Benjamin Widman, TE19E

Isak Livner Mäkitalo, TE19E

Gymnasiearbete 100p

HT-2021

3D Rendering Techniques

Implementation & Analysis

An in-depth study of ray tracing variants

Supervisor: Julia Brännström

Abstract

Ray tracing algorithms are important in the area of computer graphics as they can often offer great visual realism for various kinds of simulations, although creating accurate programmatic implementations of the algorithms can be of difficulty for the layman. This study aims to present a basis for a practical implementation of the ray tracing algorithms "distributed ray tracing" and "path tracing" as well as analyze and compare the implementations of these algorithms that were developed for this study through metrics such as visual quality, performance and algorithmic complexity. In this study two rendering engines were created, one for each of the two ray tracing techniques. The path tracing engine attempted to solve the rendering equation and featured BRDFs for plastic, dielectric and metal materials as well as a BTDF for light transmittance. The distributed ray tracing engine calculated both direct and indirect lighting, for soft shadows and color bleeding, and just like the path tracer it featured reflections and a plastic BRDF. Both of the engines were then tested for both visual analysis and measuring of the algorithmic time complexity they each had, in relation to samples per pixel or bounce. The results showed that the distributed ray tracer had a lot of noise on rough surfaces and it was also visible on the same scene rendered in roughly the same amount of time by the two engines that the distributed ray tracing algorithm suffered more from noise than the path tracing algorithm did.

Keywords: Computer graphics, 3D rendering, path tracing, distributed ray tracing, Monte Carlo integration, PDF, Importance sampling, BSDF, BRDF, BTDF

Table of Contents

1. Introduction

There is no doubt that ray tracing algorithms are an important class of algorithms in the area of computer graphics as they have the potential to offer visual results heavily based on the Newtonian physics of light, which results in highly realistic images. This is important for both scientific and industrial simulation as well as in entertainment media for creating immersive virtual environments. This may include movies and video games. Physically based ray tracing algorithms, such as path tracing, are sometimes also used in the architecture industry to visualize construction plans. On another note, path tracing has also, in recent years, become more relevant in the area of real-time rendering, like video games, as computer hardware has become powerful enough to render some modern games entirely using path tracing, for the first time in history. However, successfully implementing more advanced aspects of algorithms like path tracing in a computer program requires viewing a lot of material, often research papers, which may be difficult to find when studying the more advanced topics.

2. Problem Definition

This study aims to present an implementation of the 3D rendering techniques: "path tracing" and "distributed ray tracing" as well as compare important metrics such as algorithmic complexity, general performance, and visual realism. When comparing general performance, variables such as samples per pixel or bounce will be varied to obtain a bigger quantity of analyzable data with different viewpoints. Other factors specific to each engine will also be factored in to evaluate them. This report serves great purpose to those who want to attempt creating their own ray tracing engine as details about how one should go about implementing certain aspects are covered and one can also gain inspiration to what effects one might want to add on top of the base ray tracing algorithms. However, even if one does not desire to create such an engine from the ground up, greater general knowledge about ray tracing techniques can also be beneficial for any individual using a pre-existing engine as one can apply some of the knowledge of this report to one's artistic capabilities within 3D graphics. For society as a whole, it is beneficial for advancements in both computing capabilities and artistic usage of such techniques as it is generally a goal for many to achieve greater visual realism in computer renders.

2.1. Thesis

How could implementations of the two ray tracing algorithms "path tracing" and "distributed ray tracing" be created and how do the two algorithms compare to each other in terms of visual quality and performance?

3. Delimitations

Originally it was intended that this study would demonstrate the presented algorithms through both the use of ray casting and ray marching to sample the scenes, and analyze how varying between these two impact the performance of the algorithms. However, due to time restraints it was deemed unnecessary, as including it would also complicate the project, although including it would have been a great example and analysis of how it could be done and to what extent performance would increase. Furthermore, it is conventional to use some form of low-level 3D graphics API, like OpenGL or DirectX, for great improvement in performance with abstracted access to special hardware. However, with no past experience in any such API there was simply no time to learn one due to the complexity they can have. It is not relevant for the study of these ray tracing algorithms either so not using one does not impact the result of this report to the extent to what is relevant for the thesis. Originally, there was also a rasterizer engine that had been made to additionally compare with the two ray tracing techniques but as the project had gotten too big it was deemed unnecessary as it was also too unlike the other engines and was hardware accelerated, deeming performance comparisons between it and the other engines running on the processor completely redundant. This limitation only decreased the number of algorithms compared but it also allowed for more time to be spent on analysis of the two ray tracing algorithms instead.

4. Background and Terminology

It is a common misconception that "ray tracing" is a singular, novel technique from the last few years because of marketing done by the computer graphics giant Nvidia. However, not only is "ray tracing" an umbrella term referring to an entire class of algorithms, but many of the major, most commonly used algorithms in this class have existed for decades. According to the book "Image Synthesis" by Nadia Magnenat-Thalmann and Daniel Thalmann there was a group of researchers in 1971 in the math department at California Institute of Technology, in the US, that used a ray tracing algorithm to render 3D objects and their shadows on a computer monitor. The difference between the graphics commonly seen in pop media a few decades ago and what is available today was more so dependent on the hardware available back then, rather than the quality of the available algorithms. Due to the computational expense of the algorithms, ray tracing algorithms did not become common in popular media until the early nineties and 2000s when Pixar started using them to render parts of their 3D animated movies, according to the paper "The Path to Path Traced Movies" published by Foundations and Trends in Computer Graphics and Vision in 2016. The essence of ray tracing algorithms is that they simulate the scattering of light rays in a scene to achieve complex visual effects like shadows, reflections, refraction, and sometimes global illumination, just to name a few. Nowadays graphics cards and hardware performance in general have increased to the point that advanced ray tracing algorithms like "path tracing" are able to be used for real-time applications such as video games like Minecraft and Quake. However, hardware improvements are not the only reason that these algorithms can be real-time, as there have also been improvements to the algorithms allowing them to achieve better results with much less work. For example, in the research paper "An Efficient Denoising algorithm for Global Illumination" published by ACM SIGGRAPH / EuroGraphics High Performance Graphics in 2017, a path tracing and rasterization hybrid method was developed to quickly reduce noise in path traced images, thereby achieving high quality results with less work.

4.1. Ray tracing branches

As stated, "ray tracing" is an umbrella term referring to a large class of algorithms. This thesis will feature two of those algorithms, called "distributed ray tracing" and "path tracing". Firstly, distributed ray tracing consists of sending out distributions of new rays for every time a ray is scattered in a given scene. Based on information gathered by these rays the method approximates effects such as direct lighting, reflection and indirect lighting. Secondly, path tracing is a more realistic method of ray tracing as it aims to solve what is known as the "rendering equation". By doing this, it is able to achieve highly accurate images that contain global illumination, physically based reflection and physically based refraction. If the scene

rendered with the algorithm is created with real world data, the algorithm can often produce photorealistic images.

4.2. Technical terms

- **Samples per pixel**: In the context of path tracing, it is a term referring to how many rays are sent through each pixel in the image to sample the scene, where each ray may scatter a probabilistic number of times until it is terminated. In the path tracer, only one new sample ray is ever created every time a previous sample ray is scattered until the sample is terminated.

- **Samples per bounce**: In the context of distributed ray tracing the term is used to discuss performance and quality of output of the method. This term refers to how many new sample rays, referred to as distributed rays, are created anytime a sample ray is reflected.

- **Monte Carlo integration**: A method for numerically approximating any given definite integral. The method states that:

  $$\int_a^b f(x)dx = \lim_{n\to\infty} \frac{|b-a|}{n} \sum_{i=1}^{n} f(R_i)$$

  for any integrable function $f(x)$ where R_i is a random input chosen uniformly between a and b. The accuracy of the method is expected to improve as n becomes larger.

- **Probability density function (PDF)**: A function used to determine the probability of a sample being picked between two bounds a and b in a continuous distribution of samples. A given PDF $p(x)$ acting on a distribution going from s_0 to s_1 must satisfy the condition that:

  $$\int_{s_0}^{s_1} p(x)dx = 1$$

  In order to be a valid PDF. The probability of a sample being picked between the bounds a and b is calculated by integrating the function between the two bounds.

- **Bidirectional reflectance distribution function (BRDF)**: A function computing the intensity and color of light *reflecting* off a surface, given the direction ω_i of the incoming light hitting the surface and the direction ω_o of the outgoing light reflecting off the surface.

- **Bidirectional transmission distribution function (BTDF)**: A function computing the intensity and color of light *transmissioning* through a surface, given the direction ω_i of the incoming light hitting the surface and the direction ω_o of the outgoing light transmissioning through the surface.

- **Bidirectional Scattering Distribution Function (BSDF)**: The sum of the BRDF and the BTDF. This function computes the intensity and color of light *scattering* off a surface.

- **Albedo color**: Simply refers to the base color of an object, i.e. the color of the light it reflects when hit with incoming white light.

- **Big O notation**: Indicates the relationship between the computational complexity of a given algorithm and the size of some given variable of the algorithm, labeled as n. For an algorithm linearly increasing in computational complexity, one would write that its big O notation is $O(n)$. Similarly, a quadratic increase would be labeled as $O(n^2)$. When using big O notation, only the term with the highest order is included in the expression and constants in front of the terms are ignored.

5. Methodology

Two types of 3D rendering engines were created during this project: one for path tracing and one for distributed ray tracing. The chosen programming language for these was C++, mainly because of the high executing speed and freedom in memory control that it offers, and these applications require this. C++ is also undoubtedly a common language in the field of computer graphics for the same reasons. Despite the language featuring object oriented aspects it was decided not to make the programs object oriented as to not risk any performance loss, instead it was deemed sufficient to use structs and functional programming. To gain easy access to window creation, pixel drawing, and handling of key inputs, a 2D graphics API called "OLC Pixel Game Engine" was used. While the name appears to suggest it to be an entire game engine, the only thing it was used for was the aforementioned things. Everything else was done using code created for this project, during the project. Additionally, the development environment chosen for this project was Microsoft Visual Studio 2017 as there was prior experience in it by the authors of this study.

The two ray tracing algorithms path tracing and distributed ray tracing shared code for its core ray casting in this project. The core ray casting algorithm takes a ray defined with a direction and a position and finds its closest intersection with any object in the scene, if there is one. The ray tracing algorithms sample points in the scene by using this algorithm. The way the two methods sample points in the scene differ as the distributed ray tracer samples multiple directions per reflection, whereas the path tracer only samples one direction per ray scattering, but takes multiple samples for each pixel on the image.

5.1. Scene representation

The goal for both of these rendering engines is to sample information from a scene with various methods to then render it. However, firstly one must define how a scene is supposed to be represented with variables which describe physical properties, and for storing of such data, C++ structs (structures) were decided to be used.

```
struct Triangle
{
	Vec3D vertices[3];
	Material material;
	olc::Sprite* texture = nullptr;
	Vec2D textureVertices[3] = { ZERO_VEC2D, ZERO_VEC2D, ZERO_VEC2D };
	olc::Sprite* normalMap = nullptr;
};
```

This triangle structure contains an array of three 3D vectors representing each vertex of a given triangle. Then there is the material of the triangle represented by a material structure. Then there is the actual

texture stored in a Sprite class from the OLC Pixel Game Engine, which is simply an abstracted pixel array that can store an image. The texture vertices are three vertex coordinates that crop out which part of the texture is to be used for the triangle. Lastly, the structure can also store a potential normal map, providing the normal vector in tangent space of the triangle at a given point of its surface. The two engines also had structures for spheres and ground planes.

```
struct Sphere
{
	Vec3D coords;
	double radius;
	Material material;
	olc::Sprite* texture = nullptr;
	Vec2D textureCorner1 = ZERO_VEC2D;
	Vec2D textureCorner2 = ZERO_VEC2D;
	Quaternion rotQuaternion = IDENTITY_QUATERNION;
	olc::Sprite* normalMap = nullptr;
};

struct Ground
{
	double level;
	Material material;
	olc::Sprite* texture = nullptr;
	Vec2D textureCorner1 = ZERO_VEC2D;
	Vec2D textureCorner2 = ZERO_VEC2D;
	double textureScalar = 1;
	olc::Sprite* normalMap = nullptr;
};
```

Unlike the triangle, the sphere also contains a rotation quaternion to allow for rotated spheres. The ground structure also contains its level, in other words its y-coordinate, and a texture scalar which controls how big the texture stretches on the ground before it repeats again.

Finally, as seen in the previous structures, the engines provide a material structure for objects. It contains important physical properties that allow for realistic representations of surfaces.

```
struct Material
{
	Vec3D emittance; // { 0, 0, 0 } to { ∞, ∞, ∞ }
	Vec3D diffuseTint; // { 0, 0, 0 } to { 1, 1, 1 }
	double specularValue; // 0 to 1
	double roughness; // 0 to 1
	double refractionIndex; // 0 to ∞
	Vec3D attenuation; // { 0, 0, 0 } to { ∞, ∞, ∞ }
	double extinctionCoefficient; // Only relevant for metals
	MaterialType type;
};
```

The emittance is the light emitted while the diffuse tint is the albedo color of the object. The specular value impacts how much specular reflection a surface provides, although it does not alone define this. The roughness defines how rough the material is and therefore how wide the distribution of specular rays reflected from the surface is. The refraction index determines how much light is bent while traveling through the medium and also impacts how much a surface reflects specular rays. The attenuation coefficient defines how quickly light is reduced when traveling through a medium, thereby determining the opacity of an object. The extinction coefficient is a property that is only applied to metals and affects the reflectivity of the material. Lastly, there is an enumeration describing which type a material is of. In it there are the options of the types: dielectric, metal, and plastic. This enumeration tells the path tracing algorithm what BRDF to apply to a given material. The distributed ray tracer always uses the plastic BRDF regardless of what option is specified in the enumeration.

5.2. Path Tracing

When creating the path tracing program, a standard variant of the algorithm called "forward path tracing" was used. Essentially, path tracing is a rendering technique that attempts to solve the limitations of distributed ray tracing. The basic algorithm of path tracing can in essence be summarised by an equation known as "the rendering equation" which takes the following form:

$$L_o(x, \omega_o, m) = L_e(x, \omega_o, m) + \int_\Omega f_s(x, \omega_i, \omega_o, m) L_i(x, \omega_i) \left| \omega_i \cdot n \right| d\omega_i \quad [1]$$

Where:

- x is a position in space
- ω_i, ω_o are incoming and outgoing directions of light
- m is a material containing properties that determine the look of an object
- n is a normal vector

- $L_o(x, \omega_o, m)$ is the outgoing light scattering from the point x in the direction ω_o
- $L_e(x, \omega_o, m)$ is the light emitted from the point x in the direction ω_o
- $\int_\Omega f_s(x, \omega_i, \omega_o, m) L_i(x, \omega_i) |\omega_i \cdot n| d\omega_i$ is the light scattering from the point x in the direction ω_o
- $\int_\Omega$ is an integral over the solid angle Ω around x containing all incoming directions ω_i
- $f_s(x, \omega_i, \omega_o, m)$ is known as the "bidirectional scattering distribution function" and computes what proportion of the incoming light from the direction ω_i is scattered from the point x in the direction ω_o
- $L_i(x, \omega_i)$ is the incoming light from the direction ω_i to the point x
- $|\omega_i \cdot n|$ is a multiplicative factor attenuating the scattered light in accordance to *Lambert's cosine law*

Simply put, the equation gives the intensity and color of the light scattering from a given point x in a given direction ω_o in the virtual scene. By solving this equation for all sets of inputs $\{x, \omega_c, m\}$ where x is a point visible by the virtual camera, ω_c is the direction from point x to the camera and m is the material at x, a realistic image representation of the scene can be obtained.

It is important to notice that equation 1 does not have a closed form solution as the integral part of the equation is too complicated to be solved analytically for any generic scene. Due to this, numerical methods have to be used to approximate solutions of the equation. Solving the integral part of the rendering equation is the key problem statement of the path tracing algorithm and the quality of the methods used are in part what determines the visual realism of the output image and the computing work required to obtain it.

5.2.1. Computing the integral

To compute the integral the Monte Carlo integration method was used. To use the method the integrand must be sampleable for any given set of inputs. As seen from equation 1 the integrand which needs to be sampled takes the form:

$$f_s(x, \omega_i, \omega_o, m) L_i(x, \omega_i) |\omega_i \cdot n|$$

In this expression there are three important factors that need to be evaluated, namely:

1. $f_s(x,\ \omega_i, \omega_o, m)$
2. $L_i(x, \omega_i)$
3. $\left|\omega_i \cdot n\right|$

The third factor is simple as it is just the absolute value of a dot product between known vectors. The first and second factors are more complex. The second factor is the light hitting the point x from the direction ω_i. This light is calculated using the rendering equation and as this factor is itself part of the rendering equation it means that the rendering equation is infinitely recursive. This complicates the sampling process. Initially this problem was solved by only allowing a finite number of recursive steps. Finally, the first factor, the Bidirectional Scattering Distribution Function (BSDF) computes what proportion of the incoming light from the direction ω_i is scattered from the point x in the direction ω_o. This function depends on the material m of the point x and is key to the realism of the output image as it determines how varying materials look. The BSDF can be broken up as the sum of two parts, a Bidirectional Reflectance Distribution Function (BRDF) and a Bidirectional Transmittance Distribution Function (BTDF). The BRDF computes the proportions and wavelengths of light that are reflected from the material m whereas the BTDF computes the proportions and wavelengths of light that are refracted through the material. This linear property of the BSDF means that the integral part of the rendering equation can be rewritten as:

$$\int_\Omega (f_r(x,\ \omega_i, \omega_o, m)\ + f_t(x,\ \omega_i, \omega_o, m))L_i(x, \omega_i)\left|\omega_i \cdot n\right|d\omega_i =$$

$$\int_\Omega (f_r(x,\ \omega_i, \omega_o, m)L_i(x, \omega_i)\left|\omega_i \cdot n\right|d\omega_i + \int_\Omega (f_t(x,\ \omega_i, \omega_o, m)L_i(x, \omega_i)\left|\omega_i \cdot n\right|d\omega_i \quad [2]$$

Where $f_r(x,\ \omega_i, \omega_o, m)$ is the BRDF and $f_t(x,\ \omega_i, \omega_o, m)$ is the BTDF. This rewriting of the equation was useful because it meant the BRDF and BTDF could be integrated separately, simplifying the integration.

For this project three different BRDFs were used. When sampling the first integrand of equation 2 the path tracing program chooses what BRDF to sample with depending on whether the material at the point x is dielectric, metallic or plastic. For dielectric materials the BRDF consists of a specular and diffuse component summed together. The diffuse component is quite simple and as such can be momentarily

swept aside. The specular component of the dielectric BRDF was taken from the paper "Microfacet Models for Refraction through Rough Surfaces" by Bruce Walter et.al. The form of this BRDF is:

$$\frac{F(\omega_i, h_r, \eta_1, \eta_2)G(\omega_i, \omega_o, n, h_r, \alpha)D(\omega_i, n, h_r, \alpha)}{4\left|\omega_i \cdot n\right|\left|\omega_o \cdot n\right|} \quad [3]$$

Where $F(\omega_i, h_r, \eta_1, \eta_2)$ is the "Fresnel term", $G(\omega_i, \omega_o, n, h_r)$ is the "Geometry term" and $D(\omega_i, n, h_r, \alpha)$ is the "Distribution term". In these functions h_r is called the "microfacet normal" at the point x and was calculated as the half vector of ω_i and ω_o. In this expression the fresnel term outputs what proportion of light hitting the point x is reflected instead of transmitted and was calculated as:

$$\frac{1}{2} \cdot \frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right) \quad [4]$$

Where:

- $g = \sqrt{\frac{\eta_2^2}{\eta_1^2} - 1 + c^2}$
- $c = \left|\omega_i \cdot h_r\right|$
- η_1 is the refractive index of air and η_2 is the refractive index of the material m or vice versa depending on whether ω_i is pointing towards or out of the object at the point x

Furthermore, the geometry term attenuates the scattered light based on statistics of how microfacets block some light from scattering. This term plays an important role in conserving the energy of the light before and after scattering and is hence important to the physical plausibility of the BRDF. The geometry term is expanded as such:

$$G(\omega_i, \omega_o, n, h_r, \alpha) = G_1(\omega_i, n, h_r, \alpha) \cdot G_1(\omega_o, n, h_r, \alpha) \quad [5]$$

Where:

- $G_1(v, n, h_r, \alpha) = \chi\left(\frac{v \cdot h_r}{v \cdot n}\right)\frac{2}{1+\sqrt{1+\alpha^2 tan^2\theta}}$
- α is the roughness of the material m
- θ is the angle between v and n
- χ is a function defined such that $\chi(a)$ equals 1 if $a > 0$ and 0 if $a \leq 0$

Finally, the distribution term presents the distribution of reflected light for all directions ω_o could reflect in relation to ω_i and was computed as:

$$\frac{\alpha^2 \chi(h_r \cdot n)}{\pi cos^4\theta(\alpha^2 + tan^2\theta)^2} \quad [6]$$

Where θ is the angle between ω_i and n. The diffuse component of the dielectric BRDF is simply:

$$\chi(h_r, n)(1 - F(\omega_i, h_r, \eta_1, \eta_2))^2 \frac{c}{\pi} \quad [7]$$

Where c is the albedo color of the material m. This term accounts for the light that is diffusely reflected off the material. For metallic materials the BRDF looks like:

$$\frac{c \cdot F_c(\omega_i, n, \eta_1, \eta_2) G(\omega_i, \omega_o, n, h_r, \alpha) D(\omega_i, n, h_r, \alpha)}{4|\omega_i \cdot n||\omega_o \cdot n|} \quad [8]$$

Where $F_c(\omega_i, h_r, \eta_1, \eta_2)$ is the fresnel term for conductive materials and can, according to a blog post called "Memo on Fresnel equations" by Sébastien Lagarde (2013), be calculated as:

$$\frac{(R_s + R_p)}{2} \quad [9]$$

Where in turn:

- $R_s = \frac{a^2 + b^2 - 2a \cdot cos\varphi + cos^2\varphi}{a^2 + b^2 + 2a \cdot cos\varphi + cos^2\varphi}$
- $R_p = R_s \frac{a^2 + b^2 - 2a \cdot sin\varphi \cdot tan\varphi + sin^2\varphi \cdot tan^2\varphi}{a^2 + b^2 + 2a \cdot sin\varphi \cdot tan\varphi + sin^2\varphi \cdot tan^2\varphi}$
- $a^2 = \frac{1}{2\eta_1^2}(\sqrt{(\eta_2^2 - k^2 - \eta_1^2 sin^2\varphi)^2 + 4\eta_2^2 k^2} + \eta_1^2 - k^2 - \eta_1^2 sin^2\varphi)$
- $b^2 = \frac{1}{2\eta_1^2}(\sqrt{(\eta_2^2 - k^2 - \eta_1^2 sin^2\varphi)^2 + 4\eta_2^2 k^2} - \eta_1^2 + k^2 + \eta_1^2 sin^2\varphi)$
- φ is the angle between ω_i and n
- k is the absorption coefficient of the metallic material

Lastly the plastic BRDF being the simplest of the BRDFs, was implemented as the albedo color c multiplied by the dielectric BRDF, which when written out becomes:

$$\frac{c \cdot F(\omega_i, h_r, \eta_1, \eta_2) G(\omega_i, \omega_o, n, h_r, \alpha) D(\omega_i, n, h_r, \alpha)}{4\left|\omega_i \cdot n\right|\left|\omega_o \cdot n\right|} \quad [10]$$

To integrate the right integrand of equation 2 the BTDF was sampled instead of one of the BRDFs. The form of the BTDF was:

$$\frac{\left|\omega_i \cdot h_r\right|\left|\omega_o \cdot h_r\right|}{\left|\omega_i \cdot n\right|\left|\omega_o \cdot n\right|} \cdot \frac{\eta_2^2 \, (1 - F(\omega_i, h_r)) F(\omega_i, h_r, \eta_1, \eta_2) G(\omega_i, \omega_o, n, h_r, \alpha) D(\omega_i, n, h_r, \alpha)}{(\eta_1(\omega_i \cdot h_r) + \eta_2(\omega_o \cdot h_r))^2} \quad [11]$$

5.2.2. Improved sampling

While the integration method presented works, it was found that it did not converge at a sufficient rate when sampling uniformly within the integration bounds. To accelerate the convergence, PDFs were used to bias the sampling towards the outputs of the integrand that correspond to the most light. When using a non-uniform PDF to bias the samples, the result of the Monte Carlo integration method does not converge to the same output as when using a uniform PDF. To fix this, the Monte Carlo method is modified as such:

$$\int_a^b f(x)dx = \lim_{n \to \infty} \frac{|b-a|}{n} \sum_{i=1}^{n} \frac{f(R_i)}{p(R_i)} \quad [12]$$

Where $p(x)$ is the PDF and R_i is a weighted random input chosen in accordance to the distribution given by $p(x)$. This method is known as "Importance sampling" and is more efficient than sampling uniformly if within the integration interval, $p(x)$ closely matches $Cf(x)$ where C is a normalization constant. To more effectively sample the left integrand seen in equation 2 two distinct PDFs were used. The first PDF was fitted to the non-diffuse BRDFs since they dominate the integrand. To fit the PDF to the non-diffuse BRDFs it was fitted to the distribution term of the BRDFs because it dominates the functions for smaller values of α. To effectively fit the PDF to the distribution term a microsurface normal m is generated such that:

$$\theta_m = arctan\left(\frac{\alpha\sqrt{\xi_1}}{\sqrt{1-\xi_1}}\right) \quad [13]$$

$$\phi_m = 2\pi\xi_2 \quad [14]$$

Where

- θ_m is the azimuthal angle of the microsurface normal

- ϕ_m is the zenithal angle of the microsurface normal
- ξ_1 and ξ_2 are two numbers uniformly sampled within the interval 0 to 1

Using this microsurface normal a sampling vector can be generated for the non-diffuse BRDFs and the BTDF. For the BRDFs, the sampling vector is generated with the equation:

$$\omega_o = 2|\omega_i \cdot m|m - \omega_i \quad [15]$$

and for the BTDF the sampling vector is generated as:

$$\omega_o = (\eta c - sign(\omega_i \cdot n)\sqrt{1 + \eta(c^2 - 1)})m - \eta\omega_i \quad [16]$$

Where

- $c = \omega_i \cdot m$
- $\eta = \frac{\eta_1}{\eta_2}$

When dividing the non-diffuse BRDFs and the BTDF by their respective PDFs many terms are canceled out leaving the final equation to be:

$$\frac{|\omega_i \cdot m| G(\omega_i, \omega_o, m, \alpha)}{|\omega_i \cdot n||m \cdot n|} \quad [17]$$

For the BTDF. For the dielectric, metallic and plastic BRDFs the result is similar, except the dielectric BRDF is multiplied by the dielectric fresnel term, the metallic BRDF is multiplied by the conductor fresnel term and the plastic BRDF is multiplied by both the dielectric fresnel term and the albedo color. Important to note is that the cosine term seen in the integrand of the rendering equation also cancels out when using this improved sampling. When sampling the scene with the diffuse BRDF the PDF used was fitted to the cosine term of the integrand, canceling it out from the equation. The sampling vector for the diffuse BRDF was generated as:

$$\omega_o = \left\{\sqrt{\xi_1} \cdot cos(\phi), \sqrt{1 - \xi_1}, \sqrt{\xi_1} \cdot sin(\phi)\right\} \text{ with } \phi = 2\pi\xi_2 \quad [18]$$

Where ξ_1 and ξ_2 are random numbers chosen uniformly between 0 and 1. Lastly, it was highlighted in section 5.2.1 that the rendering equation has to be recursed infinitely for true sampling. Initially, a hard limit for the number of recursive steps was put in place as an attempt to solve this problem. This solution

does however not guarantee the convergence of the integration method as the number of samples goes to infinity. To avoid the problem of infinite recursion while still guaranteeing the convergence of the integration, a method known as "russian roulette" was used to decide when to stop the recursion. This method keeps track of the weight of a given sample and then probabilistically determines at every recursion step whether to stop the sample from recursing further based on its weight, with higher weights corresponding to higher probabilities of continued recursion. Samples that survive the russian roulette process have their weights boosted by the reciprocal of the probability that they would not survive. This is done to account for the lost energy of samples that are stopped. How specifically the weight of a given sample is calculated is not of utmost importance, what matters most is that samples with a higher contribution to the integration receive higher weights.

5.3. Distributed ray tracing

The distributed ray tracing algorithm was implemented in such a way that the shading pipeline would be recursive, in this case meaning that the shading function would call itself for every ray bounce that occurred. However, for the function to always have a finite compute time, a selectable limit to the number of bounces allowed was implemented, with the function always receiving the number of previous bounces as a parameter.

Firstly, soft shadows are approximated by sending out new rays towards all light sources in the scene, in other words all spheres that emit light. For every light source a distribution of rays is cast from a given point x on some surface to random points on the surface of the light source. The approximate contribution of direct lighting reflecting from x can then be calculated as:

$$L_d = c \sum_{i=1}^{n_L} r_i \sum_{j=1}^{n_R} \frac{L_i}{1-cos(\theta)} \quad [19]$$

Where c is the albedo color of x, n_L is the number of light sources, r_i is the proportion of rays that were not blocked from hitting a given light source, n_R is the number of distributed rays, L_i is the emittance of a given light source and θ is the maximum angle between x and the given light source.

Secondly, the indirect light reaching x is approximated by spawning a new distribution of rays. Each ray is given a direction generated according to equation 15 seen in section 5.2.2. From there on the indirect light reflecting from x can be expressed as:

$$L_{id} = \sum_{i=1}^{n_R} L_i \frac{c \cdot |\omega_i \cdot m| F(\omega_i, m) G(\omega_i, \omega_o, m, \alpha)}{|\omega_i \cdot n||m \cdot n|} \quad [20]$$

Where L_i is the incoming light from the direction of a given distributed ray. The right side of this equation features the plastic BRDF mentioned in section 5.2.2. Important to note is that this method is infinitely recursive like the integration method presented for the path tracer. After each distributed ray has hit an object in the scene, a new distribution of rays is created. Unlike the path tracer, the distributed ray tracer program simply resolves this issue by only allowing a finite number of recursions. Lastly, the final light reflecting from x is then calculated as:

$$L_f = L_d + L_{id} + L_e \quad [21]$$

Where L_e is the light emitted from x.

5.4. Analysis methodology

In order to compare the different rendering techniques, rendering tests had to be done to both measure performance and also compare visual realism. Hence, several different renders were made, all with changing variables, such as samples per pixel or bounce, to then both compare the visual results and render times. All rendered test images consistently used a resolution of 900x720 pixels and for the distributed ray tracer a maximum of two ray bounces was allowed. Firstly, a pink reflective sphere in a small room was rendered with both the path tracer and the distributed ray tracer to compare the visual differences. However, in the path tracer a dielectric material was used and a plastic material in the distributed ray tracer respectively as they looked more alike in the two engines. Secondly, there were render time tests made for both engines. These tests were done with three different renders for each engine, with each render increasing the number of samples per pixel or samples per bounce respectively to measure how render time is increased. For the path tracer a room with three spheres, with one of them having low light attenuation and refractive properties, was rendered to show off its refractive capabilities. Lastly, for the distributed ray tracer's render time tests a famous type of test scene consisting of two white cuboids called the Cornell box, was rendered with both of the rendering techniques. The render time measures for both rendering engines were made on an Intel Core i5 9600K running at 5GHz and utilizing 4 threads. This processor actually has 6 threads but by not utilizing every single one the risk of other applications stalling the rendering engine is decreased, hence increasing the credibility of the render time measurements and comparisons.

6. Results

6.1. Performance and visual tests

Here are the performance and visual tests that were made for both the path tracer and the distributed ray tracer.

6.1.1. Path tracing

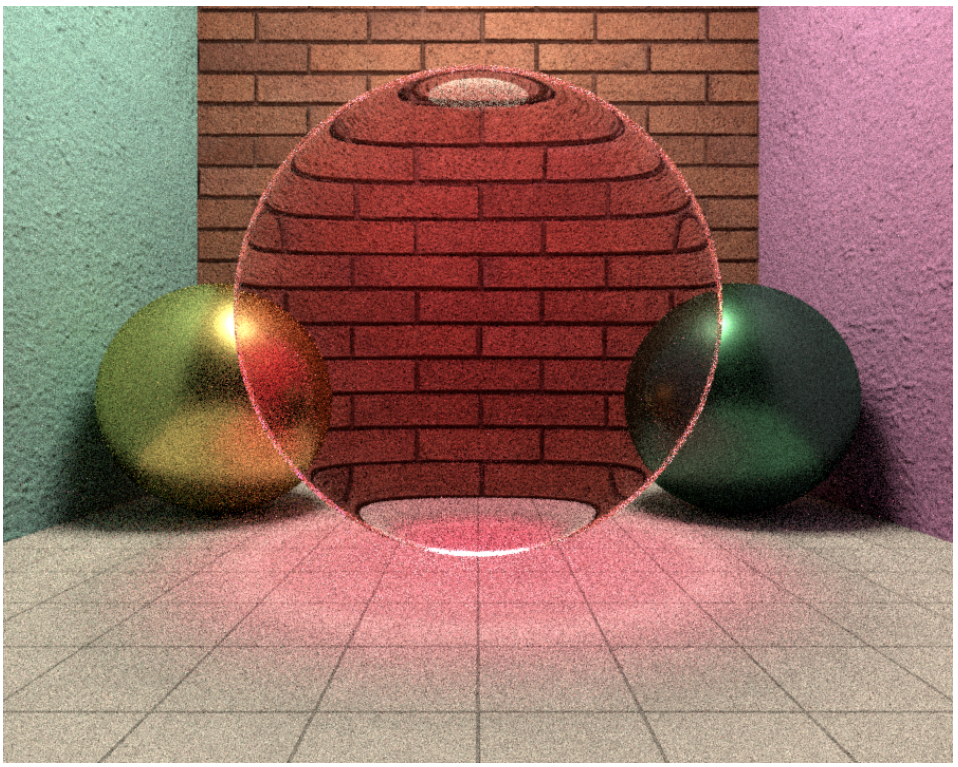

Image 1: Path tracer, 1 000 samples/pixel, render time: 426 seconds (7 min 6 s)

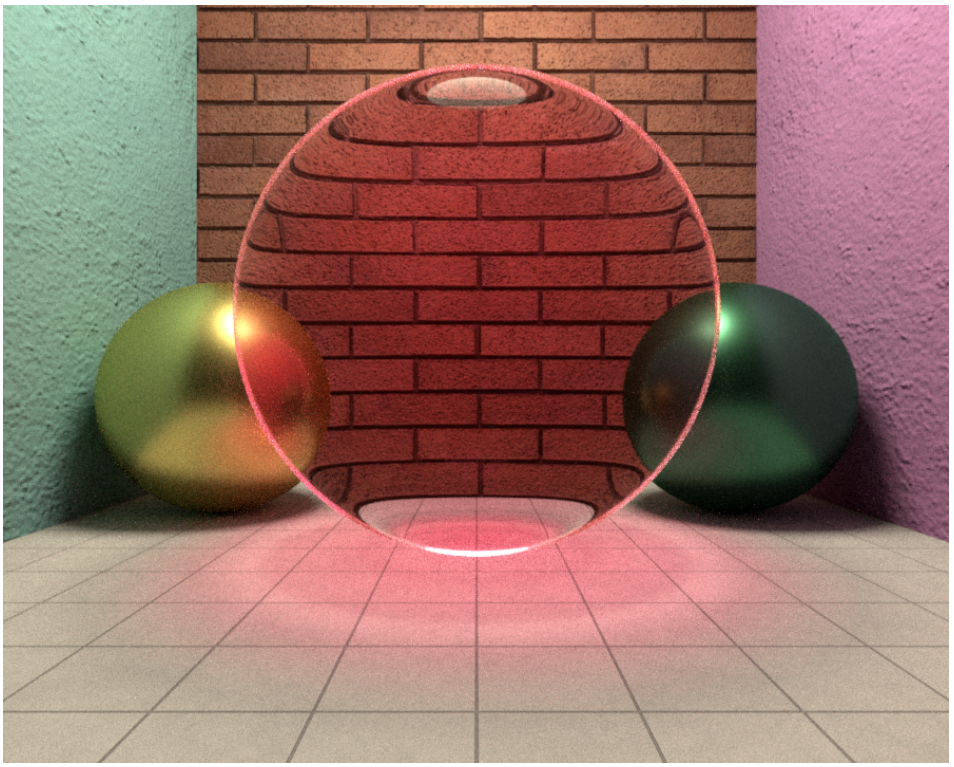

Image 2: Path tracer, 10 000 samples/pixel, render time: 4191 seconds (1 h 10 min)

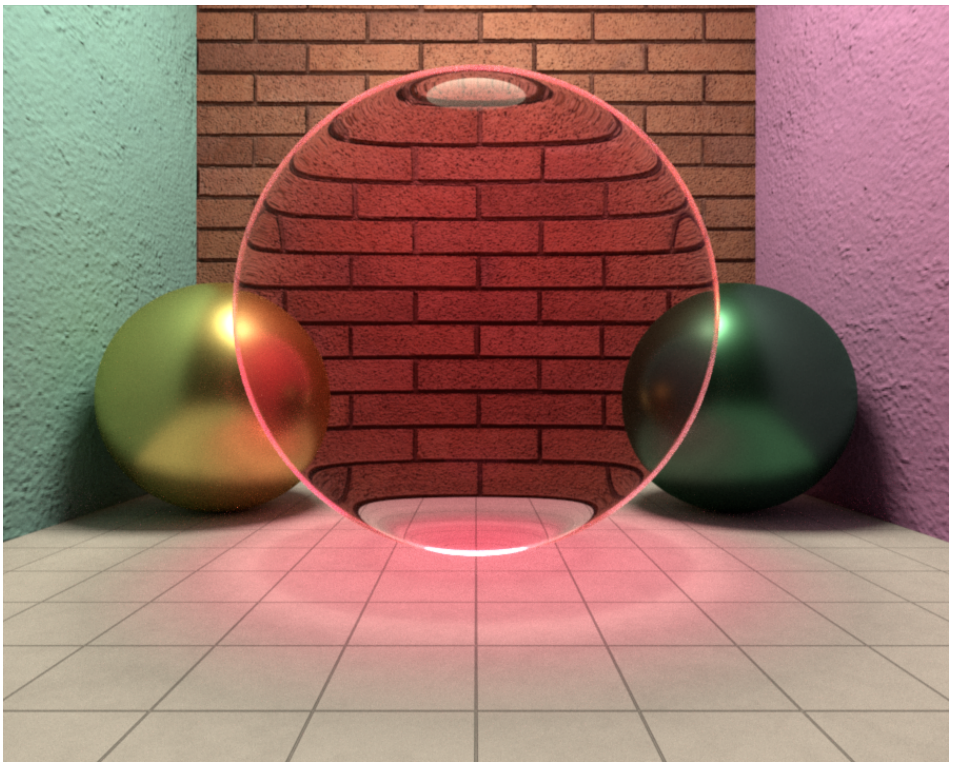

Image 3: Path tracer, 50 000 samples/pixel, render time: 21467 seconds (5 h 58 min)

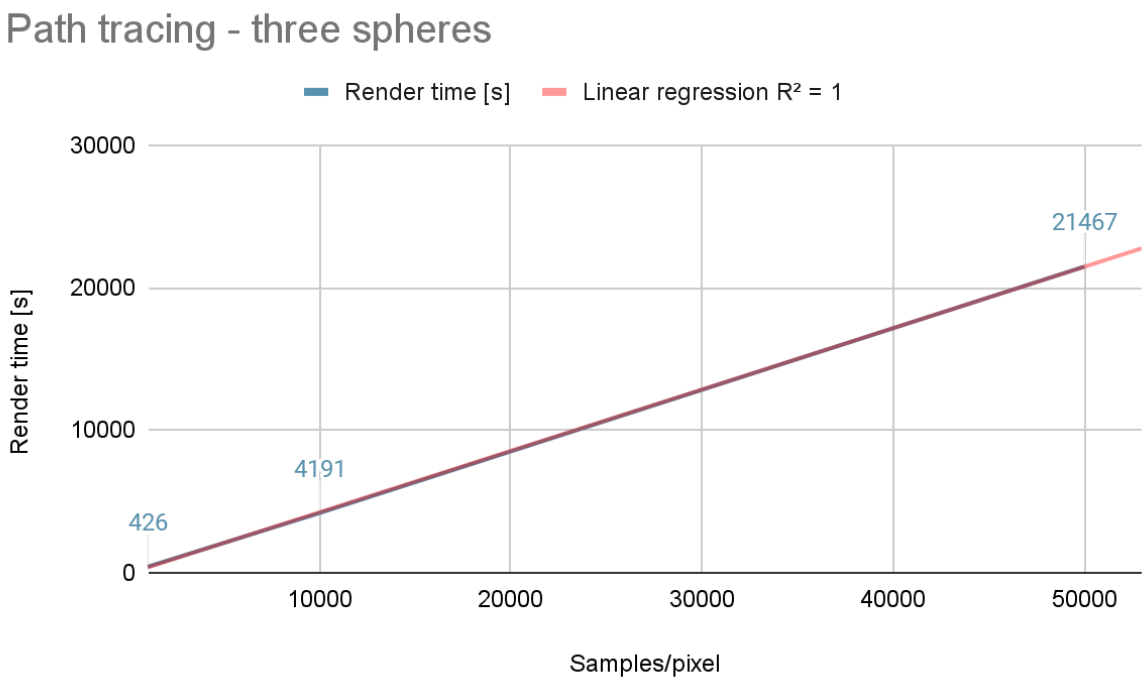

Diagram 1: plotting and linear regression of render time in relation to samples/pixel (path tracer)

6.1.2. Distributed ray tracing

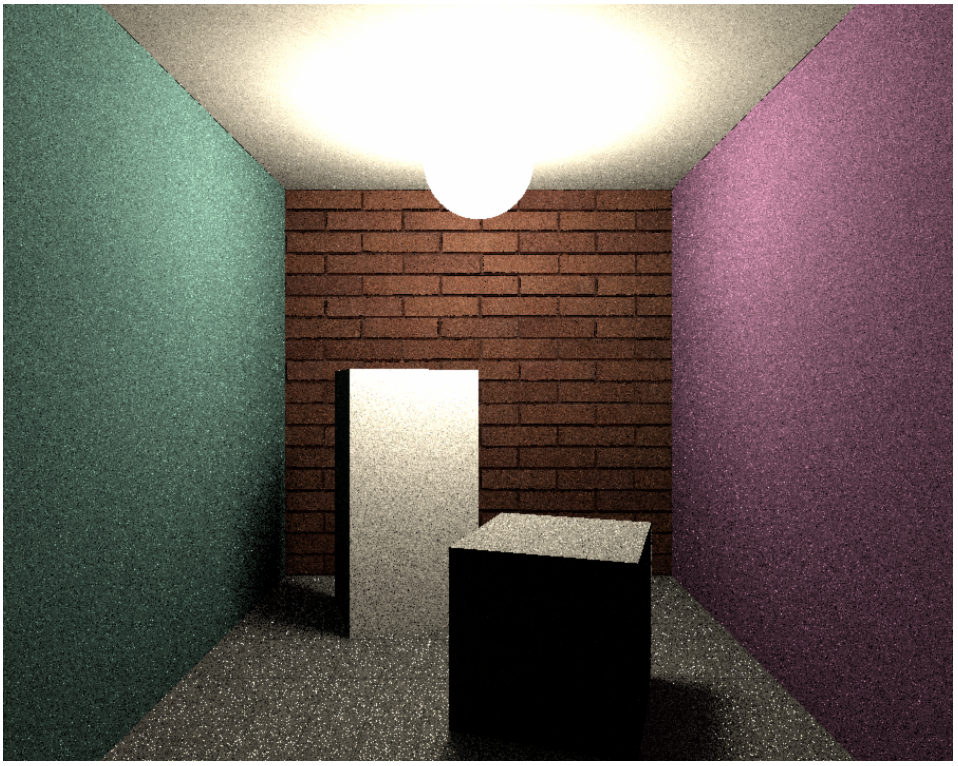

Image 4: Distributed ray tracer, 15 samples/bounce, render time: 562 seconds (9 min 22 s)

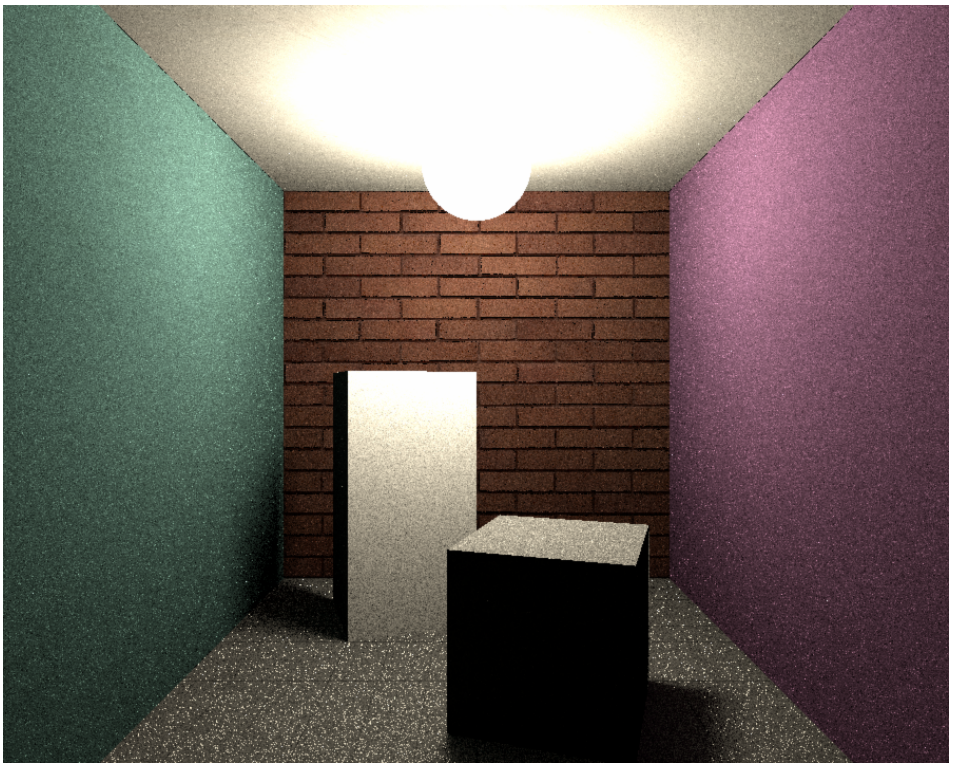

Image 5: Distributed ray tracer, 30 samples/bounce, render time: 3993 seconds (1h 14 min 22 s)

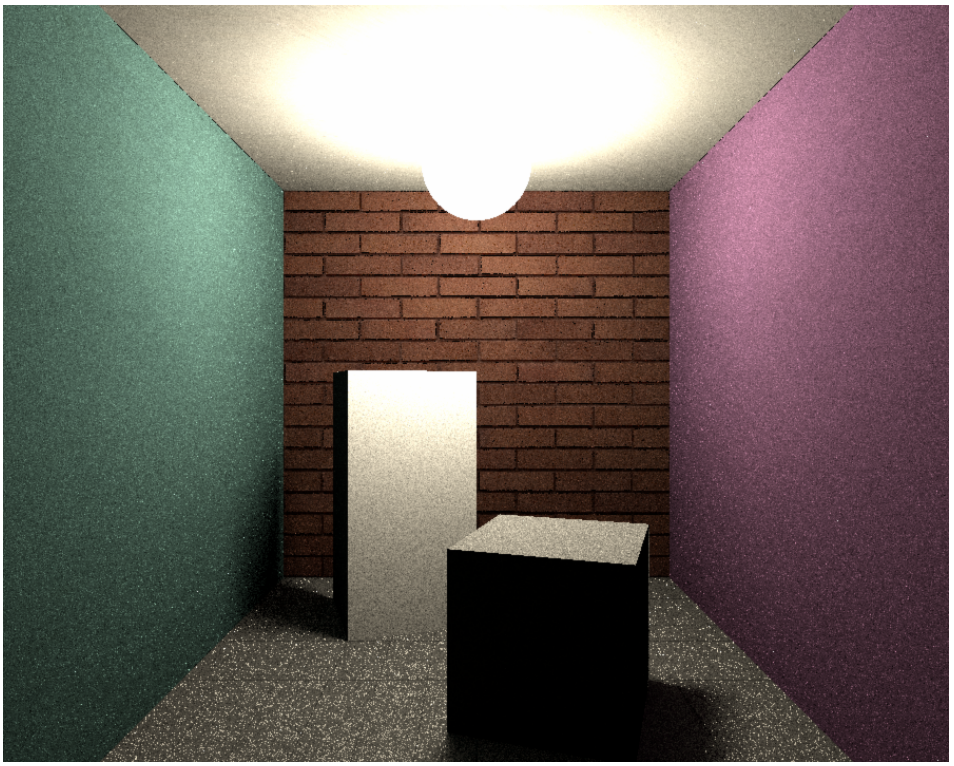

Image 6: Distributed ray tracer, 40 samples/bounce, render time: 9480 seconds (2h 38 min)

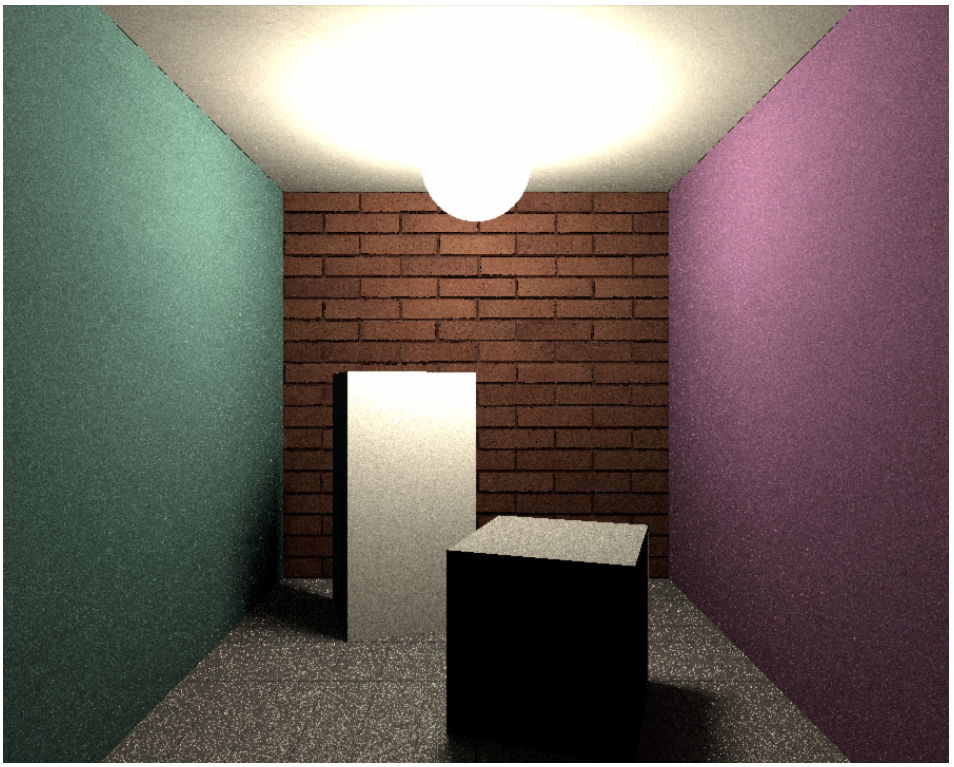

Image 7: Distributed ray tracer, 50 samples/bounce, render time: 17750 seconds (4 h 55 min 50 s)

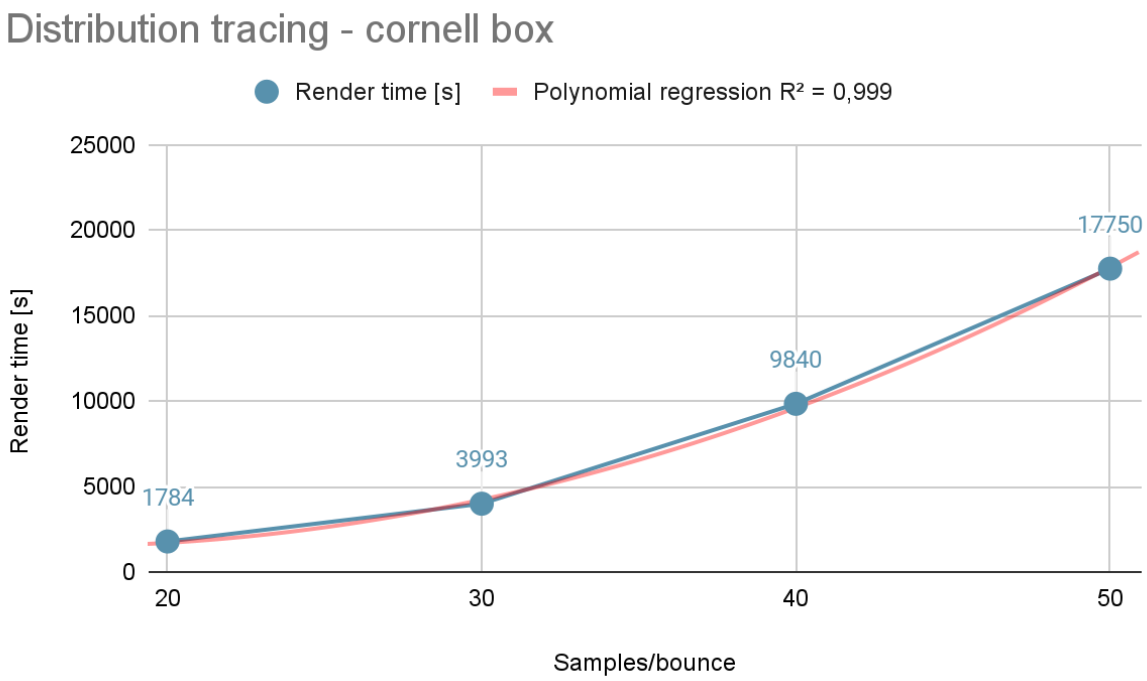

Diagram 2: plotting and polynomial regression of render time in relation to samples/pixel (distributed ray tracer)

6.2. Visual comparison between the engines

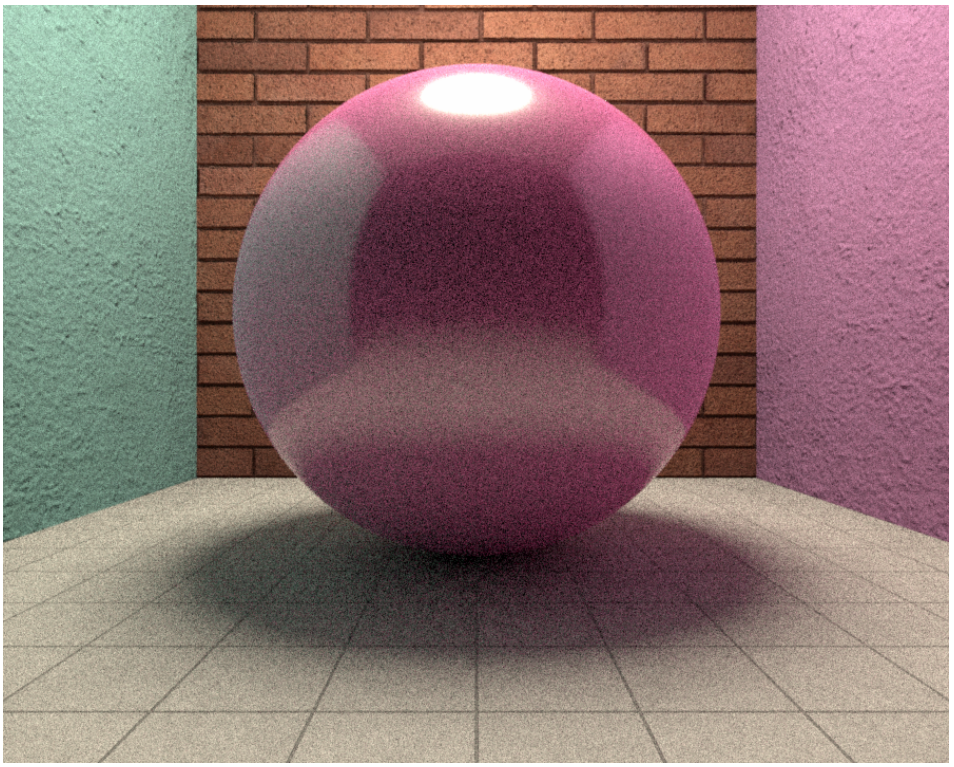

Image 8: pink dielectric sphere (path tracer). 1 750 samples/pixel (11 min 47 s)

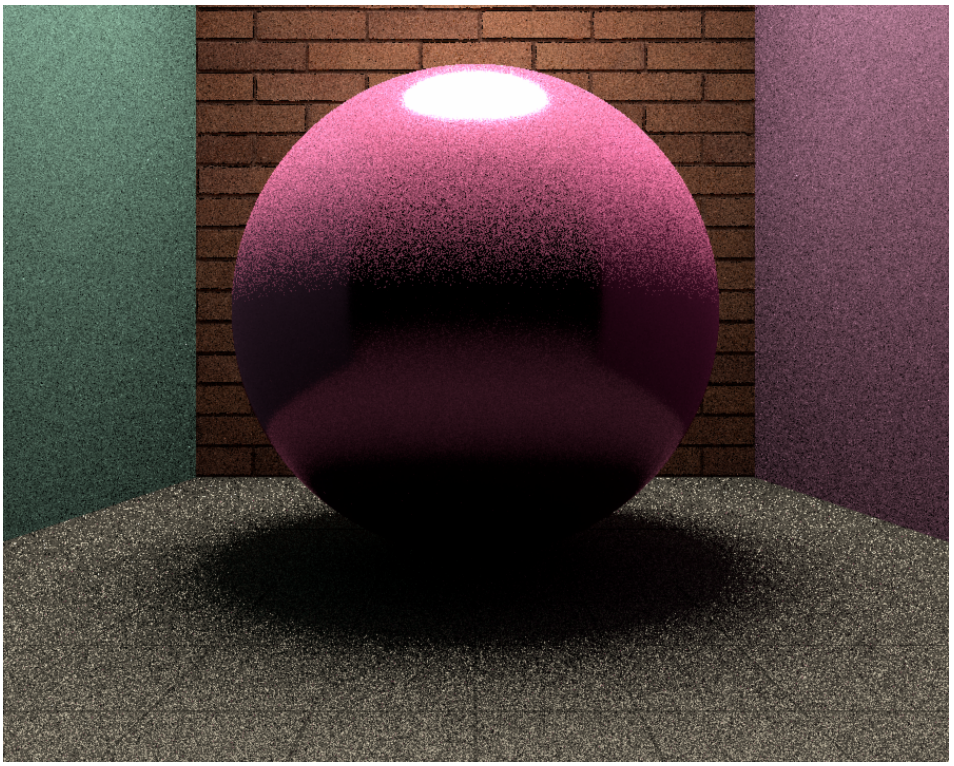

Image 9: pink plastic sphere (distributed ray tracer). 20 samples/bounce (11 min 28 s)

7. Discussion and Conclusion

In this report it has been shown how implementations of the two ray tracing algorithms "path tracing" and "distributed ray tracing" can be created, in section 5. From the data shown in section 6, it is clear that the path tracing algorithm suffers less from visual artifacts than the distributed ray tracing algorithm and that it is more realistic. It is also clear that the distributed ray tracing algorithm has a worse algorithmic complexity than the path tracing algorithm and is, as such, expected to perform worse with respect to the number of samples used.

In image 1, 2 and 3, the visual quality of the path tracer can be seen for differing numbers of samples per pixel. From these images it is clear that the amount of noise in the image appears to decrease as the number of samples per pixel is increased. This is to be expected as sampling the light in more places in the scene improves the accuracy of the Monte Carlo integration method used to approximate solutions to the rendering equation as highlighted in section 5.2.1. In these images the effects of various components of the rendering equation can also be observed. For example, the transparent sphere in the middle of the image with an index of refraction of 1.04 was used to test the BTDF of the rendering equation. As can be seen near the edges of the sphere, the light refracting through the sphere is indeed bent, as one would expect from a transparent sphere with an index of refraction greater than one in the real world. Another phenomena of the transparent sphere worth highlighting is that of caustics. A caustic is a pattern of light caused by light traveling through a transparent refractive medium being unevenly distributed around a surface. A caustic can be seen under the transparent sphere in the renders, as light from the lightsource on the ceiling is distributed unevenly when traveling through the transparent scene due to its refractive nature. The effects of the different BRDFs examined in section 5.2.1. can also be observed in the scene as the gold sphere rendered with the metal BRDF and the green sphere rendered with the plastic BRDF have a more specular appearance than the rough walls rendered with the diffuse and dielectric BRDFs.

Additionally, based on the collected data, analysis of the theoretical and experimental computational complexities for both algorithms can be made. Based on the method of sampling for the path tracer, it is to be expected that the computational complexity of the algorithm grows linearly with respect to the number of samples taken per pixel. From the linear regression seen in diagram 1 it can be observed that the R^2 number is close to being equal to 1, which in turn means that the data points closely match the plotted linear function. Hence, it can be said that the computational time complexity of the path tracer, in relation to the number of samples per pixel, indeed is linear and can therefore be assigned a big O notation of $O(n)$. Additionally, a conclusion can also be obtained from diagram 2, plotting the render time of the distributed ray tracer in relation to the number of bounces. As a polynomial regression was most suited to the data points and the value of the R^2 number is very close to 1 it can be said that the

computational time complexity in these render tests follow a polynomial. It is known that the renders for the distributed ray tracing engine was done with a maximum of two bounces from the initial ray, meaning the initial ray creates a new distribution where each ray in said distribution itself creates a new distribution and so on one more time, thereby raising the count of samples per bounce to the power of three. Therefore, it can be said with confidence that the degree of this polynomial is three and that would give the algorithm a big O notation of $O(n^3)$ where n is the number of distributed rays spawned per reflection and the maximum allowed number of reflections per ray is 3. For the general case, the algorithm has a computational complexity of $O(n^k)$ where k is the number of allowed reflections per initial ray.

Furthermore, when observing the renders of the distributed ray tracer, especially image 9, it can be observed that there is more noise on highly diffuse surfaces than on more specular surfaces. This is to be expected based on the algorithm. Unless the maximum bounce count has been reached, the algorithm always spawns the same number of new rays per ray reflection. This means that diffuse surfaces with a wider reflection distribution have more variance in their reflection samples, as there are fewer samples being done per solid angle. The increase in sampling variance for wider reflection distributions result in points on diffuse surfaces having higher variance and therefore appear to be more noisy. This suggests that the distributed ray tracing algorithm is not well suited to accurately sample highly diffuse reflections, as it would require a much larger number of samples per reflection, which would be problematic for an algorithm that is exponential in complexity with respect to the number of allowed reflections per ray. When comparing image 9 rendered with the distributed ray tracer with image 8 rendered by the path tracer in approximately the same time, it is clear that the path tracer on average suffers less from noise in the final image. This also matches theoretical expectations as the method of integration used by the path tracer samples the scene more sparingly. The russian roulette model used by the path tracer ensures that the method spends less time on samples with little contribution to the integration. The distributed ray tracer on the other hand does not take the importance of samples into account. This is demonstrated clearly by the fact that when a ray is reflected, each reflection of the ray spawns equally many new rays for the first $n$reflections. This is usually wasteful as later reflections generally contribute less to the image, and should thereby not be sampled as much. Apart from distinct differences in noise between image 8 and 9 other visual differences can be observed as well. The top half of the sphere in image 9 appears much brighter than what would be expected from a sphere using the plastic BRDF. The reason for this is that as the distributed ray tracer calculates direct lighting for a given surface, it assumes that the surface is diffusely reflecting the direct light. This is not the case for specular surfaces, hence the lighting looks inaccurate.

Potential sources of error can be found in, for example, the render time measurements as the system used to render the images was at times running other software in the background that might decrease the performance of these engines. Additionally, to ensure that the conclusions derived from these tests are actually accurate the tests should be repeated with more variation in variables as the observed results might have been caused by other factors such as the specific geometry of the scene or properties of the materials used. Furthermore, as no one is a perfect programmer, an application will theoretically not ever be at its best, meaning that these implementations alone do not represent how such algorithms always perform. An algorithm may be described to behave a certain way but it is important to realize that each implementation comes with much freedom to both tweak and add but also fill in the gaps of what might not always be described when researching how a certain algorithm is supposed to work.

An improvement that could be made if one were to do this study again would perhaps be to focus on only one algorithm implementation as making two engines limited the possibility of what could be included in the algorithms. As the aim was to provide a good example of what one might want to include in a realistic ray tracing algorithm, it would have been better to demonstrate more features with the path tracing algorithm. As an example, there were no performance enhancing features added, such as k-trees for more triangles or any GPU implementation to drastically parallelize the computations. Solely focussing on path tracing would allow for these things to be added as well, and therefore giving a more detailed study on the algorithm as one could, for example, study the ease of implementing parallelization and the performance increase it would bring.

Lastly, one could conclude from this study that it does not make much sense to use distributed ray tracing due to the aforementioned noise issues and the fact that the implementation was not even faster than that of the path tracer when comparing visual realism to time taken to render. Path tracing is used widely together with denoising filters that removes the need to render with as many samples as used in the renders of this study. What one might want to study further is how denoising filters help improve the quality of the two algorithms and how they would compare against each other then.

8. References

[1] Nadia Magnenat-Thalmann & Daniel Thalmann, *Image Synthesis*, Springer, Tokyo, 978-4-431-68060-4.

[2] Per H. Christensen & Wojciech Jarosz, Foundations and Trends in Computer Graphics and Vision, "The Path to Path Traced Movies", https://graphics.pixar.com/library/PathTracedMovies/paper.pdf, 2016.

[3] Michael Mara et.al., ACM SIGGRAPH / EuroGraphics High Performance Graphics, "An Efficient Denoising algorithm for Global Illumination", https://cs.dartmouth.edu/wjarosz/publications/mara17towards.pdf, 2017.

[4] Henri Ylitie, Tero Karras & Samuli Laine, NVIDIA, Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs, https://research.nvidia.com/sites/default/files/publications/ylitie2017hpg-paper.pdf, 30-07-2017.

[5] Bruce Walter et.al., Eurographics Symposium on Rendering, Microfacet Models for Refraction through Rough Surfaces, https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf, 2007.

[6] Sébastien Lagarde, Memo on Fresnel equations, https://seblagarde.wordpress.com/2013/04/29/memo-on-fresnel-equations/, 29-04-2013.

9. Appendices

Source code: https://github.com/Benjaneb/3DRenderingTechniqueAnalysis

Rendered images:

https://drive.google.com/drive/folders/1qLwRbFJEqdFp2MU_PZq2PhO8HlahYhxV?usp=sharing