# Ray Tracer

Camden Senneff

December 14, 2025

Figure 1: The final raytraced image, House MD's office

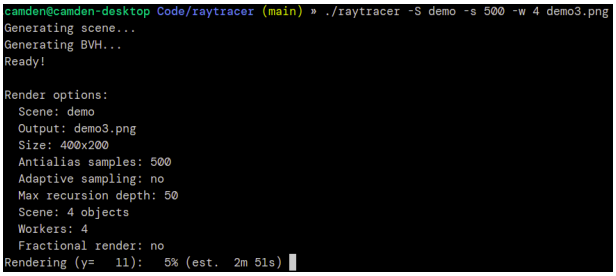


Figure 2: Terminal interface while mid-render

## 1 Introduction

This document outlines the features (visually and technically) that are implemented in my raytracer.

The code for the raytracer is hosted on GitHub at `https://github.com/humeman/raytracer`. Instructions on compiling and running it are in the `README.md` file.

The project uses algorithms (with some modifications) from the Ray Tracing in a Weekend series and other papers. Each section will contain citations of the algorithms used to implement them after the section heading in subscript.

### 1.1 Technical Overview

The ray tracer is implemented in C++ and is operated via the commandline. It's set up to compile with C++17 using g++. When run, it outputs a single PNG or PPM file to a default path, which can be overridden. A bunch of commandline flags are available to modify the parameters of the resulting image, primarily in processing power available to the raytracer and the resulting image quality.

### 1.2 Feature List

Each of these are described in more detail in the remainder of this report.

- Configurable camera (position, orientation, FOV)
- PNG and PPM read/write
- Anti-aliasing
- Spheres, triangles, and quadrilaterals
- Specular, diffuse, dielectric, emissive, and volume (ie, smoke) materials
- Color and image texturing on all shapes
- Triangle meshes
- Bounded Volume Hierarchy spacial subdivision

- Motion blur
- Depth of field (defocus blur)
- Importance sampling
- Adaptive sampling
- Multithreading and cross-machine parallelization

# 2 Features

## 2.1 Camera

Algorithm references: 1

The camera is the primary component of the ray tracer. It casts rays pixel-by-pixel into the scene, getting the color of the ray after it bounces around, to make an image.

The camera accepts an image size along with an origin (eye) location, orientation, and field of view. To fill in the image, it iterates over each pixel, casts a ray (or many rays) for each pixel into the scene at the angle controlled by the FOV, and writes the resulting color (or average color) to the image.

To get the color, rays 'bounce' around the scene. The ray color algorithm checks the ray against every object in the scene for a hit. If a hit is detected, it gets a direction to bounce in and an attenuation color back from the object (depending on its shape and material). The algorithm is then called recursively with the new direction, and the resulting colors are multiplied into one overall pixel color. The recursion repeats until the ray bounces too many times, or until it doesn't hit anything (in which case the scene's configured background color gets returned).

The camera also employs anti-aliasing. This sends many rays (with a small random offset) into each pixel, averaging the color returned between each ray and setting that on the image instead. This allows it to generate more realistic images, since with just one ray per pixel there are often many jagged edges along shapes. With anti-aliasing, it looks a lot smoother.

The camera also has depth of field, or defocus blur. This sets a point of focus somewhere in front of where the camera is looking and a defocus angle that applies a blur to areas of the image that are far away from that point. This makes the image look more like what the human eye would see, where objects that are not in focus look blurry.

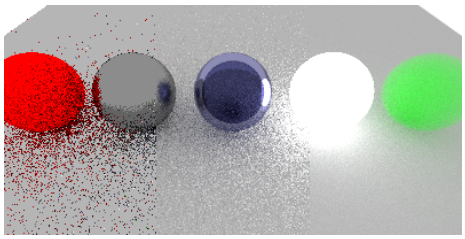

Figure 3: 1 vs 10 vs 250 anti-alias samples

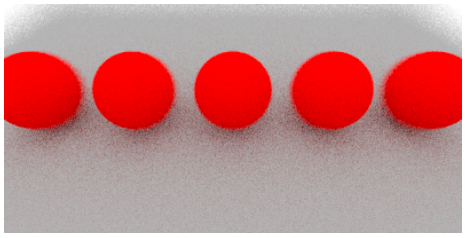

Figure 4: Defocus blur

The camera also allows for shapes to have motion blur with a time parameter that the camera passes along when casting rays. A shape that uses motion blur will change its location depending on that time field. In this project, spheres implement this to allow for a bouncy ball effect.

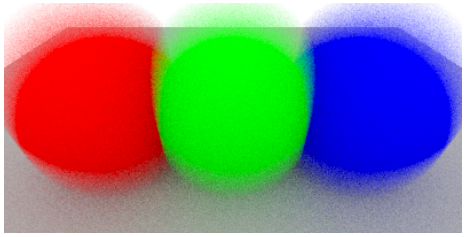

Figure 5: Bouncy balls

## 2.2 Shapes

Algorithm references: 1, 2

The ray tracer implements three basic shapes: spheres, triangles, and quadrilaterals.

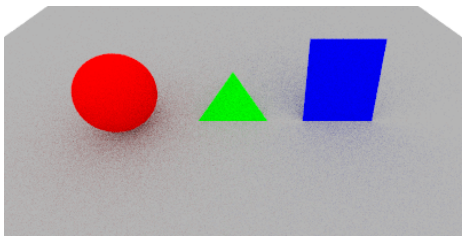

Figure 6: Three basic shapes

Spheres are implemented by solving the sphere equation, based on a center point and a radius. One or two solutions to the quadratic equation indicates a hit, and none means the ray never intersects the sphere. Triangles are implemented using the algorithm described in Tomas Möller and Ben Trumbore (4)'s paper, "Fast Minimum Storage Ray-Triangle Intersection." Quadrilaterals are implemented by extending a plane along the shape and testing if the intersection of the ray along that plane is within the boundary of the shape.

It also implements constant density mediums, which wrap any of the three primitive shapes along with a density parameter to produce a shape that bounces only some rays (with a probability based on the density), resulting in a smoke look.

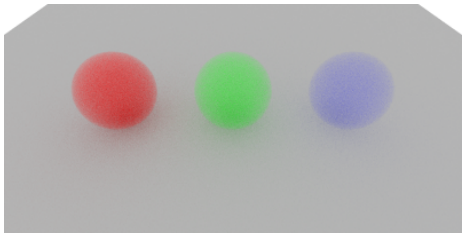

Figure 7: CDMs of varying densities

## 2.3 Textures

Algorithm references: 1, 2

Textures are implemented to modify the color that a shape returns. Specifically, solid color textures and image textures are implemented, and can be applied to any shape.

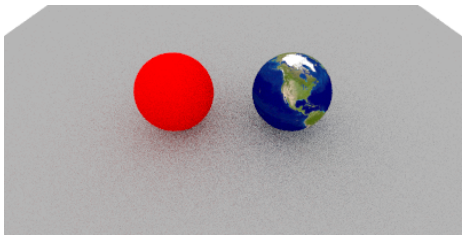

Figure 8: Solid color and image textures

Solid color textures always return the same color, independent of the input parameters. Image textures take in any PNG or PPM image (using the same models that the image writing portion of the tracer uses), and are sampled when hitting a shape with two parameters, $u$ and $v$. These each are numbers between 0 and 1 reflecting the location on the texture to grab the color of. How the shape maps $u$ and $v$ differs by shape (ie, triangles allow picking the u and v for each corner and interpolate between them, quads cover the whole shape, and spheres stretch the image around).

## 2.4 Materials

Algorithm references: 1, 2

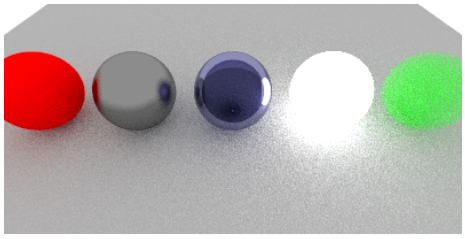

Figure 9: All 5 materials

The raytracer implements five materials:

- **Diffuse** (Lambertian): regular solid color reflection, good to represent a solid non-reflective object.
- **Metal**: mirrored reflection, good for a shiny metal or a mirror.
- **Dielectric**: a transparent reflection with some refraction. Good for representing glass or water.
- **Diffuse Light**: solid objects that have the ability to emit light.
- **Isotropic**: for use with a constant density medium, reflecting in a random direction.

## 2.5 3D Models

The ray tracer supports loading in 3D models in most 3D model file formats and displaying them on the scene with any location, scale, and orientation.

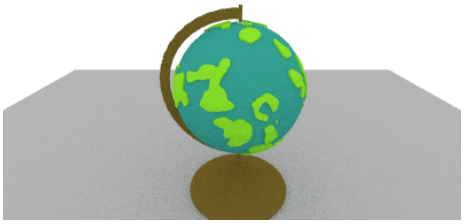

Figure 10: Globe, loaded via a .glb file

Behind the scenes, this feature uses the ASSIMP library to read the object file, and iterates over each triangle in the object mesh. It turns the coordinates into a vector, then performs translation, scaling and rotation. It also gets the color from the object (in a fairly primitive way – it does not support objects with image textures, just solid colors, and anything that uses image textures appears in solid black). Then, each triangle is added to the scene.

## 2.6 Optimizations

Algorithm references: 1, 3, 5

The ray tracer employs a few optimizations to reduce the number of calculations necessary to get a pixel's color.

The first (and most effective) of these is the bounding volume hierarchy. The scene's objects are divided into bounding boxes that include the entire shape, and those bounding boxes are in turn surrounded by their own bounding boxes, all the way to one parent bounding box. When casting a ray, it is much less expensive to check if it hits the larger, rectangular bounding box, than it is to compare against each individual shape. We know that if the bounding box is not hit, then we don't have to check any of the shapes in that bounding box for a hit, which gets rid of a lot of unnecessary work.

Another optimization, adaptive sampling, reduces the number of anti-aliasing samples that need to be taken on certain pixels. If enabled, after a certain number of rays are cast, the standard deviation of the rays cast so far is calculated, and that is used to calculate a confidence interval. If the results so far are within that confidence interval (set with a config flag), meaning the pixels returned so far are close in color, the color is returned as-is and no more anti-aliasing samples are taken. If the colors are not close, iteration continues until they are or until the anti-aliasing sample size is reached. This cuts out quite a few ray casts with a minimal impact on image quality.

Importance sampling is also implemented. This uses a probability distribution function (applied on selective shapes) to bias the direction rays go towards light more often than other locations, given a list of lights in the scene. This is implemented for Quad shapes, since that is my chosen light source in each scene. The result is a brighter and better looking image with far fewer samples.

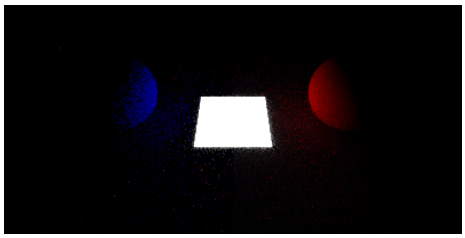

Figure 11: Left half without importance sampling, right half with

### 2.7 Parallelization

Ray tracing can easily be parallelized because the color of one pixel doesn't depend on the color of another: each one can be calculated independently. This raytracer implements a few controls to allow parallelizing the generation of a single image.

The first of these is multithreading. The ray tracer is set up to divide work by image rows. Before starting, it prepares a queue with a list of each y-coordinate that needs to be processed. It then spins up as many 'worker' threads as configured (with a default of 4). These worker threads iterate until there's nothing left in the queue:

- Pop the next row off of the queue
- Iterate over every pixel in the row
- Run the ray color algorithm against that pixel
- Write the color to the shared image object

This allows the ray tracer to easily use an entire CPU, rather than just one thread, to process the image as quickly as possible, providing a very significant speedup.

In addition, the ray tracer supports working on one fraction of a larger image, then joining parts of images into one. This allows for parallelization across multiple machines by having each machine work on one of the fractions of the image, then send each result to one machine where they're joined together (using the `joiner` program).

My practical use case of this feature is in the `rayblaster.sh` script. It accepts a list of hosts that are available over SSH, and:

- Builds the ray tracer and copies the binary to each remote machine
- Spins up a tmux session on each machine and calls the raytracer (with the fractional rendering flags set)
- Waits for each file to appear, and copies them to the host machine over SCP
- Joins each part of the image into one

This allows for an additional speedup factor over just multithreading.

## 3 References


1. Ray Tracing in One Weekend. Peter Shirley, Trevor D. Black, Steve Hollasch. `https://raytracing.github.io/books/RayTracingInOneWeekend.html`
2. Ray Tracing: The Next Week. Peter Shirley, Trevor D. Black, Steve Hollasch. `https://raytracing.github.io/books/RayTracingTheNextWeek.html`
3. Ray Tracing: The Rest of Your Life. Peter Shirley, Trevor D. Black, Steve Hollasch. `https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html`
4. Fast Minimum Storage Ray-Triangle Intersection. Tomas Möller, Ben Trumbore. `https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf`
5. Adaptive Sampling. Berkeley CS184/284A. `https://cs184.eecs.berkeley.edu/sp21/docs/proj3-1-part-5`