École Polytechnique, Bachelor of Science, CSC3S006EP (Computer Graphics)

Project Report: Ray Tracer

Luca-Andrei Fechete[a,1]

[a] *École Polytechnique*

Course Professor: Nicolas Bonneel

Abstract—This report details the C++ implementation, logic and results of a ray tracer, completed as the first project for the Computer Graphics course at École Polytechnique, Paris.

1. Introduction

The features implemented as part of this project are the following:

- Diffuse, mirror and transparent surfaces including Fresnel law.
- Direct and indirect lighting and shadows for point light sources.
- Direct and indirect lighting and shadows for sphere light sources.
- Anti-aliasing and depth of field.
- Ray-mesh intersection with the boundary box (BB) and boundary volume hierarchy (BVH) optimizations.

The following sections will explore more in depth the project's general setup along with each feature, its implementation, and the resulting picture obtained.

2. General Structure

This project's implementation can be found in the source directory which includes the source files and the headers of the classes used. The linking and the external package management was done using CMake with a template structure obtained from GitHub [1].

- `Vector`: This class was used for the computations involving three-dimensional vectors as well as for defining the color (albedo) of the rendered objects. It is defined using three coordinates and includes methods related to basic vector operations.
- `Ray`: This class was used to define the imaginary rays launched from the camera. It is defined by an origin and a unit direction.
- `Intersection`: This structure is used to define the intersections between rays and objects of type `Geometry`. An intersection is defined through the point of intersection, the normal to the surface from this point, the color of the intersected object, the distance from the ray origin to the object and an unique identifier of the intersected object.
- `Geometry`: This class represents the generic type of physical object that can be rendered. Every defined object inherits the color attribute, the material flags (mirror, transparency, refraction index, or light source), and a method that checks intersection with a ray.
- `Sphere`: This class represents a physical spherical object that inherits its generic attributes from the `Geometry` class. The particular attributes related to a three-dimensional sphere are defined as a center vector, a radius and a flag that decides whether or not to invert the normals (useful to simulate empty spheres).
- `TriangleMesh`: This class represents a collection of triangles that also inherits its generic attributes from the `Geometry` class and is used to simulate real world objects that are too difficult to represent using standard geometrical approaches. The attributes of this class are the indices of the triangles, the coordinates of the vertices, the normal vectors for each triangle and color related attributes to be used for textures. Because these collections of triangles are usually very large, rendering is really slow and so there is the need for optimization in terms of ray-object intersection. For this purpose, this class also includes two objects the types `BoundingBox` and `BVHTree` respectively.
- `Scene`: This class represents the collection of objects that constitute the rendered scene. The scene's particular attributes are a list of pointers to the `Geometry` data type, a point light source, a spherical light source and the refraction index of chemical element predominant in the scene's environment (air in our case).
- `Camera`: This class represents the imaginary camera used for capturing the rendered picture. Its particular attributes are its center, the visual angle which covers the image, the focal distance and aperture used for the depth of field and the spread used for the anti-aliasing. This class contains the method used for computing the ray corresponding to a certain pixel of the image.
- `Renderer`: This class represents the collection of utilities for building and rendering the final image.

3. Surfaces

This section details the implementation and mathematical particularities of the different types of surfaces that can be rendered with the ray tracer.

3.1. Diffuse Surfaces

Rendering diffuse surfaces (Fig. 2a, Fig. 2b) requires an initial setup defining the scene, the camera viewpoint, and the logic for calculating intersections between rays and scene objects. The core rendering process then involves a loop that, for each pixel, casts multiple rays, computes their respective intersections (`Intersection`) with objects, and averages the resulting colors to determine the final pixel value.

3.2. Mirror Surfaces

Rendering mirror surfaces (Fig. 2c) motivates us to add recursion up to a certain preset depth to the function that gets the color of the intersection between a ray and an object. Intuitively, this recursion represents the number of times rays reflect in the scene.

Formally, the reflected ray's center is the intersection point and its unit direction is:

$$\omega_r = \frac{\omega_i - 2\langle \omega_i, N\rangle N}{\|\omega_i - 2\langle \omega_i, N\rangle N\|},$$

where N is the normal to the surface.

In terms of implementation particularities, there was the need to offset the center of the reflected ray by an epsilon in the direction of the normal due to numerical precision issues.

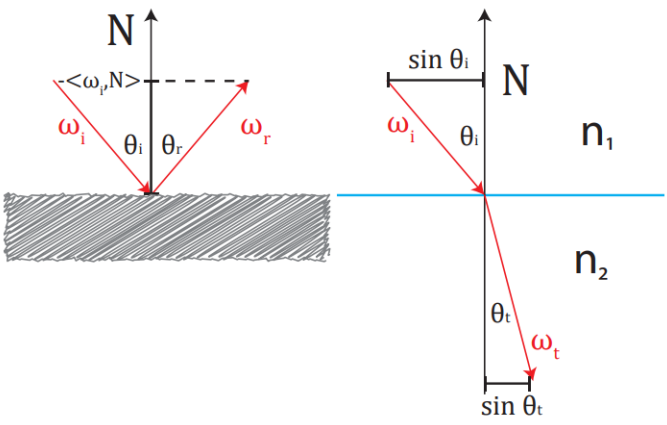
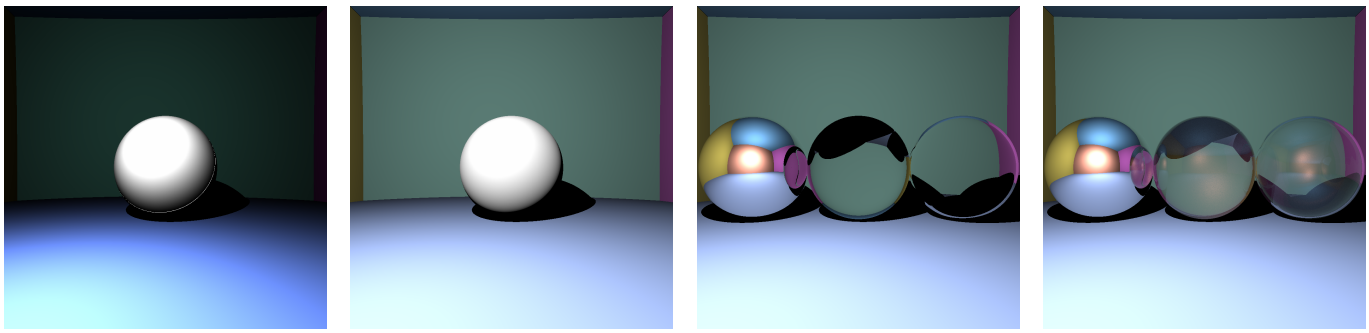


Figure 1. The reflection and refraction phenomena explained visually.



(a) Diffuse sphere without gamma correction.

(b) Diffuse sphere with gamma correction.

(c) Mirror, transparent and empty spheres without Fresnel law.

(d) Mirror, transparent and empty spheres with Fresnel law.

Figure 2. First images generated using the ray tracer. The configuration used is same as the one used in the lecture notes. Rendering these images took under one second per image.

3.3. Transparent Surfaces

Very similar to reflection in terms of implementation, during refraction, rays go through the intersected object and get the color of the first diffuse object they intersect after bouncing through the scene (Fig. 2c).

Formally, the refracted ray's center point is defined as the intersection point between the ray and the object and the refracted ray's non-normalized direction can be defined as follows:

$$\omega_t = \frac{\eta_1}{\eta_2}(\omega_i - \langle \omega_i, N\rangle N) - N\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \langle \omega_i, N\rangle^2)},$$

where η_1, η_2 are the indices of refraction of the two media, ω_i is the initial ray's unit direction and N is the normal to the surface.

While implementation shares aspects with reflection, refraction through a transparent object introduces complexities because rays intersect it twice. Consequently, offsetting intersection points must be managed differently, and calculating the refracted ray's unit direction necessitates interchanging the indices of refraction and adjusting the normal vector's sign.

3.4. Fresnel Law

In realistic scenarios, total refraction (or total reflection, outside of specific conditions like the critical angle) does not typically occur. Instead, light rays are partially reflected and partially refracted at an interface. To simulate this accurately, we implement a hybrid system that uses a random number to probabilistically determine whether a given ray follows the reflection or the refraction path (Fig. 2d). This decision is based on the following formula:

$$R = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 + \left(1 - \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2\right)(1 - |\langle N, \omega_i\rangle|)^5.$$

The sampled random number determines the ray type: if it is less than R, the ray is reflected; otherwise, it is refracted.

4. Light Sources

Effective scene rendering necessitates the inclusion of light sources, as their absence would result in a dark scene. Furthermore, the presence of lighting is required for shadows to appear. In this section, we will detail direct point light sources, indirect point light sources, and spherical light sources.

4.1. Point Light Sources

For these types of light sources, we assumed a simple material model, the Lambertian model. Formally, the color of a given pixel is:

$$L = \frac{I}{4\pi d^2}\frac{\rho}{\pi} V_P(S)\langle N, \omega_i\rangle.$$

This formula calculates the intensity of light reflected (L) from a diffuse surface point (P). It depends on the light source's intensity (I), its distance ($d = \|S - P\|$), the surface's albedo (ρ), the surface normal (N), the unit vector towards the light source ($\omega_i = \frac{S-P}{\|S-P\|}$), and whether the light source is visible from (P) ($V_P(S)$). The dot product ($\langle N, \omega_i\rangle$) accounts for the angle of incoming light relative to the surface.

4.2. Indirect Lighting

In a path tracer, implementing indirect lighting for diffuse surfaces involves using Monte Carlo integration to estimate the rendering equation, which for diffuse surfaces simplifies to:

$$L_o(x, \omega_o) = \frac{\rho}{\pi}\int_\Omega L_i(x, \omega_i)\langle \omega_i, N\rangle d\omega_i$$

This is achieved by randomly sampling outgoing light directions ω_i from a point x on the surface, typically using importance sampling biased towards directions aligned with the surface normal N. A common way to generate such samples for N uses random numbers $r_1, r_2 \sim U(0, 1)$ and the formulas:

$$\omega_i = \begin{pmatrix} \cos(2\pi r_1)\sqrt{1 - r_2} \\ \sin(2\pi r_1)\sqrt{1 - r_2} \\ \sqrt{r_2} \end{pmatrix}$$

These samples have a probability density function (PDF) proportional to $\langle \omega_i, N\rangle$. A ray is then traced recursively in the sampled direction to gather incoming light L_i. The Monte Carlo estimator for the integral involves dividing the integrand by the PDF, which in this case simplifies the calculation (as noted in the text, cosine terms and π factors cancel out). The total illumination is the sum of the direct lighting contribution and this recursively sampled indirect contribution. While this random process introduces noise, averaging the results from multiple sampled rays per pixel reduces the variance and converges to the correct result (Fig. 2a).

4.3. Spherical Light Sources

To efficiently implement indirect lighting from spherical light sources, we stochastically sample points directly on the surface of the sphere (x'), rather than just directions towards it. This sampling is biased towards the hemisphere of the light visible from the shading point (x) and regions where the light's surface normal faces the shading point (higher contribution). This is achieved by using a cosine-weighted sampling function, directed towards the visible hemisphere defined by the vector $D = \frac{x-C}{\|x-C\|}$. A sampled direction V is obtained, and the point on the sphere is $x' = RV + C$. The probability density of sampling this point x' is $p(x') = \frac{\langle V, D\rangle}{\pi R^2}$ (Fig. 5c).

5. Anti-Aliasing and Depth of Field

This section details the particularities of the ray computation in the `Camera` class.

Anti-Aliasing. In order to implement anti-aliasing, we will use the `boxMuller` function defined in the lecture notes.

```
void Camera::boxMuller(double stdev , double &x , double
    ↪  &y) {
    double r1 = get_random_number();
    double r2 = get_random_number();
    x = sqrt((-2) * log (r1)) * cos (2 * M_PI * r2) *
    ↪ stdev;
    y = sqrt((-2) * log (r1)) * sin (2 * M_PI * r2) *
    ↪ stdev;
}

Ray Camera::get_ray(const int& i, const int& j) {
    // ...
    double dx, dy;
    boxMuller(stdev, dx, dy);
    dx *= spread;
    dy *= spread;

    // Computation of the location of pixel (i, j)
    double pixel_x = vec_center.get_x() + (j + dx) + 0.5
    ↪  - (W / 2.0);
    double pixel_y = vec_center.get_y() + (H - (i + dy) -
    ↪  1) + 0.5 - (H / 2.0);
    double pixel_z = vec_center.get_z() - (W / (2 * std::
    ↪ tan(alpha_rad / 2)));
    Vector vec_pixel_pos = Vector(pixel_x, pixel_y,
    ↪ pixel_z);
    // ...
}
```

Code 1. The sampled values with the use of the `boxMuller` function are multiplied by the spread value and added to the index values of the pixels.

. The result of this technique is an image with smoother corners and overall better quality which can be seen in Fig. 2b with the comparison present in Fig. 3.

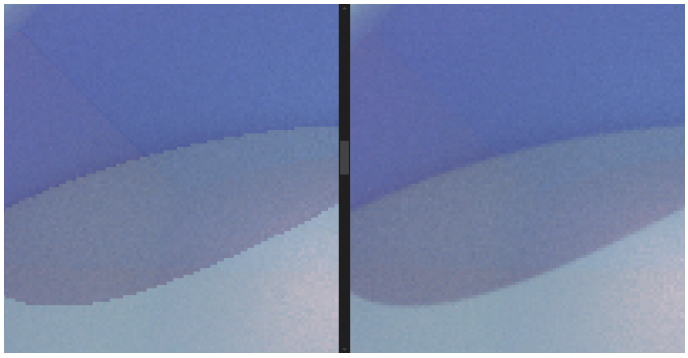


Figure 3. A portion of an image without anti-aliasing on the left and a portion of the same image with anti-aliasing on the right.

Depth of Field. In order to implement depth of field, i.e. blurring effects caused by the camera not focusing, we must first sample random points of the the circle of radius given by the aperture and then launch the ray from the sampled point instead of from the camera center. In terms of implementation, we do this in the `Camera` function as seen below.

```
Ray Camera::get_ray(const int& i, const int& j) {
    // ...

    if (aperture <= 1e-8) {
        return Ray(vec_center, vec_unit_direction);
    }

    double t_focus = focal_dist / std::fabs(
    ↪ vec_unit_direction.get_z());
    Vector vec_focus_point = vec_center + (
    ↪ vec_unit_direction * t_focus);

    double r     = std::sqrt(get_random_number());
    double theta = 2.0 * M_PI * get_random_number();
    double lens_r = aperture * r;

    double lens_x = lens_r * std::cos(theta);
    double lens_y = lens_r * std::sin(theta);

    Vector vec_dof_origin = Vector(
        vec_center.get_x() + lens_x,
        vec_center.get_y() + lens_y,
        vec_center.get_z()
    );

    Vector vec_dof_dir = vec_focus_point -
    ↪ vec_dof_origin;
    vec_dof_dir.normalize();

    return Ray(vec_dof_origin, vec_dof_dir);
}
```

Code 2. The sampled values are used to compute the new unit direction and the new center of the returned ray.

The results of the depth of field feature can be seen in Fig. 5d.

6. BVH and BB for Ray-Mesh Intersection

In the last part of the project, I implemented the ray-mesh intersection and added the bounding volume hierarchy (BVH) or the bounding box (BB) optimizations.

The ray-triangle intersection test determines if a ray $R(t) = O + tu$ intersects a triangle defined by vertices A, B, and C. An intersection point P exists if it lies on the ray (for $t > 0$) and within the triangle. The point P is within or on the boundary of the triangle if it can be expressed as a convex combination using barycentric coordinates α, β, γ: $P = \alpha A + \beta B + \gamma C$, where $\alpha, \beta, \gamma \in [0, 1]$ and $\alpha + \beta + \gamma = 1$.

While α, β, γ geometrically represent area ratios (e.g., $\alpha = \frac{\text{area}(PBC)}{\text{area}(ABC)}$), it is often more practical for ray intersection to use a 2-parameter representation. By substituting $\alpha = 1 - \beta - \gamma$, we can re-parameterize P relative to vertex A using edge vectors $e_1 = B - A$ and $e_2 = C - A$ as $P = A + \beta e_1 + \gamma e_2$. Equating the ray equation with this form, $O + tu = A + \beta e_1 + \gamma e_2$, and rearranging yields a linear equation involving the unknowns β, γ, and t:

$$\beta e_1 + \gamma e_2 - tu = O - A$$

Solving this linear system for the parameters β, γ, and t using vector algebra yields the following formulas:

$$\beta = \frac{\langle O - A, e2 \times -u \rangle}{\langle e1, e2 \times -u \rangle} = \frac{\langle e2, (A - O) \times u \rangle}{\langle u, N \rangle}$$

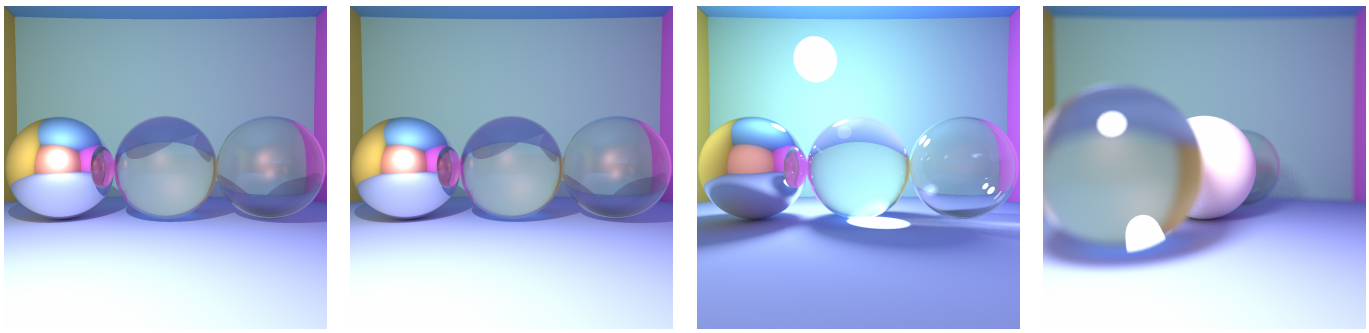
$$\gamma = \frac{\langle e1, (O - A) \times -u \rangle}{\langle e1, e2 \times -u \rangle} = -\frac{\langle e1, (A - O) \times u \rangle}{\langle u, N \rangle}$$

$$\alpha = 1 - \beta - \gamma$$

$$t = \frac{\langle e1, e2 \times (O - A) \rangle}{\langle e1, e2 \times -u \rangle} = \frac{\langle A - O, N \rangle}{\langle u, N \rangle}$$

For a valid intersection point P within the triangle and along the ray, the calculated parameters must satisfy $\beta \geq 0$, $\gamma \geq 0$, $\beta + \gamma \leq 1$, and $t > 0$.

Using the BB optimization, I managed to obtain a runtime which was 6 times smaller than the default one. By using the BVH optimization, I managed to obtain a runtime which is approximately 100 times smaller than the default runtime. The implementation of these optimizations are performed on the basis of the pseudocode implementations from the lecture notes. The result can be viewed in Fig. 4.



(a) Mirror, transparent and empty spheres with indirect lighting and without anti-aliasing.

(b) Mirror, transparent and empty spheres with indirect lighting and with anti-aliasing.

(c) Mirror, transparent and empty spheres with spherical light source and without anti-aliasing.

(d) Mirror, transparent and empty spheres with spherical light source and with depth of field.

Figure 5. These are the images relevant to the different types of lighting as well as to depth of field (Fig. 5d) and anti-aliasing (Fig. 5b) . For the first two images, 1000 rays per pixel were used with an intensity depth of 8, and a spread of 0.5 for the second one. For the third image, 5000 rays per pixel were used with an intensity depth of 8 while for the fourth one 3000 rays pe pixel were used with and intensity depth of 8, spread of 0.5, aperture radius of 1.5 and focal distance of 55. The first two images were rendered in approximately 10 minutes (for a dimension of 1024 by 1024 pixels) and approximately 1 minute for a dimension of (512 by 512 pixels). The third image took approximately 27 minutes for a dimension of 1024 by 1024 pixels while the fourth image took approximately 17 minutes for a dimension of 1024 by 1024 pixels.

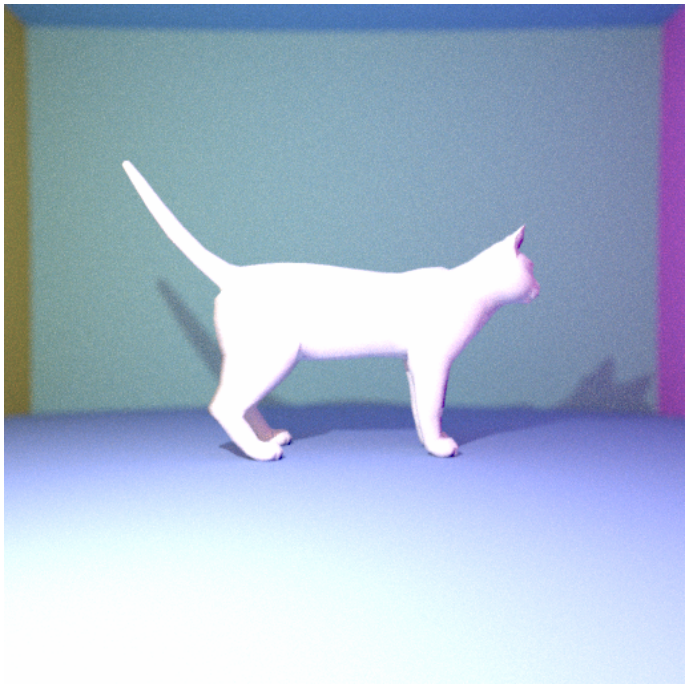


Figure 4. The rendered image for the ray-mesh intersection functionality with 100 rays per pixel which took about 7 seconds.

7. Reproductibility Features

In order to reproduce the majority of the results, there are a number of features in the `main.cc` file. First of all, in order to reproduce the image with the three spheres, the `SETTING` definition at the beginning of the file should be set to 1. In order to reproduce the image with the spherical light source, the flag `light_source_sphere` should be set to true. Also the optimization strategy for the ray-mesh intersection can be chosen between BVH, BB or no optimization.

All of the features are implemented using the indications and pseudocode from the lecture notes.

Difficulties. My work on the project began with familiarizing myself with the assignment requirements and the fundamental principles of ray tracing, including camera operation and image rendering. After implementing reflection, significant effort was directed towards refraction, where the primary difficulty involved correctly managing sign changes as rays traversed objects. The addition of indirect lighting introduced performance bottlenecks, necessitating the parallelization of the program to achieve acceptable runtime. A separate challenge arose with sampling for anti-aliasing and depth of field; the issue stemmed from computing multiple colors per ray instead of multiple rays per pixel, which was resolved by correctly seeding the random number generator for each thread. The final major difficulties occurred during the implementation and optimization of triangle meshes, with bugs typically traced back to either incorrect formulas or the redundant construction of the BVH tree.

References

[1] franneck94, *Cppprojecttemplate*. [Online]. Available: https://github.com/franneck94/CppProjectTemplate.