Ray Tracer Report

COMS 3360: Introduction to Computer Graphics

Ethan Roe

December 2025

1 Codebase Overview

1.1 File Structure

- run.sh: A shell script that first compiles the program using make and launches the program. After the program has ended, it converts the .ppm image to a .png and then displays the .ppm. The display command may or may not work on your system.

- Makefile: The project is compiled using make. Use make to compile the program, which produces a binary named "ray-tracer".

- /src: Contains main.cc which is where the main function and all the scene functions are located. The program is interactive, enter the number associated with the scene in the list to render that scene.

- /obj: Contains .obj files for loading and rendering triangle meshes.

- /media: Contains .jpg files for loading textures onto shapes.

- /include: Contains the entire implementation of the ray tracer in the form of header files. See README for a description of all the files.

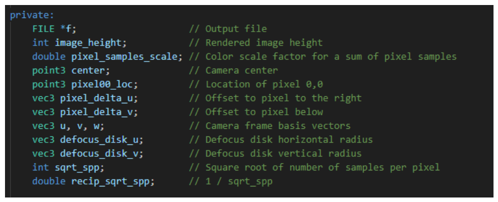
- /images: Contains images I generated from scenes that I used while developing the ray tracer. All images have at least 1000 samples per pixel.

- /external: Contains external libraries. The only external library the ray tracer is currently using is stb_image which is used for loading .obj files.

- /demo: Demo code provided in class and on piazza (not used in program at all).

1.2 Ray Generation

The camera defines an orthonormal basis (u, v, w) from the look-from, look-at, and up vector, respectively. Rays are generated by mapping each pixel coordinate to a point on the image and constructing a direction vector from the camera origin through that point.



```
private:
    FILE *f;                     // Output file
    int image_height;            // Rendered image height
    double pixel_samples_scale;  // Color scale factor for a sum of pixel samples
    point3 center;               // Camera center
    point3 pixel00_loc;          // Location of pixel 0,0
    vec3 pixel_delta_u;          // Offset to pixel to the right
    vec3 pixel_delta_v;          // Offset to pixel below
    vec3 u, v, w;                // Camera frame basis vectors
    vec3 defocus_disk_u;         // Defocus disk horizontal radius
    vec3 defocus_disk_v;         // Defocus disk vertical radius
    int sqrt_spp;                // Square root of number of samples per pixel
    double recip_sqrt_spp;       // 1 / sqrt_spp
```

Figure 1: Private variables for the camera class (camera.h)

The ray class (defined in ray.h) stores an origin, a direction, and a time parameter. The time parameter is used for a motion blur effect.

1.3 Scene Representations

The renderer supports three different geometry types. All three geometry types have one parameter in common, that being a material parameter (more on materials in 3.2).

- Spheres (defined in sphere.h)
  The sphere class stores a center point, radius, and material parameter.

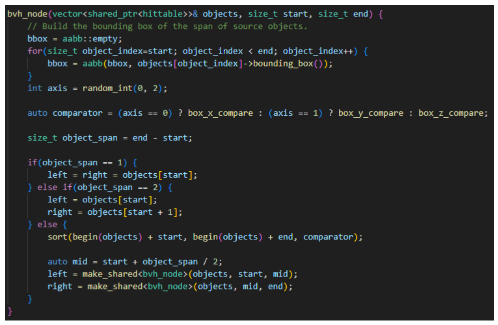
- Quads (defined in quads.h)
  The quad class stores a corner point Q, two vectors u, v that originate from the point Q, and a material parameter. The other three corners of the quad can be determined using Q, u, and v

- Triangles (defined in triangle.h)
  The triangle class stores three point parameters, a, b, and c, and a material parameter.

1.4 Acceleration Structure

- Axis-aligned bounding boxes (aabb.h):
  The aabb class stores the min and max coordinates for each object or BVH node. We determine if a ray intervals with an aabb using the slab method. Functions related to intervals can be found in interval.h.

- Bounding volume hierarchy node (bvh.h):
  The bvh_node class is used to build an aabb around a list of hittable objects. The bvh is recursively split along a random axis, sorted based on the objects bounding box, and then split into sub-nodes from the middle of the sorted list (Shown in figure 2).



```
bvh_node(vector<shared_ptr<hittable>>& objects, size_t start, size_t end) {
    // Build the bounding box of the span of source objects.
    bbox = aabb::empty;
    for(size_t object_index=start; object_index < end; object_index++) {
        bbox = aabb(bbox, objects[object_index]->bounding_box());
    }
    int axis = random_int(0, 2);

    auto comparator = (axis == 0) ? box_x_compare : (axis == 1) ? box_y_compare : box_z_compare;

    size_t object_span = end - start;

    if(object_span == 1) {
        left = right = objects[start];
    } else if(object_span == 2) {
        left = objects[start];
        right = objects[start + 1];
    } else {
        sort(begin(objects) + start, begin(objects) + end, comparator);

        auto mid = start + object_span / 2;
        left = make_shared<bvh_node>(objects, start, mid);
        right = make_shared<bvh_node>(objects, mid, end);
    }
}
```

Figure 2: bvh_node constructor (bvh.h)

2 Rendering Process

2.1 Intersections

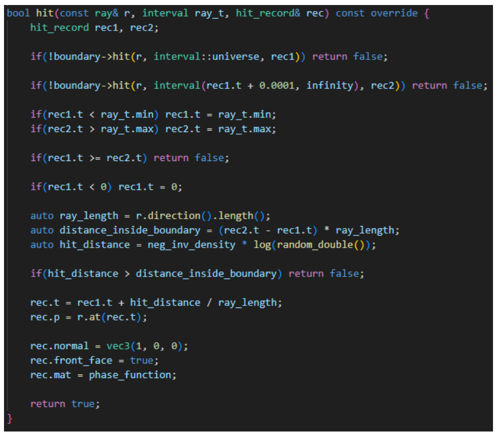
Each ray generated by the camera uses the scenes BVH structure to determine the closest intersection relative to the camera. When a ray intersects an object, the respective hit function is called to record the surface normal, intersection point, and material of the object.

2.2 Materials (material.h)

The renderer supports lambertian, metal, dielectric, diffuse, and isotropic materials. Each of these material types are defined in their own classes. They each implement a scatter method that determines how rays reflect off of an object of said material.

2.3 Volumes (constant_medium.h)

Volumetric effects are defined by a boundary and a density parameter. The volumetric effect is represented by a volume of constant medium. The hit function determines the entry and exit points of a ray and the samples from an exponential distribution on the density to determine the distance the ray scatters (Figure 3).

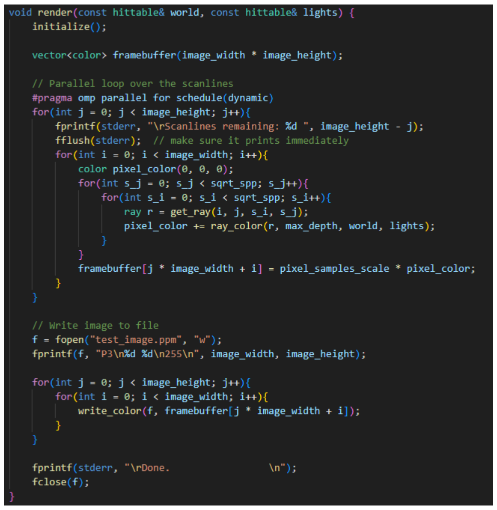


```
bool hit(const ray& r, interval ray_t, hit_record& rec) const override {
    hit_record rec1, rec2;

    if(!boundary->hit(r, interval::universe, rec1)) return false;

    if(!boundary->hit(r, interval(rec1.t + 0.0001, infinity), rec2)) return false;

    if(rec1.t < ray_t.min) rec1.t = ray_t.min;
    if(rec2.t > ray_t.max) rec2.t = ray_t.max;

    if(rec1.t >= rec2.t) return false;

    if(rec1.t < 0) rec1.t = 0;

    auto ray_length = r.direction().length();
    auto distance_inside_boundary = (rec2.t - rec1.t) * ray_length;
    auto hit_distance = neg_inv_density * log(random_double());

    if(hit_distance > distance_inside_boundary) return false;

    rec.t = rec1.t + hit_distance / ray_length;
    rec.p = r.at(rec.t);

    rec.normal = vec3(1, 0, 0);
    rec.front_face = true;
    rec.mat = phase_function;

    return true;
}
```

Figure 3: Hit function in constant_medium.h

2.4 Parallelization (camera.h)

The renderer uses OpenMP to parallize the rendering of pixels. The outer loop of the renderer dynamically allocates scanlines to CPU threads. The render function can be seen in figure 4.



```
void render(const hittable& world, const hittable& lights) {
    initialize();

    vector<color> framebuffer(image_width * image_height);

    // Parallel loop over the scanlines
    #pragma omp parallel for schedule(dynamic)
    for(int j = 0; j < image_height; j++){
        fprintf(stderr, "\rScanlines remaining: %d ", image_height - j);
        fflush(stderr);  // make sure it prints immediately
        for(int i = 0; i < image_width; i++){
            color pixel_color(0, 0, 0);
            for(int s_j = 0; s_j < sqrt_spp; s_j++){
                for(int s_i = 0; s_i < sqrt_spp; s_i++){
                    ray r = get_ray(i, j, s_i, s_j);
                    pixel_color += ray_color(r, max_depth, world, lights);
                }
            }
            framebuffer[j * image_width + i] = pixel_samples_scale * pixel_color;
        }
    }

    // Write image to file
    f = fopen("test_image.ppm", "w");
    fprintf(f, "P3\n%d %d\n255\n", image_width, image_height);

    for(int j = 0; j < image_height; j++){
        for(int i = 0; i < image_width; i++){
            write_color(f, framebuffer[j * image_width + i]);
        }
    }

    fprintf(stderr, "\rDone.                 \n");
    fclose(f);
}
```

Figure 4: Render function in camera.h

2.5 Triangle Meshes (mesh_loader.h)

Meshes are loaded in mesh_loader.h. The program can only load meshes using the .obj file format. Figure 5 shows a render of the stanford bunny mesh.

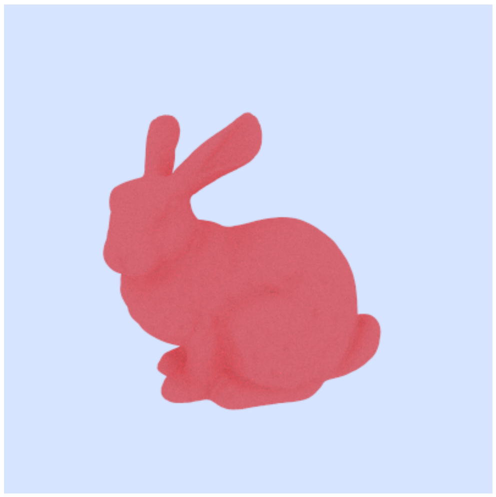
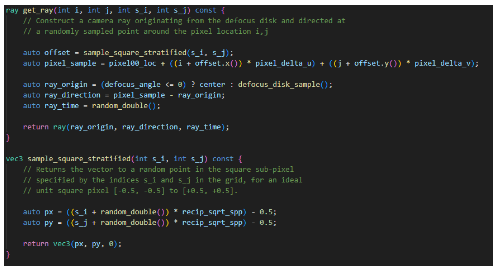


Figure 5: Render of the stanford bunny mesh

2.6 Sampling (camera.h)

The renderer uses Monte Carlo sampling to estimate the pixel color. Pixels are sampled using stratified sampling to evenly sample pixels and reduce noise in the image compared to uniform sampling (Figure 6).

Importance sampling is implemented in the ray_color function in camera.h, with the use of probability density functions from pdf.h. Importance sampling helps reduce noise in scenes by preferring rays that are more likely to contribute light to the scene. This has the effect of reducing noise in scenes with small or bright lights when compared to a uniform sampling method, making the image smoother.



```cpp
ray get_ray(int i, int j, int s_i, int s_j) const {
    // Construct a camera ray originating from the defocus disk and directed at
    // a randomly sampled point around the pixel location i,j

    auto offset = sample_square_stratified(s_i, s_j);
    auto pixel_sample = pixel00_loc + ((i + offset.x()) * pixel_delta_u) + ((j + offset.y()) * pixel_delta_v);

    auto ray_origin = (defocus_angle <= 0) ? center : defocus_disk_sample();
    auto ray_direction = pixel_sample - ray_origin;
    auto ray_time = random_double();

    return ray(ray_origin, ray_direction, ray_time);
}

vec3 sample_square_stratified(int s_i, int s_j) const {
    // Returns the vector to a random point in the square sub-pixel
    // specified by the indices s_i and s_j in the grid, for an ideal
    // unit square pixel [-0.5, -0.5] to [+0.5, +0.5].

    auto px = ((s_i + random_double()) * recip_sqrt_spp) - 0.5;
    auto py = ((s_j + random_double()) * recip_sqrt_spp) - 0.5;

    return vec3(px, py, 0);
}
```

Figure 6: Stratifed sampling function and its use in get_ray().

3 Final Renders

Figure 7 and figure 8 were both rendered using $10,000$ samples per pixel. Figure 7 took approximately 1 hour and 45 minutes to render and Figure 8 took approximately 1 hour to render.

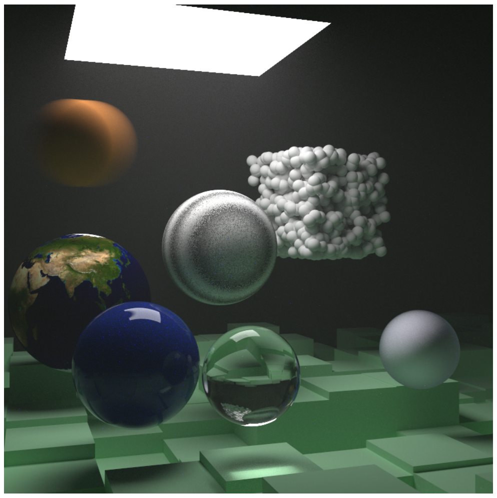


Figure 7: Final scene from "Ray Tracing: The Next Week".

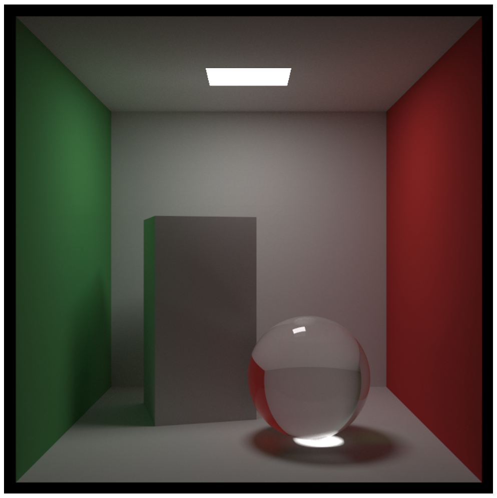


Figure 8: Final scene from "Ray Tracing: The Rest of Your Life".