Hannes Bennet - hbenn@kth.se
Mattias Larsson - matlar4@kth.se
Fredrik Lundkvist - flundkvi@kth.se

Project report - Path Tracing

Figure 1: Image rendered using the final version of our path tracer. 32768 samples per pixel at a resolution of 300x300 pixels, and 3 bounces per ray.

Introduction

In computer graphics, ray tracing is a technique to project a three-dimensional scene on to a two-dimensional plane. In other words: how to render three-dimensional scenes to a computer screen or image file. In ray tracing, a ray is projected through each pixel in the image plane and as the ray intersects objects within the scene, various calculations determine the color of the pixels. This process is performed for each pixel of the image and thus, a ray traced image is created [1].

An alternative to standard ray tracing is *path tracing* which, while being more computationally demanding, results in more realistic results than normal ray tracing. One example of this is that images rendered with path tracing have more realistic global illumination than images rendered with standard path tracers. It is commonly used within fields where realism is essential, and render times are not a problem, such as movies or architectural visualization. Examples of renderers that use path tracing are Disney’s hyperion renderer, and Pixar's renderman.

In this project, we implemented a simple path tracer in C++, using code from a ray tracing lab as our basis. To do this, we will implement emissive properties for materials, uniform sampling of unit vectors in a hemisphere, and multi-threaded rendering. The goal is to render images like the image below:

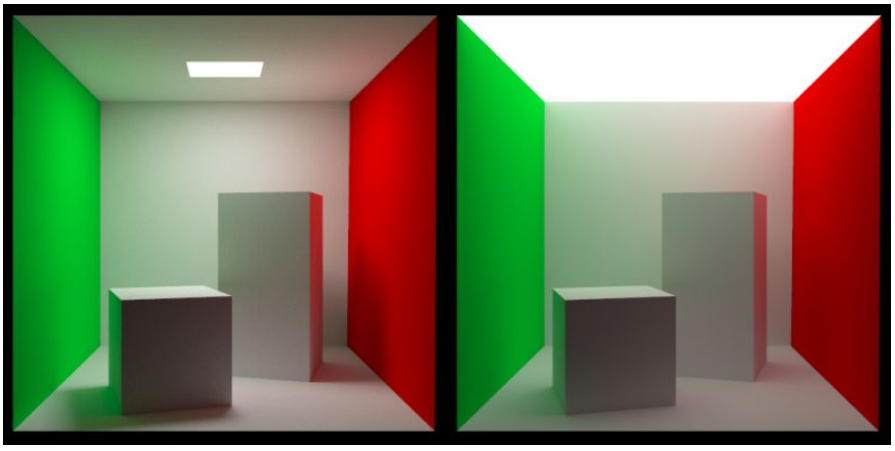

Figure 2: The Cornell box rendered with path tracing. Image credit: demofox.org

Theory

The rendering equation, presented by James Kajiya, is the foundation of path tracing. It is an equation to calculate the emitted light from any point in a scene. All physically-based rendering algorithms render images by solving this equation in one way or another. The full equation is presented below:

$$L(\mathbf{x}, \omega_0, \lambda, t) = L_e(\mathbf{x}, \omega_0, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_0, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) d\omega_i$$

This equation is very expensive to calculate, meaning that it is not really feasible to use in practice. Most physically based renderers use some alternate method to approximate the answer to the rendering equation. In the case of path tracing, we make use of the concept of *monte carlo integration*.

The general idea behind monte carlo integration is to randomly sample random values in the result domain of the function we want to approximate, averaging them out. Given enough samples, the result will converge towards the correct solution that we want to find. By doing this, we can treat the rendering equation as a discrete sum instead of a continuous integral, which simplifies the computations significantly. Thus, monte carlo integration allows us to find a very good approximation to the rendering equation, much faster than solving the equation in itself. Using these methods, the light I emitted from a point in the scene then becomes:

$$I = \frac{1}{N} \sum_{i=1}^{N} \frac{L(X_i)}{pdf(X_i)}$$

Where N is the number of samples, pdf is the *probability distribution function* for X_i, and X_i is a random sample from the integration domain. In our project we perform uniform sampling, which means that I becomes the average of uniformly randomly selected rays, due to the probability function being the same for every sample. As the number of samples increases, the more realistic the image will become.

With this knowledge, we can establish the following pseudocode for the path tracing algorithm:

```
tracePath(ray,depth):
	If depth >= max_depth
		Return black
	findNearestObject(Ray)
	If ray did not hit something
		Return black
	Newray.origin = ray.HitPoint
	Newray.direction = randomUnitVectorInHemisphereOf(ray.HitPoint.normal)
	Emittance = thinghit.emittance

	Probability = 1 / 2 *π

	Cos_theta = dot(Newray.direction, ray.hitpoint.normal)
	BRDF = thinghit.reflectance / π

	Incoming_light = TracePath(newray, depth+1)
	Return Emittance + (BRDF * Incoming_light * Cos_theta / p)

Render(image, numSamples)
	for pixel in image
		for i in numSamples
			Ray = generateRay(pixel)
			pixel.color += TracePath(Ray, 0)
		Pixel.color /= numSamples
```

Implementation

We began by using code from a completed lab on ray tracing as our starting point, given the similarities between the two rendering techniques. Using the aforementioned pseudocode, we began creating the path tracer and we were quick to get some noticeable results.

Sampling functions

The functions for generating random sample vectors are crucial for a path tracer; the better they are, the better the result becomes. We decided to implement uniform sampling, which is what is used in the most basic path tracers. To do this, we generate two floating-point numbers between 0 and 1 from a uniform distribution: $X1, X2 \in U(0,1)$, these numbers are then used to create a vector in the unit hemisphere using the following equations:

$sin(\theta) = \sqrt{1 - X_1^2}$

$\phi = 2 * \pi * X_2$

$x = sin(\theta) * cos(\phi)$

$y = X_1$

$z = sin(\theta) * sin(\phi)$

Which is just a translation from spherical to cartesian coordinates.

This unit vector is then transformed to be in the hemisphere of the hit point normal based on the method found in [3].

Speed

Our naïevly implemented path tracer worked reasonably well, but required considerable amounts of time to render. A test image of 500 by 500 pixels, with three bounces and 25 samples per pixel took more than 15 minutes to render. Therefore, the next step in our project was to optimise our code to run faster. In this section, we will present our optimisations.

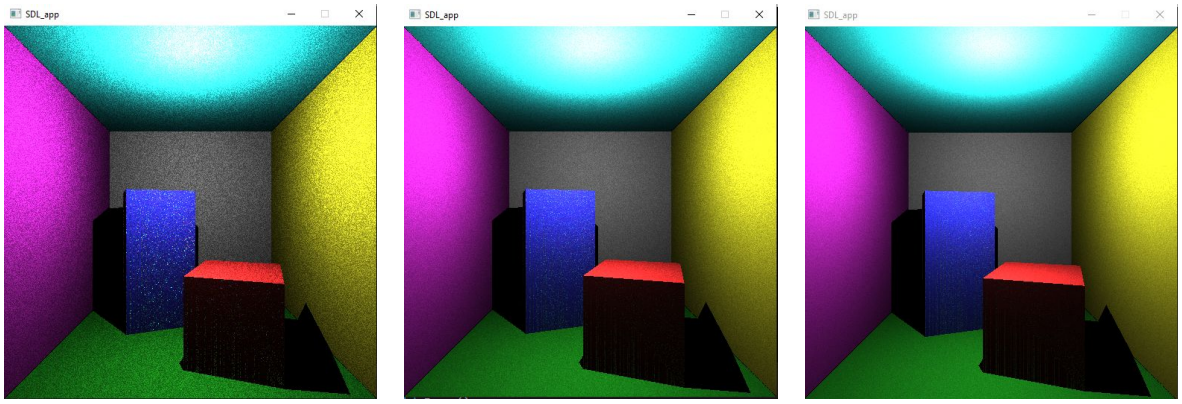

Figure 3: The first images from the path tracer. From left to right: 1 sample per pixel, <1 minute render time. 5 samples per pixel, ~4 minutes render time. 10 samples per pixel, >10 minutes render time.

The first thing we did to speed up rendering was making our draw function multi-threaded. Using 10 threads at the same time, and thus rendering 10 pixels in parallel, allowed us to reduce rendering times by a factor of 2. While multithreading sped up the program significantly, we were not entirely satisfied. Our next idea was to reduce the time spent testing for ray-triangle intersections. The end result of this was that we replaced the code that we had taken from the ray tracing lab, and implemented the Möller-Trumbore intersection algorithm[5] instead. This resulted in further speedups of the code, with the same test image (500x500px, 3 bounces, 25 samples) only taking 90 seconds to render, reducing rendering times to 1/10th of what they were before we started optimising our code.

Implementation challenges

We realised quite early that we had misunderstood how the sampling function was supposed to work. We implemented an emittance vector to each triangle, added a square light source in the ceiling and rewrote the TracePath function to implement this new functionality. Upon rendering at 100 samples per pixels the image below was created.

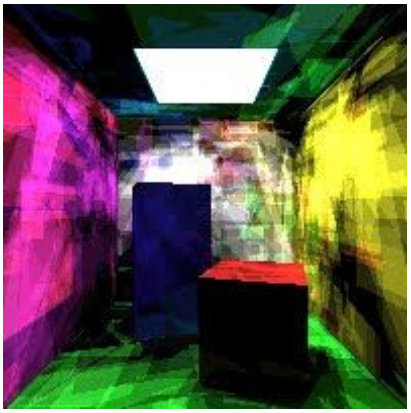

Figure 4: First render created with our altered path tracer. 100 samples per pixel.

We figured that these strange artefacts were caused by the lack of samples per pixel and upon increasing that value we created the image pictured below. While the image looked much better and had realistic lighting (the strange artefacts even adding an illusion of texture to the walls), it did not remind us of the noise which is commonly associated with path tracing.

Figure 5: 500 samples per pixel with an issue with our sampling function.

We concluded that the artefacts were effects of an issue with our implementation of the hemisphere sampling function. After rewriting this function we rendered the image below. The image still had the same realistic lighting but the strange shadow artefacts were gone, and we could once again see the noise associated with path tracing.

Figure 6: 1000 samples per pixel without the sampling issue.

Result

Some renderings from the final version of our program can be seen in the figure below:

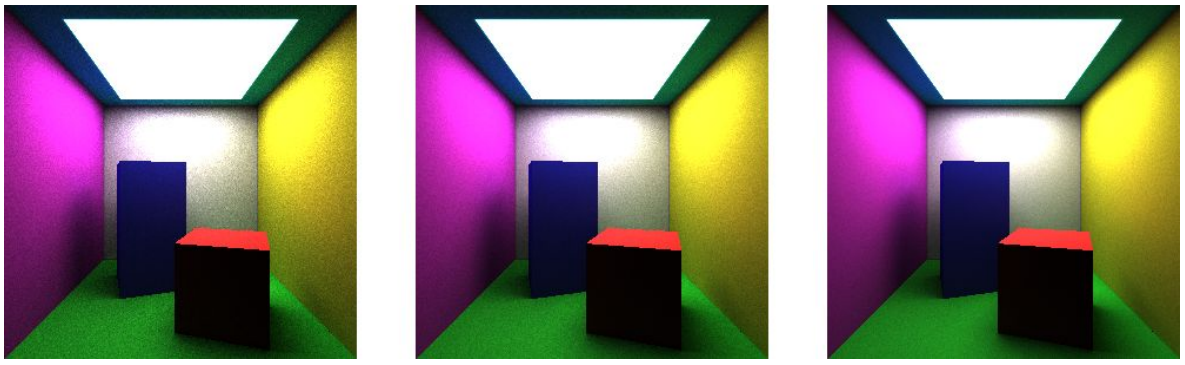

Figure 7: the cornell box rendered with our path tracer. From left to right: 2048 samples/pixel, 4096 samples/pixel, 8192 samples/pixel. As the amount of samples increase, the noise becomes less noticeable.

It is worth noting that the image is still relatively noisy, even when using 4000+ samples per pixel.

We can see realistic global illumination across the image, as well as color bleeding in the reflections of the roof. These effects can be quite hard to simulate with other rendering methods such as rasterization. Here, we have managed to achieve fairly realistic light transport with a relatively simple algorithm, albeit at the cost of rendering time.

The image seen below was rendered on a 6-core CPU, with a resolution of 300x300 pixels, 32768 samples per pixel, and a max ray depth of 3. The rendering time for the image was approximately 190 minutes.

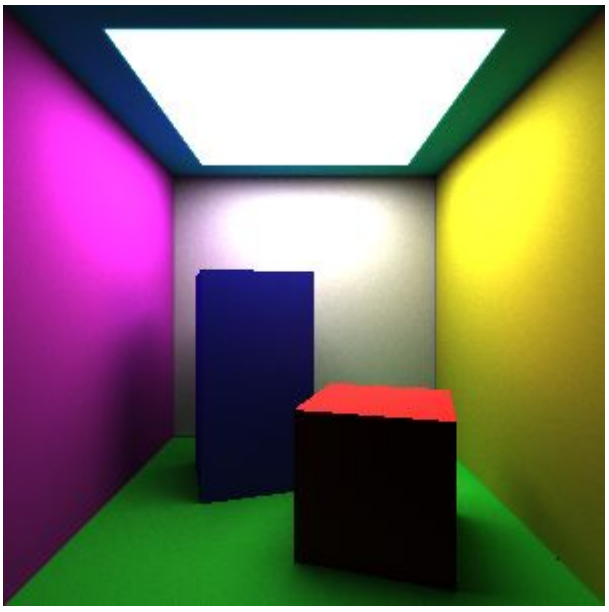

Figure 8: Final image of the path tracer.

Discussion

General

Our path tracer manages to produce realistic images, albeit after quite long render times. The basic version of the algorithm is relatively straightforward to implement, but understanding it fully and optimising it requires quite a lot of mathematical knowledge from different fields. In general, we are satisfied with our results, as the images look quite good to our eyes, and we managed to achieve realistic global illumination.

Method

Our finished product was made with ideal diffuse surfaces with the original intention of extending it with both ambient and specular properties. The implementation of various components such as materials and a sophisticated Phong illumination model would have improved the result with even more realistic results. Intentions of creating translucent materials and perhaps different object shapes such as spheres or any other objects with various properties, such as refraction, could have made the endresult more comprehensive and visually satisfying.

An improved sampling model would have led to faster convergence of the monte carlo integration towards the solution to the rendering equation. Since the convergence of naive monte carlo integration is proportional to $\frac{1}{\sqrt{N}}$, where N is the number of samples per pixel. Using an alternate sampling method, such as stratified sampling, or the cosine-weighted sampling described in [4], would probably increase our rate of convergence, meaning that less samples would be required to render images with relatively little noise.

The monte carlo method delivers high quality visualisation of the applied model but, as mentioned previously, it is a time consuming process which we experienced during the implementation and testing process. In order to optimise this, [6] suggests implementing various filtering methods to reduce the noise the monte carlo method integration produces, in order to minimise or at least reduce the time it takes to render. These filtering methods could definitely have contributed to our project in order to increase the rendering speed. Because of the time frame this project had, we did not prioritise this as implementation but rather focused on the visual part of the rendering instead of the speed it required.

During the project we documented our progress on this blog: https://dh2323path.blogspot.com/, where we discussed some aspects of our project more in-depth.

Perception studies

When it comes to rendering methods such as path tracing, the most natural perceptual study to conduct is, in our view, finding the point of diminishing returns; i.e trying to find a "threshold" for amount of sample relative to resolution where perceived improvements in the image start to drop off. This might not be the easiest research to conduct, as it might be hard to quantify what exactly is meant by improvements, but the results could be very valuable to the graphics research community. Knowing what amount of effort is required for "good enough" can potentially remove hours of unnecessary rendering, while also reducing power costs and consumption.

If we were to perform a perception study, we would gather enough participants for statistical relevance, have them look at various images rendered using path tracing. These images would be rendered using different amounts of samples per pixel, along with different levels of depth for each ray. The order of the images would be randomized so that the participants would not notice a pattern in the change in quality. We would create a survey which allows the participants to rate each image on a 7-point likert scale in order to gain qualitative data. The data could then be analyzed to find the lowest amount of samples where the rating does not significantly change after an increase in samples per pixel.

Project members contribution

Hannes worked on implementing the new light source in the scene, attempting to create new materials for various objects in the scene (which was eventually skipped because of other project priorities) and assisting with the implementation of the path tracer we used.

Mattias worked on implementing the pseudocode and merging it with the existing code from the previous lab. He also worked on general program structure and rendered many of the images with higher sample rates per pixel.

Fredrik worked on the monte carlo sampling functions, implemented the möller-trumbore algorithm, and multithreaded rendering.

All the group members contributed during the project sessions we had but also with the blog entries and this project report.

References

[1] Glassner, A. S. (Ed.). (1989). *An introduction to ray tracing*. Elsevier.

[2] Kajiya, J. T. (1986). "The rendering equation". *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. ACM

[3] "Global Illumination and Path Tracing (Global Illumination and Path Tracing: a Practical Implementation)", Scratchapixel.com, 2009-2016. [Online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing/global-illumination-path-tracing-practical-implementation. Accessed: 19- May- 2020.

[4] M. Pharr, W. Jakob and G. Humphreys, "2D Sampling with Multidimensional Transformations", *Pbr-book.org*, 2018. [Online]. Available at: http://www.pbr-book.org/3ed-2018/Monte_Carlo_Integration/2D_Sampling_with_Multidimensional_Transformations.html. Accessed: 19- May- 2020.

[5] Tomas Möller & Ben Trumbore (1997) Fast, Minimum Storage Ray-Triangle Intersection, Journal of Graphics Tools, 2:1, 21-28, DOI: 10.1080/10867651.1997.10487468.

[6] Suykens, Frank, & Willems, Yves. (2000). Adaptive Filtering for Progressive Monte Carlo Image Rendering. Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media '2000, 220-227.