Monte Carlo Ray Tracer

TNCG15 - Advanced Global Illumination and Rendering

Chris Paulusson
Adam Morén

November 17, 2020

Abstract

This report presents an implementation of a Monte Carlo ray tracer for the course TNCG15 - Global Illumination and Rendering at Linköping University. It covers the underlying physics and theories behind global illumination, including ray tracing, ray-object intersections, bidirectional reflectance distribution functions, and indirect illumination. The result is a scene with objects of different materials and shapes, with features like color bleeding, soft shadows, reflections, and refractions. The materials are perfect reflectors, transparent, and diffuse (Lambertian and Oren Nayar). Finally, a discussion is presented, covering an analysis of the results as well as ideas for improvements and future work.

Contents

Chapter 1

Introduction

Conventional 3D rendering has since the beginning of the 1990s used the process of *rasterization* to convert 3D models into pixels on a 2D screen, also known as *image synthesis*. The pixels are often processed or "shaded" to approximate local behaviour of light on the 3D object's surface. However, to produce photo-realistic images with rendering, it is vital to not only simulate the effect of local illumination but also consider the impact of *global illumination*.

Global illumination, or indirect illumination, add realistic lightning within a scene with algorithms that take into account not only light that comes directly from a light source, but also the reflected light from other objects within the scene. These algorithms have, in recent years, become more and more specialized to effectively simulate global illumination and most advanced software today often combines two or more techniques to obtain good results at reasonable costs. What is typical for global illumination algorithms is that they are numerical approximations that attempt to solve the so-called *rendering equation*.

In 1986, James Kajiya introduced the rendering equation [6]. The rendering equation describes the complete transport of light within an environment as emitted *radiance* and reflected radiance, see equation 1.1. L_s is the total surface radiance in point x. L_e is the radiance emitted by the surface itself (needs no further calculation), L_i is the incoming radiance in direction Ψ_i and f_r is the *bidirectional reflectance distribution function* (BRDF) of the surface. The equation integrates over the *hemisphere* Ω of x, which is the infinite number of incoming directions.

$$L_s(x, \Psi_r) = L_e(x, \Psi_r) + \int_\Omega f_r(x, \Psi_i, \Psi_r) L_i(x, \Psi_i) cos\theta_i dw_i \tag{1.1}$$

A common technique to simulate the physical behaviour of light within virtual 3D environments is *Ray tracing*. With ray tracing, it is possible to calculate pixel colors of a 2D screen by tracing the light paths from the eye of the viewer, through the scene, and back to its original light source. Through this path, the light might be reflected, refracted, or blocked, all in which will result in different pixel values of the 2D image[3]. The idea of tracing the light from the eye to the light source is called *backward tracing*. Backward

tracing saves a lot of computational power compared to tracing the path from the light source to the eye, known as *forward tracing*. The reason for this is that the majority of the rays in forward tracing would not reach the viewer, making the method wasteful.

One of the most classical algorithms using the ray tracing technique is the *whitted ray tracing* algorithm presented by Turner Whitted, already back in 1980 [9]. With his paper, Whitted proposed a solution to simulating complex reflections and refractions in a scene, something that had never been done before. The idea of Whitted ray tracing is to generate new rays if the intersected objects are either mirror-like surfaces or transparent surfaces. If a mirror-like object is intersected at point A, a *reflection ray,* is traced from that point to get information from the surface of another object B. A similar approach is used if the intersected object is a transparent surface. In that case, two new rays are cast, one reflection ray and one refraction ray from that intersection point of the object. If the rays hit diffuse surfaces, the color of these are calculated using an arbitrary illumination model and with *shadow rays* that check if the light source directly illuminates the object or if another object blocks it. You might ask yourself what happens if both point A and point B in the scene are mirror-like surfaces. In that situation, we could get an infinite amount of reflection rays. That is why Whitted ray tracing also has a necessary stopping condition that is to stop when the depth of the reflection rays reach a certain threshold. The Whitted ray tracing method was a significant improvement. However, simulating certain effects such as soft shadows, *color bleeding* and *caustics* remained a problem.

When Kajiya released the rendering equation, eight years later, he extended Whitted ray tracing with *Monte Carlo integration*, also known as Monte Carlo Ray Tracing. This global illumination algorithm tries to solve the rendering equation by approximating the integral of the equation using the Monte Carlo estimation. A Monte Carlo ray tracer is what we implement in this report, and we will see in great detail how it solves both color bleeding and soft shadows, as well as reflection and refraction.

Monte Carlo Ray tracing approximates the integral of the rendering equation by averaging the radiance of multiple samples. The samples are secondary rays with random directions over the hemisphere that are traced in these directions into the scene. The monte Carlo estimator is good in the sense that it converges in probability to the exact solution (the analytical solution of the integral). Increasing the number of samples also decreases the level of variance (noise) of our estimation. However, this comes with a cost since the render time increases with the number of samples used. The monte Carlo ray tracer is thus known to be slow and noisy. Another issue with the monte Carlo ray tracer is that it still has problems simulating caustics in an efficient matter.

The third and last method that we introduce in this introduction is *Photon mapping* from Wann Jensen's paper *Global Illumination using Photon Maps* [5]. Jensen released this paper in 1996, and it covers a method that combines previously known techniques with the use of photon maps. The method is a two-pass algorithm. In the first pass, packets of energy (flux) are emitted from the scene's light sources into the scene and stored in photon maps. Two different photon maps are used. One global photon map which, is a rough approximation of the light/flux within the scene, and one caustics photon map, used only to store photons corresponding to caustics. The second pass is rendering. During the rendering process, the global photon map is used to optimize the Monte Carlo

ray tracer, by computing optimized sampling directions and by eliminating unnecessary calculations by using approximations from the photon map instead. Caustics are easily rendered by using the caustic photon map. We will not go into more detail of Photon Mapping than this, and the algorithm will not be implemented in this report. However, we consider it essential to cover how photon mapping can be used to improve the Monte Carlo ray tracer further.

A commonly used 3D test model for realistic rendering is the Cornell Box [4]. The basic environment of the Cornell box consists of one light source in the centre of a white ceiling, a green right wall, a red left wall, a white back wall and a white floor. Objects with mirror-like surfaces and transparent surfaces are often added to the scene to test that physical properties like reflection and refractions are rendered correctly. Another physical property that is often tested is color bleeding. If indirect illumination and color bleeding has been implemented correctly, parts of the diffuse white surfaces should appear slightly green or red.

The scene that will be using to test our implementation of the Monte Carlo ray tracer in this report is not the standard Cornell box. Instead, a similar scene, presented by Mark E Dieckmann, will be used [2]. This scene is a hexagonal room with diffuse walls of different colors and with white floor and ceiling. The scene setup can be seen in Figure 1.1.

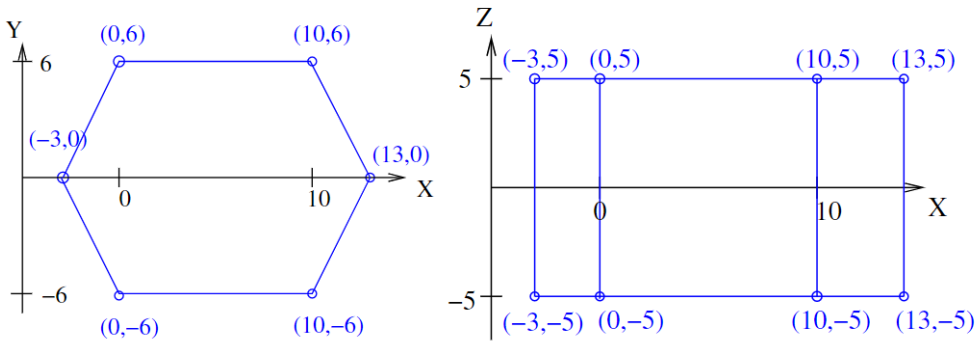


Figure 1.1: The view of the scene from the top (left) and from the side (right)

Chapter 2

Background

This report's implementation of the ray tracer is based on the Monte Carlo method for approximating the rendering equation. Importance rays are cast from the eyepoint, intersecting the camera plane, and enter the scene. The radiance returned by the ray is given to the intersection point, displayed as a pixel in the resulting image.

2.1 Ray-Object Intersections

The first step was to define a Ray. In 3D a parametric ray (line) can be defined as equation 2.1,

$$r(t) = r_0 + t(r_d) \tag{2.1}$$

where r_0 is the start point, r_d is the intersection point, and t is the parameter defining the length of the ray.

Each ray holds a radiance value that consists of a color as well as an intensity.

2.1.1 Ray-Plane Intersections

To determine if a ray has intersected a plane within the scene, the *Möller - Trumbore Algorithm* [7] was implemented. All planes within the scene were made up of triangles with three vertices v_0, v_1, v_2. With barycentric coordinates (u, v) a point on a triangle can be defined as 2.2

$$T(u, v) = (1 - u - v)v_0 + uv_1 + vv_2 \tag{2.2}$$

If the ray-plane intersection point in barycentric coordinates (u, v) meets the requirements $u \geq 0, v \geq 0, u + v \leq 1$, the ray does in fact intersect the triangle.

2.1.2 Ray-Sphere Intersections

Ray-sphere intersections are handled differently from ray-plane intersections since the spheres in the scene do not consist of triangles, but of *implicit surfaces*. Remember how our ray is defined as $r(t) = r_0 + t(r_d)$. Spheres can too be defined using an algebraic form, see Equation 2.3

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

Where x, y and z are the coordinates of a Cartesian point and R is the radius of the sphere. A point on the ray $r(t)$ is located on the surface of a sphere if Equation 2.4 is fulfilled.

$$||r(t) - c||^2 = R^2 \tag{2.4}$$

where c is the center of the sphere and r is the sphere's radius.

Developing Equation 2.4 leads to a quadric function $f(x) = ax^2 + bx + c$ where the roots can be found using Equation 2.5

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.5}$$

The sign of the discriminant $b^2 - 4ac$ indicates whether there is two, one or no root to the equation. If the discriminant value is less than zero, we have no solutions. If it is equal to zero we have one solution (tangent) and if it is greater than zero we have two solutions. In the case of two solutions we have three different cases that must be considered:

- If both solutions are positive, the ray direction is towards the sphere and the smallest value is the first intersection point of the sphere.
- If one of the solutions are positive and the other negative, the ray is shooting from inside the sphere.
- If both solutions are negative, the ray direction is away from the sphere, which means that there are no intersections.

2.2 Bidirectional Reflectance Distribution Functions

The Bidirectional Reflectance Distribution Function (BRDF) is used to decide how much light is reflected in the view direction for a given incident light direction. Different types of materials have different BRDF's, for the BRDF of Lambertian materials see equation 2.6,

$$f_r = \frac{\rho}{\pi} \tag{2.6}$$

where $\rho \in [0, 1]$ is the constant reflection coefficient.

For Diffuse Oren-Nayar [8] materials, which is a more realistic looking diffuse material, a different, more complex BRDF was implemented. Oren-Nayar surfaces are made up of V-shaped microfacets that are Lambertian reflectors. Each normal of the microfacets forms an angle with the average normal. The angles have a Gaussian distribution. The BRDF for Oren-Nayar surfaces can be seen in equation 2.7

$$f_r = \frac{\rho}{\pi} cos\theta_i (A + (B \cdot max[0, cos\phi_i - \phi_r] sin\alpha \tan\beta)) E_0 \tag{2.7}$$

where

$$A = 1 - 0.5 \cdot \frac{\sigma^2}{\sigma^2 + 0.33} \tag{2.8a}$$

$$B = 0.45 \cdot \frac{\sigma^2}{\sigma^2 + 0.09} \tag{2.8b}$$

and

$$\alpha = max(\theta_i, \theta_r) \tag{2.9a}$$

$$\beta = min(\theta_i, \theta_r) \tag{2.9b}$$

The angles θ_i and ϕ_i represent the azimuth and zenith angles of the incoming ray respectively, while θ_r and ϕ_r correspond to the respective angles of the outgoing ray.

2.3 Reflection and Refraction

The scene has transparent objects as well as reflecting objects. A perfectly reflecting object will look like a mirror while a perfectly refracting object will look like a perfectly polished glass.

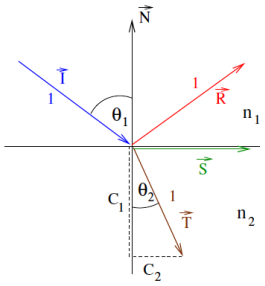


Figure 2.1: Demonstration of how reflection and refraction works, assuming that $n_2 > n_1$

For perfect reflectors, the the ray needs to reflect perfectly from the surface of intersection. The direction of the perfect reflection is computed with equation 2.10

$$\hat{R} = \hat{I} - 2(\hat{I} \cdot \hat{N})\hat{N} \tag{2.10}$$

where I is the direction of the incoming light and N is the surface normal.

For refracted rays, Snell's law can be applied. The angle θ_2 between the surface normal and the refracted ray T's direction is calculated with equation 2.11

$$\frac{sin\theta_1}{sin\theta_2} = \frac{n_2}{n_1} \tag{2.11}$$

where n_1 n_2 are the refractive indices of the different materials, see Figure 2.3

For transparent objects, Schlicks equation for radiance distribution was implemented of the refracted and reflected rays. Schlicks equation is an approximation of the Fresnel formula, and it was used since the Fresnel formula is computationally expensive. With Schlicks equation a reflection coefficient is decided with equations 2.12a and 2.12b

$$R(\theta) = R_0 + (1 - R_0)(1 - cos\theta)^5 \tag{2.12a}$$

$$T(\theta) = 1 - R(\theta) \tag{2.12b}$$

where R_0 is defined in equation 2.13.

$$R_0 = ((n_1 - n_2)/(n_1 + n_2))^2 \tag{2.13}$$

2.4 Monte Carlo Ray Tracing

To get a realistic looking scene, the indirect light, which is the integral part of the rendering equation, needs to be approximated. In this project, this approximation was done with

a Monte Carlo estimator. In equation 2.14a the analytical solution for the integral is shown, where the integral of ω is the area of the hemisphere of incoming radiance of each intersection in the scene. Since implementing the analytical solution is impossible, estimations needs to be done. This is why the Monte Carlo estimator is used, seen in equation 2.14b

$$L_P = \int_\omega L_\omega \tag{2.14a}$$

$$L_P \approx \frac{1}{N} \sum_{n=1}^{N} L_n \tag{2.14b}$$

The larger number of samples N, the smaller the error between the analytical and the approximation becomes. At each intersection, N samples would be used to compute the indirect illumination, however for this implementation only one sample will be used at each intersection, and a larger amount of samples per pixels will be used instead to give a similar effect while reducing the computational power.

2.5 Ray Termination

To avoid having the program run forever, it is important to set a ray termination strategy. To terminate rays with an unbiased scheme, the *Russian Roulette* [1] termination strategy was implemented, updating our estimator to equation 2.15

$$L_P \approx \frac{1}{N(1-\alpha)} \sum_{n=1}^{N} L_n \tag{2.15}$$

where $\alpha \in [0, 1]$ is the probability of a ray being terminated.

2.6 Rendering

To render the scene, direct and indirect illumination were computed.

2.6.1 Direct Illumination and Shadows

The direct illumination was computed with equation 2.16

$$L_D = \frac{AL_0}{M} \sum_{k=1}^{M} f_r \frac{cos\alpha_k cos\beta_k}{d_k^2} V_k \tag{2.16}$$

where A is the area of the light source, L_0 is the light intensity, and M is the number of shadow rays. Looking inside the sum, f_r is the BRDF which was mentioned earlier, and d_k is the distance from the point x and the light source.

The amount of light reflected towards the eye depends on the normal of the intersected surface as well as the normal of the light source. The equations for computing $cos\alpha$ and $cos\beta$ are shown in equations 2.17 and 2.18

$$cos\alpha = -S \cdot N_A \tag{2.17}$$

$$cos\beta = S \cdot N_x \tag{2.18}$$

where S is the shadow ray, N_A is the light source normal, and N_x is the normal at point x.

V_k in the direct illumination equation is the visibility variable. If the shadow ray is is intersected it is set to 0.0 and if it reaches the light source it is set to 1.0. Since multiple samples are used to shoot shadow rays to a random part on the light source to get an average approximation, shadows will become smooth.

2.6.2 Indirect Illumination

As mentioned earlier, for this implementation, one sample ray per intersection was used, which reduces the overall computational time. Computing the indirect illumination can be done with equation 2.19.

$$L_i = \frac{\pi}{N} \sum_{i=1}^{N} f_r L_0 \tag{2.19}$$

Where N is the number of sample rays per intersection, f_r is the BRDF of the diffuse surface of the intersection, and L_0 is the radiance of the sample ray. So for this project implementation, where $N = 1$, this equation can be simplified, see equation 2.20

$$L_i = \rho L_0 \tag{2.20}$$

2.7 Noise and aliasing

Aliasing and Moiré patterns may occur if we undersample, and increasing the image-plane resolution could solve this. However, two other methods can be used to give better results with similar increase in computational time.

The first method is supersampling. This method consists of dividing each pixel into multiple subpixels and shooting a ray into the center of each subpixel, and then taking an average of the irradiance of each ray. The irradiance calculation is exact if we use an infinite number of rays. To reduce the noise of the Monte Carlo method, larger amounts of samples per pixel will be used.

The second method is ray randomization. This method consists of shooting a ray through a random position of each pixel instead of shooting it through the center. As a result, the aliasing in the image is replaced by noise which is less disturbing to viewers.

In this project we have combined both of these techniques to achieve better results.

Chapter 3

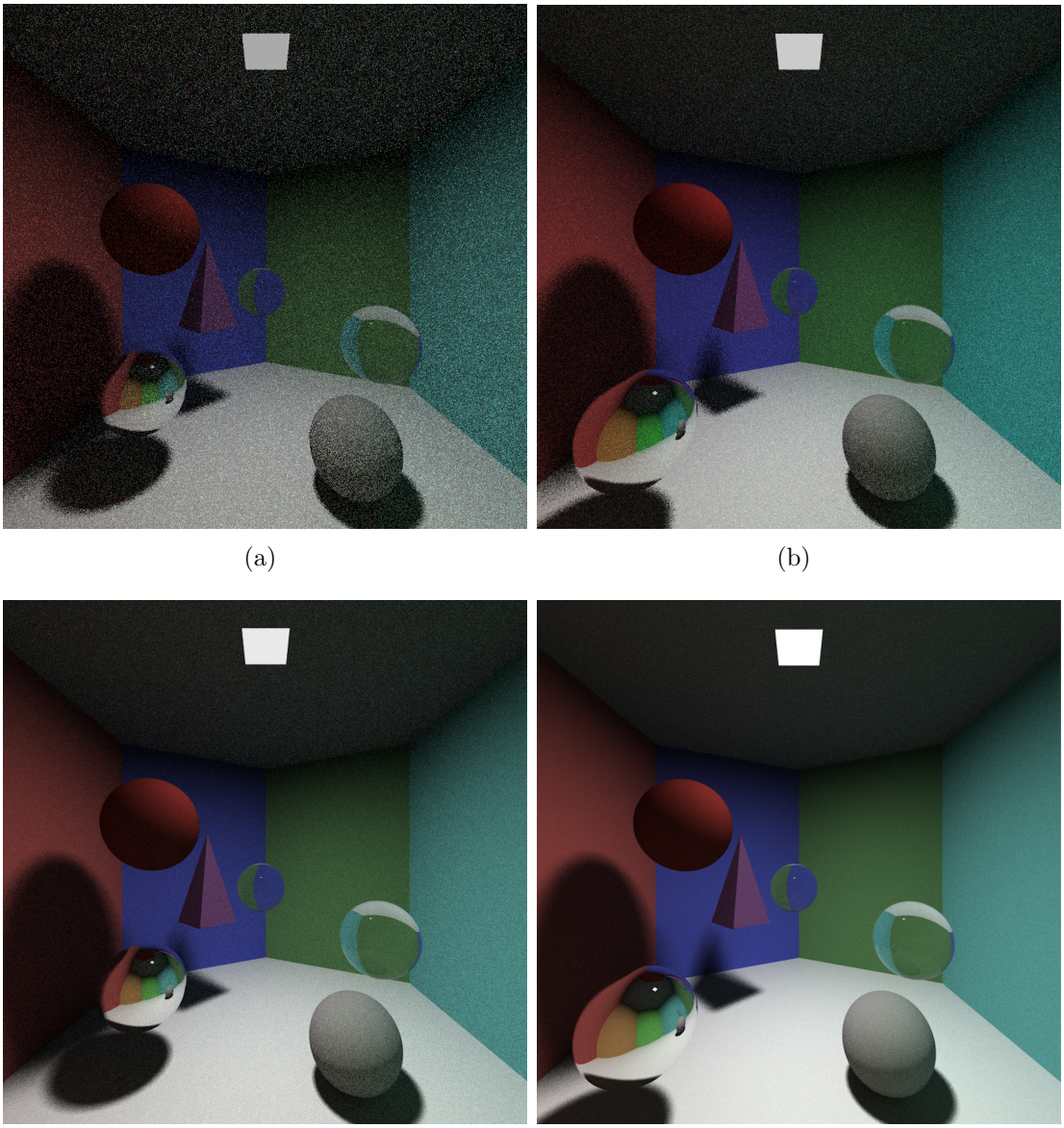
Result

The Monte Carlo ray tracer was implemented in C++ and compiled with CMake. The OpenGL Mathematics library (GLM) was used to implement vectors, matrices, and mathematical operations. This section covers the ray tracer's different results when varying the number of samples per pixels (SPP), the number of shadow rays, the surface materials, and the BRDFs.

In Figure 3.1, the scene was rendered with 1, 4, 16, and 64 samples per pixel. Comparing (a), (b), (c), and (d), it can be seen that the increased amount of SPP leads to less noise in the final image.

Figure 3 shows a 800x800-pixel image of the scene with 64 SPP. The scene contains a perfect reflector, two transparent spheres, two diffuse spheres (one Lambertian and one Oren-Nayar), a tetrahedron, and a light source. The image is rendered using three shadow rays per intersection. For diffuse objects, ρ was set to 0.8, and σ was set to 0.9 for the Oren-Nayar material. In Figure 3, all diffuse surfaces are Lambertian except the gray sphere on the ground, which has the Oren-Nayar material type.

Figures 3.3 and 3.4 are zoomed in versions of Figure 3, displaying soft shadows and color bleeding, respectively.



(a) (b)

(c) (d)

Figure 3.1: The scene rendered with a variation of SPP. (a) is rendered with 1 SPP, (b) with 4 SPP, (c) with 16 SPP, and (d) with 64 SPP.

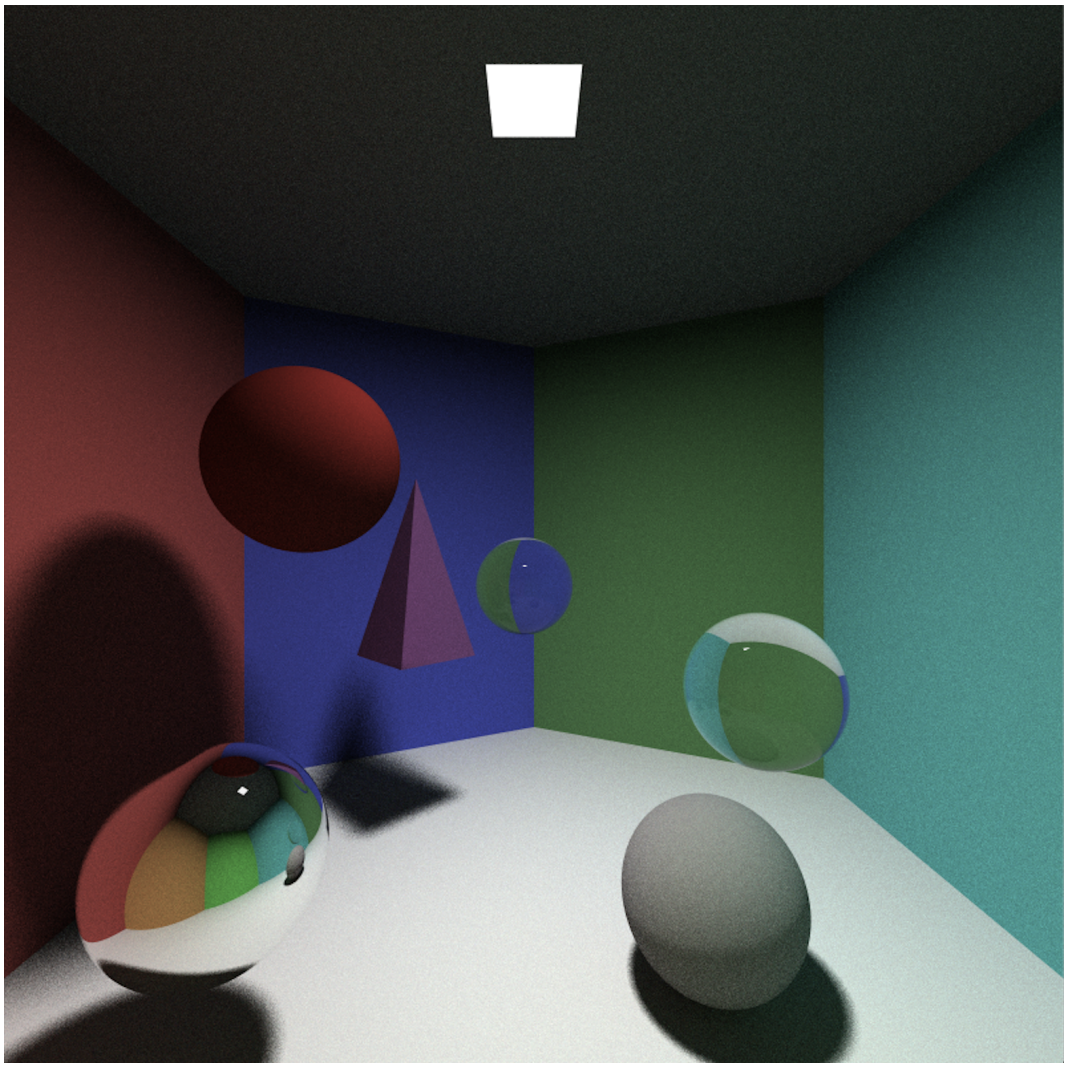


Figure 3.2: The scene rendered with 64 SPP. The scene contains a perfect reflector, two transparent spheres, two diffuse spheres (one Lambertian and one Oren Nayar), a tetrahedron, and a light source.

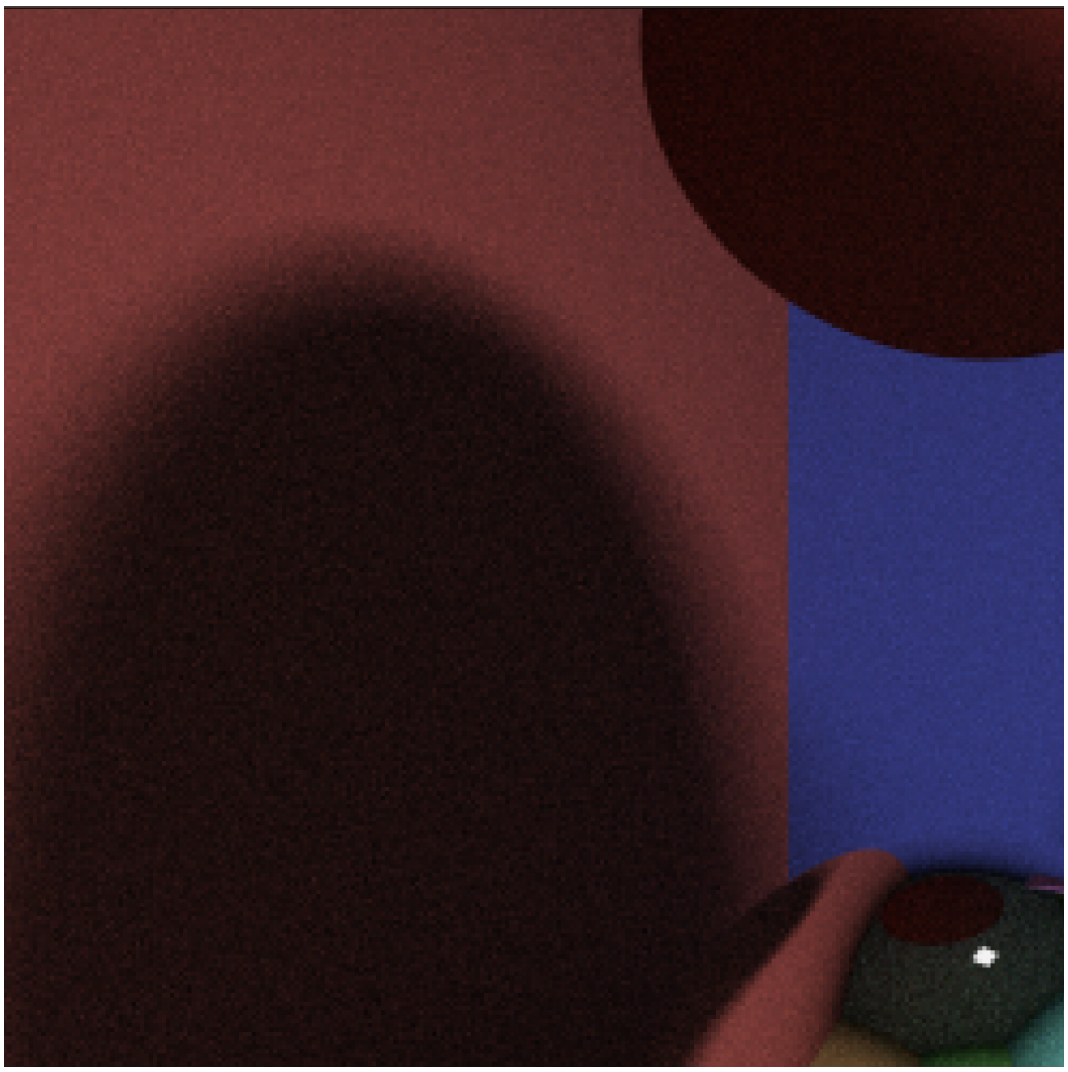


Figure 3.3: Zoomed in image displaying soft shadows in the rendered result.

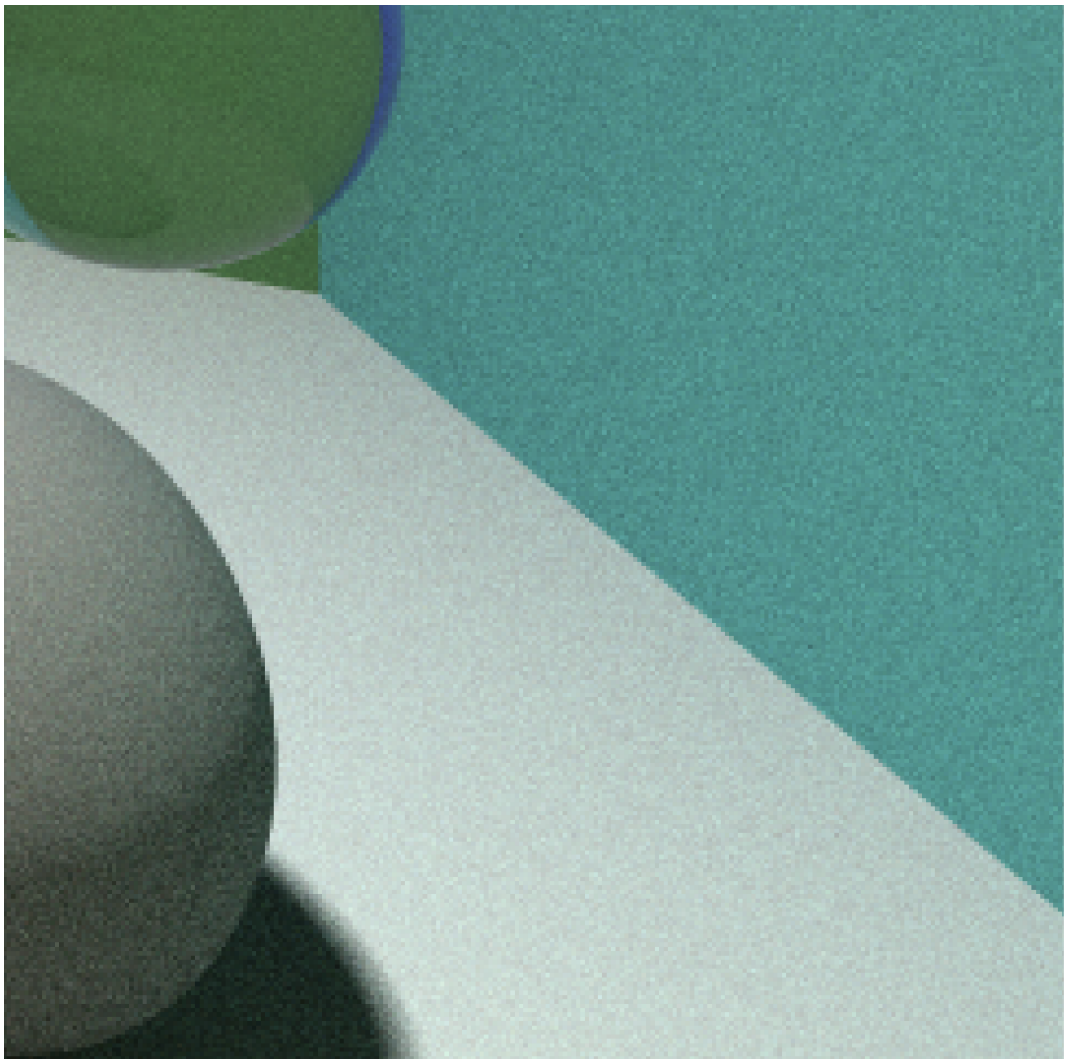


Figure 3.4: Zoomed in image displaying the color bleeding from the wall onto the floor and the diffuse sphere.

Chapter 4

Discussion

Eight years after Whitted ray tracing was introduced, Kajiya released the rendering equation and an extension of Whitted ray tracing known as Monte Carlo ray tracing. Monte Carlo ray tracing solved issues present in Whitted ray tracing, such as hard shadows and a lack of color bleeding, which the paper displays.

To give the scene soft and realistic shadows, shadow rays were sent to a random part of the scene's area light. With the implementation of supersampling, an average of all the shadow rays gives shadows a smooth interpolation between fully lit and wholly covered from direct light. To give shadows an even smoother result, more shadow rays are cast for every sample ray. However, using too many shadow rays will significantly increase the computational time, and we found that using just three shadow rays per intersection was sufficient.

Color bleeding was achieved with Monte Carlo integration as well. A photo-realistic image has color bleeding due to the indirect illumination factor. In our results, we highlight that we have, in fact, achieved color bleeding. The indirect illumination also corrected shadows from being unrealistically dark.

The technique behind simulating a scene with Monte Carlo ray tracing is pretty simple, although a significant drawback is the computational power that it requires. Monte Carlo is known for being slow and noisy, which was present in this implementation. More samples were used to reduce the noise, but this also significantly increased the time to render the scene. Anti-aliasing was achieved by using supersampling together with ray randomization. This technique gave all the edges in the scene softer transitions in high contrast areas, which is less disturbing for a viewer than seeing jagged edges, which added to the photo-realism.

The results of the Monte Carlo ray tracer look pleasing. However, they are computationally heavy. Photon mapping could have been implemented to decrease computational time. This implementation was done from scratch in C++ and computed with the CPU. It could have been implemented with an API that interacts with the GPU for hardware-accelerated rendering. Moreover, multithreading could also have been implemented to speed up the rendering process.

Bibliography

[1] James Arvo and David Kirk. "Particle transport and image synthesis". In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990, pp. 63–66.

[2] Mark E Dieckmann. "Global illumination using photon maps". In: *TNCG15: Advanced Global Illumination and Rendering: Lecture 6 ("The scene")*. MIT group, ITN. 2020.

[3] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[4] Cindy M Goral et al. "Modeling the interaction of light between diffuse surfaces". In: *ACM SIGGRAPH computer graphics* 18.3 (1984), pp. 213–222.

[5] Henrik Wann Jensen. "Global illumination using photon maps". In: *Eurographics workshop on Rendering techniques*. Springer. 1996, pp. 21–30.

[6] James T Kajiya. "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150.

[7] Tomas Möller and Ben Trumbore. "Fast, minimum storage ray-triangle intersection". In: *Journal of graphics tools* 2.1 (1997), pp. 21–28.

[8] Michael Oren and Shree K Nayar. "Generalization of Lambert's reflectance model". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, pp. 239–246.

[9] Turner Whitted. "An improved illumination model for shaded display". In: *ACM Siggraph 2005 Courses*. 2005, 4–es.