COMS 3360 Project Report

Benjamin Kam Jia Zhiang
`bkam@iastate.edu`

1 Abstract

This document outlines the development of a ray tracer. We will go through the features implemented and explain how they work, with visual displays of the results produced by each feature as well as a combination of the features. The photos in this report are .png files as LaTeXdoes not support .ppm files

2 Features

2.1 Configurable Camera

The camera is used to generate rays that simulate how light enters the viewer's eye and these generated rays are sent into the scene to retraces the rays through the scene to reconstruct the image. The implementation of the camera can be found in camera.h. For this camera, we fix the aspect ratio to 1.0, producing square images. To remain computationally-friendly, the max_depth is capped at 10. The camera is adjustable by changing the `lookfrom`, `lookat`, `vup` and `focus_dist`.

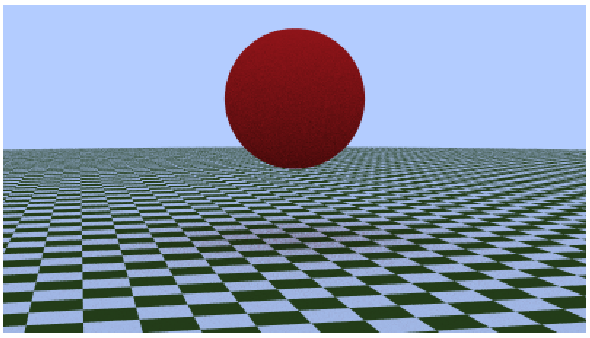
Figure 1: Looking from point3(13, 2, 3)

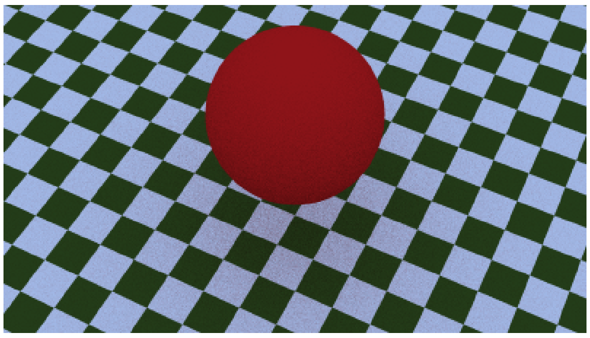
Figure 2: Looking from point3(-6, 10, 3)

2.2 Anti-aliasing

To reduce the sharp edges caused by the edge pixels, we added anti-aliasing to our camera as it helps to visually soften the edges. We move from point sampling to sampling the region centered at the pixel by sending multiple rays per pixel at random points within the pixel's dimensions and then averaging the colors from the rays. This idea was inspired from `A Pixel is Not a Little Square`.

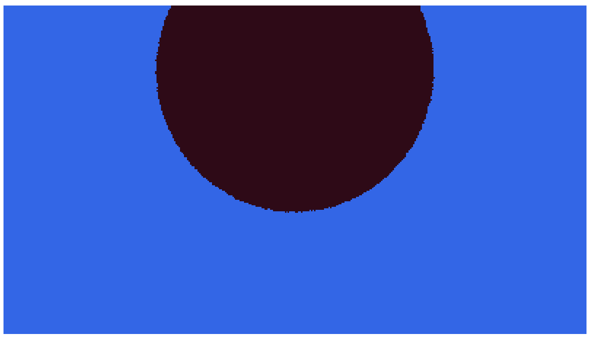
Figure 3: Anti-aliasing Before

2.3 Ray-Shape Intersections

For ray/sphere intersections, we based the logic off mathematical definition of a sphere with Center C and radius r, where p is any point on the sphere.

$$(p - C)(p - C) = r \tag{1}$$

The intersections are calculated by substituting the ray equation into (1), where the algorithm ultimately solves a quadratic equation.

For ray/triangle intersections, we implemented the Möller–Trumbore Algorithm using barycentric

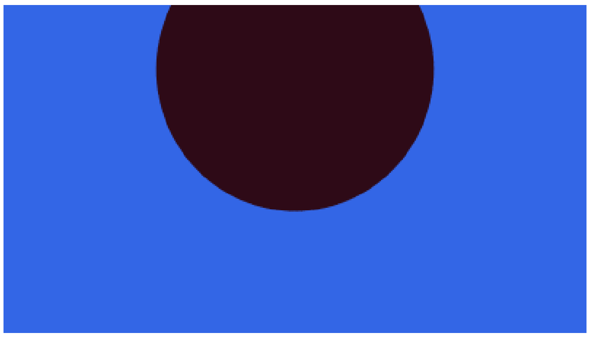
Figure 4: Anti-aliasing After

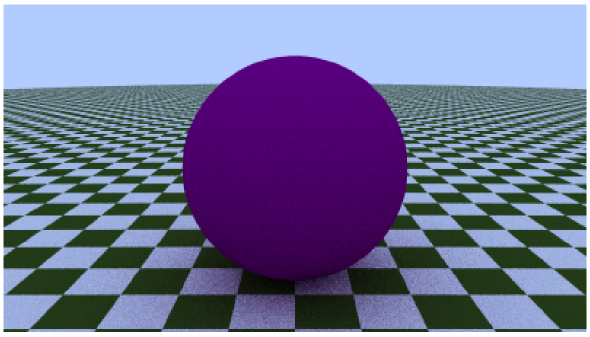
Figure 5: Purple Sphere

coordinates. Every point on a triangle can be represented as:

$$P = v0 + u(v1 - v0) + v(v2 - v0) \quad (2)$$

Where $u, v \geq 0$ and $u + v \leq 1$. This algorithm uses vector identities to avoid building an exhaustive matrix, being cost-efficient.

For ray/quadrilateral intersections, we employ a two-step method. Firstly, we identify the point where the ray intersects the plane of the quadrilateral. Having found point p, it is expressed in a 2D coordinate system:

$$P - Q = \alpha u + \beta v \quad (3)$$

Where we solve for α, β accordingly:

$$\alpha = w((P - Q)v) \quad (4)$$

$$\beta = w(u(P - Q)) \quad (5)$$

Lastly, `aabb.h` is implemented to assist `bvh.h`. Axis-Aligned Bounding Boxes uses 3 slabs, treating the bounding box as the intersection of all 3 slabs. For each slab, we compute a time interval - when a ray enters and leaves a slab. The ray only hits the aabb when there is a time interval where it is inside all 3 slabs. This allows us to better filter unwanted rays.

2.4 Texture loading

We used `stb_image.h`, a third-party library to load textures and in this class, we support these texture types - single color, checker pattern, external image files and perlin noise texture.

To use `stb_image.h`, we had to add `image_helper.h`, which wraps the `stb` image library to allow us to load images.

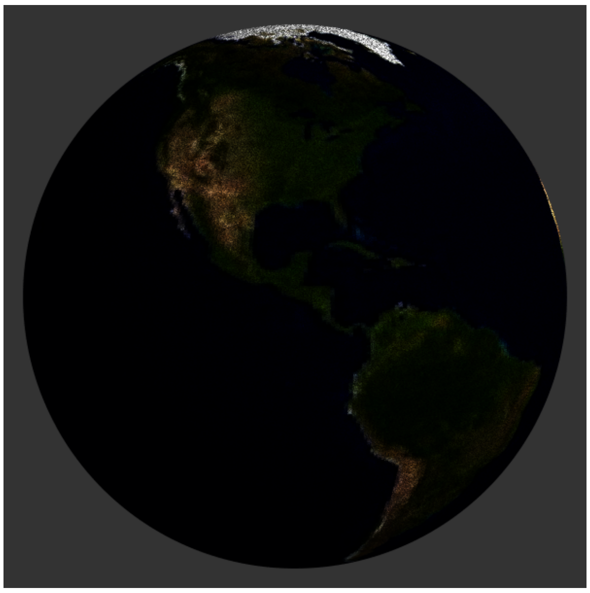
Figure 6: Image of Earth as Texture

2.5 Triangle meshes and textures

We used a third-party library to help with loading triangle meshes, `tiny_obj_loader.h`. The system reads OBJ files and extracts vertex and face data, creating individual triangle objects. Finally, they are organized into a BVH acceleration structure where it is rendered efficiently. Although this is implemented, there is a scaling issue, but it works fine with properly scaled models.

2.6 Bounding Volume Hierarchy

The spatial subdivision acceleration structure that was implemented in our ray tracer is BVH. Each scene object provides an AABB and through ray traversal, AABB tests can be conducted quickly as only child nodes with intersected bounding boxes are explored. As a result of this selective filtering of rays, our ray tracer experienced massive speed ups for complex scenes.

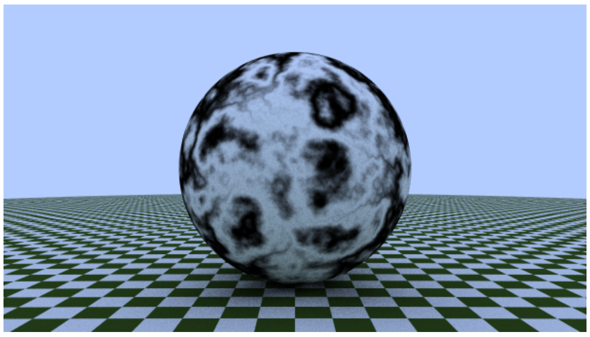
Figure 7: Perlin Texture on Sphere

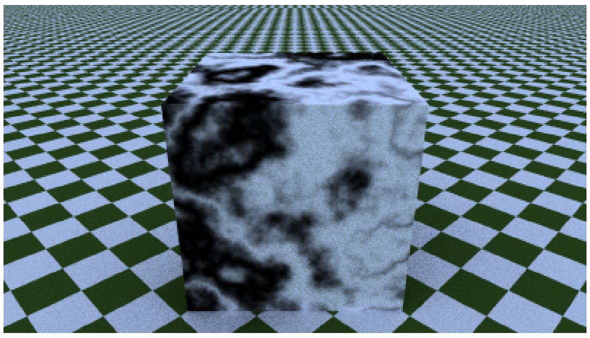
Figure 8: Perlin Cube produced with `cube.obj`

2.7 Materials

In the `material.h` class, we implemented lambertian, metals, dielectric, diffuse and isotropic materials. Lambertian was built by scattering rays randomly around the normal. Metal, by creating `reflect()`, which makes a perfect mirror reflection. Dielectric was able to refract light and execute total internal reflection through applying the physics of Snell's law, the Fresnel Equations and Schlick's approximation. To implement diffuse, we made sure that the material only emitted light, not scatter them. Whereas, isotropic scattered light equally in all directions.

2.8 Motion Blur

To implement motion blur, the rays need to be constrained by time. Now each ray carries a time parameter to represent when they were emitted from the camera. Moving shapes now holds their start and end times, interpolating between them based on the current time stamp. The motion blur streaks are due to the camera sampling random times uniformly across the shutter interval.

2.9 Emissive materials

Emissive materials are implemented in the `diffuse_light` class, overriding the `emitted()` method. This returns a bright color while returning

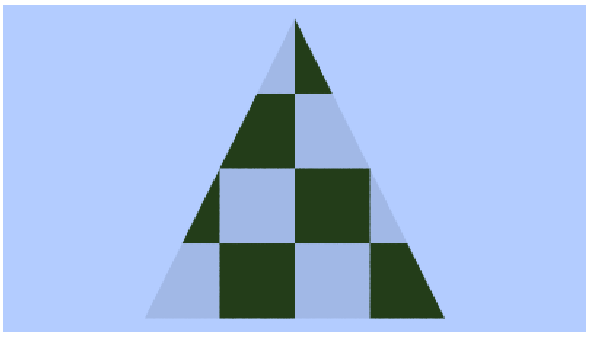
Figure 9: Checkered Texture on Triangle

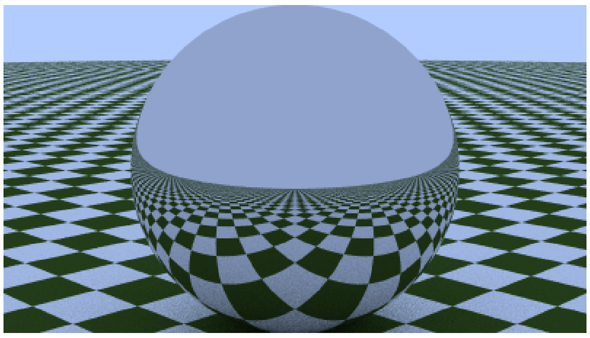
Figure 10: Specular Material

false from scatter, which means that there is no reflection, only emission.

2.10 Volumes

In `constant_medium`, we implemented volumetric rendering. The volume is defined by a boundary object and has constant density throughout. When a ray enters the volume, we sample a random scattering distance based on the density. If there is scattering within the volume, the ray is redirecte using isotropic scattering. Through this, the Beer-Lambert law for light attenuation in participating media is implemented.

2.11 Depth of Field

In `camera.h`, we utilize a defocus disk as a virtual lens with its size represented by `defocus_angle` and the focus plane at `focus_dist`. For each ray, we sample a random point on the defocus disk while maintaining the ray direction toward the corresponding point on the focus plane. This causes objects close to the focus distance to appear sharper whereas going further from the focus distance appear more blurry. This effect produces an physically accurate implementation of real camera optics.

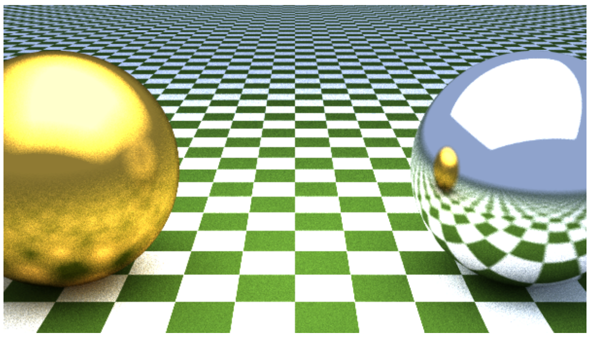
Figure 11: Metal Material

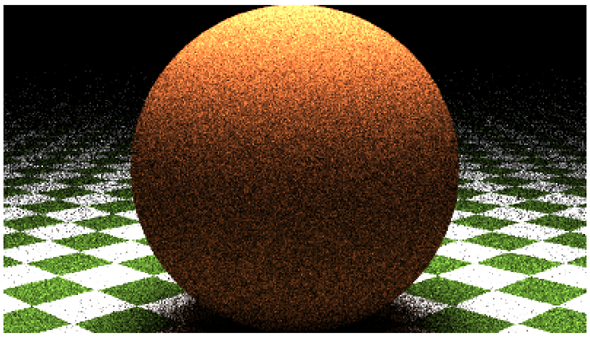
Figure 12: Diffuse Material

3 Third Party Resources

To achieve the results that this ray tracer produced, we required the use of these external resources:

1. stb_image.h from `https://github.com/nothings/stb/blob/master/stb_image.h`
2. tiny_obj_loader.h from `https://github.com/tinyobjloader/tinyobjloader`
3. Inspiration from https://raytracing.github.io/, both *Ray Tracing in One Weekend* and *Ray Tracing the Next Week*

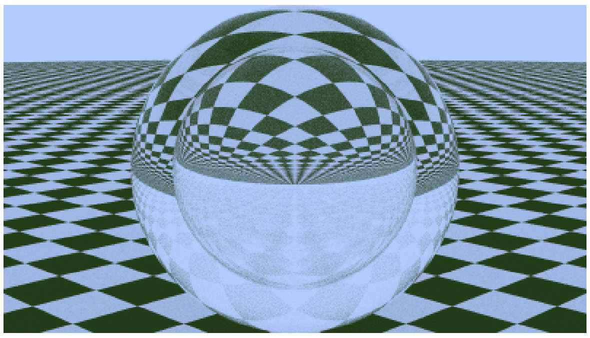
Figure 13: Dielectric Material

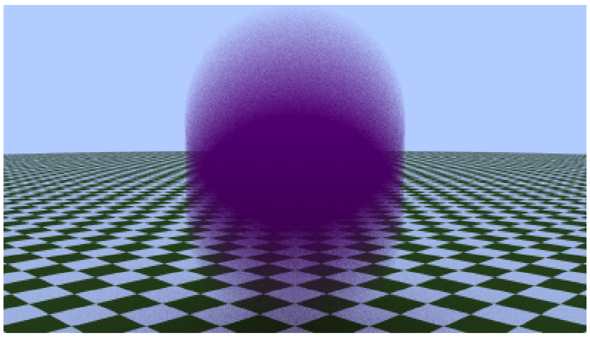
Figure 14: Bouncing ball

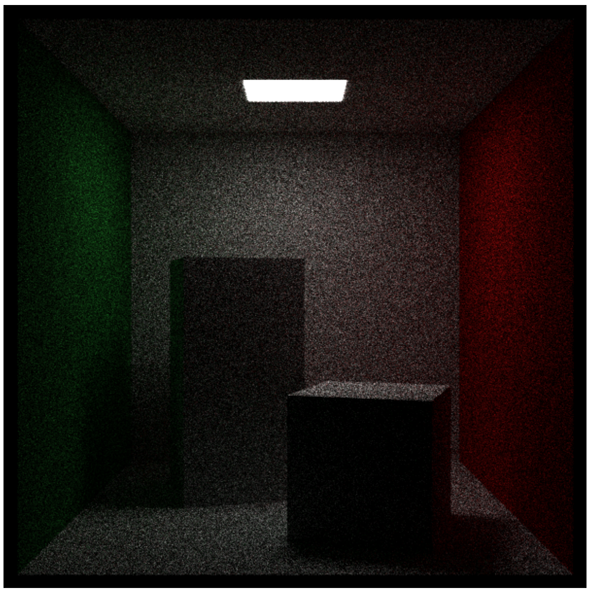
Figure 15: Cornell Box

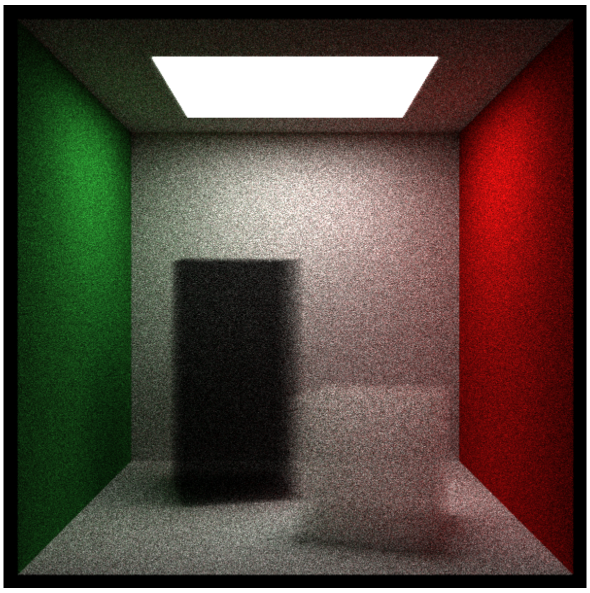
Figure 16: Volume Cornell Box

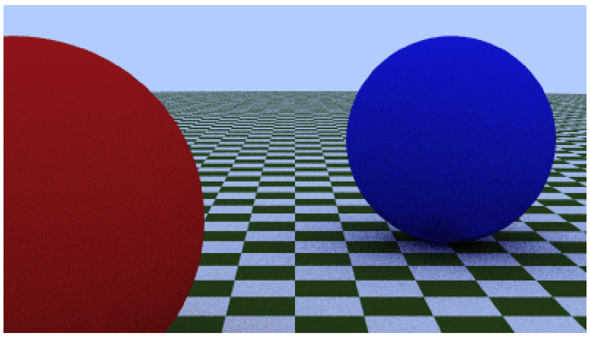
Figure 17: Without DOF

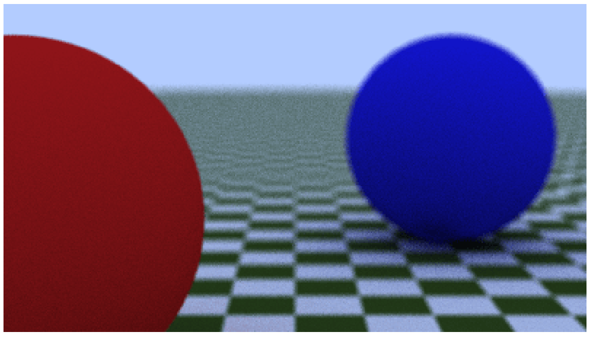
Figure 18: With DOF