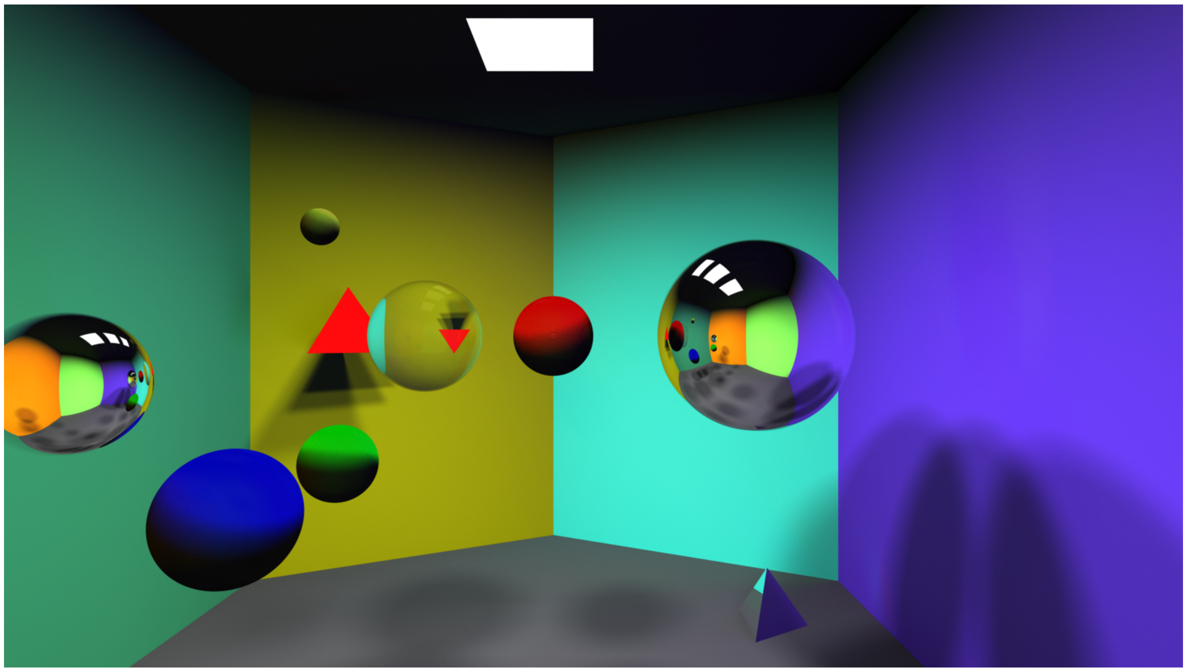
Monte Carlo path tracing - a global illumination method

Rasmus Hogslätt, rasho692



Abstract

This report covers the background and theory of Monte Carlo path tracers, using radiance as a means of realistic light transport. It also covers a practical implementation of a Monte Carlo path tracer, that results in rendered images of diffuse surfaces, ideal reflections and refractions. Resulting images and render times indicate a strong correlation between the amount of samples sent per pixel in the camera's image plane and the quality of the final rendered image. Methods for improving render times, such as multithreading and bounding volume hierarchies, as well as methods to improve image quality and simulation of other optical effects are also discussed.

Contents

1 Introduction

1. Need synthetic scenes
2. Local illumination models cannot resolve shadows and reflections
3. Whitted introduced ray tracing scheme
4. Need methods that can illuminate all types of surfaces −¿ MC

With advances in computer performance over the last decades, computers have been able to do more and more complex tasks, including rendering of synthetic scenes. These have become more important and can fulfill various purposes such as rendering movies, visualizing medical data and much more. Early illumination models provided local lighting of these scenes, such as Phong's illumination model. These however, did not account for shadows and refletions, which are proven useful for getting a sense of depth in a synthetic scene.

In 1980, Turner Whitted published a paper[Whi80] in which he proposed a method to render photorealistic images that also simulated perfect reflections and refractions. This was a significant improvement compared to previous common rendering methods, such as Phong's illumination model and its variations. It was achieved by letting rays be sent from a virtual camera, through pixels in an image plane, and into a virtual scene, as illustrated in Figure 1. If the ray would hit a diffuse object and it was also visible to a light source, the pixel would be given that color, otherwise, the object was in shadow. If it hit a mirror or transparent object, another ray could be traced into the scene from that location until a diffuse object was hit. This approach is very similar to how real light transport works, except that real light travels from a light source to the observer, and not vice versa. This difference meant that any indirect light from nearby illuminated objects was not accounted for.

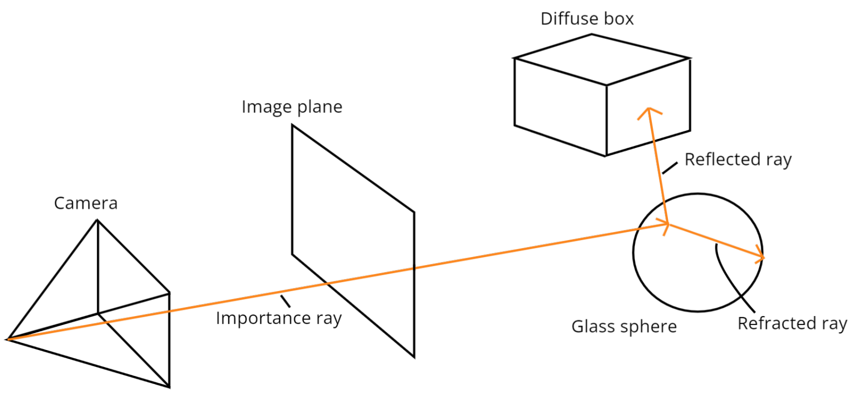


Figure 1: Initial ray sent through image plane. When hitting a glass sphere, it perfectly reflects and refracts and is sent further into the scene.

Whitted had now accounted for shadows and reflective and refractive surfaces, but still, not all types of surfaces and illuminations were possible to render accurately. For example, ambient lighting, i.e. light reflected of other surfaces were not accounted for, something which was later solved by utilizing the Monte Carlo integration technique. This method is commonly called Monte Carlo path tracing and can illuminate all types of surfaces. An implementation of this will be presented in this report.

2 Background

This chapter covers fundamental concepts of global illumination and relevant physics needed to understand and implement a Monte Carlo path tracer.

2.1 Rendering equation

In the field of computer graphics, mathematical and physical advances have resulted in what is commonly referred to as the rendering equation. This is an integral equation that is impossible to solve analytically, but if solved, would simulate real illumination very well. Thus, many different methods have been proposed to approximate this equation as well as possible. A better approximation yields a better rendering, and tricky parts of the equation can be left out if deemed to complex for a given application. The rendering equation is commonly written as

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) + \int_{S^2} L_i(X, \hat{\omega}_i) f_X(\hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| d\hat{\omega}_i \tag{1}$$

where X is a given point on a surface, $\hat{\omega}_o$ is the outgoing light direction, $\hat{\omega}_i$ the incoming light direction and $\hat{n}$ the surface normal at point X integrated over all possible incoming light directions. The L_e-term indicates emissive objects and light sources.

2.2 Radiance and radiosity

In order to approximate the rendering equation, a representation of light has to be introduced. One such representation is radiosity, which means that radiance at a surface can also be computed. Radiance is the radiosity received, emitted, transmitted or reflected at any surface in the scene per solid angle per unit projected area. Radiosity is the emitted flux divided by the area of the emitting object. Thus, a light with a small surface area yields a higher radiosity than one with a larger area, when the flux of both light sources are equal. Each light source would thus have flux given in the unit of Watts, and a shape that has an area.

2.3 Bidirection reflectance distribution function - BRDF

The f_X part of the rendering equation 1 represents how a surface is locally illuminated and is often called the bidirection reflectance distribution function or *BRDF*. The BRDF is a probability distribution function, describing the probability that an incoming ray scatters in a specific random direction within the hemisphere above the surface. It takes the incoming and outgoing light directions, which are parameterized to an azimuth and inclination angle. Thus, it is reduced to a function of four variables[RM12].

2.4 Ray properties

As seen in Figure 1, a ray has an origin and an end position. A ray can thus be represented by a starting point and an end point, which means that ray direction can be deduced by

$$ray_d = (ray_e - ray_o) \tag{2}$$

where ray_e is the ray's endpoint and ray_o is its origin. As they are also used to represent light transport, they will also carry radiance and importance.

2.4.1 Diffuse reflection

Diffuse surfaces store a color, a reflectance factor and are assumed to only receive light from above. Thus, the BRDF for diffuse surfaces simplifies to

$$f_X(\hat{\omega}_i, \hat{\omega}_o) = \frac{\rho}{\pi} \tag{3}$$

where ρ is the reflectance of the surface, and π is a normalization factor for the hemisphere above the surface.

2.4.2 Perfect reflection and refraction

Perfect reflections and refractions obey the law of conservation of energy. Thus, the radiance of a incoming ray must be equal to the sum of the radiance of the reflected and refracted rays. So a ray hitting a perfect mirror result in a reflected ray with the same radiance.

The direction of the a perfect reflection is calculated as

$$R = I - 2(N \cdot I)N \tag{4}$$

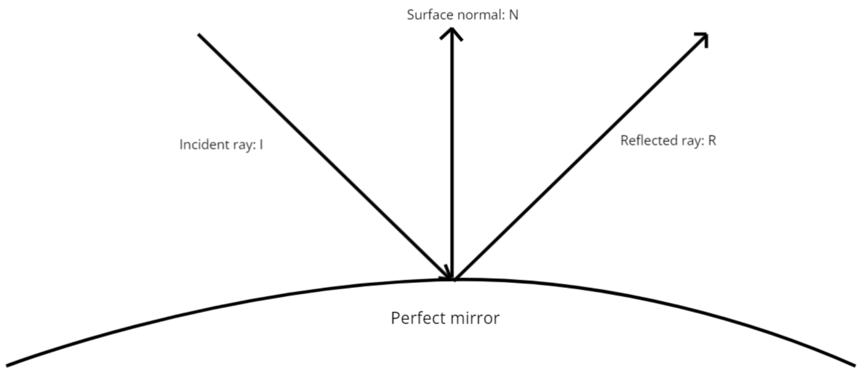


Figure 2: Incoming ray I reflects of a perfect mirror, about the normal N, resulting in outgoing ray R.

where N is the normal of the intersected surface, I is the incoming ray's direction and R is the new ray's direction, as seen in Figure 2.

The refracted ray's direction is derived from Snell's law [Fle21]

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \tag{5}$$

where θ_1 is angle between normal N and incoming ray I in Figure 2. θ_2 is the refracted ray's direction, and is what is computed. It is the angle between the inverted normal N and the refracted ray's direction. n_1 and n_2 and the indices of refraction of the two mediums that the ray is being refracted through.

An example ray path through a transparent sphere is shown in Figure 3.

For refractions, total internal reflecton also had to be computed. If the incoming angle θ was greater than the critical angle $\theta_{critical}$ given by equation 6

$$\theta_{critical} = \arcsin\frac{n1}{n2}, n_1 < n_2 \tag{6}$$

where n_1 and n_2 were the indices of refraction of the mediums, there would not be any refraction.

2.4.3 Fresnel equation

The law of conservation of energy means that both the reflected and refracted rays can not have an importance of one, despite being perfectly reflected and refracted. They obey the fresnel equation, which describes the importance of the rays depending on the angle of incidence. This equation requires a significant amount of computational power. Thus, an approximation, called Schlick's approximation[Maj21] is commonly used.

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{7}$$

where

$$R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2 \tag{8}$$

and θ is the incident ray's direction relative to the surfaces normal, and n_1 and n_2 and the indicies of refraction. $R(\theta)$ is a coefficient for the importance of the reflected ray. The refracted ray's importance is given by subtracting $R(\theta)$ from one. This equation is an approximation, but is faster to compute and still yields okay results.

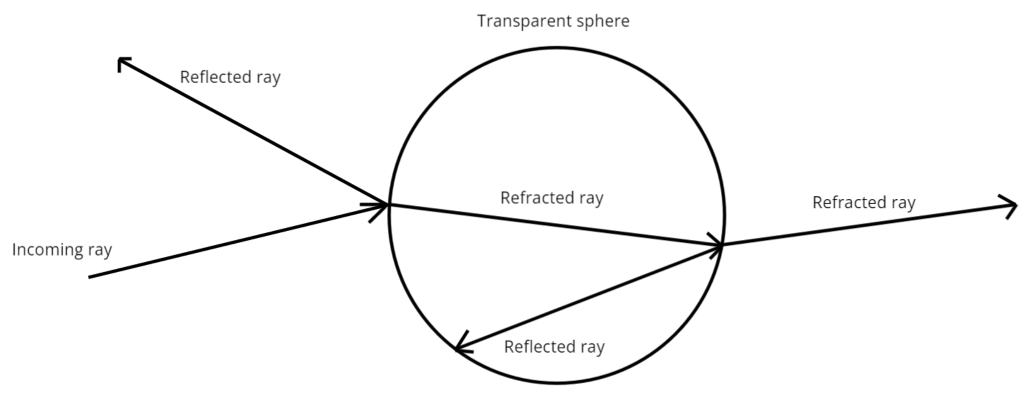


Figure 3: Incoming ray I reflects of a perfect mirror, about the normal N, resulting in outgoing ray R.

2.5 Lambert's cosine law

In the rendering equation, the term $|\hat{\omega}_i \cdot \hat{n}|$ comes from Lambert's cosine law which states that the radiosity at an ideal diffuse surface is given by the cosine term between the incoming light direction and the surface normal. This means that a ray hitting a surface at a close to perpendicular angle, will have a lesser contribution to the surface's illumination than one hitting the surface parallel to the surface's normal. Due to the conservation of energy, any negative values are set to zero.

2.6 Monte Carlo integration

The integral part of the rendering equation is the trickiest part to compute. Computers can not solve integral equations analytically, as their memory and computational power is finite. Therefore, methods of approximating integrals exist, and one such method is the Monte Carlo integration method. It approximates integrals by averaging the sum of randomly chosen samples. The more samples, the more likely the result is going to converge to its analytical solution.

2.7 Monte Carlo path tracing for indirect light

The integral in the rendering can be approximated by using the Monte Carlo integration method. When a ray hits a diffuse surface, by letting it randomly reflect within the hemisphere above the surface, and normalize its importance by accounting for the surface's BRDF, the integral can be approximated. Furthermore, sending more rays through each pixel in the image plane would make the approximation of the integral converge to its actual value, but would increase the computational cost.

2.8 Ray termination

Russian roulette is a common method to determine when a ray should be terminated, which is needed to avoid spawning an infinite amount of rays. Given a probability p, it determines whether a ray should be reflected or not. For a mirror, the probability of spawning another ray would always be one, but for a diffuse surface, this probability would be given by a randomly generated azimuth angle in the hemisphere above the surface. If another ray is spawned, its importance would be scaled by the factor $\frac{1}{p}$ to compensate for errors.

2.9 Ray-Object intersections

Rays must be able to intersect objects within the scene. Thus, methods to determine if a ray intersects each type of object shape must be implemented. If an intersection is found, the intersection point is given by

$$x(t) = ray_o + t \cdot ray_d \tag{9}$$

where $x(t)$ a point along the ray and t is the distance to that point. However, intersection computations can be terminated early if culling is assumed. Thus, computing the dot product of the objects normal n and the ray's direction ray_d means that any results greater than zero means that the object should not be rendered.

2.9.1 Implicit sphere intersection

A sphere can be described in implicit form as

$$(P - C)^2 - R^2 = 0 \tag{10}$$

where P is a point on its surface, C is the origin of the coordinate system and R is the sphere's radius. Replacing P with equation 4, yields an equation where t can be solved for according to equation 6.

$$(ray_o + t * ray_d)^2 - R^2 = 0 \tag{11}$$

The results is two values for t, where the smallest non-negative value is chosen. This is because a negative value would mean that the intersection point lies behind the ray's origin, which is not wanted. Having one negative value means that the ray's origin was inside the sphere, which is useful to know for transparent objects.

2.9.2 Triangle intersection

One method for computing triangle intersection is the Möller-Trumbore algorithm. A point on a triangle's surface can be represented as

$$x(u, v) = (1 - u - v)v_0 + uv_1 + uv_2 \tag{12}$$

where $x(u, v)$ is the point on the triangle, v_0, v_1, V_2 are its vertices and u and v are the points coordinates in the plane.

Substituting with equation 4 and applying Cramer's [Sta21] rule and computing edges of the triangle results in equation 8

$$t = \frac{\vec{Q} \cdot \vec{e_2}}{\vec{P} \cdot \vec{e_1}} \tag{13}$$

$$\vec{e_1} = \vec{v_1} - \vec{v_0} \tag{14}$$

$$\vec{e_2} = \vec{v_2} - \vec{v_0} \tag{15}$$

$$\vec{Q} = (\overrightarrow{ray_o - v_0}) \times \vec{e_2} \tag{16}$$

$$\vec{P} = \vec{ray_d} \times \vec{e_2} \tag{17}$$

where t is the intersection depth, u and v triangle coordinates given by Cramer's rule, and e_1 and e_2 are two edges of the triangle with a common vertex. Q is a vector orthogonal to e_2 and vector between ray's origin and the vertex shared by e_1 and e_2. P is a vector orthogonal to ray_d and e_2. For an intersection to occur, u and v must fulfill certain conditions in equation 9.

$$u \geq 0 \tag{18}$$

$$v \geq 0 \tag{19}$$

$$u + v \leq 1 \tag{20}$$

$$t \geq 0 \tag{21}$$

2.9.3 Other objects

Other types of objects such as tetrahedrons, rectangles and cubes can be constructed using triangles. Thus, intersection with these objects only need to call the intersection routine of the triangle for all triangles making up the object and determine which, if any, intersection is the closest.

2.10 Light intensity and shadows

In the physical world, light sources always have a shape, meaning that they also have an area. Thus, when a ray hits an object in the scene, a ray is sent randomly towards some point on the light's visible surface. If no objects were intersected between the casted ray and the light source, the area is illuminated and its intensity is scaled according to the inverse square law[Bro14]. This means that the intensity drops of as the distance between the light source and object increases. If an object was intersected inbetween, then the ray was in shadow, and no intensity was calculated. Sending rays from the object to a random point on the light source yields soft shadows. If an object is only partially in shadow, this will then be depicted in the rendering.

2.11 Anti aliasing

In path tracing, several rays are sent through each pixel and the resulting color is averaged. This improves the result, but some aliasing can still appear. In order to minimize aliasing, sending the rays at different points through each pixel reduces these artifacts. Thus, generating rays is typically done by randomly sampling each pixel in the image plane according to Figure 2.

3 Implementation

This section explains how the theory in the background was used to implement a Monte Carlo path tracer. The code for the project was written in C++ 17 and rendered to an OpenGL texture, which was then exported to a PNG file. A graphical user interface for easy scene manipulation was also added using ImGui.

3.1 Scene

The virtual scene was constructed as a hexagonal room. The walls and roof were made of rectangles, and the room was then filled with spheres and tetrahedrons. The walls were given diffuse materials and the spheres diffuse, mirror or glass materials. There were two tetrahedron in the scene, one mirror, and one diffuse. Three square area lights were added by the roof and a camera was positioned in the back of the room, at a slight offset to the right with a 55 degree field of view.

3.1.1 Direct scene illumination

The scene was directly illuminated by the three light sources in the room. If a ray hit a light source before anything else, it would terminate and the ray was given the radiosity of the light. If it hit a diffuse surface, shadow rays were sent towards all light sources and if a diffuse object was positioned along the shadow ray, the ray would not contribute to the final color. For mirrors and transparent objects, reflections and refractions were calculated and the new rays were traced. The reflected light off of diffuse surface was calculated as

$$\theta_r = \sum_{i=1}^{N} \frac{max(\alpha \cdot \beta \cdot \theta_i, 0)}{||shadowRay_i||^2 \cdot N} \tag{22}$$

$$\alpha = -\hat{dir_{shadow}} \cdot \hat{normal}_{light} \tag{23}$$

$$\beta = \hat{dir_{shadow}} \cdot \hat{normal}_{object} \tag{24}$$

where θ_r is the reflected intensity at the object and N is the number of lights. If the dot product is negative, it is set to zero as to adhere to the law of conservation of energy.

3.1.2 Indirect scene illumination

When a ray hit a diffuse object, besides computing direct lighting, russian roulette on the azimuth angle would determine whether to send another ray into the scene. The radiance of this ray would then be added to the previous ray, and thus cause indirect lighting from other objects to occur. The indirect lighting was calculated using

$$L_{total} = L_{currentRay} + \sum \frac{L_{childRay} \cdot W_{childRay}}{W_{currentRay}} \tag{25}$$

where L_{total} is the total radiance, $L_{currentRay}$ is the radiance given by the direct lighting at the current ray, $W_{currentRay}$ is the importance of the current ray. $L_{childRay}$ and $W_{childRay}$ are the corresponding values for the current ray's child ray.

3.2 Camera

The camera implemented was a perspective camera with a property to vary the horizontal field of view, or *FoV*. From the horizontal FoV, the vertical FoV was computed using 26

$$FoV_{Vertical} = \frac{height}{width} \cdot FoV_{Horizontal} \tag{26}$$

and the points in the image plane could then be computed using equation 27 and 28

$$x = \frac{2 \cdot u - width}{width} \cdot \tan FoV_{Horizontal} \tag{27}$$

$$y = \frac{2 \cdot v - height}{height} \cdot \tan FoV_{Vertical} \tag{28}$$

where u and v are the pixel coordinates as integers and the z-component is set to negative one relative to the camera. This means that a ray can constructed, starting from the origin of the camera with direction towards the coordinates in the image plane. Using equations 27 and 28, a random offset was calculated and scaled according to the size of a pixel. This allowed for introducing anti-aliasing, as described in previous section.

3.3 Shapes

The abstract base class *Shape* was implemented with pure virtual functions to access information such as normals and positions and most importantly, a function to check if a ray intersected it. Derived classes for implicit spheres and triangles were then implemented. Triangles would then be further derived into rectangles and tetrahedrons.
As the intersection calculations were executed on a computer with floating point precision, a small bias was added to ray's origin in the direction of the normal. Thus, a ray that intersected a surface would not send a ray that intersected with that point the surface again. For refractive objects, a check was done to see whether the ray started from within the object or not. If so, the bias had to be added towards the inverted normal.

3.4 Material

Another base class called *Material* was implemented which was derived diffuse surfaces, mirrors and transparent objects. Each material was given a function to get the f_X in the rendering equation 1. For diffuse surfaces, this returned $\frac{\rho}{\pi}$ where ρ was the material's reflectance. Mirrors and transparent object were assumed to be perfectly reflecting and refracting, and would thus not utilize this function.

3.5 Objects

Objects was the base class later added to the scene. An object was given a pointer to a shape and a material. Thus, various types of objects could be added to the same scene vector, which allowed easy manipulation of the scene.

3.6 Lights

Similar to the *Object* class, the *Light* class also had a pointer to a shape, and also a value for flux and a function for getting its radiosity.

3.7 Ray

The was constructed using a vectors that stored its origin, end point, radiance and importance. From the origin and end point, its direction could be derived. It also stored pointers to its parent ray and child ray. The first ray was constructed by the camera with origin set to the camera's origin, and the end point set to a point in the image plane. The ray would be traced into the scene and be terminated by russian roulette on a diffuse surface or when nothing was intersected. However, the latter never occured as the scene was a closed room. For intersected object, calculations for radiance were computed as described in previous sections. If a ray intersected a transparent sphere, the dot product between the ray's direction and the object's surface normal was used to determine whether the ray started within or outside the sphere.

4 Results

The final implementation of the Monte Carlo path tracer rendered images seen in Figure 4, 5, 6, 7 and 8. All images were rendered at a resolution of 1920x1080 pixels. Figure 4 through 7 were the same scene rendered with an increasing amount of rays sent per pixel and Table 1 one shows the render time for each of these renders. Figure 8 is the same scene, with the camera positioned at a corner and with and added light rendered at 5 samples per pixel to more clearly visualize the indirect lighting of close surfaces.

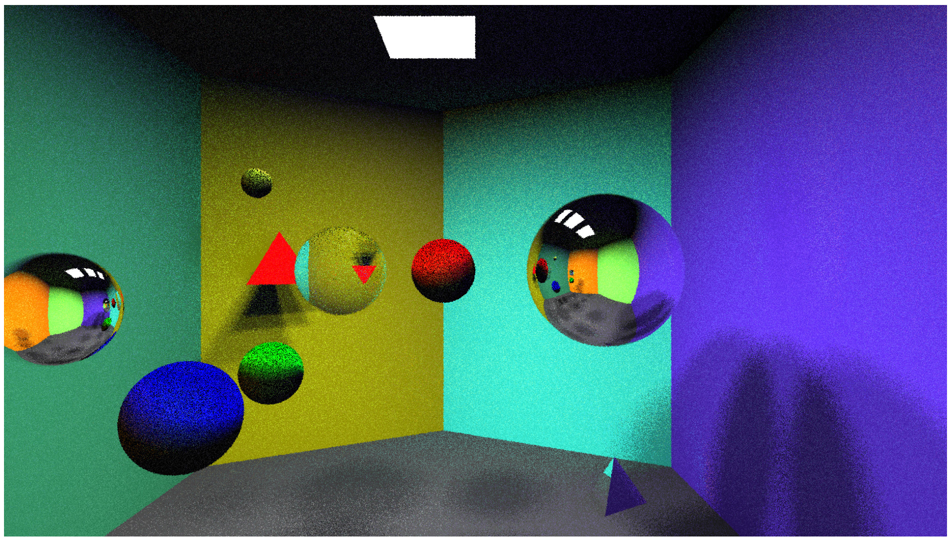


Figure 4: Scene rendered at 1 sample per pixel.

Samples per pixel	1	5	10	1000
Render time [seconds]	5	17	40	3940

Table 1: Render times

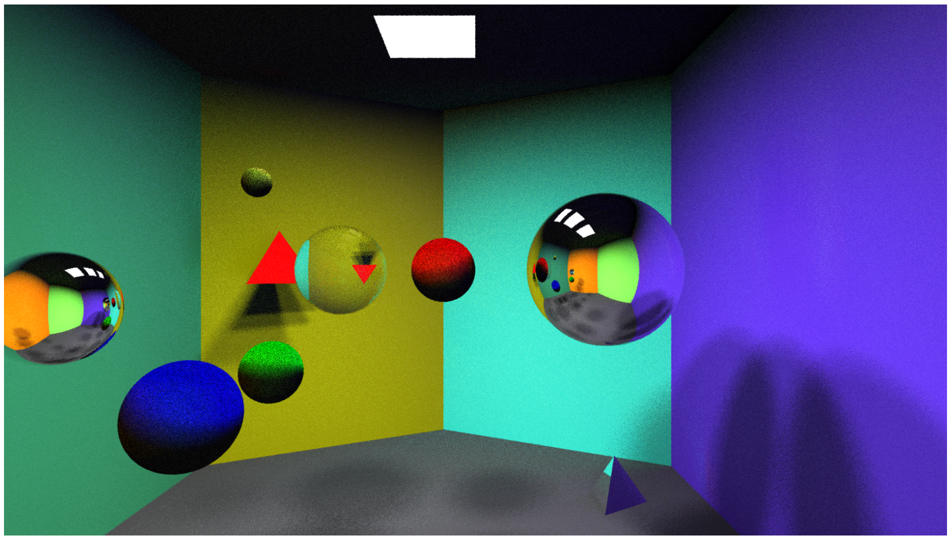


Figure 5: Scene rendered at 5 sample per pixel.

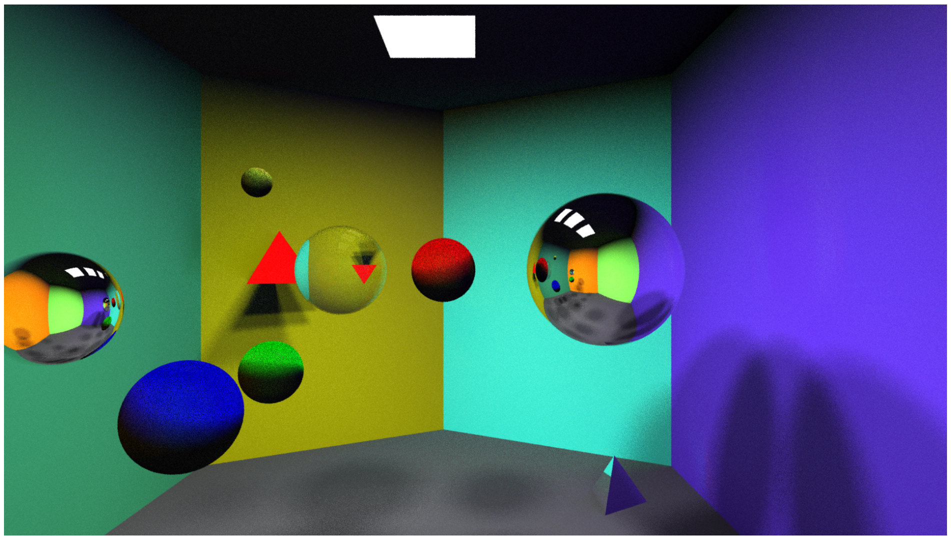


Figure 6: Scene rendered at 10 sample per pixel.

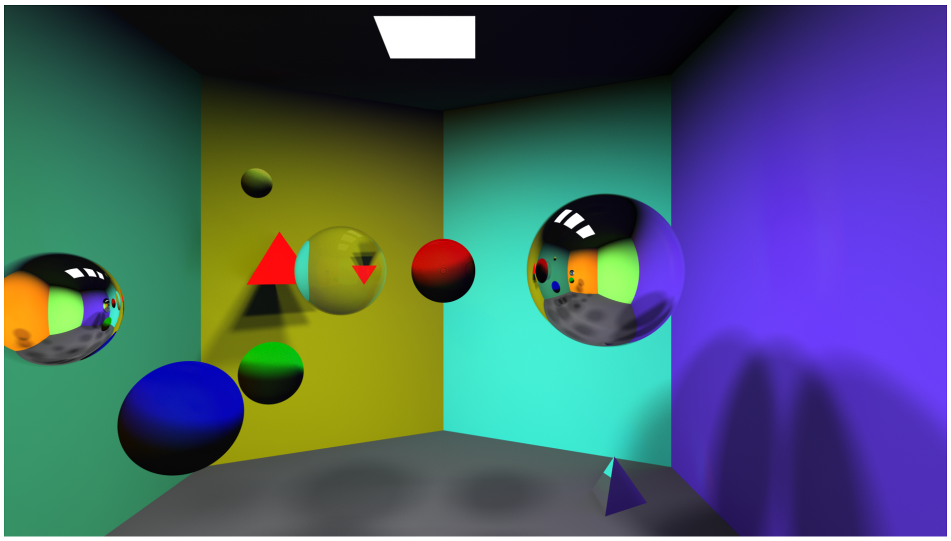


Figure 7: Scene rendered at 1000 sample per pixel.

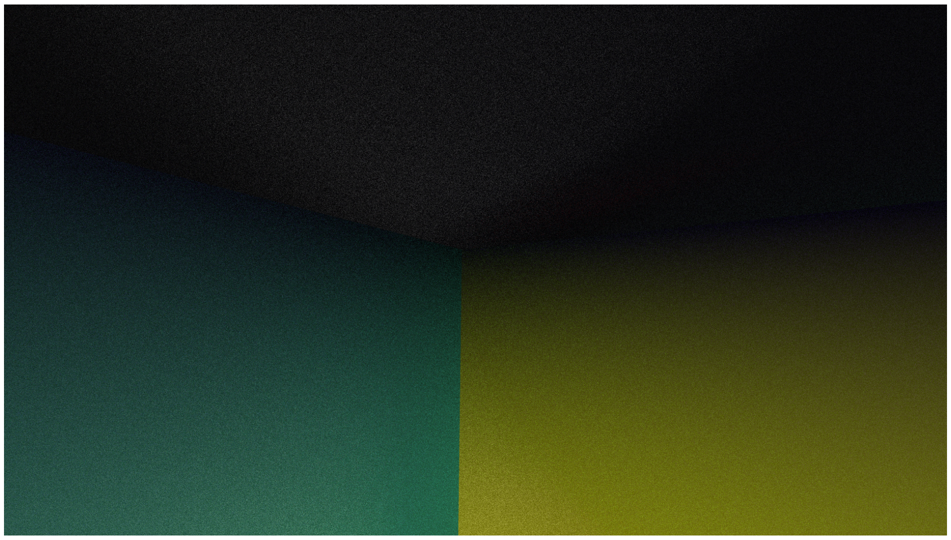


Figure 8: Close up render of a corner. Each face's color is affected by nearby objects. A light was added below, causing bright spot on the ceiling.

5 Discussion

The implementation of the Monte Carlo path tracer produced images with realistic reflection, refractions and illuminated diffuse objects directly and indirectly.

5.1 Samples

An obvious conclusion of the resulting images is that an increasing sample rate drastically reduces noise in the image, but also increase the time it takes to render the image. This is due to the approximated integral in the rendering equation converging closer to the integrals true value due to the randomness of Monte Carlo integration. The strong pepper noise when rendering with one sample per pixel is thus likely due to early ray termination, which means that no more light contribution would be given to those pixels. When more samples per pixels are used, having one sample returning no radiance, because of being in shadow, and another sample returning a bright color, would average out and still result in a colored pixel. So increasing the sample count even more would then cause a lower variance of the final pixel's color across different amount of samples.

5.2 Render times

5.2.1 Multithreading - CPU

In table 1, it appears, as discussed above, that render times are drastically increased with increased sample count. The current implementation of the path tracer is using one processor thread for rendering the entire image. Thus, only one sample can be computed at any given time. Methods for decreasing render times, without compromising quality of the rendered image, would be to use several threads of the processor. For example, the image plane may be divided into a queue of squares covering the entire image plane. Thus, several squares may be render at once, and when finished, a free thread could start rendering the next square in the queue.
Using this approach would likely cause a significant decrease of render times, but even then, one might want to improve this further. An issue with that approach is that one square might be very complex to compute compared the other squares. A diffuse surface could cover most of the image plane's window, whereas only a small section of the image plane's window may contain a several refractions, that require more computations. This could mean that one may want to further experiment with dividing the workload of each thread, to yield the fastest render times.

5.2.2 Multithreading - GPU

Similarly to using several threads of the CPU, one could also utilize the graphics processor of the computer to render using significantly more cores. A drawback with graphics processors is that they contain their own memory, and it's costly to send data back and forth between the CPU and GPU. Thus, each core of the GPU may not have access to locations of objects in the scene, causing intersection calculations to be much more tricky to compute. One could work around this issue by using textures, buffers or even using compute shaders. This would require a massive refactoring of this implementation, but given time, that would be an interesting comparison to do.

5.2.3 Bounding volume hierarchies and scene structures

Yet another way of decreasing render times would be to utilize bounding volume hierarchies BVH or similar structures. These allow objects within close proximity to be grouped together, and any intersection calculations can be made for the entire groups. If a group of objets are not intersected, that group can be ignored. For a group of five objects, that would mean that checking intersections five times would be reduced to only once. If a group would intersect though, intersections within that group could be made. In a nutshell, this means that objects located behind the ray's origin would never be checked, which makes sense, as the ray is directed in the opposite way.

5.2.4 Machine learning denoising

With the advances in machine learning, several methods have been introduced to reduce noise in the image. This allows, low samples rates, whilst still minimizing noise. However, those algorithms, such as NVIDIA's optix ai-accelerated denoiser[Ail17], only try to predict what color a pixel should be in pixel-space. Thus, they would most likely break common laws of optics and thereby also break photorealism. However, the Monte Carlo integration is only an approximation, that converges closer to the actual integrals value as samples increase. Therefore, neither AI-denoising nor pure Monte Carlo path tracing actually follows all laws of optics. It would be interesting to compare the photorealism between the two methods in future studies.

5.3 Indirect lighting

As seen in Figure 8, and perhaps also Figure 4, indirect light is illuminating objects close to other objects. This is a major improvement over methods such as Whitted's[Whi80] raytracing method, which only included direct lighting. The effect is very subtle, but still slightly visible. For example, the floor in Figure 4 is visibly slightly colored by the cyan wall in the back.

5.4 Solid angle

The method for solid angle in the results are using cosine-weighted solid angles. However, there are more ways to compute solid angles, that may yield different results. This could be experimented with further if given time. For example, letting the indirect rays be uniformly distributed within the hemisphere may cause the indirect light to appear more visible, but may be less accurate. There are also methods, such as power cosine-weighted sampling of the hemisphere, that may yield even more accurate results[Ame22].

5.5 Conclusion

Several features are still missing in the Monte Carlo path tracer. For example, it does not account for polarization of light, depth of field, motion blur or caustics. These are effects that could be further implemented. A common method for handling caustics is photon mapping. A more advanced camera may also be implemented to get effects such as depth of field and motion blur.

References

[Ail17] Chakravarty R. Alla Chaitanya (NVIDIA) Anton Kaplanyan (NVIDIA) Christoph Schied Marco Salvi Aaron Lefohn Derek Nowrouzezahrai (McGill University) Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. https://research.nvidia.com/publication/2017-07_interactive-reconstruction-monte-carlo-image-sequences-using-recurrent, 2017.

[Ame22] Alexander Ameye. Sampling the hemisphere. https://alexanderameye.github.io/notes/sampling-the-hemisphere/, 2022.

[Bro14] Jeffrey R.S. Brownson. Light decreases with distance. https://www.sciencedirect.com/topics/engineering/inverse-square-law, 2014.

[Fle21] Hannah Flens. Snell's law. https://eng.libretexts.org/Bookshelves/Materials_Science/Supplemental_Modules_(Materials_Science)/Optical_Properties/Snell, 2021.

[Maj21] Zander Majercik. The schlick fresnel approximation. https://link.springer.com/chapter/10.1007/978-1-4842-7185-8_9, 2021.

[RM12] Carlos Ureña Rosana Montes. An overview of brdf models. https://digibug.ugr.es/bitstream/handle/10481/19751/rmontes_LSI-2012-001TR.pdf, 2012.

[Sta21] Elizabeth Stapel. Cramer's rule. https://www.purplemath.com/modules/cramers.htm, 2021.

[Whi80] Turner Whitted. An improved illumination model for shaded display, 1980.