Global Illumination Report TNCG15

Viktor Tholen - vikth187, Julius Halldan - julha230

October 19, 2021

1 Abstract

We present an implementation of a Monte Carlo ray tracer by using methodology from the course TNCG15 at Linköping University. The methodology contains creation of a simple scene with cubes and spheres with different surface properties inside a room similar to the Cornell Box. Light is simulated with a Monte Carlo ray tracer by implementing importance rays that are sent from a viewpoint through an image plane containing pixels. When surface intersections occur, a fixed number of new rays gets distributed in different directions of a hemi-sphere with a probability density function. Ray termination is done with a Russian Roulette algorithm. Intersection algorithms has been implemented for the different types of objects in the scene. The ray tracing process has also been parallelized for the CPU with performance scaling.

Contents

2 Introduction

Generating images from 3D-scenes is widely used within many fields and industries. The idea of ray tracing comes from the 16th century but was first implemented with the use of computers in the 1970s. One of the first movies that used the early ray tracing techniques was *Tron* in 1982. The early ray tracing techniques did not have any global illumination support. The rendering was instead based solely on local lighting models which created a rather unrealistic result. Global illumination is often faked to create an illusion of realistic lighting as global illumination is expensive and time consuming. Apart from movies, global illumination has recently been introduced to real-time video games. However, that approach is mostly focused on using specific hardware acceleration to calculate fast denoising of an undersampled result, to still be able to get a visually pleasing image with a high frame rate.

Global illumination is meant to add realistic lighting to 3D scenes. Global means that objects are illuminated with regard to other objects within the scene instead of only itself, which the local model was based on.

2.1 Rendering equation

The rendering equation was introduced in 1986 as a model describing the light transport which also follows the physical law of conservation of energy. The equation can be written in the following form:

$$L(x \leftarrow \omega) = L_e(x \leftarrow \omega) + \int_{\omega_1} f^*(x_1, -\omega, \omega_1) L(x_1 \leftarrow \omega_1) d\omega_1 \tag{1}$$

where $L(x\,,\omega)$ is the radiance that arrives at the point x from direction ω, $L_e(x \leftarrow \omega)$ is the radiance emitted by a light source that arrives at x from the direction ω, and $L(x_1\,,\omega_1)$ is the radiance that arrives at a point x_1 from ω_1. $f^*(x_1, -\omega, \omega_1)$ gives the fraction of the radiance that is reflected by the surface point x_1 from the direction ω_1 into the direction $-\omega$. [1]

The rendering equation can not be solved as a continuous method as computers require numerical solutions. Since 1986, several different realistic rendering methods has been developed with different approaches to solving the equation.

2.1.1 Radiosity

One approach to solving the equation is based on using finite elements, called patches, and iteratively gather information to solve the radiance equation. It is a simplification of the rendering equation, as the method only accounts for lambertian reflectors. Specular reflection can not be calculated with radiosity. The performance of the algorithm can be directly controlled with the number of patches and iterations used. Radiosity has great potential to be calculated

in parallel on a GPU, as the method can be fully expressed as a single matrix-vector multiplication. [3]

An example of a modern implementation of radiosity is presented by Shcherbakov et al. in *Dynamic radiosity*[3]. The method is capable of calculating the lighting with more importance around the observer. That is achieved by only updating patches in a set distance range from the observer in the scene. The method also supports dynamic objects, which might be interesting for real time applications like games.

2.1.2 Whitted ray tracing

Whitted's ray-tracing only considers glossy or diffuse reflection. This method assumes that all the objects are perfect reflectors and does not solve the entire rendering equation. This method casts rays from the camera through a pixel plane into the scene. Each ray has a direction ω_o, which is used to advect the ray into the scene.[1]

Whitted ray tracing relies on *importance* and *radiance*. *Importance* is a quantity that is emitted from the eye point. It is emitted towards the direction ω_o and represents the sampling that defines what information that should be sent to the pixel. *Radiance* is a quantity emitted from the light sources into the scene. It travels in the opposite direction of the importance. Shadow rays are also implemented in this method. When there is a ray intersection in the scene, a shadow ray will be drawn between the intersection point and the light sources in the scene. If there is an object in the way of the light path, the light does not contribute to the radiance of that intersection point.[1]

2.1.3 Monte Carlo ray tracing

The Monte Carlo methodology have been implemented in different areas for over 100 years. For instance, it was used for simulations when developing the atomic bomb during world war 2. Monte Carlo methods compute results with the help of random numbers. The methods can be seen as simulations of random processes, like light rays bouncing around in a scene. [2]

The versatility of Monte Carlo methods allows to solve problems without making big simplifications of the problem to be solved. That suits our needs, as the methods only sample the functions of the geometry and optical properties. The methods have a slow convergence, which will be visible in the results as noise. This will however be easily controlled by the amount of samples being calculated. The trade off for this is calculation time. [2]

The methods have great potential of being unbiased. That means that the discretizations made will converge to the same results as the original mathematical expressions. The nature of the Monte Carlo methods and their close relation to probability opens up opportunities to do that. [2]

The Monte Carlo method introduces indirect illumination. This method no longer assumes only perfect reflections, but also evenly reflect a new ray within a hemisphere of the hit surface point. The indirect illumination represents the global illumination which is an important ingredient for simulating realistic lighting in imagery. This feature allows for soft highlights and shadows. Without the indirect lighting, the result would be the same as a local lighting model. It is not possible to visualize caustics with the Monte Carlo method in an efficient way. The method can be complemented with photon mapping to generate such effects. [1]

A big part of the Monte Carlo ray tracer is its important sampling. Within the ray tracing theory it is commonly used for selecting where rays should be sent in a higher density, to highlight the precision of the sampling in certain parts of the scene. This adds great opportunities for optimisations. [2]

3 Background

Raytracing works by shooting importance rays from the eye point e_0 through each pixel of the camera plane. From these rays, it attempts to find the irradiance of that pixel which determines the final color. Each ray bounces multiple times within the scene and is then terminated, see section 3.6. This is a recursive process that ends when rays terminate, which returns the sample value. There are two types of light contributions, direct and indirect light, and both contribute to the final pixel color in their own way. [1]

Tracing one ray through each pixel will not be enough to approximate the rendering equation with enough precision. Thus, it tends to generate a lot of noise. However, this can be countered with supersampling by calculating the color of the pixel multiple times and thereafter dividing the color by the number of samples. The number of samples heavily affects the render time for the image, but also the accuracy. This is where a decision has to be made regarding the trade-off between time and accuracy. Each time a new sample is taken for the pixel, a random position within that pixel should be calculated with an unbiased probabilistic density function(PDF). Otherwise, artifact issues like aliasing and moire-patterns can appear in the render. Although there are other patterns to use instead of this unbiased random function. This method counters the artifact problem with great results but tends to generate more noise. [1]

The pixel color is stored as a double precision number in order to preserve the dynamic range of the render. The image will consist of integer values between 0-255 which requires a conversion from the sample space to the image space.

There are different types of color scales with different dynamic range. To more accurately resemble the dynamic range of the human eye a logarithmic gamma curve is applied before storing the integers. The conversion can be expressed as

$$f(x) = \sqrt{x} \tag{2}$$

where x is the input value.

3.1 The scene

The scene represent the objects in the scene. Apart from meshes and spheres, a room was created according to figure 1. This layout will be used to display the results and comparisons. All walls are Lambertian reflectors with the left wall colored red and the right wall green which more clearly shows the global illumination on other objects. The scene is illuminated with two area lights located on the roof.

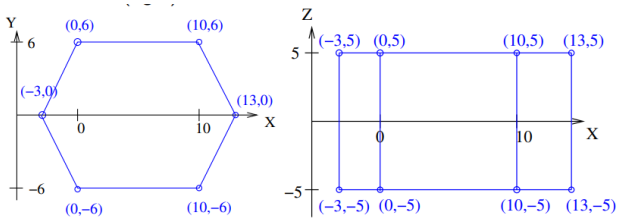

Figure 1: The room metrics

3.1.1 Vertex

The vertices represents the points in the scene. These are used to build the triangles that the meshes contains of. A vertex is described with four float precision values that represents the world coordinates for the point.

3.1.2 Triangle

The triangle consists of three vertices and its normal vector. The normal N of a triangle can be calculated by the following equation:

$$N = E_1 \times E_2 \tag{3}$$

where E_1 and E_2 are two of the three edges of the triangle.
The surface area A of the triangle will also be needed and can be calculated accordingly:

$$A = \sqrt{S \cdot (S - E_1) \cdot (S - E_2) \cdot (S - E_3)} \tag{4}$$

where

$$S = \frac{E_1 + E_2 + E_3}{2} \tag{5}$$

3.1.3 Mesh

The mesh consists of a list of triangles. These triangles represents the surface of the meshes and are used to create objects like cubes, tetrahedrons and a custom room layout.

3.1.4 Sphere

The sphere is calculated procedurally which means that it does not have a geometric representation of triangles. Instead, the surface of the sphere is related to its radius and center point. The equation for a sphere can be expressed as:

$$||x - c||^2 = r^2 \tag{6}$$

where x is a point on the spheres surface, c is the center coordinate and r is the radius.

3.2 Intersection algorithms

In order to trace rays through the scene, we need to know where a ray hits an object in the scene. This has to be efficient as every object and every triangle in the scene has to be checked for possible intersections in the worst case.

3.2.1 Mesh

The Möller Trumbore algorithm is used to find the intersection point of a triangle. The algorithm is fast for calculating the intersection between a ray and a triangle without needing any precalculation of the triangle plane.

A local coordinate system is created using the three vertices of the triangle. The barycentric coordinate system is used for determining a point on a triangle surface with two coordiantes (u, v) where $u \geq 0, v \geq 0$ and $u + v \leq 1$. The point on the triangle is given by:

$$T(u, v) = (1 - u - v)v_0 + uv_1 + vv_2 \tag{7}$$

and the ray is defined by:

$$x(t) = p_s + t(p_e - p_s) \tag{8}$$

the intersection of the two equations gives the following matrix solution:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{P \cdot E_1} \begin{pmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{pmatrix} \tag{9}$$

where $T = p_s - v_0$, $E_1 = v_1 - v_0$, $E_2 = v_2 - v_0$, $D = p_e - p_s$, $P = D \times E_2$ and $Q = T \times E_1$

By ensuring that u and v are greater than zero and $u + v \leq 1$ we know that the point on the triangle is defined. But it is also required to check if $t > 0$ as a value lesser than 0 will yield in the triangle being behind the camera. Although the ray technically hits the triangle, it should not be taken into account. If all of the mentioned conditions are true, the ray hits the triangle. [1]

3.2.2 Sphere

The intersection algorithm for a sphere needs to be approached a bit differently due to its procedural implementation. Since there is an analytical representation of the sphere, an analytical solution can also be used for the intersection points. When there is an intersection point, the point on the sphere and the point on the ray is in the same position. This can be utilized in the intersection algorithm. [1]

If a ray is defined by $x = o + dI$, where o is the starting point. d represents one point on the ray. The analytical definition of a circle, as in equation 6 can be combined with this to find intersection points. [1]

By substituting $||x - c||^2$ we get $(x = o + dI) \cdot (x = o + dI) = r^2 \iff d^2(I \cdot I) + 2d(I \cdot (o - c)) + (o - c) \cdot (o - c) - r^2 = 0$.

We substitute $a = (I \cdot I) = 1$, $b = 2I * (o - c)$ and $c = (o - c) \cdot (o - c) - r^2$.

We get $ad^2 + bd + c = 0 \iff d = -\frac{b}{2} \pm \sqrt{(\frac{b}{2})^2 - ac}$.

Mathematically there will be two intersection points, which makes sense since the sphere has two sides from the eyes perspective. The closest point to the camera is the point that we take into consideration.

3.3 Surface Properties

The radiance is calculated by considering the direction of the surface hit, the light direction etc. There are several different algorithms for diffuse surfaces.

3.3.1 Lambertian

The shading model used is a Lambertian reflectance model. It is an ideal diffuse reflectance model, which means that the brightness of the surface will stay the same, even if the observer changes its angle of view. That means that the BRDF will be equal to the same result for any direction ω_{in} and its corresponding ω_{out}. The BRDF is defined as $f_r = \frac{\rho}{\pi}$, where ρ is a constant coefficient representing reflectance and π is for normalizing the integral $\int f_r \cos\theta \, d\omega$. [1]

3.3.2 Perfect reflection

The perfect reflector surfaces will only show the specular reflections. Compared to a glossy surface, that is adding the specular reflections onto its diffuse information, the perfect reflector will only show its reflected data. These surfaces look like mirrors. The reflected direction(R) can be calculated as:

$$R = D - 2(D \cdot N)N \tag{10}$$

where D is the incoming direction and N the normal of the surface. [1]

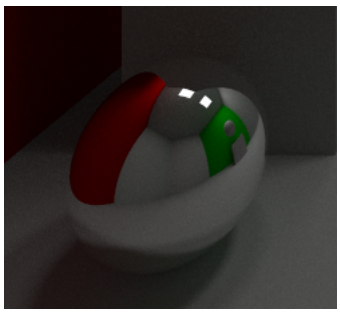

Figure 2: Perfect reflection applied to the sphere in the scene

3.3.3 Refraction

Glass/transparent objects. How rays handle refraction, split into two new rays.

When light goes from one translucent medium to another, it will change direction. This is called refraction. The direction of the light can be calculated with Snell's Law (11). This will require both mediums index of refraction (IOR), and the incoming angle. [1]

$$\sin\theta_1 n_1 = \sin\theta_2 n_2 \tag{11}$$

θ_1 is the incoming angle, θ_2 is the outgoing angle, n_1 is the first mediums index of refraction and n_2 is the second mediums index of refraction.

3.4 Direct light

Since most importance rays never hits a light source, they instead have to contribute to the radiance. Shadow rays are cast from the ray intersection points to the light sources where the direct light contribution from each point is added to the total radiance. If an object blocks the shadow ray, then the point is in shadow and should not contribute to the radiance. Furthermore, if the hit normal vector is not in the same hemisphere as the light direction, it indicates that

the light source is shadowed by the object that was hit. This way there is no need to check intersection for the shadow ray further which reduces computation time. [1]

3.4.1 Light sources

The light sources illuminate the scene and it is done differently depending on the type of light. Whitted ray tracing only supports point lights, while Monte Carlo allows for use of area lights too. [1]

Point lights are the simplest way to define a light source. This type only requires one shadow ray from the intersection point to the position x of the light source. If the ray is not blocked, it will contribute to the radiance according to the Lambertian cosine falloff, where the diffusely reflecting surface is directly proportional to the cosine angle of Θ between the light direction and the surface normal. [1]

Area lights consists of triangles and is considered a mesh compared to the point light sources which is merely a point in space. Area light sources consequently require approximation for multiple points on the mesh surface. This is done with a Monte Carlo estimator which attempts to solve the local lighting model,

$$L_D(x_N \rightarrow -\omega_{N-1}) = \int f_r(x_N, \omega_{in}, -\omega_{N-1}) L(x_N \leftarrow \omega_{in}) cos\Theta_{in} d\omega_{in} \quad (12)$$

with the discretizised estimator:

$$< L_D(x_N \rightarrow -\omega_{N-1}) >= \frac{AL_0}{M} \sum_{k=1}^{M} f_r(x_N, \omega_{in}, -\omega_{N-1}) V(x_N, q_k) G(x_N, q_k) \quad (13)$$

where $V(x_N, q_k)$ returns one if the point q_i on the light source is visible from x_N or zero if it is not, $G(x_N, q_k) = cos\alpha_{qi} cos\beta_{qi}/d_{qi}^2$ and M is the number of shadow rays that are cast to the light source. The surface area A of the triangle is calculated as:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad (14)$$

where a, b, c are the euclidean distances between each edge of the triangle. The points are picked randomly on the surface with a uniform PDF $p(q_k) = A^{-1}$. Here, the point q_k on the surface is parametrized with $q_k = v_0(1-u-v) + uv_1 + vv_2$ where u and v are randomly drawn numbers using uniform distribution. [1]

3.5 Indirect light

The indirect lighting adds another layer of light into the scene compared to direct lighting which is shown in Figure 4. To simulate this global illumination we need a way of sampling a direction of the indirect light at every intersection point that has a diffuse surface. A model is needed for which direction that the

indirect rays can travel. This can be represented with a hemisphere. [1]

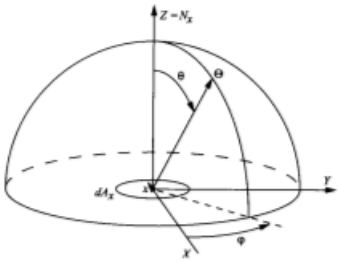

Figure 3: A hemisphere

A continuous expression of the hemi-sphere would be ideal, as that represents the whole surface that we are after, and it would cover all possible directions for the indirect light. The outgoing radiance for an intersection point x from all incoming hemi-spherical directions can be expressed as

$$L(x \to \omega_{out}) = \int \int f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in}) \cos \rho_{in} \sin \theta_{in} d\rho_{in} d\theta_{in} \quad (15)$$

The expression can be discretized in the following way:

$$\langle L(x \to \omega_{out}) \rangle = \frac{\pi}{N} \sum_{i=1}^{N} f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in}) \quad (16)$$

Where N is the number of new rays that will be sent into the scene from the intersection point x. f_r is the BRDF and $L(x \leftarrow \omega_{in})$ is the incoming radiance. The direction of the sampled incoming radiance should be distributed evenly over the surface of the hemi-sphere. That can be achieved by defining the random direction $\theta_j = \sin^{-1}(\sqrt{y_j})$ and $\rho_i = 2\pi x_i$. This process is then being executed recursively until the ray gets terminated. [1]

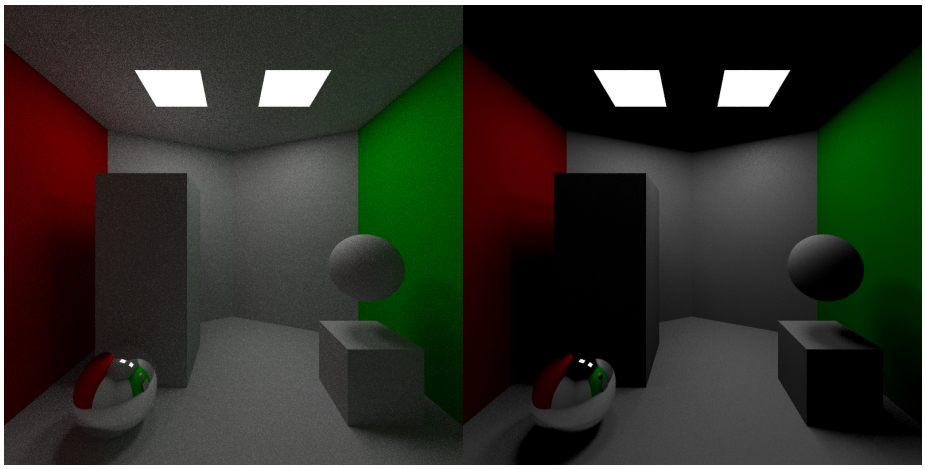

Figure 4: Image rendered with both indirect and direct light on the left, and only direct light on the right.

3.6 Ray termination

There are some rules for how the rays should be advected in the scene. The ray shall be terminated if it hits a light source or a lambertian reflector. The remaining rays can not be traced to infinity although the rendering equation is defined that way. Thus, the rays terminate using Russian Roulette. [1]

3.6.1 Russian Roulette

In order to get a natural fall off of ray depth, probability is used to terminate rays. An absorption probability $0 \leq \alpha \leq 1$ is selected and a random number $0 \leq t \leq 1$ is generated. If $t > 1 - \alpha$ the ray is terminated. By that, the absorption will control how many bounces the rays will have and the amount of bounces will vary with every generated ray. We do not add a constant ray limit to control the ray termination, as that adds bias to our implementation. By using probability the method will mathematically still be converging to the desired result over many iterations. [1]

3.7 Parallelization

By the nature of a Monte Carlo ray tracer there are in many cases millions of rays that are being calculated during a render. It depends on the resolution of the render as more pixels will add more rays to the scene, but also the amount of rays per pixel. When no parallelisation is enabled the code gets executed in a sequential manner, from the top of the code to the bottom. The ray tracing is calculated for one pixel at a time. The algorithm is set up in such a way that every pixel is independent from each other. No information is passed between the pixels, and therefore they can be calculated individually which allows for

parallel operations. Every pixel is assigned to a software thread, which gets distributed to the hardware threads by the operating system.

4 Result

The project resulted in a Monte Carlo raytracer that was written in C++ with Visual Studio 2019. First, we show the different phenomena that the raytracer support, such as reflection and color bleeding. This will be followed by a benchmark of the raytracer where several different variables that affect the renders are tested. The scene remains the same through all of the renders as it will be easier to determine the differences this way.

The supported features of the Monte Carlo raytracer is shown in Figure 5. Here, the left image shows that the scene is clearly reflected onto the sphere since the rays are reflected to gather irradiance according to the reflection law. An example of colorbleeding is shown in the middle image and is the result of indirect lighting when light is reflected multiple times. This causes the color from the green wall to be reflected onto the white diffuse sphere although this is rather subtle in the example. Lastly, the soft shadows are displayed which is a direct result of this raytracing scheme.

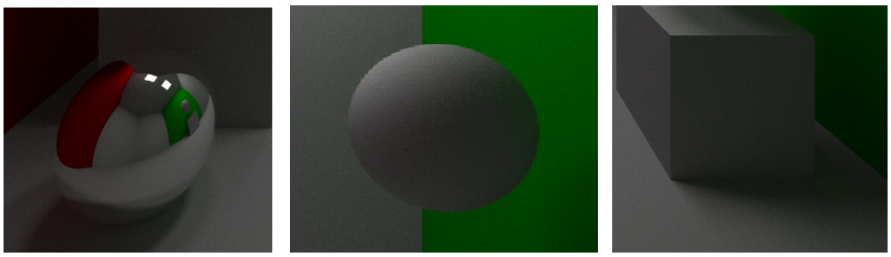

Figure 5: Reflection on the sphere on the left, colorbleeding in the middle and soft shadows on the right.

The scene was tailored to be used with logarithmic color scale. Thus, all of the resulting renders are rendered with this color scale but the general difference between linear and logarithmic color scale is shown in Figure 6.

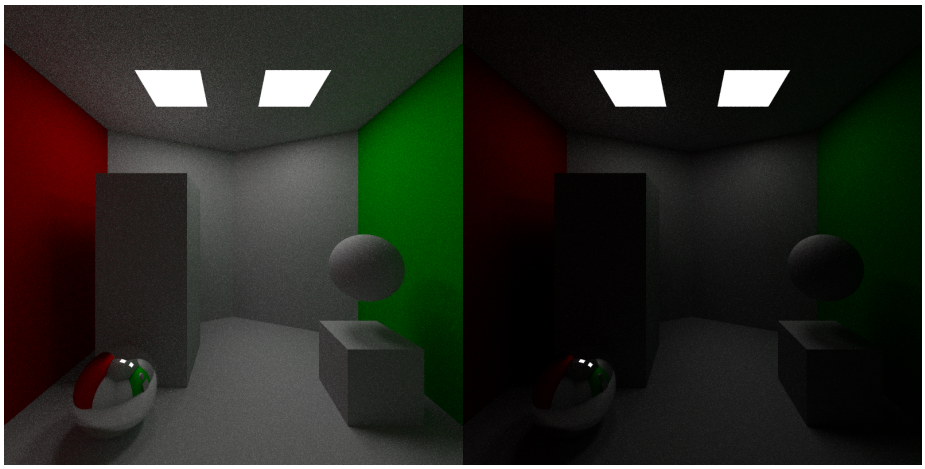

Figure 6: The logarithmic color scale on the left and the linear on the right.

4.1 Benchmark

This section will test different parameters and compare them in order to be able to determine what parameters affect the renders the most, and in what way they do that. All the images were rendered in the resolution 800x800 pixels.

Time (mm:ss)	Samples	Shadow rays	Depth	MultiCore(Y/N)	Placement
44:59	256	2	13	Y	L
12:00	64	2	14	Y	M
02:56	16	2	13	Y	R
01:43	16	1	14	Y	L
02:56	16	2	14	Y	M
05:20	16	4	14	Y	R
01:43	16	1	14	Y	L
01:35	16	1	7	Y	M
01:31	16	1	5	Y	R
01:43	16	1	14	Y	
05:07	16	1	14	N	

Table 1: Benchmark

The first test was done with the sample amount. This decides how many rays that are shot through each pixel. Since this will cause an exponential increase in computations, it is very expensive which makes it arguably the most important modifiable parameter. Table 1 shows all of the tests that were made on the raytracer. As can be seen here, the samples greatly affects the time but it also affects the amount of noise in the image, which is compared in Figure 7.

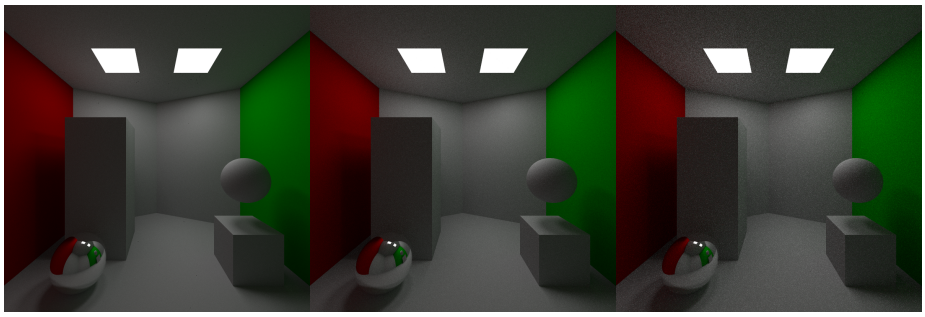

Figure 7: Sample test: 256 samples on the left, middle 64 and the right was rendered with 16.

Another test was done to see if the number of shadow rays visibly affected the render accuracy and was worth the extra time. Figure 8 shows the comparison between one, two and four shadow rays. In this result, there is barely any difference between the images which tends to suggest that one shadow ray is sufficient enough.

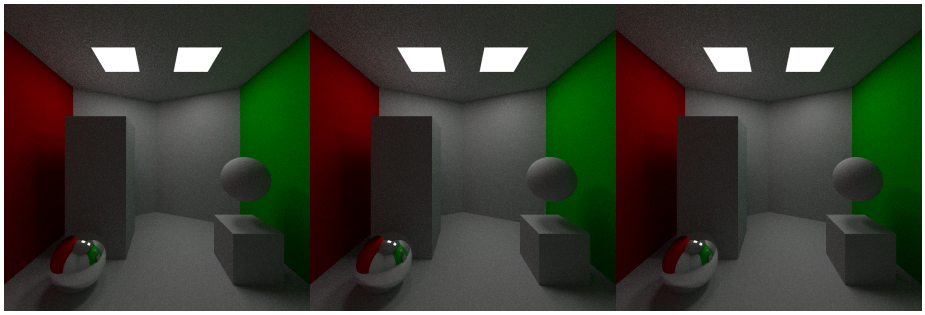

Figure 8: Shadow rays test: 1 shadow ray on the left, 2 in the middle and 4 on the right.

Lastly, the ray depth was tested. Although there is no maximum depth limit in this implementation, modifying the absorption coefficients within the scene will affect the amount of ray bounces. The depth count in Table 1 refers to the maximum measured depth for any ray in the image. The images are displayed in Figure 9 where a slight difference in reflected light can be observed though the differences are small. Note that the additional time for the extra bounces is relative small compared to other parameters impact. A render without any bounces will result in no indirect lighting and can be observed in Figure 4.

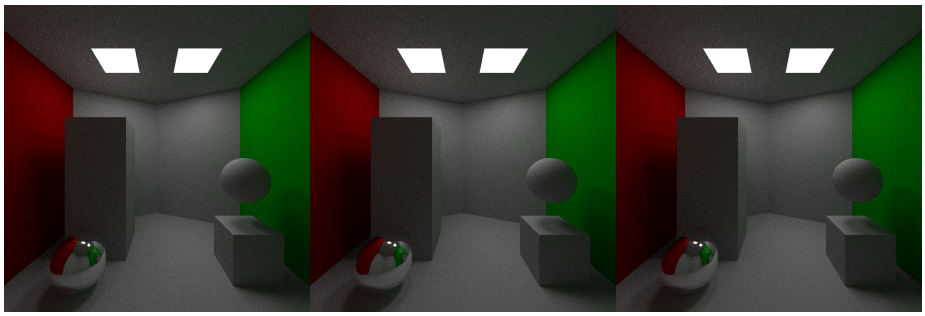

Figure 9: The ray depth test: Left rendered with 14 bounces, middle with 7 and right with 5.

Every test was done with parallel support from the CPU. The last two tests in Table 1 rendered the same image with and without parallelization. This feature reduced render times significantly and did reduce it even further on renders with higher sample numbers. The exact increase is impossible to tell here as the implementation does not guarantee parallization, only enables it when it is possible.

5 Discussion

The results of our implementation are visually pleasing. The effect of the indirect illumination can easily be distinguished with the smooth highlights and shadows, and accurate specular reflections. A satisfying result indeed.

There are several things that could be done for future work to improve the features of the renderer. More material models could be added such as transparent, refractive and sub surface scattering materials. The abilities of image based shading would also be interesting to implement, so that the surface properties can be controlled by image values.

It would be interesting to implement manipulation of surface normals. With combination of image based shading, normal mapping can be added to our renderer. This helps to efficiently visualize realistic uneven surfaces.

The BRDF that is used right now is a very simplistic model. That can be replaced with for example the Oren-Nayar model which has a more accurate representation of the falloff of intensity over a surface.

Photon mapping can also be implemented. This technique emit photons from the light sources in the scene towards specular and refractive objects. That simulates caustics, which is an interesting effect of light being distributed in an uneven manner, like in the bottom of a swimming pool.

To improve the performance of our renderer it would be interesting to add support for the GPU. This should be possible, as we have seen good performance scaling when parallelizing the ray tracing for the CPU. It should speed up the rendering process a lot as the GPU contains thousands of cores instead of just a few. As there are so many pixels that is supposed to be rendered, the threads should be able to be distributed effectively.

As science has moved on, there are also some improvements to the methodology itself that would be interesting to experiment with. For example *Continous Multiple Importance Sampling*, presented in [4]. This is interesting, as optimizations for the model of the importance sampling itself can reduce the level of noise and thereby also the performance of the renderer. To then combine this with a GPU implementation and a denoiser would be a lot more efficient as a whole package.

References

[1] Mark E Dieckmann. *Lecture Notes from the course TNCG15 at Linköping University.* 2020.

[2] Eric Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. *Department of Computer Science, Faculty of Engineering, Katholieke Universiteit Leuven*, 20:74–79, 1996.

[3] Alexandr Shcherbakov and Vladimir Alexandr. Dynamic radiosity. 2019.

[4] Rex West, Iliyan Georgiev, Adrien Gruson, and Toshiya Hachisuka. Continuous multiple importance sampling. *ACM Trans. Graph.*, 39(4), July 2020.