CS184 Project 3-1: Pathtracer

Due: March 15, 2022

Author: *Albert Wen*

In this project I was able to successfully generate rays in a scene. My primary goal was to render the provided .dae files into PNGs to demonstrate the concepts related to path tracing.

Part 1: Ray Generation and Scene Intersection

Ray generation and primitive intersection implementation involved writing code in `camera.cpp`, `triangle.cpp`, and `spheres.cpp`. In order to perform ray tracing on a setting and render the results within a virtual camera, I performed the following steps to generate the rays: * Perform a mapping between the point of ray intersection on the plane of an image sensor to the camera coordinates then world coordinates. * Generate a `ray` object with the `Vector3D` positions of the camera's `pos` and ray intersection on the plane of the original image. The correct implementation of `camera.cpp` results in the images below.

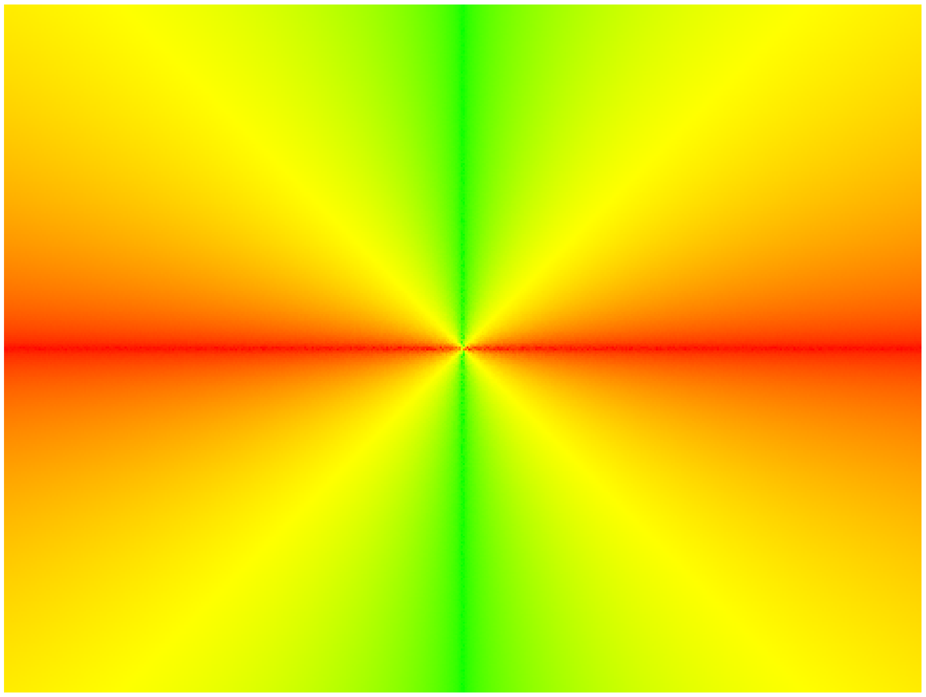

Figure 1: CBempty.dae after task 2

Next came the raytracing of primitives in 3D space. Two primitives were

Figure 2: banana.dae after task 2

used for this implementation: triangles and spheres. Determining the point of intersection between a triangle and ray emitted from the camera required implementing the Möller Trumbore algorithm, which establishes a linear system of equations for generating barycentric coordinates of a point in a triangle and the time of intersection from the ray's origin. The solutions to this system of equations determine the time of intersection of the ray with the plane in which the triangle lies and if the intersection lies within the triangle itself. The time of intersection, point of intersection, normal unit vector of the point of intersection, and the triangle's bidirectional scattering function (BSDF) are then stored in an `Intersection` object for rendering. One critical check that was performed was to see if the time of intersection fell between the `ray` object's `min_t` and `max_t` in order to be valid, as the `ray` is not intended to continue infinitely.

```
bool Triangle::intersect(const Ray &r, Intersection *isect) const {
    Vector3D E1 = p2 - p1;
    Vector3D E2 = p3 - p1;
    Vector3D S = r.o - p1;
    Vector3D S1 = cross(r.d, E2);
    Vector3D S2 = cross(S, E1);
    float t = dot(S2, E2) / dot(S1, E1);
    if (has_intersection(r)) {
        isect->t = t;
        Vector3D intPt = r.at_time(t);
        float b1 = dot(S1, S) / dot(S1, E1);
        float b2 = dot(S2, r.d) / dot(S1, E1);
        Vector3D nIntPt = (1 - b1 - b2) * n1 + b1 * n2 + b2 * n3;
        isect->n = nIntPt;
        isect->primitive = this;
        isect->bsdf = get_bsdf();
    }
    return has_intersection(r);
```

- Code snippet from `Triangle::intersect` in `triangle.cpp`

Part 2: Bounding Volume Hierarchy

Bounding Volume Hierarchy (BVH) is a binary tree of bounding boxes to divide the space among the primitives within a scene. I was not able to get this part working because I have not figured out how to divide the space along the longest axis created by the primitives' respective centroids. This prevented me from successfully implementing this part and the remaining parts of this project.

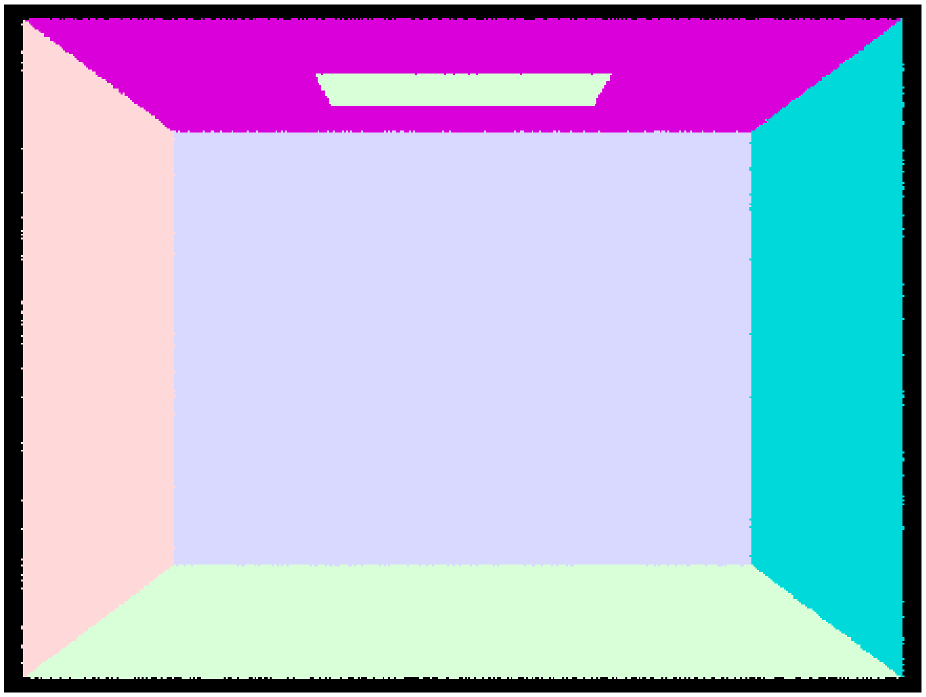

Figure 3: CBempty.dae after task 3

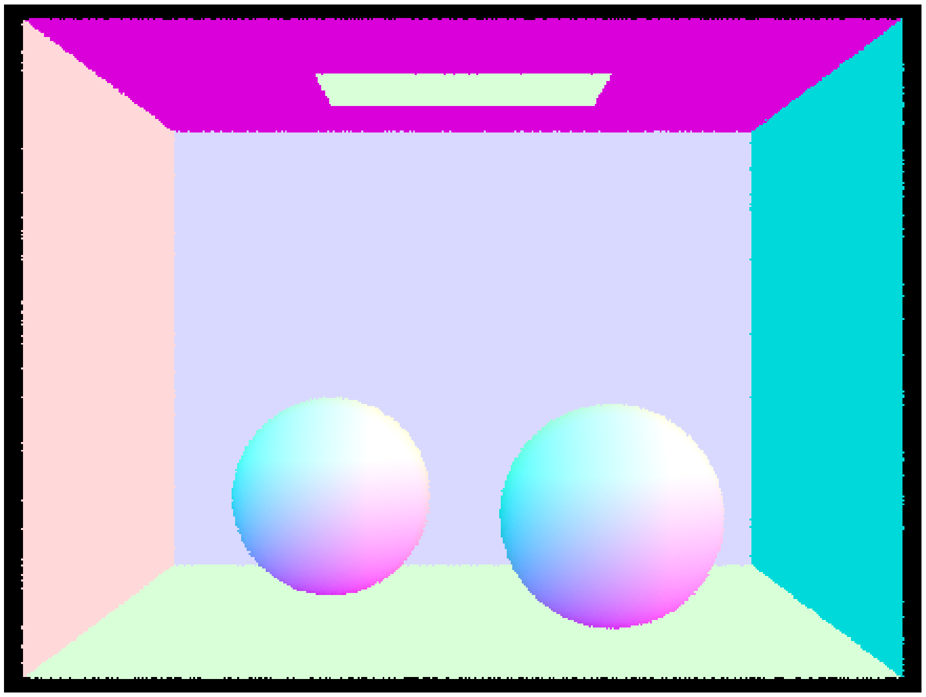

Figure 4: CBspheres_lambertian.dae after task 4

Part 3: Direct Illumination

Part 4: Global Illumination

Part 5: Adaptive Sampling

Web Page

Written in Markdown, hosted on GitHub