Monte Carlo Ray Tracer

TNCG15 - Linköping University

Johan Beck-Norén, johbe559@student.liu.se
Andreas Valter , andva287@student.liu.se

December 13, 2013

Abstract

This report begins by outlining an introduction to the field of global illumination for computer graphics and discussing some basic concepts. The background section goes on and describes algorithms and techniques implemented for this project in more detail, specifically what methods and techniques were used to approximate global illumination in a computer generated scene. The report focuses on the techniques required for an implementation of a stochastic global illumination algorithm by using Monte Carlo integration to estimate a solution to the rendering equation, and presents the results from our implementation using that technique. It will cover topics like intersection testing, anti-aliasing, seperating the rendering equation into direct and indirect illumination integrals, as well as and other techniques that has been implemented during the project. Results are presented in the form of images rendered with varying settings and evaluated, along with data tables describing variable values compared to rendering times. The report is concluded by a discussion section where we discuss our implementation along with possible improvements that could be the subject of further work.

Contents

1 — Introduction

Realistic lighting in computer graphics scenes has been a challenge and a big area of research for many years. The equation commonly used describing light propagation through a scene is recursive in nature and infinite, which is something that computers can not handle. Researchers has found different methods to estimate the analytic solution. These methods are iterative and use computing power to calculate these estimates that converge towards the real solution.

Many of these are so called ray tracing methods where rays are traced from the viewer into the scene. When rays hits surfaces in the scene, new rays for reflection and refraction are spawned. If a ray hits a light source or reaches a given recursion depth it terminates. These are then combined using importance to create the final image. Other methods start from the light source and from there propagate through the scene. Both of these methods are called global illumination methods because they take the whole scene into consideration. In this section we will give a short overview of some of the different approaches and techniques in global illumination.

1.1 Whitted ray tracing

The Whitted ray tracing method described by Turner Whitted in [2] was the first global illumination method that took the whole scene into consideration when calculating intensity values. In previous so called ray marching solutions, the calculations stopped as a ray hit a surface and local lightning models like Lambert's cosine rule to model perfectly diffuse reflections, or the Phong model.

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\mathbf{N} \cdot \mathbf{L}_j) + k_s \sum_{j=1}^{j=ls} (\mathbf{R}_j \cdot \mathbf{V})^n \tag{1.1}$$

The Phong model given by Equation 1.1 is a local lighting model. It assumes that each light source L_j is located at a point with infinite distance to the objects in the scene. The total reflected intensity I is calculated using the contribution by ambient light I_a together with two terms, one term for diffuse light and one specular term. The two constants k_d and k_s are reflection constants that regulate their contributions. The diffuse term uses the Lamberts cosine law to describe the color using a dot product between the normal and the light direction to calculate if that part of the surface is directed towards the light source.

The specular term uses the direction that a perfect reflection $\bar{V}$ from the light source would take on the surface together with a ray towards the viewer $\bar{V}$.

Whitted extended the previous techniques and allowed for rays to bounce in the scene and thus creating the basis for global illumination. The solution he presented allowed for true reflections, shadows and refractions. By using the Phong model on each ray hit and spawning new rays recursively for reflections, shadow rays and refractions, a method for simulating global lightning is created.

When using several rays in the scene, a term called importance is introduced. It states that the total importance of all rays for each pixels is one. If multi sampling is used, importance is divided among the rays. The importance flows in the opposite direction as the light, from the camera and into the scene.

1.2 Radiosity

Radiosity is an algorithm for global illumination which only handles diffusely reflected light, and is therefore not dependent on any camera position or direction. This means that the solution converges toward a steady state and can for example be pre-computed for use in static scenes. The algorithm works by splitting the scene geometry into a number of finite area elements called patches and solves the rendering equation for surfaces that diffusely reflects light. A visibility factor, called *form factor*, is used to describe a patch's visibility toward all other patches in the scene. The size of a view factor is dependent on the distance between any two patches, their orientation in relation to each other, partial or total occlusion etc. and describes the amount of radiance leaving a patch j arriving at patch i for all patches $j \in N$ for a scene consisting of N number of patches. The form factors make up a system of linear equations, which when solved produces the radiosity for each patch. The process can be iterated to perform several passes and allowing for multiple bounces to be computed.

1.3 Monte Carlo ray tracing

A rendering equation (1.2) presented by Kajiya in [4] describes a full global illumination solution for a given scene. It is an integral equation called a Fredholm equation of the second kind because of the unknown radiance

quantity, $L(...)$, appears on both sides of the equation. This makes the equation recursive, corresponding to a ray's multiple bounces through a scene, and makes the equation very hard to solve analytically. It is described below using the same notation as Dutré et al. in [1], and we will use this notation throughout this report.

$$L(x \to \theta) = L_e(x \to \theta) + L_r(x \to \theta) = \\ L_e(x \to \theta) + \int_{\Omega_x} f_r(x, \Phi \to \theta) L(x \leftarrow \Phi) cos(N_x, \Phi) d\omega_\Phi \tag{1.2}$$

The Monte Carlo integration method introduces a method for calculating an estimated value for an integral expression. This is done by sampling the integral function by random discreet numbers, and as the number of samples increase the produced solution's accuracy increases as well.

$$I = \int f(x) dx \tag{1.3}$$

$$\langle I \rangle = \frac{1}{N} \sum_{i}^{N} \frac{f(x_i)}{p(x_i)} dx \tag{1.4}$$

The intergral in Equation 1.3 with a value of I can me *estimated* as in equation Equation 1.4 as $\langle I \rangle$ by N number of samples distributed according to the pdf (probability distribution function) $p(x_i)$. This method will later be used to estimate the solution to Equation 1.2.

Similar to a Whitted ray tracer, a Monte Carlo ray tracer starts by shooting rays from a virtual camera through a view plane into the scene. One difference is that Monte Carlo ray tracing handles diffuse interreflections and specular-diffuse surface interreflections, color bleeding between objects, and soft shadows by sampling area light source surfaces for each ray intersection. As the number of samples increase the more accurate the integral estimation will be and the algorithm produces noise if the number of samples are too low. There are several ways to reduce the noise. One could separate the calculations for direct and indirect illumination, using shadow rays to more quickly compute direct diffuse illumination and soft shadows. Sampling directions for diffuse indirect illumination could be chosen in more informed ways to reduce the noise. There have also been research on methods combining the benefits of radiosity with the benefits of ray tracing.

1.4 Two-pass rendering

Since a radiosity method handles diffuse reflections rather well, and a ray tracing method handles specular reflections well. a lot of research has been done to find ways to combine these two methods. One of these methods descibed by Smits et al. [7] works from the fact that radiosity algorithms propagate light from a light source through the scene, while a ray tracing algorithm only cares about the light that reaches the eye. The method in [7] consists of splitting the rendering into two passes. The first pass uses a ray tracing approach to emit importance from the eye through the scene. This importance is then used to refine the patches used for the second pass, a radiosity pass. By subdividing patches in parts with high importance, and ignoring patches in areas with little or no importance, great speedups can be achieved.

1.5 Photon mapping

Monte carlo methods for ray tracing suffers from noise in the final results if not enough samples are used when calculating radiance from diffuse indirect illumination. Photon mapping is a two-pass global illumination method presented by Jensen [3] which computes diffuse indirect illumination faster than a Monte Carlo solution. The method works by emitting packets of energy (light) from the light sources toward different objects in the scene. Different packets can be used for different materials, e.g. a high resolution photon map can be used for caustics that are visualized directly, while a lower resolution map can be stored for later use in the ray tracing step. Shadow photons can be used to more efficiently compute shadows, and be used as directional information to produce more accurate sampling directions during the rendering step. The method has been shown to reduce rendering time as well as the noise in the final image when used in a Monte Carlo ray tracer algorithm.

1.6 Iso surface ray tracing

An iso surface is a surface described implicitly by a mathematical expression or equation. An iso surface representation of a scene is preferable over polygonal objects in a ray tracing context for a few reasons. Implicitly described primitives and objects usually scale well if the resolution or precision of the illumination algorithm should increase, since they are not bound by a finite number of triangles, but rather a mathematical description. They are

also easy to define and describe, and are rather straight-forward to calculate ray intersections against. From an intersection calculation we need to decide weather a ray is intersecting the surface and if so at what position. The surface normal for that position is also needed.

2 — Background

In this section we will talk about the methods we have implemented in our Monte Carlo ray tracer using C++. Although there are several approaches to some of these methods we will only talk about the methods we chose to implement in this project.

2.1 Scene storage

When calculating global illumination for the scene, all objects need to be taken into consideration when calculating the illumination for one part of the scene due to the recursive nature of Equation 1.2. For ray tracing solutions, this means that as rays bounce inside the scene the first intersecting object needs to be located. A naïve solution would be to store all objects without any information about where in the scene they are located. When traversing a ray through the scene, the algorithm would have to check against all objects, even if almost all of the checks would be misses. For a simple scene with only a few implicit objects this is a good solution. The overhead when dealing with an more complex storage method would make it slower than simply checking against all objects in the scene. But when using a more complex scene consisting of models with hundreds of thousands of elements, it is possible to take advantage of the spatial information of each element to subtract a small subset and only do collision tests against those.

2.1.1 Bounding box

An axis aligned bounding box is an essential part of most scene storage methods. This is the smallest box with its axes aligned to the Cartesian coordinates axes that can encapsulate the primitive that owns the bounding box. The bounding box will not be the smallest box that can encapsulate the primitive but the intersection tests are much simpler compared to doing collision tests against arbitrarily rotated primitives.

In reality, any simple shape could be used to define the bounding box of another object. The general rule is that it should be much simpler to calculate intersections against that, compared to the primitive that it surrounds. The choise of bounding box is usually dependent on the encapsulated primitives shape, the importance of amount of extra volume in the bounding box or constraints in the storage algotithm.

2.1.2 Bounding volume hierarchy

Bounding volume hierarchy (BVH) uses a tree structure with bounding volumes like axis aligned bounding boxes. Small sets of primitives are divided into different bounding volumes. These small volumes are then combined to larger bounding volumes recursively until there is only one bounding volume left surrounding the entire scene. By using this technique, it is known that if the ray does not intersect a larger bounding box, the child boxes within that larger box can be ignored as well.

2.1.3 Octree

The octree method uses the bounding boxes in a slightly different way than described in the previous section. The scene is encapsulated into a large bounding box. When each primitive is added, the scene bounding box is subdivided into eight sub-volumes recursively until the current level of bounding boxes are to small to fit the primitive. The primitive is then added to the level above that as a leaf. Using this structure, it is possible to traverse the ray through the volume and, just as when using the BVH tree, large parts of the tree can be ignored by checking only parent nodes for intersection.

2.2 Intersection

The implementation needs to be able to handle several different intersection algorithms, one for each type of primitive in the scene. To make sure that all kinds of primitives are supported, inheritance is used where each primitive inherits from an abstract class. During rendering calculation, the ray tracer calls an abstract method for calculation of intersection points on the given primitive and therefore the implementation supports all primitives that can define a intersection method. Each of these methods uses an incoming ray with a direction vector d and a origin o described by Equation 2.1. The intersection methods has to calculate the normal of surfaces to decide the direction of new rays spawned at the intersection point.

$$\mathbf{p} = \mathbf{o} + t\mathbf{d}, t \geq 0 \tag{2.1}$$

2.2.1 Implicit sphere

A sphere is defined by a position vector c of its center and a radius scalar r. The goal is to find if any t in Equation 2.1 satisfies Equation 2.2. By

inserting Equation 2.1 into Equation 2.2, we receive an expression for solving the intersection point described by Equation 2.3.

$$(\mathbf{p} - \mathbf{c}).(\mathbf{p} - \mathbf{c}) = r^2 \tag{2.2}$$

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2(\mathbf{o} - \mathbf{c}) * \mathbf{d}t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0 \tag{2.3}$$

There are two solutions to Equation 2.3 described in Equation 2.4 with coefficients described by Equation 2.5. When intersecting, they represent the entry and exit point on the sphere surface. The collision point is decided dependent on if the solution is real, this means that $B^2 - 4AC >= 0$ is true for all rays that collide with the sphere.

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \tag{2.4a}$$

$$t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \tag{2.4b}$$

$$A = \mathbf{d} \cdot \mathbf{d} \tag{2.5a}$$

$$B = 2(\mathbf{o} - \mathbf{c})\mathbf{d} \tag{2.5b}$$

$$C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 \tag{2.5c}$$

When both values of t are calculated, the lowest value is used for the collision point. The normal is calculated by normalizing the point on the surface and subtracting the center of the sphere.

2.2.2 Quadrilateral

The collision tests for quadrilaterals follows the technique presented by A. Lagae and P. Dutré [5]. A quadrilateral is described by four corner points $\mathbf{v}_{00}$, $\mathbf{v}_{01}$, $\mathbf{v}_{11}$ and $\mathbf{v}_{01}$, one for each edge, listed in a counter clockwise order. These must all be on the same plane and create a convex shape. The method for collision uses bilinear coordinates to calculate the point on the plane, but this is an expensive operation and therefore, rejection tests are made early to make sure that these are only calculated when it is really needed. The quadrilateral is divided into two triangles T, T' and the collision test is decided comparing the barycentric coordinates with the two triangles.

Each point $Q(u, v)$ in the plane Q can be described by Equation 2.6. Here u and v are the bilinear coordinates of $Q(u, v)$. If u and v lies in the range

[0..1], then $Q(u, v)$ lies inside Q.

$$Q(u,v) = (1-u)(1-v)\mathbf{V}_{00} + u(1-v)\mathbf{V}_{10} + uv\mathbf{V}_{11} + (1-u)v\mathbf{V}_{01} \quad (2.6)$$

Equation 2.6 is a bilinear mapping of the unit square into a quadrilateral. It is computed by using linear interpolation along the top and bottom edges of the quad and then applying a linear interpolation between these two interpolated points.

$$T(\alpha, \beta) = \mathbf{V}_{00} + \alpha(\mathbf{V}_{10} - \mathbf{V}_{00}) + \beta(\mathbf{V}_{01} - \mathbf{V}_{00}) \quad (2.7)$$

Each point on in one of the triangles is described by Equation 2.7 and when Equation 2.8 is satisfied, the point is inside the triangle. If the ray intersects the quadrilateral, Equation 2.1 is inserted into Equation 2.8 to find t.

$$\alpha \geq 0 \quad (2.8a)$$
$$\beta \geq 0 \quad (2.8b)$$
$$\alpha + \beta \leq 1 \quad (2.8c)$$
$$(2.8d)$$

2.2.3 Triangle

Triangle collisions uses the same method as the quadrilateral for finding if a ray intersects with the triangle, using Equation 2.7 to determine if it intersects with the ray and computing t by inserting Equation 2.1 into that equation.

2.2.4 Octree

Collisions against the octree uses a iterative solution to traverse a ray trough the tree. When a ray is created, it is compared against all the top boxes. As the ray is evaluated, the bounding box closest to the ray are found, iterating down. If no collisions is found, the algorithm iterates up again, finding the smallest neighboring bounding box that has data in it. This process is visualized in Figure 2.1. The ray first collides with bounding box 1, but no child nodes exist in it, so the ray traverse forward to the edge of bounding box 2 as it collides with the sibling to 1. It now iterates down to bounding box

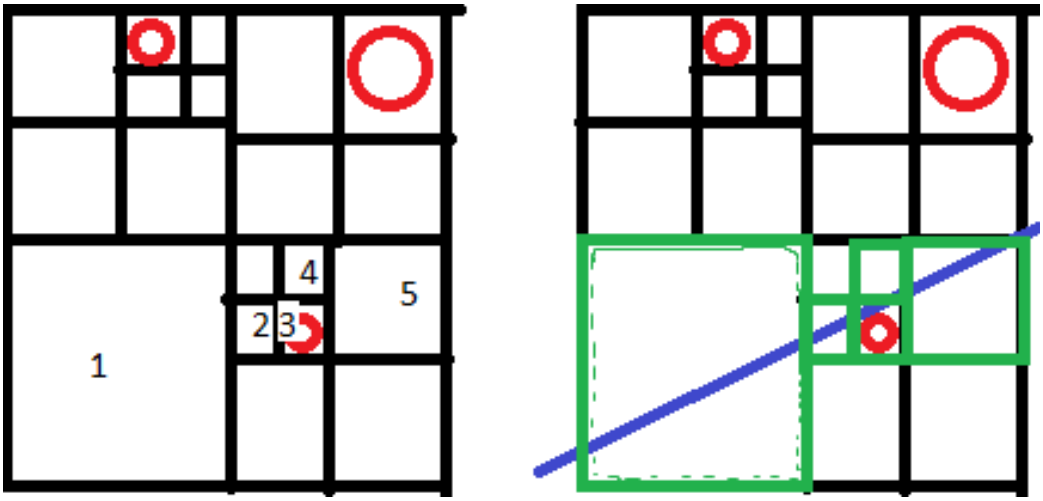


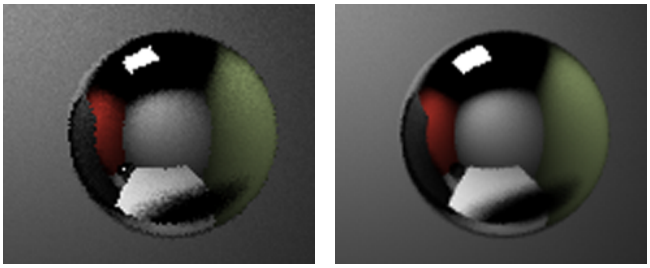
Figure 2.1: Visualization of the collision tests a ray traveling trough an octree data structure.

2 in the figure and finds that no child's are in this bounding box. The ray moves forward to bounding box 3 and finds a primitive in this bounding box, collision tests are done against it but no collision is found. The ray traverse forward to the edge of bounding box 3 and trough bounding box 4. The ray exits out of bounding box 5 and no collisions were found and therefore, no collisions between the scene and the ray were found.

This allows for using the fast and cheap AABB intersection method instead of expensive collision tests using all primitives.

2.3 Anti-aliasing by sub-pixel sampling

To reduce aliasing effects in the final image we have implemented support for sub-pixel sampling. This means that instead of sending one ray from the virtual camera through the center of a pixel on the view plane, we send several rays distributed on the pixel and average their results. The view ray distribution on the pixel area is done in a uniform fashion, placed evenly spaced on the pixel. For all rendered images in this report this anti-aliasing method was used. In Figure 2.2 we see a comparison between using 1 ray per pixel on the left which is equivalent to no sub-pixel sampling, and 10 rays per pixel on the right.



(a) 1 ray per pixel (b) 10 rays per pixel

Figure 2.2: Antialising by sub-pixel sampling

2.4 Russian Roulette

Russian roulette is a method for ray termination that specifies a probability for continuing or terminating at each bounce. How the probability is decided depends on how complex the implementation. A simple implementation only states a fixed threshold of probability for each ray. As the ray continues forward, the probability of terminating increases. More complex implementations could use reflectance of the surface to decide, giving a higher probability of termination for diffuse primitives. Light intensity could also be taken into consideration when determining the probability, giving a higher chance of termination for low intensity values.

2.5 Monte Carlo ray tracing

As described in the introduction a Monte Carlo ray tracer is similar to a Whitted ray tracer in that it is only interested in the radiance reaching the virtual camera, or the eye. It is a complete global illumination algorithm, and solves the rendering equation Equation 1.2, repeated below for convenience, by random sampling.

$$L(x \to \theta) = L_e(x \to \theta) + \int_{\Omega_x} f_r(x, \Phi \to \theta) L(x \leftarrow \Phi) cos(N_x, \Phi) d\omega_\Phi \quad (1.2)$$

The left hand side $L(x \to \theta)$ corresponds to the radiance leaving at a point x in the direction θ. The right hand side consists of an emittance part and a reflectance part. $L_e(x \to \theta)$ corresponds to outgoing radiance emitted from x, and the integral term corresponds to reflected radiance received from all directions over a hemisphere over the point x. In [1] it is shown that

Equation 1.2 can be represented by separating the terms for indirect and direct illumination as in Equation 2.9.

$$L(x \to \theta) = L_{direct} + L_{indirect} \tag{2.9a}$$

$$L_{direct}(x \to \theta) = \int_A L_e(y \to \mathbf{yx}) f_r(x, \mathbf{xy} \to \theta) G(x,y) V(x,y) dA_y \tag{2.9b}$$

$$L_{indirect}(x \to \theta) = \int_{\Omega_x} L_r(x \leftarrow \Phi) f_r(x, \Phi \to \theta) cos(N_x, \Phi) d\omega_\Phi \tag{2.9c}$$

2.5.1 Direct illumination

The term $L_{direct}(x \to \theta)$ expresses the contribution to a point x from light sources directly. Since the term $L_{direct}(x \to \theta)$ contains the emittance term L_e it is only non-zero for light sources, we can describe the direct illumination contribution as in Equation 2.9b, which is an integral over the area of the light sources in the scene instead of an integral over the hemisphere. Or as a discrete representation as a sum over all N light sources in the scene. In our implementation only a single planar light source is considered. For generality's sake all equations presented will be defined to handle an arbitrary number of light sources.

$$L_{direct}(x \to \theta) = \sum_{k=1}^{N} \int_{A_k} L_e(y \to \mathbf{yx}) f_r(x, \mathbf{xy} \leftrightarrow \theta) G(x,y) V(x,y) dA_y \tag{2.10}$$

In Equation 2.9b, A refers to the area of a light source in the scene. For every intersection point retrieved from the scene, that point's direct illumination is calculated in two separate ways, one for direct diffuse illumination and one for direct specular illumination. For diffuse direct illumination the light source is sampled. Each sample point y_i is distributed uniformly on the surface of the light source. The visibility function $V(x,y)$ makes sure that only sample points visible from x will contribute to the direct illumination by checking if y_i is in line of sight of x. Because of this these rays are often called shadow rays, and will result in soft shadows if enough samples y_i are taken. For a single planar area light source sampled with N_s shadow rays the Monte Carlo estimator becomes

$$\langle L_{direct}(x \to \theta) \rangle = \frac{1}{N_s} \sum_{i=1}^{N_s} \frac{L_e(y_i \to \mathbf{y_i x}) f_r(x, \theta \leftrightarrow \mathbf{xy_i}) G(x, y_i) V(x, y_i)}{p(y_i)} \tag{2.11}$$

as described in [1].

A uniform probability distribution function $p(y_i)$ of $\frac{1}{A}$ is used where A is the area of the light source. From Equation 2.11 we see that it is required to evaluate visibility, BRDF and radiance emitted for each sample point.

Direct specular illumination is handled more straight forward. If a ray intersects with a specular object, a new ray is traced from the intersection point in the direction of a perfect reflection of the first incoming ray. If this new ray intersects a light source the point at the initial intersection receives radiance contribution in the form of emittance from that light source.

2.5.2 Indirect illumintaion

Indirect illumination corresponds to radiance received through one or more reflections against other objects in the scene. The contribution from indirect illumination for a point x is done by evaluating the integral over the hemisphere in Equation 2.9c. Unfortunately we cannot transform this expression into an integral over a smaller domain as we did for direct illumination since the reflectance term L_r, which in turn contains radiance from other points in the scene, appears on both sides of the equation. The recursive nature of this algorithm is evident and it is clear that an exact solution is cannot be found. We instead use a Monte Carlo estimator as presented in [1].

$$\langle L_{indirect}(x \to \theta) \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{L_r(r(x, \Phi_i) \to -\Phi_i) f_r(x, \theta \leftrightarrow \Phi_i) cos(\Phi_i, N_x)}{p(\Phi_i)} \tag{2.12}$$

In Equation 2.12 N is the number of rays used to sample indirect radiance, L_r is the reflected radiance, Φ_i is the hemisphere direction of the i:th sampled ray, f_r is the BRDF of the surface material of x, N_x is the surface normal, and finally $p(\Phi_i)$ is the probability distribution function. Specular indirect illumination och diffuse indirect illumination are handled separately.

Diffuse indirect illumination

In this project all diffuse surfaces are of Lambertian type. Since a Lambertian diffuse surface per definition reflect equal amounts of radiance in every direction, the BRDF and the pdf for such a surface can both be treated as constants. We define the BRDF and the pdf for all diffuse surfaces as in

Equation 2.13.

$$BRDF = \frac{color}{\pi} \tag{2.13a}$$

$$p(\Phi_i) = \frac{cos(\Phi_i, N_x)}{\pi} \tag{2.13b}$$

Inserting Equation 2.13 into Equation 2.12 and reducing and realising that constants can be placed outside the sum produces Equation 2.14.

$$\begin{aligned}
&\frac{1}{N}\sum_{i=1}^{N}\frac{L_r(r(x,\Phi_i)\rightarrow -\Phi_i)f_r(x,\theta\leftrightarrow\Phi_i)cos(\Phi_i,N_x)}{p(\Phi_i)}\rightarrow \\
&\frac{1}{N}\sum_{i=1}^{N}\frac{L_r(r(x,\Phi_i)\rightarrow -\Phi_i)\frac{color}{\pi}cos(\Phi_i,N_x)}{\frac{cos(\Phi_i,N_x)}{\pi}}\rightarrow \\
&\frac{color}{N}\sum_{i=1}^{N}L_r(r(x,\Phi_i)\rightarrow -\Phi_i)
\end{aligned} \tag{2.14a}$$

The random hemisphere sample direction Φ_i is created by randomly picking two angles θ and ϕ, which will represent a sampling direction in spherical coordinates. The angle ϕ is assigned a uniform random value in the interval $[0, 2\pi]$ and θ is assigned a value of $cos^{-1}(r^{\frac{1}{1+n}})$ where r is a uniform random number in the interval $[0, 1]$ and n is a number in the interval $[0, 1]$ which dictates the sample direction's alignment within the cosine lobe around N_x. Lastly, the sample direction is transformed to Cartesian coordinates and is rotated along N_x.

Specular indirect illumination

Perfect specular surfaces are implemented in this project.

The BRDF for a specular surface is different from 0 in only a single direction, the perfect reflection direction. The BRDF and pdf are defined in equation Equation 2.15 where θ is the incident ray's angle to the surface normal and Φ_i is the reflected ray angle. Since we are dealing with perfectly specular materials the pdf will be equal to 1 as well, and $\Phi_i = \theta$ thus the dirac function will yield a result of 1.

$$BRDF = \frac{color}{cos(\Phi_i, N_x)}\delta(\Phi_i - \theta) = \frac{color}{cos(\Phi_i, N_x)} \tag{2.15a}$$

$$pdf = 1 \tag{2.15b}$$

Inserting Equation 2.15 into Equation 2.12 yields

$$\frac{1}{N}\sum_{i=1}^{N}\frac{L_r(r(x,\Phi_i)\rightarrow-\Phi_i)f_r(x,\theta\leftrightarrow\Phi_i)cos(\Phi_i,N_x)}{p(\Phi_i)}\rightarrow$$

$$\frac{1}{N}\sum_{i=1}^{N}\frac{L_r(r(x,\Phi_i)\rightarrow-\Phi_i)\frac{color}{cos(\Phi_i,N_x)}cos(\Phi_i,N_x)}{1}\rightarrow$$

$$\frac{color}{N}\sum_{i=1}^{N}L_r(r(x,\Phi_i)\rightarrow-\Phi_i) \tag{2.16a}$$

which, if Φ_i does not intersect a light source, is a recursive call that lets the reflected ray continue through the scene.

An intersection with a specular surface can have two results. For a specular material that is perfectly opaque, for example a mirror-like material, a reflected ray will be created. If the reflected ray in turn does not intersect with a light source it will be recursively iterated and continue through the scene. If the material instead is not perfectly opaque two rays will be created. One reflected ray created in the same fashion as described above, and one refracted (transmitted) ray that will continue into the object. To decide how much of the radiance should be contributed from the reflected ray an approximation of Fresnel's equation as presented by Schlick et al. [6] in Equation 2.17 is used.

$$R_0 = \left(\frac{n_1-n_2}{n_1+n_2}\right)^2\rightarrow \tag{2.17a}$$

$$R(\theta) = R_0 + (1-R_0)(1-cos(\theta))^5 \tag{2.17b}$$

R_0 is the reflection coefficient for light incoming parallel to the normal direction of the surface, $n1$ and $n2$ are the refractive indices for the mediums the ray is come from and going into respectively. The term $cos(\theta)$ is the dot product between the halfway direction vector and the viewing direction vector. Equation 2.17 describes what fraction of the reflected ray should be used in computing the radiance for the intersection point. The radiance contribution from the refracted (transmitted) ray them becomes $T = 1 - R$.

R_0 is the reflection coefficient for light incoming parallel to the normal direction of the surface, $n1$ and $n2$ are the refractive indices for the mediums the ray is coming from and going into respectively. The term $cos(\theta)$ is the dot product between the halfway direction vector and the viewing direction

vector. Equation 2.17 describes what fraction of the reflected ray radiance should be used in computing the radiance for the intersection point. The radiance contribution from the refracted (transmitted) ray them becomes $T = 1 - R$.

2.5.3 Reflection

When a ray travels through a medium, colliding with another medium, the ray will be reflected. The law of reflection says that the angle of incidence θ_i and the angle of reflection θ_r is equal to each other for a perfect reflection. The incoming ray can be divided into two different parts, the tangent part to the surface $\mathbf{i}_\perp$ and the normal part $\mathbf{i}_\parallel$. The tangential part can be found using orthogonal projection onto the normal using Equation 2.18. The normal part is found using Equation 2.19.

$$\mathbf{i}_\perp = \frac{\mathbf{i} \cdot \mathbf{n}}{|\mathbf{n}^2} \mathbf{n} = (\mathbf{v} \cdot \mathbf{n})\mathbf{n} \tag{2.18}$$

$$\mathbf{i}_\parallel = \mathbf{i} - \mathbf{i}_\perp \tag{2.19}$$

By doing a dot product between these two rays, it is possible to prove that the result is zero and thus proving that these two are orthogonal and that $\mathbf{i}_\perp$ is an orthogonal projection on $\mathbf{n}$. By combining Equation 2.19 and Equation 2.18 the reflected ray $\mathbf{r}$ can be calculated using Equation 2.20.

$$\begin{aligned} r =& \mathbf{i}_\parallel - \mathbf{i}_\perp && \text{(2.20a)} \\ =& [\mathbf{i} - (\mathbf{i} \cdot \mathbf{n})\mathbf{n}] - (\mathbf{i} \cdot \mathbf{n})\mathbf{n} && \text{(2.20b)} \\ =& \mathbf{i} - 2(\mathbf{i} \cdot \mathbf{n})\mathbf{n} && \text{(2.20c)} \end{aligned}$$

2.5.4 Refraction

When calculating refractions, Snell's law are used. It states that the product of the refractive indices and sines of the angles must be equal, as stated in Equation 2.21.

$$\eta_1 sin\theta_i = \eta_2 sin\theta_t \tag{2.21}$$

This equation has a problem, when $sin\theta_1 > \frac{\eta_1}{\eta_2}$ the other term $sin\theta_2$ has to be greater then 1, which of course is impossible. This case is what is called

total internal reflection. When calculating the refracted ray t, continued calculation assumes that the condition is not satisfied.

The refracted vector is also divided into two parts, just the same as in Equation 2.19 and the norms are calculated first. By taking advantage of the fact that norms and tangents to the plane is equal to the sines in Equation 2.21 it can be rewritten into Equation 2.22.

$$\mathbf{t}_{\parallel} = \frac{\eta_1}{\eta_2}\mathbf{i}_{\parallel} = \frac{\eta_1}{\eta_2}[\mathbf{i} + cos\theta_i\mathbf{n}] \tag{2.22}$$

The tangential calculation uses Pythagoras which gives Equation 2.23. The combination of these two, described in Equation 2.24 is all that is needed to calculate the refraction vector

$$\mathbf{t}_{\perp} = -\sqrt{1 - |\mathbf{t}_{\parallel}|^2}\mathbf{n} \tag{2.23}$$

$$\mathbf{t} = \frac{\eta_1}{\eta_2}\mathbf{i} + (\frac{\eta_1}{\eta_2}cos\theta_i - \sqrt{1 - |\mathbf{t}_{\parallel}|^2})\mathbf{n} \tag{2.24}$$

2.6 Multi-threading

Multi-threading was implemented using C++11 threads. The application spawns a number of threads n equal to the number of CPU cores in the computer, each is responsible for a part of the computation. The image is divided so that each thread is responsible for every n:th column in the image which counteracts work load issues where parts of an image might be harder to compute then others. To support rendering while calculating, the threads are joined after each row and a new image is rendered. This gives a small overhead of spawning and joining threads but the benefit of seeing progress in real time is high, and the overhead is only noticeable when dealing with a small computation time per row which is not common when doing ray tracing computations.

3 — Results and benchmarks

3.1 Shadow rays

The quality of the direct diffuse illumination is directly related to the number of shadow rays used to sample the light source from each intersection point. Figure 3.1 shows images with different number of shadow rays per sample. Notice how the amount of noise on the floor decreases as the number of shadow rays increases. Table 3.1 shows the effect the number of shadow rays has on the rendering time.

Shadow rays	Time
1	6s
10	20s
20	34s
80	126s

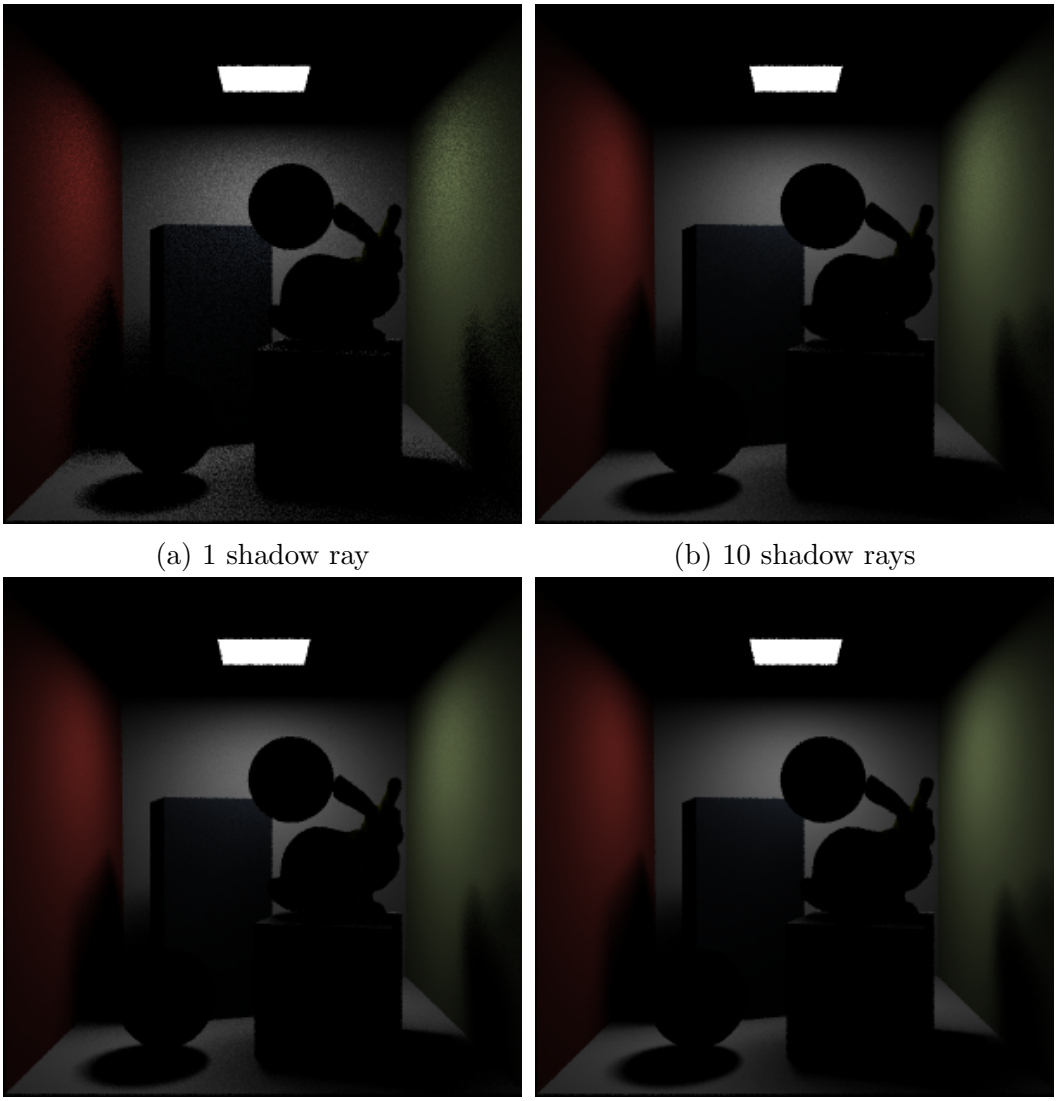
Table 3.1: Rendering time for shadow rays.

3.2 Octree

The octree allows for a great speedup when using a scene with a lot of primitives. As can be seen in Table 3.2, even with a mesh with relatively low polygon count the computation time is decreased by a factor of 10. The same table also shows the problem with the octree structure. It does require a lot of extra collision tests and when the scene is simple, time is lost instead of gained.

Bunny	Octree	Time
yes	yes	37.8s
yes	no	6m 16s
no	no	6.6s
no	yes	16.9s

Table 3.2: Rendering time with and without octree data structure in scene. Rendered with 10 rays per pixel. Bunny consists of 4968 triangles.



(a) 1 shadow ray

(b) 10 shadow rays

(c) 20 shadow rays

(d) 80 shadow rays

Figure 3.1: Number of shadow rays increasing from left to right, top to bottom.

3.3 Threads

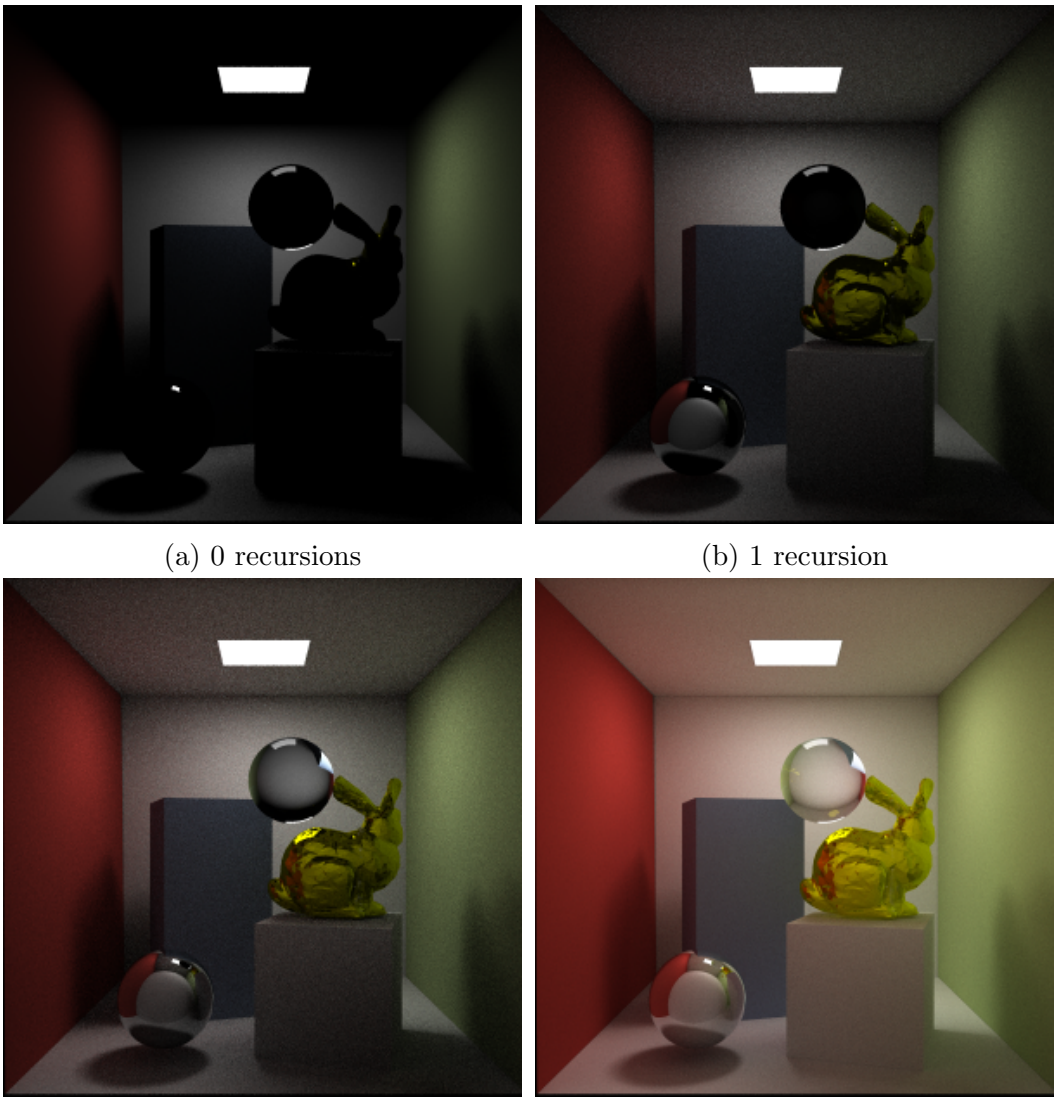
The number of threads used in the solution is dependent on how many cores the CPU of the rendering computer has. By allowing several threads to divide the work, the algorithm is allowed to compute several parts of the image in parallel. As can be seen in Table 3.3 a speedup of almost 70 percent is gained when using four threads, compared to using only one.

Threads	Time	% decrease
1	116s	0
2	56.5s	51
3	50.4s	57
4	37.1s	68

Table 3.3: Rendering time using different number of threads also showing percentage decrease when adding more threads. Rendered with 10 rays per pixel, with only walls and bunny.

3.4 Recursion depth

Recursion depth refers to the number of recursive calls allowed in the algorithm, e.g. how many bounces a ray will make in the scene before terminating and returning. The recursion depth affects the noise in the final image, but it also has an impact on the resulting reflections and refractions for specular objects. As seen in Figure 3.2 a recursion depth of 0 results in only direct illumination present. Notice how all specular objects are void of any indirect illumination since only once bounce per ray is allowed. A recursion depth of 1 yields slightly better results. We now see objects that are opaque and specular since rays are allowed to bounce once before termination. Notice that the glass sphere is still void of indirect illumination since a refraction occurrence involves at least 2 recursive iterations. A depth of 2 recursions allows for a single refraction instance and the glass sphere is now better represented. Lastly a render depth of 6 is shown in Figure 3.2d. This is the recursion depth used for most of the images in this report. Table 3.4 show the effect recursion depth has on rendering time.



(a) 0 recursions

(b) 1 recursion

(c) 2 recursions

(d) 6 recursions

Figure 3.2: Increasing recursion depth from left to right, top to bottom.

Recursion depth	Time
0	11s
1	17s
2	24s
6	55s

Table 3.4: Rendering time for different recursion depths.

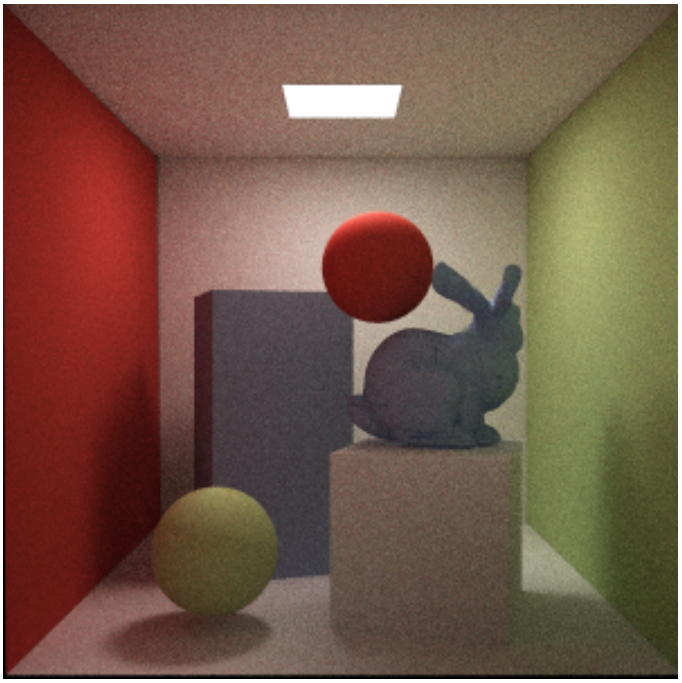
3.5 Noise introduced by specular materials

Specular materials that produce reflected and to some extent refracted rays is and additional source for noise in the final image and require more samples to reach the same quality level as a scene with only diffuse samples.

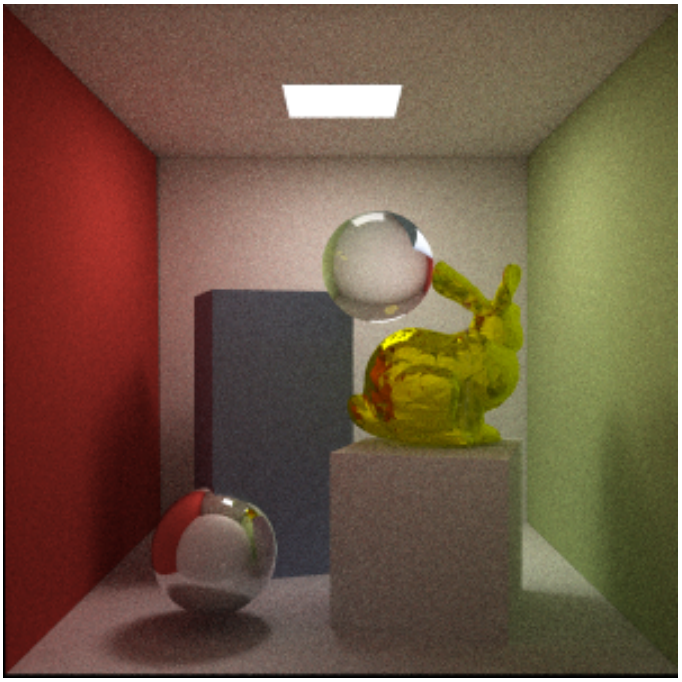
Notice the increased noise on the surface of the back wall and on the box nearest to the camera in Figure 3.3.

3.6 Samples per pixel

Another way of reducing the noise in the images is by sending several rays through the same pixel from the camera. Instead of sending a single ray through the center of the pixel we send multiple rays placed in the pixel in a uniform random fashion. The contributions from each view ray are summed up and normalized to form the final intensity value for the pixel. This technique is sometimes referred to as sub-pixel sampling. In Figure 3.4 we see a scene rendered with an increasing amount of view rays per pixel from left to right, top to bottom. Notice how the noise seen on the back wall in Figure 3.4a subsides rather quickly, while the noise in areas not directly visible to the light source persists much longer even though the number of samples increase.

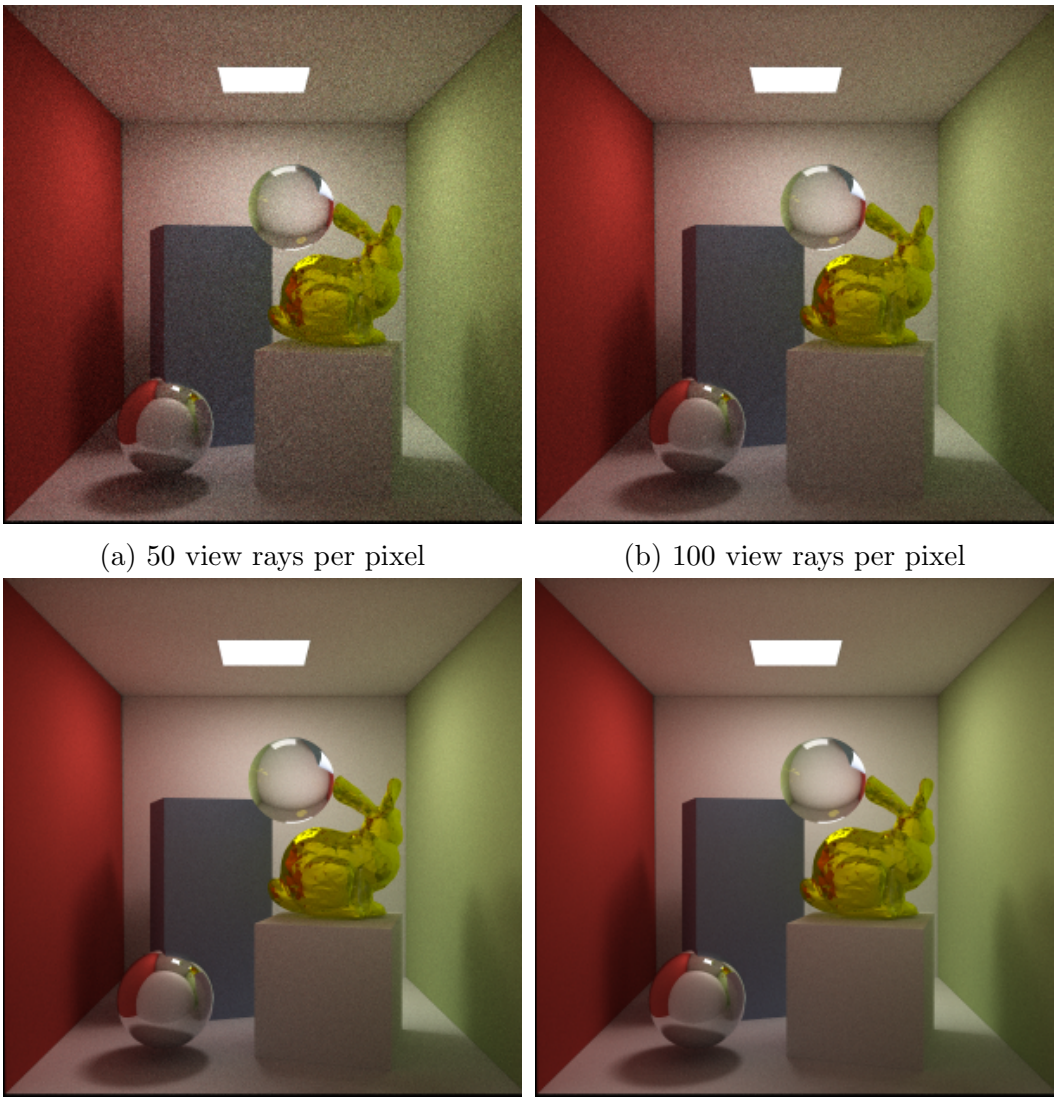


(a) Diffuse materials only



(b) Specular and diffuse materials

Figure 3.3: Scene rendered with the same resolution, recursion depth, shadow rays, and a low number of samples per pixel to illustrate the increased noise when specular objects are used in a scene.



(a) 50 view rays per pixel

(b) 100 view rays per pixel

(c) 600 view rays per pixel

(d) 1000 view rays per pixel

Figure 3.4: Reducing noise by increasing the number of view rays.

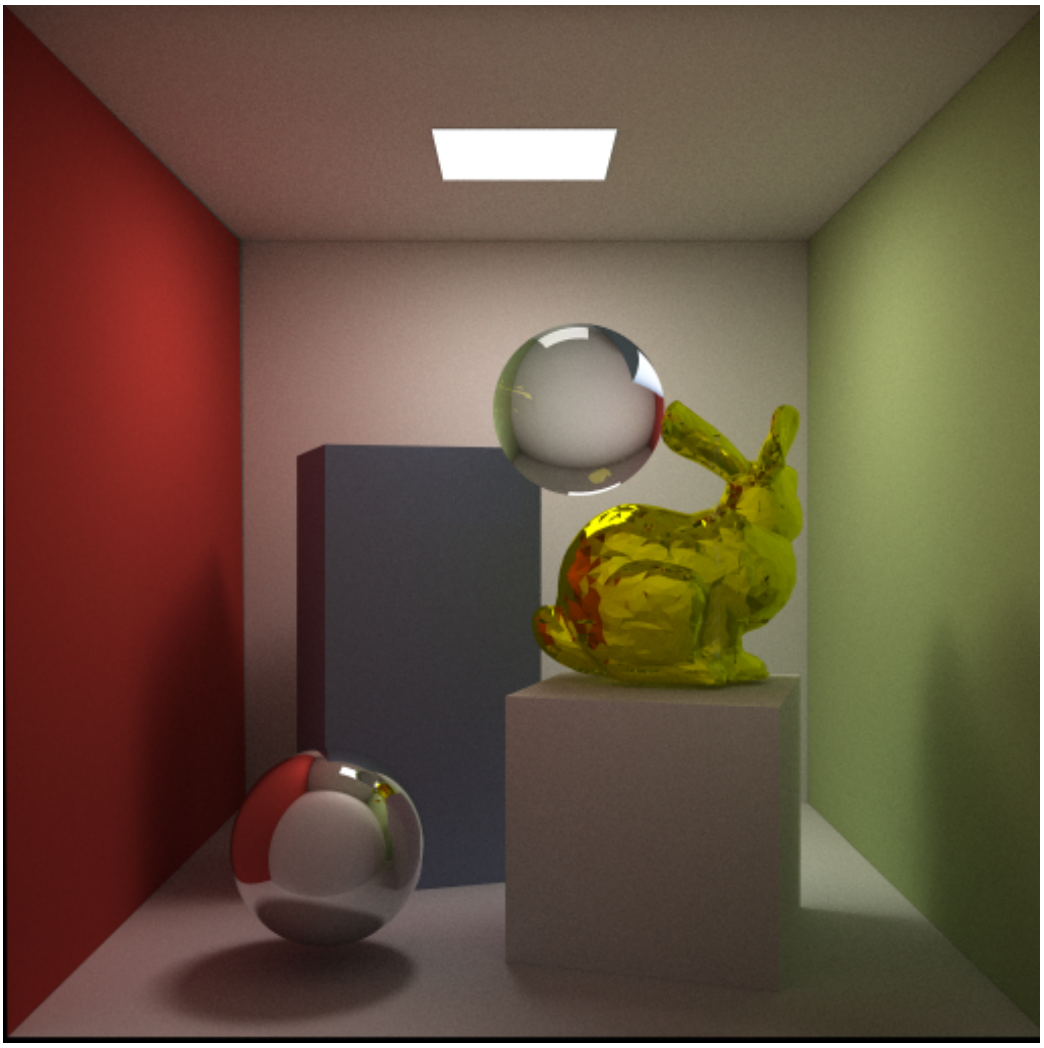


Figure 3.5: Final render of Monte Carlo ray tracer.

4 — Discussion

The beginning of the project consisted of lots of reading and researching different methods used in a ray tracer. Initially a simple ray caster and later a Whitted ray tracer were implemented to allow us to familiarize ourselves with scene set-up, equations for path tracing, reflections, ray versus primitive intersections and so on. Moving on to a Monte Carlo ray tracer introduced a number of new variables to take into consideration, as well as other challenges compared to a Whitted ray tracer. We implemented multi-threading early on in the project and we are happy with doing so. Ray tracing suits parallel computing very well since every pixel is solved completely independant on the calculations for other pixels, and we achieved an almost linear speedup when running the rendering on a multicore CPU. One additional improvement that could be done would be to implement a cluster where several computers can work on different parts of the same image.

By implementing a ray tracer based on the Monte Carlo method it is possible to create high quality renders of the scene, given that the solution is allowed to converge enough. But even with a lot of iterations and samples the final output result still suffers from artifacts like noise. To remedy this, photon mapping could be implemented which would decrease the time it takes for the solution to converge.

The created ray tracer only handles air to glass and glass to air refractions, not between any two mediums. Implementing this would allow us to create glass objects that are placed on other surfaces and not only floating in the air. To extend our solution to support this would not be too hard, because we would still be dealing with the same two cases, ray travelling to a medium with lower refractive index or ray travelling into a medium of higher refractive index.

The implementation of the Octree data structure allowed for a large decrease in computation time for scenes with a high number of primitives. For life-like scenes that consists of an even higher polygon count, a technique like this is crucial. We have also shown that this structure implies an overhead for scenes that are very simple. A method for deciding if an Octree would be beneficial could be implemented where the decision of collision method is decided depending on the complexity of the scene. But because the speed gained from complex scenes is far greater than the time lost when dealing with complex scenes, this is still acceptable for us.

When computing direct diffuse illumination, smarter methods for sampling the light source could be employed. Using a pdf of $1/A$ e.g uniform

sampling of the light source area as in this report is not ideal. This approach means for example that points far away from the light source uses the same amount of shadow rays as points close to the light source. Importance sampling could instead be used to use more shadow rays for points close the light source, as vice verse for points far away.

Bibliography

[1] P. Dutré, K. Bala, and P. Bekaert. *Advanced global illumination / Philip Dutr, Kavita Bala, Philippe Bekaert.* Wellesley, Mass. : AK Peters, cop. 2006, 2006.

[2] J. D. Foley and T. Whitted. An improved illumination model for shaded display.

[3] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[4] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986.

[5] A. Lagae and P. Dutré. An efficient ray-quadrilateral intersection test. *Journal of Graphics, GPU, and Game Tools*, 10(4):23–32, 2005.

[6] C. Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, pages 233–246, 1994.

[7] B. E. Smits, J. R. Arvo, and D. H. Salesin. An importance-driven radiosity algorithm. *SIGGRAPH Comput. Graph.*, 26(2):273–282, July 1992.