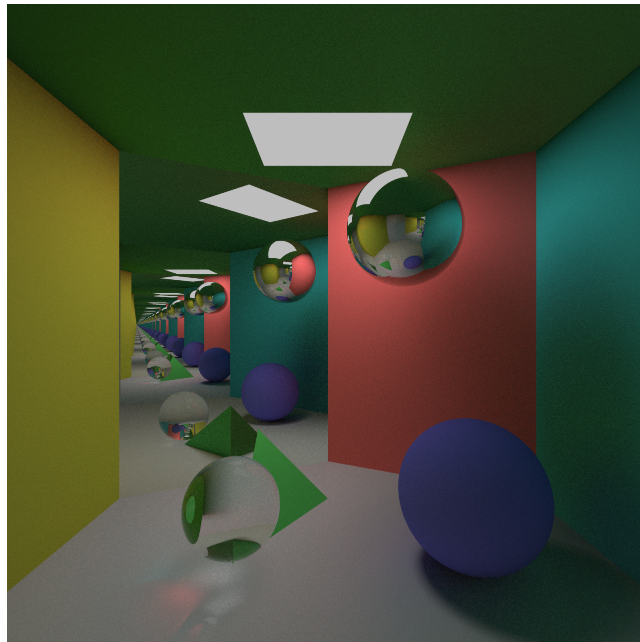
TNCG15 - Monte Carlo Raytracer

A Monte Carlo raytracer written in C++

Ludwig Boge, Jonatan Ebenholm



Faculty of Science and Engineering
Linköping University
May 28, 2026

Abstract

This report first introduces the importance of ray tracing and describes methods that combined will achieve photorealism in rendered images. The definition of a ray is presented and the way objects are represented is explained. Different intersection algorithms are explained, Möller-Trumbore and Ray-sphere intersections. Afterwards the report explains depth tests for the objects in the scene. A camera and a basic room is defined and different surfaces are introduced. The surfaces implemented are Lambertian reflectors, perfect mirrors, transparent surfaces and area light sources which are defined as a surface. Then, the rendering equation describing the radiance is introduced and Monte Carlo estimators are introduced in order to estimate the integral. A Monte Carlo estimator is then derived for calculating the direct illumination of a point by the light source.

In the result, the benchmarks of the rendered image is presented, depending on variables such as the number of shadowrays and number of samples during rendering.

In the discussion some points about the result are brought up and what further development might entail is described. Afterwards, the issue of floating point precision is presented and how it was solved in this project.

1 Introduction

Raytracing is a subject that has been studied ever since computers have been able to render images. Whitted raytracing was introduced in 1980 and showed good results for calculating reflections and refractions but did not take into account global illumination and thus making it not photorealistic. This project aims to implement true global illumination by taking into account reflected light off diffuse surfaces and by solving the rendering equation using Monte Carlo integration.

2 Background

In this section the theory needed and equations used to implement the raytracer will be presented.

2.1 Rays

A ray as defined in computer graphics is a cylinder with an infinitesimally small diameter, which in the case of a light ray carries light from one end of the cylinder to the other end. In this project the medium in which the ray travels is non-participating. A participating medium would be for example fog which scatters the light ray, but since there is no participating medium the same amount of light that enters the cylinder also leaves on the other end, and it does so instantaneously. Because rays are infinitesimally small it means they cannot carry power, so instead the ray carries *radiance* which a measure of brightness in a ray.

Rays have a few key features. They have a starting point and a direction in which it travels along, and an end point in which it will stop travelling.

In a Monte Carlo raytracer, rays will bounce around in the scene, the mechanics of their direction and other qualities are described in 2.4, but in short; when a ray hits a surface it may bounce in another direction from the point it hit. The way that this is handled is with a doubly linked list in the ray class which connects the rays path and allows for the path of the ray to be traced, as well as the change in radiance. The change in radiance along the ray path is explained further in 2.4 since it depends on the surface the ray hits.

2.2 Objects and intersections

In this project only a select few objects were implemented. These objects were: Rectangles, triangles, tetrahedrons, and spheres. All of the objects bar the sphere are defined using vertex points in the scene space and can therefore have their surface normals calculated beforehand. The dot product between the ray direction and the normal of an object is first checked to see if it is negative, if it is not, then the object surface is pointing away form the ray and can therefore not hit it and does not need to have an intersection test computed. The sphere,

however, is defined using its' mathematical definition, which will be important when defining the intersection points of the rays in respect to the sphere. The rectangle and the tetrahedron in this project are also defined using triangles, which is to say that everything in the scene that is not a sphere is compromised of triangles. for the tetrahedron it is obvious why this is done, but for the square its' not as intuative. However, this simply has to do with the fact that using the Möller-Trumbore surface intersection algorithm is a faster operation than the course suggested surface intersection algorithms for rectangles which also is not as general.

2.2.1 Möller-Trumbore algorithm

The Möller-Trumbore algorithm takes the edges of the triangle using its' vertices

$$\begin{aligned} \mathbf{e_1} &= \mathbf{v_1} - \mathbf{v_0} \\ \mathbf{e_2} &= \mathbf{v_2} - \mathbf{v_0} \end{aligned} \tag{1}$$

, where $\mathbf{e_1}$ and $\mathbf{e_2}$ are the calculated edges and $\mathbf{v_0}$, $\mathbf{v_1}$ and $\mathbf{v_2}$ are its' vertices. The algorithm essentially parametrises these values using barycentric coordinates u and v which gives the point of intersection given the equation

$$\mathbf{T}(u,v) = (1-u-v)\mathbf{v_0} + u\mathbf{v_1} + v\mathbf{v_2}. \tag{2}$$

Combining this with the equation for defining the endpoint of a ray

$$x(t) = \mathbf{p_s} + t\mathbf{d} \tag{3}$$

, where $\mathbf{x(t)}$ is a point along the ray, $\mathbf{p_s}$ is the starting point on the ray, t is the distance the ray has traversed and $\mathbf{d}$ is the ray direction, we can derive the equation

$$\mathbf{T} = \mathbf{ue_1} + \mathbf{ve_2} - t\mathbf{d}. \tag{4}$$

An intersection has occurred when the equation is fulfilled with the requirements that

$$0 \leq u, v \leq 1 \quad and \quad u + v \leq 1. \tag{5}$$

If an intersection has occurred then the distance of the ray can be found with the equation given from equation 4

$$t = \frac{\mathbf{Q} \cdot \mathbf{e_2}}{\mathbf{P} \cdot \mathbf{e_1}} \tag{6}$$

where $\mathbf{Q}$ is the cross product between $\mathbf{T}$ and $\mathbf{e_1}$, and $\mathbf{P}$ is the cross product between $\mathbf{d}$ and $\mathbf{e_2}$.

2.2.2 Ray-sphere intersection

Since the sphere does not have a normal pre-defined all spheres in the scene has to have their intersections tested and the normal set after an intersection has occurred. The equation of a sphere is given by

$$(\mathbf{x_c} - \mathbf{C})^2 = \mathbf{r}^2 \tag{7}$$

, where $\mathbf{x_c}$ is any point on the surface of the sphere, $\mathbf{C}$ is the center of the sphere, and $\mathbf{r}$ is the radius vector from the center of the sphere to the surface. The ray equation was given by equation 3. By inserting the ray equation into the surface point on the sphere $\mathbf{x_c}$ we get

$$\mathbf{r}^2 = (\mathbf{x_r} - \mathbf{C})^2 = ((\mathbf{p_s} - \mathbf{C}) + \mathbf{D}t)^2 \tag{8}$$

Substituting the coefficient of the expanded equation given in 8 gives a solution to the distance t with the equation

$$t = \frac{-c_2 \pm \sqrt{c_2^2 - 4c_1c_3}}{2c_1}. \tag{9}$$

The project was optimized using the fact that the values within the square root give away in what manner the sphere was hit. From 9 we get that

$$arg = c_2^2 - 4c_1c_3 \tag{10}$$

if $arg < 0$ then no intersections have occurred and no further calculations need be computed. If $arg = 0$ then the ray has touched the sphere, only hitting it once. If $arg > 0$ then the ray has intersected the sphere twice. Finding the hit point with the lowest distance value is of course the closest, and therefore also pointing towards the ray direction and the first point in which the sphere was hit.

2.2.3 Picking a surface

A single ray may intersect multiple surfaces in the scene during the intersection tests, but this of course does not make sense with reality. Therefore, a surface must be picked in accordance with reality in which the end point of the ray will end up in. This is done simply by picking the surface with the smallest distance t, since any other object further along the ray would have to be behind said object.

2.3 Camera and room

The camera consists of a point in which the camera lays and a camera-plane which is used to send the rays out into the scene in an appropriate direction. Depending on the desired resolution a certain number of rays are sent out at every pixel of the plane. Assuming that an HD image with the resolution 1920x1080 pixels is rendered, that number multiplied by the amount of samples at each pixel is the amount of rays that are sent into the scene in total, not counting bounced rays as additional ones. The camera point in this project is in the coordinates (-1,0,0) and is looking towards the positive x direction. The camera plane is in between (0,-1,-1) and (0,1,1). To send out rays into the pixels at random for sampling we find the coordinates u and v where u is position in the y-axis (width) and u is the position in the z-axis (height) where they are both scaled between -1 and 1. The room that the project has as a bounding box is a elongated hexagon as a floor and roof, and straight walls. The room can be seen in figure 1 which was a render done early on in the project.

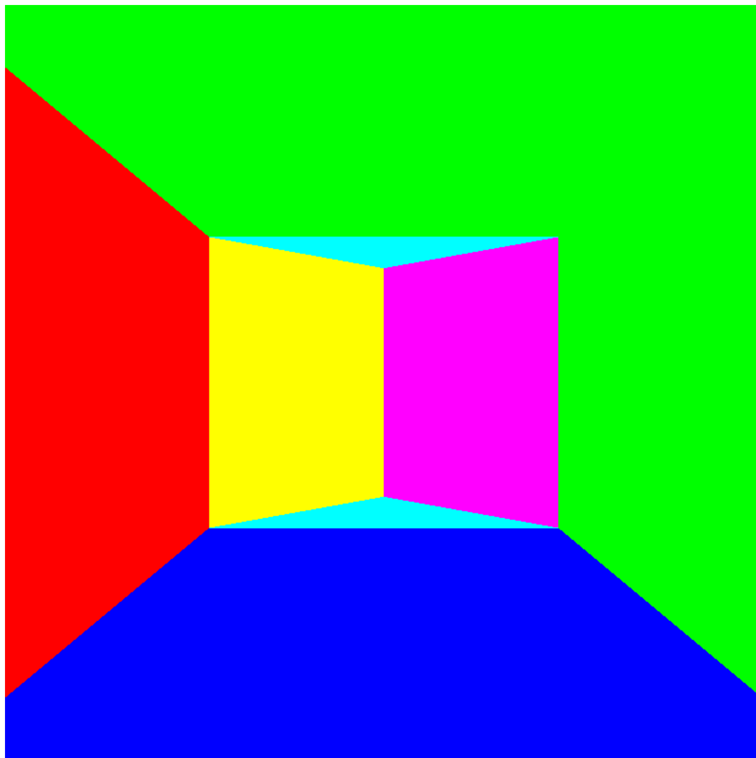


Figure 1: Early render of the room.

2.4 Surfaces

The raytracer created in this project is designed to handle three different types of surfaces. Lambertian reflectors which reflect diffuse light, transparent surfaces that reflect and refract light into the surface and perfect mirrors which perfectly reflect all light.

2.4.1 Lambertian Reflectors

A Lambertian reflector is an ideal diffuse reflector that disperses light into all possible directions according to Lambert's cosine law. When a ray hits a Lambertian surface, we transform to the local coordinate system and generate a random azimuth and inclination angle. These random values are then used in the Russian Roulette ray termination scheme to determine if the raypath should be terminated or reflected based on the reflectance of the surface. If it is not terminated, one new ray is sent from the surface in the direction determined by the angles. Since we send lots of rays through each pixel, we can get away with just reflecting one ray from the surface while keeping the scheme photorealistic.

When traversing the tree of rays from the end of the raypath, the radiance of the ray is multiplied by the color of the surface. If the surface is directly hit by a lightsource, the direct illumination from that lightsource is also added to the ray's radiance. The direct illumination is calculated by a Monte Carlo estimator, more on that later.

2.4.2 Perfect Mirrors

Perfect mirrors reflect all incoming light in the perfect reflection direction. This direction is easily calculated directly in the global coordinate system using equation 11.

$$d_o = d_i - 2(d_i \cdot N)N \tag{11}$$

where d_o is the outgoing direction vector, d_i the incoming direction vector and N is the normal of the surface.

When traversing the raypath from the end, the radiance from the previous ray is transferred to the next ray since all light is reflected in a perfect mirror.

2.4.3 Transparent Surfaces

Transparent surfaces are surfaces that both reflect and refract light according to Snell's law based on the refractive index of the object and the index of the outside object, in our case air. The indecies have in this program been set to be that of air and glass so that all transparent surfaces act like glass. When a ray hits the surface, it is in theory split up into two rays, a perfectly reflected ray and a perfectly refracted ray. Since the program works by traversing the raypath from the end of the raypath towards the eye, splitting of the raypath

has to be avoided. Instead of splitting the path, a similar approach as Russian Roulette is used. The probability of reflection $R(\theta)$ is calculated using Schlick's approximation of the Fresnel factor which can be seen in equation 12. A random number between 0 and 1 is then generated, if the number is below $R(\theta)$, the ray is perfectly reflected and otherwise perfectly refracted into the surface.

$$\begin{aligned} R(\theta) &= R_0 + (1 - R_0)(1 - \cos(\theta))^5 \\ \text{where } R_0 &= (\frac{n_1 - n_2}{n_1 + n_2})^2 \end{aligned} \tag{12}$$

where θ is the incident angle, n_1 the refractive index of the current medium and n_2 of the outer.

Once the ray is inside the object it will hit another point inside the object, at this point it can either keep reflecting or refract out of the object according to the same probability scheme used for the initial hit. This is unless total internal reflection occurs (TIR). TIR occurs when the external medium has a lower refractive index than the internal one and the incident angle is lower than a certain threshold. The threshold can be determined by the condition seen in equation 13. If true, then TIR has occurred and the ray should be perfectly reflected, otherwise use the same probability scheme as was used for the initial hit.

$$\frac{n_1}{n_2} \cdot \sin(\theta) > 1.0 \tag{13}$$

When accumulating radiance along the raypath starting at the end, the radiance of the previous ray is transferred into the next ray. Transparent objects as implemented in this program thus do not affect the color of the radiance and like with perfect mirrors only lets the radiance pass through it.

2.4.4 Area light sources

Since the type of light source implemented in the raytracer is a area light source, it will be treated as if a surface. Rays can hit the lightsource but not reflect off it since the contribution would be very small in comparison to that of the light source. If a ray hits an area light, the raypath is ended and the radiance of the ray is assigned the emitted radiance of the light source. This is one of the two ways a light source contributes with light to the scene in the current implementation. The other way is through direct illumination of Lambertian surfaces by sampling a certain number of points on the light source and determining the illumination of the surface through a Monte Carlo scheme.

2.5 The Rendering Equation

The rendering equation describes the radiance of an outgoing ray from the camera. The equation is written as

$$L(x \to \theta_o) = L_e(x \to \theta_o) + \int_\theta f(x, \theta_i, \theta_o) L(x \leftarrow \theta_i) cos(\Omega_i) d\theta_i. \tag{14}$$

$L(x \to \theta_{o)}$ is the computed radiance from hit point x with the outgoing direction θ_o, which is specified with the inclination angle Ω_o and the azimuth φ_o. $L_e(x \to \theta_o)$ is the direct illumination contribution on the point. The equation integrates over the of the half sphere on the surface, which is θ. $f(x, \theta_i, \theta_o)$ is a bidirectional reflectance distribution function that depends on the hit surface, and $L(x \leftarrow \theta_i)cos(\Omega_i)$ is the incoming irradiance where i means incoming as opposed to the previous outgoing.

2.6 Monte Carlo Estimators

The rendering equation defines how the lighting is calculated mathematically using a double integral. This cannot be solved using a computer since the integrals are continuos, hence a estimator has to be created. This estimator is a Monte Carlo estimator and uses random numbers to estimate the integral. The general estimator that can be derived from a two-dimensional integral can be seen in equation 15.

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i, y_i)}{p(x_i, y_i)} \tag{15}$$

In the equation, $f(x_i, y_i)$ and $p(x_i, y_i)$ are the function and probability distribution function sampled using two random points. When the sum goes to infinty, the result converges to the exact value of the original integral.

2.7 Direct illumination

Points on diffuse surfaces can be hit by the lightsource directly and this is called the direct illumination. It works by using a Monte Carlo estimator to solve the rendering equation. To solve the rendering equation one has to project the lightsource onto the hemisphere and then integrate over the projection to calculate the contribution of the lightsource from all possible incoming directions. This is not possible on computers since there can be an infinite amount of directions, instead one can select a certain number of random points y_i on the lightsource and calculate the average radiance of these points, giving us an estimation of the light contribution from the lightsource. The estimator for this can be seen in equation 16.

$$\begin{aligned}\langle L_D(x)\rangle &= \frac{AL_e}{N}\sum_{i=1}^{N} f(x,d_i,\theta_o)V(x,y_i)\frac{\cos\Omega_{x,i}\cos\Omega_{y,i}}{\|d_i\|^2} \\ d_i = y_i - x, \quad \cos\Omega_{x,i} &= N_x \cdot \frac{d_i}{\|d_i\|}, \quad \cos\Omega_{y,i} = -N_y \cdot \frac{d_i}{\|d_i\|}\end{aligned} \tag{16}$$

This equation consists of the area A of the lightsource, the radiance of the lightsource L_e and the visiblity function $V(x, y_i)$ which is used to check if a object is in shadow. For each iteration of the sum, one shadowray is sent to the point on the lightsource, if it intersects an object before reaching the light, the visibility function is set to 0, otherwise 1.

3 Results

3.1 Benchmarks

In order to investigate the performance of the raytracer, a benchmark scene has been created and can be seen in figure ... The scene took 795 seconds to render with a resolution of 800 x 800 pixels, 50 samples and 15 shadowrays. In addition, experiments were held to examine how the render was affected by different numbers of shadowrays and samples.

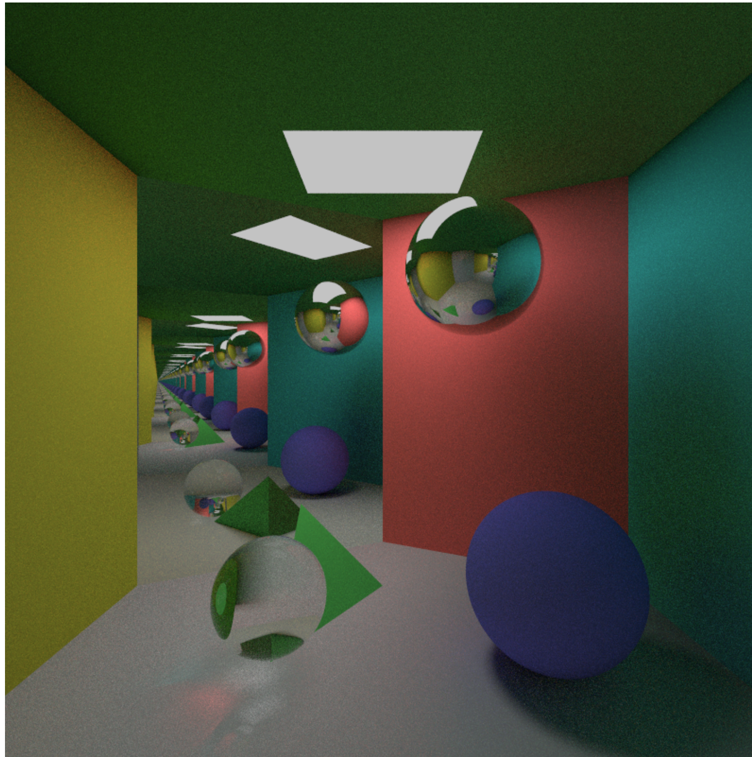
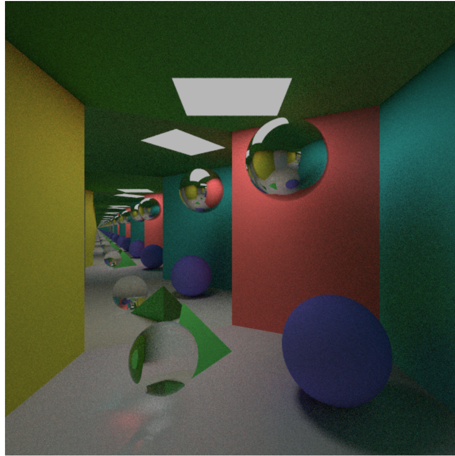


Figure 2: Benchmark render.

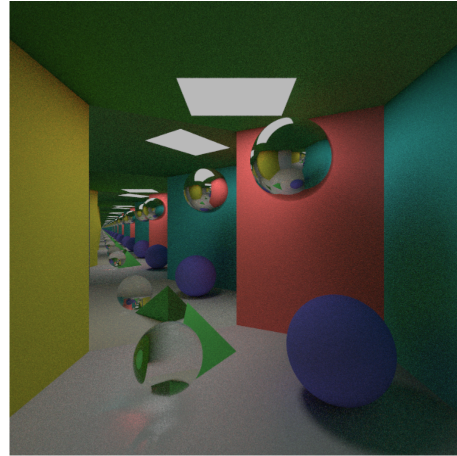
3.1.1 Number of Shadowrays

One of the factors affecting the performance of the raytracer is the number of shadowrays and points on the light source being sampled when calculating the direct illumination. Different numbers of shadowrays were used to generate the images seen in figure 3. The difference in quality is mainly noticed around the shadows since more shadowrays sent to different parts of the lightsource gives more accurate soft shadows. Since a relatively high sample count of 50 was used to generate all the images, the difference is not as noticable as with a lower

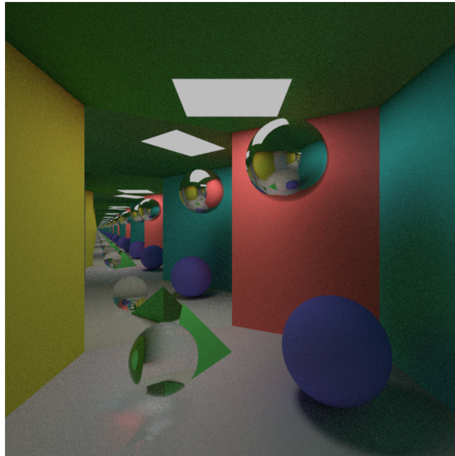
sample count. What is noticeable however is the difference in computing time which can be seen in table 1.



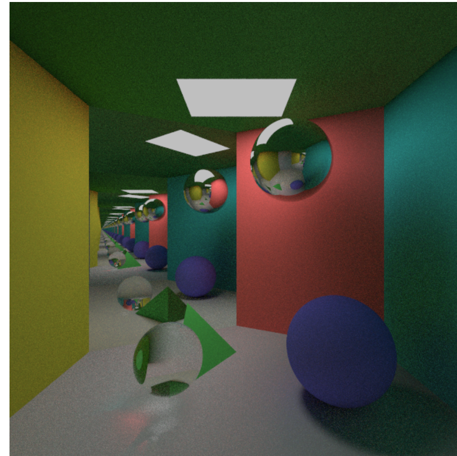
(a) 1 shadowray



(b) 2 shadowrays



(c) 8 shadowrays



(d) 15 shadowrays

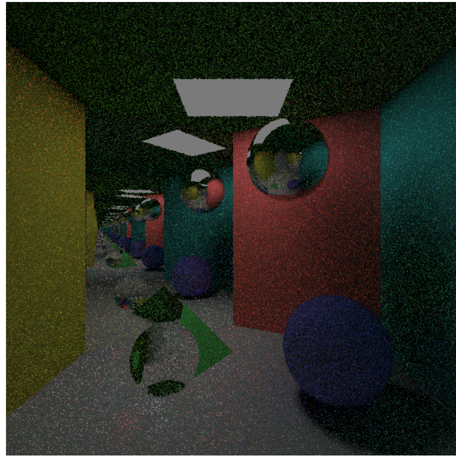
Figure 3: Number of Shadowrays Benchmark, 50 samples

Shadowrays	Time (sec)
1	84.4
2	107
8	236.1
15	384.8

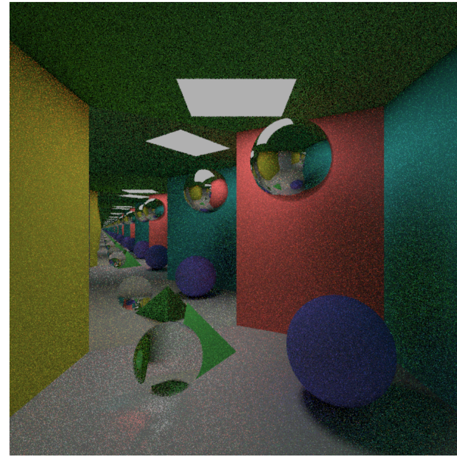
Table 1: Table of number of shadowrays as compared to time taken to render the image

3.1.2 Number of Samples

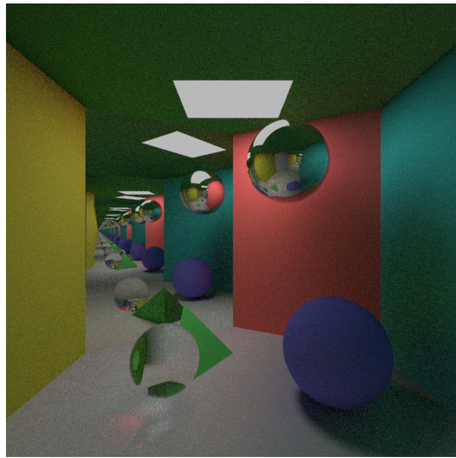
As is shown in the images in figure 4, the amount of samples matter a lot for lower samples when it comes to quality of rendered image. However, it only matters to a point, because while the difference between 25 and 75 samples are noticeable, they are not very dramatic. However, with that being said the amount of samples do matter in respect to render time, and that is especially true with higher amount of samples. As seen in table 2 it took 2.5 times longer to render 75 sampled images over 25. This is a big increase in time, making it go from five minutes to twelve for a marginally low increase in quality.



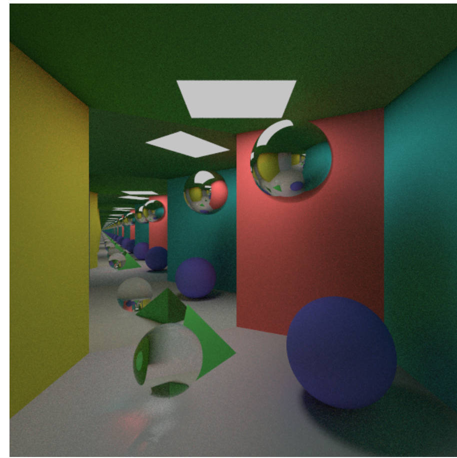
(a) 1 samples



(b) 5 samples



(c) 25 samples



(d) 75 samples

Figure 4: Number of samples Benchmark, using 15 shadowrays on every image.

Samples	Time (sec)
1	10.44
5	56.98
25	282.21
75	726.71

Table 2: Table with number of samples as compared to time taken to render the image.

4 Discussion

4.1 Discussing results

From the results it is clear that having too low of a sampling rate makes the image look messy and not photorealistic at all, but too high and it suddenly does not change enough to rationalise the necessary computing power. If this project made use of the GPU instead of doing what it currently does, which is to run on the CPU and non multi-threaded, the necessary computing power might be plausible even for real time rendering, meaning it is less of a set in stone measure and more of a question of on what hardware the program is currently running on.

For the shadow rays it is the same story. The more shadowrays the better the result, but by some point the increase in quality is negligible. Having more than one or two might be worth it though since it allows for softer shadows that do not look as weird.

4.2 Floating point precision

For objects like the sphere there were issues with floating point values where sometimes, after hitting a sphere in the scene, the ray would immediately hit the same sphere at pretty much the exact same point on its way out. To account for this, if a hit object was closer than epsilon (1e-8), then it was not considered.