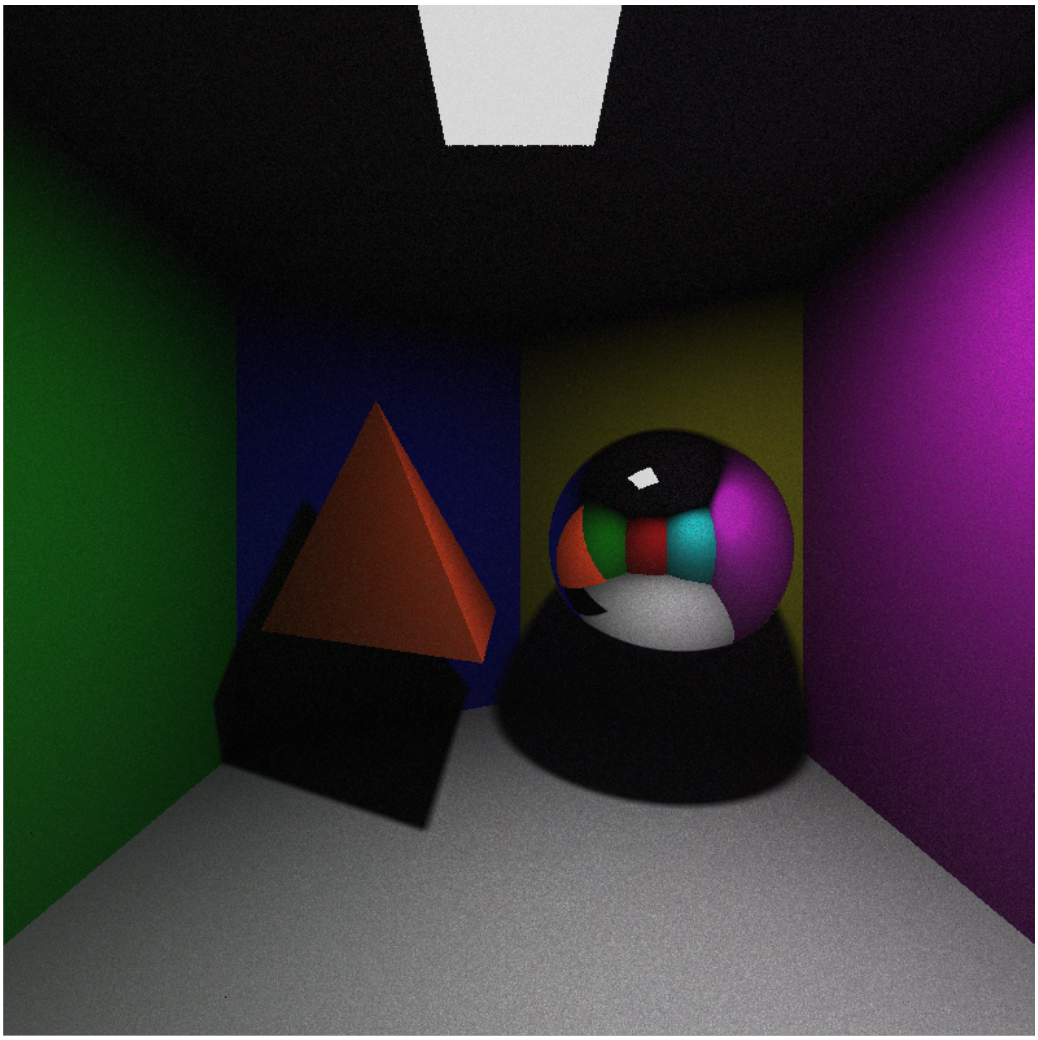
Monte-Carlo Ray Tracer

Edvin Nordin and Jimmy Cedervall Lamin

Febuary 20, 2023



Contents

1 Introduction

Global illumination is a common term used to describe a set of techniques and algorithms that tries to simulate the behavior of light in a virtual environment such as computer graphics. Compared to other techniques used in computer graphics that tries to simulate light, that for example assumes all light comes from one single source, global illumination captures a much more complex behavior where the light rays is affected by all surfaces in the scene. Accounting for reflection, refraction and certain indirect lightning effects.

Global illumination is a very powerful technique that is used in multiple applications ranging from video games to movie production and renders. By using global illumination and creating a very highly realistic model of how the light behaves, the scene becomes much more immersive and captures the nuances of real-world lightning conditions. An example of how important global illumination is for a realistic scene is shown in figure 1.

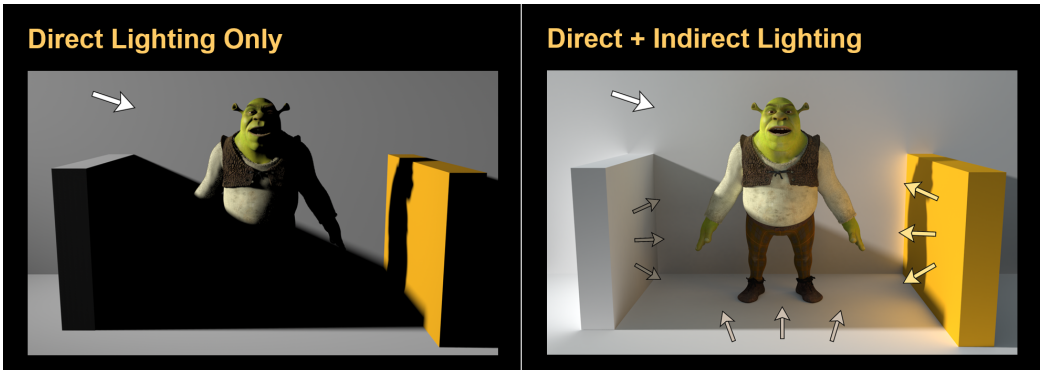


Figure 1: An example of how global illumination affects a scene.

There are more than one way to create a global illumination algorithm, including ray-tracing, radiosity, path tracing, photon mapping and many more. All with their different strengths and weaknesses. In this project we will be using a Monte-Carlo ray tracing method to simulate a scene.

2 Background

In order to get a better understanding of our Monte-Carlo ray tracer there are some methods and techniques that we will talk about in this chapter.

2.1 The Rendering Equation

The rendering equation is used to calculate the outgoing light from a point and is used all the time. The equation looks as follows:

$$L(x \rightarrow \omega_{out}) = L_e(x \rightarrow \omega_{out}) + \int f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in}) cos\theta_{in} d\omega_{in} \quad (1)$$

- $L(x \rightarrow \omega_{out})$ describes the outgoing light from the point x with the direction of ω_{out}.
- $L_e(x \rightarrow \omega_{out})$ describes the emitted light.
- $f_r(x, \omega_{in}, \omega_{out})$ describes the BRDF (bidirectional reflectance distribution function) of the objects material.
- $L(x \leftarrow \omega_{in})$ describes the radiance, the incoming light.
- $cos\theta_{in}$ describes the amount of falloff depending on the difference of the angles between the incoming and outgoing ray.

2.2 Light Bouncing

When a light ray hits a surface it will react in different ways depending on what material the objects that is hit has. In this project two different surfaces was used: a Lambertian surface and a perfect reflector but multiple other material can be used and they all affect light in different ways. The Lambertian material is an ideal matte surface that either absorbs or bounces the light in a random direction. This is the complete opposite to the perfect reflector which always reflects the light ray perfectly like a mirror. These two materials requires two different reflection models. A calculation of where the ray hits an object must be made to then determine what reflection model to use. These models also differ depending on what primitive the object hit is. To get the correct lighting of a point, shadows must also be taken into consideration. To determine if a point is in shadow, a ray is cast from the

point to the light source. This is called a shadow ray. If the shadow ray hits something other than the light source the point is in the shadow of the hit object. The shadows also depends on the distance and angle to the light. This method produces a phenomenon called color bleed which is when some of the color of nearby objects "bleed" into another as the light has a bigger probability to bounce onto it.

3 Implementation

The implementation of the Monte-Carlo ray tracer is described in this chapter.

3.1 The Environment

The scene is comprised of six walls, a roof and floor, a tetrahedron, a sphere, and the square light placed in the roof. To make it as simple as possible all objects in the scene, except for the sphere, is made out of triangles. This makes the lights interaction with each objects, not including the sphere, the same and no new method has to be implemented. The room is a hexagon shape with each wall composing of two triangles each, this makes the floor and roof also hexagonal. To differentiate the different the objects each wall, floor, roof, and the tetrahedron has different colors and the sphere is fully made out of totally reflective material. A rough sketch of how the room is built without the black roof can be seen in figure 2. The camera can be in two positions, either (-1, 0, 0) or (-2, 0, 0). From this its position it send ray into a square grid, called the camera plane, which then creates the pixel in the end image. This plane is between the points (0, 1, 1) to (0, -1, -1).

3.2 Calculating the Light

The bouncing light is calculated using a recursive function. Using a method called "Russian Roulette" some light bounces will be "absorbed" when they hit a Lambertian material.[1] For each bounce a random number between 0 and 1 is generated and if it's below 0.75 the ray will be "absorbed" by the material and return the color of that point multiplied by a bounce coefficient, in this project set to 0.8. A limit to the amount of bounces an ray can have is also set to quicken the computing time. This method creates a lot of noise in the rendered image so super sampling of each pixel is used. Each pass in the super sampling has a small random offset from the pixel position to further decrease noise and prevent artifacts. For each bounce the rays intersection with the object has to be calculated.

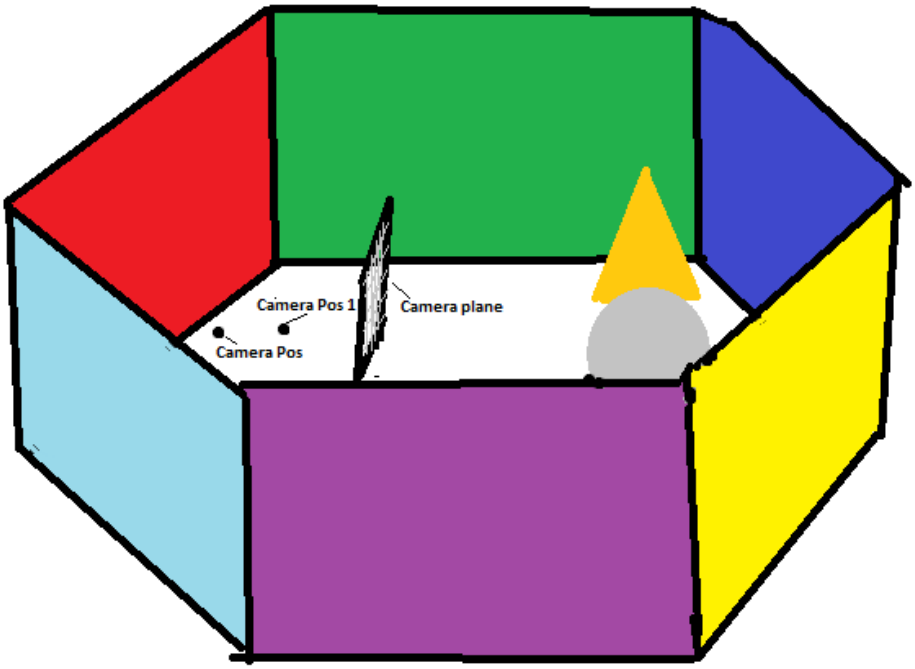


Figure 2: Rough image of the scene.

3.3 Triangle Intersection

The first step to find the intersection between a triangle and a ray is to find the length of the ray and the intersection point on the plane of the triangle.[3] These are done by using equations 2 and 3.

$$t = -((N \cdot P_o) + D)/N \cdot d \tag{2}$$

$$P = P_o + t * d \tag{3}$$

Where N is the surface normal, P_o is the starting point of the ray, D is the distance from the origin, and d is the ray direction. The next step is to check if ray and triangle are parallel and if the intersection point is inside of the triangle. The intersection is inside if it is to the left of all edges from the direction of each edge illustrated in figure 3. Which side the point is on is

done with equation 4. If the product is positive the intersection point is left of the edge. If this is true of all three edges on the triangle the intersection point is inside.

$$N \cdot (v_1 - v_0 \times (P - v_0)) \tag{4}$$

v_0 and v_1 are two vertex points on the triangle.

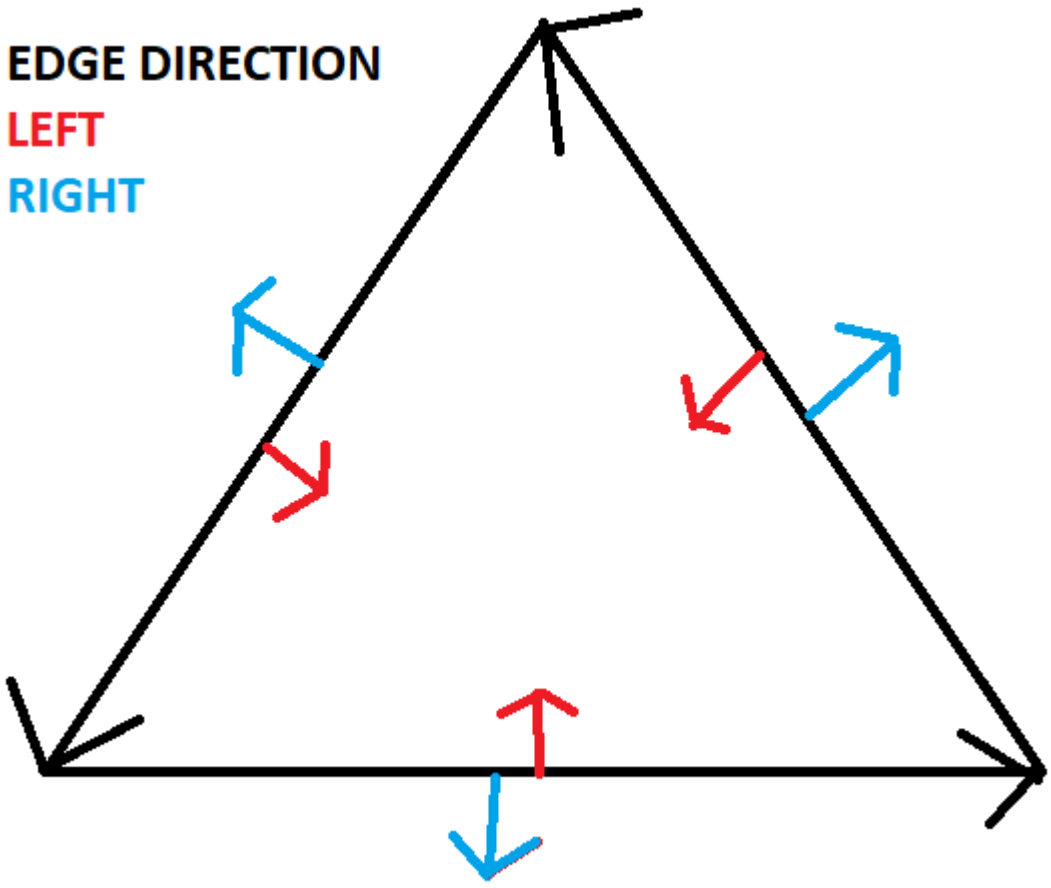


Figure 3: Left and right of the edge in their own direction.

Since all triangles has a Lambertian matieral the outgoing direction is calculated the same way. This is done by calculating a local cartesian coordinate system. The z-axis is the triangles normal, the x-axis is calculated by sub-

tracting one of the triangles vertices with another, and the y-axis is calculated using the cross product of the z- and x-axis. A random point on the half hemisphere generated by the new coordinate system is then generated which is turn into the outgoing direction.

3.4 Sphere Intersection

The ray intersection with the sphere was done differently than the triangle intersection. The method was done by following Kyle Halladay's method.[2] The first step is to create a triangle from the sphere's center and the point in the direction of the ray as can be seen below, in figure 4. The line tc is calculated using equation 5.

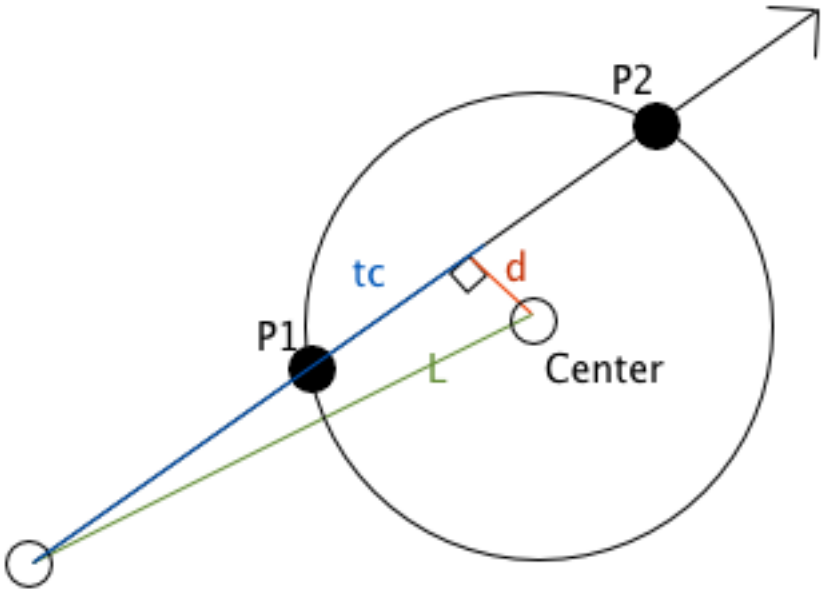


Figure 4: Ray and sphere intersection.

$$tc = (Center - RayStart) \cdot RayDirection \tag{5}$$

d can then be calculated by using Pythagoras theorem. Since the distance between the intersection point, *P1*, and the center of the sphere is the radius of the sphere, another triangle can be made. The position of *P1* is then calculated by subtracting the base leg of the new triangle from *tc*.

The bounce direction on the sphere is also different than on the triangles since it has a fully reflective material. Equation 6 is used to calculate this.

$$OutDirection = InDirection - 2 * (N \cdot InDirection) * N \tag{6}$$

The spheres normal (N) is computed by subtracting the sphere's center position from the intersection point.

3.5 Shadows

To determine if a point is in shadow, a ray is cast from the point to the light source. These are called shadow rays. If the shadow ray hits something other than the light source the point is in the shadow of the hit object. The shadows also depends on the distance and angle to the light. The light of the point is calculated using equation 7.

$$L = (c * \alpha * \beta)/d^2 \tag{7}$$

c is the color of the object, α is the dot product of the light source normal and the ray direction, β is the dot product of the hit objects normal and the opposite direction of the ray, and lastly d is the distance between the point and the light source. If the light source is a point light this is only done once but as the scene uses an area light the process must be done multiple times. Point lights are random placed on the area light and an average of these are calculated across the area of the area light. This enables soft shadows in the render.

4 Result

To create a high fidelity image the processing time increases exponentially. Multiple variables can be change which improves the image but at the cost of increasing the computing time. These variables are the amount of super samples per pixel, amount of shadow rays, max amount of bounces and the image resolution. The resulting images below uses a mix of these variables.

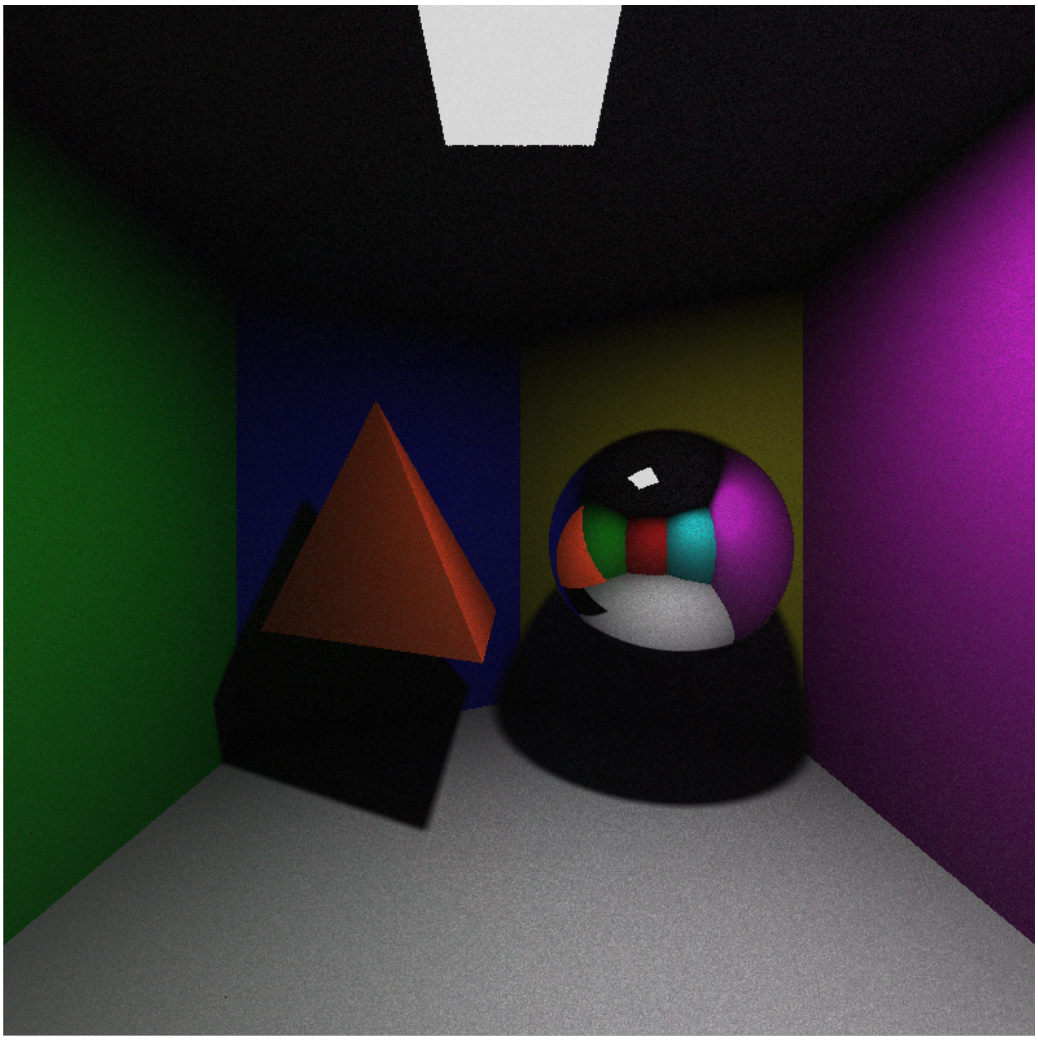


Figure 5: Rendered 800x800 image with 100 super sampling, 30 shadow rays, and a max of 8 bounces.

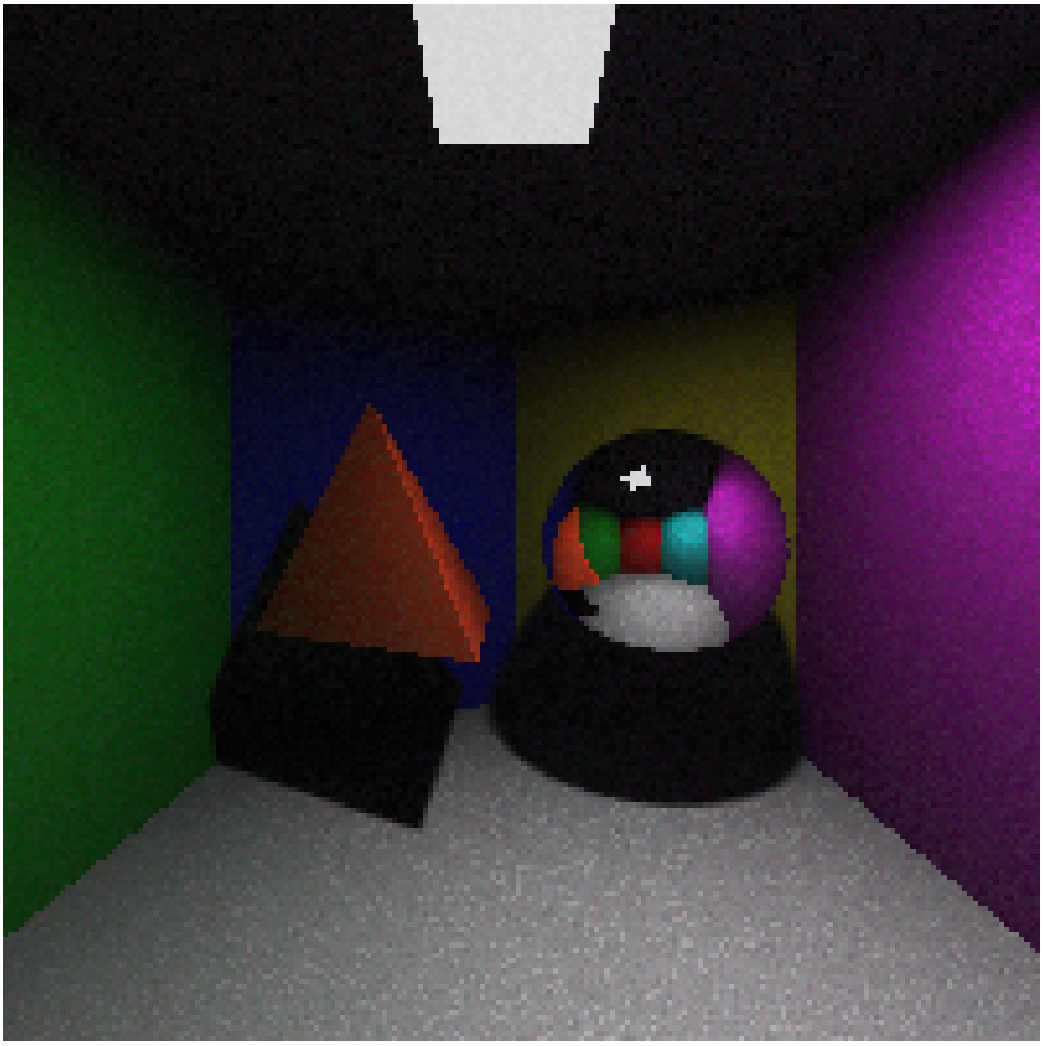


Figure 6: Rendered 200x200 image with 50 super sampling, 30 shadow rays, and a max of 6 bounces.

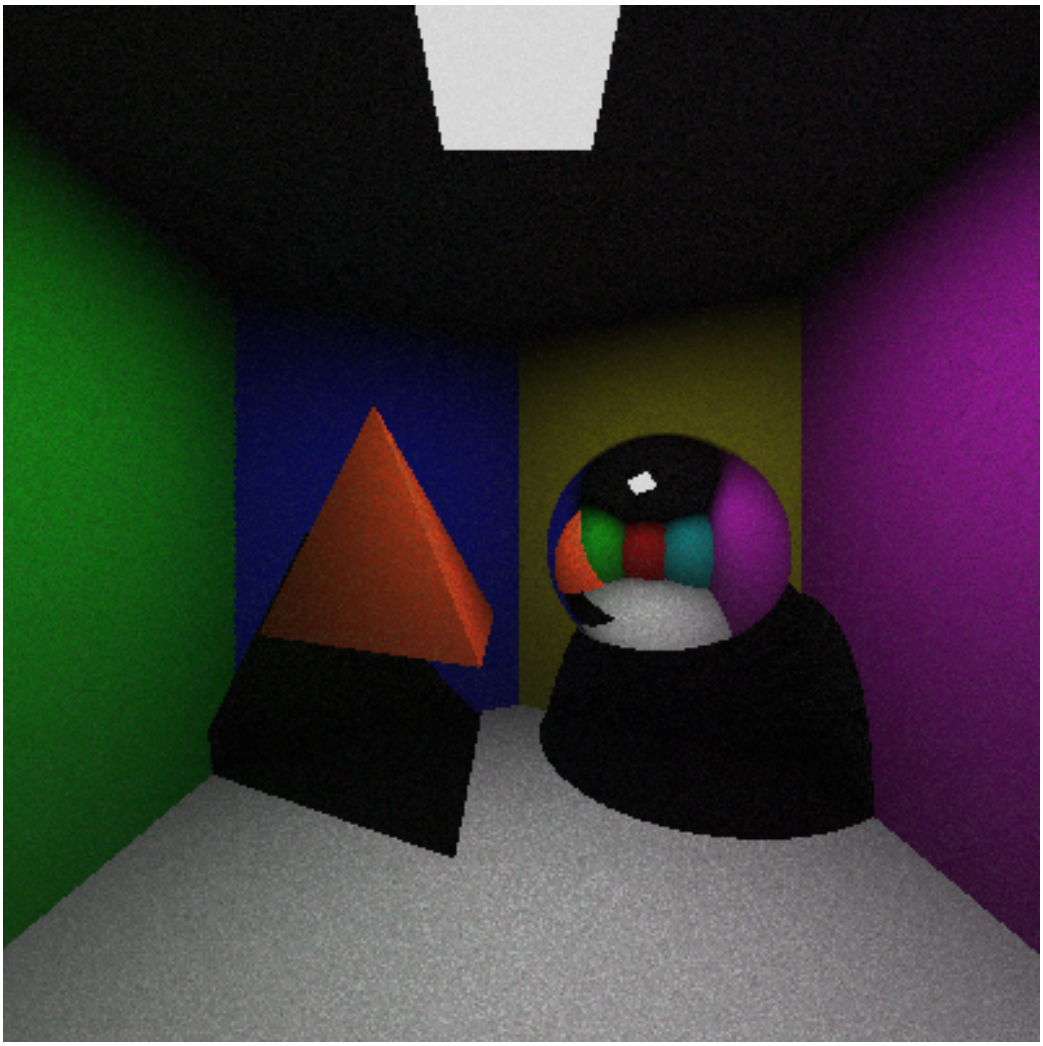


Figure 7: Rendered 400x400 image with 100 super sampling, 30 shadow rays, and a max of 6 bounces.

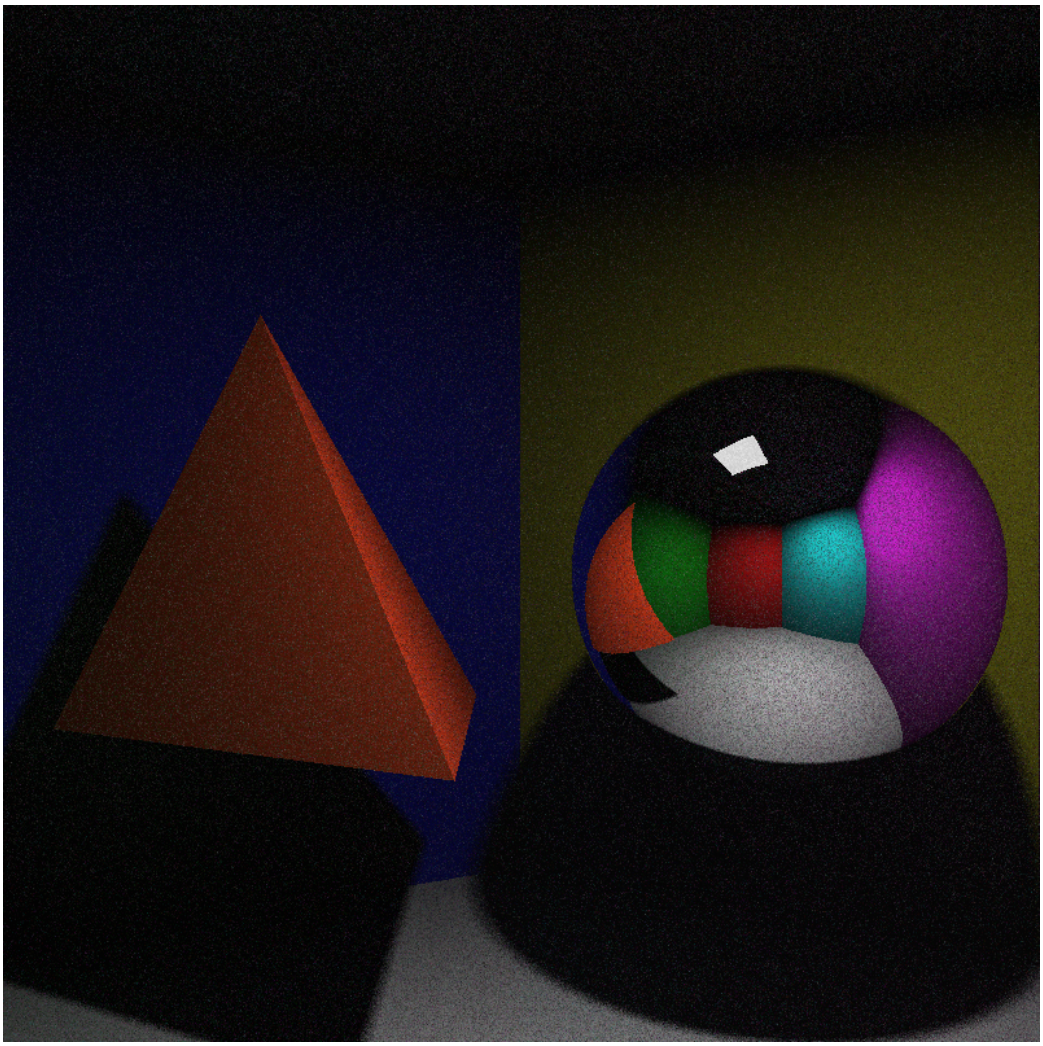


Figure 8: Rendered 200x200 image with 10 super sampling, 1 shadow ray, and a max of 6 bounces. This uses the second camera position (-2,0,0).

5 Discussion

An example of the color bleed in the results can be shown below in figure 9. This is two parts of the same image showing the floor has more green pixels

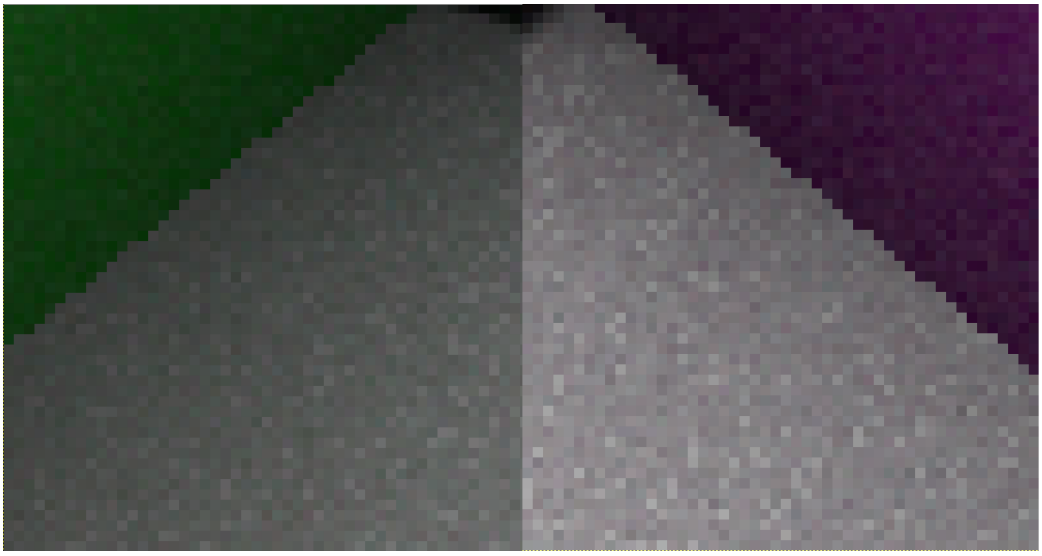


Figure 9: Example of color bleed in the resulting images.

next to the green wall and more purple pixels next to the purple wall. This is a subtle effect, and we thought the effect would be stronger, but it still makes the image look more realistic.

Photo realism can be partially subjective to the viewer and it is therefor sometimes hard to put exact measurements for what is good enough for photo realism. With that said we think that the final renders produced by our algorithms are close to photo realism as it is intended with this project. The shadows and the color bleeding is enough to create an immersive experience where they portray the real world behavior of light. Using higher variables for super sampling, shadow rays, and max bounces will make the scene more photo realistic. To further make the scene more photo realistic it would be possible to add textures to some of the materials, as no material in the real world is as perfectly smooth or matte as in the scene.

There are multiple ways to optimize the code to reduce the computation

time. As it is right now some processes that are done are redundant because it is not always used, such as if the bounce hits the sphere the code still checks if it also hits any other triangle. This is not always possible in our implementation. The only triangle it could possibly hit is the tetrahedron so trying to see if another triangle is hit is unnecessary. This is an example of what we would like to improve with more time. Since we are using single threading when computing the renders on the CPU the amount of time required to render one image could sometimes become very large. If the implementation instead were to be applied to the GPU or using multi threading the renders would be a lot faster.

Some of the problems we had was artifacts, that was created by still unknown reasons. Vertical and horizontal lines across the planes was shown, but this was fixed by using a random offset when super sampling. An example of these artifacts is shown in figure 10, below.

In conclusion, we are satisfied with the results of this project and are happy with the final renders that were produced and we believe that we have shown, in this report, that Monte Carlo ray tracing is a suitable algorithm to create realistic light behaviors in computer graphics.

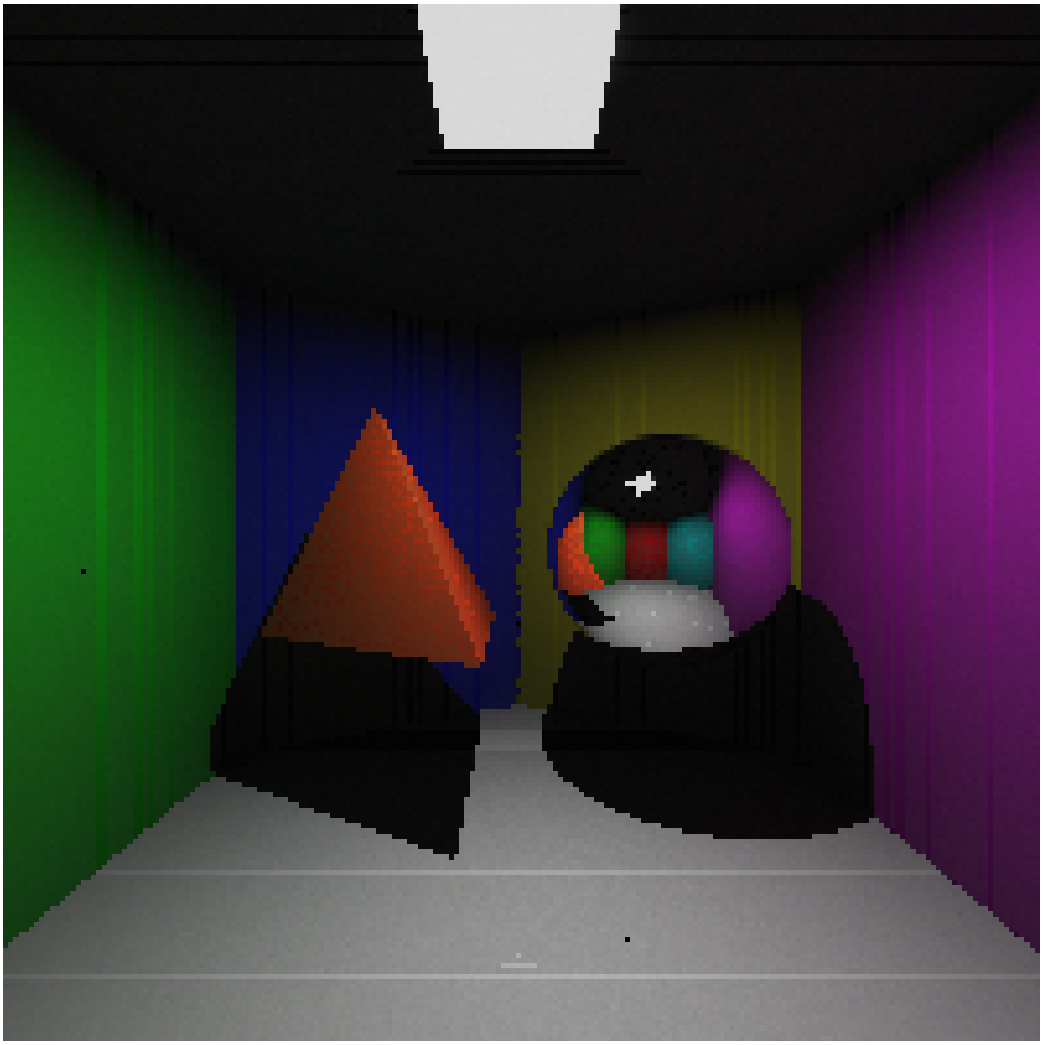


Figure 10: Example of artifacts showing up in the final image.

References

[1] James Arvo and David Kirk. Particle transport and image synthesis. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, page 63–66, New York, NY, USA, 1990. Association for Computing Machinery.

[2] Kyle Halladay. Ray-sphere intersection with simple math. Accessed: 2023-02-18.

[3] Scratchapixel. Ray-tracing: Rendering a triangle. Accessed: 2023-02-15.