# Rendering Photorealistic Computer Graphics Using Ray-Tracing

Zach Clayburn

April 18, 2022

**Abstract**

Modern computer generated imagery is predominantly created using a family of algorithms called ray-tracing. Unlike traditional rasterization rendering, these algorithms simulate rays of light as they travel through a scene and are captured by a virtual camera. This paper implements a basic backward ray-tracing renderer, as well as analyzes its runtime complexity is discussed. Examples of output images are given, as well as discussion of the various input parameters for the algorithm.

## Introduction

In the past few decades that they have existed, computer graphics have evolved and improved far beyond what was initially possible. Where it once was an accomplishment to create a simplistic image that roughly approximated the real world, computers can now create imagery that is near indistinguishable from reality. Improvements in both the algorithms used and the hardware running those algorithms have made these improvements possible. A key example of one of these improvements is the introduction of ray-tracing based rendering algorithms.

Early computer graphics used a process called rasterization, where the color of each pixel is determined by computing the color of the object in front of each pixel, and then adding more effects through post-processing steps called shading. This method can be quite effective and much effort has been put into optimizing and improving this technique. Ray-tracing, however, takes a fundamentally different approach, and actually simulates individual rays of light as they travel through the scene being rendered. This results in images that much more closely approximate real photography. While this technique can be conceptually simple to understand and implement, it is much more computationally expensive than simple raster based rendering algorithms.

## Motivation

This paper discusses the basics of ray-tracing, and goes over my implementation of a simple ray-tracing based renderer. I selected this topic because I have always been interested in computer graphics. From video games to animated films, the topic has always been fascinating to me. In the past, I have experimented with traditional raster based rendering libraries (e.g. OpenGL, Vulkan), and I have a fairly decent working knowledge of how those processes work. I have not, however, had any experience with ray-tracing based rendering, and wanted to learn more about how it works. I had heard of an excellent book available online that walks through the process of writing a ray-tracing rendering engine [1], and given the opportunity to study and algorithm, I decided this would be a great opportunity to learn more about this process.

## Background

As mentioned previous, 3D computer graphics used to be created using rasterization and shading. This process involves computing each pixel color in several shading passes, adding more detail with each pass. The base pass determines the base color of each pixel, and then effects like lighting or motion blur are added with each subsequent pass. The advantage to this rendering style is that it can be done incredibly fast, with highly efficient algorithms and purpose built hardware able to massively parallelize the process. Because of this speed, this technique is still used in most real-time applications (e.g. video games), where images must be produced tens to hundreds of times every second.

When not working under such strict time constraints, rendering is usually done using a ray-tracing based algorithm. Because the process involves simulating light similar to how it behaves in the real world, the images produced have a much higher fidelity. Most computer generated imagery used in movies, like special effects or fully animated movies, are created using a ray-tracing based renderer.

Up to this point ray-tracing has been treated as a singular rendering technique. In reality, it describes a category of algorithms that all function similarly. To illustrate this, the following is a (non-exhaustive) list of ray tracing [2] techniques:

- **Forward:** In this method, light rays are simulated emitting from a light source, and a virtual camera records when those light rays collide with it. The resultant image is generated from the color and position of those rays, just as they would be in a physical camera. This method is the most accurate, as it is a direct simulation of reality, however, a major drawback is that many computations go to waste, as not all rays emitted actually end up contributing to the image.
- **Backward:** In this method, rays are simulated backwards in time. Rays are emitted from the virtual camera

at points corresponding to each pixel, and cast outward to find what object is visible. This method is similar to rasterization, with the difference being that after colliding with the principal object, the ray is simulated traveling around the scene, similar to how as a real photon would. While this is more efficient, in some cases it is less accurate than the forward method.
- **Hybrid:** Hybrid methods are an attempt to compromise speed and accuracy, and involve performing some forward ray-tracing on a scene, and then storing the data produced. A backward ray-tracing is then performed, using the stored data from the forward step to compute the color.

## IMPLEMENTATION

For my implementation, I went with a backward ray tracing algorithm. As can be seen in algorithm 1, the actual render loop is not overly complicated. The image is iterated over pixel by pixel, and a set number of samples are taken for each pixel. To take a sample, a random number is added to the x and y components, which are then normalized by the image dimensions. The reason multiple samples are taken for each pixel is to provide anti-aliasing (smoothing the transition between colors). Additionally, because some ray computations are stochastic in nature, taking multiple samples help to reduce noise by decreasing the visual weight of outliers in those random computations.

The process for computing the actual color, what takes place in the GETCOLOR function in algorithm 2, is where the similarity to rasterization ends. The ray is checked against all the objects in the scene, to see if it intersects any of them. If no objects are struck, then it is assumed that the ray that came from the global light source (e.g. the sun), and so it is colored according to the scene's global illumination. If an object or objects are impacted then the object nearest to the ray source is selected. Then that object's material is used to "scatter" the ray, which means to bounce the ray off of the surface in a material dependent way. Then, the function is recursively called with that scattered ray to determine where that ray would have come from and what color it would be. Because this is a recursive function, there is the possibility of a ray getting trapped and taking an indeterminately long time to reach a light source. To prevent wasting time on a ray that has no chance of contributing significantly to the image, or even to get stuck in an infinite recursion, a max depth parameter is used to limit how deep the recursion will go. If that maximum is met, the ray is assumed to have no light, and the source pixel is colored black.

My implementation has two types of materials: lambertian and metal. Lamberitan materials have a diffuse surface, meaning they scatter light evenly in all directions while removing a portion of the energy from that light, which leads to a flat, non-reflective surface appearance. The other material, metal, reflects light beams in a deterministic method, resulting in reflections showing up in the surface of the object. Metal materials also have another property, roughness, that adds a small amount of randomness to the reflection, which results in a surface looking rough, and similar to a lambertian material. This property varies from 0 (completely smooth), to 1 (completely rough).

## ANALYSIS

While Ray tracing is a stochastic algorithm, some assumptions can be made to bound in the time complexity. First, the main Render function loops over each pixel in the image, which requires $W \times H$ iterations where $W$ is the width of the output image, and $H$ is the height of the output image. Next, the algorithm takes $S$ samples by calling the RAYCOLOR function. This function is recursive, but because of the depth constraint, it can be treated like a loop the runs $D$ times, where $D$ is the parameter of maximum recursion depth. This is an approximation, and depending on the scene geometry and the location of the ray, this function could be called any amount $\leq D$ times. From there, if all the computations involved in scattering are assumed to be a constant operation (which is true in my implementation), they can be used as our unit of cost, resulting in an overall time complex of $O(WHSD)$.

## RESULTS

There are several interesting parameters that can be modified to get varying results out of the renderer. This

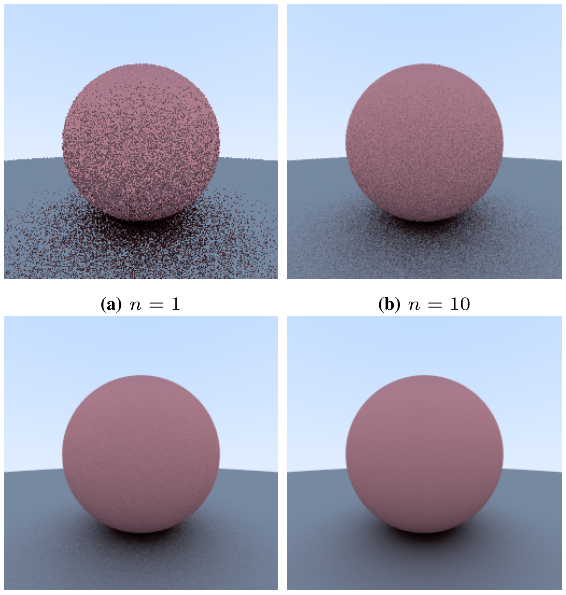
**Algorithm 1** Backward Ray Tracing Algorithm

```
1: procedure RENDER($imageWidth, imageHeight,$
        $scene, samplesPerPixel, maxRecursion$)
2:     $camera \leftarrow$ GETCAMERA($imageWidth, imageHeight, scene$)
3:     $image \leftarrow$ CREATEIMAGE($imageWidth, imageHeight$)
4:     for $x \leftarrow 0 \ldots imageWidth$ do
5:         for $y \leftarrow 0 \ldots imageHeight$ do
6:             $color \leftarrow (0, 0, 0)$
7:             for $0 \ldots samplesPerPixel$ do
8:                 $u \leftarrow (x +$ RANDOM$(-0.5, 0.5))$
9:                 $v \leftarrow (y +$ RANDOM$(-0.5, 0.5))$
10:                $ray \leftarrow camera.$GETRAY$(u, v)$
11:                $color \leftarrow color +$
           RAYCOLOR($ray, scene, maxRecursion$)
12:            end for
13:            $image.$AT$(x, y) \leftarrow color/samplesPerPixel$
14:        end for
15:    end for
16: end procedure
```

**Algorithm 2** Procedure for Computing ray color

```
1: procedure GETCOLOR($ray, scene, depth$)
2:     if $depth = 0$ then
3:         return $(0, 0, 0)$
4:     end if
5:     $hitRecord \leftarrow scene.$COMPUTEHIT$(ray)$
6:     if $hitRecord.hit$ then
7:         $object \leftarrow hitRecord.nearest$
8:         $material \leftarrow object.material$
9:         $scattered \leftarrow material.$SCATTER$(hitRecord)$
10:        if $scattered.good$ then
11:            $attenuation \leftarrow scattered.attenuation$
12:            $newRay \leftarrow scattered.ray$
13:            return $attenuation *$ RAYCOLOR$(ray, scene, depth - 1)$
14:        else
15:            return $(0, 0, 0)$
16:        end if
17:    else
18:        return GETGLOBALILUMINATIONCOLOR$(ray)$
19:    end if
20: end procedure
```

**Fig. 1:** The effect of sample count on image quality. Each image is rendered with $n$ samples per pixel.



**(a)** $n = 1$ **(b)** $n = 10$ **(c)** $n = 100$ **(d)** $n = 1000$

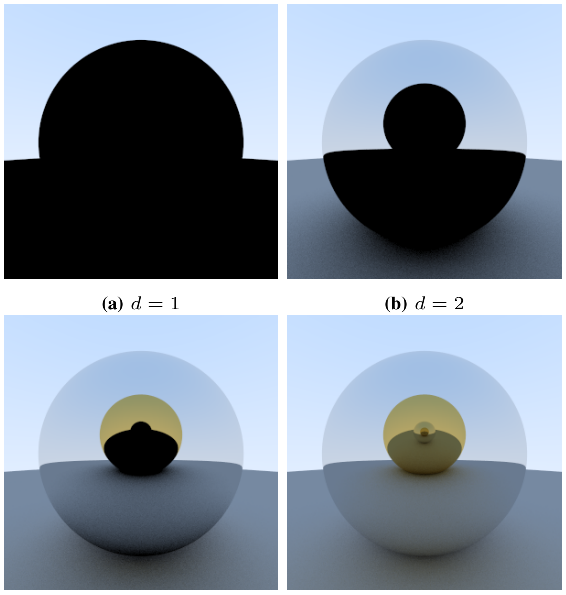
section will describe these changes, and include figures to demonstrate different values. Some parameters, like image size, do have an impact on the runtime, but will not be covered here, as their effect on the output can be assumed.

The first significant parameter is the number of samples taken for each pixel rendered. As mentioned previously, the ray-tracing process is a stochastic method, and the results vary between each calculation. Because this variation, pixels that are adjacent to each other can be vastly different colors, simply because of variations in the random numbers involved. This results in noisy images. As can be seen in fig. 1, by taking more samples and averaging the results, image noise is greatly reduced.

The next parameter of interest is the maximum depth of recursion. As fig. 2 demonstrates, if a maximum depth of one is used, only light emitting objects are drawn, as no reflections are computed. As the recursion depth increases, then the number of reflections also increases. While the appearance of deeper reflections in metallic surfaces is the obvious outcome, an additional outcome is more accurate shadows. As can be seen in fig. 2d, the shadows underneath the two spheres become much lighter, and the ground under the yellow sphere is tinted yellow by the reflections off of the sphere above it.

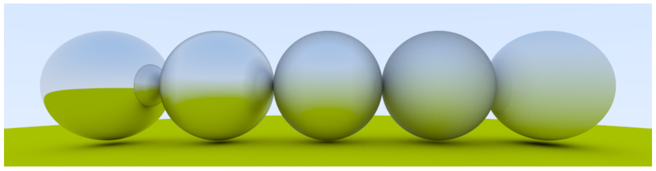
The last parameter to discuss is the roughness parameter of a metallic material. The image in fig. 3 shows the transition from completely smooth to completely rough. It can also be seen that a fully rough metal material is still behaves differently than a lambertian type material. This difference is caused by rough metals' still maintaining directionality, while the lambertian material's scattering is completely uniform.

**Fig. 2:** The effect of recursion depth on image quality. This is an image of two metallic spheres, immediately in front and behind the camera, rendered with $d$ max recursion depth. Note that, because the camera has no ray collision logic, it remains invisible.



**(a)** $d = 1$ **(b)** $d = 2$ **(c)** $d = 3$ **(d)** $d = 60$

**Fig. 3:** Four metallic spheres with roughness values (from left to right) of $0$, $1/3$, $2/3$, and $1$, as well as a lambertian sphere for comparison.



## Conclusion

Ray-tracing based rendering algorithms while, extremely computationally expensive, produce strikingly realistic imagery. By using stochastic methods to simulate light as it behaves in real photography, details that would take meticulous care and on the part of an artist to replicate are generated automatically by the algorithm. This detail comes with the cost of incredibly long run times, but the results have justified those costs for every field where it is feasible to use.

## References

[1] P. Shirley. "Ray tracing in one weekend." (Dec. 2020), [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[2] C. Lu, A. Roetter, and A. Schultz, *Types of ray tracing*, 1997. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/types.html.