Introduction

Global illumination

Global illumination algorithms are a generic name describing any algorithm trying to accomplish a more realistic lighting in a scene than a local illumination model would. The main difference between a global illumination algorithm and a local illumination algorithm is; a local illumination model would take into account only the light arriving at a surface point directly from a light source. A global model will in most cases include a local lighting model and also take into account light which has been reflected or refracted a number of times before arriving at the surface point.
Since the light which has been reflected or refracted before encountering a surface is not a direct contribution from the light source it is often called *"Indirect Light"* in literature.

One can make the distinction between two separate models for global illumination; the view-*dependent* and the view-*independent*. The methods are used to calculate different components of the indirect light arriving at a surface point;
View-*dependent* methods are most commonly used to calculate directional dependent reflections and refractions such as refractions through glass or reflections in a mirror.
View-*independent* methods are used to calculate the directional independent indirect light arriving at a diffuse surface after a number of reflections or refractions from other diffuse surfaces.
Each of the two models excels at computing their own contribution to the full global illumination and they are preferably used together in what is usually called a "*Two Pass Solution"*.
The main difference between the two models is that a view-*independent* illumination model, taking into account directional independent indirect light, is not dependent on where the camera or "*eye"* is placed in the scene and does not need to be re-computed when the viewport is changed.
View-*dependent* illumination models on the other hand, taking into account the directional dependent indirect light, are entirely dependent on where the camera is placed in the scene and will need to be re-computed as the viewport changes.

There are two common methods used for the computation of the view-*independent* part of the indirect light; Photon Mapping and Radiosity.

Photon mapping

Photon mapping was presented by Jensen in the article "Global Illumination using Photon Maps" published in 1996 [1].
It is a two pass solution to global illumination which takes into account caustic effects and diffuse reflections between surfaces. It is a method which can easily simulate realistic effects such as caustics and color bleeding which are often lacking in simpler global illumination algorithms.
The first pass is the construction of two separate photon maps; the first map is a *global* map representing photons which have been traced through a number of diffuse reflections before encountering a diffuse surface. The second map is a *caustic* map representing photons which have been refracted or reflected before encountering a diffuse surface. The second pass uses the two photon maps as a lookup table in the estimation of indirect light at a surface point.
Typically the resolution of the caustic map is much higher than that of the global map. This

separation of caustic effects from the global photon map enables caustic effects to be rendered at a very high resolution without spending an enormous amount of photons in the global photon map.

Radiosity

The Radiosity method was presented the article “Modeling the Interaction of Light Between Diffuse Surfaces” by Goral et al in 1984 [2].
Radiosity is a method which has its basis in thermal dynamics, more specific heat transfer, and has been adapted for use in computer graphics.
It can simulate finite area light sources and the diffuse reflections between surfaces in a scene and thus produce many of the realistic effects lacking in simpler methods such as color bleeding and soft shadows.
The main idea in Radiosity is to divide a scene into a finite number of patches and for each patch calculate the total Radiosity arriving at that patch from all other patches in the scene, plus the patches self-illumination. This is an iterative algorithm which initially only has self-illumination on the patches corresponding to the light source. Each iteration will increase the number of patches which emit radiance to all other patches, depending on the relative reflectivity of the patch and the *form factor* between any pair of patches. Thus the algorithm converges to a true solution as the number of iterations increase.

When it comes to computing the view-dependent component of the global illumination two quite common methods will be discussed; *Whitted Ray Tracing* and *Monte-Carlo Ray Tracing* (*Distributed Ray Tracing*).
Note: The ray tracing methods mentioned below are sometimes named *backwards ray tracing algorithms* in literature. Simply because they do not in fact trace the proper path that light propagates in a scene (from the light source, through the scene and into the eye) but rather in the opposite direction (from the eye and into the scene).

Whitted Ray Tracing

Whitted ray tracing, as proposed in an article by Turner Whitted in 1979 [3], is a method for computing global illumination in a scene by means of tracing a ray of light from the viewport and into the scene.
A classical Whitted-style ray tracer will only consider perfect reflections/refractions and direct lighting from the light source. The maximum propagation length of rays into the scene is pre-determined by a *ray depth* variable which will stop the recursion once a specific amount of rays have been spawned.
To be able to handle surfaces which are shaded by other surfaces in the scene a *shadow ray* will be spawned, directed towards the light source, as a ray hits a diffuse surface. The shadow ray will determine visibility towards the light source and eventually the direct light contribution from the light source.

As a ray is traced into the scene the following cases can occur;

- The ray hits a reflective surface and a new ray is spawned in the perfect reflection direction [4], the *ray depth* is incremented by one and the tracing of the new ray begins.

- The ray hits a refractive surface and two new rays are spawned, one refracted ray in the perfect refraction direction [4] and one in the perfect reflection direction (except for the critical angle case wherein only a reflected ray is spawned). The rays are weighted according to the *Fresnel equations* [5] and traced, the ray depth is incremented by one.
- The ray hits a diffuse object and is absorbed, a shadow ray is sent to the light source to determine visibility and the resulting radiance is returned. No new ray is spawned and recursion stops.
- The maximum ray depth has been reached, no new ray is spawned and recursion is stopped.

Because of the inherent limitations of only dealing with perfect reflection and refraction the Whitted-style ray tracer will fail to simulate many of the more realistic properties of light propagation such as; diffuse reflections, e.g. reflections between diffuse surfaces and glossy reflectors, caustic effects and soft shadows.

Monte Carlo Ray Tracing (Distributed Ray Tracing)

An improvement to the Whitted style ray tracer was suggested by Cook et al in the article "Distributed Ray Tracing", published in 1984 [6]. The proposed method would distribute rays over several dimensions and thus enabling the simulation of effects such as glossy reflections and refractions, motion blur and depth of field. Furthermore Kajiya applied ray tracing to the rendering equation in his paper "The Rendering Equation", published in 1986 [7]. The technique Kajiya proposed allowed for full global illumination e.g. any possible type of interreflections between surfaces in a scene.
The term Monte Carlo ray tracing or Distributed Ray Tracing as it is sometimes referred to actually mean that *Monte Carlo Integration Techniques* [8] are applied in order to solve the rendering equation.

As opposed to a Whitted-style ray tracer a the Monte Carlo approach has several advantages; it is able to simulate diffuse interreflections between surfaces, glossy reflectors, caustics (although this is unusual and inefficient in a pure Monte Carlo ray tracer), area light sources and color bleeding. The Monte Carlo approach will *randomly* distribute rays in any direction upon a ray-surface intersection and the recursion does not stop after a predetermined ray depth but rather *Russian Roulette* [9] is used as a stopping condition. To be able to handle shadows and especially *soft shadows* caused by surfaces occluding other surfaces in the scene, shadow rays will be randomly distributed over the light source area to determine visibility. The resulting direct contribution from the light source will be averaged with respect to the number of shadow rays.
Monte Carlo ray tracing works much like a Whitted-style ray tracer with the exception of the stopping condition, the directions of the spawned rays and the number of spawned rays.
Monte Carlo ray tracing will, if given the time, produce very realistic pictures and will be able to deal with any form of surface-light interaction. The drawbacks of the algorithm are the long rendering times and the fact that it produces very noisy pictures due to its random nature. There are several well-known methods for reducing the noise in the picture and the rendering time, they will be addressed later in the report.

Structure of the report

In section one a brief description to Global Illumination is given and the history and techniques of; Photon Mapping, Radiosity, Whitted Ray Tracing and Monte Carlo Ray Tracing are described in short.
Section two describes the techniques regarding Monte Carlo Ray Tracing and Photon Mapping in detail and the basic ray-surface intersection algorithms are introduced.
Section three presents the results and benchmarks obtained from the method discussed in the report and section 4 contains discussion about the results and possible future work.

Background

The first paragraphs of this section will introduce the techniques and concepts needed to perform ray tracing algorithms. Once these concepts have been introduced a detailed description of the rendering algorithm implemented, both pure Monte Carlo Ray tracing and photon mapping will be given. A Whitted-style ray tracer has also been implemented for the purpose of comparing results, since this type of ray tracer is not the focus of the report no detailed description of its implementation will be given.

Ray-surface intersection

The two types of objects the renderer presented in this report deal with are; *implicit Spheres* and *Polygonal Objects*. The methods used to determine whether an intersection has occurred between a ray and an object are presented and explain below. For the algorithms listed below a ray is defined as a starting point and a direction e.g. a line of infinite length.

Implicit Spheres

The first step in determining if a ray has intersected a sphere is to define a set of variables which will be used for further calculations; a vector ***V*** with its origin in the starting position of the ray, pointing towards the center of the sphere, the length ***L*** of the vector ***V***, the dot-product ***D*** between the vector ***V*** and the direction of the ray and the sphere radius; ***R.***

If the dot-product ***D*** is smaller than zero it means that the ray direction and the vector ***V*** points in opposite direction thus the only possible intersection is if the length ***L*** is smaller than the radius ***R*** which means the ray origin is within the sphere as seen in figure 1.

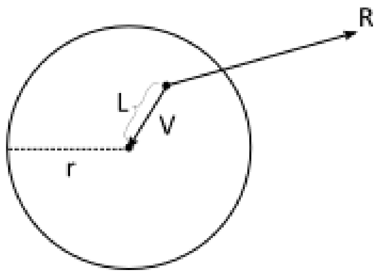

Figure 1 – Ray origin inside sphere

Define a new variable $M = L - D$. Since the ray direction is normalized ***D*** actually represents the magnitude of the projection of ***V*** onto the ray direction. If ***M*** is larger than the radius ***R*** it means the ray cannot hit the sphere, else the ray will hit the sphere and it's a matter of finding the intersection point(s). The variables ***V***, ***M***, ***R*** and ***D*** are displayed as their geometric representation in figure (2).

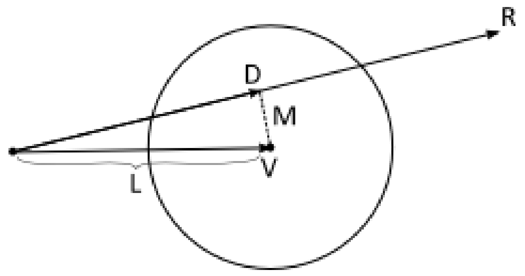

Figure 2 - Geometric representation of the variables

Define a variable $Q = \sqrt{R - M}$ and a variable ***T***, initially set to zero. If the ray origin is outside of the sphere set $T = D - Q$, else set $T = D + Q$. The intersection point is now given by; $Ray_{origin} + Ray_{direction} * T$.

Polygonal Objects

To determine whether a ray has intersected a polygon or not the first step is to define a set of vectors which span the plane created by the polygon and check for an intersection between the ray and the plane which the polygon spans. The polygon is identified by a set of three points; P_0, P_1, P_2. Two vectors which span the plane of the polygon can be identified as $u = P_1 - P_0$ and $v = P_2 - P_0$. With the set of points and vectors it is now possible to define the *parametric equation* of the plane which the polygon spans. Any point on this plane must satisfy; $P(s,t) = P_0 + s * u + t * v$

Determining whether a ray has intersected this plane is simply a matter of examining the position and direction of the ray; if the ray origin is above the plane and the origin points downwards towards the plane or if the ray origin is below the plane and points upwards towards the plane the ray will hit. This is achieved by creating a vector from the ray origin to one of the corner points of the polygon and examining the sign of the dot-product between that vector and the polygon normal. There is one special case when the ray direction is parallel to the plane; in that case we consider it a miss even though the ray origin may lie on the plane.

To determine the intersection point, define a vector ***V*** between the ray origin and one of the corner points of the polygon. The dot-product of ***V*** and the polygon normal, defined as ***a***. ***a*** will be the magnitude of the projection of ***V*** onto the normal. Define a variable ***b*** which equals the dot-product of the ray direction and the normal, it will represent the magnitude of the projection of the ray direction onto the normal.

The intersection point of the ray and the plane will now be given as $P = Ray_{start} + Ray_{direction} * \frac{a}{b}$. If given an intersection point $P = P(s,t)$ which lay in the plane, it can be determined whether ***P*** lies in the polygon by examining the parameters ***s*** and ***t***, if $s \leq 0, t \leq 0, s + t \leq \boldsymbol{1}$ the point ***P*** lies in the polygon.

To simplify the forthcoming equations which will determine the intersection point ***P*** between a ray and a polygon, define the "*generalized perp operator*" [10].

With the help of the perp operator the plane's parametric equation can be solved for the intersect point ***P***, define a vector $W = P - P_0 = s * u + t * v$ which will lie in the plane. Then solve the equation for ***s*** and ***t***; apply the perp operator of ***v*** to the equation and solve for ***s*** and apply the perp operator of ***u*** and solve for ***t***. This yields the following equations;

$$s = \frac{w * (n\,x\,v)}{u * (n\,x\,v)}$$
$$t = \frac{w * (n\,x\,u)}{v * (n\,x\,u)}$$

By taking advantage of the left associative property of cross-product the cross-products can be rewritten as;

$$(n\,x\,u) = (u\,x\,v)x\,u = (u * u)v - (u * v)u$$
$$(n\,x\,v) = (u\,x\,v)x\,v = (u * v)v - (v * v)u$$

And finally the equations for ***s*** and ***t*** can be solved;

$$s = (u * v)(w * v) - \frac{(v * v)(w * u)}{(u * v)^2 - (u * u)(v * v)}$$
$$t = (u * v)(w * u) - \frac{(u * u)(w * v)}{(u * v)^2 - (u * u)(v * v)}$$

The final stages of simplification and the transformation from a cross-product to a dot product is done simply for efficiency purposes to speed up the algorithm.

View plane and view rays

The first step of the ray tracing process is to shoot rays from the camera, or "*eye*", into the scene. These rays will be shot into the scene through what is called a view plane. The view plane is what will become the resulting image once the ray tracing pass is done.
To construct the view plane the height and width in pixels of the resulting image need to be specified, the plane will lie at ***Z = -1***. To construct the image the ***X -*** field of view (FoV) need to be set, the ***Y*** – FoV is calculated from the aspect ratio as;

$$Y_{FoV} = aspect\ ratio * \ X_{FoV} = \frac{height}{width} * X_{FoV}$$

The FoV will greatly affect the resulting image since it determines how much of the scene is seen in the picture and how the objects in the scene appear. An appropriate value for the X_{FoV} when dealing with computer monitors is around 45 degrees ($\frac{\pi}{4}$). The last thing needed to launch a ray is to determine the coordinates in the plane which the ray will be launched through. In the simplest case where a ray will be sent through the middle of each pixel the coordinates of the plane can be calculated from the pixel image coordinates (***u,v***) as;

$$x = \frac{2 * u - width}{width} * \tan X_{FoV}$$
$$y = \frac{2 * v - height}{height} * \tan Y_{FoV}$$

A ray can now be constructed with ray origin at the eye and going in the direction ***(x,y,-1)***.
A visualization of the view plane is displayed in figure 3.

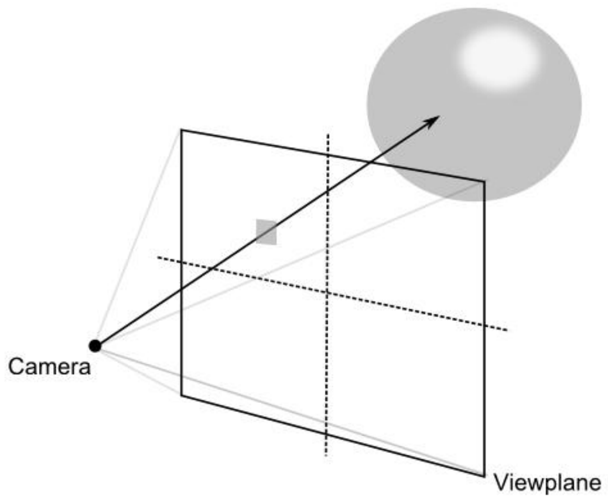

Figure 3 – A visualization of the viewplane

Jittered sampling

Aliasing artifacts occur when a single ray or too few rays are shot through one pixel. This results in jagged lines around objects in the image. By introducing randomness, noise, into the picture the jagged lines will be reduced but noise will be added to the final image. However our eyes are less sensitive to irregular artifacts like noise, than regular artifacts such as jagged lines and are therefore preferred over aliasing artifacts. Introducing random rays shot through the pixel, problems with clumping of pixels could arise and will add more noise and artifacts to the image. Many different ways exists to reduce the jagged lines and one way to reduce these effects is Jittered sampling, which is used in this paper. It subdivides the pixel into a regular grid of cells where in each cell a ray is randomly shot through, see figure 4. The result is an image with less or none aliasing artifacts. The regular grid reduces clumping of rays, but artifacts of regularity arise if they are not randomized inside.

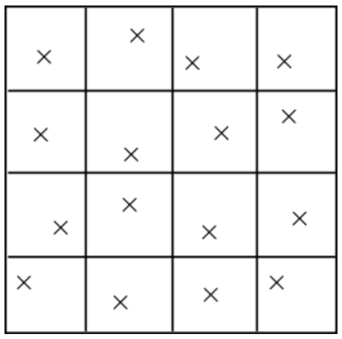

Figure 4 – Jittered sampling over the regular grid of a pixel.

Kd-tree and nearest neighbor search

The datastructure for holding the photon map data is called a Kd-tree (K Dimensional). A Kd-tree is a binary tree for fast lookup when the space coordinate data is known.
A Kd-tree subdivides it's given points in space, making it fast for lookups. The condition of each inner node will allow for a fast search to the correct subspace in which the point searched for exists. Construction of a Kd-tree is rather straight forward. Photons are emitted and stored in an allocated array.

When the emission pass is done, the Kd-tree recieves the array and sorts it corresponding to the photons position in space, first by x-coordinate, then by y and z and so on (Kd-trees can deal with any dimensionality).
When the array has been sorted, the array is split in the median of the sorted photons. The corresponding left half of the array gets sent to the left child of the root node, and the right half to the right child. And the procedure continues, cutting the array in half every step until a leaf node is reached.
This means that each child is a Kd-tree of its own. The array is sorted on x, y and z and then back to x continuing on for each depth. Each inner node corresponds to a photon on the split plane of each subspace (e.g. the root is on the splitting plane of the whole point set).
Time complexity of creating the tree depends on which sorting algorithm is used. $0(nlog2n)$ can be achieved with an $O(nlogn)$ search algorithm. In this case a regular shell sort with worst case complexity $O(n^2)$ is used and therefore the construction can take some time (in the range of seconds).

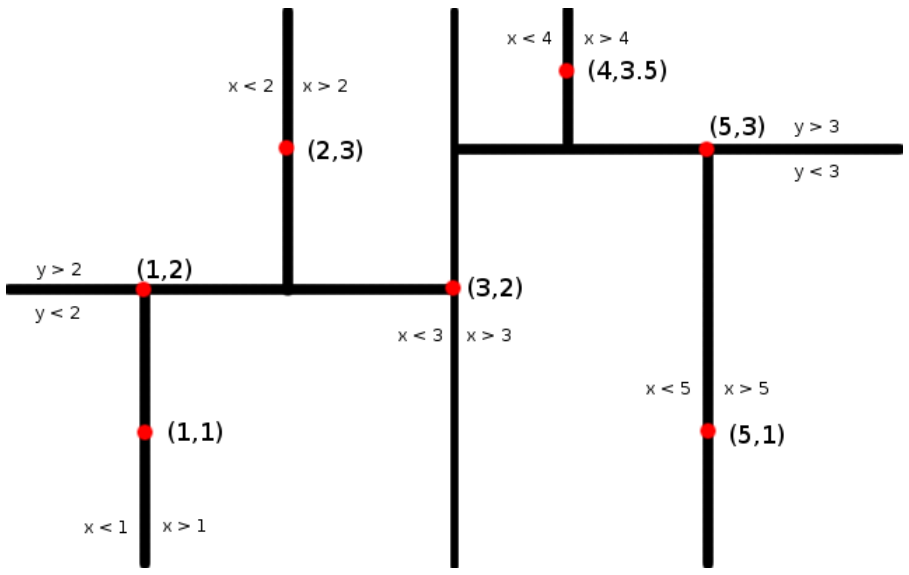

Figure 5 – Visualization of Kd-tree with splitting planes.

However, the most important part of using a Kd-tree is the nearest neighbor search which has an $O(logN)$ time complexity. This is important since the number of lookups can be in the range of billions during a render pass.
The lookup can be done in many ways considering if the photons are stored as inner nodes or as leaf nodes. By recursively traversing the tree by comparing the point in space to look around with the condition of each inner node (meaning if on x-depth and the point has x > split plane-x then search right, otherwise search left) the correct subspace of the point is easily located.
When the corresponding subspace has been found a candidate for nearest neighbor has also been found, namely, either one of the leaf nodes in the subspace. On the way back up the tree after finding correct subspace the distance to each splitting plane is compared to the best distance found so far.
If the distance to the splitting plane is smaller than the distance between the point and the found neighbor candidate, then the subspace of the splitting plane (neighbor subspace) has to be searched since it can hold better candidates for the neighbor search.

Soft shadows and shadow rays

To render shadows in a scene a ray called a "shadow ray" has to be sent from the point towards the light source. If the ray intersects with an object on the way, then the point is in shadow.
However, to achieve realistic soft shadows from a more diffuse light source (eg. area light source) additional computations has to be made. The quality of the shadows is determined by a predefined number of shadow rays.
When a ray in the ray tracing procedure hits a surface, a number of shadow rays are sent towards the area light source. While the start of the ray is fixed, the direction of the shadow rays is randomized over the area of the light source.
Each of the predefined number of shadow rays are checked for intersection and if intersection occurs, the ray did not reach the light source. This means that some of the rays hit the light source while others do not.
Only the rays that hits the light source contributes to the final color of the direct illumination component.
When a ray hits a light source, color gets added to the direct illumination component by multiplying the surface color with the light color and the scalar product of the outgoing ray direction and surface normal, and the reverse ray direction and light normal.
The product is then divided by the square length of the ray and added to the direct illumination component. This is then done iteratively for each shadow ray. Shadow ray calculations can be computationally heavy but is important for a realistic rendering.
However, by using photon mapping and shadow photons (as described in photon mapping) the shadow ray casts can be reduced to areas where there exist shadow photons. If an area does not have shadow photons near it means the area has a free path to the light source and intersection testing is not necessary.
When the procedure for rendering soft shadows is finished and the direct illumination component is calculated it has to be scaled by the number of shadow rays sent and the BRDF of the surface.

$$directComp = directComp * \frac{BRDF}{NUM\_SHADOW\_RAYS}$$

Bidirectional reflectance distribution function – BRDF

The bidirectional reflectance distribution function (*BRDF*) is a function which describes how a material reacts with light. More specifically it relates the differential radiance reflected in an exitant direction to the differential radiance incident through a differential solid angle. It is denoted as;

$f_r(x, \psi \leftrightarrow \Theta)$

The BRDFs used by the renderer for this report are; the *Lambertian BRDF* for diffuse surfaces and the *Modified Blinn-Phong* BRDF for specular surfaces. They will be described more in detail in the next section.

Materials

The definition of material properties for surfaces in the scene is what will determine the final look of an object. There are three types of materials used by the render; “*Diffuse*”, “Reflective” and ” Transmittive” they all react differently to light and emit light in different ways.

Diffuse materials reflect light equally in all directions on the hemisphere. That is, the reflected radiance is independent of exitant direction. Diffuse materials use the Lambertian BRDF;

$$f_r(x, \psi \leftrightarrow \Theta) = \frac{\rho_d}{\pi}$$

Where the ρ_d is the reflectance and represents the fraction of incident energy reflected at a surface.

Even though Reflective and Transmittive materials are not at all the same they are dealt with in much the same ways. Reflective/ Transmittive materials will have a preferred reflection/ refraction direction in which almost all of the light is reflected/ refracted. Reflective/ Transmittive materials use the Modified Blinn-Phong BRDF [11];

$$f_r(x, \psi \leftrightarrow \Theta) = k_s * (\theta_r * \theta_s)^{\,n}$$

Where θ_r is the outgoing direction of light and θ_s is the perfect specular direction with respect to the incoming direction of light.

The Rendering equation

The Rendering equation is an equation which for each point ***x*** and each direction ***Θ*** formulates the exitant radiance at that surface point and in that direction. It is the aim of global illumination algorithms to solve this equation. For the formulation used in this report it is assumed that light propagates instantaneously and that there is no participating media present in the scene.
In its *hemispherical* formulation, after some simplifications, the rendering equation is;

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_{\Omega_x} f_r(x, \psi \leftrightarrow \Theta)\, L(x \leftarrow \psi) \cos(N_x, \psi)\, d_{\omega\psi}$$

The rendering equation is because of its form an integral equation known as a *Fredholm equation.* This is because of that its unknown quantity, radiance, appears on both the left hand side of the equation and on the right hand side, integrated with a kernel.

Monte Carlo techniques

Monte Carlo techniques and especially Monte Carlo integration is an extensive field and will not be dealt with in detail in this report, it is however highly relevant to Monte Carlo ray tracing so a brief introduction will be given.
Monte Carlo techniques excel at computing approximate solutions to function which are not possible to analytically solve by hand, e.g. the rendering equation. The basic idea behind the solution to the rendering equation with the help of Monte Carlo integration is to construct a random variable such that it has an expectation value which equals the solution to the equation. Random samples would then be drawn of this variable and averaged to compute an estimate of its expected value. By this construction that estimate of the expected value would also be an estimate of the true solution to the rendering equation. When Monte Carlo integration is used as a solver to estimate the value of the rendering equation it is inevitable to introduce noise into the final picture.

Probability distribution functions (PDFs) and importance sampling

Since Monte Carlo integration by definition relies on random variables it is only natural that the probability distribution should affect the result. It is possible to engineer the probability that a

certain variable is generated by use of so-called *"inverse PDFs"*. The theory behind this is out of scope for this report but the results are used with great success.
For ray tracing purposes a *"wise"* choice of probability distribution is called importance sampling. It is a very effective way to reduce noise in the resulting picture by introducing specific PDFs engineered based on knowledge about the function to be evaluated. For example if the surface currently dealt with is a specular surface it is quite clear that most of the reflected light will be along the perfect reflection direction. Thus evaluations at direction which differ significantly from the perfect direction will not contribute much to the final result.
This is used in the renderer where PDFs can be tweaked to produce different visual results such as a wide range of reflections between totally diffuse and perfect reflection.

Russian Roulette

In order to not introduce bias into the picture an alternative to simple ray-depth termination must be implemented. Russian Roulette has been developed as a viable alternative which keeps the renderer unbiased while still keeping ray paths from becoming too long. It is in practice a very simple solution to ray termination; for any recursive evaluation of a function ***A***, pick a value $\alpha \in \{0,1\}$ and generate a uniform random number $r \in \{0,1\}$**.**If α is greater than ***r*** then $\boldsymbol{A} = \frac{\boldsymbol{A}}{\alpha}$, if α is less than ***r*** then $\boldsymbol{A} = \mathbf{0}$.
The number α should of course be chosen carefully and should preferably be related to material properties.

Modified Blinn-Phong specularity

To represent shiny material such as plastic and glass, a specular light is needed to simulate realistic material. The most popular way to simulate specular light is with Phong shading, but Phong shading has some physical limitations such as energy conservation, and cannot simulate material realistically [12]. A modified model of the Phong shading model is Blinn-Phong which generates a half-vector H that is the vector half-way between incoming light ψ and the view direction θ, see figure 7.
The half-vector H is calculated by adding the incoming light with the direction;

$$H = \psi + \frac{\theta}{|\theta+\psi|}$$

Though the Modified Blinn-Phong cannot render physical realistic material it is still easy to deal with and gives good result which makes it suitable for rendering. Just to mention a better model for simulating more accurate models is Cook-Torrance, but it will not be described or discussed in this paper.

Modified Blinn-Phong model

$$f_r(x, \psi \leftrightarrow \Theta) = k_s (N * H)^n + k_d$$

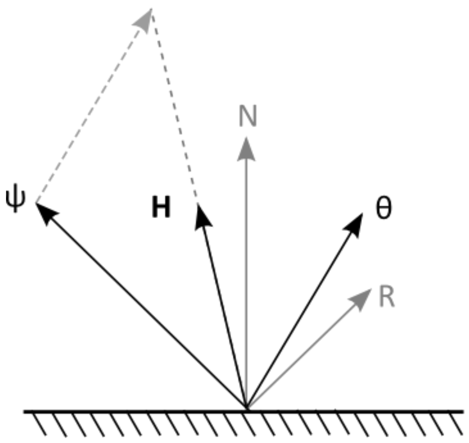

Figure 6 – The Half-Vector (H) for Blinn-Phong

Transforming normal to off-normal axis

The method used to construct rays which have a direction described by two angles relative to the normal of a surface is a two-step process. The first step is to generate the two angles $\theta \in \left\{0, \frac{\pi}{2}\right\}, \varphi \in \{0, 2\pi\}$ and the second step is to construct a direction which equals the normal of the surface transformed by these two angles.

The rotation in the transformation is achieved by utilizing *quaternions* [13].

In the first step of the transformation the perfect specular reflection/refraction direction is constructed by equation (1) [4] for reflection or equation (2) [4]for refraction. Figures 8 and 9 illustrate the two cases geometrically.

$$R = 2(N * L) - L \tag{1}$$

$$\sin\theta_r = \frac{n_1}{n_2}\sin\theta_i \tag{2}$$

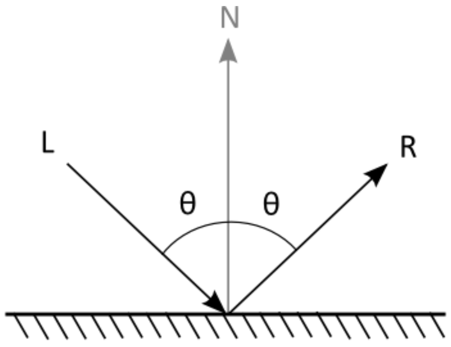

Figure 7– perfect reflection vector (R)

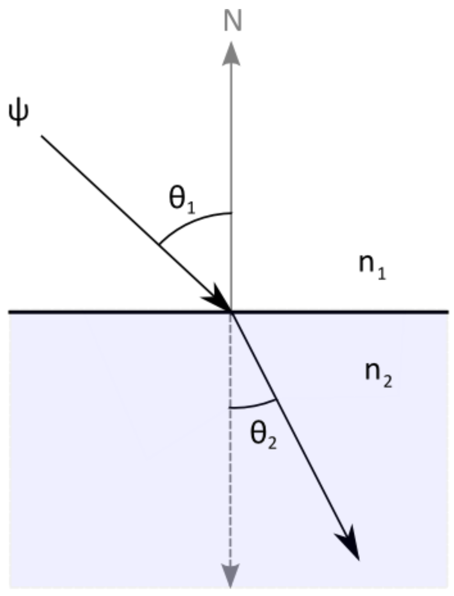

Figure 8 - Perfect refraction angle

The angle between the perfect refraction/reflection direction and the normal is computed by means of dot-product and denoted as θ_r.
In the next step of the transformation the normal is rotated by an angle of $\theta_r + \theta$ about an axis which is perpendicular to both the normal and the perfect refraction/reflection direction (constructed by cross-product). The last rotation is about the transformed normal with the angle ***PHI***. Thus giving a ray which has the desired direction relative to the perfect reflection/ refraction direction.

Rendering

In the rendering pass all of the concepts described above are put to use. A ray is launched from the eye, through the viewplane and into the scene. For each intersection Russian Roulette is performed to determine whether the ray is absorbed and recursion stops or if it should be traced further.
If the ray is not absorbed it will be reflected or refracted depending on what type of material the surface which the ray hit has.

For a diffuse surface a number of rays will be launched at random directions towards the hemisphere of the surface. The random directions generated are proportional to a cosine-weighted solid angle about the normal of the surface with the probability distribution function;

$$p(\Theta) = \frac{\cos(\theta)}{\pi}$$

Where Θ is the direction generated and θ represents the angle between the surface normal and the direction Θ. The direction Θ with the given probability distribution function is constructed by generating two uniformly distributed variables $r_1,\ r_2\ \in \{0,1\}$and with the help of them generate two directions $\theta \in \left\{0, \frac{\pi}{2}\right\}, \varphi\ \in\ \{0, 2\pi\}$ according to the formulas below.

$$\varphi = 2\pi r_1$$

$$\theta = \cos^{-1}(\sqrt{r_2})$$

The new ray direction is computed by transforming the normal by the two angles.

For a refractive or reflective surface there will also be a number of rays generated at random directions but the probability distribution function differs significantly from that of a diffuse surface. The random directions generated on the hemisphere in this case are proportional to a cosine lobe around the normal with the probability distribution function

$$p(\Theta) = \frac{n+1}{2\pi}\cos(\theta)^{\mathrm{n}}$$

Where Θ is the generated direction and θ is the angle relative to the surface normal.
This choice of PDF allows for full control of how diffuse the reflections and refractions should be; ranging from the perfect reflection/refraction direction with a very large value of n to a totally diffuse reflection/refraction with a small value of n.
The PDF is normalized according to the equation for normalization [14] and the angles φ, θ are generated according to the formulas below.

$$\varphi = 2\pi r_1$$
$$\theta = \cos^{-1}(r_2^{\frac{1}{n+1}})$$

The new ray direction is then computed by transforming the normal to an off normal axis (the perfect reflection or refraction direction) and then transforming by the two angles generated.
The resulting cosine lobe around the off-normal direction is shown in figure 10.
In the case of refraction there will also be a reflecting ray spawned with the same probability distribution but with its lobe placed around the perfect reflection direction. The refracting and reflecting rays will be weighted according to the Fresnel equations.

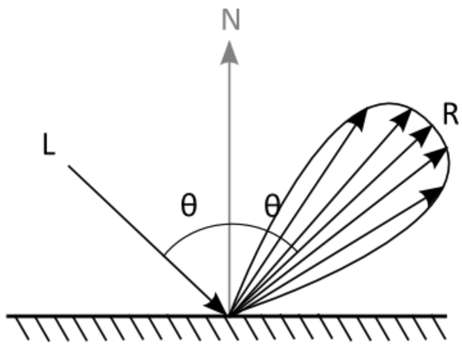

Figure 9 – Cosine Lobe around off-normal axis.

The direct contribution from the light source is computed with the help of shadow rays and if there is a specularity component in the surface the modified Blinn-Phong specularity is used.

The contributions from the newly spawned rays are then multiplied by their respective BRDFs; Lambertian BRDF for diffuse surfaces and modified Blinn-Phong BRDF for reflective/transmittive surfaces. They are then divided by the PDF for the current generated direction and summed with the direct contribution from the light source.

Photon mapping

Before the rendering pass starts, a photon emission pass occurs. Two maps are created, one for representing global illumination and the other for caustics. The caustic map needs to be a lot denser than the global map which is one of the reasons they should be separated. For the global map, photons are emitted from the area light source by randomizing directions in the hemisphere of the light. The photon is then traced through the scene, interacting with the objects it hits. As a photon hits an object, three actions can happen. Either the photon is reflected, refracted or absorbed. This is determined with the help of material properties and Russian roulette. Until the photon is absorbed, at each diffuse surface-hit a photon is stored in the photon map.

After the first intersection between an emitted photon and an object a shadow ray from the intersection point and along the light ray direction is traced. At each consecutive hit, a shadow photon is stored. Shadow photons helps to speed up the rendering pass by determining whether or not shadow rays needs to be cast. A photon has information about its power, incoming direction and whether it is a shadow photon or not. The same technique is used for the caustic map. However, the caustic photons are launched only towards specular objects which can reflect and refract.

When the photon emission pass is done the photons are stored in a suitable data structure. This implementation uses a kd-tree data structure. The kd-tree builds a binary tree by splitting the point set in half for each depth, splitting in fixed dimensions (x, y and z) until each leaf contains the photons at the corresponding subspace.

In the rendering pass, the photon map can be used in several ways. The global photon map can be used to optimize sampling directions for diffuse indirect lighting. It can also be used to determine whether shadow rays needs to be traced to the light source for rendering shadows. The radiance estimate from the nearest photons of a ray-traced point can be used instead of the regular diffuse indirect lighting. Instead of visualizing the radiance estimate directly at a diffuse surface a technique called final-gathering is used. This technique works by: at a diffuse surface a number of rays are shoot at the hemisphere of the surface. At the second intersection the photon map is called for the nearest photons and the radiance estimate is returned to the first intersection.

$$L_r = \frac{1}{\Delta A} * \sum f_r(x, \omega_p, \omega) * \Omega_p(x, \omega_p)$$

The caustic map is used as an additional term to the rendering equation. This means that the radiance estimate from the caustic map is visualized directly and added to the final color of each pixel.

Results and Benchmarks

Different amount of jittered samples per pixel compared with time and quality.

Because of the inherent randomness in a Monte Carlo Ray tracer the amount of samples taken per pixel will inevitably affect the quality of the picture. Any pictures rendered with a low amount of samples per pixel will give a noisy result; this noise rapidly decreases as the amount of samples increase. In figures 10, 11 and 12 below are three similar scenes rendered using 49, 1024 and 3600 samples per pixel the reduction in noise is most obvious in the ceiling since it has only a diffuse indirect component, it lies above the lamp and thus receives no light from the light source. All of the images displayed below have been rendered on an Intel core i7 with 4 cores and 8 threads.
There is a slight variation in brightness in the pictures which is due to a minor change in material color for the three walls.

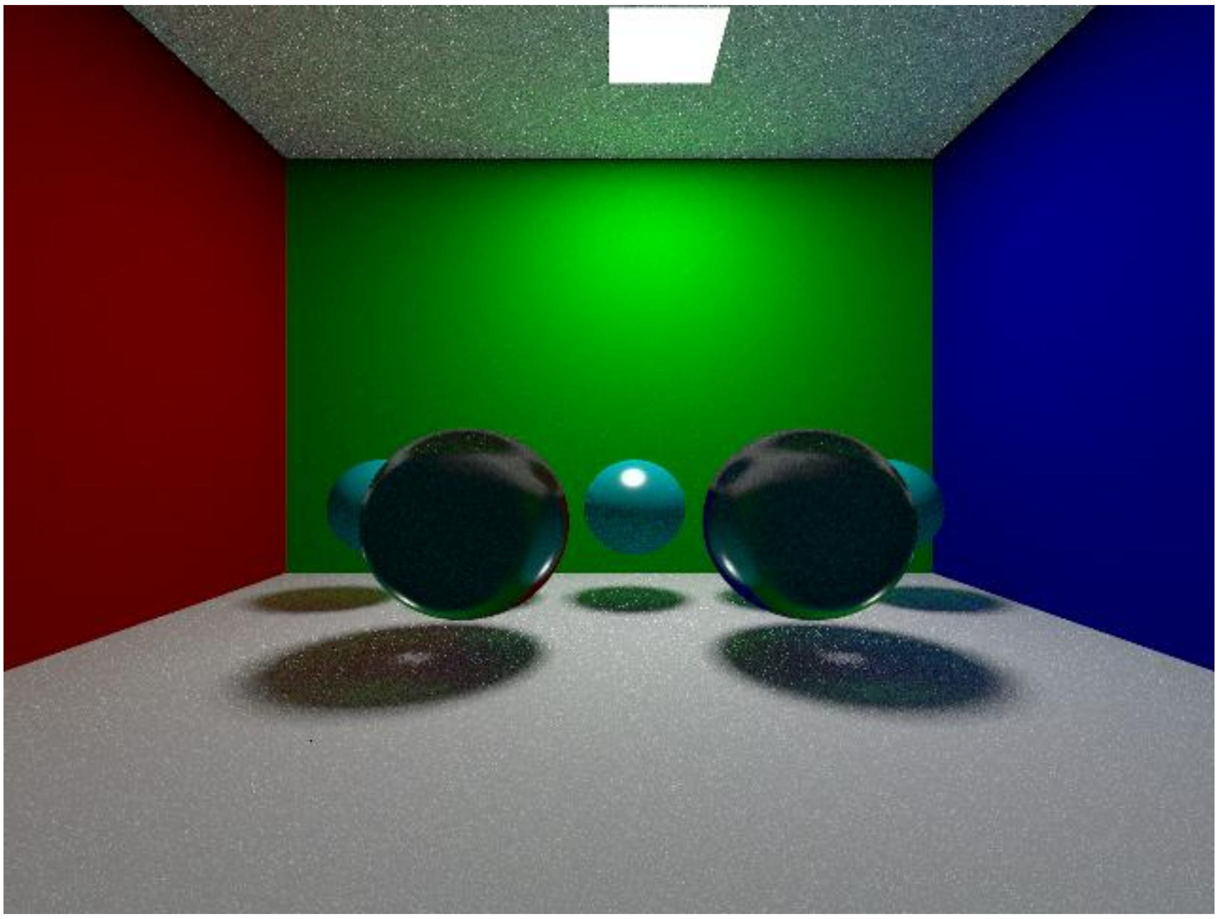

Figure 10– rendered at 49 samples per pixel

Figure 10 shows two refracting glass spheres and 5 diffuse spheres rendered at 49 samples per pixel. The noise is very apparent in the shadows and in the ceiling. What is also noteworthy are the caustic effects created by light focusing from the light source through the glass spheres. The image took 10 minutes to render.

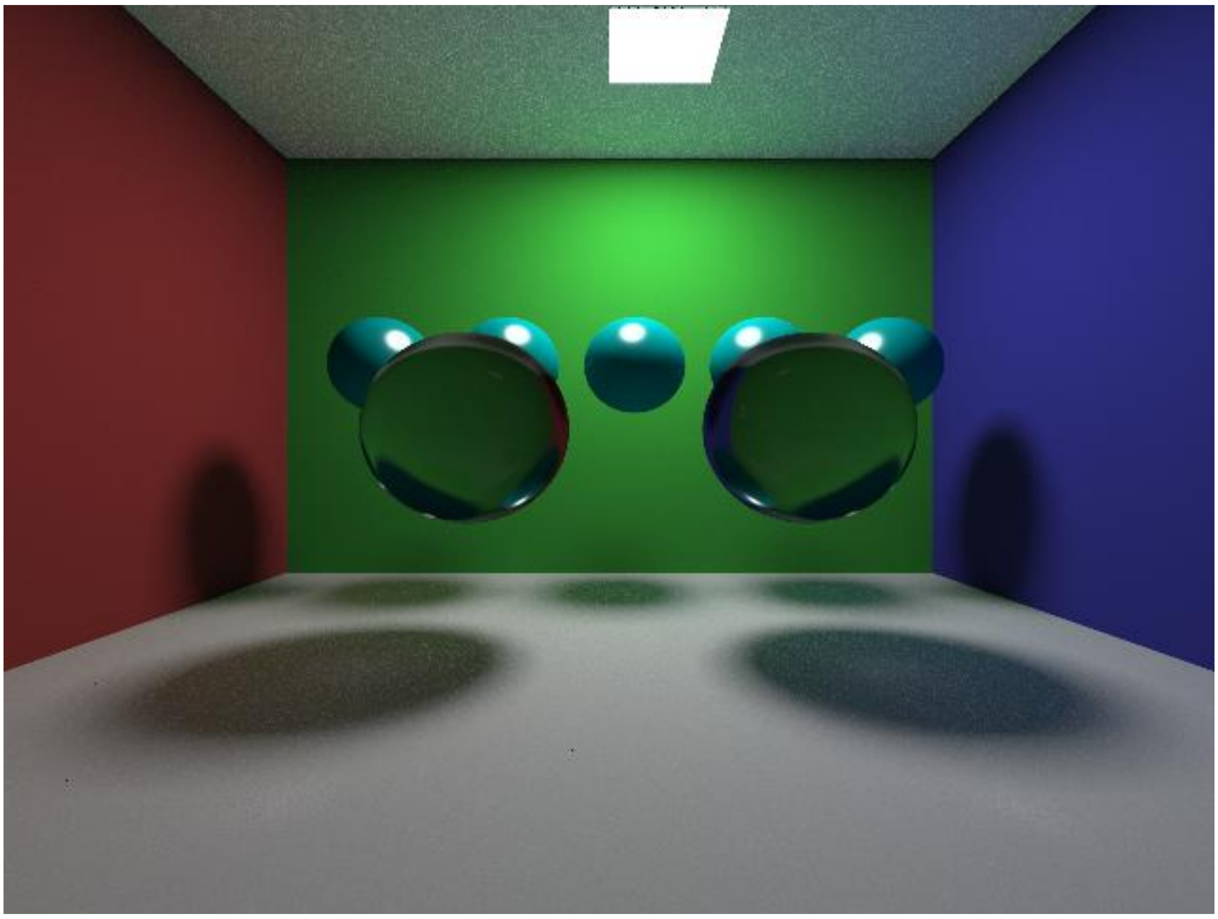

Figure 11 – Rendered at 1024 samples per pixel

Figure 11 show the same glass spheres but slightly elevated and rendered at 1024 samples per pixel. The reduction in noise can be seen in the ceiling and in the shadows. The shadows have also become smoother than in the picture rendered using 49 samples per pixel. The image took 1 hour and 30 minutes to render which is significantly higher than the rendering time of the picture with 49 samples per pixel.

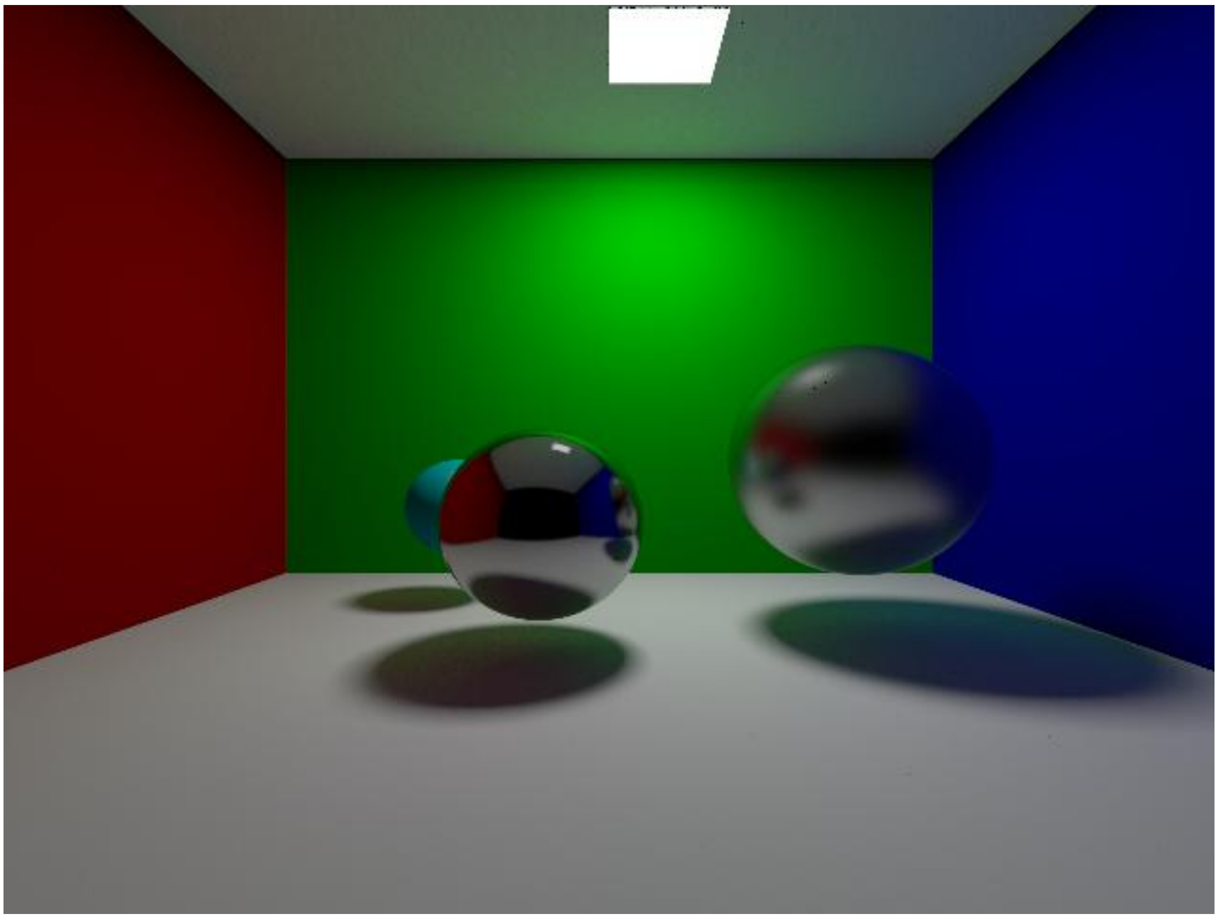

Figure 12– Rendered at 3600 samples per pixel

Figure 12 shows two reflecting spheres of different level of "*diffuseness*" but the interesting features of the image is the significant reduction in noise which can be seen if it is compared to the picture rendered at 1024 samples per pixel. At this amount of super sampling nearly all of the noise earlier seen in the ceiling and in the shadows is gone. The shadows have become even smoother than they were in the picture rendered and the color bleeding effects have smoothened out. The image took 5 hours and 30 minutes to render.

Perfect vs diffuse reflection/refraction

One of the major strengths of the Monte Carlo Ray tracer is its ability to simulate diffuse or *glossy* reflections/refractions and thereby producing more realistic pictures due to the fact that real reflections/refractions are never perfect.
The control of the width of the cosine lobe around the reflection/refraction direction via a single parameter "*n*" produces an efficient and easy way of dealing with glossy reflectors/refractors.
The figures below demonstrate the range of "*diffuseness*" the renderer is able to produce for reflections and refractions.

Both of the figures have been rendered using 1024 jittered samples per pixel on an Intel core i7 with 4 cores and 8 threads. The slight variation in brightness between the two figures is due to a minor change in the material color of the walls.

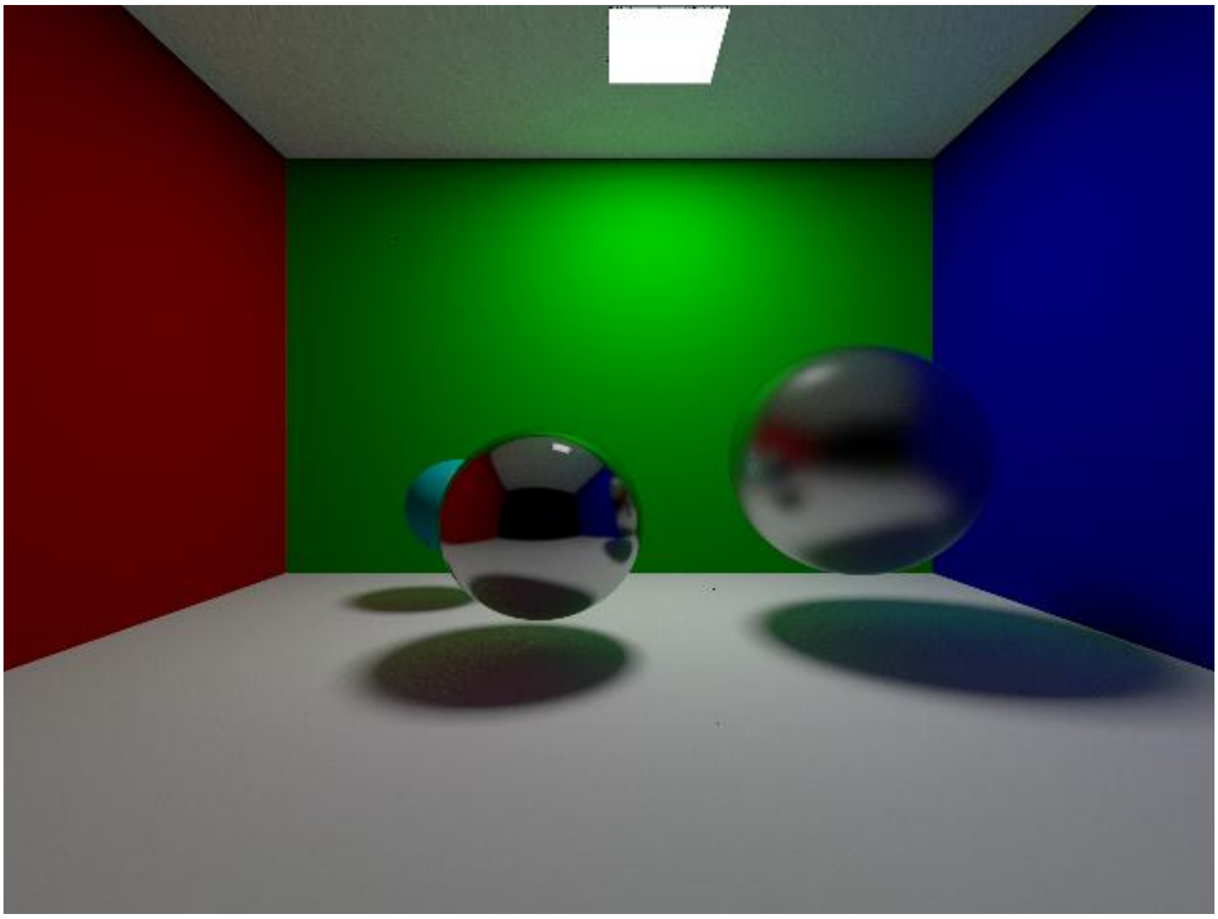

Figure 13 – glossy reflection at work

Figure 13 shows glossy reflection at work, the rightmost sphere has the parameter *n* set to 21 and the leftmost sphere has it set to 2001. There is no measurable difference in rendering time between using glossy and perfect reflectors. This image took 1 hours and 25 minutes to render. Also noteworthy is the color bleeding clearly seen in the shadows and the ceiling.

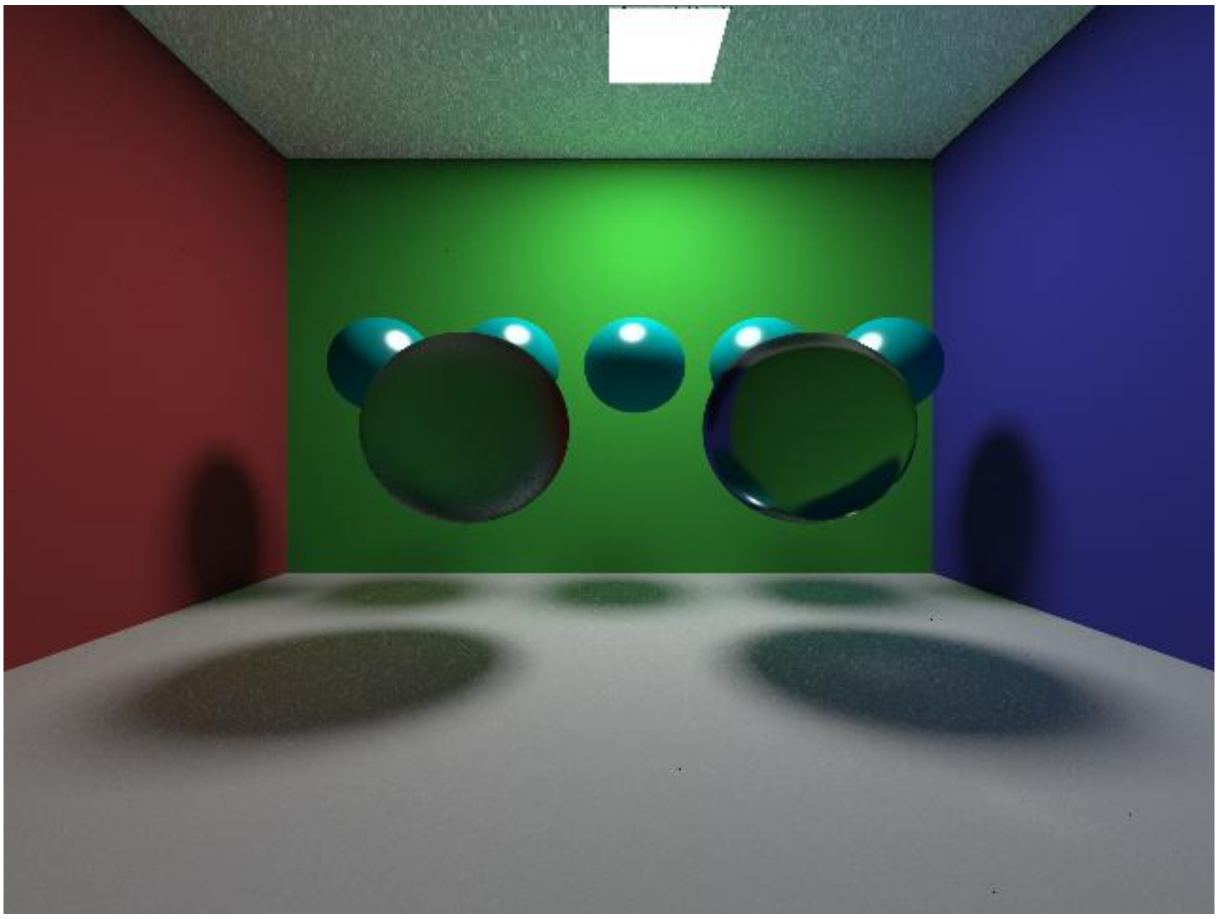

Figure 14 – glossy refractors at work.

Figure 14 illustrates the range in *"diffuseness"* which can be produced by the renderer in refracting materials. The leftmost sphere has its parameter *n* set to 21 and the rightmost has it set to 2001. There is no measurable difference in rendering time between the two values of the parameter *n* for refraction either and this image took 2 hours and 5 minutes to render.

Comparing photons vs pure monte-carlo

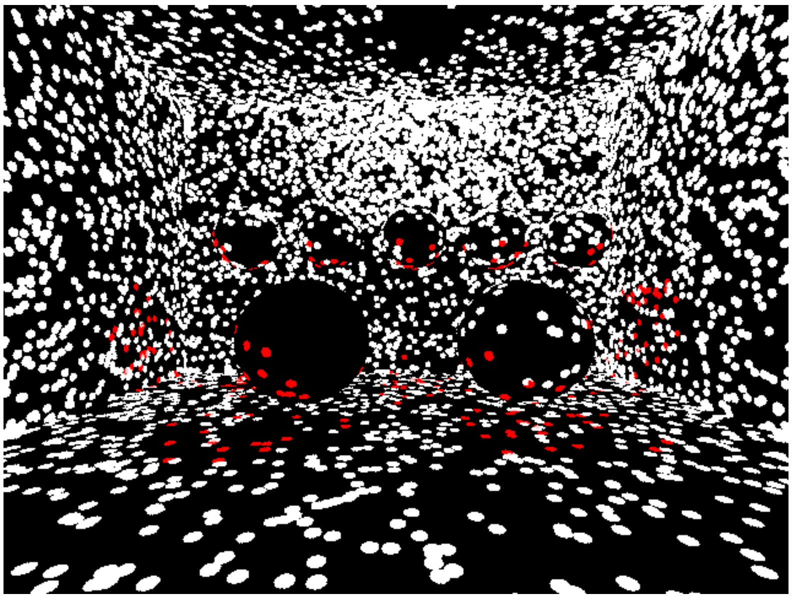

Figure 15 – Visualization of global(white) and shadow(red) photons.

The above figure, figure 15, is a representation of 10000 photons traced through the scene. The white dots are photons stored after direct or indirect hits on diffuse surfaces, note that the black ball to the left has a transmittive surface, not a diffuse surface, and therefore have no white dots stored on it. The red dots are the shadow photons which are the photons that is not lit by the light source. On the wall in the background the photons are closer together than compared to the photons on the floor, which is correct because it should be brighter near the light source. This was a good way to make sure that the photons where correctly distributed throughout the scene before the nearest neighbor algorithm was applied.

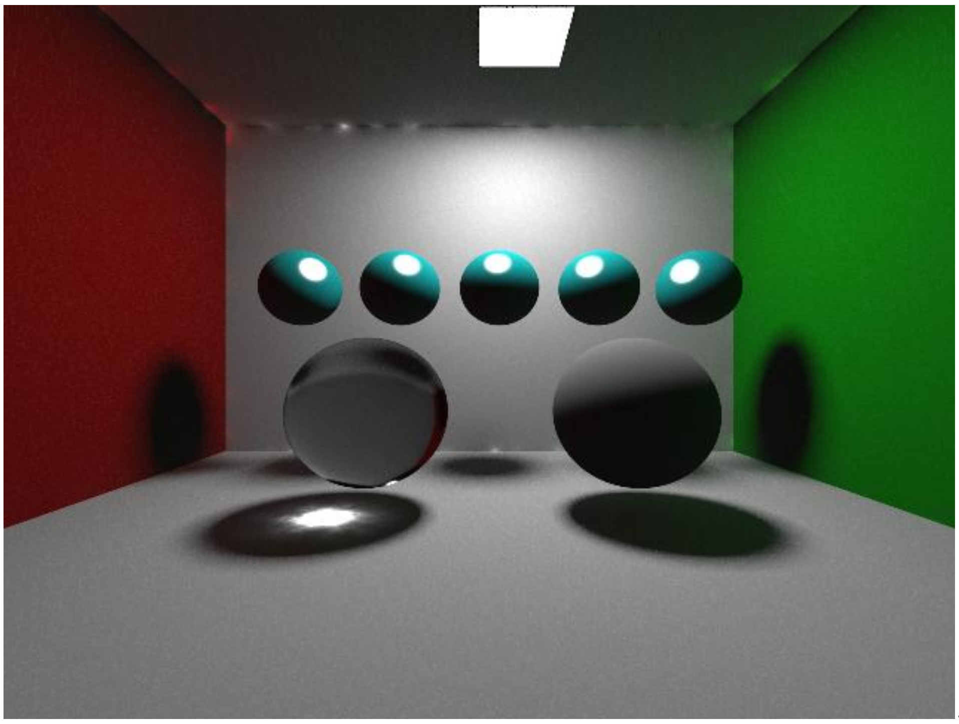
.

Figure 16 – Caustic and Global photon map.

This is the result of combining the caustic and global photon map, see figure 16. Note the nice caustic bellow the glass sphere. Some global color bleeding is seen, but it has some flaws, for example if one looks at the edges, the intensity varies quite a bit. This is the result of that there are too few photons gathered in the nearest neighbor method in the radiance estimate. Here follows a comparison of the amount of nearest neighbor photons, see Figures 17, 18.

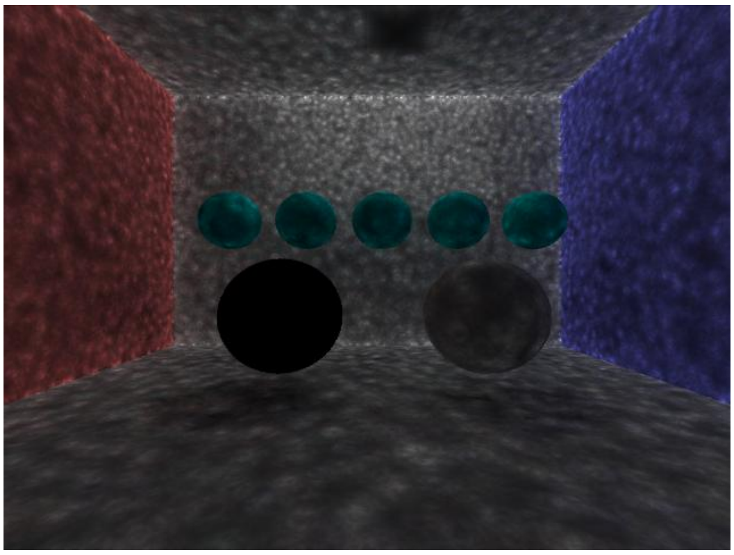

Figure 17 - The Radiance Estimate with 10 nearest neighbor photons and 100000 photons.

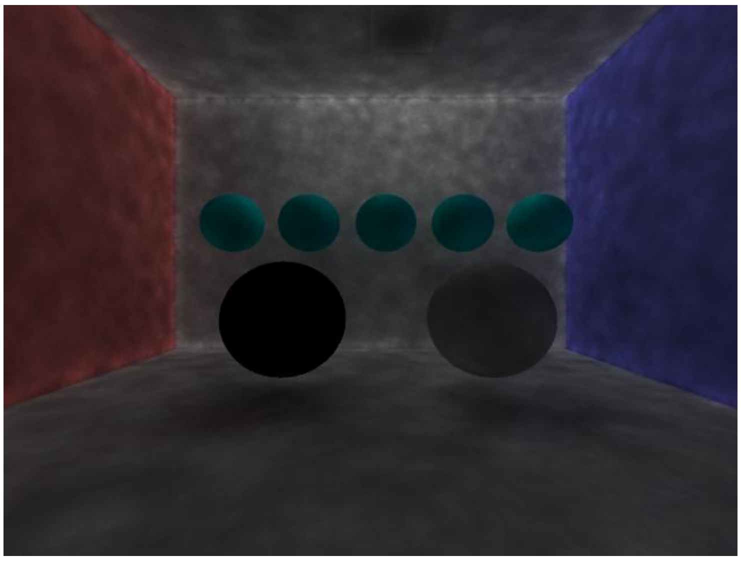

Figure 18 - The Radiance Estimate with 50 nearest neighbor photons and 100000 photons.

The amount of photons shot into the scene is 100000 photons and the amount of nearest neighbor photons is 10, figure 17, respectively 50 in figure 18. By adding more nearest neighbor photons the walls gets more smoothed out, which is desirable, but the rendering time increases dramatically.

Monte Carlo vs pure Whitted Ray tracing

The figures below show the exact same scene with the same number of jittered samples per pixel rendered with two different methods. Both of the pictures are rendered with 1024 samples per pixel on an Intel core i7 with 4 cores and 8 threads.

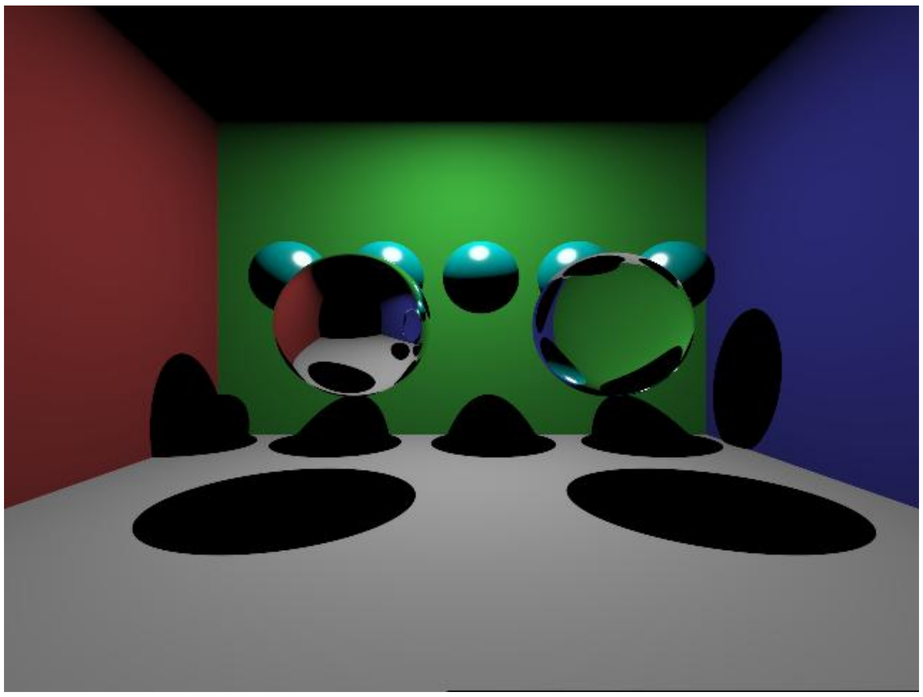

Figure 19 – The scene rendered with a pure Whitted ray tracer

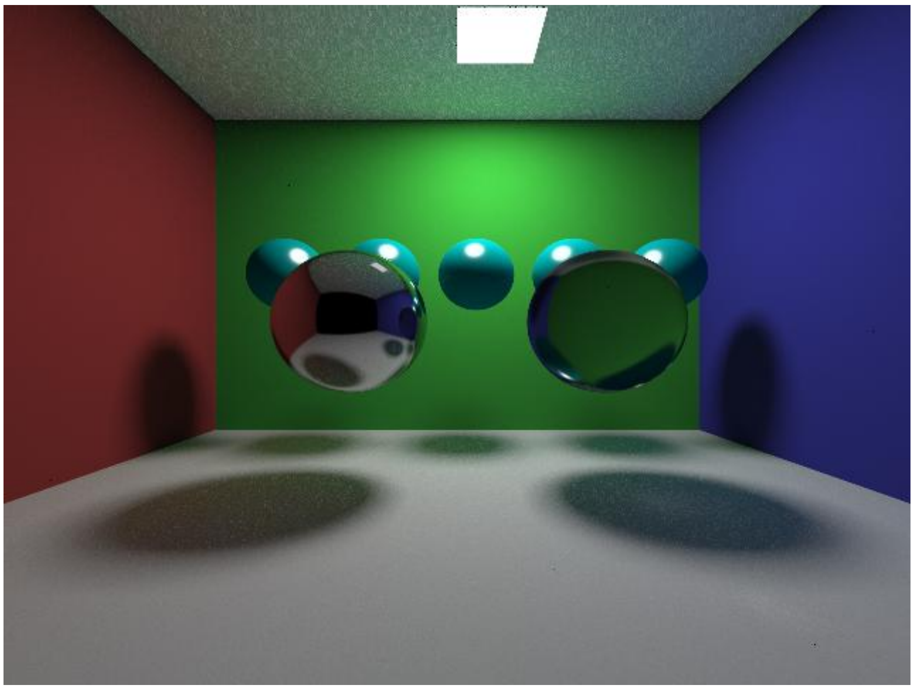

Figure 20 - The scene rendered with a pure Monte Carlo ray tracer

Figure 19 shows the scene rendered with a pure Whitted ray tracer, the image took 25 minutes to render. Figure 20 shows the same scene rendered with a pure Monte Carlo ray tracer, the image took 50 minutes to render. The limitations of a pure Whitted ray tracer are obvious when the two images are compared; the Whitted ray tracer fails to produce the soft shadows seen in the Monte Carlo ray tracer. The reflections and refractions are too sharp to be realistic in the Whitted ray tracer and the ceiling has vanished since it lies above the lamp and therefore cannot receive a direct contribution from the light source.

Monte Carlo vs Monte Carlo optimized by Photon Mapping

Figures 21 and 21 display the same scene rendered with Monte Carlo ray tracing and Monte Carlo ray tracing using Photon Mapping. The images are rendered with 1024 jittered samples per pixel on a Intel core i7 processor with 4 cores and 8 threads.
In figure 21 the scene is rendered with a pure Monte Carlo approach, the scene took one hour and 5 minutes. The noise is quite low in the picture and there is a trace of a caustic effect near the rightmost sphere.

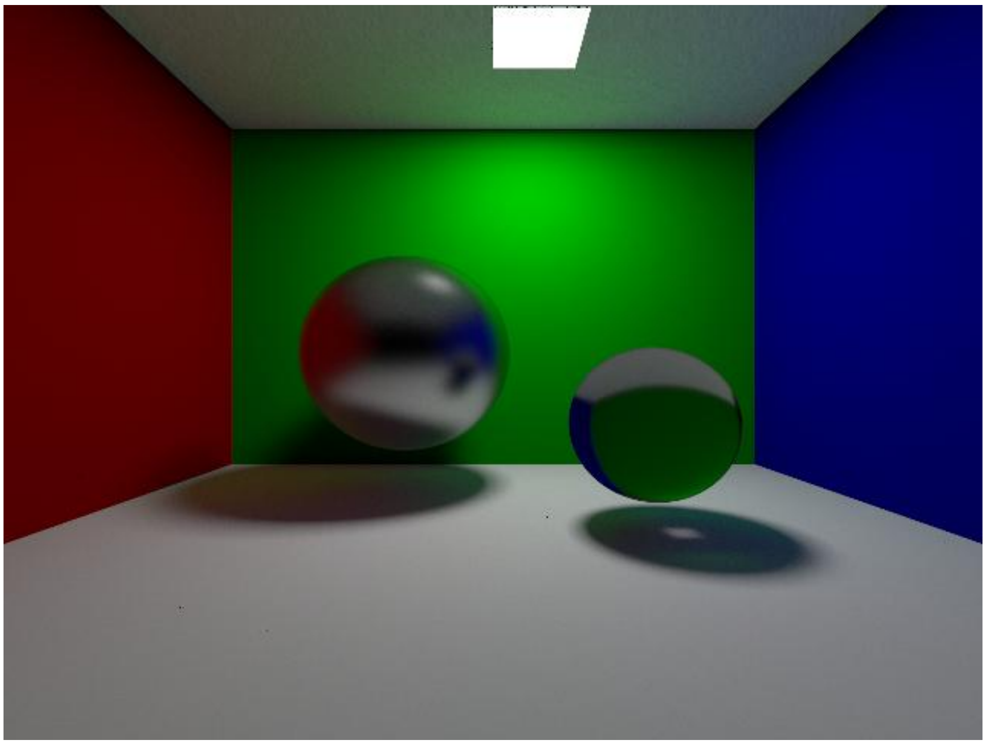

Figure 21 – The scene rendered with pure Monte Carlo

In figure 22 the scene is rendered with a Monte Carlo approach and Photon Mapping has been used to render the caustic effects. The scene took one hour and 40 minutes to render.
It is clear that the Photon Mapping approach clearly out performs a pure Monte Carlo in comes to caustic effects. The caustics are much brighter and clearer in this picture than in the one in figure 21.

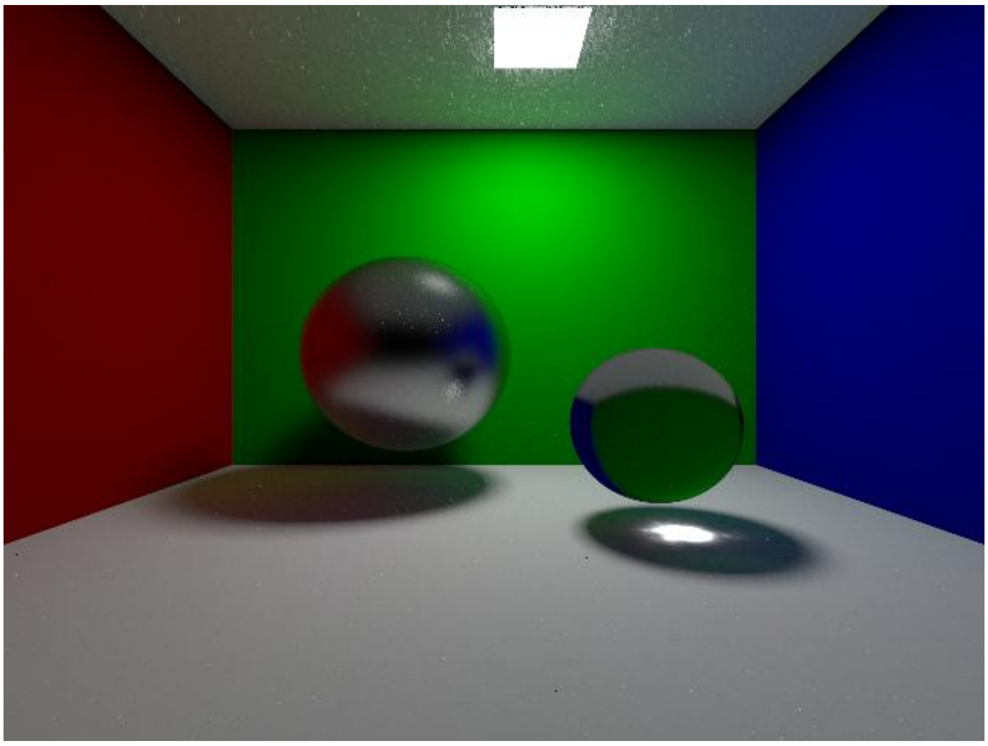

Figure 22 – The scene rendered with a Monte Carlo ray tracer using Photon Mapping.

Discussion

The difference in quality for realism between Whitted ray tracing and Monte Carlo is vast. The color bleeding, the soft shadows and the glossy reflections/refractions make the scene look more real. The drawback however for Monte Carlo ray tracing is the large rendering times.

It is possible to simulate caustics by using pure Monte Carlo ray tracing, but needs a huge amount of rays to generate it. The caustic with Monte Carlo ray tracing looks rather poor, see figure 21. With the use of photons, caustics can be rendered with a photorealistic result, see figure 22. The rendering time is increased when rendering with caustics, which is due to the fact that the caustic emission is not focused on the objects that are transmittive and instead shoots out photons all over the scene. This is a huge loss of photons but by focusing the photons to the transmittive objects the rendering time can be reduced.

Both pure Monte Carlo and Monte Carlo ray tracing with photon maps can generate color bleeding. However the lookup in our Kd-tree implementation is not optimal and the rendering times increases dramatically with increasing nearest neighboring photons. The color bleeding with pure Monte Carlo is however satisfying, see figure 13.

Instead of emitting photons in a completely randomized way over the lights hemisphere, the photons can be emitted for example with a more even distribution model like stratified sampling or N-rooks [15]. This would result in a less noisy image with the same amount of photons.

An error occurs when nearest neighbor search is performed near edges between two walls. This artifact can be seen in figure 16. This happens when wrong photons are collected; only photons from the same plane should be taken into account in the radiance estimate.

The rendering times are optimized with importance sampling. However, further improvements could be achieved by using quasi-Monte Carlo techniques [16].

Possible further work could include incorporating volume photon maps with participating media [17] and creating a scene importer for more complex scenes and objects.

Reference list

[1] Wann Jensen, Henrik: “Global Illumination using Photon Maps”. Department of Graphical Communication, The Technical University of Denmark, 1996.

[2] Goral Cindy M.;Torrance Kenneth E.;Greenberg Donald P.;Battaile Bennett.:”Modelling the interaction of light between diffuse surfaces”. SIGGRAPH '84 Proceedings, Cornell University,Ithaca,New York.

[3] Whitted, Turner: “An improved illumination model for shaded display”. Magazine Communications of the ACM Volume 23 Issue 6,June 1980.

[4] Dutré Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 36-37.

[5] Wikipedia: http://en.wikipedia.org/wiki/Fresnel_equation, 2010-12-08.

[6] Cook R. L.;Porter T. Carpenter L.:”Distributed Ray Tracing.”, Computer Graphics 18:3, 1984.

[7] Kajiya James T.:”The rendering equation ”, SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques,ACM SIGGRAPH Computer Graphics, volume 20 Issue 4, Aug. 1986.

[8] Dutré Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 47-81.

[9] Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 111-113.

[10] *Sunday Dan:* “Intersections of rays, segments, Planes and Triangles in 3D”, *http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm, 2010-12-06.*

[11] Dutré Philip:”Global Illumination Compendium”, http://people.cs.kuleuven.be/~philip.dutre/GI/TotalCompendium.pdf, Computer Graphics, Department of Computer Science, Katholieke Universiteit Leuven, Sept. 29, 2003, pp. 32.

[12] Dutré Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 39.

[13] Wikipedia: http://en.wikipedia.org/wiki/Quaternions, 2010-12-08.

[14] Dutré Philip:”Global Illumination Compendium”, http://people.cs.kuleuven.be/~philip.dutre/GI/TotalCompendium.pdf, Computer Graphics, Department of Computer Science, Katholieke Universiteit Leuven, Sept. 29, 2003, pp. 18.

[15] Dutré Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 71-72.

[16] Dutré Philip; Kavita Bala; Bekaert Philippe:”Advanced Global Illumination”, second Edition, pp. 75.

[17] Wann Jensen, Henrik; Christensen Per H.:”Efficient simulation of light transport in scenes with participating media using photon maps”, 1998.