LINKÖPINGS UNIVERSITET

TNCG15

ADVANCED GLOBAL ILLUMINATION AND RENDERING

Monte Carlo Ray Tracer

Authors
JOHAN LINDER: *johli153@student.liu.se*
OSCAR OLSSON: *oscol517@student.liu.se*
Examiner
MARK E. DIECKMANN *mark.e.dieckmann@liu.se*

December 14, 2020

Abstract

This report presents the process of implementing a Monte Carlo ray tracer. The project was made as a part of the course *TNCG15: Global Illumination and Rendering* at Linköping University. The aim of the project was to implement a Monte Carlo ray tracer and with the help from the theory covered in the course lectures, demonstrate deeper knowledge of how to use relevant physical processes such as reflection, light propagation and material properties to render photo realistic images.

The report treats the different aspects of rendering photo-realistic images. In the beginning, the different concepts around Global illumination and ray tracing is introduced and then goes deeper into more details regarding how to implement a renderer using the Monte Carlo integration. The different steps of the process are presented and illustrates how different parts of the integration affects the final image. The results of the implemented renderer is presented in the end of the report were also conclusions and discussions regarding the results are made.

In this implementation project, a Monte Carlo ray tracer is implemented in the programming language *C++*. The renderer is written in C++ from scratch with the help of the library *GLM*, which is a C++ library for mathmatics used in computer graphics.

Chapter 1

Introduction

The field of computer graphics has many different objectives, one which is photo realistic images. The definition of photo realism means that the quality of an image converges to a realistic image with increasing computer power.

Global illumination is a vital step in the process of creating high quality renders. The process of Global illumination is about simulating indirect lighting in terms of light bouncing and color bleeding. To be able to to render these photo realistic images there needs to be an accurate computation of the global illumination in a scene. The global illumination requires a few steps. First, the measurements and data in the scene. This includes all the objects, positions, materials and properties of scene. Properties such as color, type of surface such as diffuse, transparent, reflective. Second, the light transport in the scene. Light transport algorithms describes the simulation of the way light propagates through space while interacting with objects. With the information about the objects and light sources in the scene, the distribution of light in the scene needs to be calculated. Lastly, the actual pixel values of the image is obtained from the values in the scene. Combining the data in the scene and calculations of the light, the values for each pixels can be obtained.

The illumination of a point in the scene can be expressed with the well known rendering equation 1.1.

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_o, \lambda, t)(\omega_i \cdot \mathbf{n}) \mathrm{d}\omega_i \tag{1.1}$$

The equation was introduced by James Kajiya and since the introduction several rendering techniques have been used to try and solve the equation [1]. Since the equation is based on the physical properties of light and the conservation of energy, solving the equation, even if approximate, will result in photo realistic images.

To achieve these photo realistic renderings of images, the *ray tracing* method can be used. It is used to emulate the way light reflects and refracts in the real world, which adds much more realism to a scene, compared to just using static lighting. Ray tracing is a technique for computing the visibility between points.

In the early days of ray tracing, the algorithms traced the rays from the eye into the scene until hitting the object and determined the color of the ray without tracing more rays recursively. The computer scientist John Turner Whitted continued the process with recursive ray tracing in 1980 with the publication of his paper ”An improved illumination model for shaded display” [3]. Instead of just tracing the ray until its first intersection , Turner concluded that when a surface is hit by a ray, three new types of rays can be generated:

- Reflection ray

  - A reflection ray is launched and traced when the intersected object is a ”perfect reflector”, like a mirror. It is traced in the mirror-reflection direction. The reflected ray can intersect with several objects on its path, but it is the closest intersection that will be seen in the reflection.

- Refraction ray
  - When a transparent object is intersected, in addition the the reflection ray, a refraction ray is also traced. This ray works works much like the reflection ray, but since it is traveling through transparent material, the refractive ray can either entering or exiting a material.

- Shadow ray
  - In each intersection point, a shadow ray is launched towards each light source. If the shadow ray intersects with an opaque object on its way from the surface to the light, the surface is in shadow and does not get illuminated by that light.

Figure 1.1 illustrates how the ray is launched from the camera/eye point and intersects with objects.

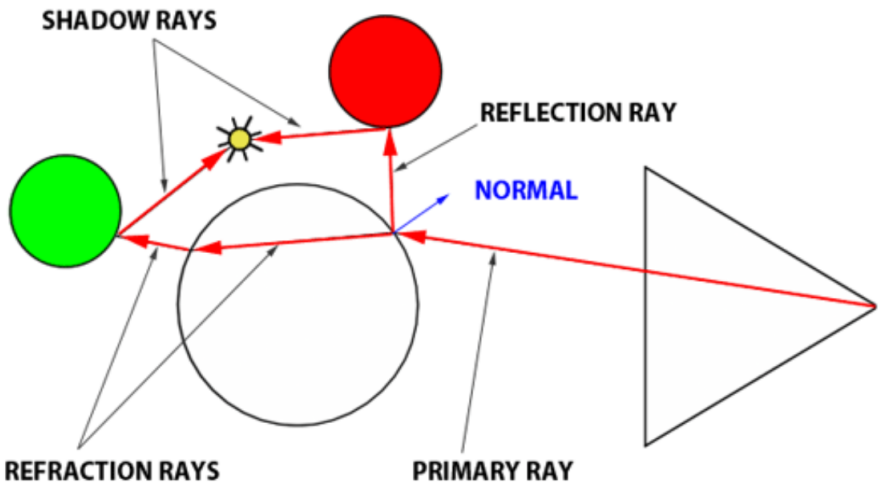


Figure 1.1: *Rays intersecting with objects*

By using ray tracing algorithms in a renderer, realistic simulations of light transport is achieved, which is something other rendering methods like the ”Rasterization method” lacks. The Rasterization method on the other hand, achieves more realistic simulations of geometry.

In Global Illumination, *Radiosity* is a method for calculating diffuse reflections. In this method, all the surfaces in the scene are determined to be Lambertian surfaces without any specular component. In this algorithm, surfaces in the scene are divided into small patches and a ”form factor” is calculated between two patches. This form factor defines how much energy that is transferred from one patch to another. It is the distance between patches that determines the value of the form factor, meaning that patches that are further apart from each other has less energy transferred between them. Also the orientation of the patches matters. Two patches that faces each other transfer more energy than if they are facing away from each other. Also the size of the patch and its visibility affects the form factor. For most common ray tracing methods, calculating diffuse reflections in a scene is the most tine consuming operation, using the Radiosity method for this instead is a good option. Because the intensity values in each patch are independent of the viewing direction, these values can be pre-computed and then used in real-time renderings.

One way to achieve great results is to approximate the rendering equation. Monte Carlo is a well known approximation technique used in mathematical and physical problems when the problem is difficult to solve in other ways. The Monte Carlo technique relies on repeated random sampling to approximate the result for a given problem. What the method basically does is to numerically computes a definite integral. Other algorithms usually evaluate the integrand at a regular grid, while the Monte Carlo technique randomly chooses points at which the integrand is evaluated.

Using the Monte Carlo technique in a ray tracer results in an unbiased, accurate render. However, to achieve realistic results, a great number of rays must be used for the approximation to be usable.

Chapter 2

Background

In this chapter the different parts of the renderer are described and each step of the implementation is described which will lead up to the results presented in Results.

2.1 The Scene

The first step of implementing the renderer, was to draw up a room in the world, where the objects could be encapsulated by walls, a floor and a roof. The room is hexagonal and it is consisting of a roof, a floor and six walls. The view of the room is illustrated in Figure 2.1.

The world of the scene is represented by a triangle mesh. The roof and floor were divided into 4 triangles each, as can be seen in the left part of Figure 2.1. The six rectangular walls were divided into to two triangles for each wall, resulting in a Triangle mesh of 20 triangles for the room. The walls, roof and floor was assigned different colors to make it easier to see the effects of the ray tracer on them. The walls were defined with the material of Lambertian properties. One of the wall were assigned the material of a perfect reflector, acting as a mirror in the room.

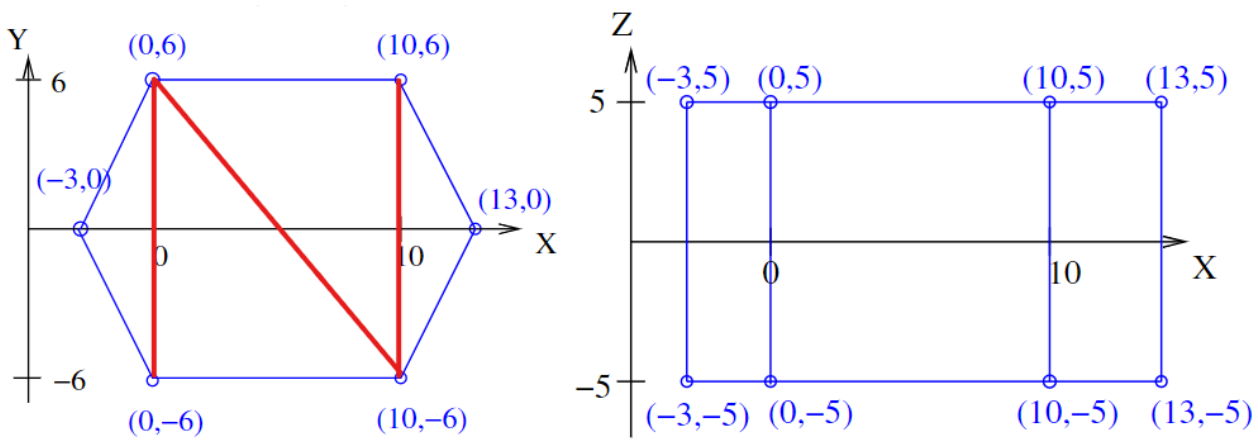


Figure 2.1: *The view of the room from above (left) and from the side (right)*

2.1.1 Camera

The camera is implemented as a fixed system in the world. Two eye positions were setup to make it easy to switch between two viewpoints. In front of the eye point, a camera plane was defined

according to Figure 2.2. The plane is seen from towards the eye point and therefore the left corner of the camera plane is (0,1,1). The eye positions were placed in (-1,0,0) and (-3,0,0).

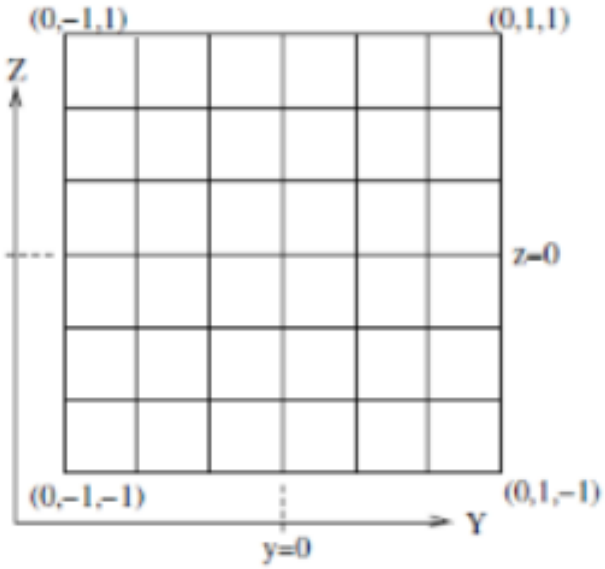


Figure 2.2: *The camera plane seen from towards the eye point*

2.1.2 Lighting

A 2x2 sized rectangular area light was placed in the roof to illuminate the scene. The rectangle was divided into two triangles and added to the triangle list in the Scene class of the code. To easily be able to determine when a ray hit the light source, the triangles were assigned a specific material called ”lightsource”.

2.1.3 Objects

Some different objects were put in the scene to see the effects of the Global Illumination implementation. A tetrahedron were made of triangles and given the properties of a Oren Nayar material. Three implicit spheres were defined in the scene: one with the Lambertian material, one as a transparent object and one as a perfect reflector. The properties of the different materials are described further in Material

2.2 Rendering equation

The rendering equation which the ray tracer attempts to solve can be broken down to more easily understand it. A simplified version without the function arguments can be seen in equation 2.1.

$$L_o = L_e + \int_\Omega f_r L_i(\omega_i \cdot \mathbf{n}) \mathrm{d}\omega_i \tag{2.1}$$

Where L_o is the total radiance at a specific point, L_e is the radiance from light sources, $\int_\Omega ...\mathrm{d}\omega_i$ is an integral over the unit hemisphere Ω, f_r is the bidirectional reflectance distribution function, BRDF, of the point, L_i is the incoming indirect radiance from angle ω_i and $\omega_i \cdot \mathbf{n}$ is the factor for the incident angle of the indirect radiance.

Computing L_e with a ray tracer is straight forward. Shadow rays can be sent from the point to determine if the point is illuminated by any light source in the scene.

Computing the integral $\int_\Omega ...\mathrm{d}\omega_i$ is the complicated part of the equation. To compute the incoming radiance, L_i, from a specific direction, the outgoing radiance from the point which L_i originates from needs to be calculated. This in turn is the rendering equation again, just from another point in the scene. This recursive property of the equation is what makes it difficult to solve and produce realistic looking renderings.

2.3 Intersections

The rays that are launched from the camera through the camera plane and into the scene are being traced. When a ray intersects with an object, the material of the intersected object determines what should happen next. Since the scene consists of both polygonal and implicit objects, the intersection point are defined a bit different, and also how the ray reflects and refracts.

2.3.1 Ray-sphere intersection

To test if a ray is intersecting with a sphere is actually a quite simple form of ray-intersecting test, which is probably why spheres are so often implemented in ray tracer projects, as this. The intersection test can be done basically with two different methods: either a geometrical solution or an analytical solution. The analytical solutions is most of the time a better option since this solution can be reused for different surfaces that are called quadratic surfaces.

A point P can be expressed using the ray equation 2.2

$$P = O + tD \tag{2.2}$$

where O is the origin point of the ray, D is the direction vector of the ray and t is a parameter of the function. t can be either positive or negative, and by varying the value, any point on the line defined by the ray origin and direction can be computed. The whole idea of testing the ray-sphere intersection like this is that also spheres can be defined in an algebraic form. The sphere equation is presented in Equation 2.3

$$x^2 + y^2 + z^2 = r^2 \tag{2.3}$$

where x, y and z are the coordinates of point given in cartesian coordinates, and r is the radius of the sphere with the origin in the center. If we now see the x, y and z as coordinates of a point P we can rewrite Equation 2.3 as Equation 2.4.

$$P^2 - r^2 = 0 \tag{2.4}$$

This is an implicit function, meaning that when the sphere is expressed like this it is an implicit shape or surface. So instead of defining a shape by connecting polygons like how the tetrahedron in the scene was defined, an implicit shape is defined in terms of equations.

Now, Equation 2.2 can substitute P in Equation 2.4 and get Equation 2.5 which can be simplified to

a quadratic function.

$$\begin{aligned} |O + tD|^2 - r^2 = 0 \\ D^2t^2 + 2PDt + O^2 - r^2O = 0 \end{aligned} \tag{2.5}$$

The only unknown variable in 2.5 is t. To get the value of t the roots for the quadratic function is solved like 2.6.

$$t_{1,2} = \frac{-2OD \pm \sqrt{(2OD)^2 - 4D^2(O^2 - r^2)}}{2D^2} \tag{2.6}$$

An intersection is only valid if the value of one of the roots, t_1 or t_2, is positive. A negative root means that intersection point is behind the ray origin relative to the ray direction, which should not be considered an intersection. If there are no roots to the equation there is no value for t along the ray which will intersect the sphere. If the roots are equal, the ray intersect the sphere on its surface.

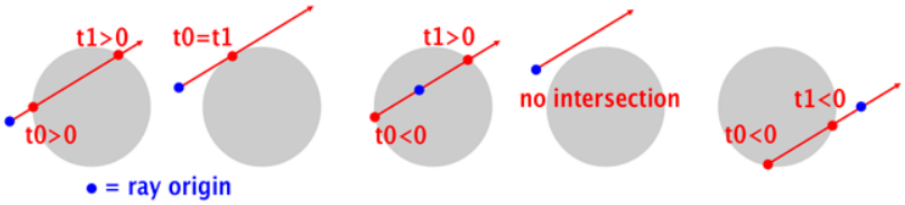


Figure 2.3: *Ray-sphere intersection test*

2.3.2 Ray-triangle intersection

To detect intersections between the rays and the triangles in the scene the *Möller–Trumbore intersection algorithm*, MT, was used [2]. MT is a fast and memory efficient algorithm introduced by Tomas Möller and Ben Trumbore in 1997.

The algorithm uses *barycentric coordinates*. With barycentric coordinates, a point on a plane can be described with two coordinates, (u, v), and two basis vectors for the plane. In the same way, a point on a triangle, T, can be described with two coordinates with the three points of the triangle or two of the edges of the triangle as basis vectors for the plane which the triangle lies in.

$$T(u, v) = (1 - u - v)V_o + uV_1 + vV_2 \tag{2.7}$$

The coordinates must fullfill $u \geq 0$, $v \geq 0$ and $u + v \leq 1$ for the point (u, v) to be on the triangle. $V0, V1, V2$ are the three points of the triangle. The intersection point between a ray and a triangle is then given by equation 2.8.

$$O + tD = (1 - u - v)V_o + uV_1 + vV_2 \tag{2.8}$$

The equation can easily be rearranged into a linear system of equations which than can be solved with the help of linear algebra and Cramer's rule.

2.3.3 Shadow rays

Shadow rays are used to determine if a point is in shadow or not. If the scene uses point lights, this process is simple. A ray is traced from the point towards the position of the light source. If the ray

intersects any object closer than the light source, the point is in shadow. This approach is simple but results in sharp shadows, which is not realistic in most scenes.

To achieve soft shadows, area lights should be used to illuminate the scene. To determine if a point is illuminated by an area light is slightly more complicated than a point light. A point can partially be in shadow and partially illuminated by the light. To determine the amount of illumination a point receives from the light, shadow rays are sent from the point onto random points of the light. The intersection for each individual ray is the same as with point light. The resulting illumination of all the samples is then multiplied with the total area of the light and divided by the number of samples. Greater number of shadow rays sampled gives a better result.

Shadow acne

When computing shadows during ray tracing, a problem that often occurs is so called shadow acne. This is a sort of shadow ray self intersection that is a common problem with for example spheres. It is a visual artefact that appears in the shape of small black dots on the surface. The reason for this is that small numerical errors are introduced due to the fact that numbers can only be represented with a certain precision and therefore sometimes the intersection point is not exactly on or above the surface. Instead it is located slightly below the surface, causing the shadow ray to hit the surface when launched from the intersection point, so instead of hitting another object or the light source the object sort of shadows itself. This is illustrated in the top part of Figure 2.4.

There are some different solutions to this problem and one of the best is to displace the origin of the shadow ray in the direction of the surface normal, to move it slightly above the surface, as can be seen in the bottom part of Figure 2.4. How big the displacement should be is up to the user to decide and can be tweaked on a scene basis. This value is often referred to in ray tracing as shadow bias.

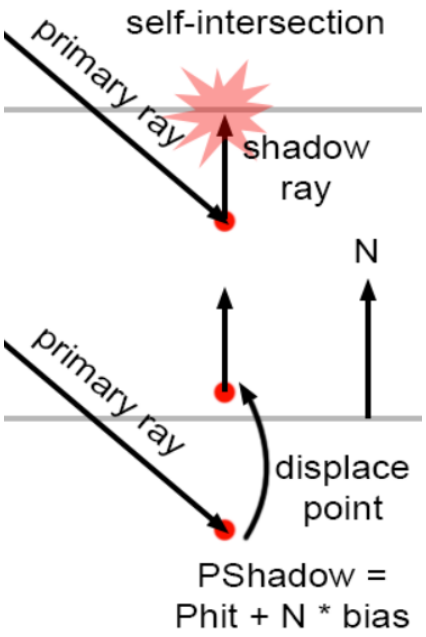


Figure 2.4: *The self-intersection issue (top) and the solution with intersection point displacement (bottom)*

2.4 Aliasing

A common problem in ray tracing is Aliasing that can occur when a surface is undersampled. The sampled information is not enough for making it possible to reconstruct the surface in a correct way. When first setting up the basic ray tracer, rays are shot from the eye through the pixels, one ray for each pixel is launched. The ray continues until it hits a surface and a sample is collected. For each sample, a normal can be computed, which is needed to determine how the ray should be reflected. The problem here is that the sampling information with the position and normal is too small to make it possible to reconstruct the surface in a correct way. This leads to aliasing that becomes very visible on the edges of an object. An example is illustrated in Figure

2.4.1 Anti-aliasing

There are some different ways of solving the problem with aliasing and two methods that give better results are supersampling and ray randomization.

With supersampling the pixels are subdivided into smaller parts like 2x2 or more. One ray is sent through the center of each subpixel. When using more than one ray, aliasing is reduced but not removed completely since the rays are still in order. To reduce the ordering and reduce the aliasing further, ray randomization needs to be introduced. By randomizing each ray-pixel intersection, a random location in each subpixel is received for the rays to be sent through. The price that have to be paid for getting rid of the aliasing is the introduction of noise, which is however less disturbing than the alias effect. Figure 2.5 illustrates a part of the rendered image with and without the use of supersampling and ray randomization.

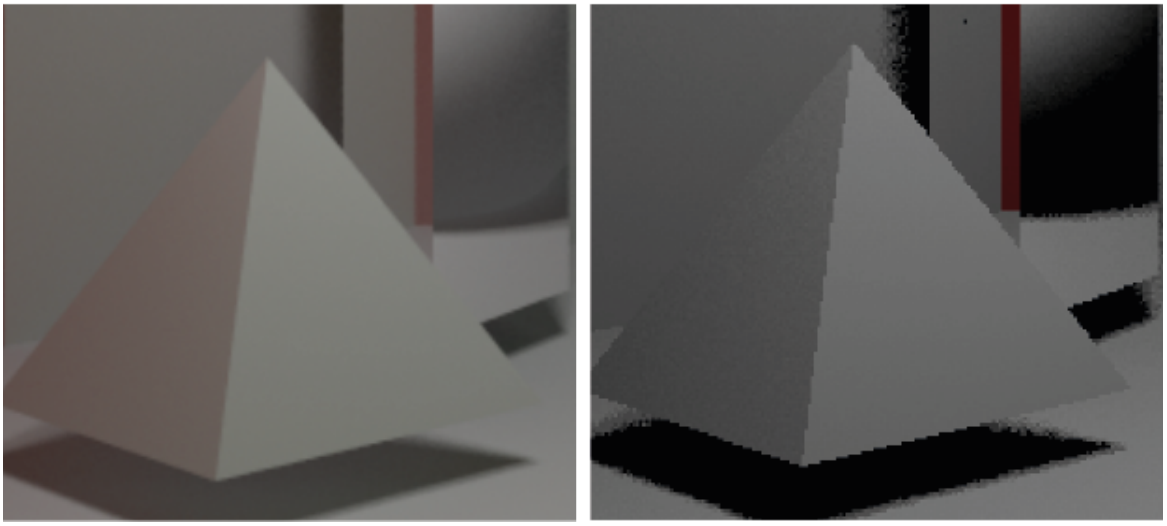


Figure 2.5: *Part of two rendered images with supersampling and ray randomization (left) and without (right).*

2.5 Material

Different materials were applied to the different objects in the scene as described in 2.1.3.The different materials will have different properties to give the objects different looks when light is reflected on them.

2.5.1 Lambertian

One of the diffuse materials supported by the ray tracer is Lambertian reflectance. It is one of the most simple models for diffuse reflective surfaces and follows Lambert's cosine law. When a surface follow the Lambert's law, it appears equally bright, no matter the viewing direction. The intensity of the light and color I_d for a certain point is calculated by equation 2.9:

$$I_d = \mathbf{L} \cdot \mathbf{N}\, C I_L \tag{2.9}$$

where L is a direction vector from the surface to the light, N is the surface normal, C is the color of the surface and I_L is the intensity of the light source.

2.5.2 Oren Nayar

The other diffuse material used in the ray tracer is the Oren-Nayar reflector model. The Oren-Nayar model is actually a generalization of the Lambertian model. Since the Lambertian model does not take the roughness of a surface into account, it is not really a correct approximation of a diffuse surface of an object with a rough texture . The Oren-Nayar model does that better. The model takes a Roughness factor into account. If this factor is set to 0, the model is just like the regular Lambertian model. Equation 2.10 shows how the radiance of the reflected light L_r is calculated.

$$L_r = \frac{\rho}{\pi} \cdot (A + (B \cdot max[0, cos\Phi_i - \Phi_r] \cdot sin\alpha \cdot tan\beta)) \cdot E_0 \tag{2.10}$$

where

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 - 0.33} \tag{2.11}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09} \tag{2.12}$$

$$\alpha = max(\theta_i, \theta_r) \tag{2.13}$$

$$\beta = min(\theta_i, \theta_r) \tag{2.14}$$

and ρ is the albedo of the surface, and σ is the roughness of the surface. If the roughness value σ = 0, A = 1 and B = 0, resulting in a simplification of the Oren Nayar model to the Lambertian. Figure 2.6 illustrates the surface reflection with the variables used in the equations for the Oren Nayar model.

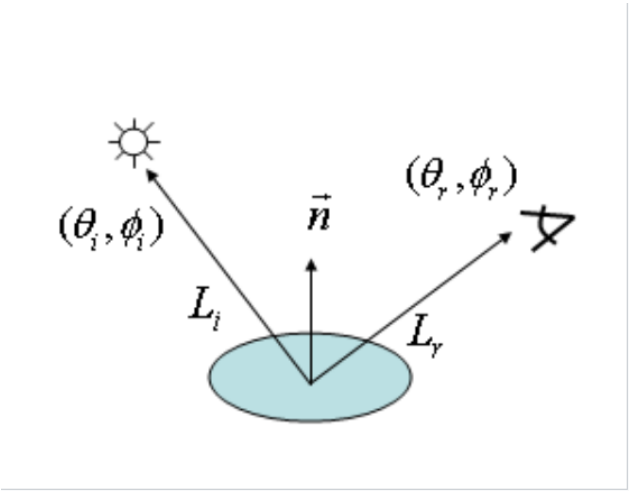


Figure 2.6: *Surface reflection.*

2.5.3 Perfect reflectors

Perfectly reflecting surfaces acts as mirrors in the ray tracer. Rays which intersect with a perfect reflecting material is totally reflected on the surface and than traced in the new direction. The color and light contribution to the pixel which the ray originates from is determined by the first diffuse surface the ray hits. The reflected ray should have the same angle to the surface normal as the incoming ray hitting the surface.

2.5.4 Transparency

Object which are transparent will both reflect and refract rays that intersects the object. Reflected rays are reflected just like perfect reflectors. The direction of the refracted ray is calculated with Snell's law. Snell's law determine the angle between the outgoing ray and the surface normal with the help of the refractive indices of the mediums and the angle between the incoming ray and the surface normal.

How much the reflected and refracted ray each should contribute can be determined by Fresnel's formula. However, Fresnel's formula is computationally expensive if it is implemented correctly. Schlick's approximation is an approximation of Fresnel's formula and is used because it is less expensive to compute. The approximation can be seen in equation 2.15:

$$R(\theta) = R_0 + (1 - R_0)(1 - cos\theta)^5 \tag{2.15}$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{2.16}$$

where θ is the angle between incoming ray and the surface normal, n_1 and n_2 are the refraction indices of the mediums and R_0 is the reflection coefficient for light rays parallel to the surface normal.

With R calculated for the angle θ the distribution between the reflected ray, R, and refracted ray, T, can be calculated with equation 2.17:

$$T = 1 - R \tag{2.17}$$

since the total contribution of the reflected and refracted ray must add up to 1.

2.6 Monte Carlo integration

Computing the indirect lighting in regular ray tracing can be difficult since the light can bounce all around the room and contribute to the illumination of any point. When light hits an diffuse object, the surface distributes the light in all direction. Monte Carlo integration solves this by sampling random rays over the hemisphere of a point. Instead of sending rays in all directions to determine the indirect lighting contribution, only a specific number of rays, samples, are sent from the point. This approximates the indirect lighting contribution from the scene onto the point.

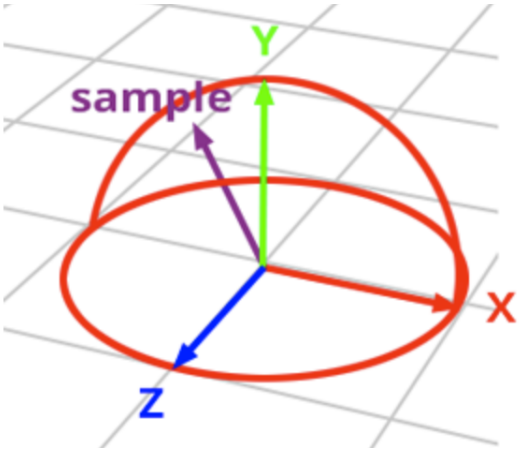


Figure 2.7: *Ray sample over the hemisphere of a point.*

Each new ray sent from the sampling of the hemisphere is treated in the same way. When the new ray hits a surface, russian roulette will decide if the hemisphere of this point should be sampled as well. This is part of the unbiased approach of the Monte Carlo technique.

Calculating accurate indirect lighting will result in effects like smoother shadows and color bleeding. Since the indirect light sampling is randomized the result will be better with more samples. When the number of random samples increases, the approximation will converge towards the real integral value and a higher quality image.

Chapter 3

Results

In this chapter different results of the render is presented. To better illustrate the effect of the implemented methods some part-results is included so that the process of the development of this Monte Carlo ray tracer is demonstrated. Larger scale versions of the presented images can be found in Appendix A

Table 3.1 demonstrates different values for the renders performed and the resulting images that follows.

Table 3.1: Some values used for rendering the images presented in this chapter

Figure	**Samples per pixel**	**Render time (s)**
3.1	1(center)	32 (unoptimized)
3.2	1	11
3.3	100	991 / 16.5min
3.4	400	4169 / 69.5 min

Figure 3.1 illustrates the rendering result without the Monte Carlo Integration. As can be seen the image looks quite good but struggles with some aliasing. Also the shadows from the objects are very grainy and not so smooth. No color bleeding effect is present.

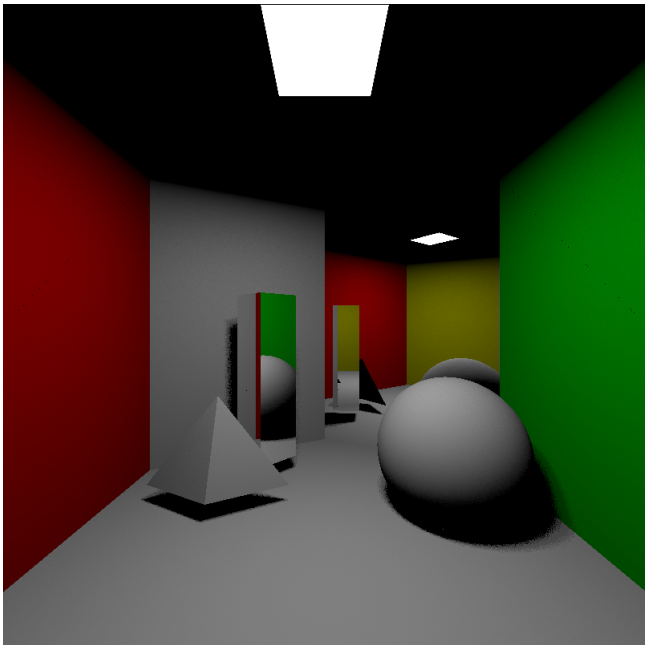


Figure 3.1: *Rendering without the Monte Carlo integration*

Figure 3.2 illustrates the result with the Monte Carlo integration. Now some color bleeding can be seen on the sphere and the tetrahedron. This image is rendered with only one sample per pixel, which results in a quite noisy image.

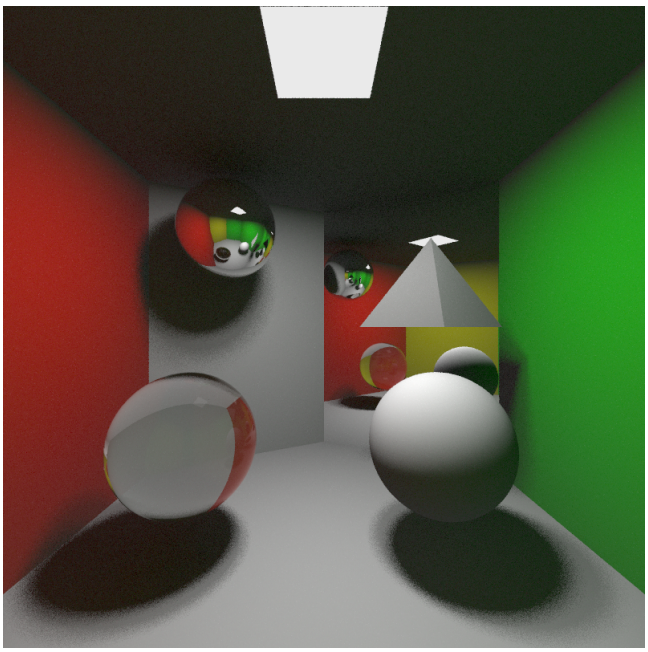


Figure 3.2: *Rendering with Monte Carlo integration and one sample per pixel*

Figure 3.3 is a render result where a higher number of samples per pixel were performed (100), which results in a very clear and smooth looking image. The noise seen in Figure 3.2 is no longer visible.

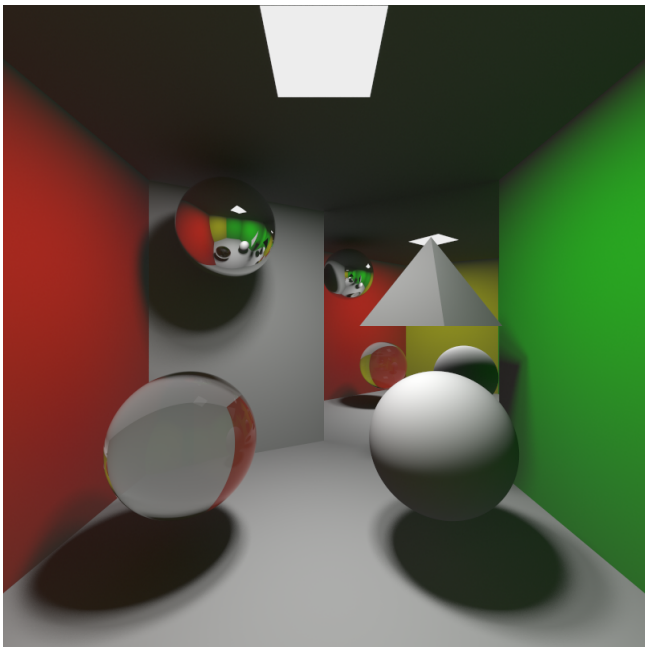


Figure 3.3: *Rendering with Monte Carlo integration and 100 samples per pixel*

In Figure 3.4 a image is rendered with 400 samples per pixel. Here the differences are no longer that visible compare to Figure 3.3. However it can be seen that the blacks in the image is better, of course on the cost of increasing the render time by a factor 4.

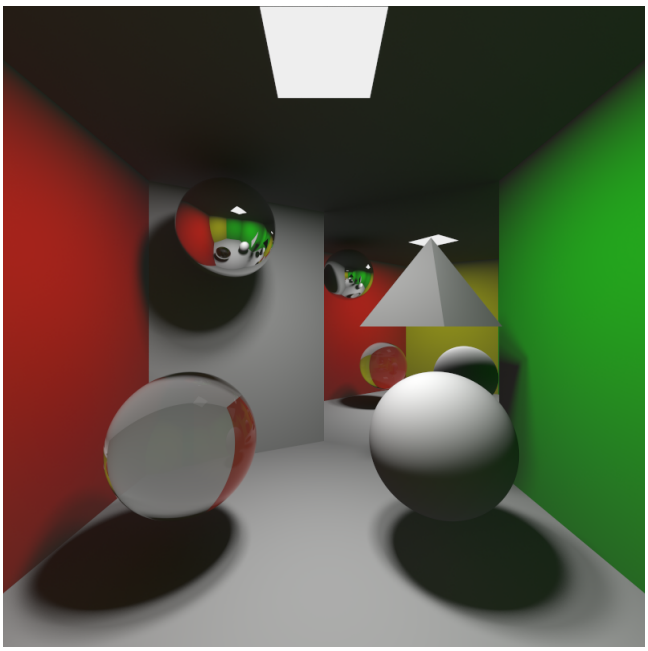
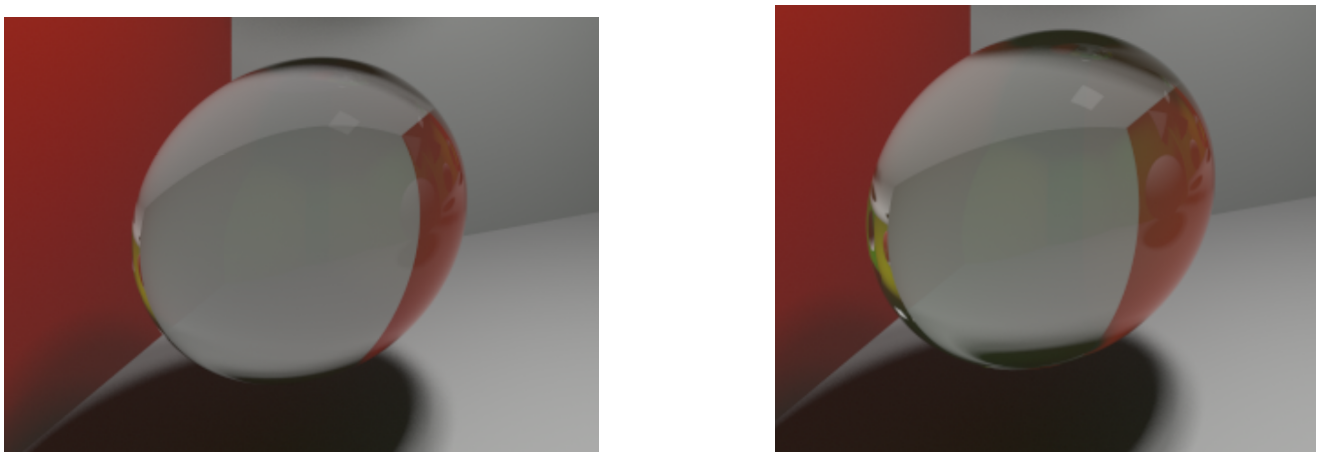


Figure 3.4: *Rendering with Monte Carlo integration and 400 samples per pixel*

To test how the refraction index affects the look of a transparent object two images were rendered with 1.5 and 1.75 as refraction index. Some difference can be seen in the right image of Figure 3.5, where the mirroring effect is a bit stronger, as well as the rays bends in a slightly different way.



(a) *Refraction index 1.5 (Glass)* (b) *Refraction index 1.75(Flint Glass)*

Figure 3.5: *Two transparent spheres with different refraction indexes*

Even though Photon Mapping was not implemented in this project, a test was made for letting shadow rays pass through the transparent object. The result can be seen in Figure 3.6.

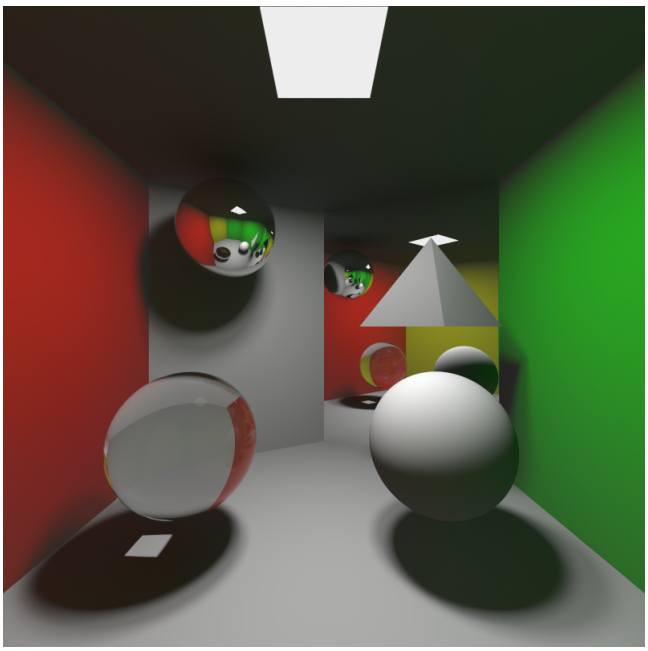
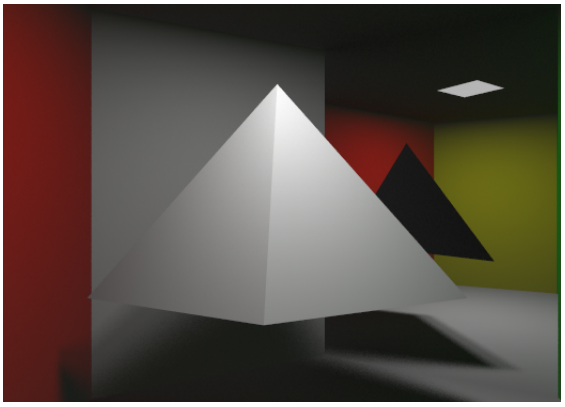


Figure 3.6: *Test for rendering caustics by letting shadow rays pass through the transparent sphere*

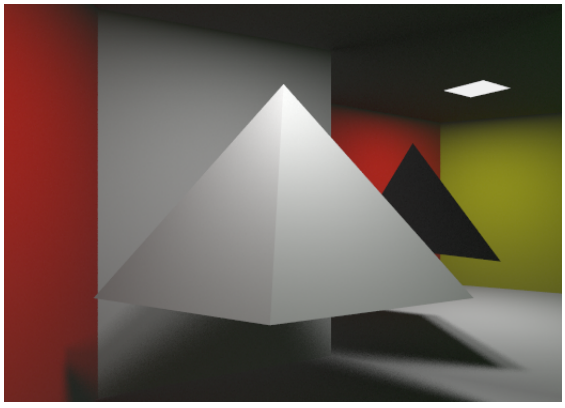
The Oren Nayar model as described in Oren Nayar takes a roughness value into account and in Figure 3.7, the effect of different roughness values are illustrated on a Tetrahedron with a Oren Nayar material. The differences are not really that large, although it can be seen when comparing 3.7a and 3.7d, that the light is more evenly scattered over the object.



(a) *Roughness* $\sigma = 0$

(b) *Roughness* $\sigma = 0.3$

(c) *Roughness* $\sigma = 0.7$



(d) *Roughness* $\sigma = \inf$

Figure 3.7: *Objects with Oren Nayar model and different roughness values*

Chapter 4

Discussion

The ray tracer can produce some satisfying results considering the simplicity of the ray tracer at this stage. The difference of image quality between a Whitted ray tracer and a ray tracer using Monte Carlo integration are great, which is illustrated when comparing the results in Figure 3.1 and Figure 3.3. The Monte Carlo integration provides a good approach to achieve the effects of indirect lighting with color bleeding between objects which greatly increases the feeling of realism to an image.

The ray tracer can render diffuse surfaces with either Lambartian or Oren-Nayar reflectance models. There are other types of reflectance models which could enhance the quality of the image. For example a specular reflection model like Phong, which would enhance highlights reflected on smooth, glossy surfaces.

One of the hardest effects to achieve with a ray tracer is caustics. Computing accurate caustics with only a ray tracer is a tough task since the ray tracer originates from a viewpoint(eye or camera), while caustics originates from the light sources in the scene. Precalculating the caustics of a scene with the help of photon mapping is a great way to complement the ray tracer, since it is a much more efficient way than to compute the caustics of like in Figure 3.6.

The photo realism of the images produced with a Monte Carlo ray tracer increases with the number of samples, therefor optimizations of any renderer should be of high priority. In the simple scene used in this report, computing 400 samples per pixel is done in approximately 70 minutes which is acceptable. However, introducing a more complex scene would certainly increase the rendering time. At this moment the objects in the scene is simply stored in an array and ray intersections is calculated by traversing every scene object and calculating which of the intersection objects is closest to the ray origin. Determine what object a ray intersects is therefor computed in $O(n)$ time complexity, for each ray. Storing the geometries of the scene in a more suitable data structure, such as an octree, would improve the time complexity to $O(log\ n)$ [4]. An octree is used to partition 3D-space by recursively subdividing the space into octants from their 3D-coordinates. This makes it possible to search and traverse a scene objects according to their positions in 3D-space.

It can be concluded that working with implementing a Monte Carlo ray tracer in a project like this, improves the understanding of the importance of Global Illumination in 3D-graphics to achieve photo-realistic render of scenes. It also shows how fast a 3D-scene can become heavy to compute, even for modern computers. Therefore optimizations needs to be made to save computer power were it is possible to focus that power on the most important part of a scene. This must always be in the back of the mind, of someone trying to implement tools for working with 3D-graphics.

References

[1] James T. Kajiya, *The rendering equation*, SIGGRAPH Volume 20, Number 4, 1986, Retrieved: 2020-10-29.
http://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf

[2] Tomas Möller and Ben Trumbore, *Fast, minimum storage ray-triangle intersection*, 1997, Retrieved: 2020-10-28.
https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

[3] J.Turner Whitted, *An Improved Illumination Model for Shaded Display*, Communications of the ACM, 1980. Retrieved: 2020-10-28
https://dl.acm.org/doi/pdf/10.1145/1198555.1198743?casa_token=sv5r76H3BkkAAAAA:PE3jLMpB-fRi_-NIdGlqH_pTiZjBu8lamxE-SAjDKjal39wY_w2bkJvBcWTQ-ZWTra3ErG-ZVP8d

[4] Donald Meagher, *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*, Octree corp, 1980. Retrieved: 2020-12-11
https://www.researchgate.net/publication/238720460_Octree_Encoding_A_New_Technique_for_the_Representation_Manipulation_and_Display_of_Arbitrary_3-D_Objects_by_Computer

Appendix A

Rendered images

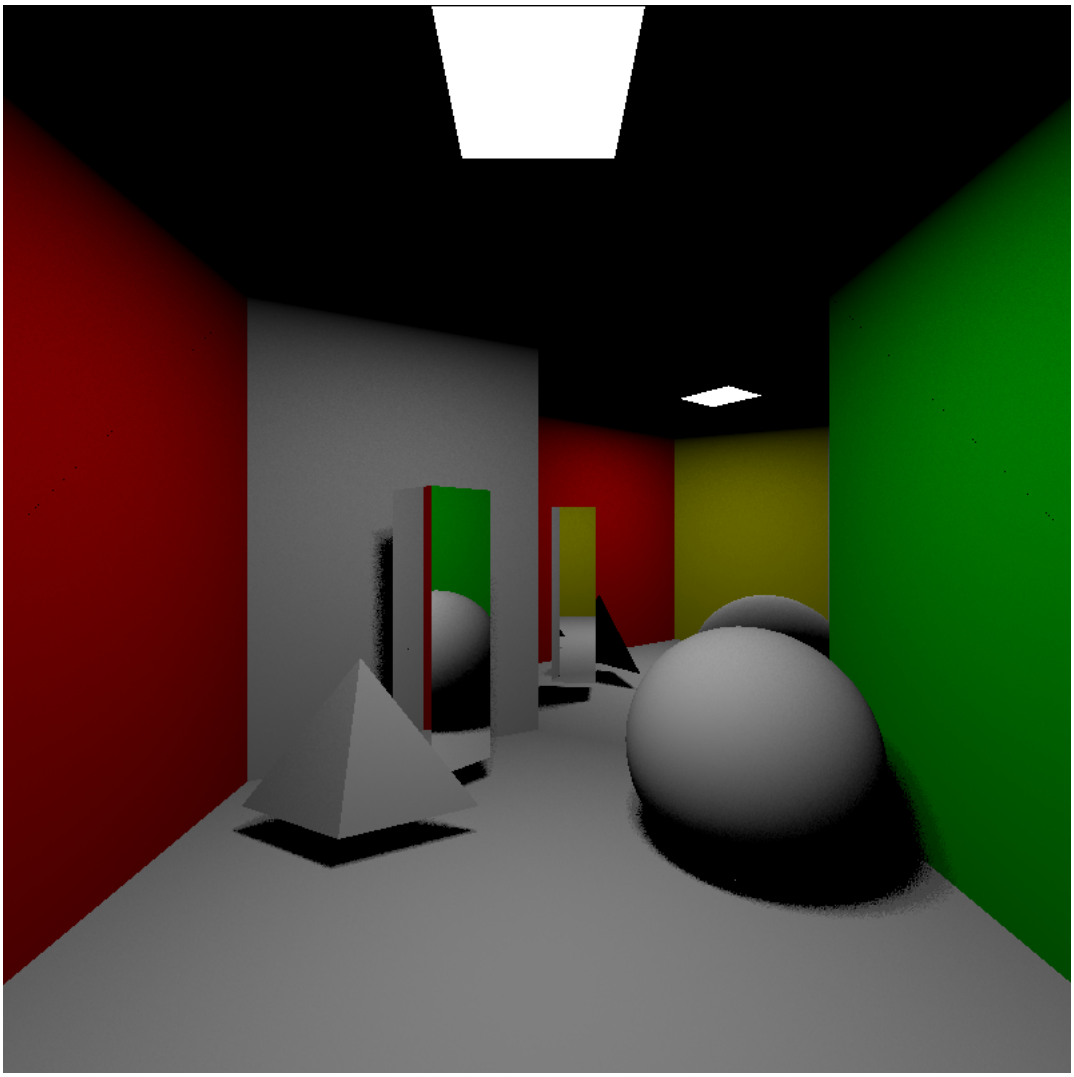


Figure A.1: *Rendering without the Monte Carlo integration.*

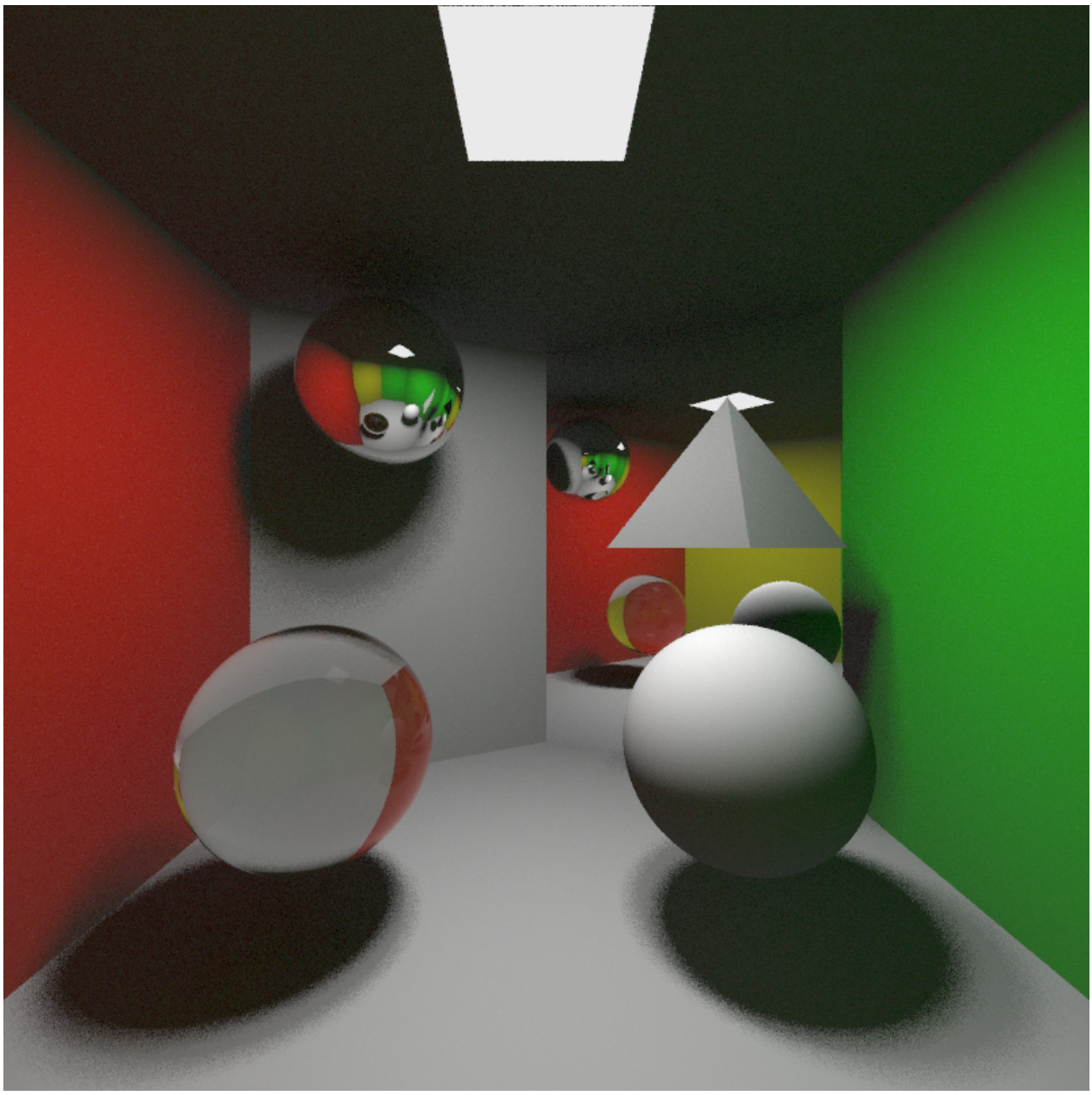


Figure A.2: *Rendering with Monte Carlo integration with one sample per pixel.*

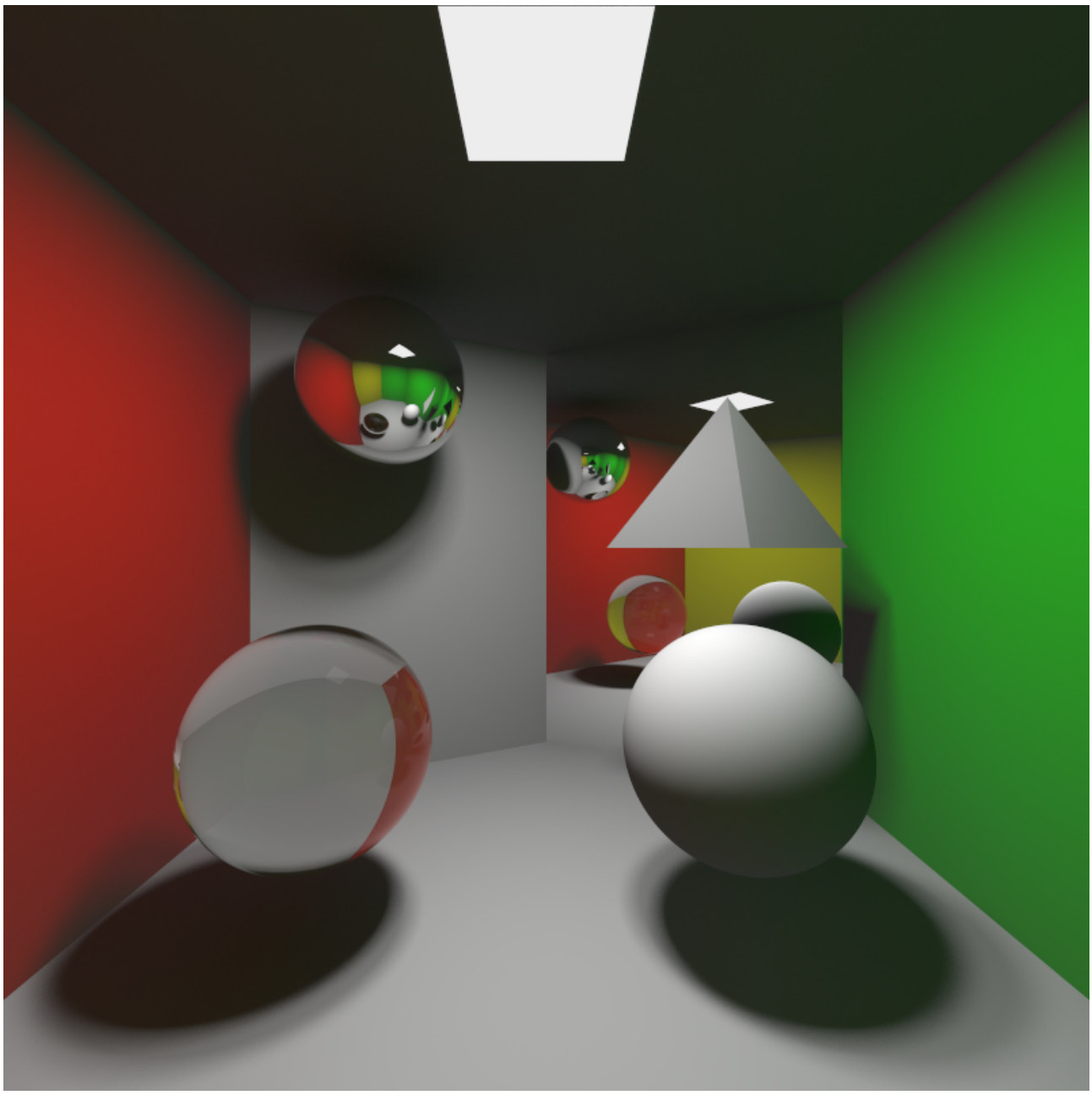


Figure A.3: *Rendering with Monte Carlo integration with 100 sample per pixel.*

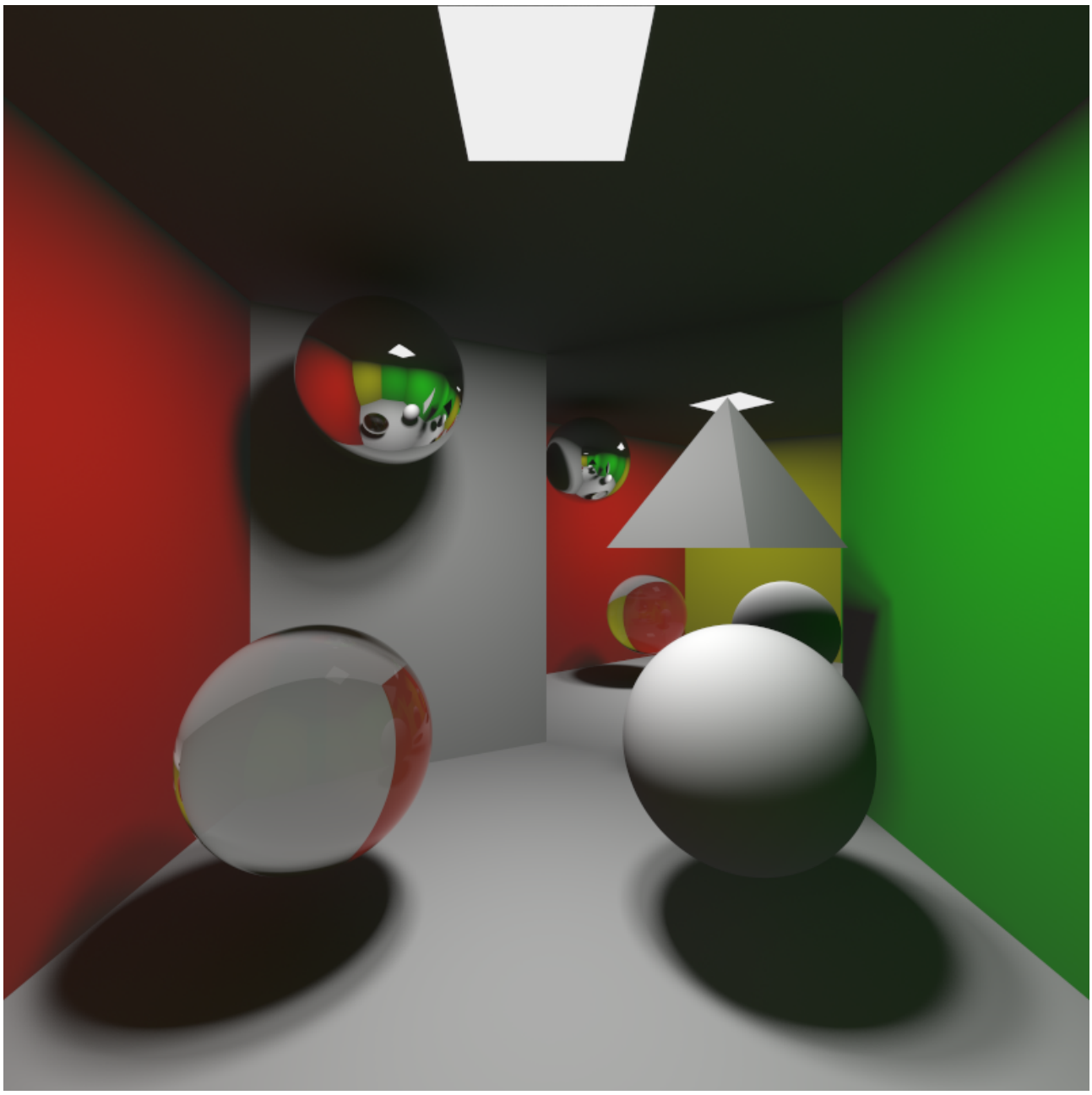


Figure A.4: *Rendering with Monte Carlo integration with 400 samples per pixel*