Monte Carlo Ray Tracer

TNCG15 Advanced Global Illumination and Rendering

Anna Wästling (annwa917)

Jessie Chow (jesch310)

January 4, 2022

Abstract

This report presents the implementation of a Monte Carlo Ray Tracer in C++. A scene with six walls, a ceiling, floor, a tetrahedron and a sphere have been rendered as a result. The scene contains different materials which results in phenomenons like color bleeding and soft shadows and different reflections. A discussion about the result is presented at the end of the report.

Table of Contents

List of Figures

1. Introduction

The aim of 3D computer rendering is to create a photorealistic image using data sets, pixels. The process of producing a photorealistic rendering is difficult because there are many elements to take into consideration, where the most central one is the light. Light has different properties depending on if it is direct light i.e. light that comes directly from the light source or indirect light i.e. if it is reflected off objects in the scene. A method to model the light in a scene is global illumination.

1.1 Global Illumination

Global illumination models both the direct light from the lightsource and the indirect light that is reflected off objects. As the light is reflected off an object, the colour of the object will be reflected onto the next surface that is hit by the light in an effect called colour bleed. In addition, light that is reflected off an object is diffused and will be softer when it hits the next object, casting soft shadows. Achieving global illumination in 3D rendering is challenging because light interacts differently with different reflection models. To describe how the light, or radiance, is transported in the scene, the *rendering equation* (1.1) [1] is used.

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

where L_s is the surface radiance in the point x, L_e is the emitted radiance by the surface, which does not need to be calculated as it can be found through the definition of the surface. The integral describes the radiance values in the scene and can be computed using ray tracing.

1.2 Ray Tracing

Ray tracing is a method to simulate the light in a scene and is done by tracing a ray that originates from the light source into the scene. The ray is traced as it bounces off different objects in the scene until it hits the point it originated from. As the ray bounces around in the scene and interacts with the objects, pixel values will be generated and results in a rendered image. This method is called *forward ray tracing*. However, many of the rays will not have any significant impact on the final render as they will not reach the viewpoint, usually the camera. In order to guarantee that the rays is evenly distributed throughout the scene, a large number of rays have to be used and will result in a computationally expensive rendering process.

1.2.1 Whitted Ray Tracer

One of the first techniques to simulate light was described by Whitted in 1980 and is called the *Whitted Ray Tracing* [2], which uses backward ray tracing. Instead of tracing rays that originate from the light source, backward ray tracing traces rays that are shot from the viewpoint, usually the camera. By tracing rays that originate from the viewpoint, it can be assured that every ray will be seen. As the ray bounces in the scene, the pixel value will be calculated based on the surfaces of the objects that the ray intersects. If the ray intersects an object with a mirror surface, the ray will be reflected in a random direction since that is the property of a mirror. To prevent a reflected ray from bouncing for an infinite number of times, a limit on how many times a ray can bounce is introduced in the algorithm. On the other hand, if the ray intersects an object with a diffuse surface, shadow rays will be sent out from the intersection point towards the light source to check if the intersected point is directly illuminated or if it is in shadow. Depending on how many shadow rays that hits an object as they are shot towards the light source, the amount of illumination on that point can be determined.

1.2.2 Monte Carlo Ray Tracer

While Whitted Ray Tracing can simulate transparent and specular (mirror) surfaces, it has difficulties with simulating reflection between diffuse surfaces e.g. colour bleed and soft shadows. A technique that covers the advantages of Whitted Ray Tracing and handles reflection between diffuse surfaces is *Monte Carlo Ray Tracing*, described by Kajiya [1], and is a way to solve the aforementioned rendering equation (1.1). The idea is to approximate the integral part in the rendering equation by tracing a ray throughout the scene. Each ray is set to recursively bounce a predetermined number of times to prevent a situation of the ray bouncing infinitely between two mirror surfaces and also to make the rendering process more effective. Another method that is used to terminate the rays is Russian Roulette. When the ray intersects a surface, it will be determined of how the ray will be reflected depending on the surface type, which is the same idea as in the Whitted Ray Tracing. The difference is that a diffuse surface will reflect the ray into a random direction, which is the indirect illumination part in global illumination. As the ray is reflected from a diffuse surface, its intensity will be reduced which results in softer shadows. The reflected ray also carries over the colour of the object in the intersection point and results in the effect colour bleed.

2. Background

The ray tracer implemented in this project is a Monte Carlo style ray tracer where rays are cast into the scene from the viewpoint and traced as it recursively bounces in the scene. The ray returns a radiance value in the form of a color that will be used to render the resulting image.

2.1 Scene

The most commonly used test scene in rendering is called the *Cornell box* [3] which is in the shape of a cube, where the light source is positioned in the middle of the ceiling which is white which is also the colour of the back wall and the floor. The basic Cornell box has a red wall to the left and a green wall to the right. The purpose of the red and green walls is to be able to see colour bleeding as the white coloured parts of the box will get a subtle shade of green and red. With the differently coloured walls, the objects in the scene with different reflection models can be tested to see if they are correctly implemented. In this project however, the Cornell box will not be used. Instead, a scene with a hexagonal room is used [4], see Figure 2.1.

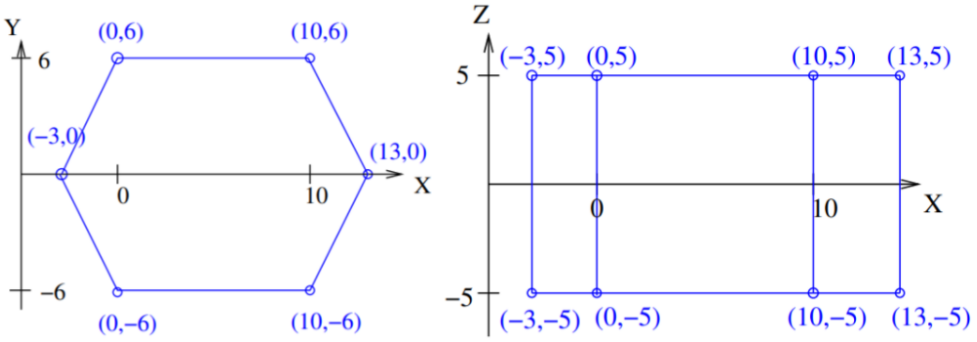


Figure 2.1: The scene with a hexagonal shape. The scene from the top is shown to the left and the scene from the side is shown to the right.

The scene has walls in different colours while the ceiling and floor are both white. The walls, ceiling and floors are made of diffuse materials. The lightsource in the room is centered in the ceiling. In this implementation, two objects of different materials are present: a tetrahedron, which is a Lambertian (diffuse) reflector, and a sphere, which is a perfect (mirror) reflector. The camera is placed in the middle of the room with two different positions on the x-axis so that four of six walls can be seen directly while the remaining two walls behind the camera can be seen through the reflection of the sphere.

2.2 Ray

The ray was constructed with a starting point, an endpoint, a direction, which was normalized, and a color.

Two positions were defined for two different ”eyes” and is the starting position of the rays that are cast into the scene through a pixel in the camera plane. As the ray hit a diffuse surface, the endpoint was set to the intersection point between the ray and the surface and the colour of the ray was set to the same color as the surface. By setting the a colour to the ray, it will give an effect called colour bleed since the next point that is hit by the ray will get some of the color from the previous surface. The ray bounces around in the scene until it is terminated, which is determined by using a counter that increments every time a ray is reflected. Once the counter hits a predetermined number, in this case the number was ten, the ray is terminated.

Aliasing is unwanted patterns that can appear when undersampling. By sending more rays into one pixel, see Figure 2.2, the average color of the pixel can be calculated and aliasing is reduced. More rays gives more intersectionpoints in the scene which gives a better approximation for the pixel. This was calculated in combination with the method supersamling. Instead of intersecting a pixel at the same point the rays gets their intersection point by using random displacement in the pixel. This contributes to a better and more random distribution of the rays and therefore less aliasing. However, by using random

displacement samples can end up being too detailed in some areas whilst not detailed enough in others.

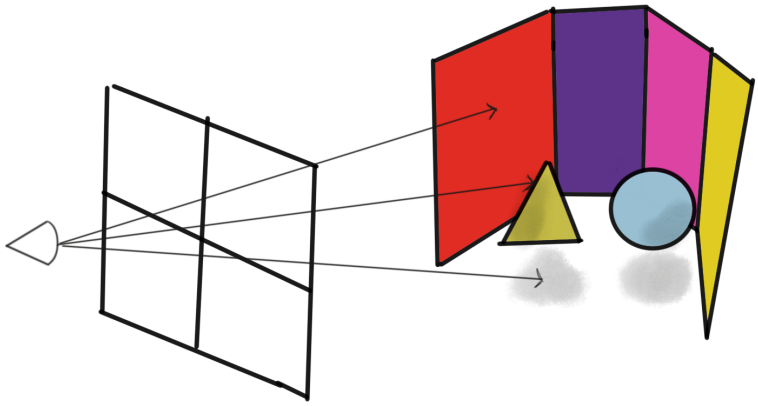


Figure 2.2: Three rays are sent through a pixel at different points towards the scene

2.2.1 Ray Intersection

The ray could intersect with two kind of objects in the scene, a sphere or a triangle. Triangles were not mentioned in section 2.1 but triangles were used to define the walls, ceiling and floor of the scene as well as the tetrahedron which is made up of four triangles. Therefore, there will only be two different intersection algorithms where one is for the triangles and one for the sphere.

The Möller-Trumbone intersection algorithm (2.1) [5] was used to determine the intersection point between a triangle in the scene and a ray. A ray intersects a triangle if the barycentric coordinates (u,v) meets the conditions where $u \geq 0$, $v \geq 0$ and $u + v \geq 1$ [4].

$$T(u, v) = (1 - u - v)v_0 + u \cdot v_1 + v \cdot v_2 \tag{2.1}$$

To calculate the intersection point between a sphere and a ray, an analytic solution [6] was implemented. The ray is defined as (2.2) where O is the point of origin of the ray, D is the direction of the ray and t is a point on the ray.

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

The sphere is defined as (2.3) in standard coordinates.

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

where (x, y, z) are Carthesian coordinates of the center of a sphere in the origin that has the radius R.

(x, y, z) is then substituted with (2.2) where the ray intersects the ray if (2.4) is true.

$$|O + tD|^2 - R^2 = 0 \tag{2.4}$$

In order to determine the number of intersection points, the quadratic form of the equation (2.5) is solved by x (2.6).

$$f(x) = ax^2 + bx + c \tag{2.5}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.6}$$

where the sign of the discriminant $\Delta = b^2 - 4ac$ determines the number of intersection points as the following:

- $\Delta \geq 0$ gives two roots which indicates that there are two intersection points between the sphere and the ray.
- $\Delta = 0$ gives one root which indicates that the ray only has one intersection with the sphere.
- $\Delta < 0$ gives that there are no roots and therefore the ray does not intersect the sphere.

2.3 Illumination

The distribution of light, both direct and indirect illumination, was calculated to create a realistic scene.

2.3.1 Direct Illumination

A spherical object with radiance 1.0 $Wm^{-2}sr^{-1}$ was introduced in the scene to act as the light source. It was positioned in the ceiling of the room with its normal direction towards the floor.

To get the shadows of the objects in the scene, shadow rays were introduced. A shadow ray was created with the starting point at the intersection point between the ray that was cast from the eye and a surface, and was cast towards the light source. By checking if the shadow ray hits any objects on its way to the light source, it can be determined if the point is in the shadow of an object or if it is directly illuminated by the light source. An example of a shadow ray that intersects a tetrahedron as it is cast from the floor towards the light source can be seen in Figure 2.3. The more shadow rays that are computed, the softer the shadow will be since there will be samples at different points below an object.

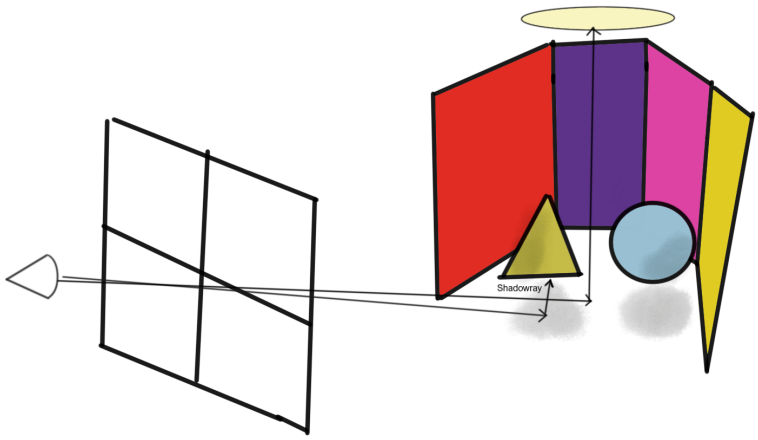


Figure 2.3: Shadow ray that hits the area under a tetrahedron

2.3.2 Indirect Illumination

To visualize the scattering of light in an realistic way, indirect illumination was computed by redirecting the ray in a random direction when intersecting a diffuse surface. The ray is

recursively reflected until it is absorbed by the surface. One of the methods to determine if the ray will be absorbed is Russian Roulette where 25% of the rays got terminated at random. This was only done when hitting a diffuse surface like the walls, ceiling, floor or the tetrahedron. To get the right amount of light in the scene, the radiance of the ray was also reduced at each redirection.

2.4 Reflection Models

The scene had two different materials, diffuse and reflective. The sphere had a highly reflective material while the tetrahedron, the walls, floor and ceiling had a diffuse material. The reflection model of the two different materials were computed differently.

2.4.1 Diffuse Reflection

When shooting a ray towards the diffuse surfaces, direct illumination will be calculated and added if certain conditions are met. The direct illumination will be computed only if the intersection point on the diffuse surface is not in shadow. Meanwhile, indirect illumination is always computed since the ray will always be reflected off a diffuse surface.

2.4.2 Mirror Reflection

A ray that intersects a reflective surface will be reflected off the surface using the normal of the surface to calculate the reflected rays direction, creating a perfect reflection. This is done by casting a new ray in the direction of the surface normal and applying the color of the cast ray on the sphere.

3. Results

The Monte-Carlo ray tracer was implemented in C++ with the use of the OpenGL Math library (GLM) to create vectors and mathematical equations. The size of the rendered image is 800x800 pixels. The scene is a hexagonal room where the walls have different colours and diffuse material. The objects in the scene is a tetrahedron with diffuse material and a sphere with mirror material. The light source is located in the center of the ceiling and has the side length 3. The rendering result of the scene using 8 rays per pixel and 4 shadow rays is shown in Figure 3.1.

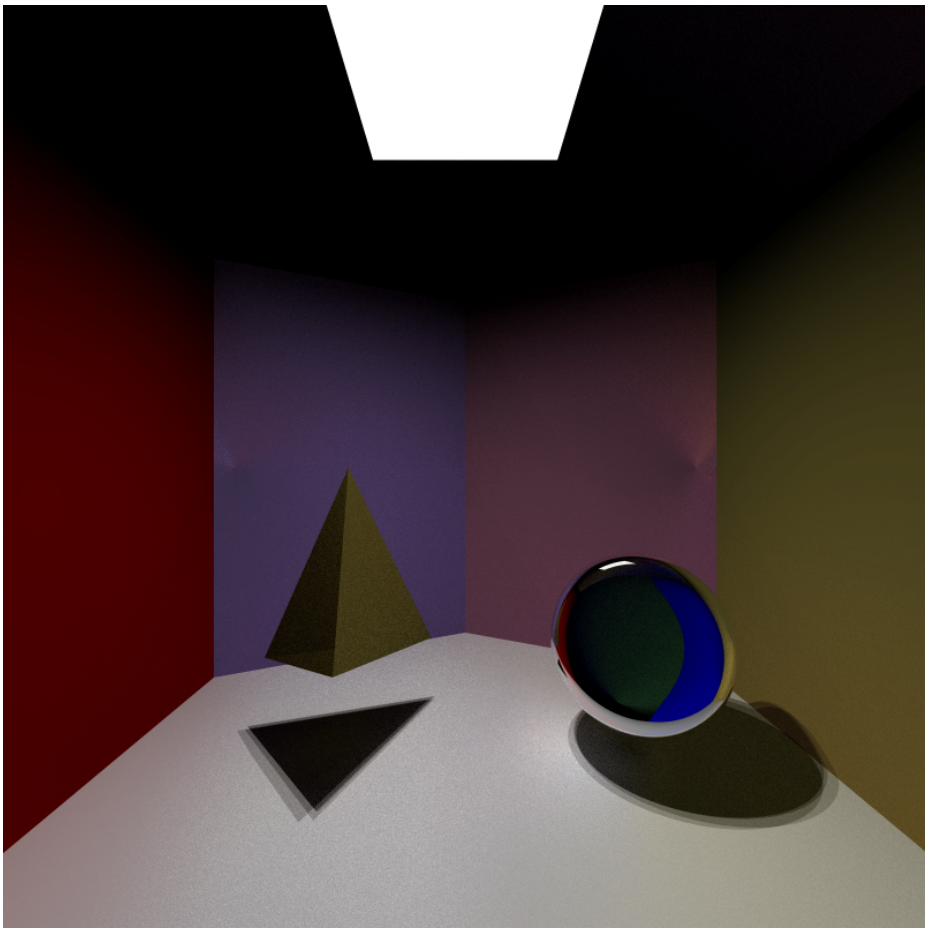


Figure 3.1: The scene rendered with 8 rays and 4 shadow rays

A zoom in to show the soft shadows and colour bleed on the rendering results of 8 rays per pixel and 4 shadow rays are presented in Figure 3.2 and Figure 3.3 respectively.

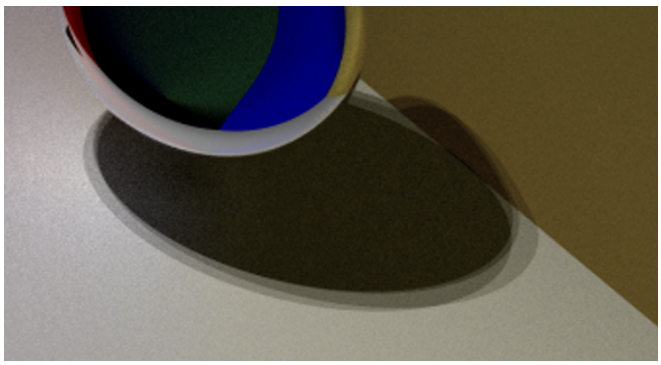


Figure 3.2: Soft shadows

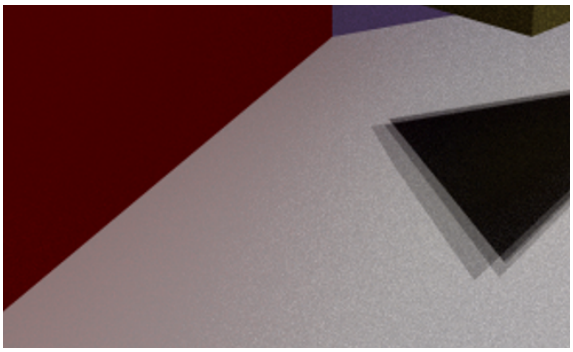
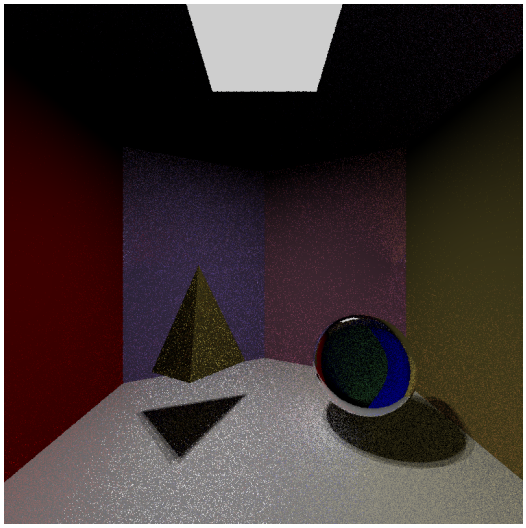


Figure 3.3: Color bleed where the red colour from the wall is reflected onto the white floor

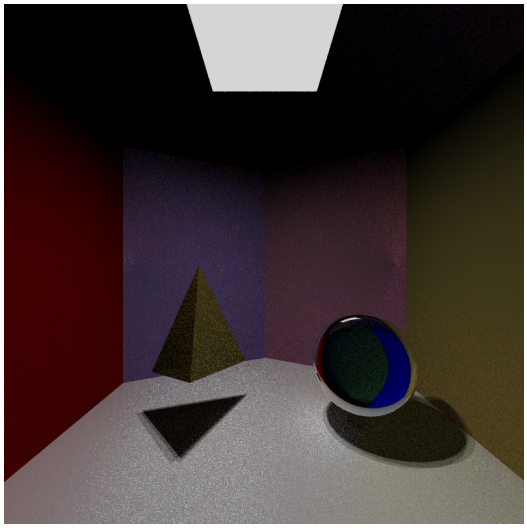
The rendering time of the scene with different number of rays per pixel (1, 2, 4 and 8 rays per pixel) are presented in Table 3.1 and the rendering results are show in Figure 3.4.

Number of rays per pixel	Render time (seconds)	Render time (minutes)
1	91.10	1.52
2	364.99	6.07
4	1481.83	24.69
8	6033.97	100.56

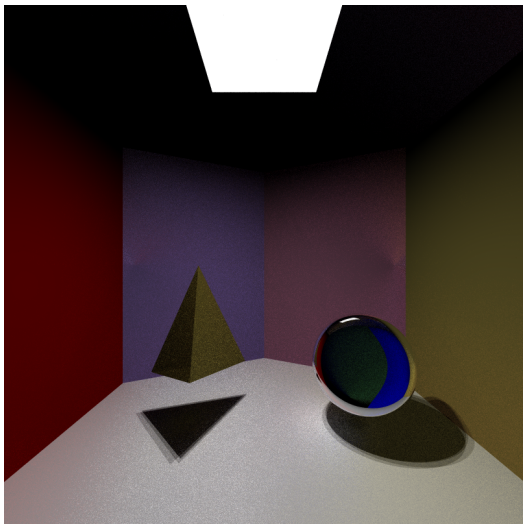
Table 3.1: Rendering time for different number of rays per pixel on a desktop



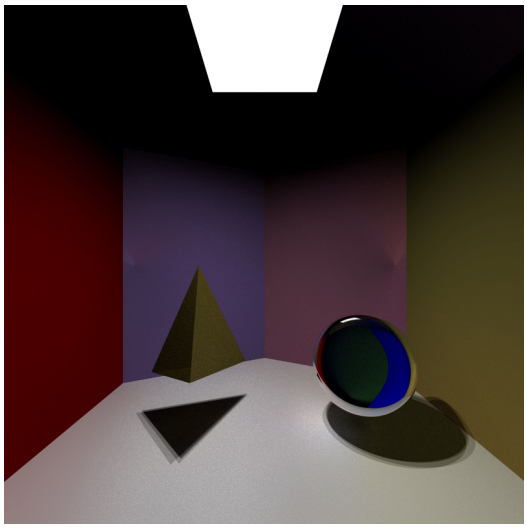
(a) 1 ray



(b) 2 rays



(c) 4 rays



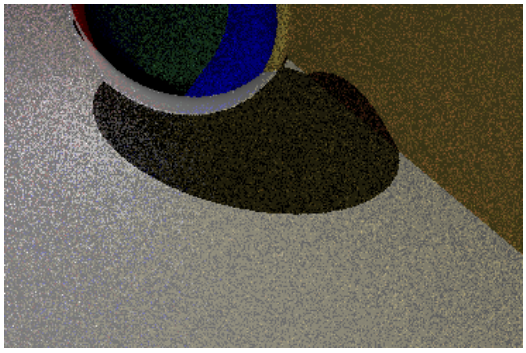
(d) 8 rays

Figure 3.4: Renders of the scene with different number of rays per pixel and four shadow rays

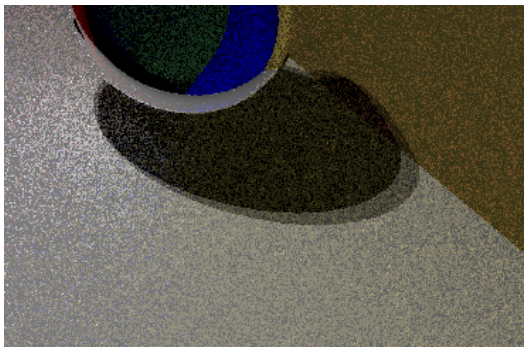
To show the effect of different number of shadow rays, the scene was rendered with 1, 2, 4 and 8 shadow rays. The rendering time for the different number of shadow rays are presented in Table 3.2. The rendering results are presented in Figure 3.5 where there is a zoom in on the renders to show what effect different number of shadow rays have on the shadow.

Number of shadow rays	Render time (seconds)	Render time (minutes)
1	149	2.48
2	158	2.63
4	164	2.73
8	219	3.65

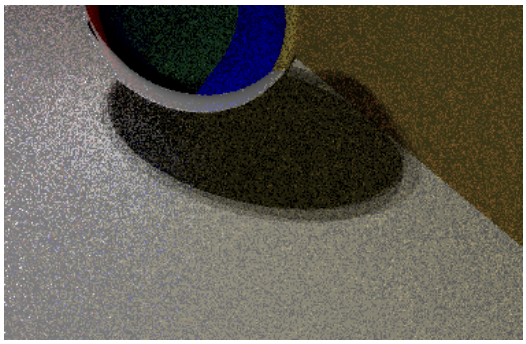
Table 3.2: Rendering time for different number of shadow rays on a laptop



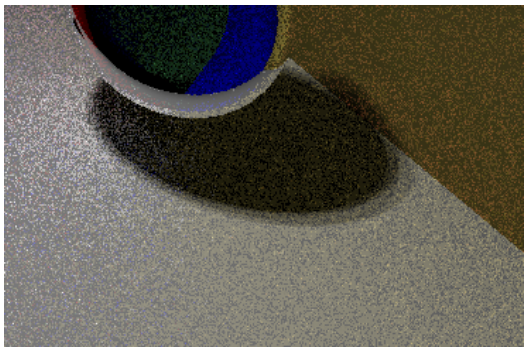
(a) 1 shadow ray



(b) 2 shadow rays



(c) 4 shadow rays



(d) 8 shadow rays

Figure 3.5: Zoom in on the rendering results of the scene with 1 ray per pixel and different number of shadow rays

Some unknown artefacts on the walls, circled in red, can be seen in Figure 3.6.

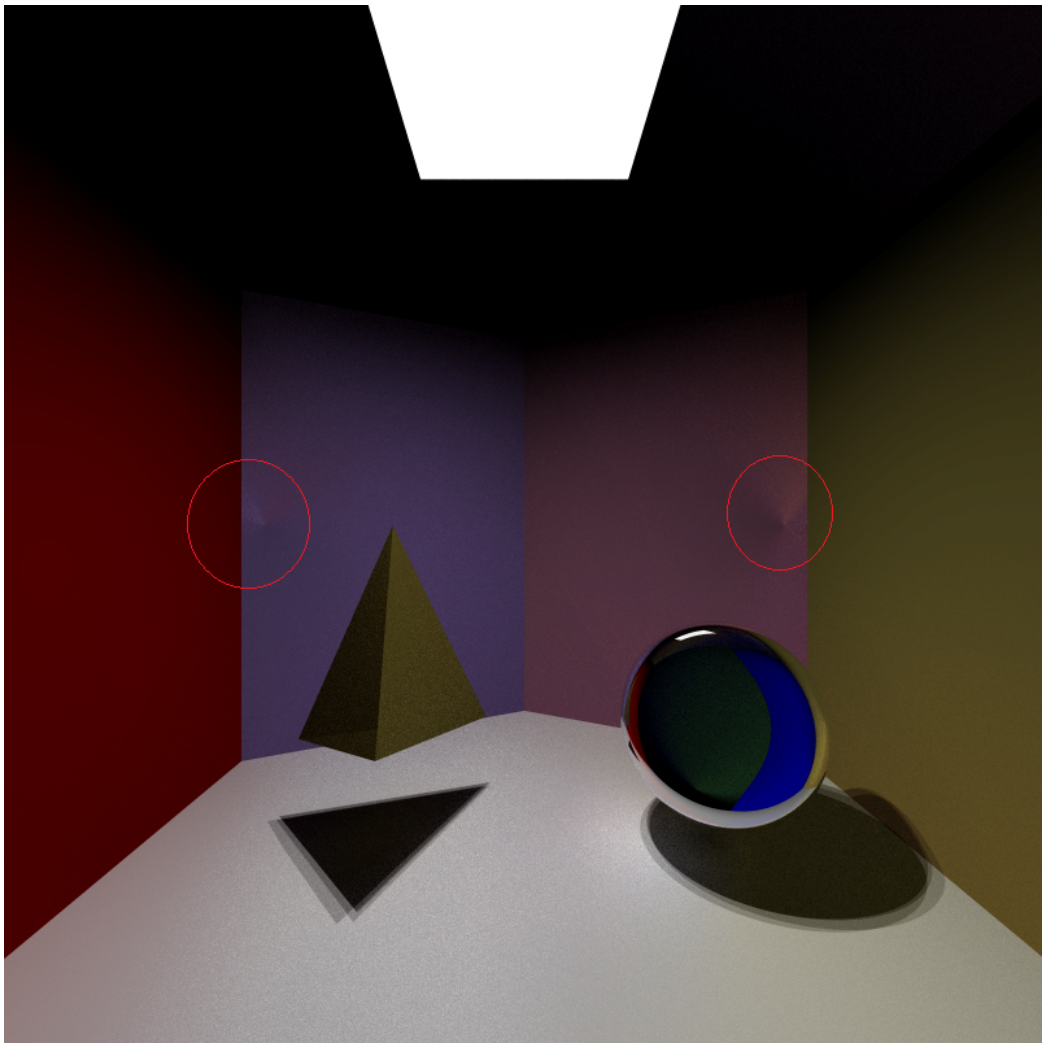


Figure 3.6: Artefacts on the purple and pink walls, circled in red

4. Discussion

The results show that global illumination was achieved using Monte Carlo Ray Tracing with soft shadow (Figure 3.2) and colour bleed (Figure 3.3) effects in the renders.

As the number of rays per pixel are increased in the renders, the quality of the render increased where the renders with a higher number of rays per pixel are less noisy, which can be best seen by comparing 1 ray per pixel (Figure 3.4a) and 8 rays per pixel (Figure 3.4d). However, the rendering time with a higher number of rays per pixel is significantly increasing(Table 3.1). During the rendering process of the project, an attempt to render with 16 rays per pixel was made but was eventually aborted since it would have taken an estimated 400 minutes which is too much time. To reduce the computation time, multi-core rendering or photon mapping can be implemented to speed up the process. Therefore, the conclusion is that 8 rays per pixels results in the best image, since it has a good computation time and has almost no noise.

Similar to how the computation time increased with a increasing number of rays per pixel, the computation time for a higher number of shadow rays also increased (Table 3.2). The difference in computation time could be due to different devices (desktop and laptop), however, the computation time is increasing for both devices. Therefore, to render the scene with different number of shadow rays, only one ray per pixel was used. The effect of one shadow ray (Figure 3.5a) is a hard edge, and with two shadow rays (Figure 3.5b) the soft shadow effect can be seen but it is not considered to be good. With four shadow rays (Figure 3.5c), the soft shadow effect is significantly better and it becomes more realistic with eight shadow rays (Figure 3.5d). However, as the computation time will increase with the number of shadow rays, four shadow rays were used when rendering the scene with different number of rays per pixel.

There are also some unknown artefacts on the walls, as seen in Figure 3.6. This can have multiple reasons but the solution is for us unknown.

The result of using Monte Carlo ray tracing gives pretty good visualisation, however it requires a lot of computational power and takes a long time to render. In this project the render time could potentially become better by improving the code and make it more efficient. Moreover, the computation time will be improved if the computer has better specifications which can be seen when comparing the computation time of 1 ray per pixel and 4 shadow rays rendered with a desktop (Table 3.1) and a laptop (Table 3.2).

References

[1] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[2] Turner Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 4–es, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[4] Mark E Dieckmann. Lecture 6 ("the scene"), 2021. Accessed: 2021-10-26.

[5] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, oct 1997.

[6] A minimal ray-tracer: Rendering simple shapes (sphere, cube, disk, plane, etc.). `https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection`. Accessed: 2021-12-23.