École Polytechnique

CSE306 - Ray Tracer - Final Report

Author:

Liam Loughman

Academic year 2023/2024

1 Introduction

In this project, we developed a ray tracer to create realistic images by simulating light interacting with objects. Our ray tracer supports spheres, which can cast shadows, reflect like mirrors, or appear fully or hollowly transparent. We also implemented other features such as indirect lighting, anti-aliasing, and depth of field to enhance image quality. Our ray tracer can load and process triangle meshes, using bounding boxes and bounding volume hierarchies (BVH) to increase rendering speed. We ran our ray tracer on a lab machine through SSH. This machine has an Intel® Xeon® W-1270P CPU with 8 cores at 3.80GHz. This report discusses the design of our ray tracer, the technical methods used, and how we optimized performance for real-time rendering.

All rendered images have a resolution of 512×512 pixels. The camera is positioned at $(0, 0, 55)$ with a $60°$ field of view. The scene includes six large spheres designed to simulate flat walls, ceiling and floor. A light source is located at $(-10, 20, 40)$ with an intensity of 2×10^{10}. Gamma correction is applied with a factor of 2.2:

```
Sphere* ceiling = new Sphere(Vector(0, 1000, 0), 940, Vector(1, 0, 0), false);
scene.addGeometry(ceiling);
Sphere* floor = new Sphere(Vector(0, -1000, 0), 990, Vector(0, 0, 1), false);
scene.addGeometry(floor);
Sphere* front = new Sphere(Vector(0, 0, -1000), 940, Vector(0, 1, 0), false);
scene.addGeometry(front);
Sphere* back = new Sphere(Vector(0, 0, 1000), 940, Vector(1, 0, 1), false);
scene.addGeometry(back);
Sphere* left = new Sphere(Vector(1000, 0, 0), 940, Vector(1, 1, 0), false);
scene.addGeometry(left);
Sphere* right = new Sphere(Vector(-1000, 0, 0), 940, Vector(0, 1, 1), false);
scene.addGeometry(right);

Vector light_position(-10, 20, 40);
double light_intensity = 2e10;
double gamma = 2.2;
```

2 Initial Ray Tracer

The initial version of our ray tracer is relatively basic; the main loop directly calculates the intersections, colors, and shadows of the objects in the scene:

```
Intersection intersection = scene.intersect(ray);
if (intersection.intersected) {
    Vector P = intersection.P;
    Vector N = intersection.N;
    Vector offsetP = P + N * 1e-4;  // Offset point for shadow ray origin
    Vector L = light_position - P;
    double d = L.norm();
    L = L.normalize();
    double dotLN = std::max(dot(L, N), 0.0);
    Ray shadow_ray(offsetP, L);  // Shadow ray from slightly above the surface
    // to the light

    Intersection shadow_intersection = scene.intersect(shadow_ray);
    double shadow = 1.0;
    if (shadow_intersection.intersected && shadow_intersection.t < d) {
        shadow = 0.0; // Light is blocked
    }

    double intensity = light_intensity / (4 * M_PI * d * d);
    Vector color = shadow * intensity * dotLN * intersection.albedo / M_PI;
    color[0] = pow(color[0], 1.0 / gamma); // Gamma correction
    color[1] = pow(color[1], 1.0 / gamma); // Gamma correction
    color[2] = pow(color[2], 1.0 / gamma); // Gamma correction
```

In our scene, we render a white sphere and achieve the following results:

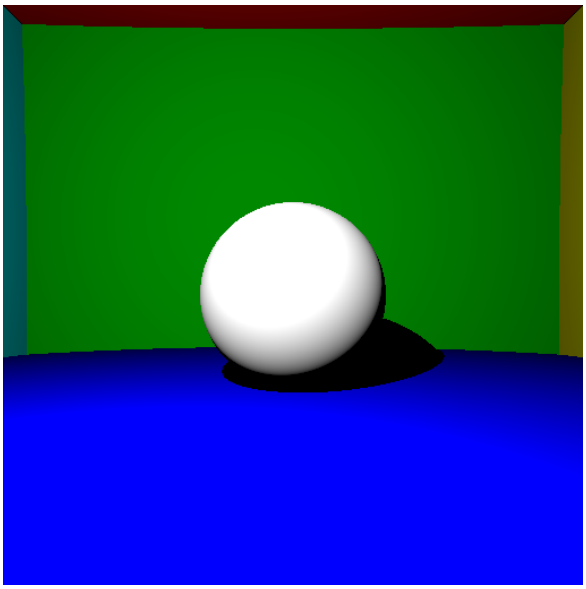

Figure 1: Rendering time: 0.436 seconds without parallelization.

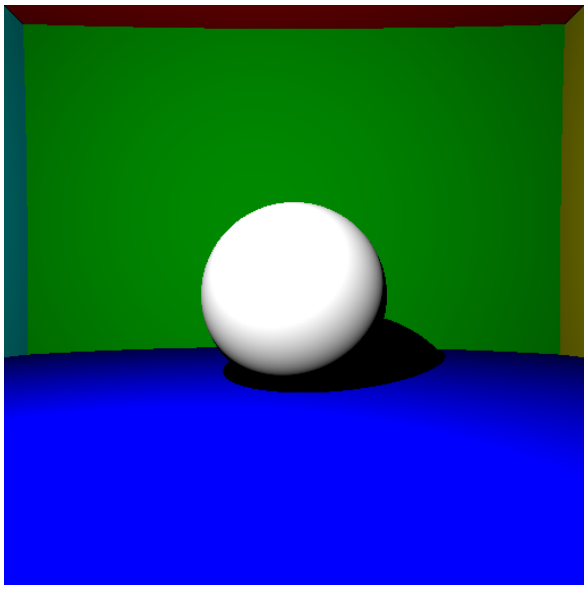

Figure 2: Rendering time: 0.04 seconds with parallelization.

3 Mirror, Full and Hollow transparent surfaces

In our ray tracer, we implemented the getColor function to compute the color of objects based on the type of their surface. This function handles different surface interactions including mirror (reflective), refractive, or normal colored surfaces. Below is a concise and early version of the getColor function, focused on the calculations for mirror and refractive surfaces, with other details omitted for clarity:

```
Vector getColor(const Ray& ray, int ray_depth) {
    if (ray_depth < 0) return Vector(0., 0., 0.);

    Intersection intersection = intersect(ray);
    if (!intersection.intersected) return Vector(0., 0., 0.);

    const double epsilon = 1e-4;
    Sphere& sphere = *spheres[intersection.index];
    Vector P = ray.O + ray.u * intersection.t;
    Vector N = (P - sphere.C).normalize();

    if (sphere.mirror) {
        Vector reflection_direction = ray.u - 2 * dot(ray.u, N) * N;
        Ray reflected_ray(P + epsilon * N, reflection_direction);
        return getColor(reflected_ray, ray_depth - 1);
    } else if (sphere.refractive_index != 1.) {
        double n1, n2, x = ((double) rand() / (RAND_MAX));
        if (dot(ray.u, N) > 0) {
            N = -N;
            n1 = intersection.refractive_index;
            n2 = 1.;
        } else {
            n1 = 1.;
            n2 = intersection.refractive_index;
        }
        double k0 = pow((n1 - n2), 2.) / pow((n1 + n2), 2.);
        if (1. - pow((n1/n2), 2.) * (1 - pow(dot(ray.u, N), 2.)) > 0) {
            Vector w_t = (n1/n2) * (ray.u - dot(ray.u, N) * N);
```

```
            Vector w_n = -N * sqrt(1 - pow((n1/n2), 2.) * (1 - pow(dot(ray.u, N), 2.)));
            Vector w = w_t + w_n;
            if (x < k0 + (1 - k0) * pow(1 - abs(dot(N, w)), 5.)) {
                Ray reflected_ray(P, ray.u - 2 * dot(N, ray.u) * N);
                return getColor(reflected_ray, ray_depth - 1);
            } else {
                Ray refracted_ray(P, w);
                return getColor(refracted_ray, ray_depth - 1);
            }
        } else {
            Ray internal_reflected_ray = Ray(P, ray.u - 2 * dot(N, ray.u) * N);
            return getColor(internal_reflected_ray, ray_depth - 1);
        }
    }

    // Omitted handling of light interaction and shadows for brevity
}
```

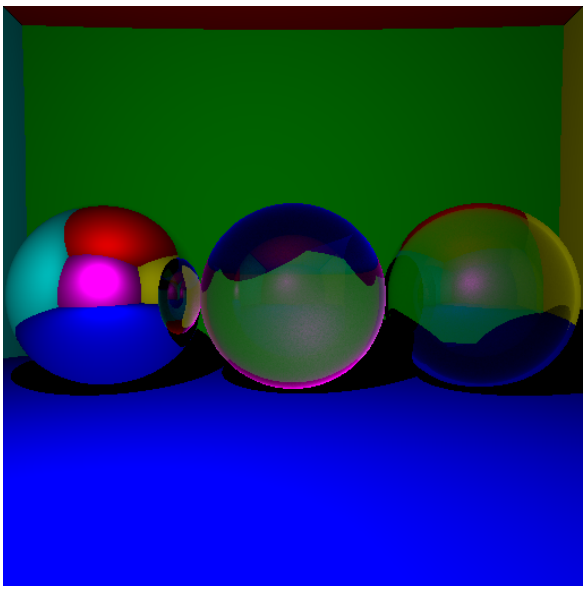

Figure 3: Rendering time: 11 minutes and 40.116 seconds without parallelization.

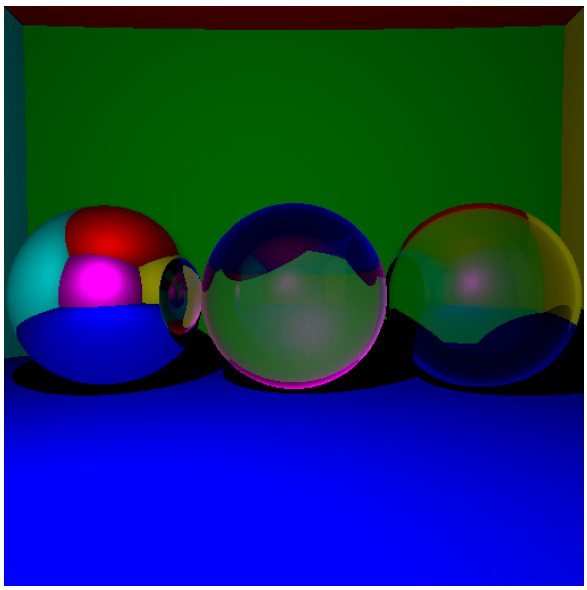

Figure 4: Rendering time: 1 minute and 10.192 seconds with parallelization.

4 Indirect Lighting and Anti Aliasing

In the development of our ray tracer, we initially focused on the implementation of indirect lighting to model the diffuse interreflection of light between surfaces, enhancing the realism of the scene illumination. Following this enhancement, anti-aliasing techniques were introduced to to reduce the visual artifacts at the edges of objects, resulting in smoother and more visually appealing images. Additionally, we implemented a technique where multiple rays are sampled per pixel and their colors averaged.

```
Vector getColor(const Ray& ray, int ray_depth) {
    if (ray_depth < 0) return Vector(0., 0., 0.);
    Intersection intersection = intersect(ray);
    if (!intersection.intersected) {
        return Vector(0., 0., 0.);
    }

    const double epsilon = 1e-4;
    Sphere& sphere = *spheres[intersection.index];
    Vector P = ray.O + ray.u * intersection.t;
```

```
    Vector N = (P - sphere.C).normalize();

    if (sphere.mirror) {
        // Omitted mirror surfaces for brevity
    } else if (sphere.refractive_index != 1.){
        // Omitted refractive surfaces for brevity
    } else{
        double d = (this->light_position - P).norm();
        Vector light_direction = (this->light_position - P).normalize();
        Vector p_shadow = P + epsilon * N;
        Ray ray_shadow(p_shadow, light_direction);
        Intersection shadow_intersection = intersect(ray_shadow);
        double inShadow = (shadow_intersection.intersected && shadow_intersection.t < d) ? 0.0 : 1.0;
        Vector pixel_color = (this->light_intensity / (4 * M_PI * d * d)) * intersection.albedo * \
            inShadow * std::max(0.0, dot(N, light_direction));
        Ray random_ray(P + epsilon * N, random_cos(N));
        pixel_color = pixel_color + intersection.albedo * getColor(random_ray, ray_depth - 1);
        return pixel_color;
    }
}

void boxMuller ( double stdev , double& x , double &y ) {
    double r1 = uniform(engine);
    double r2 = uniform(engine);
    x = sqrt(-2*log(r1))*cos(2*M_PI*r2)*stdev;
    y = sqrt(-2*log(r1))*sin(2*M_PI*r2)*stdev;
}

int main() {
    // Omitted beginning of main function for brevity
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < H; ++i) {
        for (int j = 0; j < W; ++j) {
            Vector color_tmp(0., 0., 0.);
            double x, y;
            for (int k = 0; k< K; ++k) {
                boxMuller(0.5, x, y);
                Vector pixel = Vector(Q[0]+(j+x)+0.5-W/2, Q[1]-(i+y)-0.5+H/2, Q[2]-W/(2*tan(fov/2)));
                Ray ray(Q, (pixel-Q).normalize());
                color_tmp = color_tmp + scene.getColor(ray, ray_depth);
            }
    Vector color = color_tmp/K;
    // Omitted rest of main function for brevity
}
```

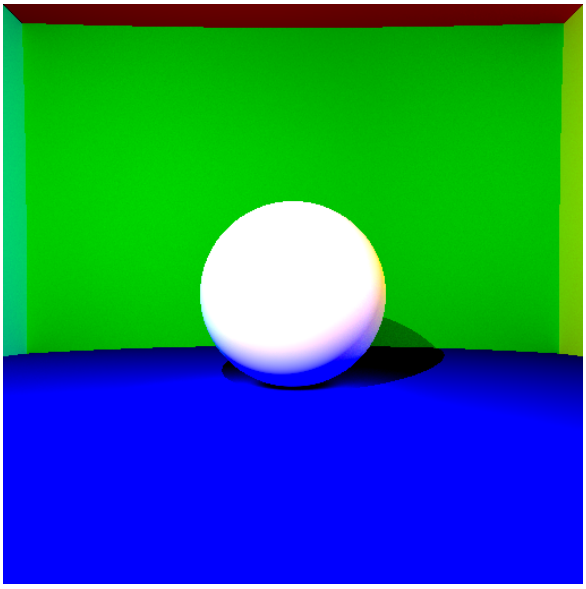

Figure 5: Rendering time: 11 minutes and 36.988 seconds with parallelization, 1000 rays per pixel, indirect lighting and without anti-aliasing.

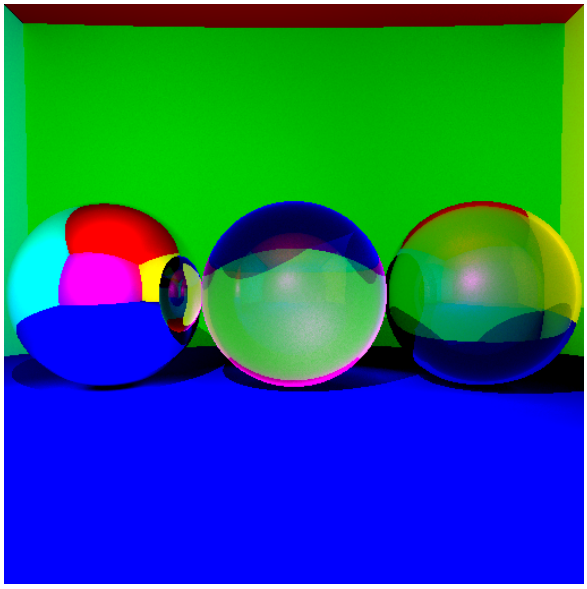

Figure 6: Rendering time: 10 minutes and 12.202 seconds with parallelization, 1000 rays per pixel, indirect lighting and without anti-aliasing.

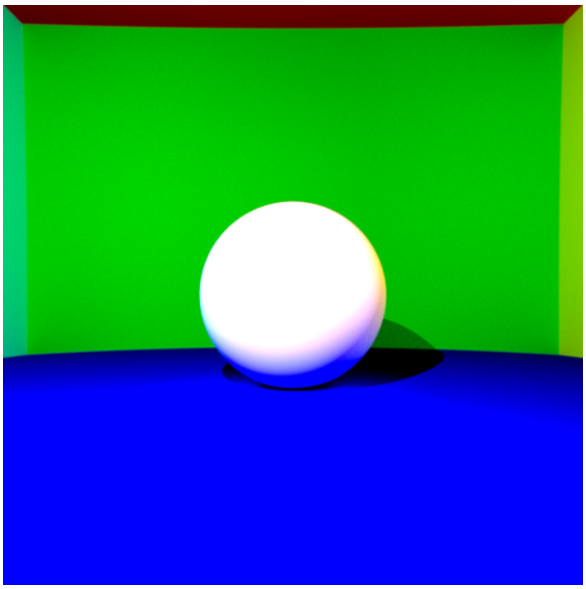

Figure 7: Rendering time: 11 minutes and 20.267 seconds with parallelization, 1000 rays per pixel, indirect lighting and anti-aliasing.

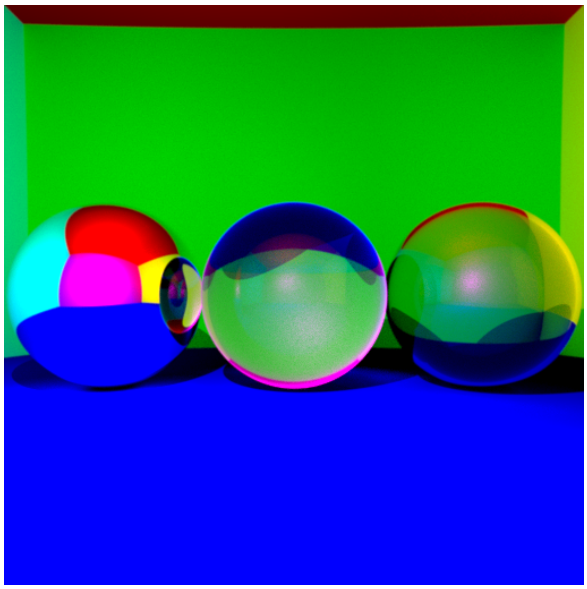

Figure 8: Rendering time: 10 minutes and 3.877 seconds with parallelization, 1000 rays per pixel, indirect lighting and anti-aliasing.

5 Depth of Field

To further refine the realism and visual quality of our ray tracer, we implemented a depth of field effect. This was achieved by sampling rays from a circular aperture, mimicking the behavior of a camera lens as seen in the code segment:

```
int main() {
    // Omitted beginning of main function for brevity
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < H; ++i) {
        for (int j = 0; j < W; ++j) {
            Vector color_tmp(0., 0., 0.);
            double x, y;
            for (int k = 0; k< K; ++k) {
                boxMuller(0.5, x, y);
                double theta = 2*M_PI*uniform(engine);
                double aperture_shape = sqrt(uniform(engine))*aperture_radius;
                Ray ray(Q+Vector(aperture_shape*cos(theta), aperture_shape*sin(theta), 0), \
                    (Q+focus_distance*Vector(0.5+j+x-W*0.5, y-i+H*0.5-0.5, -W/(2*tan(fov/2))).normalize() \
                    - Q - Vector(aperture_shape*cos(theta), aperture_shape*sin(theta),0)).normalize());
                color_tmp = color_tmp + scene.getColor(ray, ray_depth);
            }
    // Omitted rest of main function for brevity
}
```

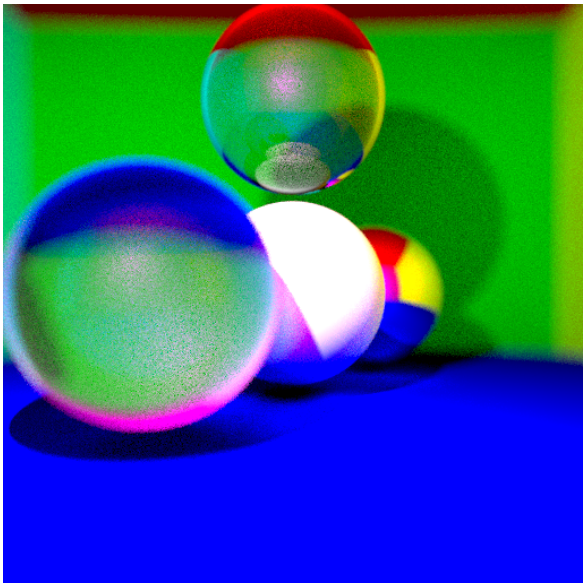

Figure 9: Rendering time: 1 minute and 0.024 seconds with depth of field, 100 rays per pixels, aperture radius of 1.5 and focus distance of 50.

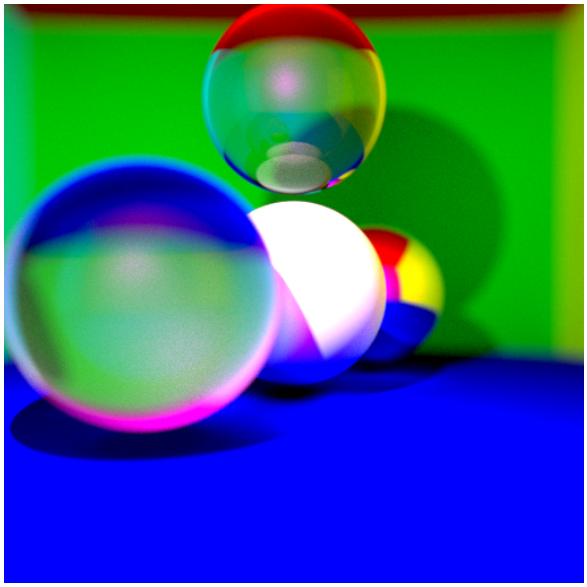

Figure 10: Rendering time: 10 minutes and 12.317 seconds with depth of field, 1000 rays per pixels, aperture radius of 1.5 and focus distance of 50.

6 Triangle Meshes and Bounding Boxes

We then added triangle meshes to our ray tracer. For this purpose, we utilize the code provided by our professor, which effectively handles the reading and loading of these meshes. To optimize performance and speed up the rendering process, we implemented bounding boxes around these triangle meshes. Bounding boxes help in quickly determining whether a ray intersects with a complex object by first checking for intersection with the simpler bounding box that encloses it. This approach reduces the number of expensive intersection calculations required with the huge number of triangles, thus improving overall efficiency. In the development of our ray tracer, we've integrated triangle meshes using code provided by our professor to enrich scene detail and complexity. To enhance rendering efficiency, we employ bounding boxes with these meshes, significantly speeding up the process by reducing the need for detailed intersection calculations. Additionally, we've implemented a Geometry base class from which both Sphere and Triangle Mesh classes inherit. This approach centralizes

common attributes like material properties and intersection methods, enabling consistent handling across different geometric forms. The Triangle Mesh class specifically supports mesh transformations and incorporates a bounding box to further optimize ray-tracing performance.

```
class TriangleMesh : public Geometry{
public:
    double scaling_factor;
    Vector translation;
    BoundingBox boundingbox;
   ~TriangleMesh() {}
    TriangleMesh(double scaling_factor, Vector translation, Vector albedo, double refractive_index = 1.0, \
        bool mirror = false) {
        this->scaling_factor = scaling_factor;
        this->translation = translation;
        this->albedo = albedo;
        this->refractive_index = refractive_index;
        this->mirror = mirror;
    }
    // Omitted readOBJ function for brevity
    void compute_bounding_box() {
        double min_x = MAXFLOAT, min_y = MAXFLOAT, min_z = MAXFLOAT;
        double max_x = -MAXFLOAT, max_y = -MAXFLOAT, max_z = -MAXFLOAT;
        for (const auto& index : this->indices) {
            std::vector<Vector> triangle_vertices = {this->vertices[index.vtxi], this->vertices[index.vtxj],\
                this->vertices[index.vtxk]};
            for (const Vector& vertex : triangle_vertices) {
                Vector transformed_vertex = scaling_factor * vertex + translation;
                min_x = std::min(min_x, transformed_vertex[0]);
                max_x = std::max(max_x, transformed_vertex[0]);
                min_y = std::min(min_y, transformed_vertex[1]);
                max_y = std::max(max_y, transformed_vertex[1]);
                min_z = std::min(min_z, transformed_vertex[2]);
                max_z = std::max(max_z, transformed_vertex[2]);
            }
        }
        this->boundingbox = BoundingBox(Vector(min_x, min_y, min_z), Vector(max_x, max_y, max_z));
    }

    bool bounding_box_intersection(const Ray &ray, const BoundingBox boundingbox, double &t) const {
        double tx0, ty0, tz0;
        double tx1, ty1, tz1;
        double t_B_min, t_B_max;
        Vector N;

        N = Vector(1,0,0);
        t_B_min = dot(this->boundingbox.B_min - ray.O, N) / dot(ray.u, N);
        t_B_max = dot(this->boundingbox.B_max - ray.O, N) / dot(ray.u, N);
        tx0 = std::min(t_B_min, t_B_max);
        tx1 = std::max(t_B_min, t_B_max);

        N = Vector(0,1,0);
        t_B_min = dot(this->boundingbox.B_min - ray.O, N) / dot(ray.u, N);
        t_B_max = dot(this->boundingbox.B_max - ray.O, N) / dot(ray.u, N);
        ty0 = std::min(t_B_min, t_B_max);
        ty1 = std::max(t_B_min, t_B_max);

        N = Vector(0,0,1);
        t_B_min = dot(this->boundingbox.B_min - ray.O, N) / dot(ray.u, N);
        t_B_max = dot(this->boundingbox.B_max - ray.O, N) / dot(ray.u, N);
        tz0 = std::min(t_B_min, t_B_max);
        tz1 = std::max(t_B_min, t_B_max);

        double first_intersection_t = std::max({tx0, ty0, tz0});
        double last_intersection_t = std::min({tx1, ty1, tz1});

        if (first_intersection_t < last_intersection_t) {
            t = first_intersection_t;
            return true;
        }
```

```
        return false;
    }
};

int main() {
    // Omitted beginning of main function for brevity
    TriangleMesh* cat = new TriangleMesh(0.6, Vector(0, -10, 0), Vector(1., 1., 1.));
    cat->readOBJ("cat_model/cat.obj");
    cat->compute_bounding_box();
    scene.addGeometry(cat);
    // Omitted rest of main function for brevity
}
```

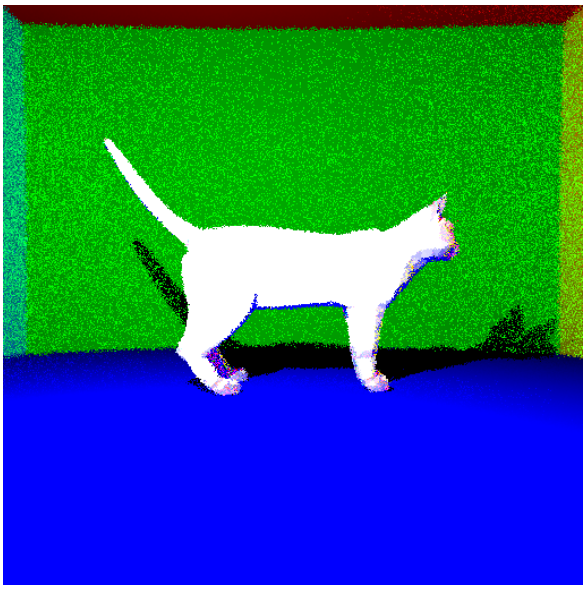

Figure 11: Rendering time: 48.37 seconds with 1 ray per pixels, cat triangle mesh and bounding box.

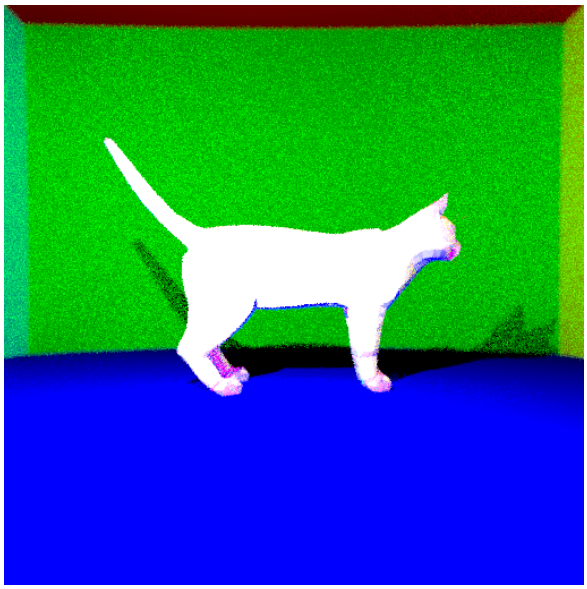

Figure 12: Rendering time: 9 minutes and 0.531 seconds with 10 rays per pixels, cat triangle mesh and bounding box.

7 Bounding Volume Hierarchies

To optimize rendering performance, we also implemented Bounding Volume Hierarchies (BVH) within our ray tracer. BVH are data structures that recursively divide a scene into nested bounding boxes, allowing for rapid traversal and intersection tests. The incorporation of BVH effectively minimizes computation times and enhances the rendering speed, especially in complex scenes with a high number of objects, in our case triangle meshes.

```
    class TriangleMesh : public Geometry{
public:
    double scaling_factor;
    Vector translation;
    BoundingBox bounding_box;
    Node* root;
   ~TriangleMesh() {}
    TriangleMesh(double scaling_factor, Vector translation, Vector albedo, double refractive_index \
        = 1.0, bool mirror = false) {
        this->scaling_factor = scaling_factor;
        this->translation = translation;
        this->albedo = albedo;
        this->refractive_index = refractive_index;
```

```
    this->mirror = mirror;
    this->root = new Node;
}

void readOBJ(const char* obj) {
    // Omitted begininng of readOBJ function for brevity
    fclose(f);
    this->build_BVH(this->root, 0, indices.size());
}

void build_BVH(Node *node, int starting_triangle, int ending_triangle) {
    node->bounding_box = compute_bounding_box(starting_triangle, ending_triangle);
    node->starting_triangle = starting_triangle;
    node->ending_triangle = ending_triangle;
    Vector diagonal = node->bounding_box.B_max - node->bounding_box.B_min;
    Vector middle_diagonal = node->bounding_box.B_min + 0.5*diagonal;
    int longest_axis = 0;
    double max = - MAXFLOAT;
    for (int i = 0; i < 3; i++) {
        if (abs(diagonal[i]) > max) {
            max = abs(diagonal[i]);
            longest_axis = i;
        }
    }
    int pivot_index = starting_triangle;
    for (int i = starting_triangle; i < ending_triangle; i++) {
        Vector v1 = this->vertices[this->indices[i].vtxi]*scaling_factor + translation;
        Vector v2 = this->vertices[this->indices[i].vtxj]*scaling_factor + translation;
        Vector v3 = this->vertices[this->indices[i].vtxk]*scaling_factor + translation;
        Vector barycenter = (v1 + v2 + v3)/3.;
        if (barycenter[longest_axis] < middle_diagonal[longest_axis]) {
            std::swap(indices[i], indices[pivot_index]);
            pivot_index++;
        }
    }
    if (pivot_index <= starting_triangle || pivot_index >= ending_triangle - 1 || ending_triangle \
        - starting_triangle < 5) {
        return;
    }
    node->left_child = new Node();
    node->right_child = new Node();
    this->build_BVH(node->left_child, starting_triangle, pivot_index);
    this->build_BVH(node->right_child, pivot_index, ending_triangle);
}

Intersection intersect(const Ray &ray) const {
    Intersection intersection;
    intersection.intersected = false;
    double t;
    double min_t = MAXFLOAT;
    if (!bounding_box_intersection(ray, this->root->bounding_box, t)) {
        return intersection;
    }
    std::list<Node*> nodes_to_visit;
    nodes_to_visit.push_front(this->root);
    while(!nodes_to_visit.empty()) {
        Node* current_node = nodes_to_visit.back();
        nodes_to_visit.pop_back();
        if (current_node->left_child) {
            if (bounding_box_intersection(ray, current_node->left_child->bounding_box, t)) {
                if (t < min_t) {
                    nodes_to_visit.push_back(current_node->left_child);
                }
            }
            if (bounding_box_intersection(ray, current_node->right_child->bounding_box, t)) {
                if (t < min_t) {
                    nodes_to_visit.push_back(current_node->right_child);
                }
            }
```

```
            }
            else {
                Vector A, B, C, e_1, e_2, N;
                for (int i = current_node->starting_triangle; i < current_node->ending_triangle; i++) {
                    A = vertices[this->indices[i].vtxi]*scaling_factor + translation;
                    B = vertices[this->indices[i].vtxj]*scaling_factor + translation;
                    C = vertices[this->indices[i].vtxk]*scaling_factor + translation;
                    e_1 = B - A;
                    e_2 = C - A;
                    N = cross(e_1, e_2);
                    double beta = dot(e_2, cross(A - ray.O, ray.u))/dot(ray.u, N);
                    double gamma = - dot(e_1, cross(A - ray.O, ray.u))/dot(ray.u, N);
                    double alpha = 1.0 - beta - gamma;
                    if (alpha > 0.0 && beta > 0.0 && gamma > 0.0) {
                        double t = dot(A - ray.O, N)/dot(ray.u, N);
                        if (t > 0 && min_t > t) {
                            min_t = t;
                            intersection.intersected = true;
                            intersection.t = t;
                            intersection.P = A + e_1*beta + e_2*gamma;
                            intersection.N = N;
                            intersection.albedo = this->albedo;
                        }
                    }
                }
            }
        }
        return intersection;
    }
}
```

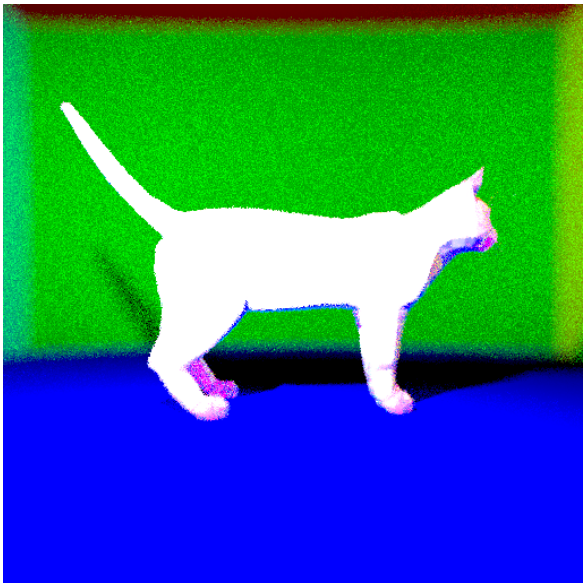

Figure 13: Rendering time: 12.341 seconds with 10 rays per pixels, cat triangle mesh and BVH.

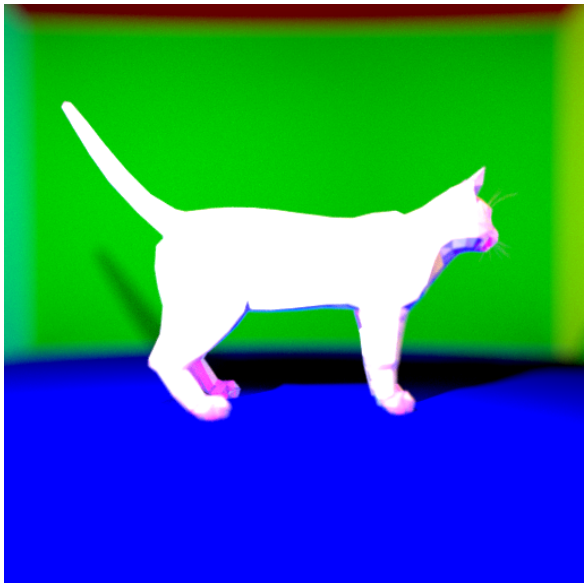

Figure 14: Rendering time: 20 minutes and 34.309 seconds with 1000 rays per pixels, cat triangle mesh and BVH.