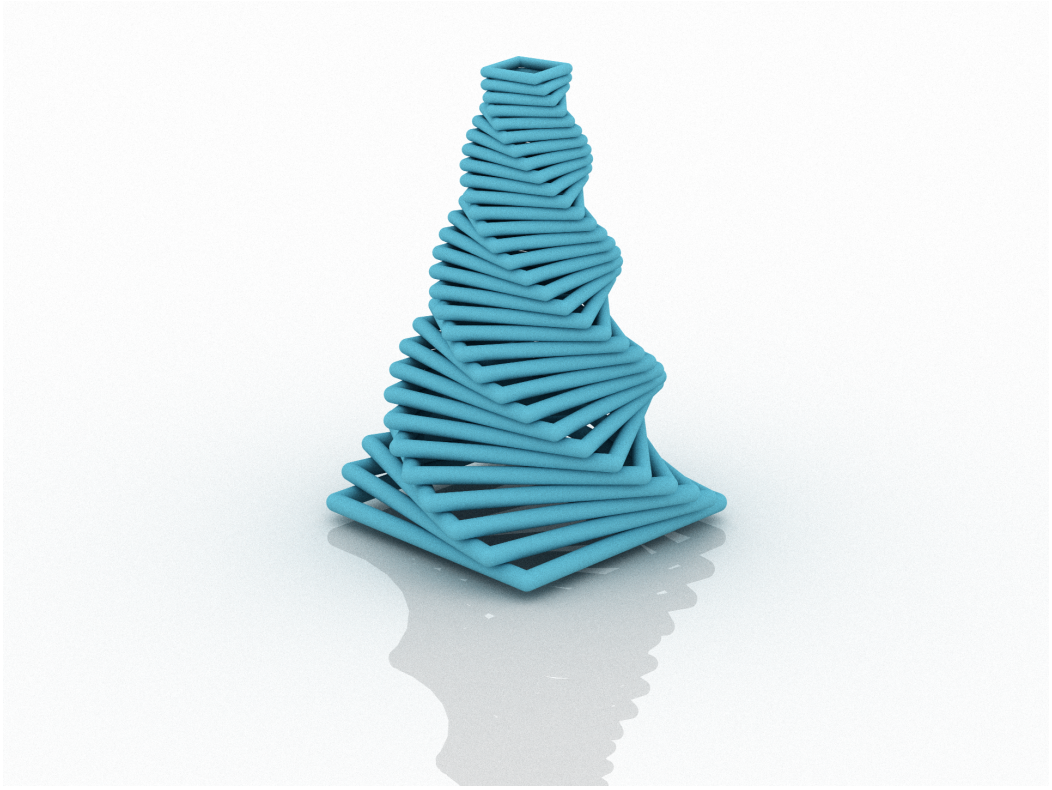
Path Tracer



Informática gráfica

January 2019

721057 Izquierdo Barranco, Sergio
716185 Guerrero Viu, Julia

Contents

Chapter 1

Introduction

This report depicts the development of a Path Tracer. It includes global illumination (both direct and indirect light) and the only parameter of the algorithm is the number of paths per pixel. Camara position and scene can be configured in the source code. To create scenes the basic path tracer includes planes and spheres as geometries, and lambertian, Phong, perfect specular and refractive as types of materials. Light sources can be modelled both with puntual lights or area/volume lights, creating a geometry that emits. This basic version is explained in chapter 2.

Additional extensions have been implemented, which include firstly new geometries: cuboids, cylinders and cones, as well as Constructive Solid Geometries, with which you can create many different forms by applying boolean operations to a combination of an arbitrary number of basic figures. For refraction materials, the Fresnel model has been implemented. Finally, image textures to modelate material appearance are included. Parallelization techniques and an analysis of render time is added as the last extension. All these extensions are explained in chapter 3.

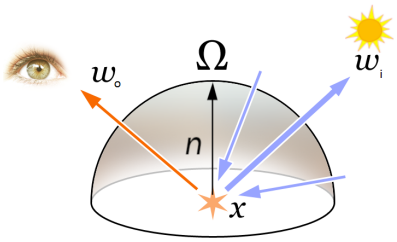
After that, in chapter 4 an explanation of the main challenges faced and how they were solved is included.

Finally, some conclusions of the project and an analysis of workload and distribution of tasks between the members of the group are included as the last two chapters.

Chapter 2

Basic Path Tracer

2.1 Render Equation



$$L_o(x, w_o) = \int_\Omega L_i(x, w_i) f_r(x, w_i, w_o) \mid n \cdot w_i \mid dw_i$$

First of all, the meaning of each term in Render Equation is explained, to be able to discuss how it has been solved by the Path Tracer algorithm.

$L_o(x, w_o)$ is the total radiance that arrives to the observer in direction w_o when it looks at point x. To calculate it, as it can be seen in the equation, it needs to be taken into account the amount of light that x receives, the material of the surface of x and the direction of the light respect to the normal of that surface. As we are considering global illumination, light in x must be integrated in the whole hemisphere defined by the point and the surface, as the light could come from any direction within that hemisphere. Regarding the $f_r(x, w_i, w_o)$ term, it is called the BRDF and it depends on the material of the object. It modules the way light is distributed within the hemisphere after hiting the object in point x. Finally, it is important to consider the cosine of the angle formed by the normal of the surface and w_i, as the more perpendicular to the surface light comes, the more intense it will be.

Once the equation is understood, a distinction between direct (light that comes directly from a light source to an object) and indirect illumination (light that comes from the whole scene) should be done.

Direct illumination is easy to calculate, as it can be obtained by tracing a shadow ray from the point to the light source and see whether it is illuminated or occluded.

Indirect illumination, however, is very complex and time consuming. As it would be infeasible to trace rays in every direction, Monte Carlo is used to approximate the integral. Monte Carlo is

based on the idea of sampling the hemisphere (solution space), calculating light in each sampled direction to sum them all. The most important part of the Monte Carlo method is then how to best sample the hemisphere to make it converge to the solution faster, what is called importance sampling.

In the algorithm developed, the following steps are conducted to calculate the equation:

- When the ray hits a surface, a Roussian Roulette is conducted based on the coefficients of the material. This means, for example, if an object has $Kd = 0.3$ (diffuse component) and $Ksp = 0.5$ (perfect specular component), 30% of the times it will sample next ray based on a Lambertian BRDF and 50% of the times on the perfect specular direction. The rest of the times (20% in this case) would go for absorption, what means the path comes to an end. Actually, this coefficients are RGB colors in three different channels, so we use the average to compute it. Finally, to assure the termination of the path without having to stop after a fixed number of bounds, a probability of 0.1 of being absorbed is always guaranteed.

- Given a component (diffuse/Phong, perfect specular or refraction), importance sampling is conducted to calculate next ray. For delta functions, we just have to calculate deterministically next ray based on the angles of incidence. For both diffuse and Phong materials, we use uniform cosine sampling as discussed in class. Phong specular lobe could be better sampled but has not been implemented.

- Paths always end under three different circumstances: it does not reach anything, the Roussian Roullete chooses absorption or it reaches an emitter material. However, when having puntual lights it is matematically impossible to reach them, as differential points, so the scene would be totally black. To solve it, we apply next event estimation to each of the lights in the scene every time a surface is hit. If the object is not occluded (there is nothing between the surface and the light source) next event estimation is calculated as direct light, multiplying the luminance of the source with the BRDF of the material and the cosine term (Render Equation), and dividing by the squared distance.

- Combining everything already mentioned, the final value of the luminance of the pixel is calculated backwards, calculating luminance in every step by multiplying all indirect light with the BRDF of the material and the cosine term diving by the *pdf* used when sampling, and adding direct next event estimation. All the steps are accumulated to give the final result.

- For antialiasing purposes, rays are traced from the camera to a chosen random point inside each of the pixels.

2.2 Convergence

The default Cornell Box image has been rendered up to an acceptable level of convergence, what we can assume as ground truth, to be able to discuss the overall convergence of the algorithm. Here is the result:

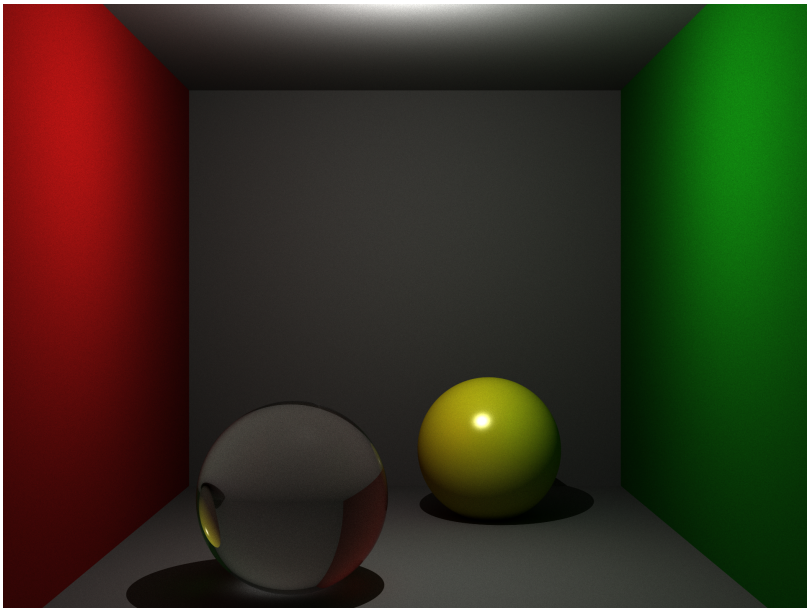


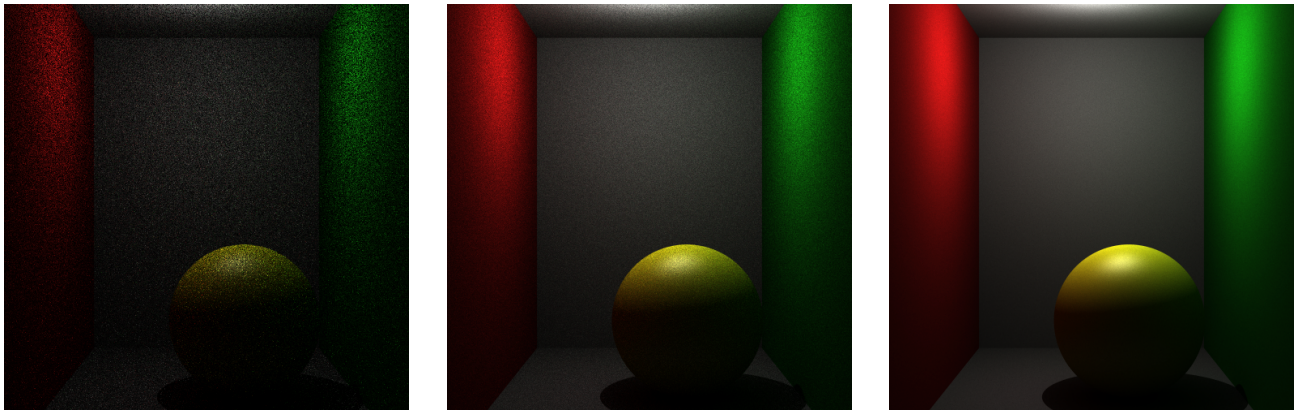
Figure 2.1: Cornell box converged, rendered with 1024 ppp. and high resolution.

This image has high resolution (1920 x 1440) and 1024 paths per pixel. Walls are all diffuse, yellow sphere is Phong with Kd=0.7, Ks=0.1, α=50 and glass sphere is modelled with Fresnel model, Kr=0.9, η=1.51. There is only a point light source at the top-middle of the box. Execution time (applying parallelization techniques that will be analyzed) was 45 minutes.

Regarding convergence, an analysis of the different parameters of the scene has been done below, comparing different renders and discussing them. All of the scenes have lower resolution (500x500) and just one sphere to make it more efficiently computed.

2.2.1 Paths per pixel

With respect to the number of paths per pixel, it can be clearly seen in the renders below how the noise reduces while the number of paths increases.



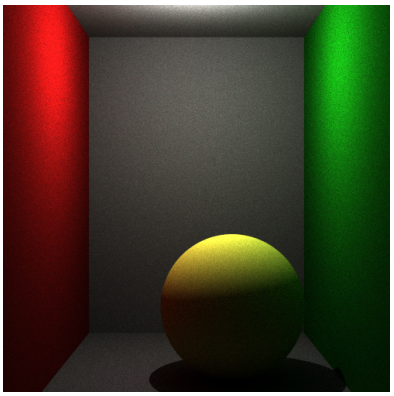
(a) 8 ppp (b) 128 ppp (c) 2048 ppp

Figure 2.2: Cornell box with one puntual light. Walls are all diffuse, sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.

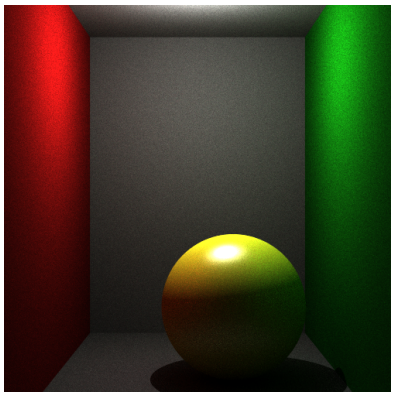
Although the ratio between both 8-128 and 128-2048 paths per pixel is 16, the convergence does not increase linearly. Path tracer converges by a factor of $1/\sqrt{n}$, what means that to make it converge twice better, you need four times more ppp, or in this case, with 16 times more ppp we obtain 4 times more convergence.

2.2.2 Materials

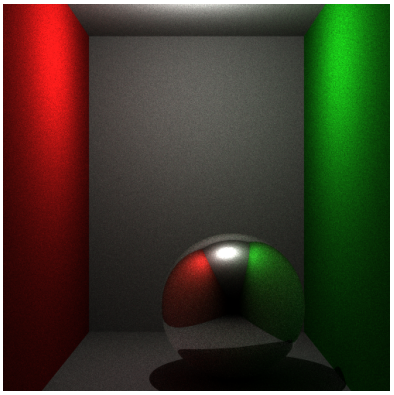
Four scenes of a typical Cornell box with illumination made by a puntual light are presented, with the only difference of the material of the Sphere: totally Lambertian or diffuse, Phong BRDF, perfect specular and perfect refraction. The algorithm developed converges a bit faster with diffuse and Phong materials, although differences are not remarkable. The reason why it converges slower with perfect reflection and refraction is that next event estimation is not calculated when ray hits one of these materials, so it is easier that a ray ends without contributing anything to the final result than on a diffuse surface. To be able to compare images fairly, all materials of the spheres should have the same probability of absorption, what in this case is 10% for all of them. Although differences are slight, it is specially appreciated between the diffuse and the refractive spheres. Reinhard tonemapper with a maximum luminance of 1.5 is used, to avoid an effect of darker scene when there are reflections of the light source.



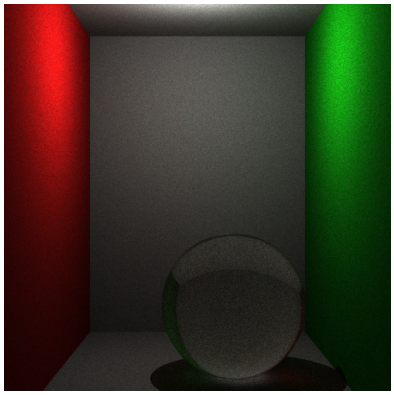
(a) Diffuse sphere, Kd=0.9



(b) Phong material, $Kd = 0.65$, $Ks = 0.25$, α=20



(c) Perfect specular, Ksp=0.9

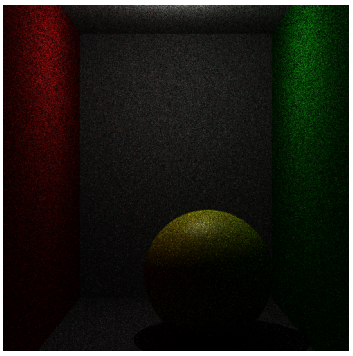


(d) Perfect refraction, Kr=0.9, $\eta = 1.51$

Figure 2.3: Cornell box rendered with 256 ppp. Walls are all diffuse and a puntual light source is illuminating the scene from the middle-top of the box.

2.2.3 Light sources

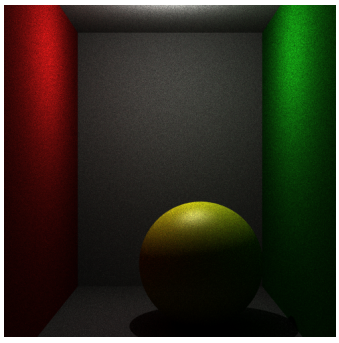
As it can be seen clearly in renders, area light sources make the algorithm converge way slower. This is due to the fact that next event estimation is just computed for puntual lights, so they always contribute to the scene. Area lights, however, must be found when doing path tracing, what causes a lot of paths to end without providing any light. Regarding execution time, due to the same reason already explained, scenes with area light have to do less calculations in each step so given the same paths per pixel, area lights are around 40% faster than puntual lights. For the renders below, one puntual light source at the top-middle of the box and two different area lights (the whole upper wall and an emitting texture that simulates a square light source) have been used. There is also a clear difference between both area lights regarding convergence, as it is way easier that a ray intersects with the whole infinite plane of the upper wall, than with a small square on the ceiling.



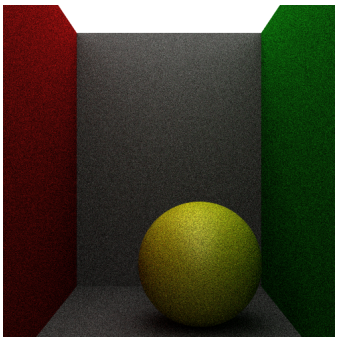
(a) Puntual light

(b) Infinite plane area light

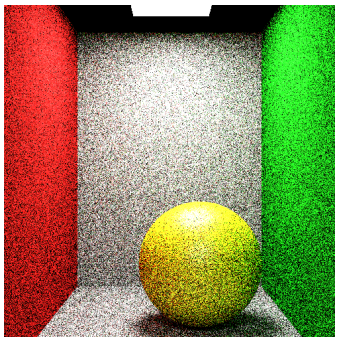
Figure 2.4: Cornell box rendered with 8 ppp. Walls are all diffuse, sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.



(a) Puntual light

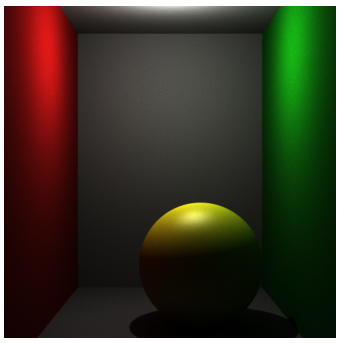


(b) Infinite plane area light

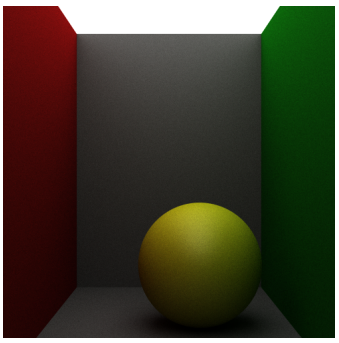


(c) Squared area light

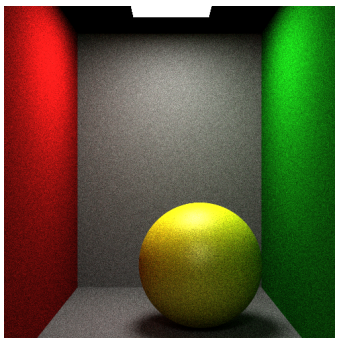
Figure 2.5: Cornell box rendered with 128 ppp. Walls are all diffuse, sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.



(a) Puntual light



(b) Infinite plane area light



(c) Squared area light

Figure 2.6: Cornell box rendered with 2048 ppp. Walls are all diffuse, sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.

2.3 Global Illumination

In this section, global illumination effects (hard and soft shadows, color bleeding and caustics) are discussed and illustrated with renders.

2.3.1 Shadows

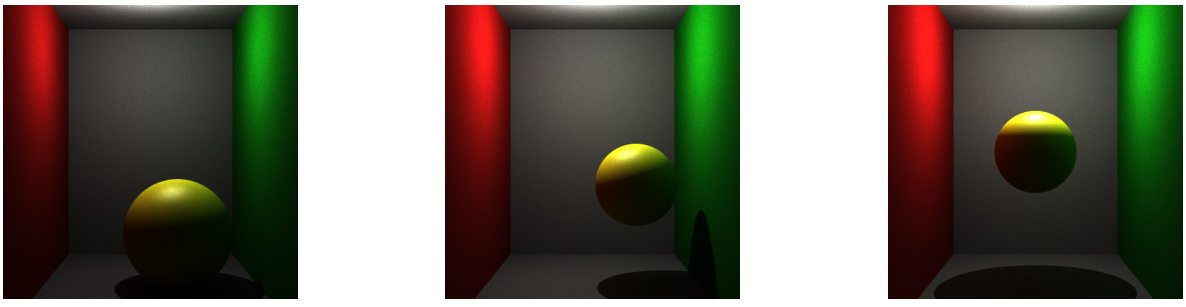
Shadows are easily obtained with path tracing, as they depend basically on the occlusion of the object by other geometries in its path to light. We could distinguised between hard and soft shadows. First of all, harder shadows are obtained with puntual light sources, as next even estimation is computed, because in that case whenever the object is directly illuminated, light contributes a lot to it and on the other hand, when the object is occluded, direct light does not contribute at all. Softer shadows are obtained with area light sources, not only because of the lack of next event estimation but also because of the geometry of the light source itself, as it is harder to be totally occluded having an infinite plane emitting source. The main difference between these two shadows is that the one from puntual light is sharper and the one from area light is more gradually blurred.

(a) Puntual light, harder shadow

(b) Infinite plane area light, softer shadow

Figure 2.7: Cornell box rendered with 512 ppp. Walls are all diffuse, sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.

Secondly, harder shadows are obtained on a wall for example, if the object is very close to it, while a softwer one forms if it is further. This is due to the contribution of global illumination, as, although it is occluded with direct light, if there is more space in between, more indirect light will come to illuminate the wall.



(a) Ball on the floor, harder shadow

(b) Harder shadow on the wall, and little softer on the floor

(c) Ball in the middle of the box, softer shadow

Figure 2.8: Cornell box rendered with 512 ppp. Sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$.

Finally, if there are more than one light, shadow effects become harder to analyze. For example, in this scene, there are two different point light sources, at the top corners. As it can be seen, whenever there is a shadow from both lights it seems harder, and when it is just from one of them but is directly illuminated by the other, the shadow becomes softer. Scenes with many light sources are interesting and nice to see but is more difficult to discuss and know exactly the contribution of each of them.

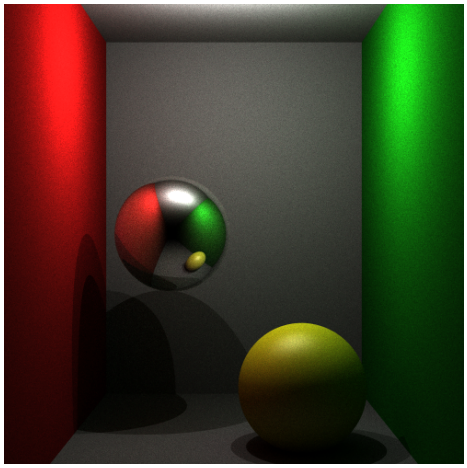
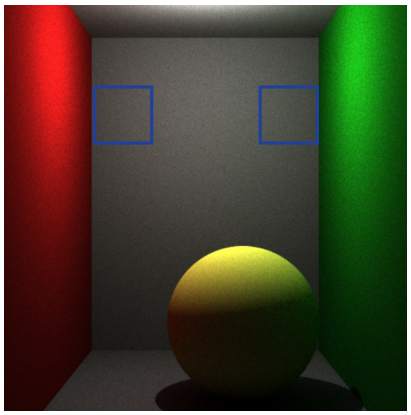


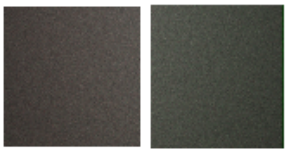
Figure 2.9: Cornell box rendered with 512 ppp. Walls are all diffuse, yellow sphere is Phong with $Kd = 0.7, Ks = 0.1, \alpha = 10$ and mirror sphere is perfect specular with $Ksp = 0.9$. Two puntual lights on the top corners of the box.

2.3.2 Color bleeding

Color bleeding can also be obtained with the algorithm, as we can see in the image. It is the phenomenon in which objects or surfaces are colored by a colored light from nearby surface due to global illumination effects. The scene used is all diffuse to avoid confusing between color bleeding effect and specular reflection or transmissive properties. It can be specially appreciated on the shadows, as on those surfaces direct illumination does not contribute, so color from indirect illumination is easily seen. Red and green colors from the walls can be appreciated on the sphere and on the shadow from the lower wall. However, although slightly, color bleeding is also present for example, on the back wall, as it is highlighted on the Figure 2.10b.



(a) Color bleeding on the Sphere



(b) Zoom-in from blue highlighted areas on the back wall to observe color bleeding

Figure 2.10: Cornell box rendered with 512 ppp. Walls and Sphere are all diffuse and there is one puntual light on the top.

2.3.3 Caustics

Caustics are a very difficult (almost impossible regarding convergence time) phenomena to simulate with path tracing. With puntual lights, caustics are impossible to be captured as, to do so, a ray should hit the light after being transmitted through a delta material, and as puntual lights are differential points in the space, this will never happen. In some of the renders included before, such as Figure 2.3d, some soft "caustics" appeared. It does not go against the idea just discussed, as that effect is seen because the top wall acts as a kind of emitting material, because of the close distance to the puntual light. Using area lights, caustics can be captured but with a very very slow convergence. The scene showed bellow has a cuboid area light on top and was rendered with 4096 paths per pixel. Even with that high number of ppp, it has not converged at all and still has a lot of noise, but a caustic can clearly be observed on the floor.

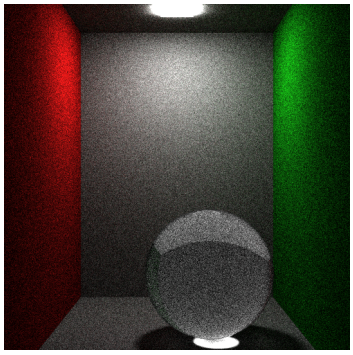


Figure 2.11: Cornell box rendered with 4096 ppp. Walls are all diffuse, sphere is a Fresnel material with Kr=0.9 and η=1.51. One small emitting cuboid is used as light source.

Global illumination render

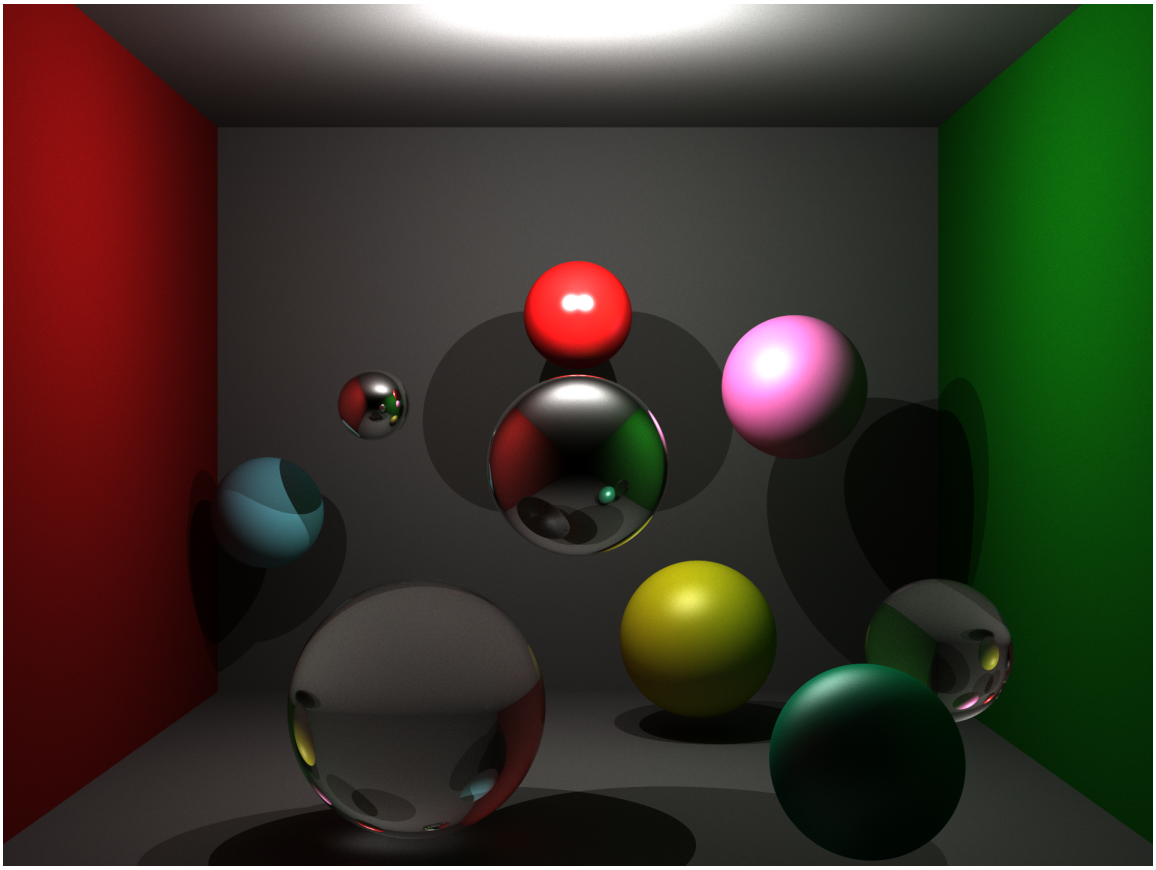


Figure 2.12: Converged Cornell Box, rendered with 2048 ppp. and high resolution. There are two puntual light sources on the top and the following materials: 2 mirror spheres with $Ksp = 0.9$, 2 glass spheres with $Kr = 0.9$ and $\eta = 1.51$, blue small sphere is diffuse with $Kd = \{0.4588, 0.85, 0.95\}$, yellow sphere is Phong with $Kd = \{0.8, 0.8, 0.085\}$, $Ks = 0.1$, and $\alpha = 10$, red sphere is more glossy, Phong with $Kd = \{0.9, 0.1, 0.1\}$, $Ks = 0.2$, and $\alpha = 30$, pink sphere is Phong with $Kd = \{0.9, 0.4117, 0.7059\}$, $Ks = 0.1$, and $\alpha = 5$, and emerald sphere is Phong with $Kd = \{0.05, 0.6588, 0.4196\}$, $Ks = 0.3$, and $\alpha = 10$. Notice the fancy shadows on the small blue ball, or that the whole scene can be seen reflected on the mirror spheres. Glass spheres are also very curious and Fresnel effect can be appreciated especially in the biggest one.

2.4 Tonemapper

Rendered images have a high dynamic range, and therefore, it is not possible to export them as 0-255 PPM directly. Instead, a tone mapper operator must be applied. Our render usually is used with the following methodology: first, several images are rendered and stored in HDR format; then, a small script accumulates the values of each image; and finally, the resulting image is tone mapped to LDR with one of our available operators and exported in LDR, ready to be visualized. The operators usually work with luminance values, scaling, applying a function or clamping it. Rendered images are represented in RGB tuples, where the luminance is not separated from the colour, and for this reason, each value is converted to xyY space (where Y denotes the luminance) before applying the operator, multiplying the RGB tuples by a constant matrix.

The following operators have been implemented:

Clamping: Each luminance is at maximum a value v.

$$L'_i = \min(L_i, v)$$

Equalization: Each luminance is scaled with reference to the maximum luminance at the image.

$$L'_i = \frac{L_i}{\max L_j - \min L_j}$$

Equalize and Clamping: Mix the two previous approaches.

Gamma curve: Each luminance is transformed by a *gamma* function with parameter γ after being equalized (L_eq).

$$L'_i = {L_{eq}}^{\frac{1}{\gamma}}$$

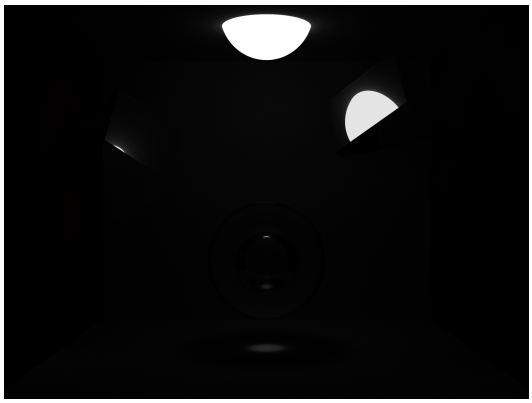
Clamp and gamma curve: The same gamma curve with a previous equalization and clamping (instead of just a previous equalization).

Reinhard 02: A tone mapper operator based on [3]. First, it computes the average luminance, $\bar{L}_w$, divides each luminance by this average and then applies a sigmoid-like function to correct the maximum luminance (L_{max}).

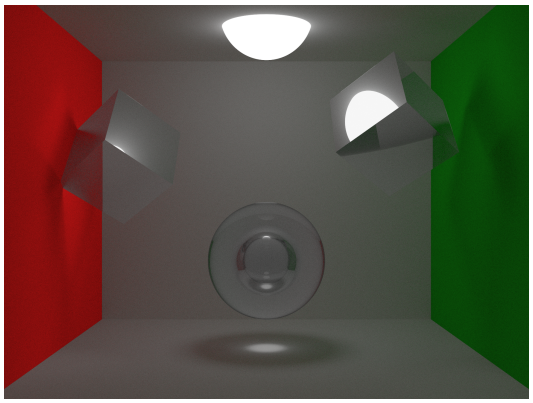
$$\bar{L}_w = \frac{1}{N} exp\left(\sum_i log(\delta + L_i)\right)$$
$$L'_i = \frac{a}{\bar{L}_w} L_i$$
$$L''_i = \frac{L'_i\left(1 + \frac{L'_i}{L^2_{max}}\right)}{1 + L'_i}$$

The value of L_{max} in the last correction can be set manually, or set to the maximum scene luminance. This operator fit very well in the vast majority of the scenes, and for this reason, it is used by default in our render. More complicated scenes could require to tweak the L_{max} value, but in most of the cases, the highest luminance works like a charm.

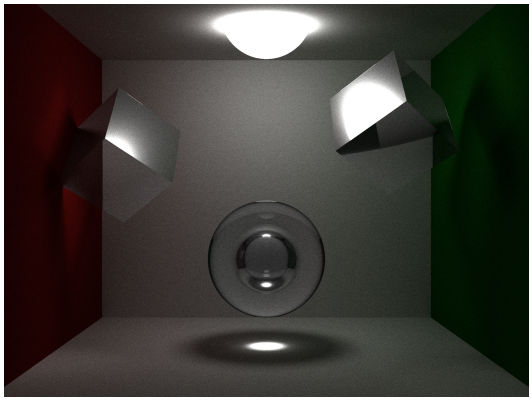
In Figure 2.13 different images of the same rendered scene but with different tone mapping operators are shown. It is a difficult scene as the light is directly visible, there are caustics, and light reflections on the glasses.



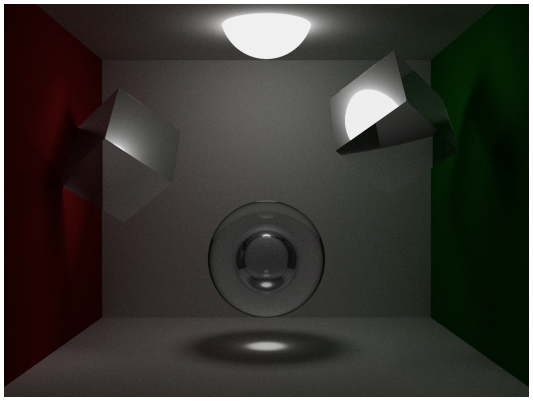
(a) Equalization



(b) Gamma, $\gamma = 3$



(c) Reinhard, $L_{max} = 1.5$



(d) Reinhard, L_{max} the highest luminance

Figure 2.13: Comparison of tone mapper operators

Chapter 3

Extensions

3.1 Geometries

The basic geometries of sphere and plane confine the creativity. In order to create scenes pleasant to the eyes, new parametric geometries have been added as well as the possibility of performing boolean operators on them, enabling the chance of creating arbitrary complex figures.

3.1.1 Cuboid

It is difficult to find an appropriate task for an infinite plane rather than being the floor, in contrast, cuboids allow a variety of shapes, fitting very well in various scenes.

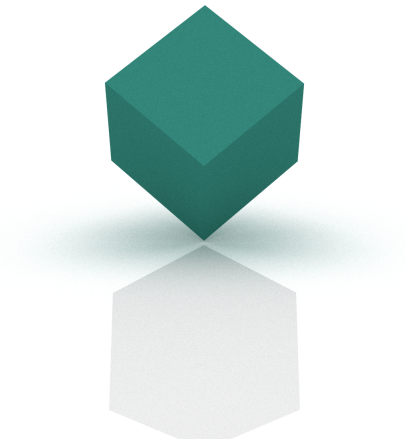


Figure 3.1: Simple cuboid

Cuboids are not axis-aligned and may have different sizes in each of its dimensions. To represent them, three directions (orthonormal basis representing the length and direction of the sides) and a point must be provided. Internally cuboids are composed of 6 planes (constructed from the basis and the point), and to check whether a ray intersects with a cuboid or not it is necessary to calculate the intersection with all of them, saving the nearest point that is inside the cuboid. To test if a point is inside the cuboid a change of local base is calculated taking as reference the directions that represent the cuboid, then the test is simply to check the distance with the origin in each of the axes.

3.1.2 Cylinder and Cone

Cylinders and cones have parametric equations and therefore its intersection is simple to calculate. Only should be taken into account that parametric equations represent infinite geometries and that extra calculations should be made in order to enclose them.

Equation for a cylinder oriented along line $p + t * v$ with point q is:

$$(q - p - (v \cdot (q - p))v)^2 - r^2 = 0$$

To calculate the intersection with a ray substitute q with the ray equation. Once calculated, the ray will be on the finite cylinder depending on the distance with both extremes. There will be an intersection with the capes if the distance from the intersected point at the plane to one of the extremes is less than the cylinder's radius.

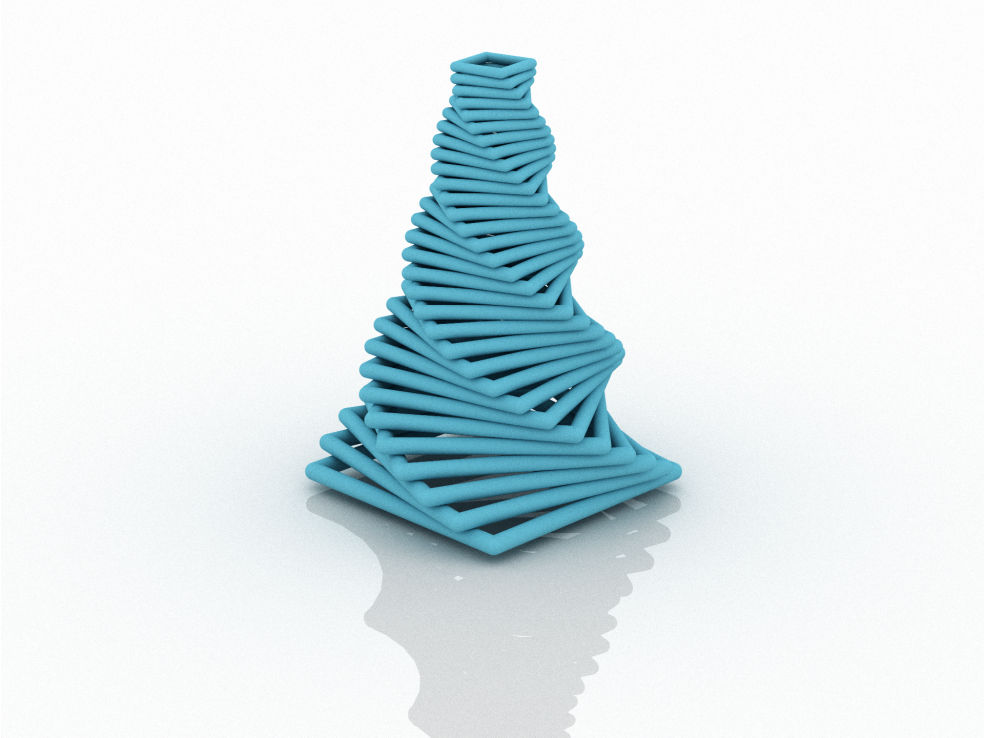


Figure 3.2: Stack of cylinders. Spheres have been added on the extremes of the cylindres for artistics purposes.

Cones follow a similar approach to cylinders as actually, the seconds are just a special case of the cones when the appex happens at the infinite. Cones follow a similar approach to cylinders as actually, the seconds are just a special case of the cones when the apex happens at the infinite. The following equation represents a cone oriented among the line $p + vt$ with apex at p and apex angle α:

$$cos^2\alpha(q - p - (v \cdot (q - p))v)^2 - sin^2\alpha(v \cdot (q - p))^2 \tag{3.1}$$

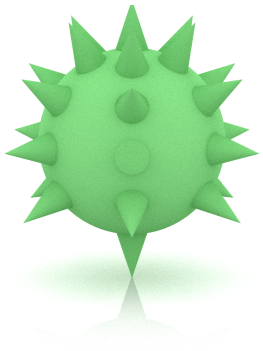
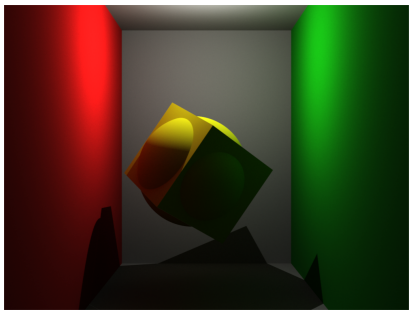


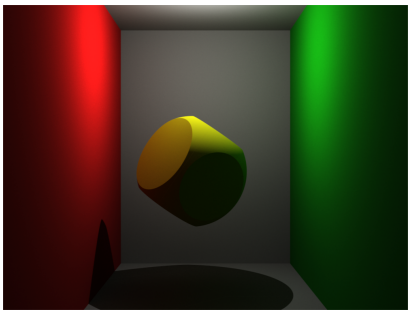
Figure 3.3: Sphere covered with cones.

3.1.3 Constructive Solid Geometries

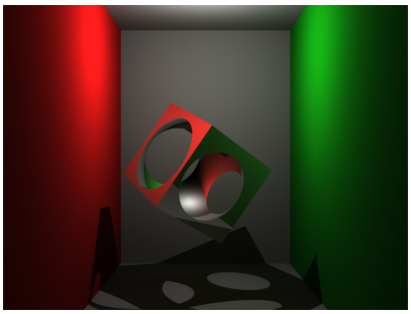
Unlike triangle meshes, that represent world figures by reducing them to a simplification made up of triangles, constructive solid geometries (CSG) enable a perfect definition of the figures by specifying boolean operations between the shapes [1]. A CSG is defined by a tree where the leaves represent figures, and inner nodes the operations between them. Operations are union, intersection and difference.



(a) Union between sphere and cuboid



(b) Intersection between sphere and cuboid



(c) Difference between sphere and cuboid

Figure 3.4: Different CSG boolean operators applied to a sphere and a cuboid.

To calculate the intersection between a ray and a CSG, first of all, all the intersection between the ray and the CSG's figures must be calculated and stored as intervals. An interval defines when did the ray entered a volume, and when it scaped (note that also must be stored information about the intersection (normal, point...). Once all the intervals are calculated, the tree is traversed recursively.

If the node is a leaf, its interval with the ray is returned (if exists). If it is an inner node, first, its two child results are calculated, and then merged depending on the node operation as shown in Figures 3.5, 3.6 and 3.7. Difference of intervals can result in more intervals that at the beginning, a situation that must be controled.

Finally, the intersection at the beginning of the first interval returned by the root of the tree is the intersection with the CSG.

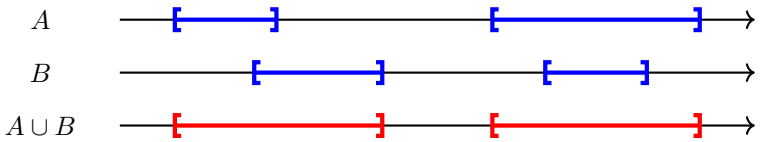


Figure 3.5: Union of two sets of intervals

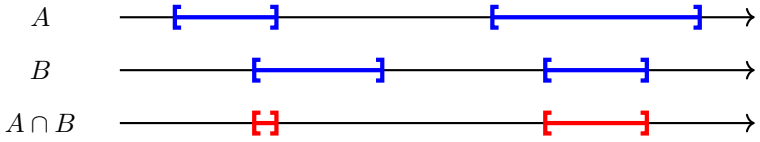


Figure 3.6: Intersection of two sets of intervals

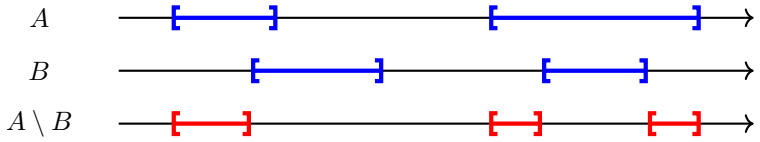


Figure 3.7: Difference of two sets of intervals

Not only these structures are useful to construct complex figures but also they act as an optimization structure. For instance, to create a Menger Sponge Fractal of 4 iterations we could require 160000 cubes. We could improve this by creating just 1 cube and defining the 58947 cubes to subtract, obtaining a significant speed-up. The first 4 iterations of this fractal are shown in Figure 3.8.

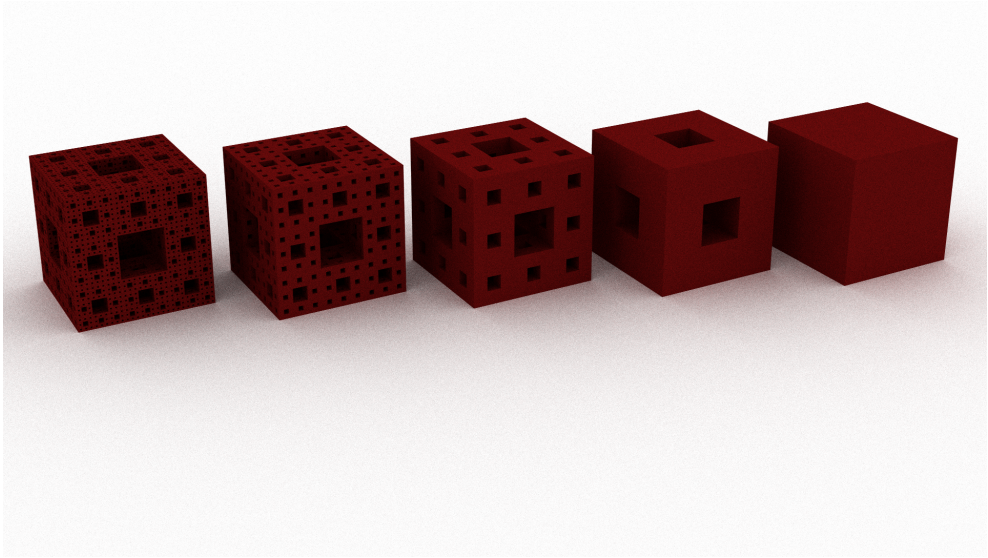


Figure 3.8: First 4 iterations of the Menger Sponge Fractal

3.2 Fresnel

Fresnel equations model dielectric interactions. Materials are not simply transmissive in a way that each ray is always refracted, instead, the amount of energy reflected and refracted depends on the incoming direction and the index of the material. These equations, for instance, model the effect that happens on lakes, when far points reflect the mountains and close points act as a glass.

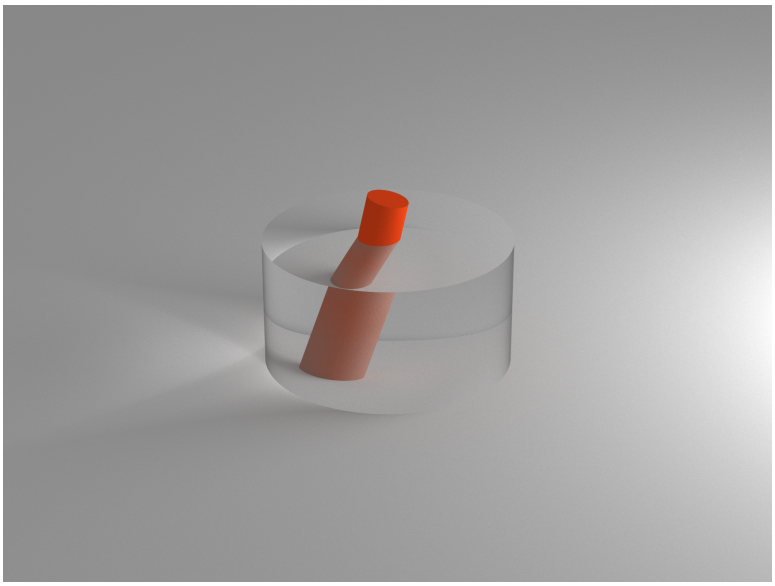


Figure 3.9: Refraction distorts the stick when it enters the glass.

According to Fresnel, the percentage of refracted light on a point is:

$$F_T = \frac{1}{2}\left(\left(\frac{\eta_2 cos\theta_1 - \eta_1 cos\theta_2}{\eta_2 cos\theta_1 + \eta_1 cos\theta_2}\right)^2 + \left(\frac{\eta_1 cos\theta_2 - \eta_2 cos\theta_1}{\eta_1 cos\theta_2 + \eta_2 cos\theta_1}\right)^2\right)$$

A ray cannot be divided in two to follow both options. To take the decission of wich path to follow a random number is generated and depending on its value compared to F_T the ray will refract or reflect. Anyway, the energy obtained by the ray will be multiplied by F_T or $1 - F_T$.

It should be taken into account that there is a special case, when the ray have a small angle (critical angle) it will always refract, what is called total reflection.

In Figure 3.10 it is easily noticeable how the cuboids perform refraction or reflection depending on the incoming rays. The top-right cuboid has one side reflecting the light, other side reflecting the green wall, and the front side is doing both, divided by its half. Also is interesting to notice how the two spheres show the wall on the opposite sides due to the effect of refraction.

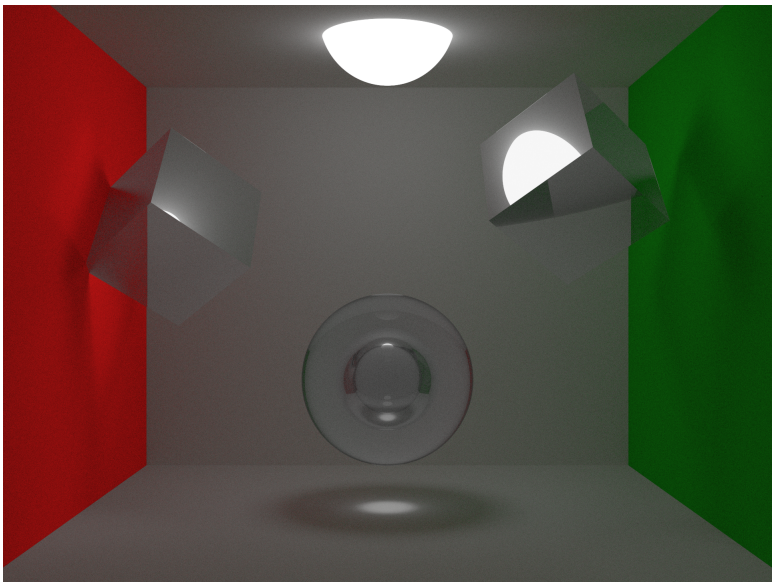


Figure 3.10: Three figures with fresnel materials. Three figures with fresnel materials. The outer sphere is glass while the inner is just air.

3.3 Textures

When a texture is added to an object, it will define the value of its K's on each point. For instance, for a diffuse material, the texture could define that in a certain point $Kd = \{0.8, 0.8, 0.25\}$ whereas in another part of the object it could be $\{0.05, 0.05, 0.85\}$. It also works, in the same way, to define where an object emits or where it is Lambertian.

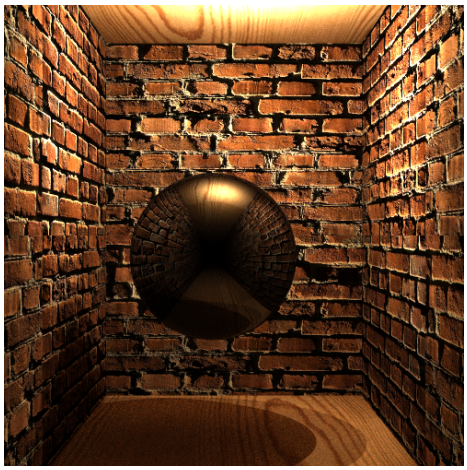


Figure 3.11: Textures on planes

For spheres, planes, cuboids, cylinders and cones its own parametric equations define how to apply an S-mapping on them. In contrast, CSGs are complex structures, applying S-mapping to each of its objects independently does not fit very well, and results ugly when seeing the figures as a whole. For this reason, an O-mapping with cylinders has been used.

Figure 3.12: Textures on sphere, cylinder and cone

Planes mapping are the simplest. A reference point in the plane is defined, and the intersection will be mapped on the image according to its distance with the reference. Images are tiled, so the distance is moduled. Cuboids just use the mapping on its planes.

When mapping a sphere, the intersection point coordinates are transformed into the sphere polar system. Then the inclination indicates the height at the image whereas the azimuth the width. Let $p = \{x, y, z\}$ the intersection at the sphere in local coordiantes, θ and ϕ can be calculated as follows:

$$\theta = atan\left(\frac{\sqrt{x^2 + y^2}}{z}\right)$$
$$\phi = atan\left(\frac{y}{x}\right)$$

If a cylinder is unrolled it will be a rectangle. Its mapping consist of calculating the intersection point at the cylinder in its rectangle. To achieve it, the distance from the point to the bottom cape indicates the height. To get the width polar coordinates must be used. The capes of the cylinders are mapped separately with the mapping of its planes. Cones mapping follow the same approach that cylinders. Let p be the intersection point, c the closest point at the cylinder's axis and v a reference vector at the normal to the cylinder's surface, θ, the width, can be calculated as follows:

$$cos\left(\theta\right) = \frac{p - c}{|p - c|} \cdot r$$
$$sin\left(\theta\right) = \left|\frac{p - c}{|p - c|} \times r\right|$$

A wrapping cylinder must be provided to correctly texture a CSG. Once the cylinder is declared the mapping consists on calculating the closest point from the intersection to the wrapping cylinder. When calculated, the image coordinates at the intersection are the cylinder's coordinates of the new point.

3.4 Parallelization

The main algorithm performed in the path tracer casts rays from the camera to the film and calculates the path, storing the resulting colour on the corresponding pixel. Each casted ray is independent of the others, and therefore, there is no need of executing the loop sequentially. Instead, execution should be concurrent, significantly reducing the rendering time.

In order to parallelise without suffering a headache, execution policies have been used. They were included in *C++17* allowing to define the preferred mode of execution in some of the algorithms provided by the standard library [2]. The main loop of the path tracer was replaced for a transform function over an array with the coordinates of the pixels. In each of the pixels evaluated by the transform several rays are cast, its results added and again stored in the array in such a way that the grid of pixels coordinates finally becomes in the resulting image.

The main problem faced is that the principal *C++* open source compiler, *g++*, has not yet implemented the execution policies. There are alternative implementations on *GitHub*, but they were discarded as they are based on external libraries. It has been taken advantage that *Intel* offers free licenses of its *c++* compiler, *ICPC*, for students, and that they have already implemented the execution policies on the compiler.

By making use of *ICPC* the execution policies work like a charm, improving the rendering time by a factor of 2.5 as shown in Figure 6.1. This figure shows rendering times with and without parallelisation, but apart from this, others optimization strategies have been implemented. For instance, each figure stores its local matrix or part of it (as far as it is possible), dynamic memory has been avoided, simple ad-hoc bounding box for complex figures have been added on the scenes and some part of the code has been refactored with the tradeoff of being difficulty readable.

Paths	Time Parallel	Time Sequential
1	0.147	0.353
2	0.270	0.681
4	0.525	1.336
8	1.047	2.631
16	2.061	5.226
32	4.122	10.793
64	8.224	22.111
128	16.430	44.448
256	37.5638	89.029
512	70.9287	175.788
1024	143.868	356.756
2048	289.921	697.720

Figure 3.13: Comparison of rendering time depending on the execution policy

Even with parallelisation, convergence in complex images can last for long periods of time, what makes risky casting a big amount of rays per pixels as final results may not be the expected. A bigger problem is that many times we want to improve an image once we have rendered it, with more rays, but we have to repeat the process from the beginning. To overcome the problem, a small *Haskell* executable that takes a bunch of images as arguments, add them and export the resulting image was created. Now, we can render an image, and after seeing the results, we can repeat the process obtaining an image with the double of rays per pixel.

Chapter 4

Main challenges

During the development of the algorithm, many challenges were faced to debug the code and deeply understand the algorithm and its problems. Some of the main or more curious ones and how they were solved are explained here, illustrated with renders.

Geometries and intersection calculations were probably the biggest source of errors during the whole development. Any time hitting a surface and shooting a new ray from there, one should be aware of not intersecting with the same surface again due to numerical errors. This was solved by adding a constant, epsilon, which controls minimum distances between points. Related to that, when calculating intersections with cuboids, errors were found that made them look like transparent. As the cuboid consists on six planes with their bounds, firstly those bounds were being calculating without taking into account numerical errors, so it was only considered as an intersection if it was hitting the cube exactly on the limit but not around it. The result was a strange appearance of transparency that can be seen in Figure 4.1. The first image is older, when there were also numerical mistakes with all intersections. Second image is still incorrect but just because of the problem of boundaries of planes.

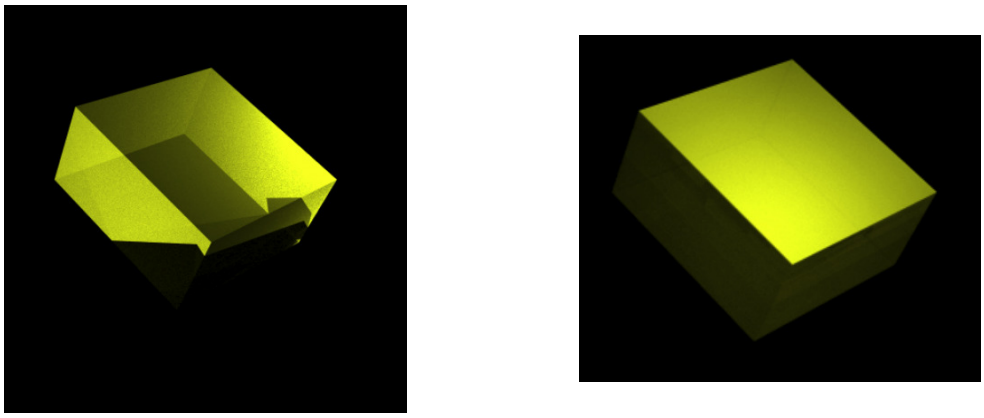


Figure 4.1: Wrong images of a cuboid shaded with Phong and illuminated with puntual light. Notice the appearance of transparency on it.

Another important issue related to geometry was how to calculate the normal of a surface. Although all figures have their own attribute storing the normal, the main difficulty was that on planes, depending on where was the light coming from, normals should be flipped. After having notice and fixing that problem, however, curious cases continue happening due to bugs on the code where normals were forgotten to be flipped. One of the most remarkable ones was when in a Cornell box with area light, the exact parallel plane to the emitting wall was totally black, as it can be seen in Figure 4.2. When rotating that plane a little bit, it became partially illuminated, although it was still a bit dark. After some time debugging, the error was found: although the normal was flipped to do the calculations, it was not when changing to local base of the plane, so when calculating next ray all of them were going in the opposite direction.

(a) Parallel plane, lower wall is black

(b) Parallel plane, right wall is black

Figure 4.2: Wrong images of a cornell box with one area light and 256ppp.

Another source of errors were the formulas of perfect specular reflection and Snell's law. When calculating perfect specular, if not assuring the w_i and w_o rays were correctly oriented, results were quite fancy such as the strange mirror on Figure 4.3.

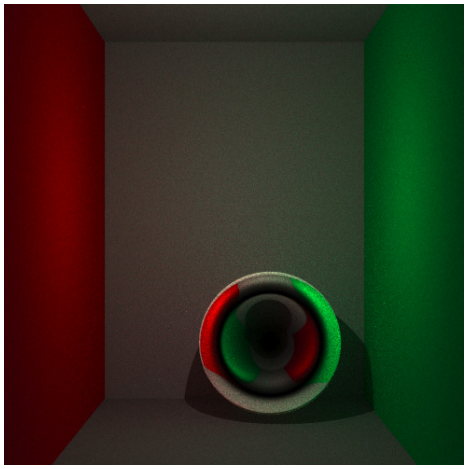
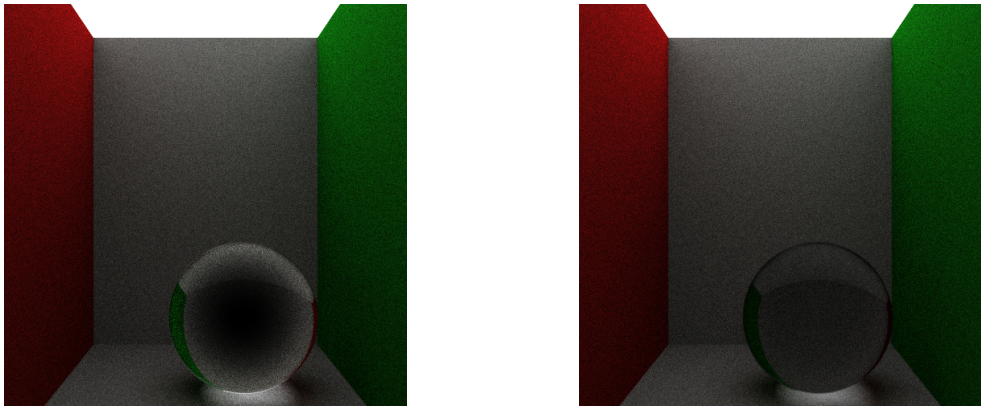


Figure 4.3: Wrong image of mirror Sphere

Finally, some doubts with Render equation Monte Carlo form when calculating delta materials, specially glasses, caused incorrect shading on those figures. For example, when adding the $sin(\theta_i)$ to the equation, as it was wrongly thought to be correct, spheres became black towards the center, where rays from camera were exactly perpendicular to it (parallel to the normal) and though luminance was been multiplied by zero (Figure 4.4).



(a) Wrong calculation of Render equation on glass ($sin(\theta_i)$ included)

(b) Correct calculation of Render equation on glass materials

Figure 4.4: Cornell box with Fresnel Sphere. Notice the difference between including or not the *sin* in the equation.

As a final curiosity, when applying different tonemappers to specific images, depending on that image results can be totally different and seem to be incorrect. This aspect sometimes led us to think that there was an error in the algorithm and in the end it was finally just a problem of dynamic range, which was not tonemapped correctly for the characteristics of the scene.

Chapter 5

Conclusions

In this assignment, we have implemented a path tracing algorithm capable of rendering scenes with a variety of materials and shapes. Starting with a simple ray tracer, upgrades were added incrementally to achieve a basic path tracer. After it, several extensions were considered, and again, incrementally added, constructing the final render engine.

The developed upgrades were chosen because they permit the design of more complex, artistic and beautiful scenes. With Fresnel, the glass looks much more real than before, as the human eye is adapted to its effects. New geometries and textures enable the creativity, as it has been shown in the examples along the report. Finally, parallelization contributed to making all the images possible, by reducing the rendering time.

We are proud of the final results of the path tracer, as we had dig deeply enough to have a very well understanding on the topic while enjoying with the development, by seeing our outcomings directly on the screen in the shape of awful and bizarre anomalies or pleasant and satisfying creations.

Chapter 6

Workload

As this was one of the most engaging courses of the semester, both members of the team were motivated about doing the assignments. For this reason, the majority of the work was made together, while some small differenciable parts were split. Everything concerning a well understanding of the topics was considered a major issue, and it was discussed together before any piece of code was typed.

More specifically:

- The basic ray tracing engine, as well as the pathing basis, were developed together
- Julia (mainly) did textures
- Sergio did cones and cylinders
- Sergio (mainly) did CSG
- Julia did Fresnel
- Both members rendered and designed the different scenes
- Both members wrote this report

The hours spent in each part of the assignment are the following:

Task	Time lasted
Geometry	10 h
Tone mapper	15 h
Ray tracer	10 h
Basic Path Tracer	40 h
Parallelization	8 h
Delta materials (Fresnel)	20 h
Cuboid, Cylinder and Cone	25 h
Constructive Solid Geometries	15 h
Textures	15 h
Report and renders	60 h
Total time:	218 h

Figure 6.1: Time spent on each part of the project

Bibliography

[1] Department of Computing - Imperial College. *Ray Tracing and Constructive Solid Geometry*. https://www.doc.ic.ac.uk/~dfg/graphics/graphics2008/GraphicsSlides10.pdf. 2008.

[2] ISO/IEC. *International Standard ISO/IEC 14882:2017(E) – Programming Language C++*. https://isocpp.org/std/the-standard. 2017.

[3] Erik Reinhard et al. "Photographic Tone Reproduction for Digital Images". In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 267–276. ISSN: 0730-0301. DOI: 10.1145/566654.566575. URL: http://doi.acm.org/10.1145/566654.566575.