Path Tracing Report

Michael Asper and Daniel Mancia

May 2019

1 Path tracing

1.1 Introduction

For our final project we decided to extend our ray tracer by implementing a path tracer. A path tracer essentially works by approximating the rendering equation using monte carlo integration. This allows us to get effects such as soft shadows and color bleed that wouldn't be present under our original recursive ray tracer. In this report we will first provide all the necessary mathematics and theory required to understand path tracing and then follow that up with the implementation details.

1.2 Rendering Equation

The rendering equation can be written as:

$$L_{out}(\hat{w}_r) = \int_{\hat{w}_i \in hemisphere} f_r(\hat{w}_r, \hat{w}_i) L_{in}(\hat{w}_i) \hat{w}_i \cdot \hat{n}$$

where f_r is the BRDF or "Bidirectional Reflectance Distribution Function" which encodes the properties of the material. The high level idea of this integral is that the amount of light leaving a surface is dependent on how much light is coming in (which is why we are integrating over the unit hemisphere).

1.3 Integration

In order to evaluate the integral we use Monte Carlo integration. Monte Carlo integration is a non deterministic method of numerical integration that works by choosing uniformly random points on the area of integration. After sampling enough points we evaluate the function being integrated at those points, multiply it by the area of integration, and take the average. In our case we sample points uniformly from the unit hemisphere and take the average of the function evaluated at these points. We have to multiply by 2π since the surface area of a sphere is 4π. Our integral then becomes:

$$L_{out}(\hat{w}_r) \approx \frac{2\pi}{N} \sum_{j=1}^{N} f_r(\hat{w}_r, \hat{w}_i^j) L_{in}(\hat{w}_i^j) \hat{w}_i^j \cdot \hat{n}$$

2 Implementation

For our implementation, we created a sampling process that every ray has to go through. It chooses a random point on a uniform hemisphere to follow instead of directly following the ray. This sort of implementation was incredibly slow for possibly two reasons: entropy and single-threaded.

2.1 Entropy

The current implementation uses a Mersenne Twister, which requires a lot of entropy. A better implementation would be using Xorshift[1], which is regarded as the fastest pseudo random number generator.

[1] https://en.wikipedia.org/wiki/Xorshift

2.2 Single-threaded

Initially, our ray tracing was single-threaded which was fine for the use case, especially considering our performance was exceptionally good; however that wasn't the case for path tracing. Fortunately, due the independent nature of rays, we were able to split up the render into a lot of smaller threads using pthreads.

3 Results

The results show as we allow more samples, the image is generally a lot more sharper. Going from left to right, we increase the amount of path tracing samples. There is slight bleeding of green on the back wall, which happens from our hemisphere sampling.

The scenes are also rendered with an area light we fixed from milestone 2. The light samples increase from top to bottom. Generally as light samples increase, you can see more light makes it to the camera.

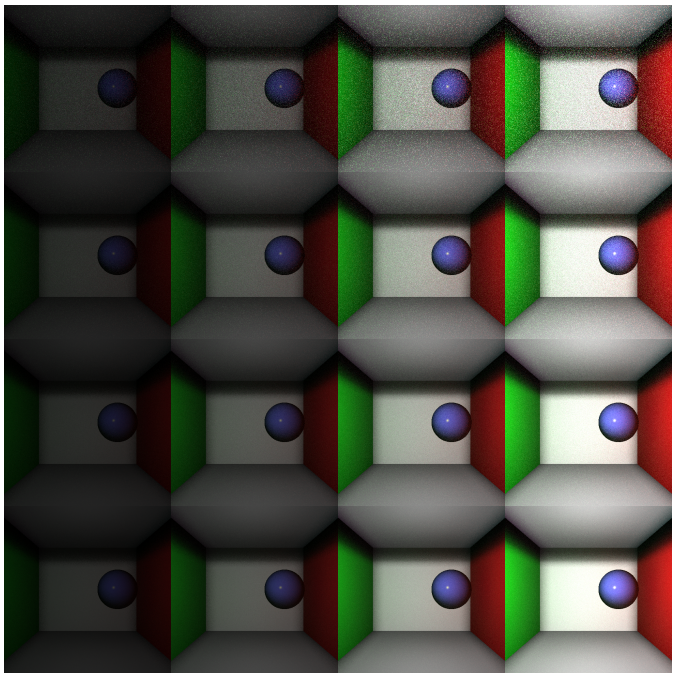

Figure 1: Path tracing results.

References

[1] *Lecture 10 - Global Illumination*