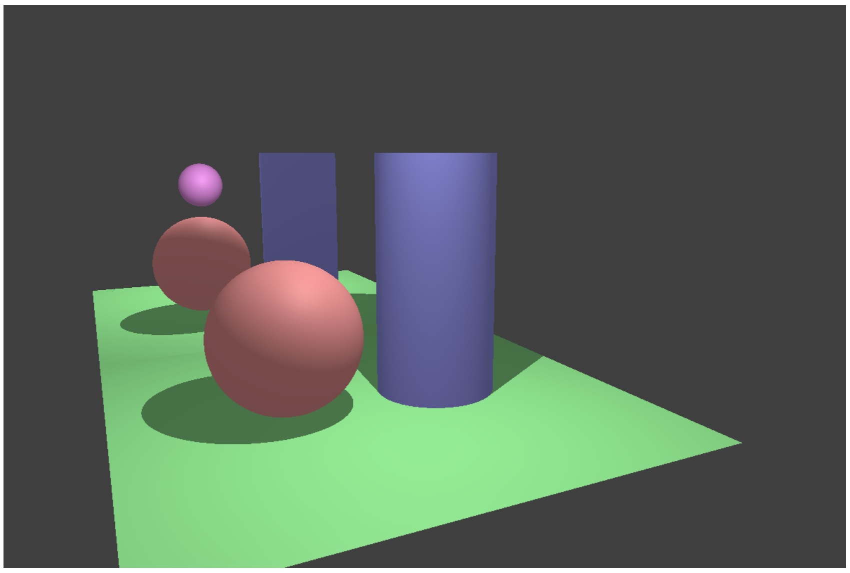


Figure 36: mirror_image.png

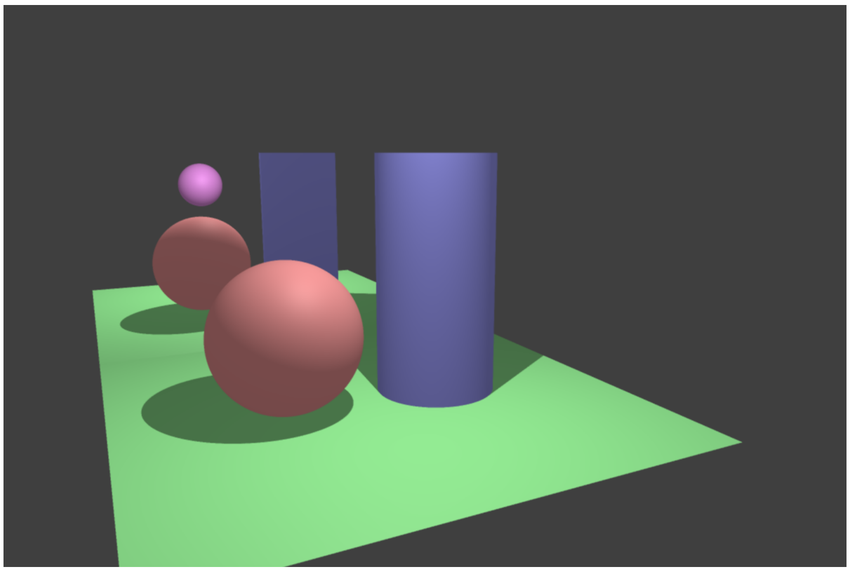


Figure 37: mirror_image_antialiasing.png

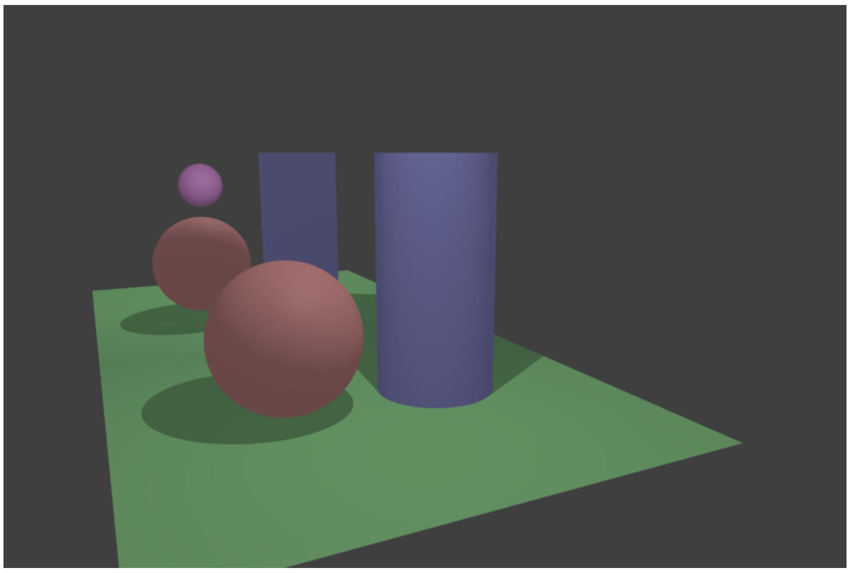


Figure 38: mirror_image_antialiasing_toneMap.png

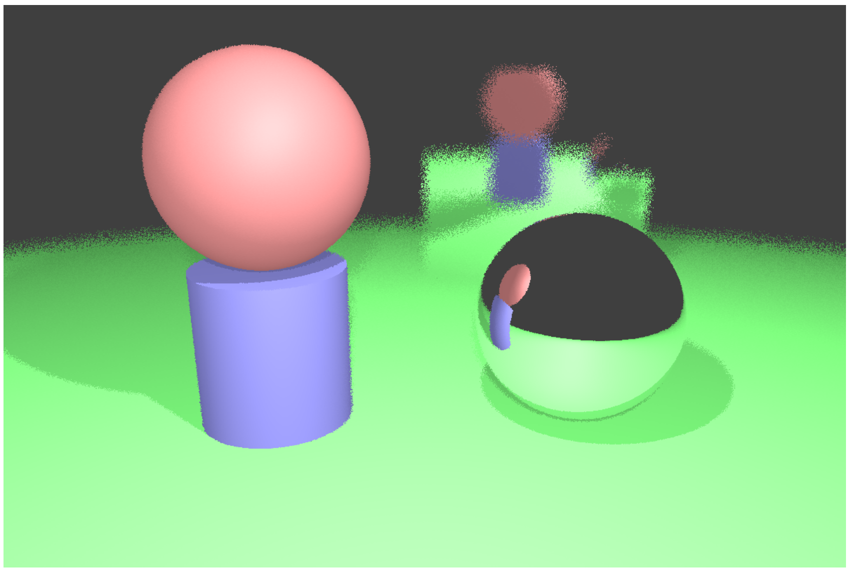


Figure 39: scene_aperture.png

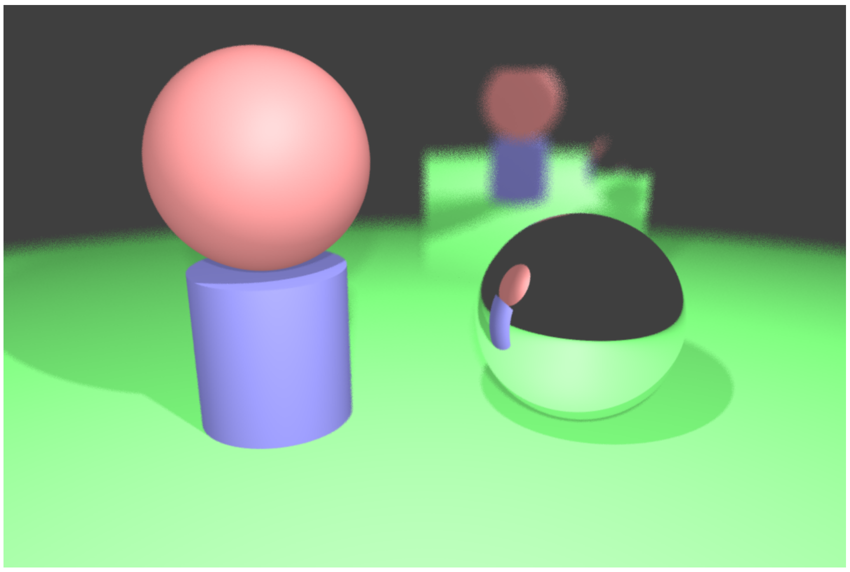


Figure 40: scene_aperture_antialiasing.png

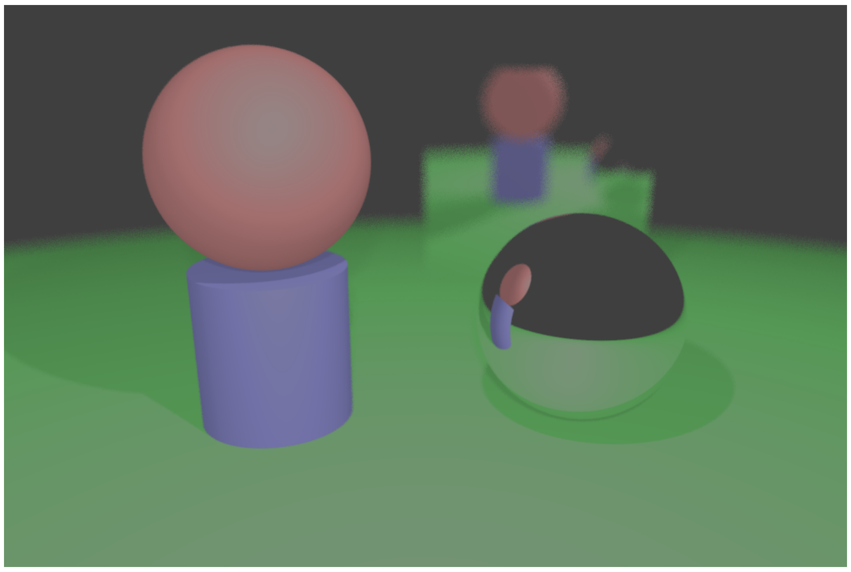


Figure 41: scene_aperture_antialiasing_toneMap.png

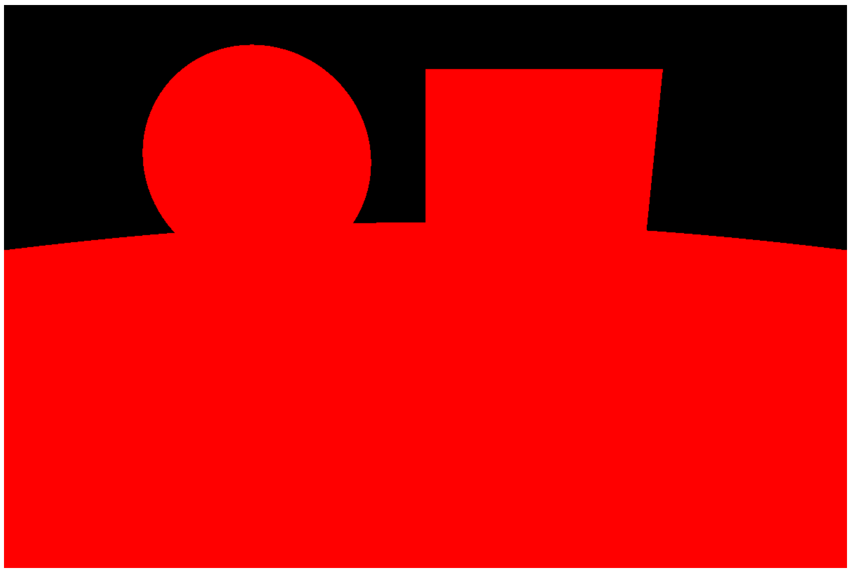


Figure 42: scene_binary.png

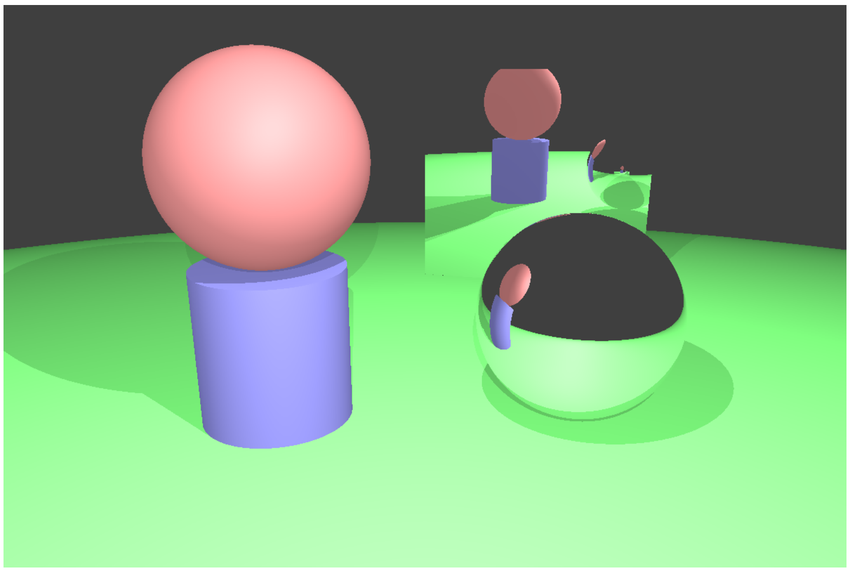


Figure 43: scene_phong.png

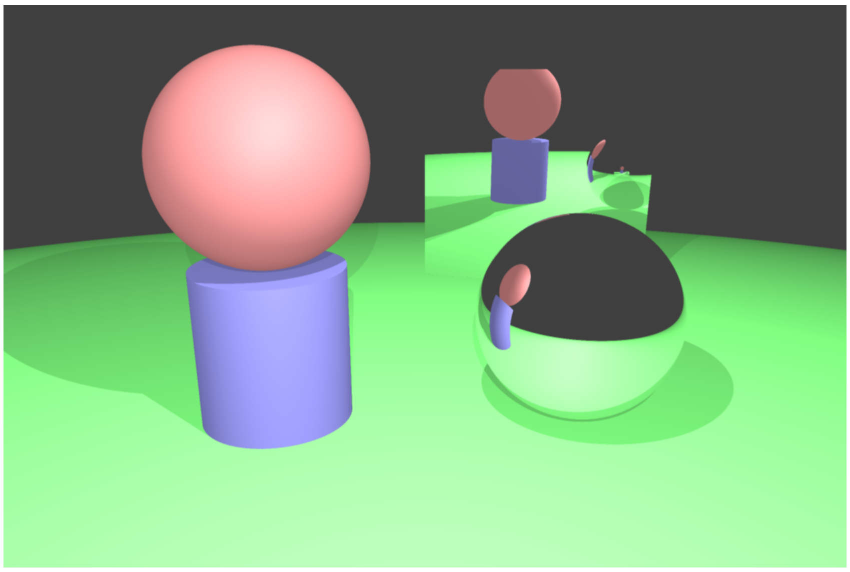


Figure 44: scene_phong_antialiasing.png

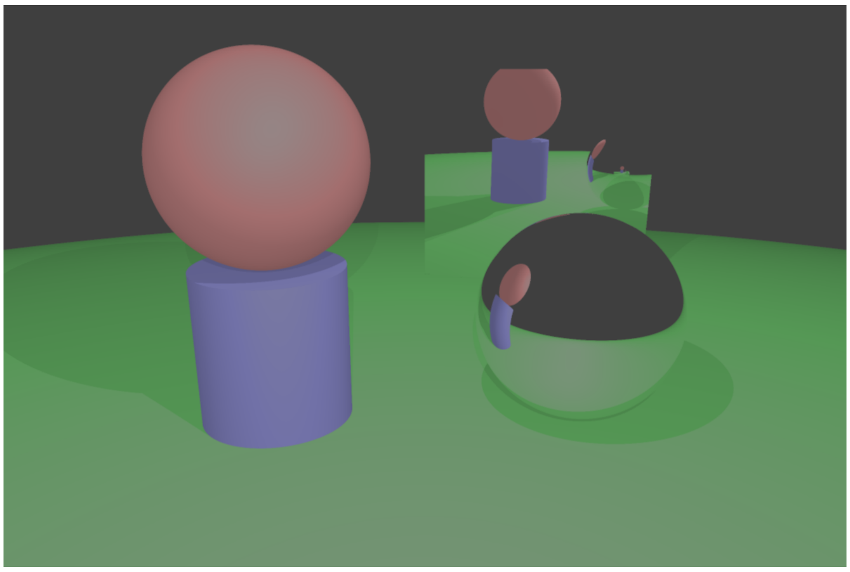


Figure 45: scene_phong_antialiasing_toneMap.png

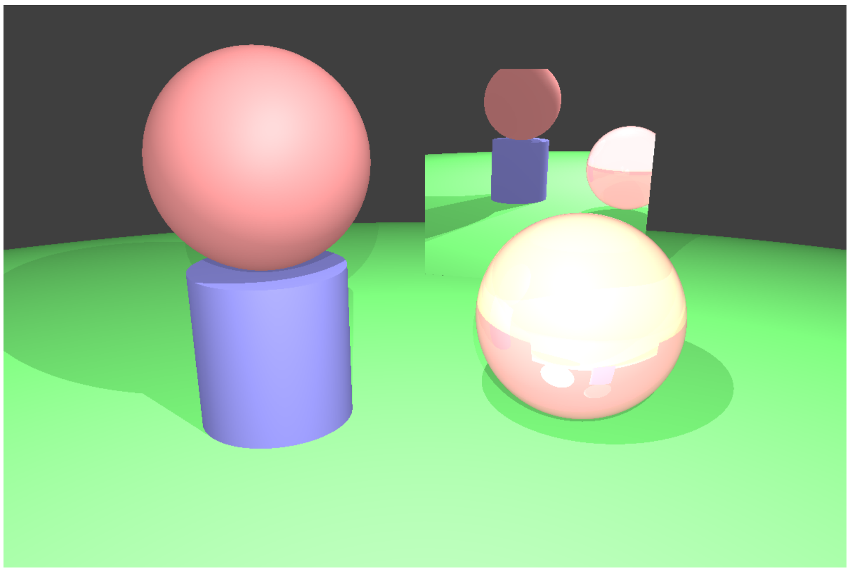


Figure 46: scene_phong_refractive.png

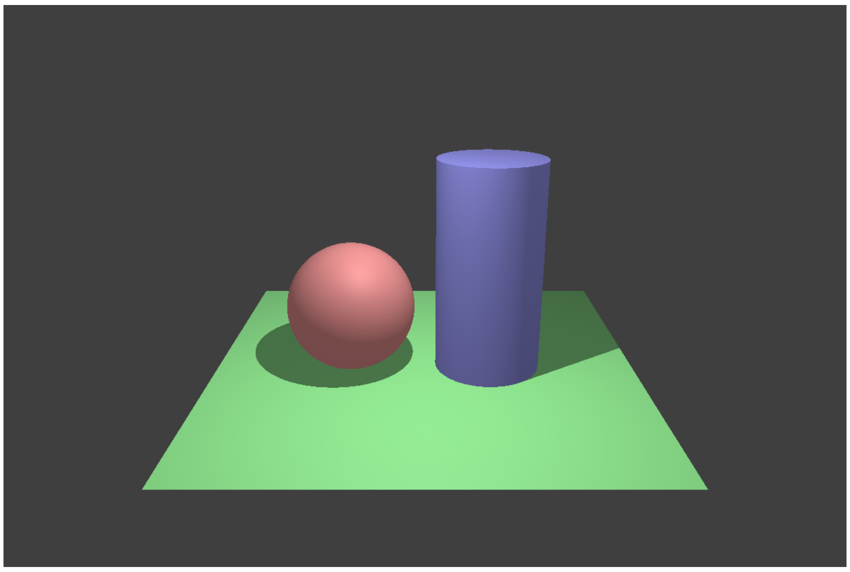


Figure 47: simple_phong.png

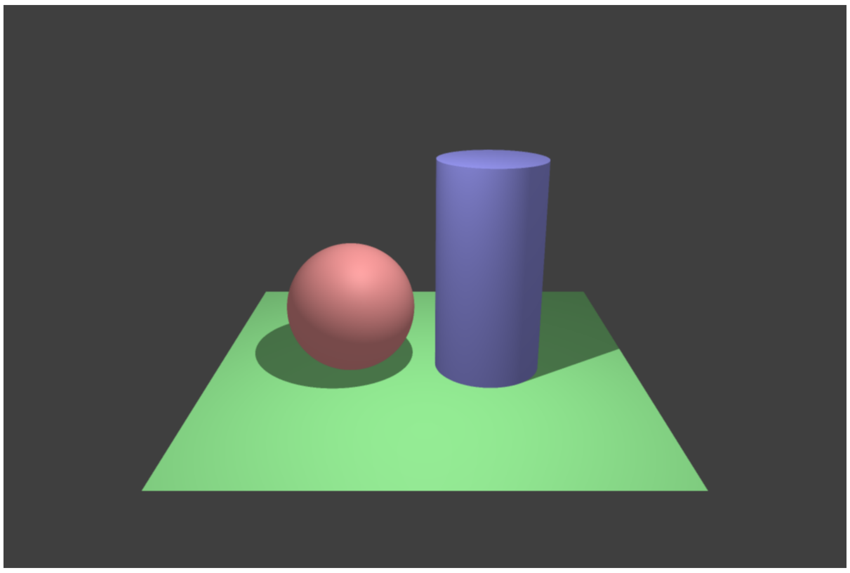


Figure 48: simple_phong_antialiasing.png

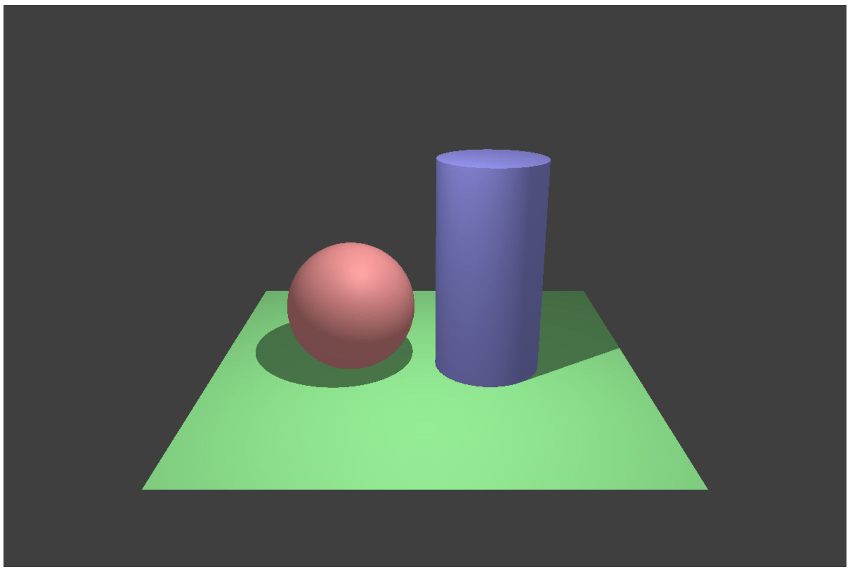


Figure 49: simple_phong.png