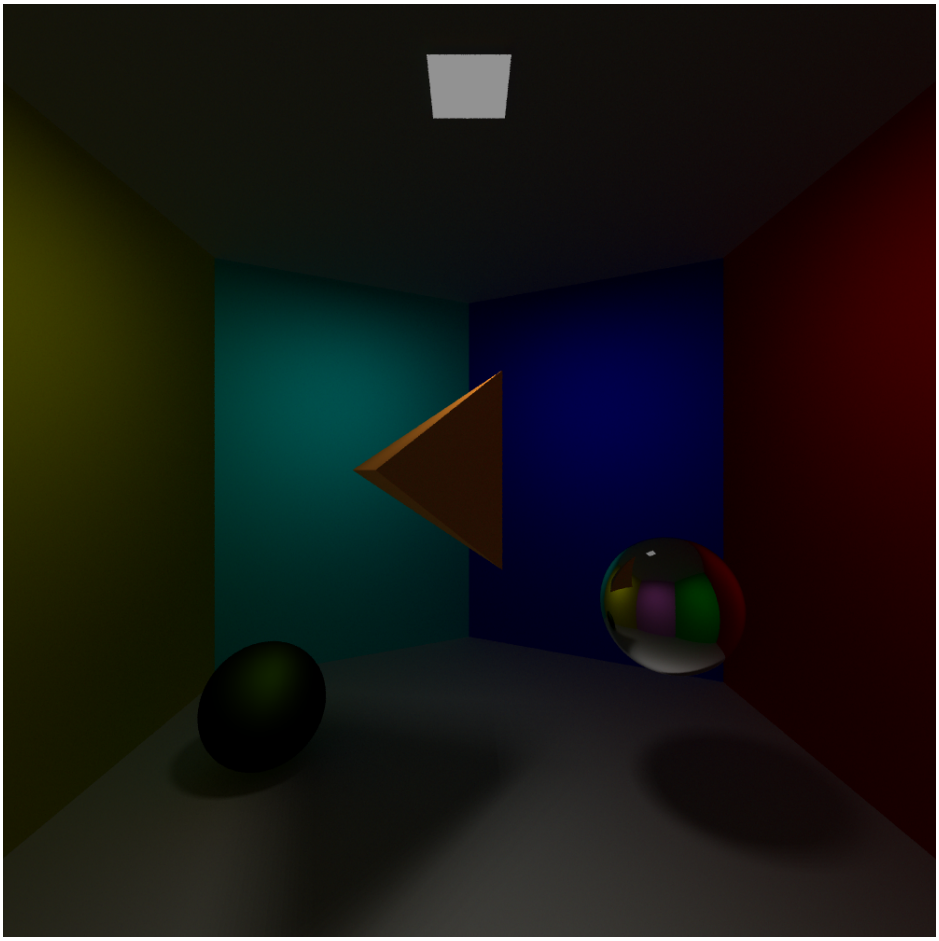
TNCG15 Report

Viktor Sjögren, viksj950
William Malteskog, wilma366
Anton Lindskog, antli559

August 9, 2022



Abstract

A global illumination ray tracing implementation of a Monte Carlo ray tracer is discussed within this report. The project involves overcoming obstacles in many shapes and forms. There will be some discussion about the interesting effects of various global illumination aspects like color-bleeding, shadow rays and shadow acne. The paper will also cover topics such as photon mapping as a theory section. However, not all of the theory discussed is implemented in the project, but rather acts as comparison points of what could improve the current implementation. The implementation was done in C++ with self-implemented vector operators and matrix operators.

1 Introduction

As we all know, we see things because of light rays being emitted from a certain light source such as a lamp or the sun hits different surfaces and bounces into your eyes. The light that reaches into your eyes could be either direct- or indirect illumination, depending on if the light bounced directly from the surface into your eyes after being emitted from the light source or if it rather has bounced several times on different surfaces before reaching your eye respectively, as seen in Fig 1 (1). The visual effect of this is that the surfaces that's not directly exposed to any light source and thus becomes shadowed, and yet they are not completely black, as there is some light that bounces and contributes with a dampened light brightness as a result.

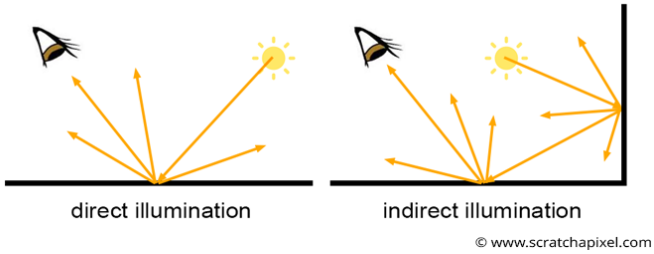


Figure 1: The left image represents direct illumination and the image to the right represents indirect illumination.

Another effect of indirect illumination is that colors also gets cast from the surfaces that the light bounces of. This can be seen on Figure 11, where the green color from the right wall is being cast onto the sphere on the right side of the image. This effect is called color bleeding.

These are two fundamental aspects of global illumination which is a key in order to produce realistic images. The necessary procedures to simulate this is rather simple, but it comes with a great cost. The simulation could be complex to solve and is generally quite expensive since it requires a lot of light rays to get a realistic result, which each requires a hefty amount of equation. The reason of this will hopefully become clearer throughout the report.

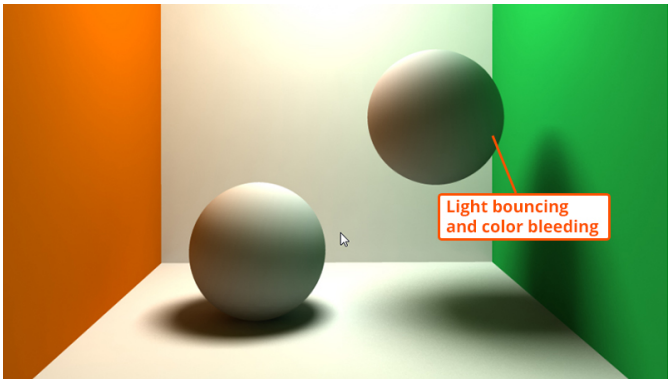


Figure 2: Visualisation of color bleeding from indirect illumination.

1.1 Photons

To achieve the effects of global illumination, it relies on the use of so called photons to optimize and handle caustics. Photons are small particles that each stores a packet of energy that contains different kinds of information. A photon is emitted from a light source and once it first hits a surface, it stores the information such as surface color and energy value of that surface. The photon then carries this packet of information to the next surface it collides with and continues bouncing around in the scene until it's absorbed (2).

1.2 Whitted Ray-Tracing

One of the earliest version of ray-tracing was implemented by Turner Whitted in 1979 and published in 1980 in his paper "An Improved Illumination Model for Shaded Display". In comparison to earlier Illumination models, such as Phong's illumination model, reflections and refractions could now be visualized using Whitted ray-tracing. This is possible by using established physical laws to describe how rays reflect off surfaces and how rays refract through transparent objects using Snell's law. In the case of a ray hitting a reflective surface, the algorithm send out a reflected ray in a direction that depends on where the light is coming from and how the surface is oriented. If a ray hits a refractive surface, the model would send out a reflective ray as well as a refraction ray. The direction of the refracted ray does not only depend on the surface orientation and light direction but also the refractive index of the material. When rays hit diffuse surfaces, a local lighting model

such as Phong's model can be used to get the color of the surface. In this case a shadow ray would be cast out from the intersection point towards the lightsource to check if anything any object is shadowing the intersection point.

What makes Whitted ray-tracing unique is that the algorithm looks at light transport backwards, in other words the rays are traced from the camera, to the light source in the scene. This is quite different than how we would think of light transport at first glance, but the backward approach proves to be more efficient in the context of computer graphics (5).

1.3 Monte Carlo Ray-Tracing

When a ray intersects a diffuse surface, in order to accurately render its color Monte Carlo Integration is applied. A diffuse surface is likely to scatter light in rather random directions, by creating a hemisphere at the point of intersection the distributed light can be calculated by integrating over the hemisphere surface. The integral can be approximated by sending random rays using the hemisphere to calculate the directions. The amount of rays sent is connected to how correct the replication become. A *BRDF* is then used to calculate the appropriate amount of light received from each ray (3).

2 Background

2.1 Caustics

When light is specularly reflected or refracted onto a diffuse surface, certain light patterns may occur. For example the patterns on a pool floor when the light is refracted in the water or when light is shining through a glass vase. These patterns are difficult to render correctly and this is where photons come of good use (13).

2.2 Scene structure

To simulate global illumination a scene was built up, consisting of six walls, floor and roof. Within these walls two sphere objects along with one tetrahedron object was defined. These parts are all except for the spheres defined in terms of triangles. Every surface in the scene also has a material attached

to it and a certain reflective and absorptive attribute. At the point (5,0,5) a quadratic lightsource is placed. A sketch of the scene is seen in Figure 3.

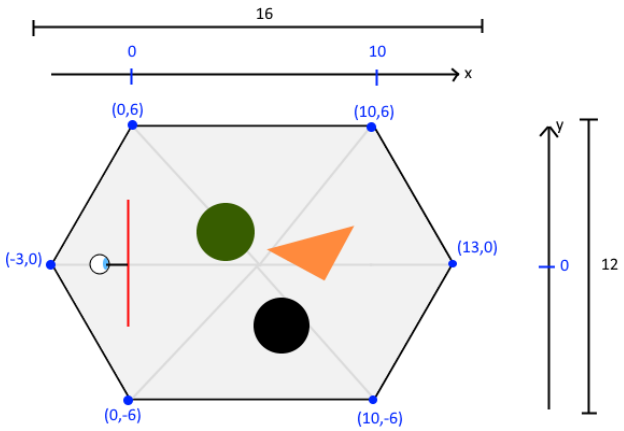


Figure 3: Scene structure in world coordinates

Each scene object has a different color defined in *RGB* values. These colors were chosen as unique for each object in order to see the effects of global illumination as much as possible (Such as color bleeding).

2.3 Camera

The camera plane is essentially the window between the eye and the scene. The camera allows the scene to be rendered using a finite number of samples and with regards to the angle to the eye. For every sample on the camera plane a ray from the eye to the camera is sent, and this ray then continues out into the scene.

2.3.1 Supersampling and ray randomization

By just doing this for every pixel on the camera, the final image would be able to replicate the scene fairly well if the resolution is sufficient enough. However only sampling one ray from one pixel has the issue of creating aliasing effects. This is generally speaking unwanted as it is not a replication of the scene, but rather a biproduct of the limitations of modern hardware (undersampling). This can be reduced by sampling more than one ray per

pixel. This means that instead of sending out just one ray from every pixel on the plane, multiple subpixels within every pixel are created and each of these subpixels then send out a ray. The resulting color of the main pixel will be the average of the sum of the subpixels colors (9).

To further reduce arising aliasing effects, the ray which each subpixel shoots out has a random starting point on the subpixel. This reduces potential unwanted pattern artefacts that could also contribute to aliasing effects. Examples of supersampling ray and ray randomization can be seen in Figure 4 and 5.

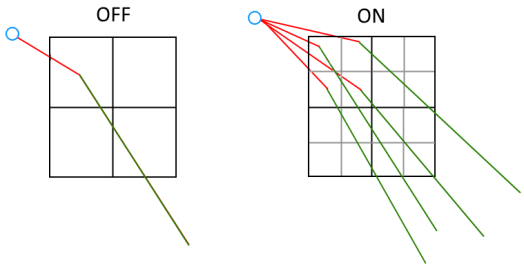


Figure 4: Supersampling with 4 subpixels per pixel

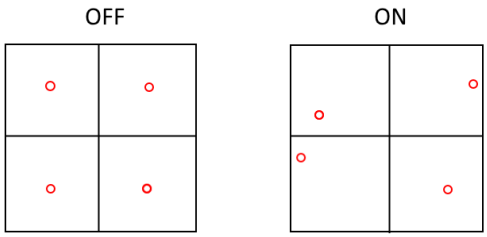


Figure 5: Ray randomisation on each subpixel

2.4 The Rendering Equation

To render the final image, the pixel radiance is computed by averaging the outgoing radiance returned from the first intersection of the camera ray by each sub-pixel. A simplified version of the rendering Equation 1 is used to calculate the outgoing radiance L_o on the intersection point x along a

direction ω_o by adding the surface's emitted radiance with the integration of all incident radiance (incoming light) over a hemisphere as follows:

$$L_o(x, \omega) = L_e(x, \omega) + \int_\Omega f_r(x, \omega_i \rightarrow \omega_o) L_i(x, \omega_i) cos\theta_i \, d\omega_i \tag{1}$$

The integration in which the incident radiance is commonly called the reflection equation which corresponds to the amount of reflected radiance $L_r(x, \omega_r)$ is received within a sphere Ω of incoming rays. L_e is the self emitted radiance of the surface, which is taken directly from the surface definition, and L_i is the incident radiance. f_r can be described as the material properties of the object that describes the probability that an incoming ray is randomly scattered into random directions, also called BRDF (Bidirectional Reflectance Distribution Function). In Whitted ray-tracing, the BRDF is calculated through Phong's reflection model (3).

2.5 Ray-Intersections

Each ray is sent out throughout the scene and intersects with a surface, which is performed different depending on the surface properties (BRDF) and the incoming angle.

2.5.1 Triangle Ray-Intersection

When a ray intersection has been computed using the Möller Trumbore-Algorithm a couple of objects are being saved. The triangle itself since the knowledge of its normal, color and material type can be used. The t-value mentioned earlier in Equation 3 is saved in order to calculate the world coordinates of the intersection point.

2.5.2 Möller Trumbore-Algorithm

For rays to be able to identify if it has hit an object built up by triangles, the Möller Trumbore-intersection algorithm can be applied. The algorithm is fast and uses barycentric coordinates. By using barycentric coordinates, any transformations applied to a triangle will not affect the intersection point coordinates (6).

Any point on a triangle $T(u, v)$ in barycentric coordinates can be computed as:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + uV_2 \tag{2}$$

u and v describes the point in barycentric coordinates, where $u \geq 0$, $v \geq 0$ and $u + v \leq 1$. If we define a ray $R(t)$ as:

$$R(t) = O + tD \tag{3}$$

where O is the origin, D is the normalized direction and each value t corresponds to a point $R(t)$ on the ray. The transformation that the algorithm performs will return a vector that has information of where the intersection point is on a triangle, as well as the distance from to that point from O (7).

2.5.3 Sphere Ray-Intersection

If a ray intersects a surface on a spherical object, there are two scenarios. Either it nudges the side of the sphere and only has one intersection point, which is very unlikely but till possible, or it goes through the sphere and gets two intersection points, one entry- and one exit point. To conclude which of the cases that occurs for the rays, a intersection test is done by combining the equation for a sphere and the the equation for a ray, Equation 3. To be more precise, an alternate version of the sphere equation is used:

$$\|\mathbf{x} - \mathbf{c}\|^2 = r^2 \tag{4}$$

to determine if a point with the coordinate vector $\mathbf{x}$ is located on a sphere with the radius r and the center point $\mathbf{c}$. To find a ray that is both on the ray and on the sphere, the ray x in Equation 4 is substituted with the right hand side of Equation 3. We then get the following equation:

$$(O + tD - \mathbf{c}) \cdot (O + tD - \mathbf{c}) = r^2 \tag{5}$$

which can further be transformed into a quadratic equation by substituting the constants:

$$t = -\frac{b}{2} \pm \sqrt{(\frac{b}{2})^2 - ac} \tag{6}$$

where $a = D \cdot D$, $b = D \cdot (O - \mathbf{c})$ and $c = (O - \mathbf{c})(O - \mathbf{c}) - r^2$. The quadratic equation 6 can be used to get the numerical values for t. The square root is in fact the term that determines whether the ray has missed, nudged or gone through the sphere depending on if the equation underneath is either negative, zero or positive respectively (8).

2.6 Hemispherical Coordinates

In order to calculate the angles of incoming and outgoing rays relative to the intersection surface, hemispherical coordinates comes in handy. Since a scene is mainly built up by multiples of triangles, intersection points often occurs on flat surfaces. This makes a hemispherical system ideal since it represents all the angles the intersection point can be viewed from correctly. By creating a coordinate system as seen in Figure 6, coordinates can be calculated using the angles $\theta \in [0, \pi/2]$ and $\varphi \in [0, 2\pi]$. These angles can be converted to local coordinates using $x = r * cos\theta * sin\varphi$, $y = r * sin\theta * sin\varphi$ and $z = r * cos\varphi$ where r is the radius of the hemisphere (4).

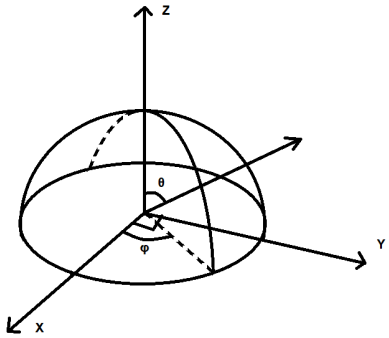


Figure 6: Hemispherical coordinate system

2.7 Shadow acne

When sending rays from surfaces and searching for potential intersections in the scene, a problem may occur where the outgoing ray intersects with the surface it is sent from. This self intersection is called Shadow acne because the surface cast a shadow on its self. To prevent this, the start point of the sent ray is moved a small amount out of the surface, adding a so called bias to the point (11).

2.8 Reflection rays

When rays hit object surfaces that are of a reflective type, perfect reflection is assumed. This means that the ray that goes out from the intersection

point is going to have the same angle θ going out, as the incident ray had going in. A visualization of this can be seen in Figure 7.

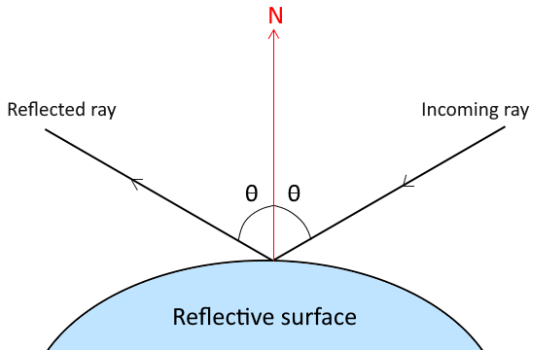


Figure 7: Reflected ray

2.9 Phong's Reflection Model

In order to calculate the amount of direct light a diffuse surface is exposed to, a local illumination model Phong is used. As seen in Figure 8 the amount of light contributing to the surface is calculated using different vectors from the intersection point. L(Light direction), N(surface normal), V(Ray direction), R(Light reflection) and H which is an easier computation of R as seen in Equation 7. The total illumination contribution from a single light source is calculated as seen in Equation 8, where k_d is the diffuse variable and k_s the specular.(12)

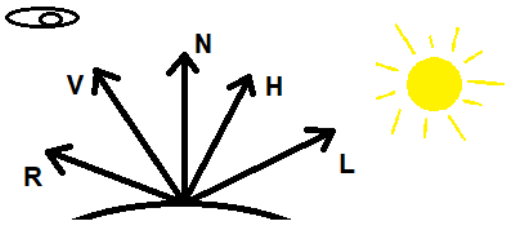


Figure 8: Phong Reflection model angles

$$H = L + \frac{V}{2} \tag{7}$$

$$I = k_d * (N \cdot L) + k_s * (N \cdot H) \tag{8}$$

2.10 Oren-Nayar reflectance model

To render a believable image, it is important that different surfaces can simulate real-life surface counterparts, or at least closely. For diffuse reflections this is done by the Oren-Nayar model. The model specializes on computing for rough surfaces by approximating the surface as small facets shaped as "V", each with different slopes. This replicates a rough surface more accurately than a more traditional reflectance model such as the Lambertian model. The model gives a reflection coefficient through the Equation 9. (10).

$$f_r(x, w_{in}, w_{out}) = \frac{\rho}{\pi}(A + B\, max(0, cos[\phi_{in} - \phi_{out}])sin\, \alpha\, sin\, \beta \tag{9}$$

A and B is computed as seen in Equation 10

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)},\ B = \frac{0.45\sigma^2}{\sigma^2 + 0.09} \tag{10}$$

The angles ϕ_{in}, θ_{in} and ϕ_{out}, θ_{out} describes the directions of the rays going in respectively out of the point on a diffuse surface. α and β is given by the maximum respectively minimum of the angles θ_{in} and θ_{out}.

The reflection coefficient is then applied to each color channel to compute the color with regards to the material type. The material type can be varied by changing the variables σ and ρ, which simply put corresponds to the roughness and how much of the light the surface absorbs. (10).

3 Results

The scene is built up by 6 walls with various color with a white roof and floor. The walls are placed in a hexagon pattern with 4 visible walls being yellow, cyan, dark blue and red from left to right followed by the two walls behind the camera which are green and pink. Only one light source was used which was placed in the roof. All surfaces except the light and right sphere are diffuse. The right sphere is of specular material and these images are rendered using Oren-Nayar and Phong's Model.

These images as seen in Figure 9 and Figure 10 are rendered in 800x800 pixels with 4 sub samples per pixel and 7*7 shadow rays. Rays are allowed to bounce 2 times and at each hemisphere 20 new rays are created. Important to note here is that these rendered images are from an early version where no consideration to the angle to the light was considered, as well as other bugs regarding the tetrahedron.

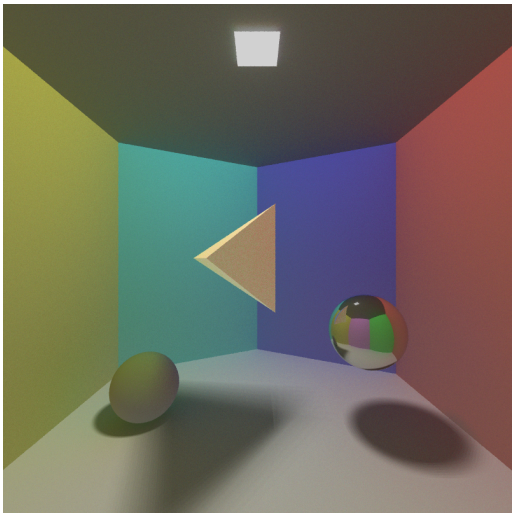


Figure 9: Early render using only Phong as direct illumination.

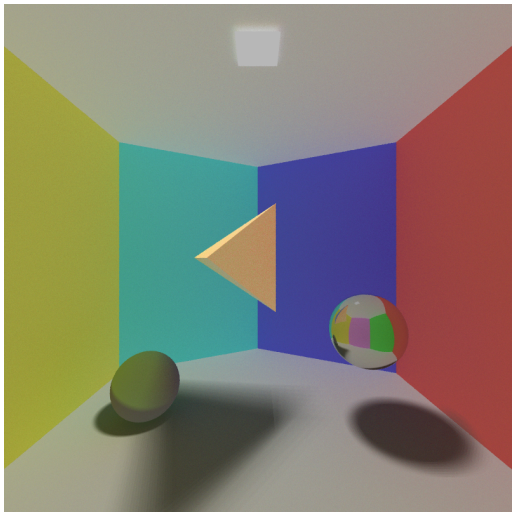


Figure 10: Early render with Phong disabled.

During some error searching an accidental but interesting outcome can be seen in Figure 11. The result is an image illustrating the color bleeding of a render.

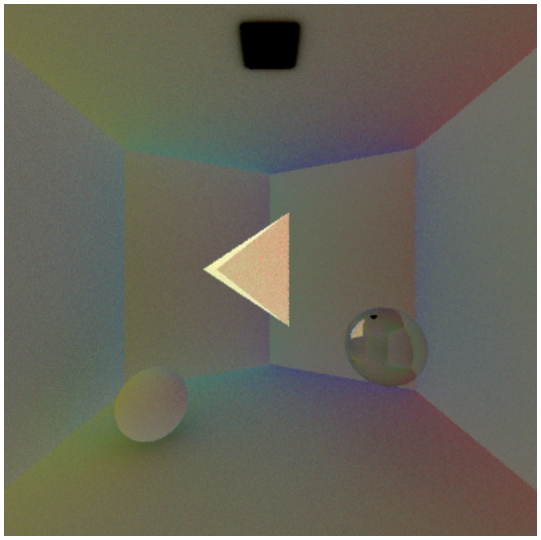


Figure 11: Render

During the early stages of implementing shadows, the phenomenon of shadow acne occurred as seen in Figure 12.

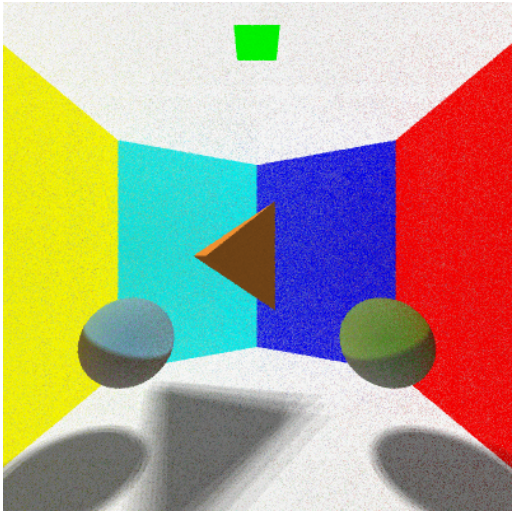


Figure 12: Render with shadow acne

These following images were rendered at the end of the project and were all rendered with 400x400 pixels with the standard variable values being 4

sub samples, 2 bounces and and 7x7 shadow rays. For each following test, one of these variables will vary while the others stay the same.

Figure 13, 14, 15 and 16 are rendered with varying amounts of shadow rays.

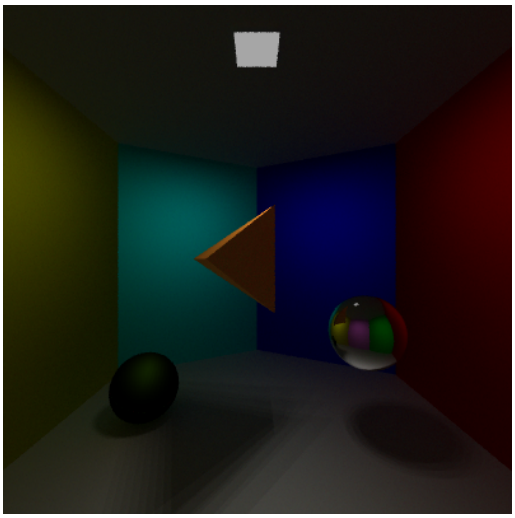


Figure 13: Rendered using 3x3 shadow rays

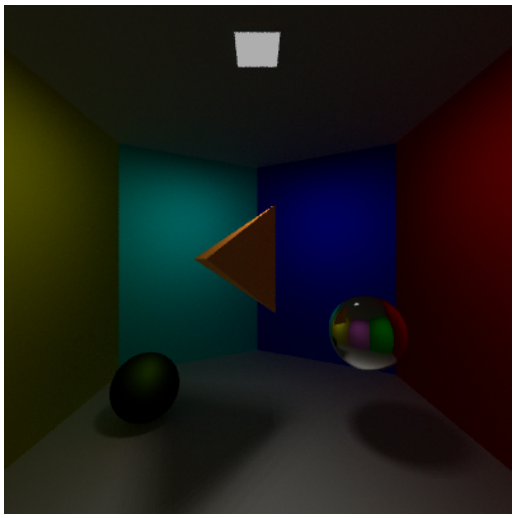


Figure 14: Rendered using 5x5 shadow rays

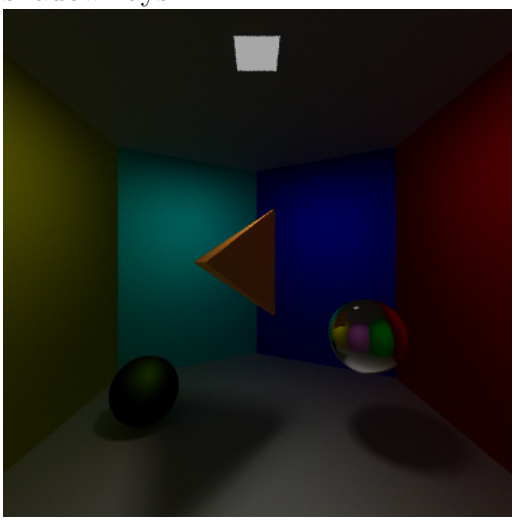


Figure 15: Rendered using 7x7 shadow rays

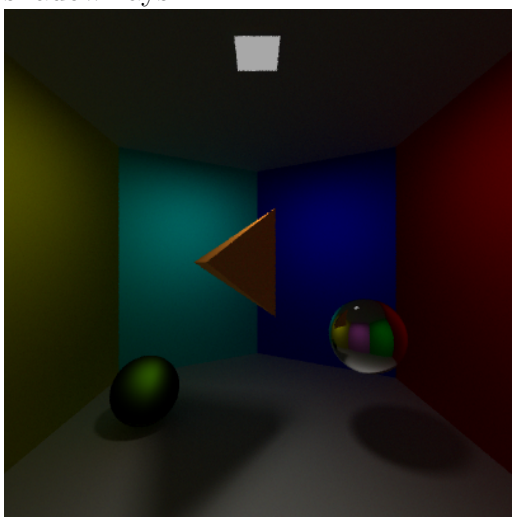


Figure 16: Rendered using 20x20 shadow rays

The rendered output when increasing the ray lifespan from 1 to 4 bounces can be seen in Figure 17 to 20 respectively. The time increase for each new bounce is at least quadratic.

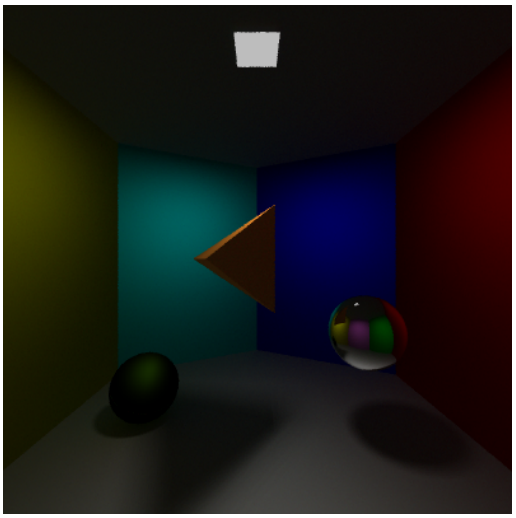


Figure 17: Ray bounces set to 1.

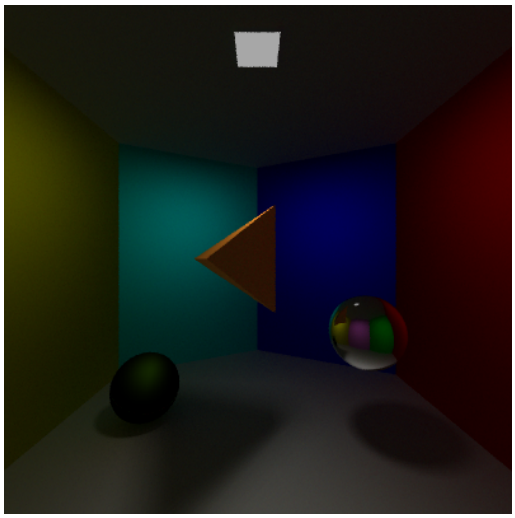


Figure 18: Ray bounces set to 2.

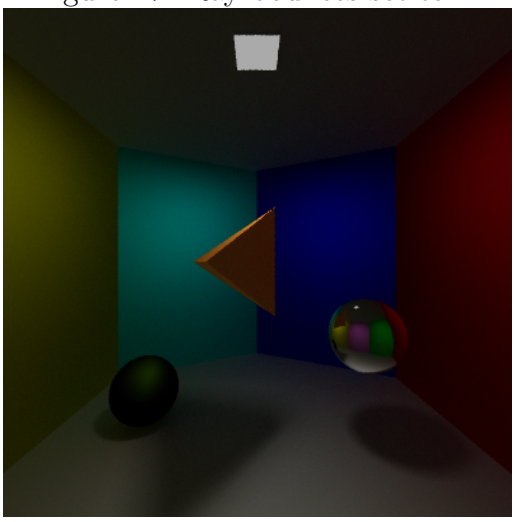


Figure 19: Ray bounces set to 3.

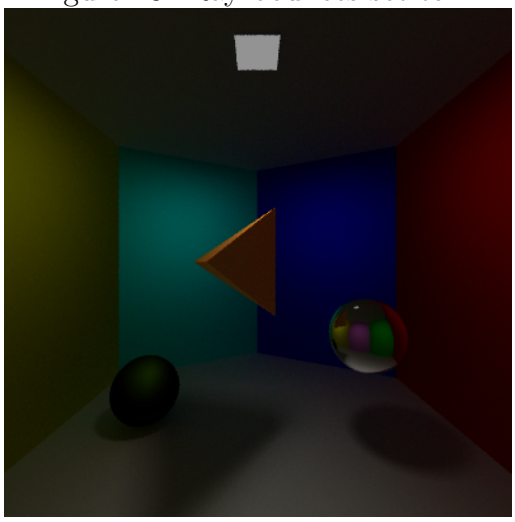


Figure 20: Ray bounces set to 4.

The images shown in Figure 21, 22, 23 and 24 are renders with varying amounts of rays sent out upon intersection with a diffuse surface, 0, 10, 20 and 30 respectively.

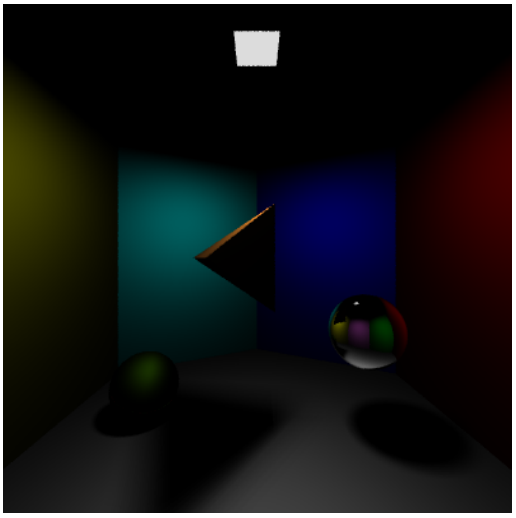


Figure 21: Rendered using 0 rays per hemisphere

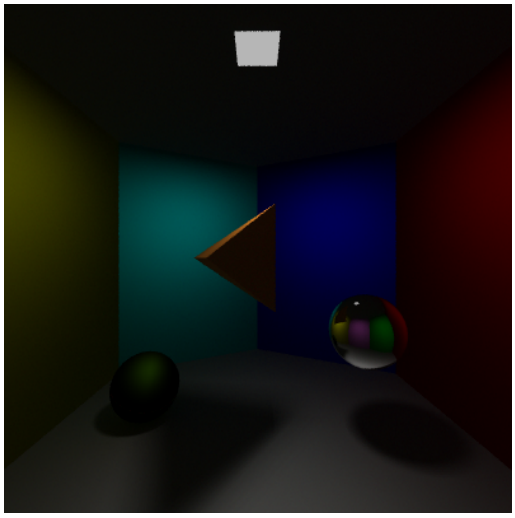


Figure 22: Rendered using 10 rays per hemisphere

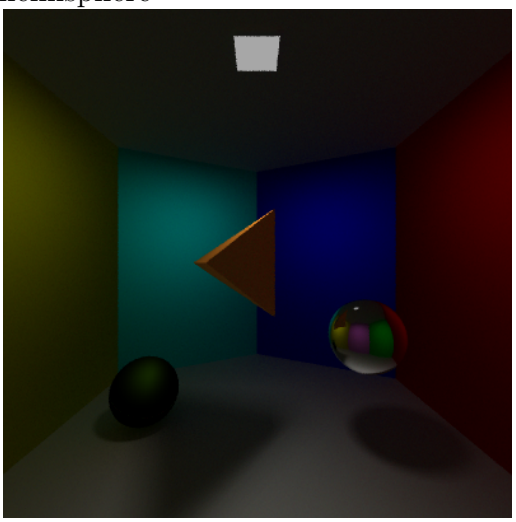


Figure 23: Rendered using 20 rays per hemisphere

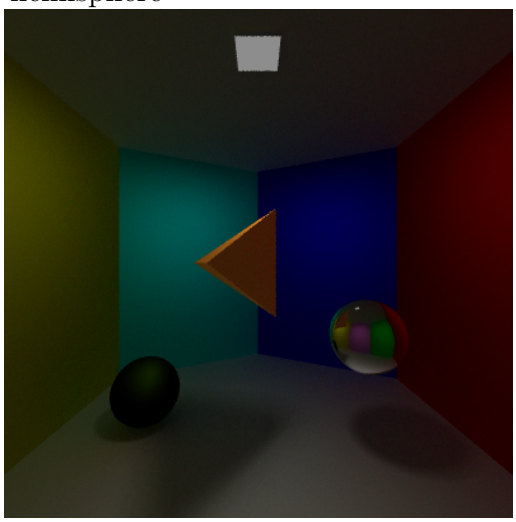


Figure 24: Rendered using 30 rays per hemisphere

4 Discussion

The rendered image seen in Figure 11 was a fortunate accident as it gave a clear visual of how color bleeding is effecting the renders. The result is as

expected, walls gets a heavier influence of color from walls connected to them and depending on angle. As seen in the image the blue wall is giving more color bleeding to the roof and to the floor due to the angle between them being 90 degrees. The neighboring walls are getting less color bleeding due to the angle being greater. The reason for this is that the lower the angle is, the larger probability that the random rays sent out from the hemisphere hits the wall which results in more color bleeding.

The images in Figure 17 to 20 reveals that in this implementation, increasing the lifespan of each ray does not yield a significant visual quality increase. However the render time does increase drastically for each incremental increase. When ray bounces are set to 2, the time is within a time frame of 2-4 minutes to render a 400x400 image on a high end PC. Increasing the bounce to 3 increases the time to be close to an hour to render, while 4 bounces reaches at the 24 hour mark.

An alternative to the uniform bounce amount approach would be to implement *Russian roulette* for the rays. This would randomly terminate the rays differently from one and another, meaning some rays could live for a long time, and some might decay in a matter of a few bounces. No implementation of this was however done which means if the result would been better or worse is just a speculation.

Regarding the images in Figure 13 to 16 where the shadows are generated using varying number of shadow rays. In the left-top image, where a total of 9 shadow samples are done, the shadows have clear artifacts that strays far from a realistic shadow. Going higher to the top-right image with 25 shadow samples, the result is better and in some ways acceptable. But with a closer look and especially if the resolution would be increased the lacking details in the shadow can become quite apparent. For the bottom-left image, with 49 shadow samples the result becomes decent with a smooth edge and look quite natural on resolutions up to at least 800x800. At the bottom-right a total of 400 shadow samples are used which is suited for higher resolution but might be unnecessary for some smaller resolutions, which is why in this project 7x7 was the standard within the resolutions 400x400 to 800x800.

In the images where the number of reflected rays from the hemisphere are varied, see Figure 21 to 24, the major difference is the amount of color that bleeds into other parts of the scene. However This also makes the scene appear somewhat brighter for every iteration due to the number of rays contributing with more color are larger.

During the implementation a large amount of issues emerged, as one

would expect from a coding implementation. Many of these problems can be seen in some of the images in the *Results* section. One of the first discovered issues was the self intersecting shadow rays that created unpleasant black noise in the rendered output, as seen in Figure 12. The issue stemmed from numerical precision errors that caused the shadow rays starting position to sometimes not be exactly on a surface. Rather the shadow ray origin could be slightly shifted right underneath the surface by the numerical error, making the shadow algorithm falsely recognize an object obscuring the surface where the shadow ray is shooting from. If the numerical error instead would happen to shift the origin to be above the surface, there would not be an issue and the results would look as expected which is why only some of the rendered output has black markings and other parts not. The fact that shifting the shadow ray origin slightly above the surface would work fine, the problem could be resolved if a numerical bias was introduced. This meant that the starting position could always be moved by the bias to be at least on or above the surface, which eliminated the shadow acne effect.

In later stages of the project, a reoccurring problem that haunted most of the following implementations was the color scaling. For each time a ray bounces, our implementation adds additional color contribution to the ray, which at the end of that rays lifespan would be a value above 1. This is not a problem on its own, as before the color values are written to a file they are downscaled based on the maximum color value. The problem instead arose when the lamp color was introduced, as the goal was to get a clear white lamp that has the brightest color in the scene. This was not compatible with the downscaling of values as if the lamp color was set to 1, the value would be extremely small once the scaling was done. This effect made it so it had the inverted effect where the lamp would instead be pitch black. Due to the structure of the code being non-ideal to change the way color scaling works, as it required everything else to be changed as well, therefore no great solution was found to fix the issue in a general way. This lead to a band-aid solution where a fixed value is scaled with the lamp color to manually set to the highest pixel value in the scene. This comes with many limitations as if other parameters in the scene are changed there might be a requirement to also adjust the fixed value to match the changes in order to keep the lamp the brightest.

There was an attempt to go around this issue to at least make the lamp look correct. This was done by simply assigning white color to the lamp pixels when looping through all the pixels after the pixel color had been

assigned but just before the color values was scaled. The result gave a white lamp as wanted but the problem of course still remained on the reflection, which was initially not accounted for. Because of this together with some reasoning this was not kept.

At project start the ideal goal was to introduce photons at a later stage to improve the rendering engine. The research behind photons itself were made, but due to many of the fundamental steps of the project requiring a large amount of troubleshooting and debugging, the expected timescale for the project was already getting out of hand. Therefore a decision was made to leave out photons and focus on the fundamental rendering parts to ensure a flawed but at the least functional rendering engine to be in place.

One thing that in hindsight could of been avoided in the project to save time, would of been to use any support library with help functions. This would of allowed implementation time, especially from early stages to not be spent on implementing vector and matrix operations that is already well defined in libraries that exists for free. Not only did this absorb time in terms of implementation but it also lead to more things to consider while debugging.

Far into the project, there were still some misunderstanding in some part of the theory. The gathered understanding was that the information which each ray included was solely the color channels. The idea of radiance was familiar but apparently far from fully clear. This led to the implementation lacking of the brightness parameter and as unnecessary amount of work on implementing operators and being far into the project with many difficulties, this was a task which was too hard to implement after hand.

An early implementation to improve the performance was to divide each sub pixel into their own thread. This worked well of course but was not fully thought through. What was not thought of was that an extended multi threading could have been used to further divide pixels into threads as well. This would have drastically improved the run time for sure. This was unfortunately too much work as late as this occurred and was therefore not applied. However, the code in it's full was not greatly adjusted to optimize performance sadly. There are still many improvements that could have been done and functions that could have been adjusted to lower the run time.

References

[1] Scratchapixel, *An Intuitive Introduction to Global Illumination and Path Tracing*, retrieved: 2020-10-29
https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing

[2] Pluralsight, *Understanding Global Illumination*, 2013-12-19, retrieved: 2020-10-29
https://www.pluralsight.com/blog/film-games/understanding-global-illumination

[3] Arvo. J, Dutre. P, *Monte Carlo Ray Tracing*, 2003-06-29, retrieved: 2020-12-30
https://www.researchgate.net/profile/Peter_Shirley/publication/340769537_Monte_Carlo_Ray_Tracing_Siggraph_2003_Course_44/links/5e9c6257299bf13079a75a3b/Monte-Carlo-Ray-Tracing-Siggraph-2003-Course-44.pdf

[4] Weisstein, Eric W., *Hemisphere*, 2020-10-23, retrieved: 2020-11-01
https://mathworld.wolfram.com/Hemisphere.html

[5] Scratchapixel, *Overview of the Ray-Tracing Rendering Technique*, retrieved: 2020-11-01
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview

[6] Scratchapixel, *Möller-Trumbore algorithm*, retrieved: 2020-11-01
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection

[7] Möller. T, Trumbore. B, *Fast Minimum Storage Ray/Triangle Intersection*, retrieved: 2020-11-02
https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

[8] Scratchapixel, *Ray-Sphere Intersection*, retrieved: 2020-11-02
https://www.scratchapixel.com/lessons/3d-basic-rendering/

minimal-ray-tracer-rendering-simple-shapes/
ray-sphere-intersection

[9] Beets. K, Barron. D, *Super-sampling Anti-aliasing Analyzed*, retrieved: 2020-11-02
http://www.x86-secret.com/articles/divers/v5-6000/datasheets/FSAA.pdf

[10] Oren. M, Nayar. S.K, *Generalization of Lambert's Reflectance Model*, retrieved: 2020-11-01 2020-11-02
https://www1.cs.columbia.edu/CAVE/publications/pdfs/Oren_SIGGRAPH94.pdf

[11] Scratchapixel, *Light and Shadows*, retrieved: 2020-11-02
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/ligth-and-shadows

[12] Tim J and Vertigo, *Phong Lighting and Specular Highlights*, 1997-02-16, retrieved: 2020-11-01
http://www.robots.ox.ac.uk/~att/index.html

[13] Jeremy Birn, *A look at caustics*, 2000, retrieved: 2021-10-27
http://www.3drender.com/light/caustics.html