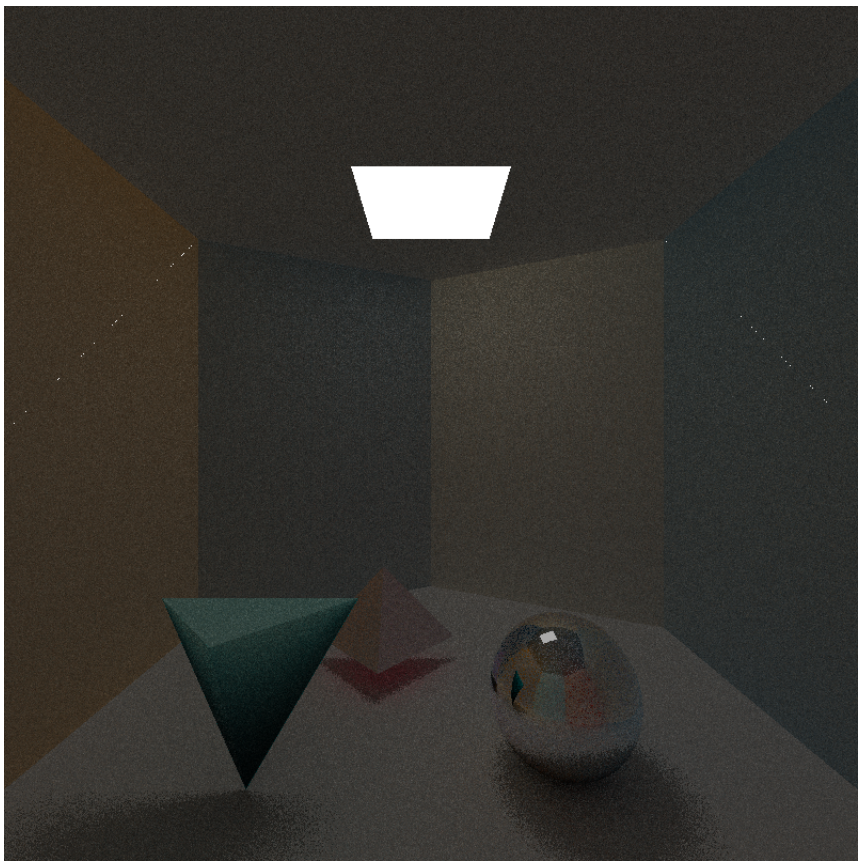
Monte-Carlo Raytracer

Advanced Global Illumination and Rendering

TNCG15



Fei Alm, feial961

Sofia Sproge, sofsp932

MT4

December 15, 2021

Abstract

This paper is about the theory and implementation of a Monte-Carlo Ray Tracer. It is a method that can render a scene in computer graphics and simulate it in indirect light by sending out multiple rays from a intersected point in order to gather data in the form of color samples. The resulting pixel color is set on the pixel plane which is the resulting image. This gives a realistic result with traits such as color bleeding, shadows, reflection and refraction. In the project a scene have been created with six walls in various colors, a floor, a roof and three objects where two of them are tetrahedrons and has Lambertian properties and one is a sphere which is a perfect mirror. The scene also has a light source that emits irradiance which contributes light into the scene.

The result proves that a larger amount of scattered rays gives a more detailed and sharp image. The ray depth in the implementation yields darker pixel colors but does not reflect on what the Monte-Carlo method should yield.

Contents

List of Figures

Chapter 1

Introduction

In this section we will introduce Monte-Carlo Raytracing and how we utilize the theory in our project.

1.1 Background

In order to render computer graphics a method for calculating light in a scene is needed. There are two ways to aproching this, one is to use a local lightning model and the other is to use a global lightning model.

A local lightning model (*Local Illumination*) only considers direct light in a scene, *see figure 1.1*. This means that while rendering, the algorithm takes in light rays from the light sources but will ignore indirect light that has been reflected or transmitted by reflective or specular materials on objects in the scene. When using local illumination, the scene will as a result lack effects that give a realistic appearance. This method will also not take shadows into consideration either which means that the objects in the scene do not seem to interact with each other. A classic example of an local illumination model is the Phong Model and it can be used to simulate glossy and diffuse surfaces.[1] [2]

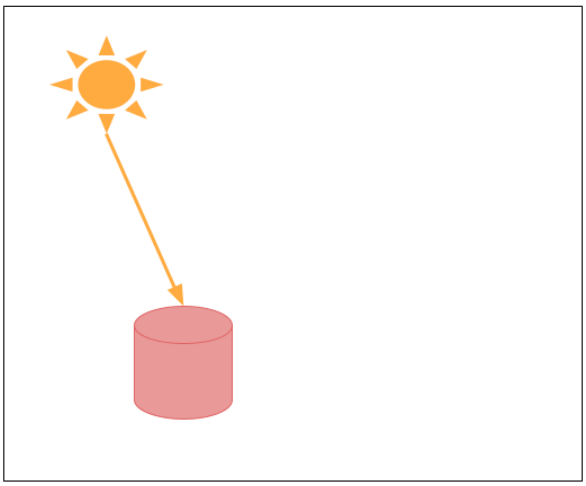


Figure 1.1: Local illumination, only takes direct light into considerations.

In the real-world objects affects each other and this means that effects like color bleeding can occur. This is because all materials reflect small amounts of light which is called indirect light and it contributes to the light of nearby objects. A global lightning model (*Global Illumination*) is able to simulate this, *see figure 1.2*.[2]

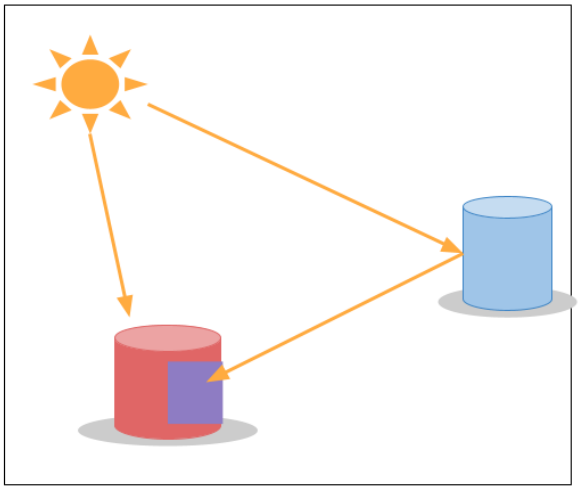


Figure 1.2: Global Illumination, considers indirect and direct light. Will result in color bleeding.

1.1.1 Whitted Ray-tracing

A classic example of a global illumination model is whitted ray-tracing and it is a method that calculate how light bounces in a scene by tracing it and creating a ray tree. Every pixel color of a 2D plane is set by using backwards ray tracing where it sends a ray from the eye/camera, through the pixel, and follow as it is reflected and refracted of objects in the scene. The ray is terminated when it reaches a diffuse object or reached a maximum set number of reflections/

refractions. The ray tree is then traversed back up in order to calculate the color based on the object it intersected in the end. This means that the method can simulate reflection and refraction. [3]

1.1.2 Radiosity

Radiosity or form-factor is a lightning model that uses forward raytracing which means, it follows the light source's perspective and path to compute light on Lambertian surfaces. One should split every surface in the scene into small patches (triangles). When a light ray hits a visible patch (irradiance) that patch will emit light (radiosity) into another direction using BRDF, *see section 2.2*. The recursive ray can be terminated after some arbitrary recursion, absorbed into material or using some other method such as Russian Roulette. [4]

The disadvantages is that it is hard to render realistic specular reflections correct, also form-factor demands a lot of memory due to the surface patches.[5]

1.1.3 Monte-Carlo Ray-tracing

Monte-Carlo ray-tracing can be used to further improve the realism of a rendered scene. It is based on whitted ray-tracing where it also uses backwards ray-tracing but instead using the last sample as the resulting color, Monte-Carlo uses all intersections as sample estimates in order to compute the resulting pixel color. This enables the possibility to get color bleeding and soft shadows.[5] See more about Monte-Carlo Ray-tracing in 2.

1.2 Aim of the project

The aim of this project is to research the Monte-Carlo Raytracer method and implementing the rendering model to demonstrate the theory. The project will also compare different settings within the code such as number of scattered rays and how deep the ray tree is allowed to be. The results will be discussed with the resulting quality of image and rendering time as the important factors.

1.3 Boundaries

This paper will not include methods that render transparent materials.

Chapter 2

Method

In this chapter the research about the Monte-Carlo Ray-tracer is construed.

2.1 Monte-Carlo Ray Tracing

Monte-Carlo ray tracing measures light by using backwards ray tracing. A pixel color is determined by sending a ray from the camera into the scene through the pixel in the pixel-plane and follow as it is reflected and refracted in scene, also called ”bouncing”. The ray will continue to bounce until it hits a light source or a diffuse surface where it is terminated. The result is a ray tree that has been created with recursion and with the rendering equation, *see equation 2.1* the tree is traversed back up in order to calculate the resulting pixel value based on all the collected samples. [4]

$$L(x \rightarrow \omega_{out}) = L_e(x \rightarrow \omega_{out}) + \int f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in}) cos\theta_{in} d\omega_{in} \tag{2.1}$$

- $L(x \rightarrow \omega_{out})$ is the outgoing reflected light (radiance) along the direction (ω_{out}) and is the resulting pixel value.
- $L_e(x \rightarrow \omega_{out})$ is the radiance emitted from the hit-point.
- $(\int f_r(x, \omega_{in}, \omega_{out}))$ is the BRDF 2.2 that represent the percentage of light reflected from the surface.
- $L(x \leftarrow \omega_{in})$ is the incoming light.
- $cos\theta_{in}$ is a factor that makes the surface diffusely reflective. It is the angle between the incoming light and the surface normal, which is based on Lambert's cosine law.[6]

The integral can't be solved numerically and therefore has to be estimated. In the Monte-Carlo method this is done by scattering multiple rays from the interaction point out into the scene. The directions are random and the main idea is to calculate the incoming light over the

hemisphere 2.1.2 based on the collected samples. The number of scent rays is limited where a higher number will result in a more accurate result.

2.1.1 Hemispherical coordinates

When a ray hits a point on a surface, called x in this case, from an incoming direction one wants to compute the outgoing direction in a manageable way in the 3-dimensional space. In order to do that one can use hemispherical coordinates. In figure 2.1 we can see how all directions are visible from the point x and that the hemisphere surrounds the point.[7]

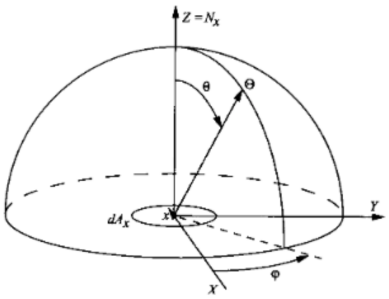


Figure 2.1: A point x surrounded by a hemisphere. The angles, θ and φ is also shown.
Source: Mark E. Dieckmann, TNCG15

To get the ray-direction in 3d, two 2-dimensional angles in spaces are used. The first angle is the zenith angle θ which rely on rely on z-axis and x-axis. The second one, the azimuth angle φ is between x- and y-axis. There ranges is: $\theta \in [0, \frac{\pi}{2}]$ and $\varphi \in [0, 2\pi[$.[7]

The *solid angle,* Ω, is a surface element A (area) on the hemisphere with radius $r = 1$ and unit *Steradian (sr)* there $\Omega = \frac{A}{r^2}, \Omega \in [0, 2\pi]$. It can be computed using hemispherical coordinates and it used to determine direction and luminosity of point x.[7]

2.1.2 Hemispherical integral

Using the hemispherical coordinates one can integrate all incoming radiance at point x. From section 2.1.1 we get the solid angle dω. It is structured by the azimuth angle dφ and the inclination angle dω according to this: $d\omega = sin\theta d\theta d\varphi$.[7]

Equation 2.2 shows how irradiance is calculated from the incoming radiance in a direction ω to point x.

$$E(x) = \int L(x \leftarrow \omega) cos\theta d\omega \tag{2.2}$$

By using the structure for the solid angle and deposit in equation 2.2 we get equation 2.3 down below.

$$E(x) = \int \int L(x \leftarrow \omega) cos\theta sin\theta d\varphi \tag{2.3}$$

Then one can integrate over the inclination angle and azimuth angle, *see figure 2.1*. Where the limits are in the intervals: $d\omega \in [\omega_1, \omega_2]$ and $d\varphi \in [\varphi, \varphi]$ between the incoming radiance and the irradiance.

2.2 Surface Material

The objects in a scene has different kinds of material and these are Lambertian surfaces and perfect mirror surfaces. In general, all materials (specular reflection, diffuse or glossy) are dependent on the bidirectional Reflectance Distribution Function (BRFD). It calculates radiance (outgoing light) and irradiance (ingoing light) as a scalar value from a point *x* on the surface and how they behave to create surface properties, equation 2.4. The radiance direction is computed using azimuth (φ) and zenit (θ) angle and considers the direction of the irradiance.[8] [9]

$$f_r(x, \omega_{in}, \omega_{out}) = \frac{dL(x \rightarrow \omega_{out})}{dE(x \leftarrow \omega_{in})} \tag{2.4}$$

In the further subsections the different surface materials and their BRDF will be accounted for.

2.2.1 Lambertian

A Lambertian surface is independent of viewing direction, which means it appears equally bright from all directions, wherever the eye is located, *see figure 2.2*.The brightness changes when the illumination direction or irradiance changes and the radiance is proportional to irradiance. In the Lambertian BRDF equation 2.5 it is seen that ρ is a albedo coefficient that describes how much light that will be reflected. The coefficient value is in the interval $[0, 1]$.[9] [10]

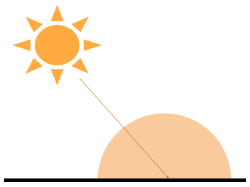


Figure 2.2: Wherever the eye is located, the material will be equally bright from all directions.

$$f_r(\theta_i, \varphi_i, \theta_r, \varphi_r) = \frac{\rho}{\pi} \tag{2.5}$$

The distributed light is $f_r = \frac{\rho}{\pi}$. φ is a integration constant that normalizes the integral $\int f_r cos\theta d\omega$. It means that we integrate over all possible incoming and outgoing directions of light and sum it up. It will be equal to 1. However, the ρ is a coefficient of scalar value so we

do only need to integrate over the incoming light. $cos\theta d\omega$ is the angle between incoming light and the normal at point x at the surface.[9]

2.2.2 Perfect mirror

In a perfect mirror all incident energy is reflected in a single direction. Also, the outgoing light has the same angle as the incoming light with respect to the normal. It can mathematically be described as:$\theta_i = \theta_r$ and $\varphi_i = \varphi_r + \pi$ there π is used to locate the outgoing light on the other side, a half circle turn in hemspherical coordinates, *see section 2.1.1*. One can rewrite the rendering equation 2.1 and describe the perfect mirror usinng the Dirac delta function 2.6.[10] [9]

$$\int f(x)\delta(x - x_0)dx = \begin{cases} f(x_0) \\ 0 \qquad \text{otherwise} \end{cases} \tag{2.6}$$

The result is shown in equation 2.7.

$$L(x \rightarrow \omega_{out}) = \int f_r(x, \omega_{in}, \omega_{out})L(x \leftarrow \omega_{in})cos\theta_{in}d\omega_{in} \tag{2.7}$$

The Dirac delta is a way to describe how all light gets reflected in one direction and not distributed evenly in all directions such as a Lambertian material.[9]

2.3 Shadows

When the pixel value is calculated it is also checked if it is in shadow or not. In order to do this a shadow ray is sent from the hit-point towards the light source and checks for intersections with any object in the scene. The resulting length of the ray is then compared with the distance between the object and the light source. The hit-point is occluded by another object if the distance is shorter and if it is the same length, the object is in direct light. In order to get soft shadows multiple rays can be sent into random parts of the light source and not only the middle. The result is a factor between 0 and 1 where 0 is completely occluded.

2.4 Intersections

In the project, two types of objects have been implemented in the scene. The first type is a polygon. The other one has a surface made up by three coordinates (implicit equation and surface). In the further subsections we will describe how a ray can detect an intersections on these two different types.

2.4.1 Möller-Trumbore

Several triangles gives us a tetrahedron. A way to detect if a ray intersect one such object is using the *Möller-Trumbore* algorithm. The ray is a straight line is used for ray-tracing and is defined as in equation 2.8. It has a starting point $\boldsymbol{p_s}$ and endpoint $\boldsymbol{p_e}$. The variable $0.0 \leq t \leq 1.0$ scales the length of the direction $\boldsymbol{p_e} - \boldsymbol{p_s}$. We also know that if t is equal to 0, the ray has no length and could not possible be able to intersect a object. Combined it defines the intersection point x.[11]

$$x(t) = \boldsymbol{p_s} + t(\boldsymbol{p_e} - \boldsymbol{p_s}) \tag{2.8}$$

A triangle is assigned by three vertex points $\boldsymbol{v_0}$, $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$. The method also uses barycentric coordinates $u \geq 0$ and $v \geq 0$ there the boundaries is according to eq.(2.9).[11]

$$u + v \leq 1 \tag{2.9}$$

When combine the vertex points and the barycentric coordinates, we can define a point (u, v coordinates) in a triangle T, equation 2.10. This is the ray intersection for a triangle.

$$T(u, v) = (1 - u - v)\boldsymbol{v_0} + u\boldsymbol{v_1} + v\boldsymbol{v_2} \tag{2.10}$$

The equations 2.8 and 2.9 combined receives the equation 2.11 down below.

$$(1 - u - v)\boldsymbol{v_0} + u\boldsymbol{v_1} + v\boldsymbol{v_2} = \boldsymbol{p_s} + t(\boldsymbol{p_e} - \boldsymbol{p_s}) \tag{2.11}$$

From equation 2.11 one can solve the variable t and then find the intersection point x using equation 2.8.[11]

2.4.2 Sphere

For spheres, a ray is defiend as in equation 2.12. $\boldsymbol{o}$ is starting point and $\boldsymbol{l}$ is a normalized direction of the ray. d is equal to a point on the ray.

$$\boldsymbol{x} = \boldsymbol{o} + d\boldsymbol{l} \tag{2.12}$$

The intersection for a ray and a sphere is defines as in equation 2.13. We have a coordinate vector x which defines a point on the sphere's surface. The center point of the sphere is $\boldsymbol{c}$ and its radius is $\boldsymbol{r}$.

$$||\boldsymbol{x} - \boldsymbol{c}||^2 = r^2 \tag{2.13}$$

When inserting equation 2.13 int equation 2.12 one can solve the value for d and due to that get the intersection point $\boldsymbol{x}$.

Chapter 3

Results

In this section the results are presented. The scene contains of one light source, two tetrahedrons and a sphere. The sphere has a perfect mirror material while the room (walls, roof, floor) and the two tetrahedron has Lambertian properties. The implementation has a big-time complexity and takes several hours to render. Due to that, the time has been measured and round off in the time-unit hours.

3.1 Result 1

In figure 3.1 we can see the scene and the objects. All three objects interact with the light and casts shadows. We can also see some shadows on the walls and how the wall turns gradually lighter towards the upper parts. It also becomes dark close to the seams towards the roof because the light cannot travel with an angle larger than 180 degrees. Under the red tetrahedron color bleeding can be found on the floor.

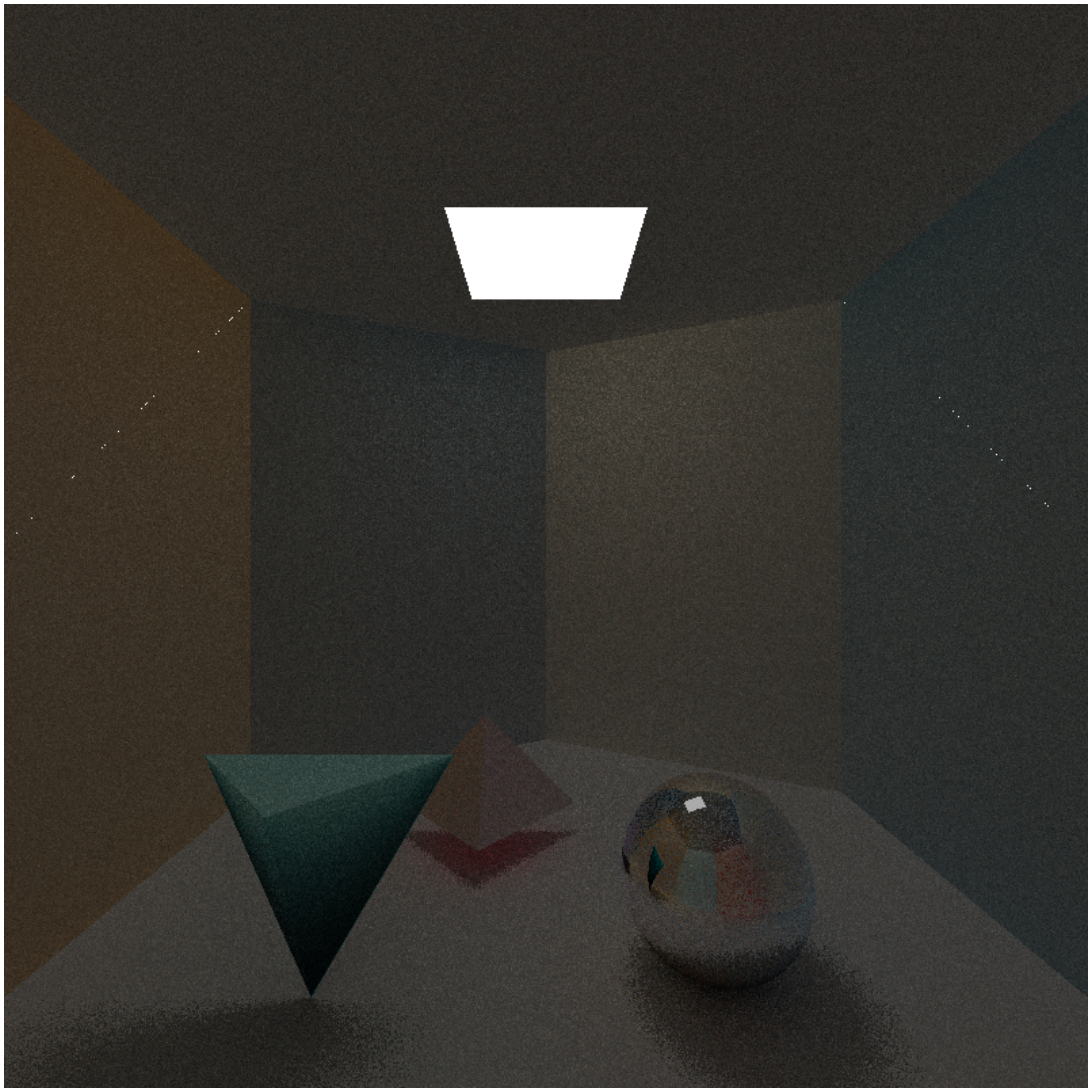


Figure 3.1: Result 1

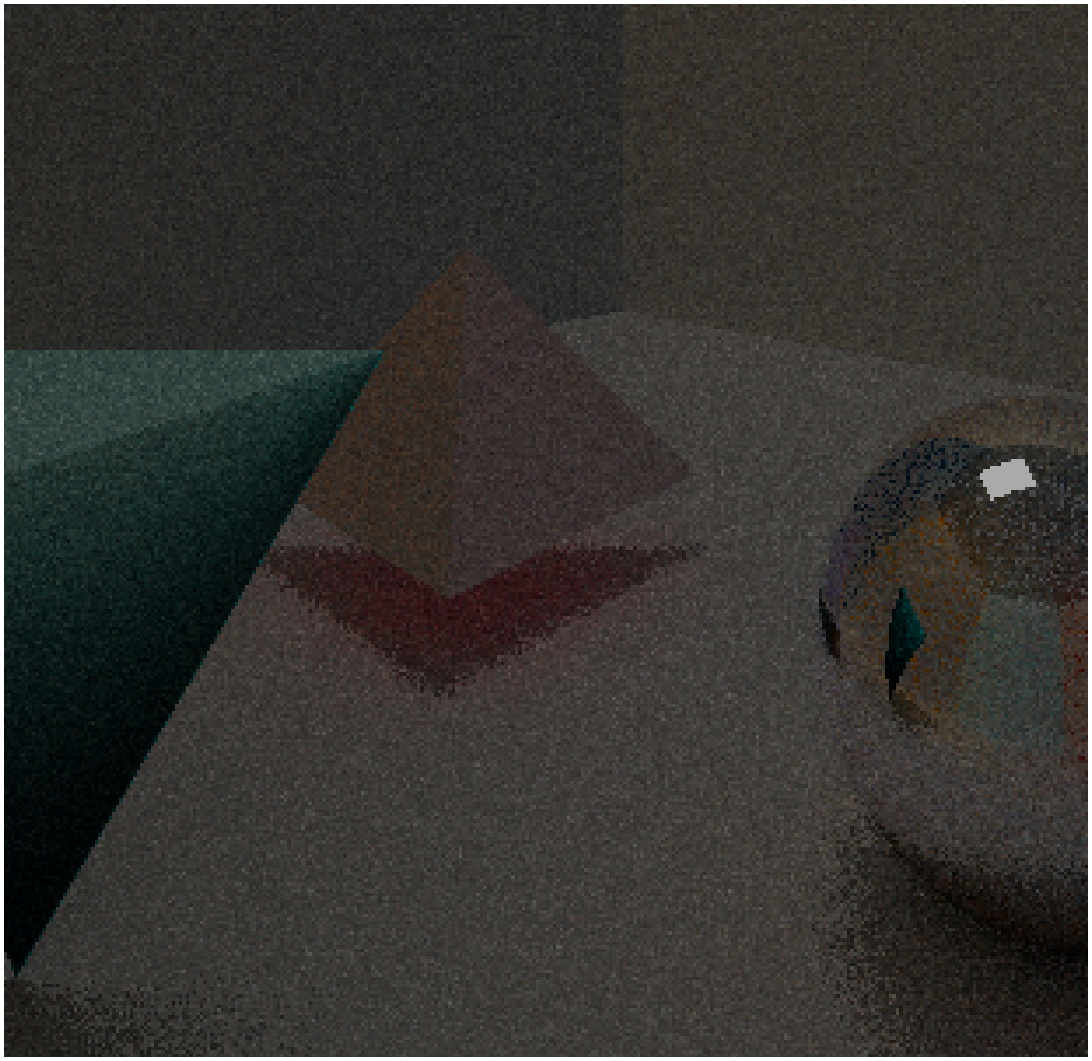


Figure 3.2: Result 1, close-up.

The image has been rendered with 30 scattered rays using the Monte-Carlo Ray Tracing method (2.1). One shadow ray has been cast and the ray tree depth is two which means that the primary ray has two children. The rendering time is 15 hours 3.1. Image 3.2 is a close up of 3.1 and shows the color bleeding and the mirror object up close.

Table 3.1: Result 1

Number of scattered rays	Shadow rays	Ray tree depth	Rendering time (≈ hr)
30	1	2	15

One can perceive the image as somewhat pixelated, see figure 3.2. It is also a little bit dark and grey with not much color. Yet the image contains varying shadows (umbra and penumbra), color bleeding, a perfect mirror and Lambertian surfaces.

3.2 Result 2

The second result, in figure 3.3 has the same content in the scene as the figure in the first result 3.1.

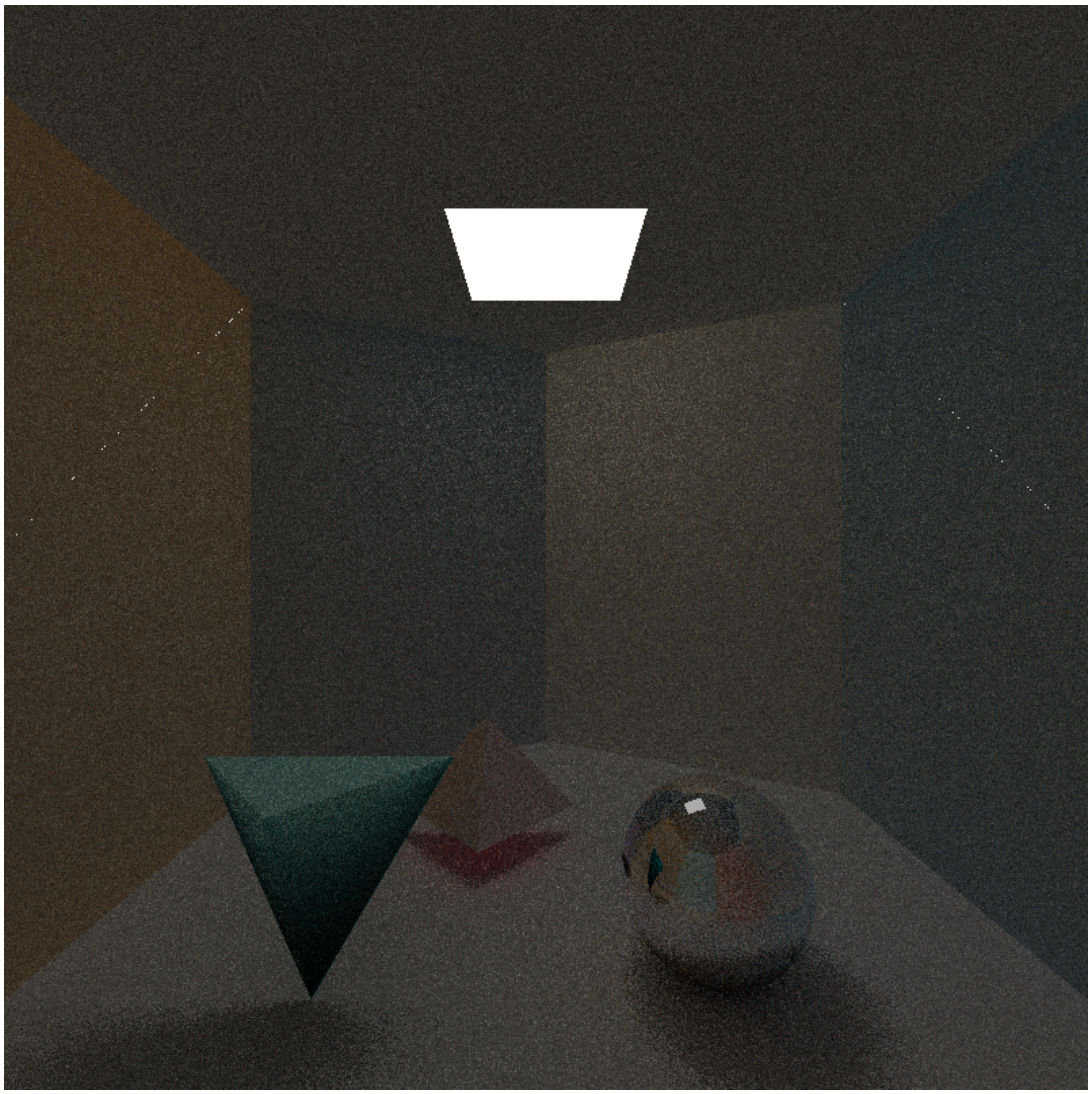


Figure 3.3: Result 2

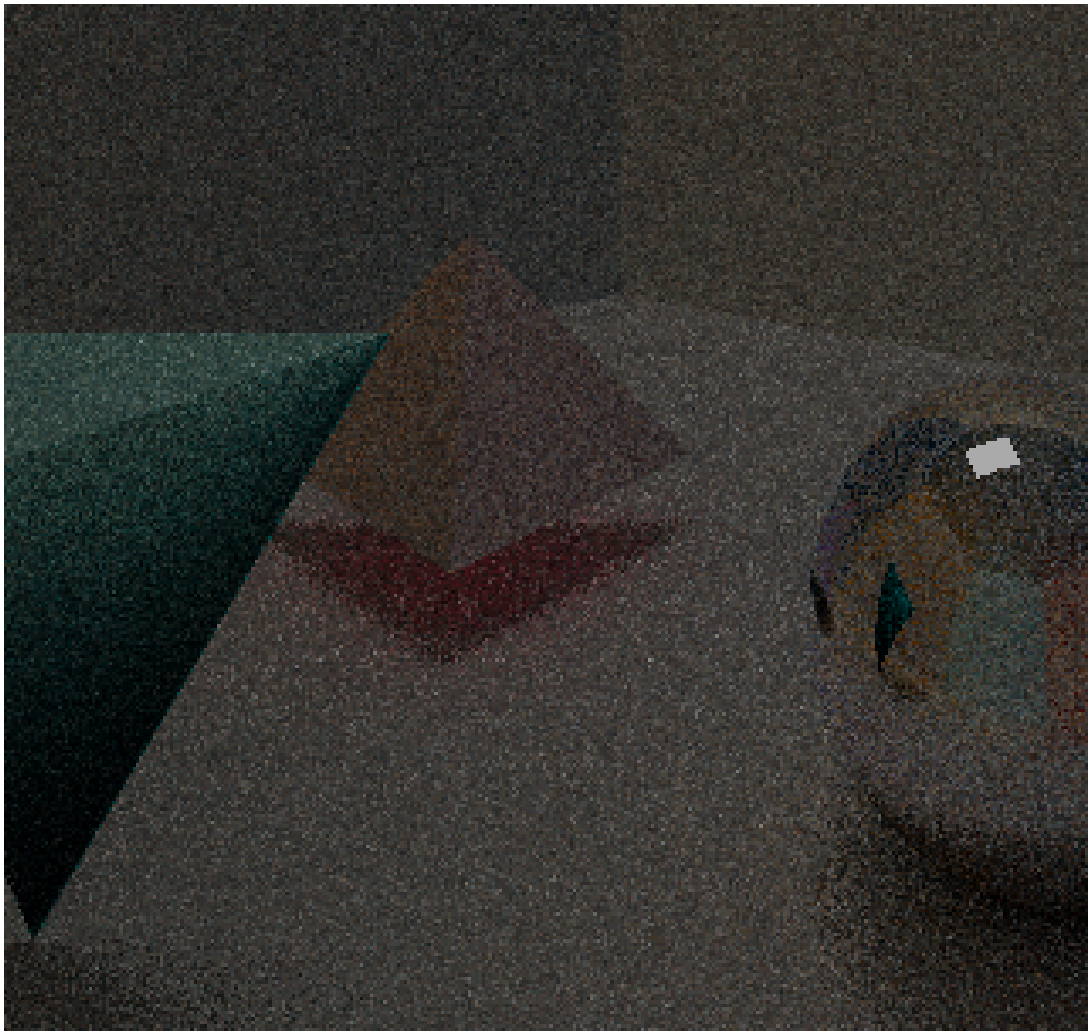


Figure 3.4: Result 2, close-up.

The image was rendered using 11 scattered rays which is fewer than in the first rendering.3.2 Compared to the previous image this is more pixelated and it is even more highlighted in figure 3.4.

Table 3.2: Result 2

Number of scattered rays	Shadow rays	Ray tree depth	Rendering time (≈ hr)
11	1	2	6

3.3 Result 3

The third result in figure 3.5 also has the same content as *Result 1* and *Result 2*, however it has a deeper ray tree depth and the scene-colors becomes dark. No color bleeding can be seen. Due to its low number of scattered rays the image is pixelated and grainy, see figure 3.6.

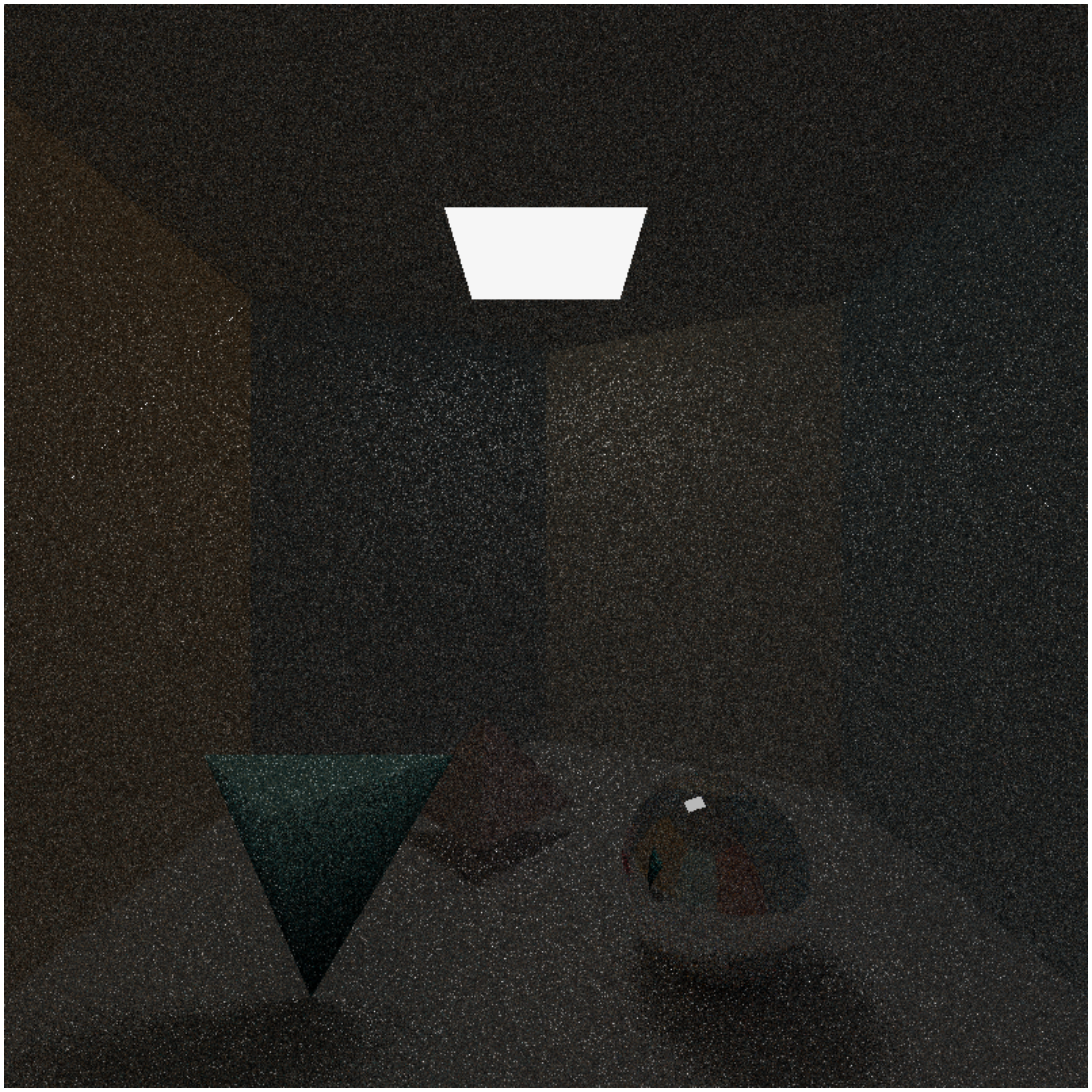


Figure 3.5: Result 3

The white pixel dots on the image are due to whenever a ray hits the light source it add the color value over and over again until the maximum depth has been reach. In order to reduce this contrast one need a higher number of scattered rays to even it out.

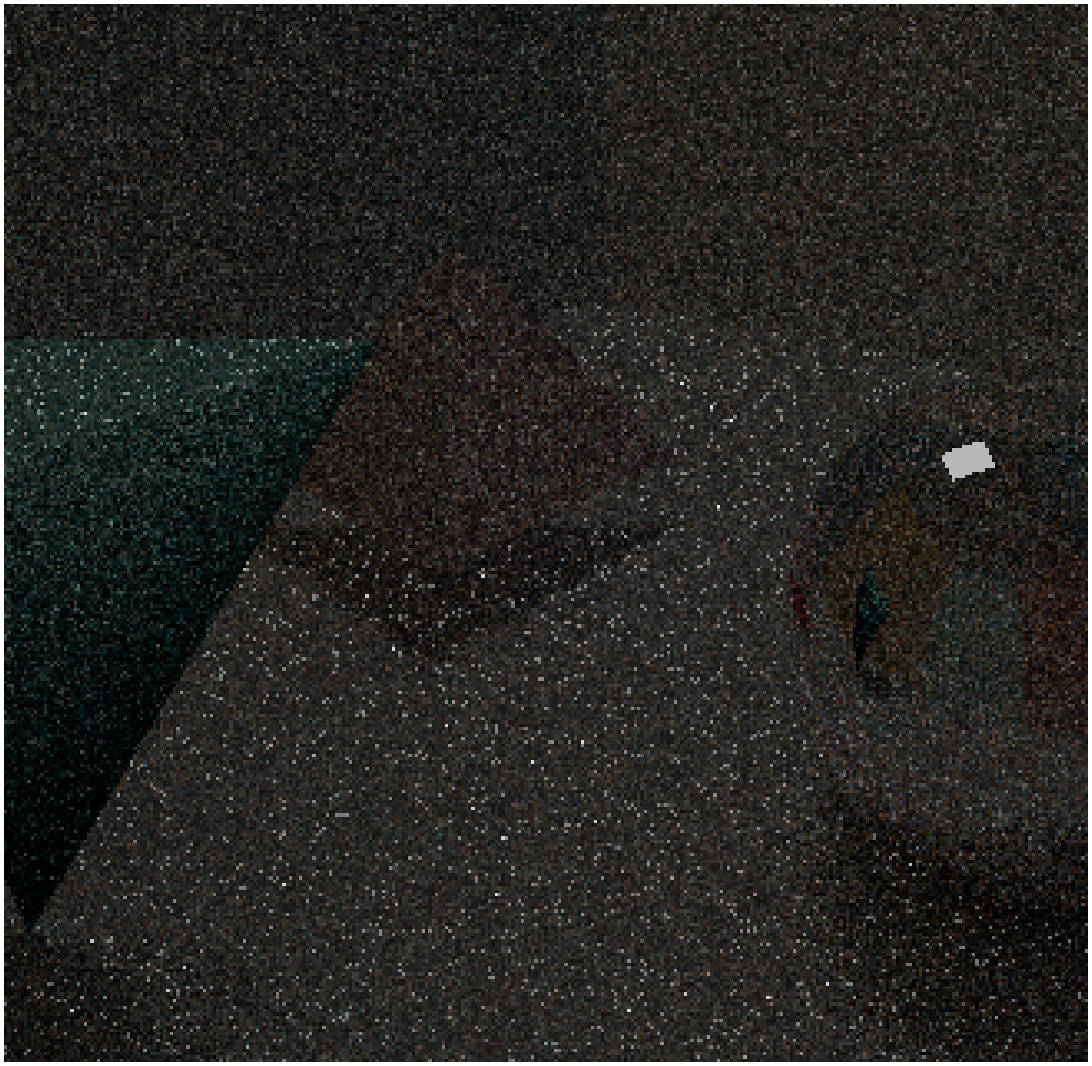


Figure 3.6: Result 3, close-up.

In the table 3.3 it says that the third result was rendered using three scattered rays and had a ray tree depth of three.

Table 3.3: Result 3

Number of scattered rays	Shadow rays	Ray tree depth	Rendering time (≈ hr)
3	1	3	2

3.4 Result 4

The result in figure 3.7 has an image with some more color than figure 3.5 in *Result 3*. However the image is clearly grainy which can be seen in 3.8.

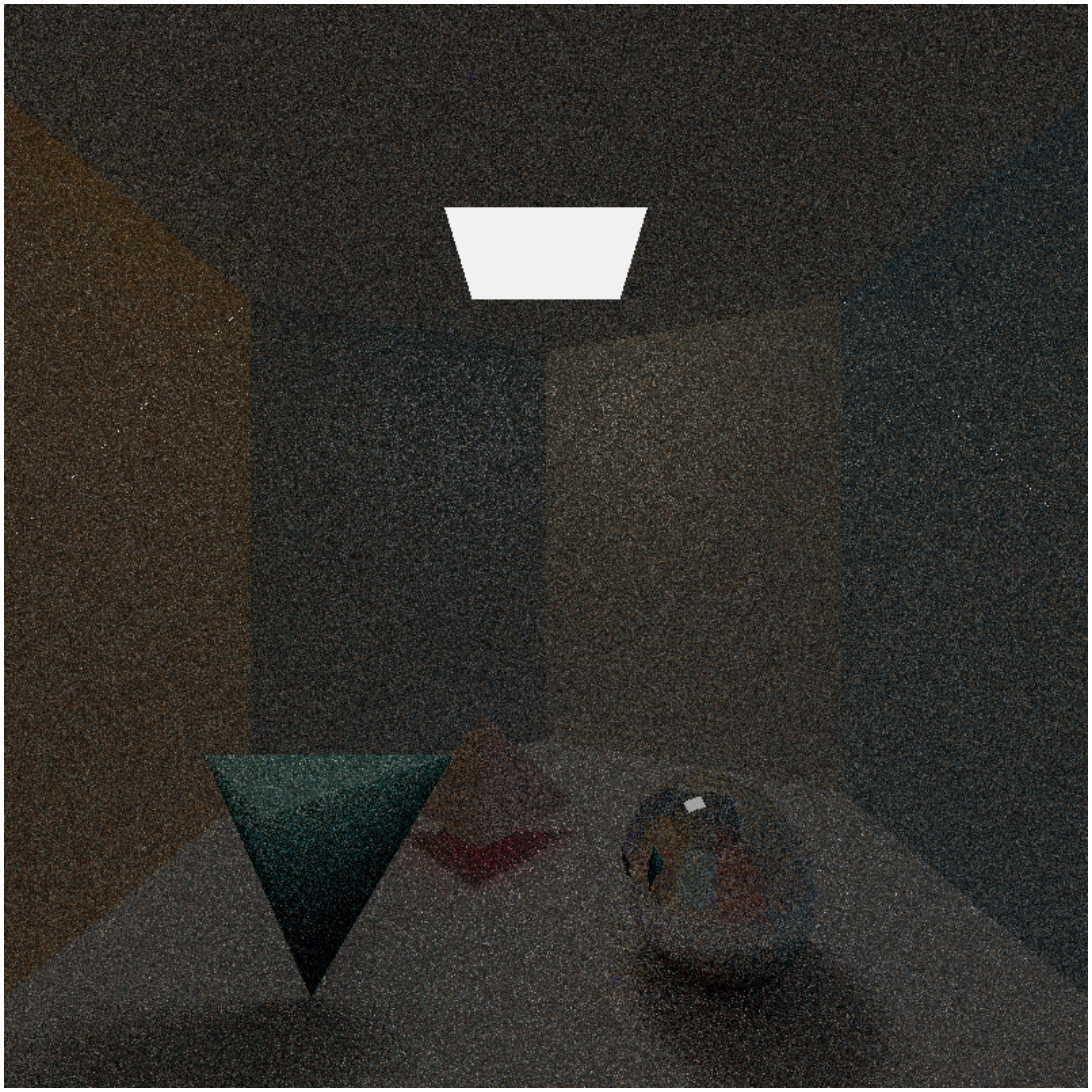


Figure 3.7: Result 4

One can see that the quality of details is low because the point of the red tetrahedron's top becomes grey in some area and the objects edges is perceived as blurry. Some color bleeding is visible, i.e. under the tetrahedron.

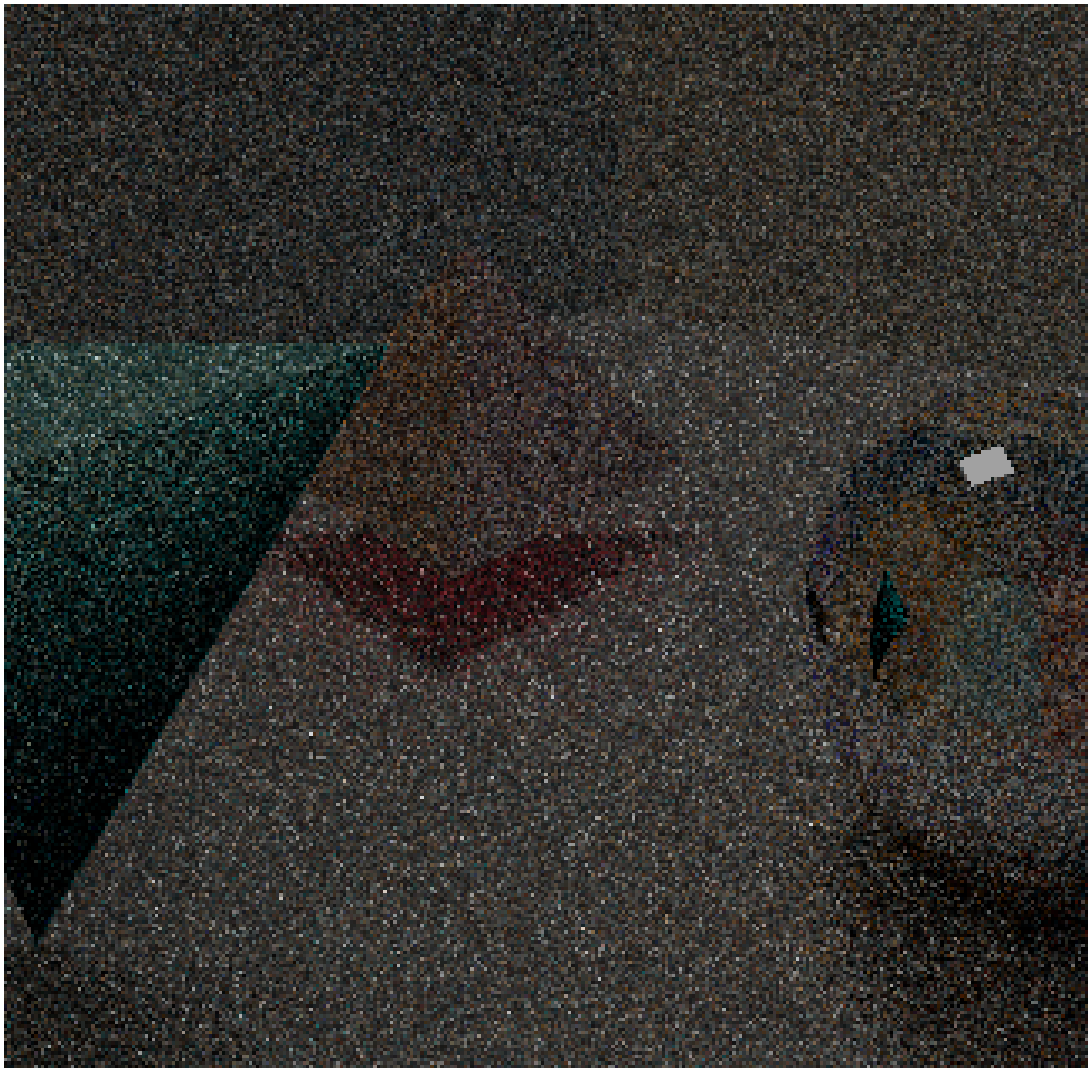


Figure 3.8: Result 4, close-up.

In the table 3.4 it is shown that the fourth result was rendered using three sample rays and a depth of 2 rays.

Table 3.4: Result 4

Number of scattered rays	Shadow rays	Ray tree depth	Rendering time (≈ hr)
3	1	2	0.5

Chapter 4

Analysis and Discussion

It is apparant that a larger number of scattered rays improves the sharpness of the image, see image 3.1 and 3.3 and 3.7, due to a larger collections of data of the scene in form of samples. The quality improves drastically when between three and 11 sample rays comparing to 11 and 30 sample rays where the improvement has slowed down.

The depth of the ray tree worsen the light quality because it turns the hue of the pixel colors into gray which can be seen in 3.5 and comparing with 3.7. It is also more cumbersome to get a similar detailed picture as 3.1 because the rendering time increases a large amount with every depth. This means that a rendered picture with a large amount of scattered rays and a high depth will take an extremely long time complexity to render based on the rendering time from previous results, see table 3.1, 3.2 , 3.3 and 3.4 for comparison.

When comparing result 3.5 and 3.7 we can conclude that a deeper ray depth will make the color bleeding close to unnoticeable. This is an implementation fault in the code and should not reflect on what a Monte-Carlo method should result in.

Chapter 5

Conclusion

A Monte-Carlo ray tracer that can handle mirror surfaces and Lambertian objects has been implmented. The code also simulate indirect light such as correct shadows and color bleeding but an further improvement would be to correct the fault that makes the pixel values grayer when the depth of the ray tree increases. The sharpness of the image improves depending on the number of scattered rays but the rendering time prevents one from using a infinite number.

Bibliography

[1] Scratchapixel. The phong model, introduction to the concepts of shader, reflection models and brdf. *Scratchapixel, Publication: version 3*, 2009 - 2018.

[2] Unity. Global illumination. *Unity Documentation, Publication: 5.6-001N*, 2017-07-12.

[3] Scratchapixel. An overview of the ray-tracing rendering technique. *Scratchapixel, Publication: version 3*, 2009 - 2018.

[4] Mark E. Dieckmann. *Course TNCG15: Advanced Global Illumination and Rendering, Lecture 3 (Radiosity)*. MIT group, ITN, Linköpings Universitet, 2021-08-31.

[5] Henrik Wann Jensen. Global illumination using photon maps, extended version of: Rendering techniques '96 (proceedings of the seventh eurographics workshop on rendering. *Department of Graphical Communication, The Technical University of Denmark*, 1996.

[6] Academic. Lambert's cosine law. *Academic*, 2000-2021.

[7] Mark E. Dieckmann. *Course TNCG15: Advanced Global Illumination and Rendering, Lecture 1 (Introduction)*. MIT group, ITN, Linköpings Universitet, 2021-08-30.

[8] Shree K. Nayar. *BRDF: Bidirectional Reflectance Distribution Function*. First Principles of Computer Vision, School of Engineering, Columbia University, 2021-03-14.

[9] Mark E. Dieckmann. *Course TNCG15: Advanced Global Illumination and Rendering, Lecture 2 (Rendering equation)*. MIT group, ITN, Linköpings Universitet, 2021-08-30.

[10] Shree K. Nayar. *Reflectance Models, Radiometry and Reflectance*. First Principles of Computer Vision, School of Engineering, Columbia University, 2021-03-14.

[11] Mark E. Dieckmann. *Course TNCG15: Advanced Global Illumination and Rendering, Lecture 6 (The Scene)*. MIT group, ITN, Linköpings Universitet, 2021-09-16.