TNCG15: Advanced Global Illumination and Rendering
Monte Carlo Ray Tracing

Iris Kotsinas (iriko934@student.liu.se)
Louise Nilsson (louni313@student.liu.se)

November 5, 2020

Abstract

An implementation of a ray tracer using the Monte Carlo technique is presented in this report. The programming language that is used for the implementation is C++ and the project is part of the course TNCG15 – *Advanced Global Illumination and Rendering* at Linköping University.

Realistic illumination in computer graphics is a challenge to achieve and is computationally expensive if light hitting an object is calculated correctly. Realistic ray calculations compared to the possibility to perform it on a computer differ significantly. Thus, an approximation needs to be made to be able to implement it into the computer graphics.

These approximations resulted in a scene with a hexagonal room containing two spheres and one tetrahedron. To show the behaviour of the light, two different materials where applied to the objects. One of the spheres and the tetrahedron have a Lambertian reflector which absorbs the light, while the other sphere has a perfect reflector that reflects all rays hitting the object.

This report presents theory behind global illumination and rendered images, followed by analysis and discussion of the method.

Contents

1 Introduction

In the field of computer graphics, realistic illumination has always been a major challenge. Correctly calculating how the light hitting one object will effect the objects surrounding it has always been difficult due to it being computationally expensive. In order to perfectly emulate the effect of a light source, an infinite amount of rays would have to be sent out. Since infinite calculations cannot be realistically performed by computers, an approximation of realistic lightning can be created. Ray tracing is a global illumination method which provides realistic lighting in rendering by simulating the physical behavior of light. This report follows from the implementation of a Monte Carlo ray tracer made for the course Advanced Illumination and Rendering at Linköping University. The purpose of this study is to evaluate how an implementation of a Monte Carlo ray tracer perform in a scene and which factors will affect the end result.

1.1 Global Illumination

Global illumination is the process that simulates indirect lightning, such as light bouncing and color bleeding. The most common model for global illumination can be described by the rendering equation 1.

$$L_s(x, \Psi_r) = L_e(x, \Psi_r) + \int_\Omega f_r(x, \Psi_i, \Psi_r) L_i(x, \Psi_i) \cos\theta_i d\omega_i \tag{1}$$

The integral, L_r, of equation 1 is difficult to approximate. It is dependent on the BRDF, f_r and the incoming radiance L_i. For most reflective materials the BRDF can be split into two parts: a diffuse part $f_{r,d}$ and a specular part $f_{r,s}$ (see equation 2). The incoming radiance can be split into three parts according to equation 3.

$$f_r = f_{r,d} + f_{r,s} \tag{2}$$

$$L_i = L_{i,l} + L_{i,c} + L_{i,d} \tag{3}$$

In equation 3 $L_{i,l}$ is the light contribution from light sources, $L_{i,c}$ is the light contribution from specular reflections and refractions and $L_{i,d}$ is the light contribution from indirect soft illumination.

$$
\begin{aligned}
L_r = & \int_\Omega f_r L_{i,l} \cos\theta_i d\omega_i + \\
& \int_\Omega f_{r,s}(L_{i,c} + L_{i,d}) \cos\theta_i d\omega_i + \\
& \int_\Omega f_{r,d} L_{i,c} \cos\theta_i d\omega_i + \\
& \int_\Omega f_{r,d} L_{i,d} \cos\theta_i d\omega_i
\end{aligned} \tag{4}
$$

The rendering equation 1 can be split into a sum of several components, as can be seen in equation 4. Equation 4 is used to compute the radiance leaving a surface.

To simulate light, the global illumination method ray tracing can be used. In order to achieve ray tracing, rays are sent from a light source which will bounce in the scene until the camera has been hit. This requires an infinite amount of rays sent from the light source in order to have enough rays hit the camera, which is difficult to create due to the amount of computation required. Therefore another method of ray tracing is required to acquire similar results without being as computationally expensive.

Scenes rendered with global illumination methods typically contain walls of contrasting colors and objects with different surfaces, allowing the light to bounce on a variant of alternatives. An area light source in the ceiling spreads light in the scene. Spheres are used as objects in the scene since they are mathematically easy to describe.

1.1.1 Whitted ray tracing

In 1980, J.Turner Whitted published a paper in which he introduced Whitted ray tracing [1]. This method uses backwards tracing to simulate reflection and refraction. Rays are traced from the camera into the scene, where they bounce a predetermined amount of times or until a diffuse surface is hit. The light contribution at each surface intersection point is calculated using a local lightning model. When a diffuse surface is hit, shadow rays are sent out to examine if there is objects between the light source and the intersection point. Whitted ray tracing works well for reflection and refraction, but does however not consider the indirect light contribution from other diffuse reflections. The diffuse reflections can be calculated using the radiosity method.

1.1.2 Radiosity

Radiosity is a global illumination method which calculates the diffuse reflections in a scene, proposed by Goral et al. [2]. Each surface in the scene is divided

into smaller surface patches, which are assigned a view factor. This value is a coefficient describing how well the patches can see each other and determines the light transport between the different patches. Patches that are far away from each other will have smaller view factors. If other patches are blocking, the view factor will be reduced or zero, depending on whether the occlusion is partial or total. Every patch receives reflected light from all other patches in the scene.

1.1.3 Monte Carlo ray tracing

Monte Carlo ray tracing is one of the most general and physically accurate methods for solving the rendering equation 1, introduced by Kajiya [3]. This method approximates the integral by Monte Carlo estimation, which can be optimized with importance sampling. Importance sampling means that more important directions are sampled more frequently. This will contribute to the method converging to a more accurate solution of the equation in a shorter time.

Similar to Whitted ray tracing, the Monte Carlo ray tracer starts by sending rays from the camera into the scene. Monte Carlo ray tracing does however handle intersections at diffuse surfaces differently. When a diffuse surface is intersected by a ray, new rays are scattered at random angles and sampled in a hemisphere around the intersection point. Russian roulette is used to calculate when a ray should stop being traced. The Monte Carlo ray tracing method contribute to effects like color bleeding, caustics and soft shadows.

2 Background

In following sections, we will describe how the implementations has been done using some theory behind the renderer. The implementation has been done in the programming language C++. Instances is divided into different classes and to handle vectors the OpenGL Mathematics library has been included.

2.1 Overview of the method

The implementation of a ray tracer for the project is based on the Monte Carlo method for estimating integrals. The ray tracer uses backwards tracing with Monte Carlo integration on diffuse surfaces. Initially, a scene is created as well as a camera. The camera sends out a number of rays into the scene, which in turn travels through each pixel in an image plane. Each ray will bounce on objects a predetermined amount of times. When a diffuse surface is intersected by a ray, new rays are scattered at random angles and sampled in a hemisphere around the intersection point. The indirect light is then the average of light contribution returned by the sampled rays. Russian roulette could be used to calculate when a

ray should stop being traced, but is not yet implemented in this project. Shadow rays will be sent out to calculate the direct light contribution.

2.2 Vertex

In this project, the data structure Vertex describes the position of a point in 3-dimensional space with the spatial coordinates x, y, z and w, where w is the homogeneous coordinate. To create the objects in the scene, such as triangles and spheres, the vertex points is needed. For triangles, the vertices creates the structure of the objects while spheres is mathematically calculated.

2.3 Triangle

A triangle contains of three vertex points and a color vector. The three vertex points define a surface and each surface has a normal. The normal decides which side of the surface is the front. Depending on if you create the vertices in counter-clockwise order or in a clockwise order the normal would point in two opposite directions to each other. When all triangles are implemented they together make the objects of the scene such as the room's walls, floor and roof.

2.4 Scene

The scene is a hexagonal shaped room created with 14 vertex points. This room is represented by a triangle mesh with 24 triangles, 2 triangles per each of the 6 walls with various of colors and 6 triangles for both the floor and the roof. The roof and floor have a distance by 10 along the z axis and are located at the planes z=5 (roof) and z=-5 (floor). Figure 1 shows the view of the room from the top (left) and from the side (right).

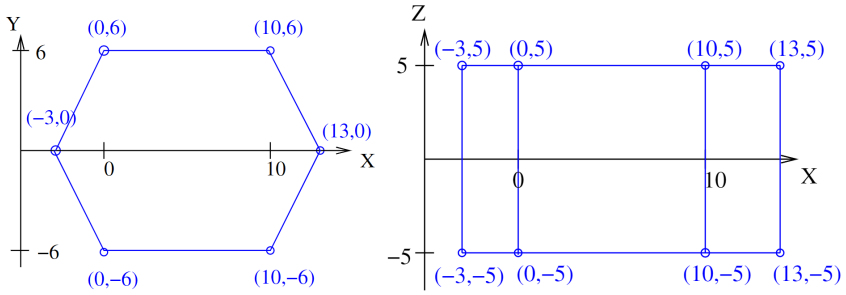

Figure 1: View of the room from the top and from the side with given vertices.

2.5 Objects

Two different objects were implemented in the scene: One tetrahedron and one sphere.

2.5.1 Sphere

Two spheres are implemented in this project, one is a perfect reflector and one is a diffuse reflector. *Analytic Solution* has been used as implementation method.

To begin with, a ray is expressed in parametric form in equation 5 where O is the origin of the ray, D is the ray direction and t is a parameter.

$$P = O + tD \tag{5}$$

A sphere is defined by the coodinates of a cartesian point and a radius R of the object. There are a set of points that represents the surface of the sphere that is centered at the origin. This gives the implicit function in equation 6

$$P^2 - R^2 = 0 \tag{6}$$

Where P is the point for the coordinates x, y and z. Equation 5 is substituted in equation 6. This is developed and substituted with a, b and c and gives equation 7 where $a = 1$ (Direction is normalized), $b = 2OD$ and $c = O^2 - R^2$.

$$f(x) = at^2 + bt + c \tag{7}$$

The roots are found using equation 8, where Δ is the discriminant that indicates if there are two, one or no root to the equation.

$$\begin{aligned} t &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ \Delta &= b^2 - 4ac \end{aligned} \tag{8}$$

If Δ is larger than zero, it results in two roots, which indicates that the sphere is intersected by a ray in two places. When Δ is equal to zero, there is one root, which means that the sphere is intersected by a ray in one place. If Δ is less than zero, the sphere is not intersected at any place[4].

2.5.2 Tetrahedron

A tetrahedron is a polyhedron composed of four triangular faces, six straight edges, and four vertex corners [5]. The tetrahedron in this project is a diffuse reflector and is created with the use of triangles. A tetrahedron is depicted in Figure 2.

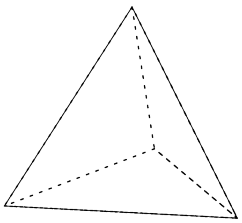

Figure 2: A tetrahedron.

2.6 Camera

The camera is implemented to view an image of the scene. It has a fixed position with 2 positions to choose from. The positions are -1 or -2 along the x axis. In this project the camera is subdivided into pixels with the resolution of 800x800 pixels for the camera plane.

2.7 Pixel

The pixel class is implemented to make the rays go through it. Each pixel holds the color and intensity with a dynamic range.

2.8 Rays

To simulate how a light-ray moves in a room, a ray class was implemented with a starting point p_s and an end point p_e, which is two instances of Vertex, and a direction vector to describe the ray. The direction of the ray is described in equation 9 where x is a three dimensional point that depends on the parameter t that has a value between 0 and 1.

$$x(t) = p_s + t(p_e - p_s) \tag{9}$$

The program contains of a bundle of rays that share the eye point as starting point. Each ray contains a color depending on the hit point and goes through a pixel. During ray casting, rays are sent from a camera through the pixels, which in order fills the pixels with color intensity information. This information is according to the color of the visible object, the amount of lightning the object receives from a light source at the pixel and the reflected light of other objects in the scene.

To bring light into the scene, an area light source were implemented by using two triangles with the vertices (6.5, 0.5, 4.9), (6.5, -0.5, 4.9), (4.5, -0.5, 4.9) and (4.5, -0.5, 4.9).

2.8.1 Ray Intersection

To implement ray tracing that includes triangle meshes the Möller-Trumbore intersection algorithm is used. It is considered a fast algorithm that makes use of the parameterization of the intersection point P. Therefore, there is no need to precompute the plane equation of the plane containing the triangle.

Assume that a triangle is defined with the three corner vertex points, $v0$, $v1$ and $v2$. By describing the point of the Bary centric coordinates u and v the Möller Trumbore algorithm is calculated in equation 10.

$$T(u, v) = (1 - u - v)v_0 + uv_1 + vv_2 \tag{10}$$

The triangle is intersected with the ray which gives equation 11.

$$x(t) = p_s + t(p_e - p_s) = (1 - u - v)v_0 + uv_1 + vv_2 \tag{11}$$

By using the Cramers rule, this equation can be solved. In equation 12 definitions is made,

$$\begin{aligned} T &= p_s - v_0 \\ E_1 &= v_1 - v_0 \\ E_2 &= v_2 - v_0 \\ D &= p_e - p_s \\ P &= D \times E_2 \\ Q &= T \times E_1 \end{aligned} \tag{12}$$

where T, E_1, E_2, D,P and Q is constant vectors. This information gives the solution of equation 4 and the intersection point can now be calculated directly with u and v in equation 13.

$$t = \frac{Q \times E_2}{P \times E_1} \tag{13}$$

The value t is later used to choose which object is the closest to the camera.

2.8.2 Shadow Rays

By using ray tracing computing shadow rays is done during the rendering process. When a ray is casted from a pixel visible in the frame to the light source and this ray intersects an object on its way to the light, this pixel is defined as a shadow (see Figure 3).

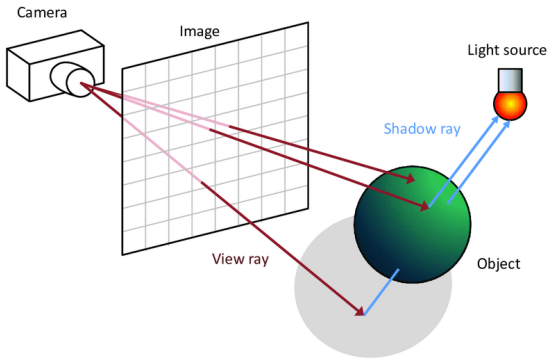

Figure 3: The concept of a shadow ray

2.9 Rendering

Each row in equation 4 is estimated in order to solve the rendering equation 1. The Monte Carlo rendering scheme is a recursive method, and some of the terms will therefore be dependent on others. This implies that direct light contribution needs to be established for indirect light contribution to be evaluated.

2.9.1 Direct illumination

The first term in equation 4 represents the contribution via direct illumination by the light sources. Normally, this term is computed by sending shadow rays towards all light sources in order to check for visibility. The light contribution considered is when the ray directly hits a light source or through one or several specular reflections or refractions. Once the ray hits a light source, it gains radiance and is then traced back through the recursion.

As previously stated, a shadow ray is sent from an intersection point when a ray hits the surface. This is done to ensure that enough light contribution gets on every object in the scene. By calculating a vector from the intersection point to the light source a shadow ray is created. It is then tested for various intersection in the scene. If an object intersects with the shadow ray, light contribution will not apply for this location and the light source will cause a shadow. The intensity of this intersection will therefore be set to black. In those cases when the shadow ray does not intersect with any object, a local intensity is calculated for the hit point.

2.9.2 Specular reflection

The second term in equation 4 is radiance reflected of specular and glossy surfaces. This value can be computed using standard Monte Carlo ray tracing. It is possible to use the BRDF to generate the optimal sampling directions. By using importance sampling the computation can in most cases be done using a limited number of sample rays.

2.9.3 Soft Indirect illumination

The forth term in equation 4 is incoming light reflected diffusely at least once since leaving the light source. The light is then reflected diffusely by the surface and the resulting illumination can be perceived as soft. The indirect illumination enables color bleeding from other illuminated surfaces in the scene.

3 Result

A Monte Carlo ray tracer was implemented for this project. It was tested on a scene of which contains three objects: a tetrahedron and two spheres. The tetrahedron has a Lambertian surface, and the spheres have a Lambertian surface and a perfect reflecting surface respectively. Various amount of samples per pixel and shadow rays per pixel have been rendered and tested in order to analyze the implemented ray tracer.

By sending several rays from the camera through the same pixel noise can be reduced in the images. When increasing the amount of samples per pixel, multiple rays are sent through the pixel in a uniform fashion instead of only sending one. The number of shadow rays used to sample the light from each intersection point is directly related to the quality of the direct diffuse illumination. The Monte Carlo integration has been set to stop after three bounces in the scene.

Table 1: Rendering statistics

Samples	Shadow rays	Rendering Time
3	1	3 min
3	10	10 min
3	3	5 min
10	3	15 min
50	3	1h 20 min

In Figure 4 a scene is rendered with one shadow ray per pixel. In Figure 5 the same scene is rendered with 10 shadow ray per pixel. The first Figure 4 contains more noise than Figure 5. It can be noticed how the amount of noise on the floor decreases as the number of shadow rays increases. However, the time to render

the image increases with more shadow rays. Figure 4 had a rendering time within 3 minutes. When the amount of shadow rays increased to 10 shadow rays per pixel, it took approximately 10 minutes to render the image. This is a significant increase of time (see Table 1).

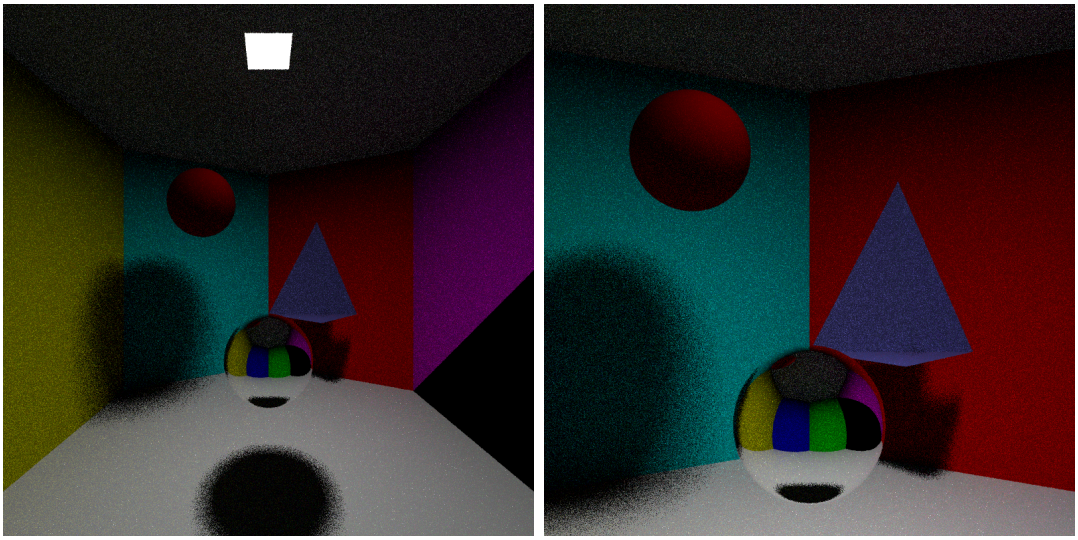

Figure 4: Scene rendered with 3 samples per pixel and 1 shadow ray per pixel.

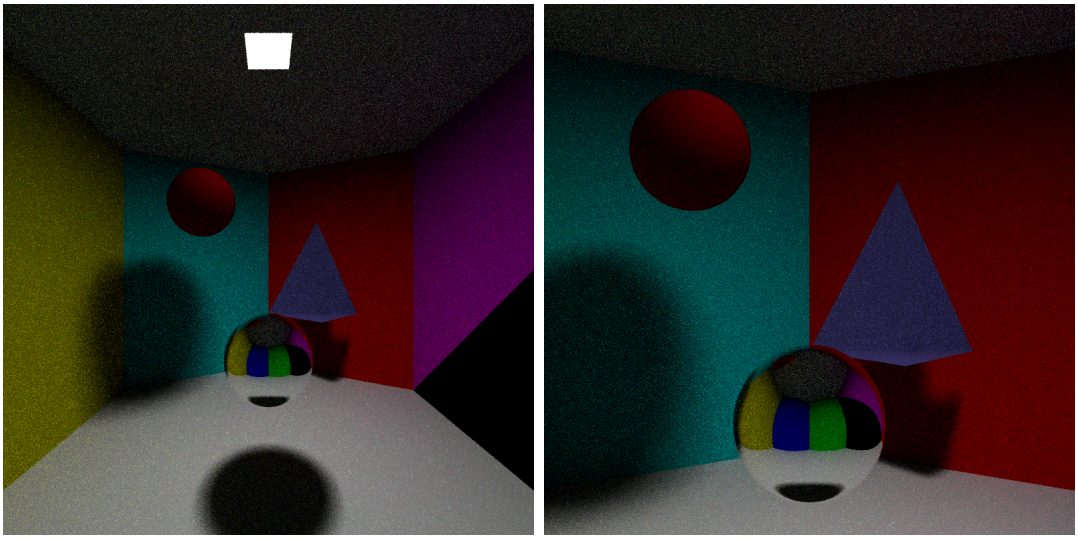

Figure 5: Scene rendered with 3 samples per pixel and 10 shadow ray per pixel.

In Figure 6 a scene is rendered with three samples per pixel. In Figure 7 the same scene is rendered with ten samples per pixel. The first Figure 4 is noisy

and not as detailed. In Figure 7 it can be seen that the noise subsides and more details can be noticed. The Figure 8 has been rendered with 50 samples per pixel. The image is noticeably less noisy than both Figure 6 and Figure 7. However, the rendering time differed considerably between the figures. Figure 6 with 3 samples per pixel rendered within a 5 minutes and Figure 7 with 10 samples per pixel rendered for around 15 minutes. When the samples increased to 50 samples per pixel the rendering time was 1 hour and 20 minutes. It is a major increase compared to the first to examples (see Table 1).

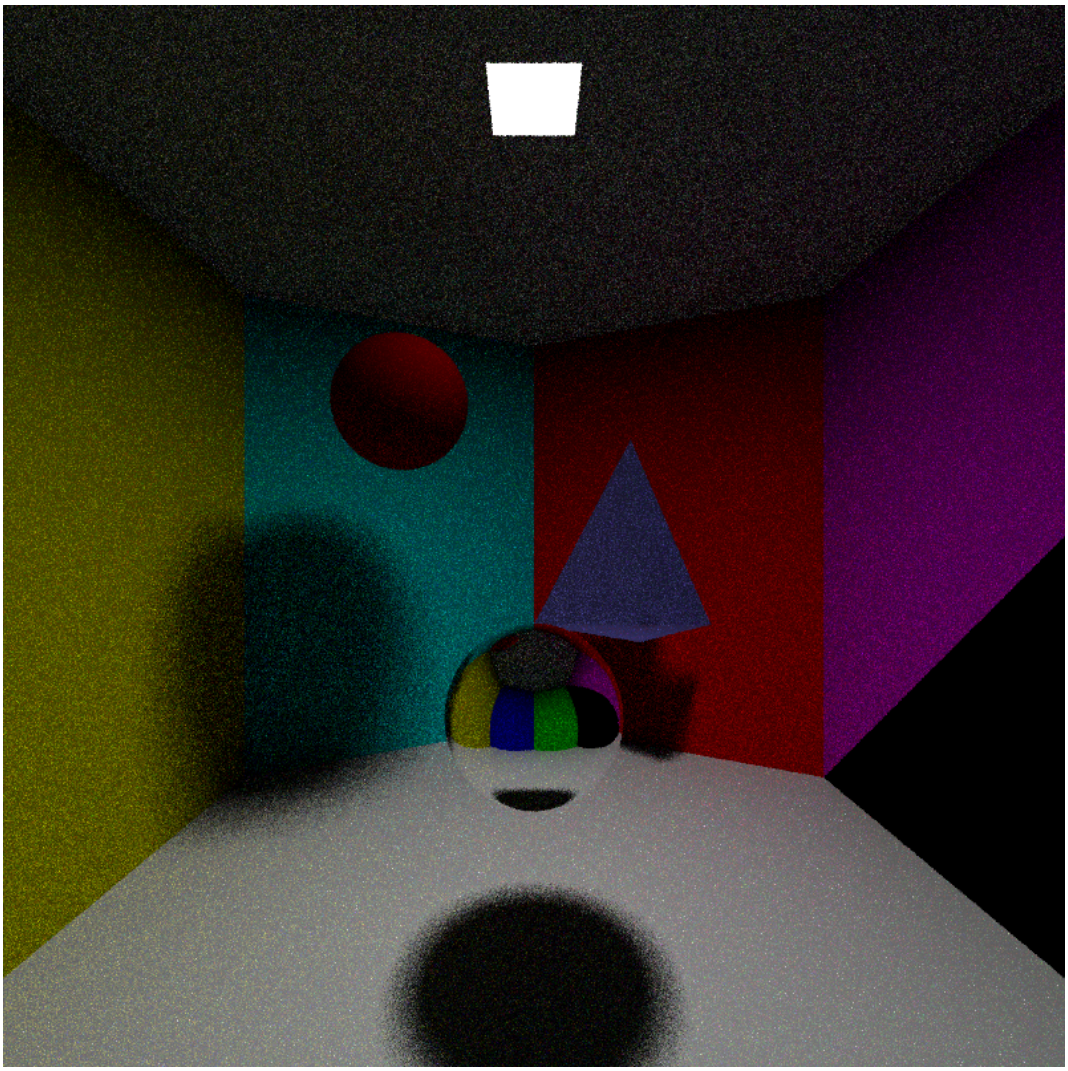

Figure 6: Scene rendered with 3 samples per pixel and 3 shadow ray per pixel.

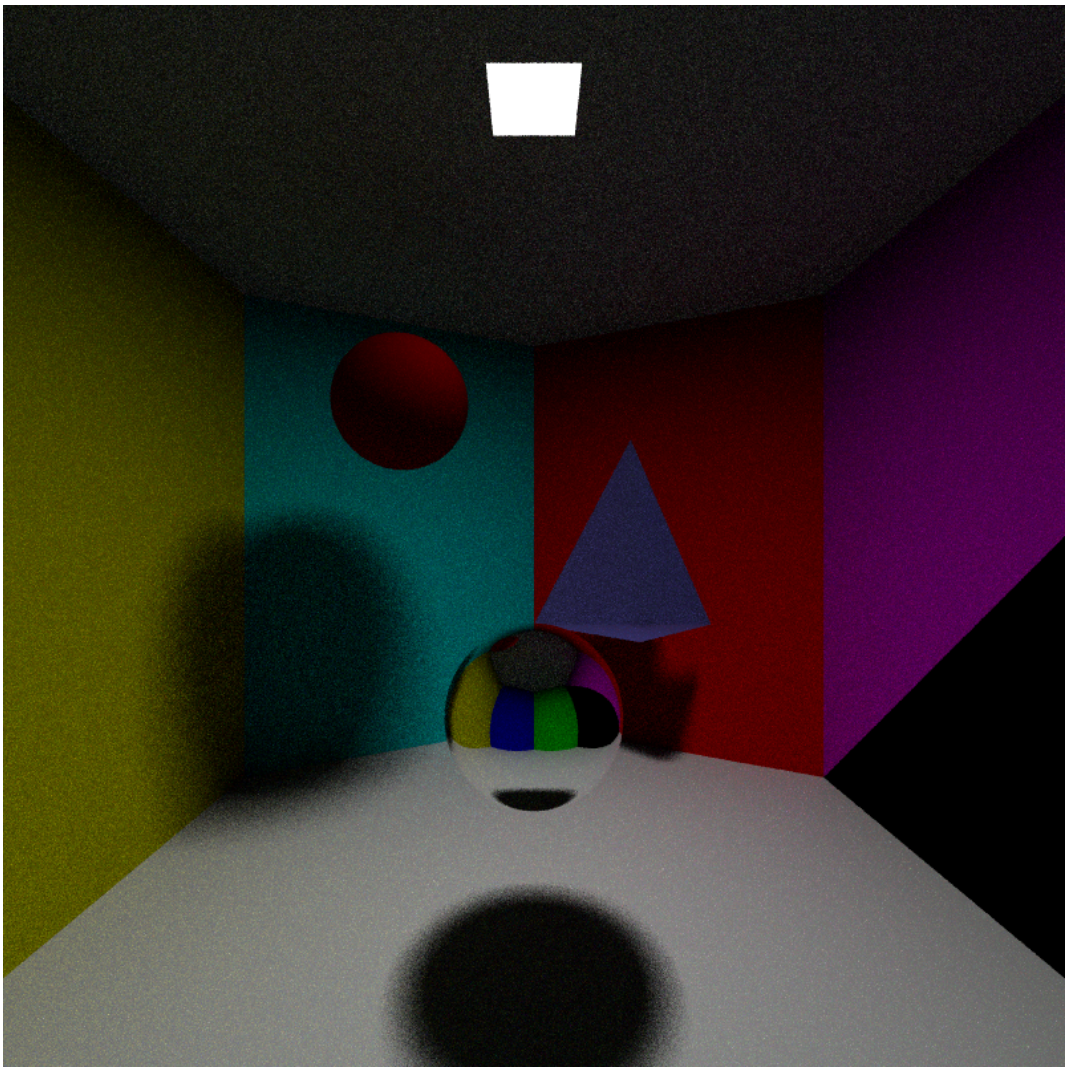

Figure 7: Scene rendered with 10 samples per pixel and 3 shadow ray per pixel.

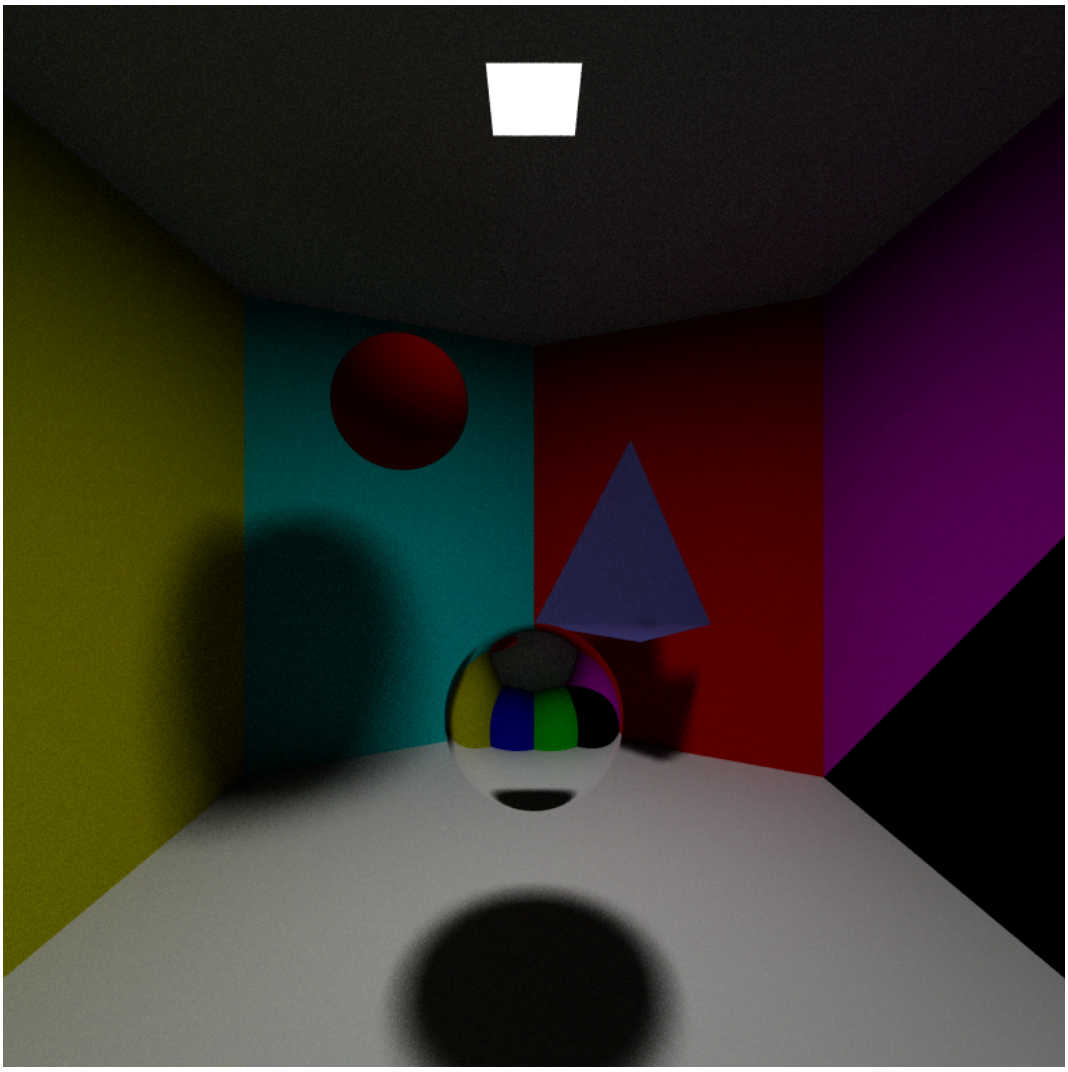

Figure 8: Scene rendered with 50 samples per pixel and 3 shadow ray per pixel.

4 Discussion

A basic Monte Carlo ray tracer was implemented and analyzed in this report. The process of creating photo realistic scenes with computer graphics is complex and computationally expensive, and a number of factors affect the outcome. The purpose of this study was to evaluate how an implementation of a Monte Carlo ray tracer perform in a scene and which factors affect the end result.

This project initially required lots of research about different methods used to implement a ray tracer. High quality renders of the scene was made possible by implementing a ray tracer based on the Monte Carlo method. The final ray tracer managed to render photo realistic images of light intersecting with diffuse and specular surfaces in the scene. The light source used in the implementation was a square light, located in the ceiling. The light source casts shadows behind the objects, which can be noticed in Figures 4 to 8.

Two different perspectives of the scene were rendered in order to help the viewer notice minor details more easily (see Figure 4 and 5). To make this possible the camera was given two different possible positions, chosen by the user. The rays in the scene are set to bounce three times before termination. An increase of this number would add to the rendering time, making it even more time consuming.

The use of 50 samples per pixel resulted in an image with less noise and more details (see Figure 8). The rendering of the image was however very time consuming due to the amount of samples. The same applied to the amount of shadow rays per pixel. Using a higher amount of shadow rays resulted in less grainy shadows, but it also led to a longer rendering time (see Figure 5). A higher amount of samples and shadow rays would increase the photo realism of an image, but it would decrease the performance of the computer, resulting in an interminable rendering time.

The time consuming rendering time is not optimal and could be optimized for the better. With that said, the implementation of the ray tracer could have been optimized by implementing other methods. The ray tracer does as of now not use Russian Roulette for early termination during Monte Carlo termination. The use of such a method would prompt the rays to terminate early, allowing the ray tracer to use a higher amount of reflections without decreasing the performance much.

Multi-threading could have been used in the implementation to decrease the rendering time. The application would then spawn multiple threads equal to the number of CPU cores in the computer. They would each be responsible for a part of the computation, reducing the amount of time required to render the image.

A polygon object, for example a teapot, could also have been implemented to increase the variety of shapes and to test the code´s performance. This would also make the image more interesting and probably even more realistic. Next step would be to implement objects with a glossy respectively a transparent surface by developing the Monte Carlo ray tracing method.

The final result of this project could be improved in a number of ways, but it is sufficient enough to present a photo realistic image created with a Monte Carlo ray tracer.

5 Conclusion

The goal of this project was to implement a Monte Carlo ray tracer that could generate photo realistic images. Even though the time it took to render the images with good results was very time consuming, the goal was achieved. Improvements could have been made for optimization of the renderer, but the result was still satisfactory.

References

[1] T. Whitted, "An improved illumination model for shaded display," Graphics and Image Processing. Bell Laboratories Holmdel, New Jersey, 1980, pp. 343–349.

[2] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," SIGGRAPH Comput. Graph. Cornell University, Ithaca, New York, 1984, pp. 213–222.

[3] J. T. Kajiya, "The rendering equation," SIGGRAPH Comput. Graph. California Institute of Technology, Pasadena, Aug. 1986, pp. 143–150.

[4] Scratchapixel, "A minimal ray-tracer: Rendering simple shapes (sphere, cube, disk, plane, etc.) (ray-sphere intersection)," Scratchapixel.com, 2014. [Online]. Available: `https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection` (visited on 10/26/2020).

[5] Math-Only-Math, "Tetrahedron," Math-Only-Math.com, 2020. [Online]. Available: `https://www.math-only-math.com/tetrahedron.html` (visited on 10/31/2020).