Monte Carlo Raytracing from Scratch

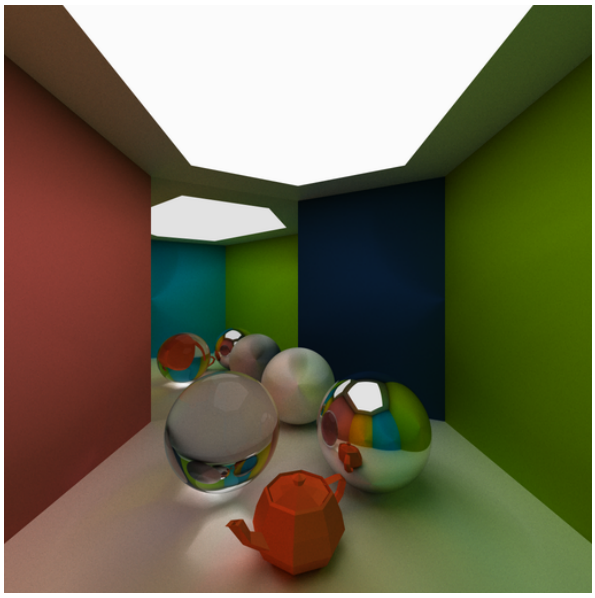
Martin Estgren <mares480@student.liu.se>
Rasmus Hedin <rashe877@student.liu.se>
Erik S. V. Jansson <erija578@student.liu.se>

Linköping University, Sweden

November 12, 2017

Abstract

A Monte Carlo raytracer is a renderer which produces photorealistic images after full convergence. It uses a global illumination model, meaning it gives optical phenomena like soft shadows, color bleeding and caustics. This report details the theory and the practical details necessary to implement one from scratch along with the photon mapping extension. After that, we benchmark the raytracer and display example renders from our raytracer. Finally, we do a discussion about the further improvements that can done in the raytracer and give some reflections on the project. The full C++ source is on GitHub [1].



[1] https://github.com/CaffeineViking/mcrt

Contents

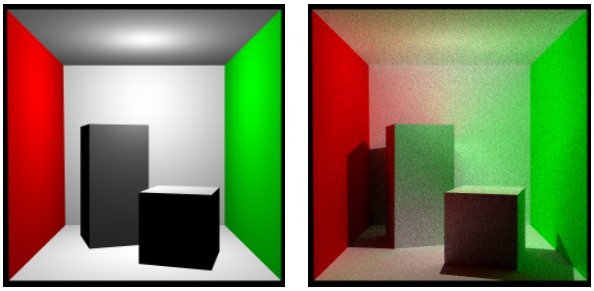
1 Introduction

Several fields of industry use *computer graphics* to generate and display synthetic images on a screen; e.g. the entertainment industry uses *raytracers* for *rendering* animated movies while *rasterizers* usually are the technology powering real-time video games. Of course, it's also widely used in the engineering and scientific disciplines for visualizing field data, which even have their own sub-field called *scientific visualization*. Since it is such a wide field, we'll only be focusing on the *rendering problem*: the task of converting one *scene description* to an *image* of it.

Rendering can usually be done in one of two ways, called the *rasterization* and *raytracing* techniques, or, by using some hybrid of these. *Rasterization* is when we geometrically project a scene, composed of primitives, onto an image plane (our camera) and then color the pixels based on a *local lighting model*. Meaning, objects in a scene are *shaded* only based on position, material properties, viewpoint direction, and light source information; never on other objects. Rasterization is very fast since there is hardware dedicated to these operations, and each of these is independent of each other, in other words, it's an *embarrassingly parallel* problem. *Raytracing* on the other hand shoots *rays* from the pixels in the *camera viewplane* and finds *intersections* with geometry in the scene. These rays bounce around the scene by *specular reflection* or *specular transmission*, until it finds a *diffuse surface*, which then *absorbs* it. It's a process by recursion, which approximates *irradiance* falling onto a pixel. Since it takes into account other objects, it's a technique which gives *global illumination*. Unfortunately, raytracing doesn't have full hardware support, but is still fast since it's also an embarrassingly parallel task (done for each sample).

In this report we'll describe all the party tricks we've used to implement our *Monte Carlo raytracer* from scratch. It simulates all *light transport* effects for *perfectly diffuse* and *perfectly specular* surfaces. We've also implemented *photon mapping* to speed up our convergence rate for higher quality *caustics*. Arbitrary *triangle meshes* can also be rendered and use *bounding volumes* to ignore low-effort intersection tests. Lastly, we also support *quadric geometry*. All of our scene is specified by using a JSON format.

We continue Section 1 by giving a brief overview of the field and introducing desirable properties for achieving *photorealism* with a model describing it. We then take a excursion through the most notable rendering schemes to solve the *rendering equation*. In Section 2 we break apart our raytracer bit-by-bit and explain each part in turn, along with any relevant theory necessary. We then play around with our raytracer's knobs in Section 3 to see if they affect the final render, and more importantly, the rendering time. We'll try to see if there is a balanced trade-off between render quality and speed. After all this, Section 4 concludes the report by critically looking at our implementation and results and also giving insight into what could be improved further.



(a) local illumination (b) global illumination

Figure 1: Two renders of the famous *Cornell box* in our raytracer; comparing both illumination models.

1.1 Global Illumination

A rendering technique is deemed to be *photorealistic* if, after full convergance, it exactly mimics reality. One of the most important ingredients for achieving photorealism is *global illumination*. We can compare *local illumination* and *global illumination* by looking at Figure 1. Notice that GI (global illumination) has *color bleeding*, *hard and soft shadows* and *caustics*. These phenomena are not possible in a local model without "hacks" like the *shadow mapping* technique.

1.2 Rendering Equation

A model describing most light transport phenomena is the *rendering equation*, shown below, presented by *James Kajiya* [7]. All of the rendering schemes are attempts at solving it. We present in the coming sections the most well known rendering techniques.

$$\mathcal{L}_o(\vec{x}, \hat{\omega}_o) = \mathcal{L}_e(\vec{x}, \hat{\omega}_o) + \\ + \int_\Omega \mathcal{L}_i(\vec{x}, \hat{\omega}_i) f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o)(\hat{n}_x \cdot \hat{\omega}_i)\, d\hat{\omega}_i$$

1.3 Radiosity

Radiosity for computer graphics assumes all surfaces are *perfectly diffuse patches* (e.g. triangles), and builds on the concept of finding the radiance that is transferred between patches in the scene using an iterative process. Each iteration solves the equation below by reflecting radiance for each patch:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \ ,$$

where B_i is the *radiosity* of *patch* i, E_i the *emitted radiosity* by i, ρ_i is the *surface reflectivity* of i, and the sum defines the *radiance* falling onto i between each patch in the scene and the patch i, depending on the *form-factor* F_{ij} (geometric term) of i and j.

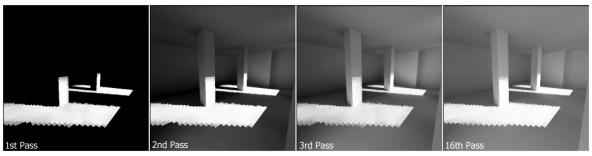


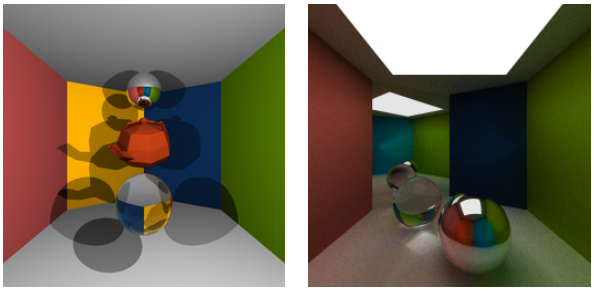
Figure 2: Shows how the radiosity technique spreads the radiance in each iteration (render by *H. Elias*).

Iterating on the above equations, a discrete approximation of the *rendering equation* is achieved for perfect Lambertian surfaces as the patch size goes infinitesimal and the iterations tends to infinity (but there needs to be some cutoff point, otherwise numerical errors might start appearing). You can see a radiosity-based renderer iterate through a scene in Figure 2. Some advantages of radiosity over GI techniques that takes into account specular light is that it calculates the radiosity of the full scene and not just based on the camera viewport, making it useful when baking light-maps with the scene. It's a spin-off of the heat transfer equations in thermodynamics into rendering by *Goral et al.* [5].

1.4 Whitted Raytracing

In the *Whitted* [13] ray tracing method initial rays are emitted from the camera into the scene. The rays will intersect with objects and new reflective and/or refractive rays are spawned. In this method it is assumed that we only have *perfect reflections and refractions*. The direction of a refracted ray is computed with Snell's law given the refractive indices of the materials at the ray intersection point.

Reflecting and refracting rays will build up a *tree of rays*. When all rays have terminated (i.e. hit a diffuse surface) the radiance at each intersection is computed from the leaf nodes back up to the root node (the camera). The radiance from the child rays of an intersection point contribute to the total radiance leaving each intersection point. *Shadow rays* are sent from each of these points to the light sources, to know if the point is in shadow. If the shadow ray reaches a source of light then it's not in shadow and the light therefore contributes to the total radiance. Figure 3 (a) is being rendered by it.



(a) Whitted-ish raytracing (b) Monte Carlo raytracing

Figure 3: Renders of *the scene* in Section 2.1 using our raytracer in the early and later progress stages.

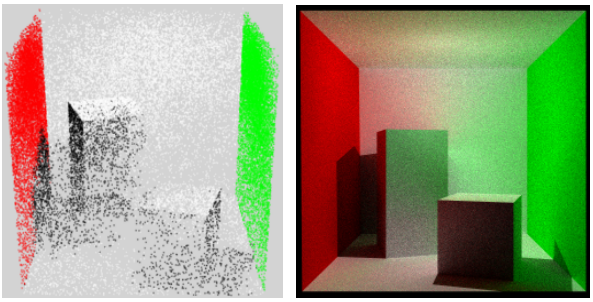
1.5 Path Tracing

The *path tracing* method works in a similar way to Whitted raytracing, but the reflections and refractions may not always perfect. Instead, when a ray intersects with an object it will be reflected in a *random direction* originating from the *hemisphere* at the intersection point. These rays are recursively reflected until they intersect with a light source. But since some rays never hit a light source another condition to terminate is needed. The solution is called *Russian roulette*, where rays have a probability of being *absorbed* at the intersection point. This gives the technique an unbiased result. A tree will be built recursively just as before, adding contribution from the child nodes and from the light sources (which may have an area now) with a local lighting model.

This is a method that needs many *samples* to converge, and thus uses *Monte Carlo integration*; making the *path tracing technique* a *Monte Carlo raytracer*. Figure 3 (b) shows a render of a scene using our Monte Carlo raytracer; we have more interesting light phenomena than Whitted raytracing.

1.6 Photon Mapping

One of the main problems with *path tracing* is that it usually takes very long to *converge* to a *noiseless* image. To speed up the raytrace convergence, *H. Jensen* proposed an algorithm where rays with low importance are approximated using a spatial map which describe how light sources deposit flux in the scene. These data-structures are called *photon maps* [6] and significantly speed up convergence of a typical *path tracer* by requiring a bit more memory.



(a) direct light photon map (b) possible raytrace result

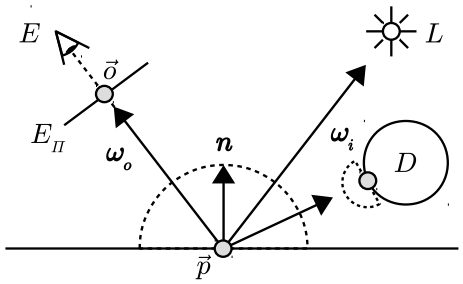
Figure 4: Here the flux coming from the light source is distributed as seen in (a) before we do raytracing.

Photon mapping is done in two passes. The first pass is done before the *path tracer* as a preprocessing step where photons are emitted from light sources and bounced around the scene in a way similar to the typical *path tracer*. These photons are usually stored in a *balanced kd-tree* whenever they hit a diffuse surface. The second pass is done in conjunction with the *path tracer* where rays with low importance are approximated by photons stored around the terminating scene-intersection. The number of photons to consider is either calculated by a bounding sphere of fixed size or a sphere expanded until it encompasses a fixed number of photons. These photons are then used to approximate radiance at the intersection point. We'll see later in the report that our implementation uses the "*fixed sphere radiance estimation*" approach for this.

In some cases a higher resolution secondary *photon map* is used to show caustics effects since these are very expensive to do in regular *path tracing*. In this case, photons are only emitted towards specular objects in the scene and in much greater numbers compared to the regular *photon map*. This enables one to get detailed caustic effects cheaply and fast.

2 Theory and Method

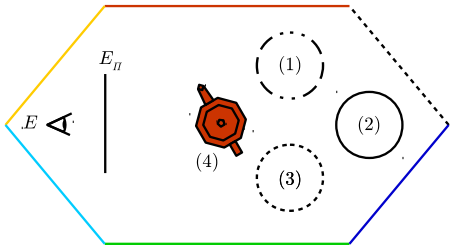
Raytracing attempts to find irradiance falling onto each pixel in the *image plane* E_Π. For each pixel at world position $\vec{o}_{ij}$ on E_Π we send off *rays* originating at E, the *eye origin*. These *importance rays* will *hit* and *bounce* around the *scene* described in Section 2.1 according to the theory at Section 2.2. Depending on the surface properties, we'll need to consider the *radiance reflectance rate*, $f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o)$, in Section 2.3, for evaluating the *radiance* at e.g. the point $\vec{p}$. The *light contributions* for $\vec{p}$ come in two forms: *direct light* (e.g. the light source L) and *indirect light* (e.g. the diffuse surface D). These are described in Sections 2.4 and 2.5. To reduce noise, and converge faster, we use *photon maps* in Part 2.6, to approximate the direct light contributions. Lastly, we may get *aliasing*, which we deal with in Part 2.7.



2.1 Scene Description

Below is an overview of the scene we'll be using to display the results and do our benchmarking. All walls are Lambertian reflectors with different colors, while the floor and roof are white. An area light source with 4 triangles exists, it's white, on the roof.

There exists four primitives in the scene, an orange teapot mesh (4) with a Lambertian BRDF, a white sphere (2) with Oren-Nayar BRDF $\sigma' = 0.1$, a perfectly reflective sphere in (3) and a perfectly refractive sphere in (1) with a refraction index 1.5.



2.2 Ray-Surface Intersections

The scene can be constructed using a number of different geometric surfaces and requires different intersection test for each surface type. Common for them all is that the incoming ray is defined on the parametric formula $\vec{o}+t\hat{d}$ where $\vec{o} \in \mathbb{R}^3$ denotes the origin of the ray, $t \in \mathbb{R}$ specifies a distance from the origin, and $\hat{d} \in \mathbb{R}^3$ indicates the direction of the ray.

2.2.1 Parametric Sphere

Spherical objects are defined as a vector $\vec{o} \in \mathbb{R}^3$ which denotes the origin of the sphere, and a scalar r as the radius of the sphere. Sphere-ray intersections are done using a geometric solution to the following:

$$\vec{L} = \vec{o}_{sphere} - \vec{o}_{ray}$$
$$t_{ca} = \vec{L} \cdot \hat{d}_{ray}$$

if $t_{ca} < 0$, there's no intersection between ray and sphere. Otherwise look for intersection points using:

$$d^2 = L \cdot L - t_{ca} \cdot t_{ca}$$

if $d^2 > r^2$, there's not intersection, otherwise:

$$t_{hc} = \sqrt{r^2 - d^2}$$
$$t_0 = t_{ca} - t_{hc}$$
$$t_1 = t_{ca} + t_{hc}$$
$$t_{min} = \min(t_0, t_1)$$

where t_{min} is the distance from $\vec{o}_{ray}$ to the closest intersection point on the sphere. If $t_{min} < 0$ we don't have an intersection as well. The intersection point is calculated as $\vec{i} = \vec{o}_{ray} + t_{min}\hat{d}_{ray} - \vec{o}_{sphere}$ and the surface normal as $\hat{n} = ||\vec{i} - \vec{o}_{sphere}||$.

2.2.2 Triangle Polygon

Intersections between triangles and rays are computed using the *Möller-Trumbore* [9] algorithm, which allow for intersection tests without having to compute plane containing the triangle. First a local coordinate system for the triangle is computed using the corner vertices $\vec{v}_1$, $\vec{v}_2$, and $\vec{v}_3$ of the form:

$$\hat{e}_1 = \vec{v}_2 - \vec{v}_1$$
$$\hat{e}_2 = \vec{v}_3 - \vec{v}_1$$

the surface normal is extracted as $\hat{n} = \hat{e}_1 \times \hat{e}_2$, we check if the ray travels parallel with the triangle surface by $\hat{p} = \hat{d_{ray}} \times \hat{e}_2$, and if the determinant $d = \hat{e}_1 \cdot \hat{p}$, $d = 0$, the ray does not intersect the triangle. Otherwise the intersection test continues:

$$\vec{t} = \vec{o}_{ray} - \vec{v_1}$$
$$\vec{q} = \vec{t} \times \hat{e_1}$$

$$u = \vec{t} \cdot \hat{p} * d^{-1}$$
$$v = \hat{d}_{ray} \cdot \vec{q} * d^{-1}$$

if the two conditions

$$u < 0 \textbf{ or } u > 1$$
$$v < 0 \textbf{ or } u + v > 1$$

don't hold, we have an intersection and the distance t the ray has to travel to the intersection point can be calculated as $\vec{e_2} \cdot \vec{q} * d^{-1}$. The triangle normal is simply $\hat{n} = \vec{e}_1 \times \vec{e}_2$, if $\hat{n} \cdot \hat{r}_{ray} > 1$, o.w. $\hat{n} = \vec{e}_2 \times \vec{e}_1$.

2.2.3 Triangle Mesh

To speed up the intersection test for geometries using multiple triangles we use a hierarchical geometric representation where the geometry of interest is encapsulated within a bounding sphere, this speeds up the test since only the triangles are taken into account if we already know that the ray will intersect the bounding sphere.

The bounding sphere is calculated by picking the smallest and largest *vertices* from the vertex set $\mathcal{V}$

$$\vec{v}_{min} = \arg\min_{\vec{v} \in \mathcal{V}}(\vec{v})$$
$$\vec{v}_{max} = \arg\max_{\vec{v} \in \mathcal{V}}(\vec{v})$$

and defining the bounding sphere as:

$$radius = \frac{||\vec{v}_{min} - \vec{v}_{max}||}{2}$$
$$\vec{origin} = \frac{\vec{v}_{max} - \vec{v}_{min}}{2}$$

2.3 Surface Properties

After hitting a diffuse surface with normal $\hat{n}$ coming from a ray direction $\hat{\omega}_i$, we need to find the reflected radiance going towards $\hat{\omega}_o$. This is affected by the *bidirectional reflectance distribution function*, f_r, of the surface. It's used to model the *ratio* of *reflected radiance* leaving the surface towards $\hat{\omega}_o$ with respect to the *irradiance* arriving from the direction $\hat{\omega}_i$. The first definition given by *F. E. Nicodemos* [10] shows:

$$f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o) = \frac{d\mathcal{L}_r(\vec{x}, \hat{\omega}_o)}{\mathcal{L}_i(\vec{x}, \hat{\omega}_i)(\hat{n} \cdot \hat{\omega}_i)d\hat{\omega}_i} \ .$$

There are several *surface reflectance models*; we'll only describe the *Lambertian* and the *Oren-Nayar reflectance models*, since those are in our raytracer.

2.3.1 Lambertian Model

Assumes the surface reflects radiance *isotropically*, meaning it reflects radiance *equally in all directions*. It was introduced by *J. H. Lambert* [8] in 1760, and realistically models perfectly diffuse surfaces which are free of imperfections (i.e. there is no roughness).

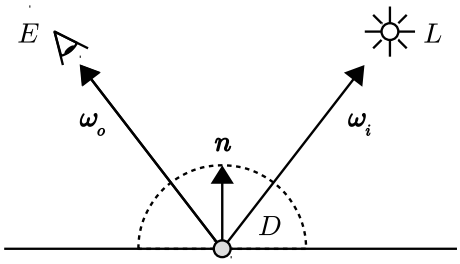


Figure 5: Diffuse "Lambertian" surface.

Since it doesn't depend on the incoming/outgoing directions, the value of f_r is constant in all directions within the hemisphere. This gives us the BRDF in Equation (1). Now, usually you see the factor $\hat{n} \cdot \hat{\omega}_i$ here as well, but we've chosen to not include it since it'll be appearing in the rendering equation anyway.

$$f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o) = \frac{\rho}{\pi} \ , \tag{1}$$

where ρ is the *albedo* of the diffuse surface D and π is the *normalization factor* for *energy conservation*. *Albedo* can hand-wavingly be interpreted as "color".

2.3.2 Oren-Nayar Model

Unfortunately, while the Lambertian model might be good for some surface types, it's inappropriate for others, such as concrete, ceramic and cloth. Which is a shame, since it's an simple and intuitive model.

Surfaces which are diffuse and have rough texture can be accurately modeled with a *Oren-Nayar* [11] *reflection model*. It's based on the *microfacet model* introduced by *Torrance-Sparrow* [12], that assumes a surface is composed of infinitesimally small, flat, Lambertian reflectors (as shown in the last section). An intuitive sketch of such a surface can be seen in Figure 6, where the V-shaped microfacets are actually supposed to be infinitesimally small sized.

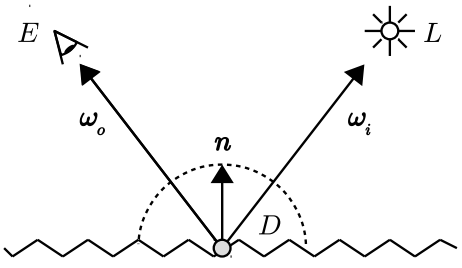


Figure 6: Diffuse "Oren-Nayar" surface.

Deriving Equation (2) is outside the scope of this paper, so if you're interested, consult the original paper. Note that this is the *qualitative Oren-Nayar* definition, and will therefore fail energy conservation laws when the *roughness* parameter is set $\sigma > 0.97$.

$$\begin{aligned} f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o) &= \frac{\rho}{\pi} \cdot (A + \sin\alpha \cdot \tan\beta \cdot \gamma \cdot B) \ , \\ A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33} \ , \ B &= 0.45\frac{\sigma^2}{\sigma^2 + 0.09} \ , \\ \alpha = \max\{\theta_i, \theta_o\} \ , \ \beta &= \min\{\theta_i, \theta_o\} \ , \\ \gamma &= \max\{0, \phi_i - \phi_o\} \ , \end{aligned} \tag{2}$$

where θ is the *inclination* and ϕ the *azimuth* of $\hat{\omega}$. By changing σ we can control the *surfaces' roughness*. Notice that this BRDF depends on the ray's angle, and is therefore *anisotropic*. Implementing this isn't that straightforward since our raytracer has $\hat{\omega}$'s that aren't easily decomposable into the form $\hat{\omega} = (\theta, \phi)$. It boils down to doing projections onto the surface plane. We can get a vector parallel to the plane with $\hat{p} = \hat{\omega}_i \times \hat{\omega}_o$ and then $\hat{q} = \hat{p} \times \hat{n}$. By projecting down $\hat{\omega}_i$ onto $\hat{q}$ with $\hat{\omega}_{i\|\hat{q}}$, we get $\theta = \hat{\omega}_i \cdot \hat{q}$, $\phi = \hat{\omega}_{i\|\hat{q}} \cdot \hat{q}$.

Because the qualitative Oren-Nayar calculations have quite a few expensive operations, we've chosen to implement *Yasuhiro Fujii's* [4] Oren-Nayar variant. It doesn't require trigonometric functions, and gives empirically almost indistinguishable results to O-N. Like regular Oren-Nayar, it suffers from not following energy conservation laws when $\sigma' > 0.97$.

Actually, experimentally, *Y. Fujii* has found that his proposed approach matches regular Oren-Nayar better than the qualitative Oren-Nayar within [11].

$$\begin{gathered} f_r(\vec{x}, \hat{\omega}_i, \hat{\omega}_o) = \rho \cdot \left(A' + \frac{s}{t} \cdot B' \right) , \\ A' = \frac{1}{\pi + \left(\frac{\pi}{2} - \frac{2}{3}\right)\sigma'} , \; B' = \frac{\sigma'}{\pi + \left(\frac{\pi}{2} - \frac{2}{3}\right)\sigma'} , \\ t = \begin{cases} 1 & \textbf{if} \quad s \leq 0 \\ \max\left\{\hat{n} \cdot \hat{\omega}_i, \hat{n} \cdot \hat{\omega}_o\right\} & \textbf{otherwise} \end{cases} , \\ s = \hat{\omega}_i \cdot \hat{\omega}_o - (\hat{n} \cdot \hat{\omega}_i)(\hat{n} \cdot \hat{\omega}_o) \; . \end{gathered} \tag{3}$$

Unlike qualitative Oren-Nayar, the above BRDF is a breeze to evaluate, for both the hardware and the programmer (who likes projecting stuff anyway).

Looking at Figure 7 we see results returned from our raytracer. We should make a couple of interesting observations. Notice that Oren-Nayar reflectors with $\sigma' = 0$ are essentially the same as Lambertian reflectors. Which makes sense, since $\sigma' = 0$ means the surface has no roughness (i.e. perfectly diffuse).

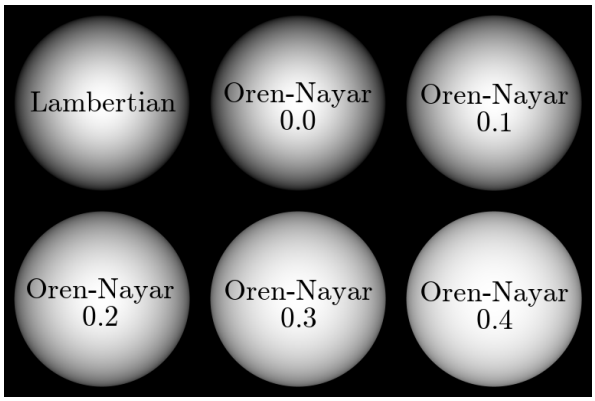


Figure 7: Spheres rendered using our raytracer with different roughness values, σ', for the Oren-Nayars.

To wrap things up, when we hit a diffuse surface, we wish to know the ratio of radiance that will leave the surface. For this we need to choose between the Lambertian and Oren-Nayar model to calculate f_r. Each surface in our scene defs. it's f_r model to use.

2.4 Direct Light Contributions

Most importance rays that are sent from the camera into the scene never hit a light source. Therefore, the direct light from the light sources to an intersection point needs to contribute to the total radiance at the point. This is computed with *shadow rays*, which are sent from each intersection point to the sources of light. The shadow rays will provide light contributions to the radiance at the ray intersection points. To determine if the intersection point lies in shadow, a visbility test is performed. Transparent objects are obviously ignored by the visibility test.

2.4.1 Point Light Source

The shadow ray $S_2 = x - y$ from an intersection point y to a point light source position x is launched. If it does not hit any object, it will pass the visibility test, and will therefore contribute radiance to the surface. The Lambertian cosine falloff, based on the angle θ_{S_2} between the normal N and S_2, is applied, as can be seen in the equation below.

$$L(y \rightarrow -\Psi_1) = f_r(y, \omega_{S_2}, -\Psi_1) L(y \leftarrow S_2) cos\theta_{S_2}$$

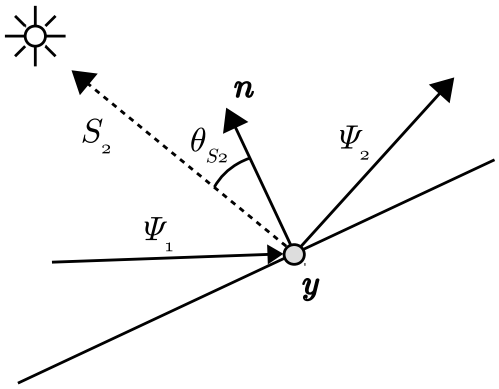


Figure 8: Shadow ray toward point light.

2.4.2 Area Light Source

The direct light from an area light source needs to be approximated for each intersection point. This is done with a Monte-Carlo estimator $< L_D >$ using M shadow rays, as will be explained in the next section. The light source is a triangle with corners

$\mathbf{v_0}$, $\mathbf{v_1}$, and $\mathbf{v_2}$ and with area $A = 0.5||(\mathbf{v_1} - \mathbf{v_0}) \times (\mathbf{v_2} - \mathbf{v_0})||$. To get the shadow rays we need to pick random points q on the light source uniformly with PDF $p(q) = \frac{1}{A}$. This is done by drawing two random numbers u, v until $u + v < 1$ and building the point q with barycentric coordinates $q = (1 - u - v)\mathbf{v_0} + u\mathbf{v_1} + v\mathbf{v_2}$. Repeat this until we have M points.

2.4.3 Monte Carlo Method

$$< L_D >= \frac{AL_0}{M} \sum_{i=1}^{M} f_r V(x, q) G(x, q_i)$$

$G(x, q) = cos\alpha cos\beta / d^2$ is the geometric term where d is the length of the shadow ray and α, β are the inclination angles to the surface normal at the start- and endpoint. $V(x, q_i)$ is the visibility function which is 1 if no object is in the way and 0 otherwise. Thus, only unblocked shadow rays will contribute to the estimate giving a soft shadow. Since the light source is a Lambertian emitter L_0 is emitted for all points and directions of the light source.

2.5 Indirect Light Contributions

When rays are emitted from the camera they will intersect with surfaces. At the intersections the rays will either terminate or spawn new rays depending on the surface material. By following the path of the ray we build up a tree with intersections as nodes connected by rays. The child nodes will contribute radiance to the parent nodes which gives indirect light, i.e. light reflected at least once since it left the light source. It's usually split it into three types:

2.5.1 Specular Reflection

If a ray hits a perfect specular surface a new ray will always spawn. The direction of the new ray is determined by the perfect specular reflection law. The direction of the new ray $\vec{R} = \vec{I} - 2(\vec{I} \cdot \vec{N})\vec{N}$ where $\vec{I}$ is the incoming ray and $\vec{N}$ is the normal.

2.5.2 Specular Refraction

If a ray hits a perfectly refractive surface, i.e. a transparent object, a new ray is spawned that enters the object. The refractive ray $\vec{T}$ is computed with the perfect refraction law and the angle to the surface normal is computed with *Snell's law*.

$$\vec{T} = \frac{n_1}{n_2}\vec{I} + \vec{N}\left(-\frac{n_1}{n_2}(\vec{N} \cdot \vec{I}) - \sqrt{1 - [\frac{n_1}{n_2}]^2[1 - (\vec{N} \cdot \vec{I})^2]}\right)$$

When a ray is moving between media of different refractive indices we might both reflect and refract. The Fresnel equations are used to determine the distribution of the radiance over the refracted and reflected rays.

$$R_s = \left[\frac{n_1 cos\theta_1 - n_2\sqrt{1 - ([n_1/n_2]sin\theta_1)^2}}{n_1 cos\theta_1 + n_2\sqrt{1 - ([n_1/n_2]sin\theta_1)^2}}\right]^2$$

$$R_p = \left[\frac{n_2\sqrt{1 - ([n_1/n_2]sin\theta_1)^2} - n_1 cos\theta_1}{n_2\sqrt{1 - ([n_1/n_2]sin\theta_1)^2} + n_1 cos\theta_1}\right]^2$$

The total reflection coefficient becomes

$$k_r = (R_s + R_p)/2$$

The radiance of the parent node to the refracted and reflected ray is computed as

$$L = Rk_r + T(1 - k_r)$$

2.5.3 Diffuse Reflection

As described in 2.3 two diffuse surfaces are implemented in the project; Lambertian and Oren-Nayar. When these surfaces are intersected by a ray we need to determine the direction of the reflected ray. For specular surfaces the ray was reflected/refracted perfectly, but for diffuse surfaces this is not the case. Instead, a random azimuth angle ϕ_i and inclination angle θ_i is picked in the hemisphere. We pick two random values $u, v \in [0, 1)$ and compute azimuth $\phi_i = 2\pi u$ and inclination $\theta_i = cos^{-1}(\sqrt{v})$. The reflected ray will contribute to the radiance at the intersection giving indirect light.

In order to not get a biased result an unbiased termination condition is required. This is achieved with russian roulette 2.5.4 that determines if a ray should be terminated.

2.5.4 Russian Roulette

In order to achieve a unbiased result russian roulette is used to get an unbiased termination condition.

The idea is to select a reflection probability P for an intersection of a ray. This probability is connected to the surface properties. The radiance at a intersection point is approximated with a Monte-Carlo scheme using one ray.

$$L(x \to \omega_{out}) = \pi f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in})$$

A new ray is spawned at the intersection point with probability P and the ray is terminated with probability $(1 - P)$. In order to remove the bias the following equation is used.

$$L(x \to \omega_{out}) = \frac{\pi}{P} f_r(x, \omega_{in}, \omega_{out}) L(x \leftarrow \omega_{in})$$

To determine if a ray should be terminated the azimuth angle of the reflected ray is modified to $\phi_i = (2\pi/P)u$ where u is a random variable $0 \leq u \leq 1$. If $\phi_i > 2\pi$ the ray is terminated and a new ray is spawned if $0 \leq \phi_i \leq 2\pi$.

2.6 Photon Mapping

Photon mapping is a technique where photons are distributed within the scene and used during the path tracing render pass. This is done in order to provide an approximative method for radiance sampling during the path tracing step. Each photon represents a tiny packet of flux which provides information that helps us to reduce the number of reflection and shadow rays during the path tracing pass.

2.6.1 Gathering Photons

Photons are propagated from the light sources in the scene and bounced around the geometry similar to how rays are emitted through the camera viewplane and reflected/refracted based on the material properties of surfaces. In this implementation the main difference between ray and photon propagation is that the photons always terminate on the first diffuse surface which it intersects. Terminated photons are stored in a balanced kd-tree to allow for efficient photon lookup during the radiance estimation.

In this implementation a fixed number of photons are defined during the pre-rendering step P. Each light source is assigned a number of photons based on the size of the light source.

First the total area for all light sources in the scene is calculated:

$$A = \sum_{l \in \mathcal{L}} area(l)$$

where $\mathcal{L}$ is the set of all light sources. To get the number of photons a specific light source l will emit, the contribution of l towards the total emission area is used with the total number of photons:

$$p_l = \frac{P}{A} area(l)$$

which results in the number of photons a given light source should emit into the scene. Each photons' outgoing direction is cosine-weighted and the position uniformly sampled from the area of the light source.

2.6.2 Radiance Estimate

In this project, the computation of the direct light is replaced by an estimation based on the photon map. To estimate the radiance at point x all photons within a sphere with radius r. A fix sized sphere is used in this project. Some condition needs to be fulfilled in order estimate the radiance with photon mapping. If these conditions is not met the Monte-Carlo scheme will be used for the approximation. Photon mapping will not be used if the sphere contains shadow photons, or if the sphere contains few photons.

$$L(x \to \omega_{out}) = \sum_{i=1}^{M} f_r(x, \omega_{in,i}, \omega_{out}) \frac{\phi(x_i \leftarrow \omega_{in,i})}{\pi r^2}$$

Areas with low photon density may give a blurry result. Therefore, a cone-filter is applied to the estimate. This is done by giving a weight w_p to each photon based on the distance d_p between the point x and photon p.

$$w_p = max(0, 1 - d/(kr))$$

The filter is normalized by $1 - \frac{2}{3k}$ giving the final equation.

$$L(x \to \omega_{out}) = \sum_{i=1}^{M} f_r(x, \omega_{in,i}, \omega_{out}) \frac{\phi(x_i \leftarrow \omega_{in,i}) w_p}{(1 - \frac{2}{3k})\pi r^2}$$

2.7 Anti-Aliasing & Sampling

Since we're *sampling* irradiance falling onto pixels, we're converting from something that's *continuous* (i.e. the scene description) to something *discrete* (i.e. the pixels on a screen). If we're only to take one *sample per pixel* (SPP) we'll most likely get multiple *aliasing artifacts* (usually *"jaggies"* on most edges).

We're *undersampling*, in signal processing theory. To perfectly eliminate aliasing, we'd need to fetch samples at the *Nyquist rate* (or higher) and filter it. Unfortunately, that generally isn't possible, and is usually also overkill in computer graphics. Instead we use an approximation, called *supersampling*, that sends importance rays from sub-pixels, and then takes the weighted average of them. Let p_{ij} be the final pixel color (unknown) at location (i, j), and let $\mathcal{S}_{ij} = \{s_1, s_2, ...\}$ be sub-pixel colors (known) we have found when raytracing. We'll estimate p_{ij} by:

$$\hat{p}_{ij} = \frac{1}{|\mathcal{S}_{ij}|} \sum_{\forall s_k} s_k \ .$$

Below you'll find an example sketch, using 4 SPP. Another important use-case of supersampling which we've failed to mention explicitly is that it replaces Monte Carlo integration over the hemisphere for the indirect diffuse light component. We've already mentioned that we do Monte Carlo integration for direct area light sources, but in this case the Monte Carlo integration we're doing is implicit. We do this by accumulating all irradiance falling onto the pixels and then uses the above "Monte Carlo" estimator. Supersampling solves two big issues with one stone: anti-aliasing and also the Monte Carlo integration.

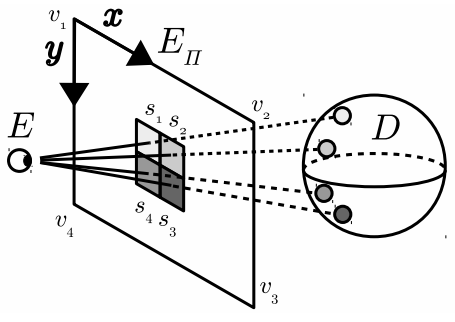
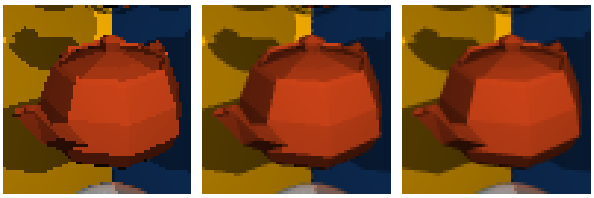


Figure 9: Intuition for *supersampling* anti-aliasing technique. Average the samples s_1, s_2, s_3, s_4 to find the pixel's expected color. Usually not four samples.



(a) 1x (b) 4x (c) 16x

Figure 10: Effects of different supersampling levels for 64x64 render of the *Utah teapot* in our raytracer.

Sometimes the locations where we evaluate our sub-pixels makes a difference in the final render. If the scene has of e.g. a texture which is of high-frequency (almost at the sub-pixel level) and has a regular pattern, then some details might be lost in the render process. It's largely affected by the *supersampling pattern* that we use to choose the sub-pixel locations which are going to be raytraced. We've implemented three patterns, as can be seen in Figure 11. The *grid* samples at regular intervals inside the pixel with a step size Δ_p. *Randomly* taking samples inside the pixel is also a valid strategy, leading to noisier, but unbiased results. Finally we have also implemented a *Gaussian sampling pattern*, with distribution $X \sim \mathcal{N}(\mu, \sigma^2)$ around the center.

From the four vertices $\{v_1, v_2, v_3, v_4\}$ of E_Π we build a coordinate system: $\vec{x} = v_2 - v_1$, $\vec{y} = v_4 - v_1$ to calculate the next sub-pixel positions (based on the pattern we use) we're going to use to send rays.

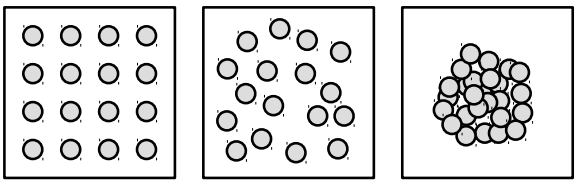


Figure 11: Visualization of *supersampling patterns* we've implemented: *grid*, *random*, and the *gaussian*. Another common technique is *jittering* (not shown).

As the last step, we need to be aware that we're storing *irradiance values* (using double precision) in our pixels. They are in the so called *sampling space*, but since we're going to be writing down the render to an image (integer values), we need to convert them from *sample space* to *image space*. We do this by $\hat{p}_{ij} \div \max\{\hat{p}_{ij}\}$, $\forall \hat{p}_{ij}$, scaling this down to $[0, 1]$.

3 Results and Benchmark

After describing our Monte Carlo raytracer we now show some example renders of the scene presented earlier in the paper, and point out the optical phenomena that arise. This raytracer has many parameters that can be toggled, and in order to identify the primary knobs that contribute to a good looking image, and those that only increase render time, we've measured the render time by changing parameters.

Final Render

Given unconstrained time, the Monte Carlo raytracer will converge to a realistic image, and will be noise-free. After 1024 samples the image below is produced, and gives pretty convincing results, with relatively little noticeable noise. It was rendered using an `Intel Core i7 860 @ 2.8 GHz` and using `8 GiB` of on-board main memory. The raytracer supports `OpenMP`, so it was rendered in 8-way parallel, one for each hardware thread (of 4 cores).

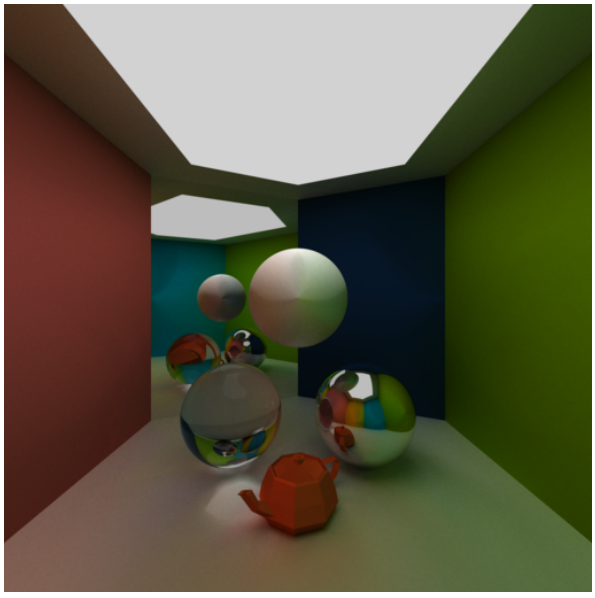
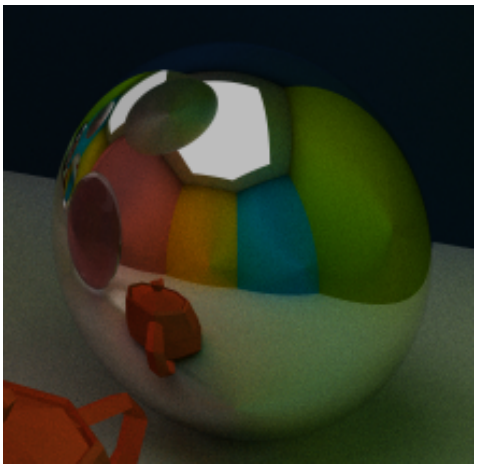


Figure 12: High-resolution render of the scene at 1024x1024, 1024 SPP, 1 shadow ray and 7 bounces. Render time taken was $\approx$ 133 minutes & 31 seconds.

The figure above contains all optical effects required for photorealism. In the coming sections each of these effects will be shown and described in detail.

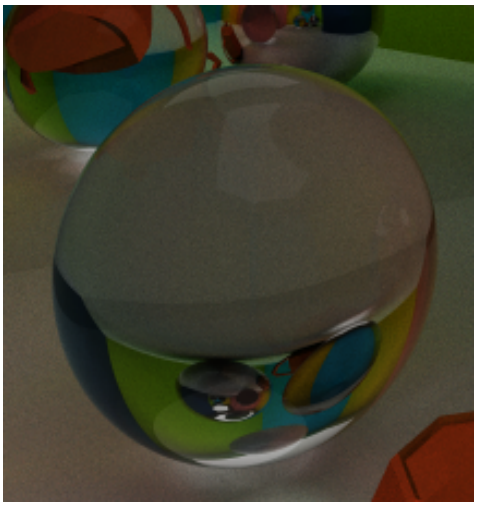
Reflection

By zooming into Figure 12 the mirror sphere on the right can be analyzed. We can see that we have a perfectly reflective surface, where the other surfaces in the scene, which aren't visible in the original viewplane directly, can be observed indirectly. We can see in the other figures that this can be applied recursively, as long as it doesn't go over ray depth.



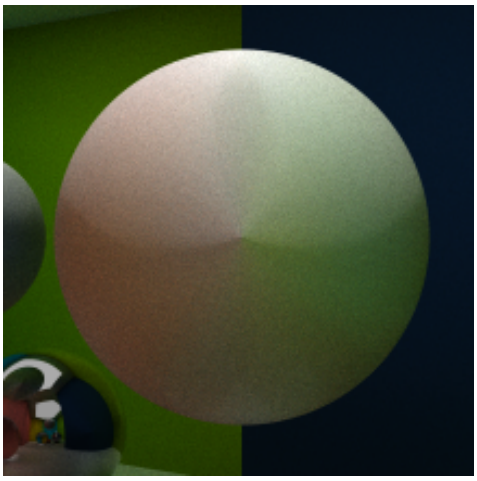
Refraction

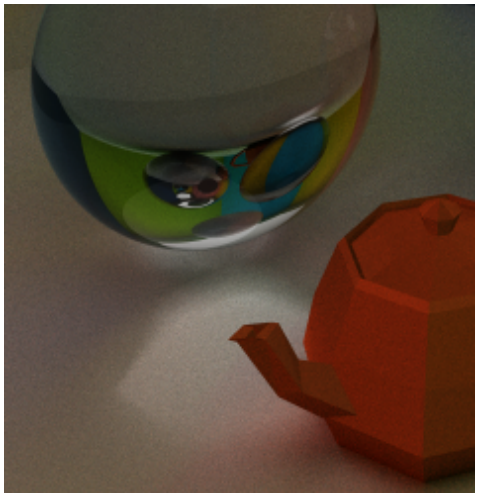
Below is a sphere with perfectly refractive material using an refractive index similar to glass. The convex lens which the sphere creates causes a mirrored refracted projection. Some Fresnel effects can be observed in the sphere, i.e, we get some reflections at grazing angles. Indirect refraction of other surfaces.



Caustics

Caustics are usually hard to reproduce with a monte-carlo ray tracer, but given enough time to converge, there is no limitation in the method itself. In the image below we can observe caustics created by refracting direct light through the sphere, as well as those created by reflecting direct light through the mirror and then through the sphere. One possible cause for the caustics being generated underneath the sphere is the self-refraction inside the sphere. It's also able to see the caustics through the reflection on the mirror sphere. Caustics are caused because the sphere acts as a lens, concentracing light/radiance into a smaller area than normal.



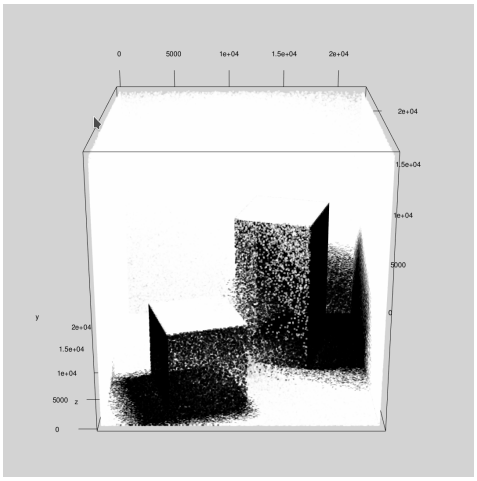


Photon Map Visualization

Below is a direct visualization of the photon map created by plotting each photon in the direct light photon map. We've chosen not to plot the scene and instead use the Cornell box since it's easier to see what's happening. The white points are regular photons carrying some flux in them, naturally each of these should carry only a little flux, so in reality they should be almost black (they are white for visualization purposes only). The black photons represent shadow photons. As can be seen, only diffuse surfaces create photons when intersected, and if there were any specular surfaces in the scene, they would only reflect or refract the photons' paths, never store them there. This was made by using `plot3d` in *R* library `rgl` (see this in `utils/photon-map.r`).

Color Bleeding

As can be seen in the image of the white Oren-Nayar sphere, it has received indirect illumination from the red wall and the green wall. This happens because the direct light of diffuse surfaces can be reflected multiple times, and this causes the radiance to be a mix of all intersected surfaces. In the mirror reflection of this sphere, there is also contributions from the blue wall via indirect illumination. This is one of the effects which are not possible with regular Whitted raytracing and is usually an important effect of global illumination and gives photorealism. You can also see this on the walls and teapot.



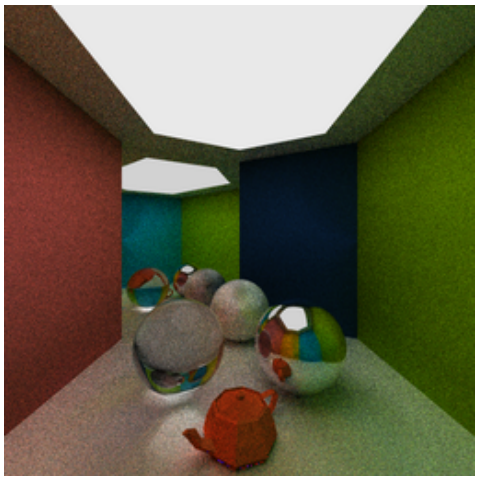
Parameter Benchmarks

In this section different parameter configurations will be tested to examine the visual and computational cost. All measurements were made on the same hardware configuration, an `Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz` with 2 cores and 4 hardware threads. The system has `16 GiB` of memory. We parallelize it 4-way in *OpenMP*.

The table below shows the parameter values for our *baseline* parameter configuration on which we'll perform pure Monte Carlo ray tracing, which will result in a photorealistic image given enough computation time. Figure 13 gives the render of these.

Parameter	Default Value
Resolution	512x512
Shadow Rays	1
Ray Bounces	7
Sampling Density	49 SPP
Photon Mapping	No
Photon Amount	1 million
Estimation Radius	0.1 units
Render Time	2 min 42 sec

Figure 13: Render of the scene 512x512 at 49 samples per pixel, 1 shadow ray and 7 max ray bounces.



Shadow Rays

This section covers how the number of shadow rays affects the result in comparison with the default parameter configuration.

Shadow Rays	Rendered	Time Taken
1	See Fig. 13	2 min 42 sec
4	See Fig. 14	7 min 26 sec
8	See Fig. 15	13 min 31 sec

Figure 14: Render of the scene 512x512 at 49 samples per pixel, 4 shadow rays and 7 max ray bounces.

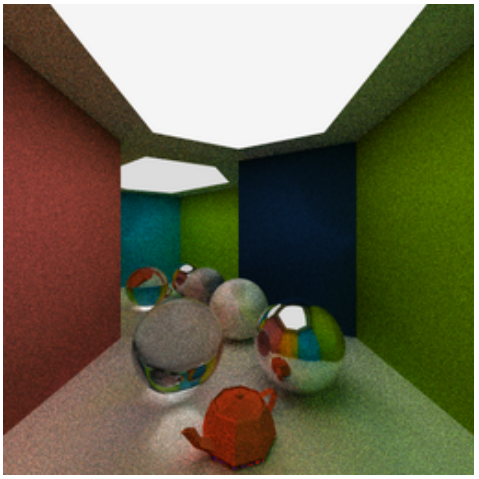


Figure 15: Render of the scene 512x512 at 49 samples per pixel, 8 shadow rays and 7 max ray bounces.

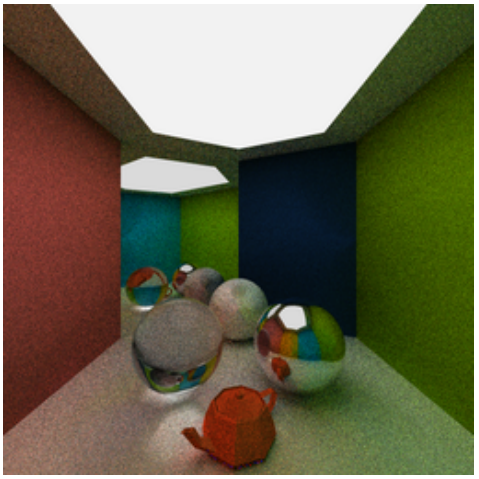


Figure 13-15 have been rendered with the same resolution and samples but with increasing amount of shadow rays. By comparing the images we can see that the number of shadow rays does not affect the result. Since we sample each pixel multiple times

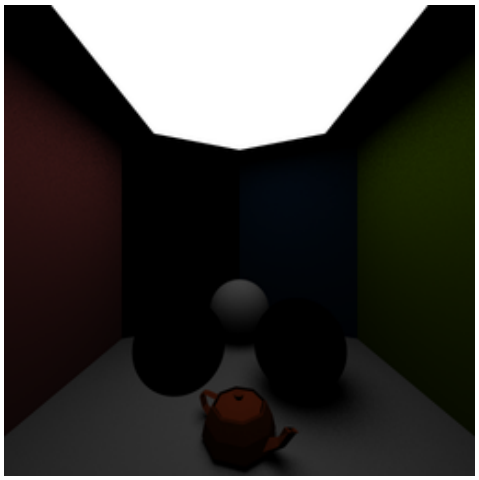
we implicitly get multiple shadow rays. Therefore, one shadow ray is enough to get a good result and the rendering time decreases as in the table above.

Ray-depth

The ray-depth parameter affects how many bounces each ray can do in the scene, with a value of 1 no reflections/refractions will occur.

Ray Bounces	Rendered	Time Taken
1	See Fig. 16	51.4 sec
4	See Fig. 17	2 min 9 sec
7	See Fig. 13	2 min 42 sec

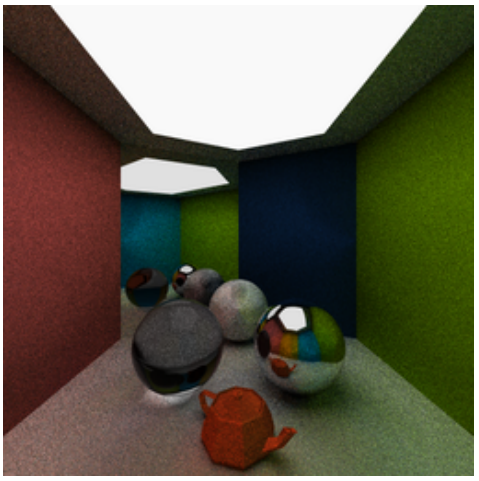
Figure 16: Render of the scene 512x512 at 49 samples per pixel, 1 shadow rays and 1 max ray bounces.



The scene becomes much darker since no indirect illumination is present in the scene. There is also no reflection and refraction present, since those count as ray bounces as well. This is a great image for visualizing the difference between the Oren-Nayar and the Lambertian surface. It might not be noticeable, but the falloff around the Oren-Nayar reflector is a less pronounced than that of the Lambertian emitter because the roughness of it is higher.

In Figure 17 we can see that indirect light seems to be arriving at the points we sample, but the reflections of high depth return black. This is something which doesn't happen in the baseline configuration.

Figure 17: Render of the scene 512x512 at 49 samples per pixel, 1 shadow rays and 4 max ray bounces.

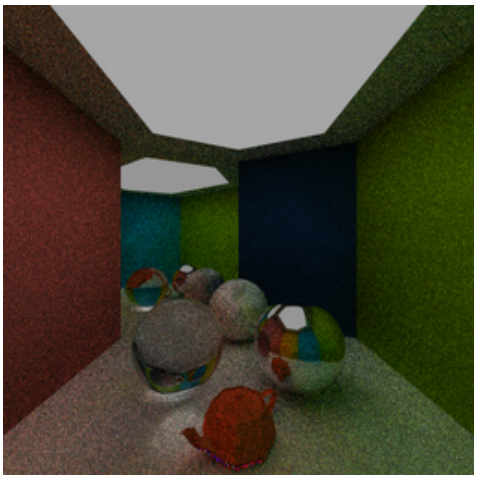


Sampling Density

The sampling density specifies how many samples will be taken for each pixel on the screen. Since it's something that increases with $O(n^2)$, it's expensive.

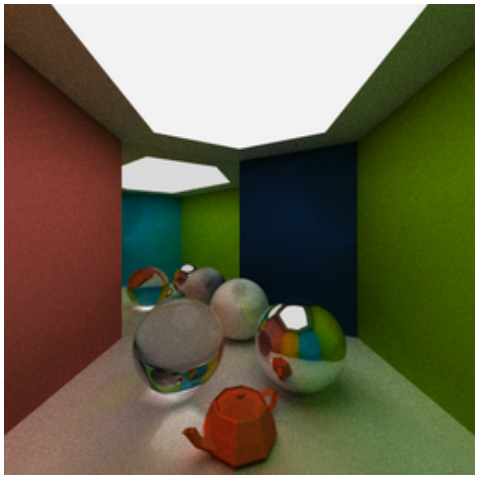
Supersamples	Rendered	Time Taken
16	See Fig. 18	50.3 sec
49	See Fig. 13	2 min 42 sec
196	See Fig. 19	10 min 24 sec

Figure 18: Render of the scene 512x512 at 16 samples per pixel, 1 shadow ray and 7 max ray bounces.



In the figure above, the scene initially seems very dark. This is expected, since not enough indirect illumination has been spread through the scene yet. It also has to do with how our raytracer does normalization of the radiance in each pixel to integer colors, something like *tone mapping* might fix this.

Figure 19: Render of the scene 512x512 at 196 samples per pixel, 1 shadow ray and 7 max ray bounces.



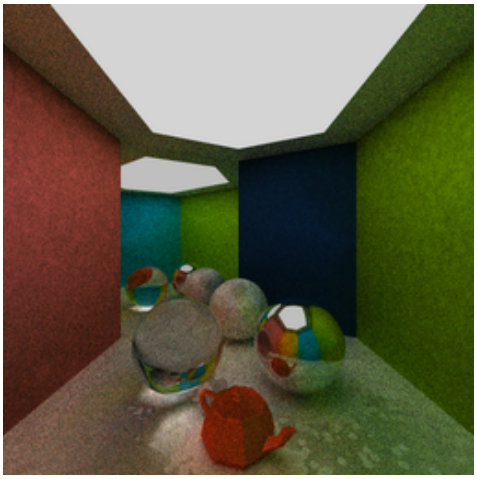
As expected the final results give less noise with more samples but at the cost of additional computation time. This also reduces aliasing and thus the jaggies which were presented earlier. The aliasing near the sources of light is especially bad at the start because it doesn't fall off as smoothly as the other surfaces in the scene. That is, it's a very sharp change in color, and it thus takes a lot of samples to blend it with the nearby colors.

Photon Mapping

The following table shows how using a photon map affects the visual and computational results of the render with a fixed number of photons but different radiance estimation bound. This photon map implementation does not use k-nn photon gathering but instead uses a fixed size sphere around each point of interest.

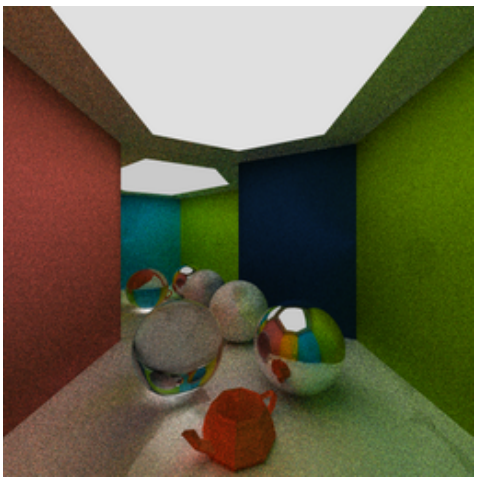
Radius	Rendered	Time Taken
0.1	See Fig. 20	51.5 s
0.7	See Fig. 21	129.6 s

Figure 20: Render of the scene 512x512 at 49 samples per pixel, 8 shadow rays and 7 max ray bounces using a photon map with 1 000 000 photons and a radiance estimation radius of 0.1.



Stray shadow-photons causes the result to look spotty since the photon map and path tracer doesn't agree with the amount of radiance for a given point.

Figure 21: Render of the scene 512x512 at 49 samples per pixel, 8 shadow rays and 7 max ray bounces using a photon map with 1 000 000 photons and a radiance estimation radius of 0.7.



With a larger sphere of interest the result takes significantly longer to compute but the result doesn't become as spotty as the previous configuration.

4 Discussion and Outlook

A couple of interesting findings can be observed when looking closely at the results. Some of these pertain to the implementation details, while others are indirectly caused by how the path tracer interacts with the raytracer's photon mapping extension.

The path tracer produces visible and crisp caustics without significant extra computational cost. This result is interesting since caustics generally are computationally expensive in regular path tracing. We believe that this could be the effect of us having very simple geometry in the scene, and how the objects are placed in it. To improve the caustics in the general case, a caustic photon map can be used instead, as proposed by *Jensen* [6]. This would have the effect of speeding-up rendering while allowing for a reduced resolution of the global photon map.

Counter-intuitively, the photon map causes the render time to increase. Our theory is that this is caused by the fact that the path tracer only uses one shadow ray and doesn't estimate indirect light. Meaning, the amount of work being done in the direct light calculations of the path tracer aren't enough to motivate the photon map optimization. The photon map still uses an expensive nearest neighbor look-up. This result could be counter-acted by reducing the look-up time of the photon map or by removing supersampling (meaning we have to increase the amount of shadow ray samples as well).

"Spotty" regions can be observed when looking at the rendered image for the photon map with a small estimation radius. These regions are caused by stray shadow photons from the photons reflected in the scene. A possible way to improve the result would be to increase the size of the estimation radius or thresholding how many shadow photons an estimation region can contain before shadow rays are used. However, this increases computation time.

The number of shadow rays does not affect the result significantly since the renderer uses supersampling, effectively doing the necessary Monte Carlo integration of the area in the camera. To increase the relevance of the shadow ray parameter, supersampling could be disabled or reduced to a degree.

While the monte carlo raytracer already gives pretty photorealistic results, it is not perfect. Below we'll provide an outline of potential improvements.

Both BRDFs in `mcrt` are for diffuse surfaces, but there are other BRDFs for specular surfaces as well, such as the *Torrance-Sparrow* [12] and *Cook-Torrance* [3] models. It would be interesting to integrate these into our renderer too for completeness.

Also, in the raytracer, all meshes are wrapped around a sphere for eliminating low-effort intersection tests. Usually, for meshes with many more triangles, it's insufficient, and causes long render times. Instead, a *bounding volume hierarchy* should have been used, potentially reducing it to $O(\log n)$ tests.

In our photon mapping implementation, we only only support estimation of the direct light contributions. In *Jensen's* [6] original paper, indirect light is also accounted for in the *global photon map*, and also has a high-resolution *caustics photon map*. As we discussed before, this might have made photon mapping more useful and actually sped things up.

Additional effects such as *depth of field* and *motion blur* can be approximated with a *distribution raytracer* such as those presented in *Cook et al.* [2]. The amount of additional work to integrate these effects into `mcrt` are not huge, and give nice results.

Something which would be interesting to implement is a *spectral raytracing* extension, with support for e.g. *black-body radiation* like shown in *Zhang et al.* [1]. Instead of using RGB-channels for storing radiance, the ray wavelength could have been stored instead. Giving a more physically plausible render, such as radiance affected by the temperature, chromatic aberrations, and thin film light interference.

Perhaps some sort of *HDR* (High-Dynamic Range) technique, like *tone mapping*, would solve the problem we get when our raytracer initially has with dark scenes, before reaching full convergence.

For producing realistic scenes, realistic materials are needed, and *textures* would have been a desirable feature to have. A *texture mapping* technique for triangle meshes isn't that hard to implement, but special care is needed when dealing with texture filtering. Some `OpenGL` things we take for granted...

Finally, parallelization support is achieved with `OpenMP`, but having support for additional hardware accelerators would have been desirable. `OpenMPI` would have been nice to have for speeding up rendering in computer clusters. The other more important addition which would have made a big difference is to support massively-parallel graphics hardware via APIs like `OpenCL` or `CUDA`, since the raytracing problem easily adapts to MC raytracing.

If the reader of this report would like to add these or other features to `mcrt`, feel free to open a PR :)

References

[1] J. C. Cecilia Zhang, Ashwinlal Sreelal. *Spectral ray tracing*. 2016. [Online in November 2017].

[2] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 137–145. ACM, 1984.

[3] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982.

[4] Y. Fujii. *A tiny improvement of the Oren-Nayar reflectance model*. 2015. [Online 2017].

[5] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 213–222. ACM, 1984.

[6] H. W. Jensen. Global illumination using photon maps. *Rendering methods*, 96:21–30, 1996.

[7] J. T. Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[8] J. H. Lambert. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. 1760.

[9] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[10] F. E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied optics*, 4(7):767–775, 1965.

[11] M. Oren and S. K. Nayar. Generalization of lambert's reflectance model. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 239–246. ACM, 1994.

[12] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Josa*, 57(9):1105–1114, 1967.

[13] T. Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, page 4. ACM, 1980.