Advanced Global Illumination using Monte Carlo Methods

Linus Mossberg

Linköpings Universitet

Abstract

This report presents and compares two Monte Carlo based global illumination methods as well as methods of processing the resulting raw image intensities to mimic physical cameras. The first rendering method presented is the Path Tracing method, which achieves unbiased global illumination by approximating the rendering equation by means of Monte Carlo integration. A probabilistic single-branching path branching method is introduced which randomly selects a new path from several differently weighted paths at each ray-surface intersection in order to replicate a wide range of physical materials while preventing the ray-tree from growing exponentially. The second method presented is the Photon Mapping method, which achieves biased global illumination in two passes by first generating photon maps and then utilizing these in the main rendering pass. A Monte Carlo based method of increasing the relative amount of caustic information during the first pass in order to resolve sharper caustics is introduced. Methods of applying histogram-based auto exposure and a filmic transfer function to the rendered raw image intensities is presented to visually improve the results.

1 INTRODUCTION

There are many different methods of generating photorealistic images in computer graphics, and what they all have in common is that they attempt to solve the rendering equation. A simplified form of the rendering equation that omits time and light wavelength is defined as follows:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega_i} f_r(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos\theta_i \, d\omega_i$$

This equation describes the radiance L_o that leaves a point x in the direction ω_o. The point x could for example be a point in a scene visible from a camera and ω_o the direction from that point to a location on the sensor of the camera. The term L_e in the rendering equation is simply the radiance that is being emitted from the point x in the direction ω_o, while the

integral describes the portion of all incoming radiance L_i that is being reflected from the point x in the direction ω_o. To solve the integral, the rendering equation must be solved for all points visible from the point x in the directions ω_i that leads back to it. To solve the rendering equation at those points, the rendering equation must be solved for all points visible from them in the directions that leads back to them and so on.

This continues forever in most cases, which means that the rendering equation usually isn't solvable. The job of a photorealistic rendering method is therefore to approximate the rendering equation rather than solving it. If this is done in such a way that the method eventually would converge to the correct solution given unlimited time, the method is said to be unbiased. Otherwise, the method is said to be biased. This work explores and implements one photorealistic rendering method from each of these two categories, one unbiased method called *Monte Carlo Path Tracing* and one biased method called *Photon Mapping*.

2 BACKGROUND

The following sections describes some of the theory behind the renderer and how this has been implemented.

1.1 PATH TRACING

Path Tracing is performed by tracing rays from a camera and out into a scene until the ray intersects with a point. At this point, the rendering equation is used to evaluate the radiance leaving the point in the direction opposite of the ray direction, which typically involves tracing new rays from this point to evaluate the integral over all incoming directions. This type of path tracing works backwards compared to how light is transported in the real world, and because of this the rays that originate from the camera are said to carry *importance* rather than radiance. This importance is attenuated the same way as radiance at intersection points and it is used to weigh the contribution of the radiance that is collected along the path in order to evaluate how much of it would reach the camera.

2.1.1 Monte Carlo Integration

As mentioned previously, the integral from the rendering equation can't be solved and a method of approximating it is required. Monte Carlo Integration is a technique of doing this that works by randomly selecting one direction to sample according to some random distribution. The contribution of that direction is then weighted by dividing it with the probability of selecting that direction, which is called the *probability density function* (PDF) of the random distribution. The result of this is an estimate of the entire integral, but this estimate is most likely not very good since only one direction has contributed to it. The estimate can however be improved by sampling other random directions in the same way and then computing the average of these estimates. This can be repeated indefinitely and the estimate will improve for each new sample, and it can be shown that this eventually converges to the correct solution given unlimited samples [1].

The random distribution used when picking a random direction should ideally be distributed similarly to the integral in the rendering equation. This includes the cosine of the incident direction and the *bidirectional reflectance distribution function* (BRDF, denoted f_r), which is used to attenuate the reflected importance or radiance. This is because we want to increase the probability of selecting directions that will contribute more rather than frequently selecting directions whose contribution would be close to eliminated by the reflectance function and the cosine term.

For diffusely reflecting surfaces, the *cosine-weighted hemispherical distribution* is well suited since directions with a steeper incidence contribute less due to the cosine term in the integral. This distribution has a probability density function $\cos\theta_i/\pi$ [2] and the Monte Carlo estimator for diffuse reflections for a single sample therefore becomes:

$$L_r \approx \frac{f_r(x,\omega_o,\omega_i)\cdot L_i\cdot\cos\theta_i}{PDF} = \frac{f_r(x,\omega_o,\omega_i)\cdot L_i\cdot\cos\theta_i}{\frac{\cos\theta_i}{\pi}} = f_r(x,\omega_o,\omega_i)\cdot L_i\cdot\pi$$

where L_r is the integral from the rendering equation, i.e. the portion of all incoming radiance (or importance) that is reflected in direction ω_o. For the perfect specular integral, there's only one possible direction that can contribute and it can be simplified to just include a specular reflectance factor multiplied with the incoming radiance from that single possible direction [1].

2.1.2 Path Branching

When a ray hits a surface point, the radiance (or importance) that the ray carries is either absorbed or distributed amongst several new possible rays depending on the material properties at the surface point. Absorption of the incoming ray is done with a method called Russian roulette path termination[1]. This is performed by absorbing (terminating) rays with a probability A and casting new rays otherwise. To compensate for the lost energy when this is done, the radiance of the new rays must be divided by the non-absorption probability $1-A$.

One way of distributing the incoming radiance into new possible rays is to split the path by casting up to three new rays from the intersection point: one specularly reflected, one specularly refracted and one diffusely reflected. The new radiance of these rays must be weighted so that the energy is conserved and adds up to be equal to the radiance of the incoming ray (equal before the radiance of the new rays have been attenuated). This is done by multiplying the radiance of the new rays with different weights that depend on the material, the medium and the incident direction of the incoming ray at the intersection point.

The weight of the specularly reflected ray is determined by the Fresnel factor [1] and is denoted R. The weight of the refracted ray is the remainder of Fresnel factor multiplied by the transparency factor[2] denoted T. Finally, the weight of the diffusely reflected ray is the

[1] Russian roulette path termination does not model some hypothetical absolute light absorption phenomenon from nature, it's just used to limit the ray depth while still conserving the energy, i.e. without introducing bias.
[2] The transparency factor T is constant for all incident directions and is defined in the material properties. It is not to be confused with the remainder of the Fresnel factor $(1-R)$.

remainder of the Fresnel factor multiplied with the remainder of the transparency factor. This can be expressed as the following weights:

$$\begin{aligned} w_{reflect} &= R \\ w_{refract} &= (1-R)\cdot T \\ w_{diffuse} &= (1-R)\cdot(1-T) \end{aligned} \qquad (1)$$

These weights add up to 1 and energy is therefore conserved if they are used to weight the radiance of the new rays:

$$R + (1-R)\cdot T + (1-R)\cdot(1-T) = R + T - RT + 1 - T - R + RT = 1$$

The benefit of doing path branching this way is that surfaces can be any combination of specularly reflective, specularly refractive and diffusely reflecting which makes it possible to define many different types of materials. Glossy paint for example reflects some of the incoming radiance specularly while the rest is reflected diffusely. Another example is a water filled plastic balloon which reflects some of the incoming radiance specularly while the remaining radiance is either refracted through the surface or reflected diffusely depending on the transparency of the balloon.

2.1.2.1 Probabilistic Single-Branching Path

Casting up to three new rays at each intersection point is not ideal as this will cause the ray-tree or call-stack to grow exponentially at the rate 3^{depth}. This would quickly cause the program to either run out of memory or the call stack to overflow and attempting to prevent this by using a restrictive maximum ray depth would introduce bias.

A better way of performing path branching is to probabilistically select a single new path out of the three possible ones and use the weights defined in (1) as the probability of that path being selected. This way, the ray-tree or call stack would grow at a more manageable linear rate and using Russian-roulette path termination would be enough to ensure that reaching a maximum ray depth would be very unlikely.

To compensate for the lost energy when only contribution from a single path is included, the radiance must be divided by the probability of selecting that path like in Russian roulette. Since this probability is defined to be the same as the path weights in (1), the radiance of the reflected ray is both multiplied and divided by the path weights which means that the path weights are eliminated from the calculation. A flow chart visualizing the path branching process and the accumulated probabilities of each path is shown in Figure 1. The probability of absorption and non-absorption have also been included, but the sum of these probabilities still add up to 1. Note also that the accumulated probability carries back to the beginning after each iteration.

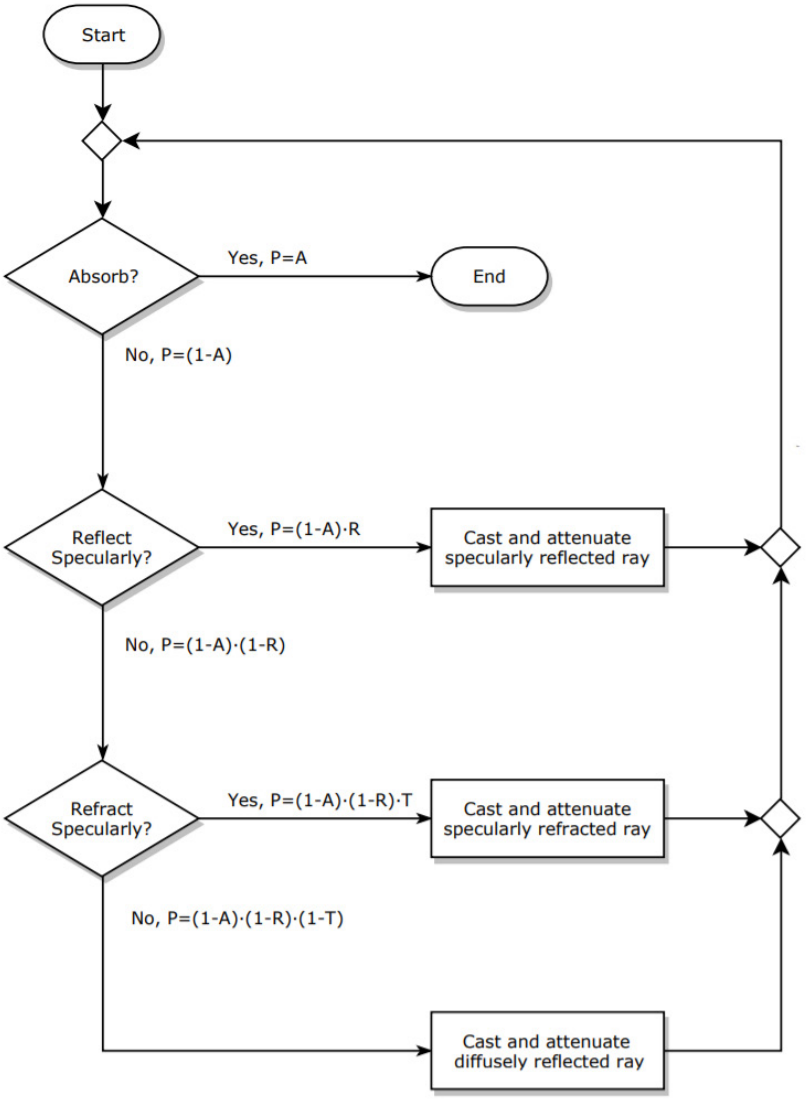

Figure 1 – Flow chart of path branching with path probabilities

2.1.3 Explicit Direct Light Sampling

The only way for radiance to feed into an importance path is for a surface with a non-zero emissive material component to contribute to that path. With the most naive brute-force path tracing method, this can only happen if an importance ray happens to intersect with an emissive surface somewhere along the path. Occurrences like this are random and they grow less and less likely the smaller the light sources are relative to the rest of the scene.

This can result in a large portion of the importance paths contributing nothing to the estimate, which significantly increases the convergence time of the renderer.

A better way of doing this is to explicitly sample the contribution from all light sources in the scene for every diffuse intersection point along the importance path. This is done by casting a ray from the diffuse intersection point to a randomly sampled point on each light source. These rays are called *shadow rays* and they are used to determine if the path to each light source is obstructed or not. If the ray can reach the light that corresponds to it, its contribution to the direct radiance is included. As usual, the potential direct light contribution of each light must be divided by the probability density function of sampling the

point on its surface to provide an estimate of the entire light surface. If the point was sampled using a uniform random distribution of the surface, then this probability density function is one divided by the surface area of the light [1].

To transform this probability density function to the solid angle probability density function at the diffuse intersection point from which the incoming direct light contribution is being sampled, it must be weighted by multiplying it with the factor:

$$\frac{distance^2}{\cos(\theta_{o,light})}$$

where $\theta_{o,light}$ is the outgoing angle from the light source and $distance$ is the distance from the intersection point to the sampled point on the light source [1]. The direct radiance contribution is therefore weighted by the factor:

$$\frac{1}{\frac{1}{area} \cdot \frac{distance^2}{\cos(\theta_{o,light})}} = \frac{area \cdot \cos(\theta_{o,light})}{distance^2}$$

In keeping with the Monte Carlo method, the program also randomly selects a single light source to sample rather than sampling all of them. The contribution is then divided with the probability of selecting that light source to create an estimate of the direct light contribution from all light sources. This reduces the variation in render times when using a constant number of samples per pixel, but varying numbers of light sources in the same scene. Figure 2 shows a comparison between using explicit direct light sampling like this and using the naive path tracing method. The difference would be even more pronounced with a smaller light source.

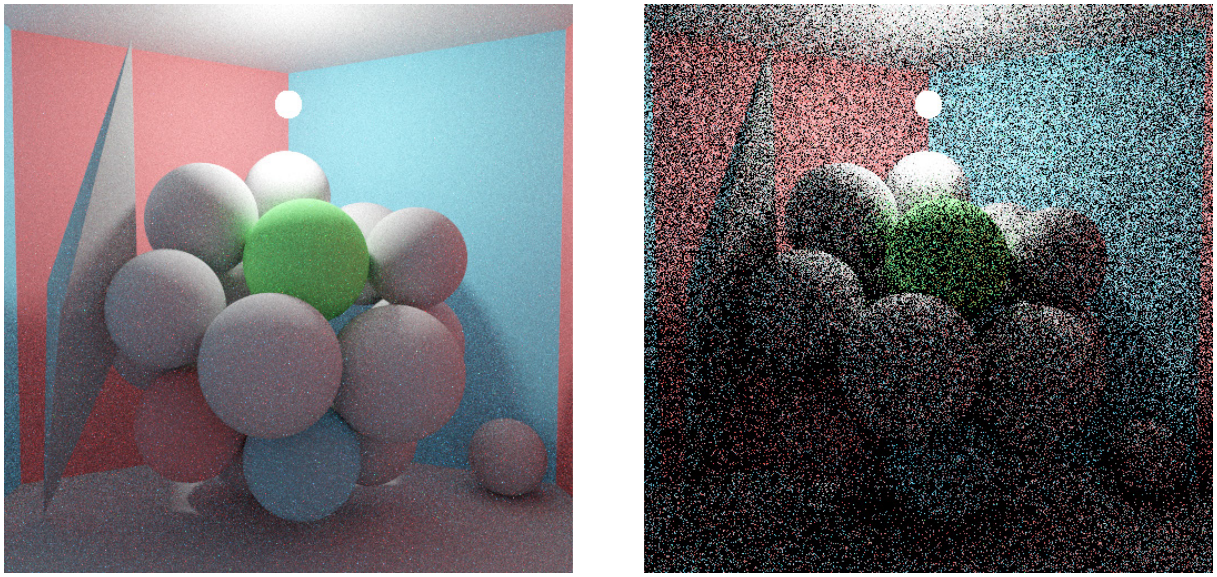

Figure 2 – Images rendered for 1 minute each, with explicit direct light sampling and 121 SPP on the left and with no explicit direct light sampling and 196 SPP on the right.

2.2 CAMERA

The camera orientation is defined by its *eye-position*, *forward-vector* and *up-vector*. The directional vectors are unit vectors that are orthogonal to each other, where the forward-vector points from the eye position to the center of the image plane[3], while the up-vector points from the center of the image plane to the first pixel row in the image. A third vector that is orthogonal to the previous two vectors called the *left-vector* is generated using the cross product $left = up \times forward$. This vector points from the center of the image plane to the first pixel column in the image. Together, these vectors form a right-handed orthonormal basis that can be used to define an image plane located relative to the eye position.

The camera is also defined by its focal length f, its sensor width s_w, its image width I_w and its image height I_h. The focal length defines the distance from the eye position to the image plane in meters, while the sensor width defines the width of the image plane in meters. The image width and height are defined in terms of pixels. The coordinate (x, y) in metric units in image plane space, which is the 2-dimensional space with origin at the center of the image plane with up and left vectors as basis vectors, can now be given by:

$$\begin{cases} x = \frac{s_w}{I_w} \cdot \left(\frac{I_w}{2} - c\right) \\ y = \frac{s_w}{I_w} \cdot \left(\frac{I_h}{2} - r\right) \end{cases} \quad (2)$$

where (c, r) is the pixel coordinates of the image pixel in terms of pixel column and row, with $(0,0)$ located in the upper left corner of the upper left pixel in the image[4]. Using only the sensor width and not the sensor height is possible with the assumption that pixels are square, because the image ratio is known from the image dimensions. Locating the position p on the image plane in global coordinates can now be done with the following formula:

$$p(x, y, f) = eye + f \cdot forward + x \cdot left + y \cdot up$$

This can also be expressed using a transformation matrix:

$$p(x, y, f) = \begin{bmatrix} left_x & up_x & forward_x & eye_x \\ left_y & up_y & forward_y & eye_y \\ left_z & up_z & forward_z & eye_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix}$$

The focal length and the sensor width properties of the camera are commonly combined into a field of view angle property. This can be more intuitive but defining the camera by

[3] A directional unit vector technically does not point from a point to another point (other than from the origin to the unit sphere), but using this language makes the purpose of the vectors in the context that they are used easier to visualize.

[4] Pixel coordinates (c, r) are typically the integer indices of the pixel, but these can be real numbers in (2) to locate points on pixels.

using the same properties that commonly are used when describing real camera bodies and lenses is better suited for photorealistic applications.

Now that we have a method to transform coordinates from image space to global space, creating rays that can be cast through any point on the image plane is simple. The ray origin is defined as the eye position, while the ray direction is defined as:

$$norm(p(x, y, f) - eye) = norm(f \cdot forward + x \cdot left + y \cdot up)$$

where $norm(v) = v/|v|$. This is simply the normalized version of $p(x, y, f)$ with the origin (eye) eliminated, leaving us with only the directional part.

2.2.1 Supersampling and ray randomization

Several importance paths are sampled from each pixel in the path tracer, and their contributions are averaged to produce a better estimate of the rendering equation for the given pixel. The ray directions of the rays leaving the camera pixel for these paths could simply be defined as the ray that passes through the center of the pixel by always adding 0.5 to each element of the pixel coordinate (c, r) in equation (2). This would however produce aliasing and it would result in a bad estimate of the total radiance falling in on the entire pixel area since only one point would be sampled. A better solution is to sample a random point on the pixel for each ray, which can be done by generating and adding a uniformly distributed random number in the range $[0,1[$ to each element of the pixel coordinate (c, r) in equation (2).

This can be improved further by using *stratified sampling.* With stratified sampling, the pixel is subdivided into several smaller subpixels and a single uniformly sampled point is selected from each of these subpixels [1]. This adds an additional subpixel offset term and a random number in the range $[0, sub\ pixel\ width[$ to each element of the pixel coordinate (c, r) in equation (2). This guarantees that an even distribution of points are sampled for each pixel which reduces variance, and it's therefore the method used in the renderer. With this method, the number of samples per pixel must be a perfect square number. This is ensured by using a *square root samples per pixel* (*SQRTSPP*) integer to define the number of samples per pixel to use in the render settings configuration.

2.2.2 Auto Exposure

In the real world, the general light intensities between different scenes can differ by several orders of magnitude. Imaging systems like digital cameras and our eyes are also only able to resolve a range of light intensities at a time, which can result in problems with over- or underexposure when a camera configured for one scene moves to another. To solve this, both our eyes and modern digital cameras can respond to different light intensities and dynamically adjust the aperture, exposure time and sensor/retinal sensitivity (gain) accordingly.

Since the lighting and reflectance models in the renderer are physically based, the variations in light intensities from scene to scene will differ in the same way. Having the ability to automatically expose the image is therefore useful in the renderer as well. Because the full precision image in the renderer has a very high dynamic range without any electrical noise or

film grain, this can be done by applying gain to the image by simply multiplying each pixel channel with an exposure factor E_f. The job of the auto exposure method is therefore to find this exposure factor for a given full precision image.

A simple method of doing this is to iterate through each pixel of the image to find the pixel that has the highest absolute intensity I_{max} and then setting the exposure factor to be $E_f = \frac{1}{I_{max}}$. This would ensure that no resulting pixel is assigned an absolute intensity above 1. All it takes for this method to produce very bad results however is that one pixel has a much higher absolute intensity than the rest of the pixels. This is very common and can happen if there's a light source visible directly or through a specular reflection, or even if there's a single pixel with high intensity produced by a stray caustic ray. In these cases, it's better to let a certain percentage of the highest intensity pixels be clipped.

This can be achieved by using a histogram. In a histogram, N bins b_i are assigned an occurrence counter and a range r_i of absolute pixel intensities defined as:

$$r_i = \left](\mathrm{i}-1)\cdot\frac{I_{max}}{N}, i\cdot\frac{I_{max}}{N}\right] \; for\ i\ in\ integer\ range\ [1,N]$$

Each pixel in the image is then iterated through and the occurrence counter of a bin b_i is incremented when the absolute pixel intensity falls within its absolute pixel intensity range r_i. Once the histogram has been constructed this way, the histogram bins are iterated through in reverse order and the value of each occurrence counter is accumulated to keep track of the total number of pixels that have been accounted for. Once this number of accounted for pixels reaches a certain percentage p of the total number of pixels in the image, the iteration stops and the intensity level that this bin b_i represents is calculated using:

$$I_p = \frac{i}{N}\cdot I_{max}$$

The exposure factor is then defined as $E_f = \frac{1}{I_p}$. The result of this is that p percent of the highest intensity pixels are clipped (intensity above 1), while the rest are in the range $[0,1]$. This method exposes different scenes very consistently if enough bins are used. The program uses $N = 2^{16}$ and $p = 15\%$ which works well. The number of bins could be reduced by converting the absolute intensity values to a logarithmic scale (*stops*) [3], but the memory reduction and potential gain in speed would be negligible relative to the other steps in the renderer.

2.2.3 Tonemapping

Once exposure adjustments have been made to the image, the pixel channels are usually gamma corrected to compensate for the gamma curve of the monitor that the image will be displayed on and then truncated to bytes before writing it to an image file for display. Real cameras transform the raw light intensities that reach the image plane before this however, either electronically in digital cameras or chemically in film cameras. This is often a side effect of how film and electronic sensors work rather than something intentional, but it can transform the image from representing intensities linearly to something that might favour

certain intensities more than others. This can potentially produce an image that resolves a higher dynamic range and make the image look more visually pleasing [4]. This is especially the case for film which is part of the reason why it is still used today.

Emulating this effect in digitally rendered images is called tonemapping and performing this in the renderer can improve the photorealism of the results. This is done by applying a transfer function to the raw pixel intensities, and one such transfer function that aims to specifically emulate film was developed by John Hable for the video game Uncharted 2 [5]. This operator was implemented in the renderer and a comparison between John Hable's filmic transfer function and the linear transfer function can be seen in Figure 3.

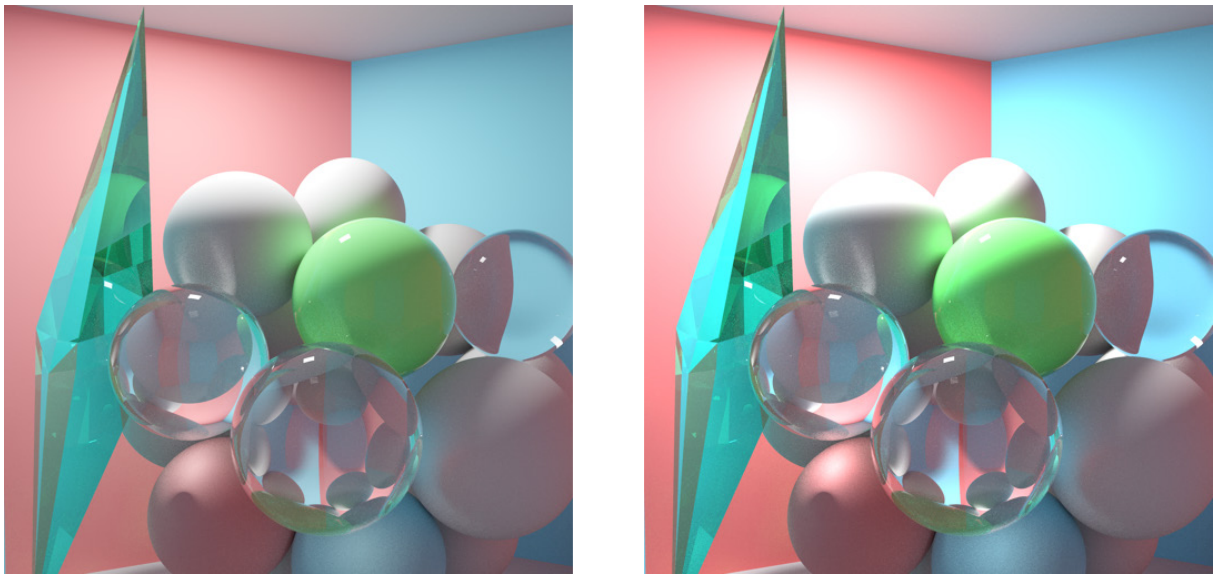

Figure 3 – Image produced with John Hable's filmic transfer function on the left and image produced with a linear transfer function on the right. The original full precision image with raw intensities is the same for both images.

2.3 PHOTON MAPPING

Photon mapping is a 2-pass rendering method which similarly to path tracing also uses Monte Carlo ray tracing techniques. The first pass is called the photon scattering pass and it does not yet involve any camera. This pass is performed to generate data that can be utilized in the second pass to render the image with a camera, which is called the main rendering pass. Both steps work very similarly to path tracing and they use the same path branching method as described earlier.

2.3.1 Photon Scattering Pass

The first step of the photon scattering pass is to cast a large number of rays from the light sources in the scene. This is done by sampling random positions on the surfaces of the light sources and then casting rays in random direction into the scene. These rays are then scattered throughout the scene by using the same path branching methods as described earlier. One major difference however is that the rays now carry flux rather than radiance, and the rays are assigned the flux of the light source that they originated from divided by the number of rays emitted from that light source. This flux is however attenuated in the same way as radiance at intersection points.

Once one of these rays are evaluated to be diffusely reflected at an intersection point, a photon is created. This photon is assigned the flux and incident direction of the ray that spawned it, as well as the position where the intersection happened. The photon is then added to one of three photon maps depending on the origin of the ray that spawned it. If the ray came directly from a light source, then the photon is added to the *direct photon map*. If the ray came from a diffuse reflection, then the ray is added to the *indirect photon map.* Finally, if the ray came from a specular reflection or refraction, then the ray is added to the *caustic photon map*. There's also a fourth photon map called the *shadow photon map* that stores special direction- and flux-less photons called *shadow photons*. These are spawned at all subsequent diffuse intersection points when a ray coming directly from the light source intersects with a surface that is evaluated to result in either a diffuse or specular reflection.

2.3.1.1 Caustic Photon Factor

As will become clear later in the second pass, a lot of caustic photons are needed for good results. Simply casting more rays from the light sources would not work since the computer memory quickly would fill up with all types of photons and not just caustic photons. One way of solving this is to explicitly cast rays from light sources towards specular surfaces by casting them from randomly sampled positions on the lights to randomly sampled positions on the specular surfaces. This would however result in situations where rays are cast into impossible directions to points that they can't possibly reach if the light source and specular surface are spheres for example. There are ways to randomly sample spheres at reachable points and compute the probability density function dynamically when doing so [1], but a simpler brute force method was implemented instead. This method works by introducing a configurable caustic photon factor cpf. The number of photons emitted from light sources is then increased by this factor, and to reduce the number of non-caustic photons stored they are simply rejected with a probability $1/cpf$. To compensate for the lost energy, the flux of the non-caustic photons that are stored is then multiplied with the factor cpf. This is a very inefficient method and a lot of direct and indirect photons are computed but never stored, but the photon scattering step is very fast and can normally, without rejection, create enough photons to fill all available memory on most systems within a minute.

2.3.1.2 Octree

A data structure that allows for fast area searches is needed to store the photons. The chosen data structure is the *Octree*, mostly because it's easy to implement. Instead of recursively dividing the octree until each photon is contained within their own unique leaf node, photons are instead added to dynamic containers stored at each leaf node. Once this container reaches a certain maximum size, the leaf node stops being a leaf and is subdivided into 8 new octants, and the photons in the container is redistributed across these new leaves. Selecting this max number of photons per octree leaf is not trivial and a test was conducted by doing radius searches in random positions in a scene that have different numbers of generated photons. This was performed for max number of stored photons per octree leaf ranging from 1 to 10000 and the result can be seen in Figure 4. This data took a long time to generate and what can be gauged from it is that a value of about 190 works well in most cases. The wide variation in the graph representing $1.5 \cdot 10^7$ photons is most likely

caused by hardware specific things like size of CPU cache and may look different for different hardware.

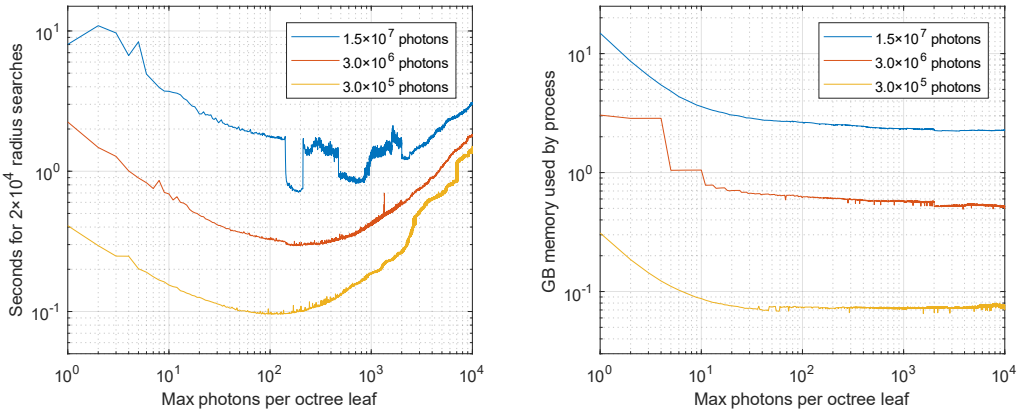

Figure 4 – Graphs of max photons per octree leaf versus radius search time and memory usage.

The octree has recursive box and radius search methods implemented, the later of which is used in photon mapping.

2.3.2 Main Rendering Pass

The main rendering pass works the same as for path tracing but with a few differences. First, in order to utilize the photon maps, a method of estimating the radiance leaving a point x in the direction w_o from the photon maps is needed. This is done by first performing a radius search with a specified radius r in the octree. The reflected part of the radiance is then estimated with:

$$L_r \approx \frac{1}{\pi r^2} \sum_{p=1}^{N} f_r(x, w_o, w_{i,p}) \cdot \Phi_p$$

where N is the number of returned photons within the radius r, and $w_{i,p}$ and Φ_p is the incident direction and flux respectively of photon p [6]. The differences in the rendering pass compared to path tracing is then:

- If a ray hits a surface and the new path evaluates to diffuse, estimate caustic radiance from the caustic photon map.
- If a ray originating from a diffuse reflection hits a surface and the new path evaluates to diffuse, check if there are shadow photons at this intersection. If there are, carry on with the diffuse reflection as usual. Otherwise if there are no shadow photons, evaluate the global radiance at this point by estimating the radiance from all photon maps and terminate path.
- If a ray originating from anything else than the camera or a specular interaction hits a surface and the new path evaluates to a specular interaction, terminate the path to not produce caustics twice.

This ensures that the direct and indirect photon maps never are directly visualized, there's always at least one diffuse reflection in between. Diffusely reflecting when there are shadow photons present must be done to reduce artefacts in corners and such where each diffuse

reflection is more likely to end up close to the same position, and therefore estimate the same direct and indirect radiance from the photon maps. The caustic radiance is always evaluated for diffuse paths, even for the first ray. This means that it will be visualized directly unlike direct and indirect photons. This is the reason why increasing the number of caustic photons relative to other photon types is required for good results.

With the method described here, the direct and indirect photon maps are used at the same times and separating them rather than combining them into one photon map has no utility. There are however ways to utilize them differently. One such way is to always evaluate direct radiance by first checking if the intersection point has shadow photons present but no direct photons present. If this is the case, then the point is likely in complete shadow and zero radiance can be returned without casting a shadow ray to check. This can be done even from the first ray without introducing much visible artefacts. This method is implemented but not used, because radius searches in the direct photon map is about as fast as casting one shadow ray which defeats the purpose. This might not be the case with a faster data structure however.

3 Results and Discussion

The source code, scene files and more renders can be found on github:

https://github.com/linusmossberg/monte-carlo-ray-tracer

The scene used for the results presented here consists of a hexagonal room filled with different object that have different materials. The room itself consists of 12 vertices which are used to form 18 triangles, 4 of which are used for the floor and ceiling each and 2 for each of the 6 walls. The axis aligned bounding box dimensions of the room is 16x10x12 meters (in X, Y and Z order, where Y is up). The floor and ceiling have been assigned the default material, which in the scene file is defined to be a *Lambertian* diffuse material with a reflectance of 0.5 (linear) for each channel. The walls have been assigned two materials which alternate, one Lambertian diffuse material with light blue reflectance and one with light red reflectance.

The light sources in the scene are two connected triangles that forms a 1 square meter square, which is located 1 decimeter below the roof. The object has been assigned a material with an emittance of 1000 flux for each channel, which means that each triangle emits 500 flux for each channel. This material also has a Lambertian diffuse component with a reflectance of 0.8 (linear) for each channel.

The objects in the room are 16 spheres with different radii and materials, and one crystal object which consists of 5 vertices and 6 triangles (two mirrored tetrahedra without “bottom” triangle). The crystal is 9 meters tall and has been assigned a fully transparent material with an index of refraction (IOR) of 2 and a green-blue specular reflectance. One of the spheres has been assigned a light green reflectance, white specular reflectance and an IOR of 1.460 to mimic plastic. This makes the material glossy based on the Fresnel factor when the scene IOR is set to be anything but the same. The material has no roughness,

which makes the diffuse part Lambertian. Three of the spheres have been assigned fully transparent materials with a specular reflectance of 1 for each channel. The two largest of these have been assigned the same material with an IOR of 2.4, while the material of the smaller one has an IOR of 1.17. These values were chosen because of the caustic patterns they produce with a scene IOR of 1. One of the spheres have been assigned a material with blue reflectance and a roughness of 10, which makes it an *Oren-Nayar* diffuse material. There's also another sphere with a red Oren-Nayar diffuse material with the same roughness. The smallest sphere in the room has been assigned a specular reflectance of 1 for each channel and is configured to be a perfect mirror material, which means that its Fresnel factor is always 1. This can be interpreted as the material having an infinite relative IOR. The remaining 9 spheres have been assigned the scene default material.

There are several cameras in the scene but only one has been used here to easier visualize the difference between methods and settings. This camera is located at coordinates *(-2, 0, 0)* and has a focal length of 23 millimeters and a sensor width of 35 millimeters. The camera is set to look at the coordinate *(13, -0.55, 0)*, which makes the program compute the forward and up vector required to do so (with roll anchored such that Y-axis defines the scenes up direction).

More details about the scene can be seen in the scene file *scenes/hexagon_room.json* in the complementary material or on github.

3.1 Path Traced Renders

The image rendered with path tracing and scene of IOR 1 can be seen in Figure 4. The image was rendered in 3840x2880 resolution with 4096 samples per pixel (64 SQRTSPP) in 24 hours, 32 minutes and 13 seconds. This image was later downscaled to 1280x960 resolution with Lanczos resampling, which means that the image seen here actually has 36864 samples per pixels. One interesting thing to note in this image is the caustic patterns on the spheres above the transparent spheres. These are caused by rays from the light refracting into the transparent spheres, at which point some of the rays have an angle that is too steep to break through the IOR 2.4 to 1.0 interface. They are instead internally reflected any number of times until they reflect back up at a more perpendicular angle to the top of the sphere, at which point they are able to break through and escape. The crystal produces more chaotic and intricate caustic patterns which are produced by a combination of refraction and internal reflection as well. The reflected and refracted rays are attenuated by the green-blue specular color of the crystal material, which makes the caustics seen on the floor the same color.

The smaller transparent sphere has an IOR set such that it focuses the rays from the light to resolve an image of the light in its caustics on the wall, which can be seen in the lower right in the image. The green glossy sphere becomes more and more specularly reflective the steeper the viewing angle which is caused by the Fresnel factor, and the diffuse portion of its reflection can be seen bleeding on to the neighbouring gray sphere. Other interesting things to note is the Oren-Nayar spheres which has less of a falloff with steeper angle, and the blue

and red caustics patterns on the gray sphere closest to the camera produced by diffusely reflected light from the walls that are refracted by the transparent sphere.

The image rendered with path tracing and a scene IOR of 1.75 can be seen in Figure 5. This was rendered with the same setting and for a similar render time as the previous image. This scene IOR can be interpreted as the scene being suspended in flint glass rather than air, and this changes the angle of refraction and Fresnel factor of objects in the scene. The smaller transparent sphere has now started to specularly reflect rays arriving at steeper incident angles since these can't break through due to the critical angle. The rest of the transparent objects have become less specularly reflective because more rays are now able to pass through due to the decreased relative index of refraction, and their caustic patterns have changed because they now refract rays at a lesser angle. A more subtle effect can be seen on the green sphere, which now isn't as specularly reflective and more rays are now diffusely reflected by it instead.

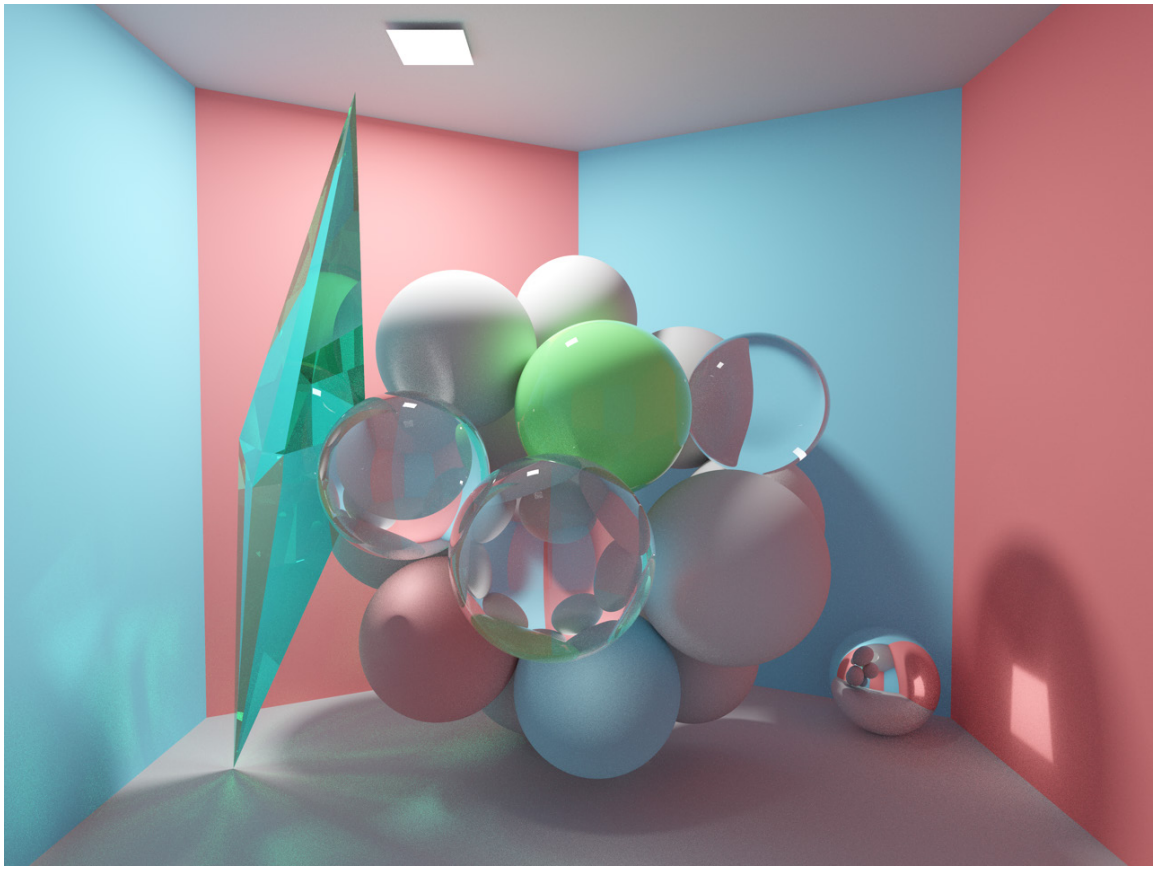

Figure 5 – Hexagon room rendered with path tracing.

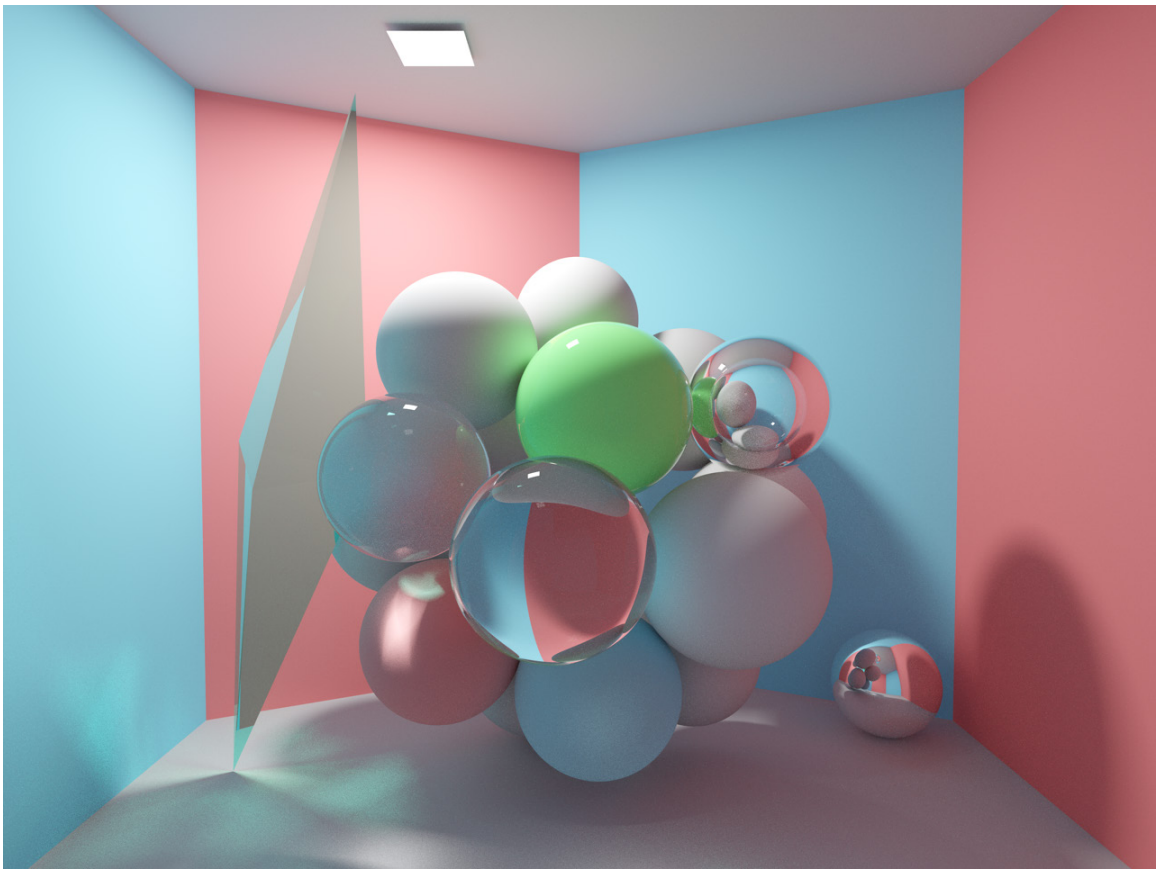

Figure 6 – Hexagon room suspended in flint glass (scene IOR of 1.75) rendered with path tracing.

3.2 Photon Mapped Renders

The image rendered with photon mapping and a scene IOR of 1 can be seen in Figure 6. It was rendered with $2 \cdot 10^6$ photon emissions and a caustic factor of 250, which results in $5 \cdot 10^8$ photon emissions in total. It took 5 minutes and 33 seconds to emit these photons, and Table 1 lists the number of generated stored photons for each photon map.

Table 1 – Number of photons generated for each photon map

Photon map	Photons
Direct	1 839 125
Indirect	2 305 472
Caustic	58 157 882
Shadow	1 355 533

The octrees used 190 max photons per octree leaf, and it took 53 seconds to construct the octrees. The main rendering pass was then rendered in 1280x960 resolution and used 256 samples per pixel (16 SQRTSPP), a global photon radius of 10 centimeters and a caustic photon radius of 5 centimeters. This took 3 hours, 38 minutes and 56 seconds to complete, which in total adds up to 3 hours, 45 minutes and 22 seconds to complete all steps. This is 6.53x less time than the path traced render.

The image looks similar to the path traced render but there are some differences. There are some high intensity dots throughout the scene from the caustic photon map in places where you wouldn't expect to see much caustics. See for example the lower right around the mirror sphere. This is likely caused by "stray" caustic photons with high energy that has taken unlikely paths, which results in there being not enough photons in these places to give an accurate estimate. Another difference can be seen by looking closely at the complex internal and external specular reflections and refractions on the transparent objects in the scene. These have not converged nearly as well as the path traced render, and this is because direct specular effects are evaluated using path tracing in both methods. The number of samples evaluated per second is much lower when using photon mapping, which means that these kinds of effects takes longer to converge with photon mapping. Some faint artefacts from the direct and indirect photon maps can also be seen in the edges of the room, see for example the edges where the walls connect with the ceiling.

Finally, the last major difference is the caustic intensity which is slightly brighter in the photon mapped render compared to the path traced one. I have not been able to figure out what causes this. The Siggraph 2000 course on photon mapping, led by Henrik Wann Jensen, mentions the following:

"Photon tracing works in exactly the same way as ray tracing except for the fact that photons propagate flux whereas rays gather radiance. This is an important distinction since the interaction of a photon with a material can be different than the interaction of a ray. A notable example is refraction where radiance is changed based on the relative index of refraction[Hall88] — this does not happen to photons." – Henrik Wann Jensen, Siggraph 2000 [7]

This could figure into the explanation somehow but implementing this change in the path branching would weight refracted caustic photons even heavier.

The image in Figure 7 shows a version of this render where the photon maps have been visualized directly by evaluating the global contribution from the photon maps at the first diffuse reflection instead of the second. This was rendered using the same settings otherwise. One thing that can be taken from this image is that the flux of the photons in the indirect photon map vary a lot, which produces bad estimates with bright spots in the scene. This can probably be improved by more carefully selecting the Russian roulette absorption factor for the materials so that photons have a similar flux regardless of ray depth and material reflectances.

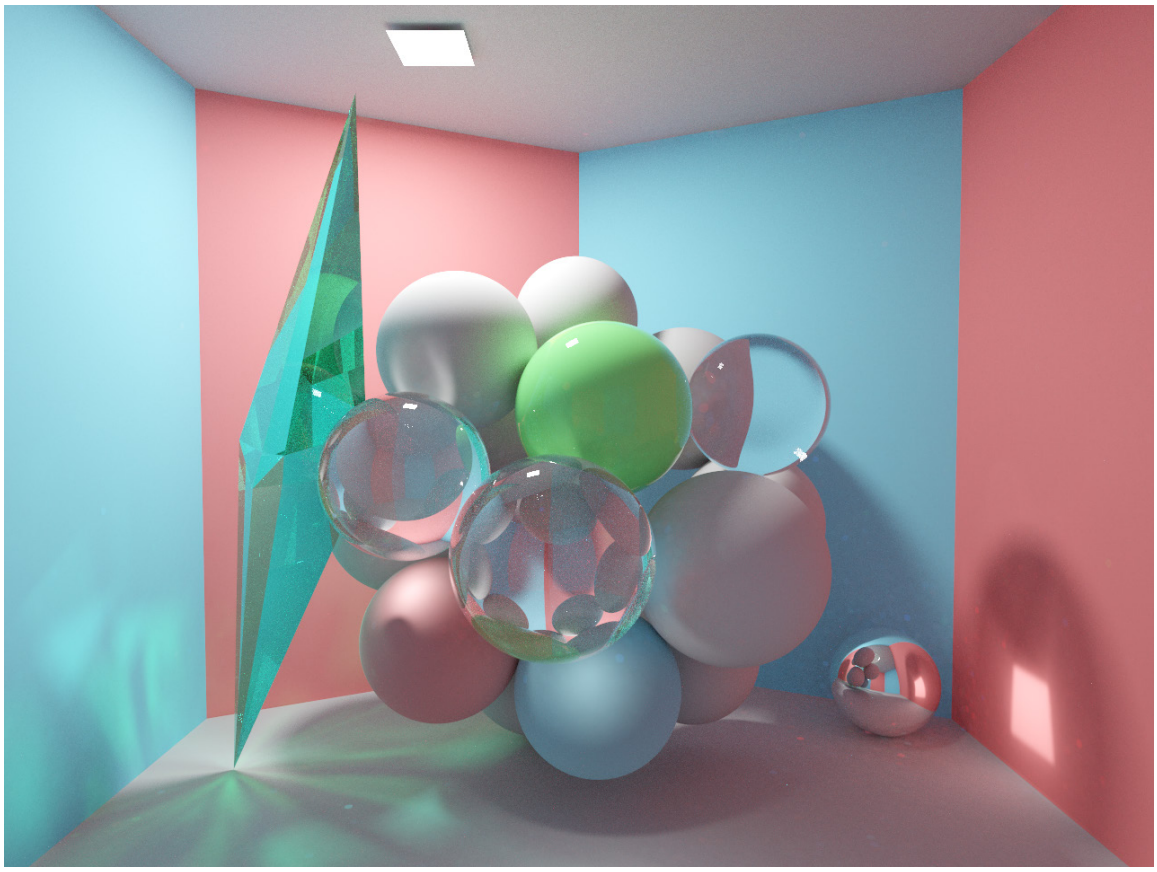

Figure 7 – Hexagon room rendered with photon mapping.

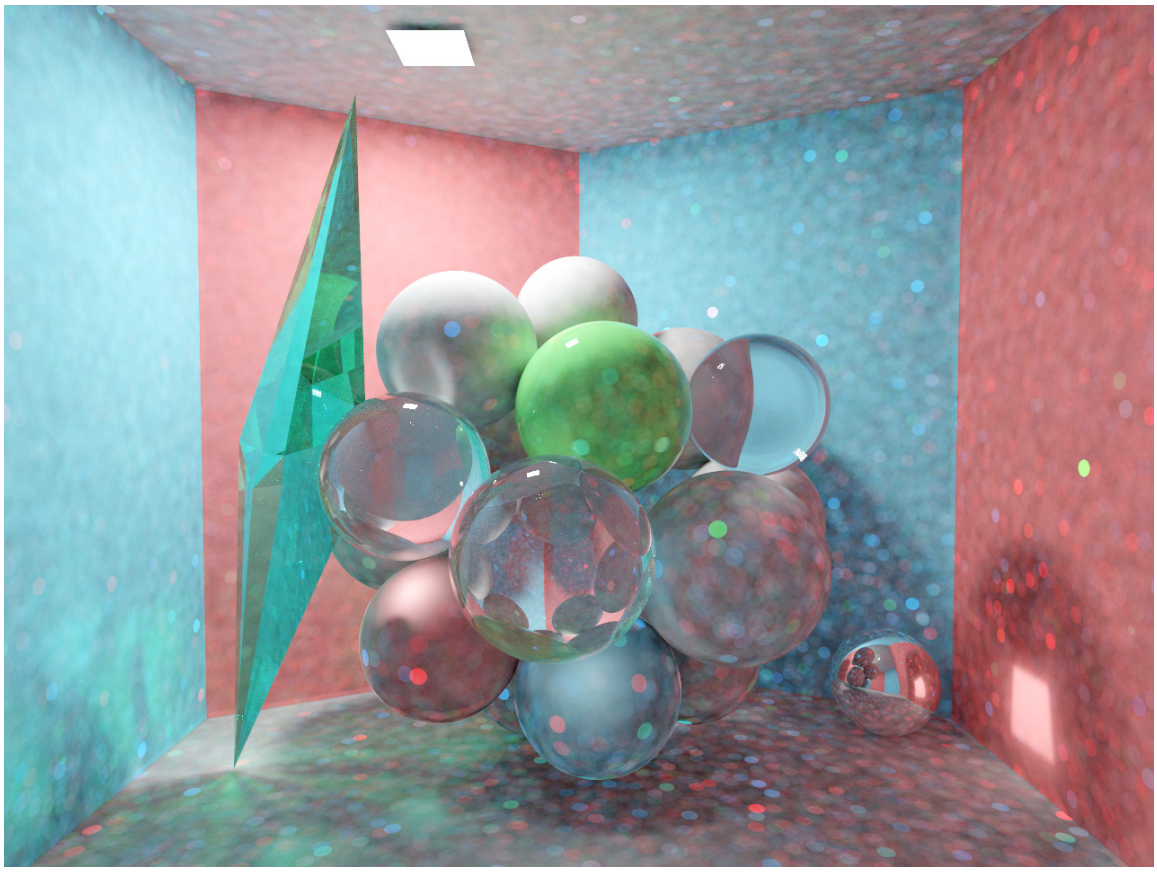

Figure 8 – Hexagon room with photon maps visualized directly from first diffuse reflection

4 REFERENCES

[1] W. J. a. G. H. Matt Pharr, Physically Based Rendering: From Theory to Implementation, 3rd Edition, Morgan Kaufmann Publishers, 2018.

[2] R. Driscoll, “Better Sampling,” 7 January 2009. [Online]. Available: http://www.rorydriscoll.com/2009/01/07/better-sampling/. [Accessed 29 10 2019].

[3] K. Narkowicz, “Automatic Exposure,” 20 May 2016. [Online]. Available: https://knarkowicz.wordpress.com/2016/01/09/automatic-exposure/. [Accessed 29 10 2019].

[4] J. Hable, “GDC Uncharted 2 HDR,” 2010. [Online]. Available: https://www.gdcvault.com/play/1012351/Uncharted-2-HDR. [Accessed 29 10 2019].

[5] J. Hable, “Filmic Tonemapping Operators,” 05 May 2010. [Online]. Available: http://filmicworlds.com/blog/filmic-tonemapping-operators/. [Accessed 29 10 2019].

[6] H. W. Jensen, “Global Illumination using Photon Maps,” 1996.

[7] H. W. J. a. N. J. Christensen, “A Practical Guide to Global Illumination using Photon Maps, Siggraph 2000 Course 8,” 23 July 2000. [Online]. Available: https://graphics.stanford.edu/courses/cs348b-00/course8.pdf. [Accessed 29 10 2019].