CSE306 - Computer Graphics

Ray Tracer

Author:
Diego Gomez

Spring, 2021

Contents

1 Introduction

Ray tracing is a technique of 3D computer graphics that is used to generate realistic images by shooting virtual rays of light and simulating the effects the environment has on them. In order to achieve satisfactory results one often needs to simulate an enormous number of rays. Nevertheless, in the recent years, with the improvement and commercialization of powerful GPU's this technique has become extremely relevant.

Some of the most notorious applications of this technique have been, the Pixar Movie "Monsters University" which was animated using Ray Tracing, its use in the newest generations of gaming consoles such as the PS5 and more generally in order to create beautiful images for several domains (architecture, marketing,etc). Youtuber "Stuff Made Here" also used this technique in order to create an Automatic pool stick https://www.youtube.com/watch?v=vsTTXYxydOE.

In the following we will implement a simple Ray Tracer. The implementation of the code is available on this repository https://github.com/diego1401/CSE306.git. This project was made as an assignment of the CSE306 "Computer Graphics" course at École Polytechnique, under the supervision of Prof. Nicolas Bonneel and Jiayi Wei.

We first started by implementing the important mathematical functions in the `classes/Vector.cpp` file. Then we defined the objects (firstly Spheres, then Geometries instances) and the scene, that is the place where our objects would be situated. Each object has an `intersection` function, which helped us determine whether a ray had crossed (intersected) the object.

After having the main functions we could start drawing the image. The basic idea is that we send from a given point in our scene, a ray object through a panel. The place where the ray intersects the panel let us assign to it a pixel coordinate. Then we follow this ray across the scene and assign a color to it according to the place where it "lands".

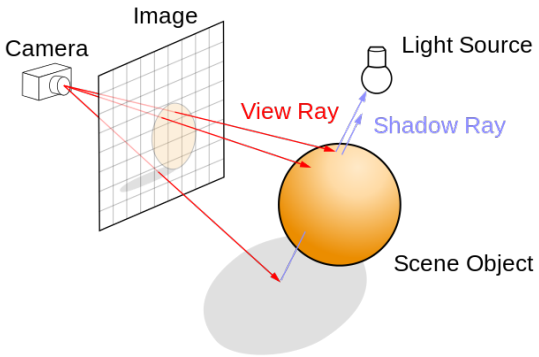

Figure 1: Illustration of the scene.

With all of this set, we where able to draw basic images as we can see in the Figure 2. In the following we will showcase and briefly explain the other features that were implemented in the ray tracer.

2 Diffuse and mirror surfaces

The Ray Tracer is capable of simulating simple diffusive and mirror surfaces. For the moment we are focusing in the effect of direct lighting, i.e light straight from the light source, and simple shadows. Shadows are simply defined as just being "visible" by the light source. In order to make the color scheme seem more natural we used the gamma correction technique. Let us look at some examples.

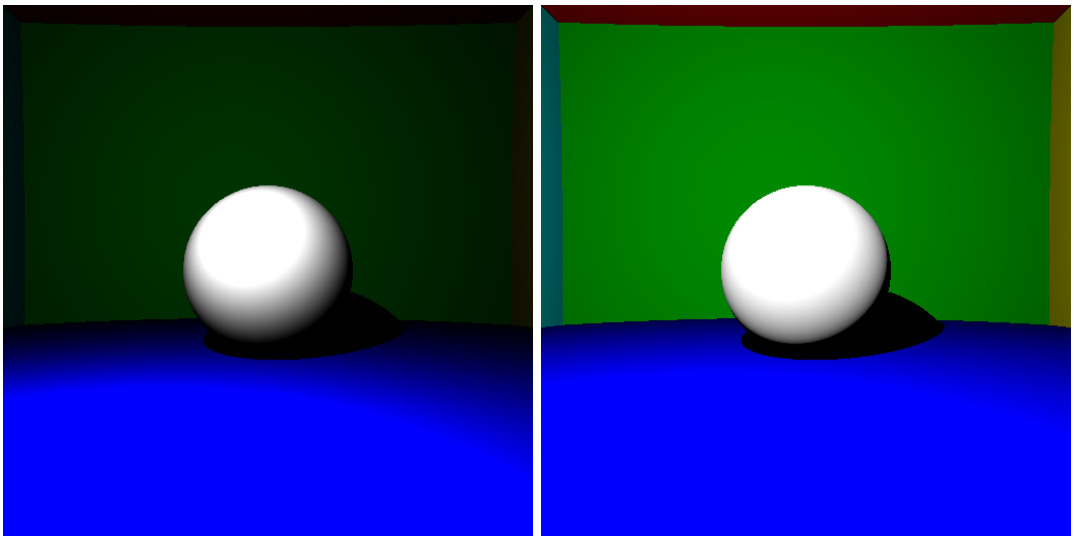

Figure 2: The first Image (left) without gamma correction ($I = 2.10^7$), and the second one with gamma correction ($I = 2.10^10$). These images were created in 89ms without parallelization.

Reflective (mirrors) and transparent surfaces were also implemented (see Figure 3). As now we can cross and bounce of off surfaces we had to define a variable `max_path_length` that dictates the number of times we can perform these actions. This was done in order to avoid the case where a ray bounces for ever. The Fresnel law was implemented in order to better imitate reality. This law simply put, allows us to give a bit of reflection to transparent surfaces.

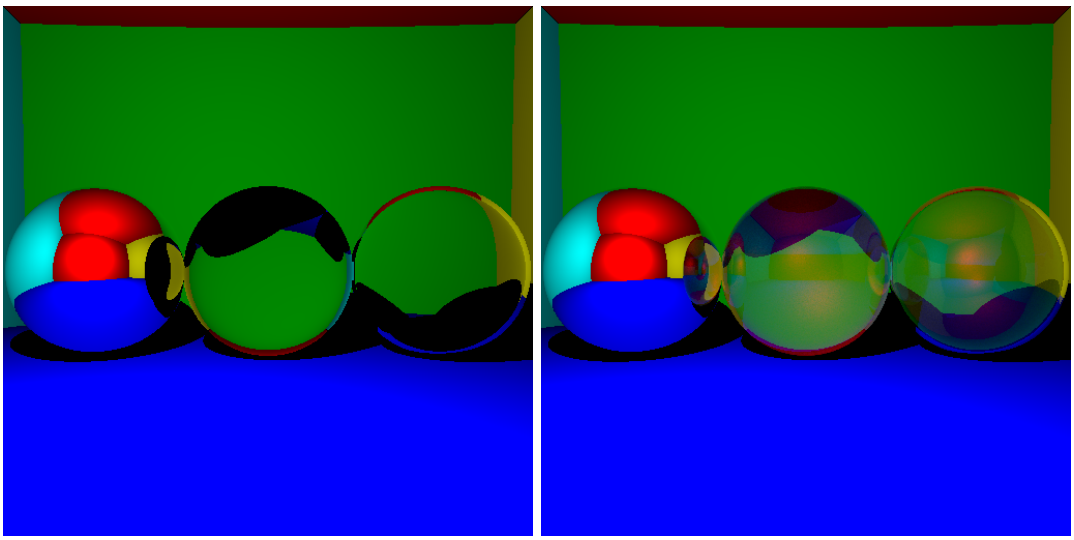

Figure 3: In both images we used a refractive index of 1.5. The first Image (left) without the Fresnel law was created in 145ms (without parallelization). The second image was rendered using 1000 rays (with a `max_path_length` of 5) in 1 min and 30 s (without parallelization). In both images we have (from left to right) a mirror sphere, a refractive sphere, and a hollow sphere.

3 Indirect lighting for point light sources

In real life the color we see in every object is not only result of the light its being shunned by. The light that bounces from other objects and into our objective object also affects this color. This was implemented with the help of Montecarlo integration, an example of this technique can be seen in the `monte_carlo_exo.cpp` file. The simulation of these spheres now require a much larger number of iterations, mainly because of the numerous stochastic techniques we are using (such as simulating the Fresnel law, and the previously mentioned Montecarlo integration). This means that we now must send several rays in order to obtain the color of a single pixel. That is why in order to increase the speed of our code we will now use parallelization from now on.

First let us look at a comparison between a white sphere with and without indirect lighting. (see Figure 4)

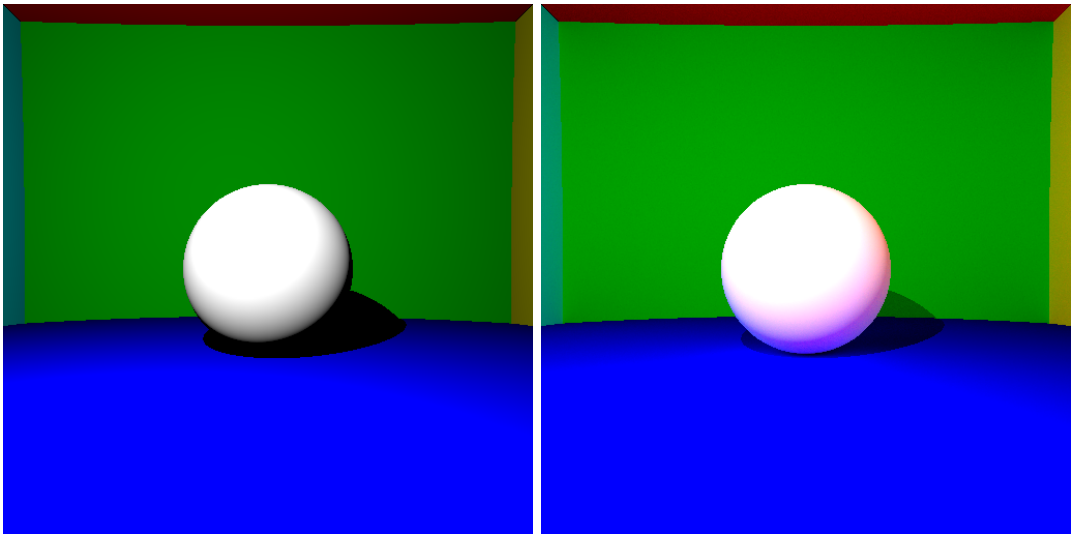

Figure 4: The left image is the same we presented in Figure 1. The right image is the one with indirect lightning implemented. It took 2 min 15s to render.

Note that our image looks more realistic. Since the light bounces on the walls the general lightning and shadows of the scene seems more natural. Moreover,the white sphere reflects a bit of the color of the wall behind the camera, that in this case is pink.

4 Anti-aliasing

Because of all the samplings that is being done, the adjacent pixels might not have a continuous progression of colors. That is why we use the technique of Anti-aliasing in order to improve the quality of our image. (see Figure 5)

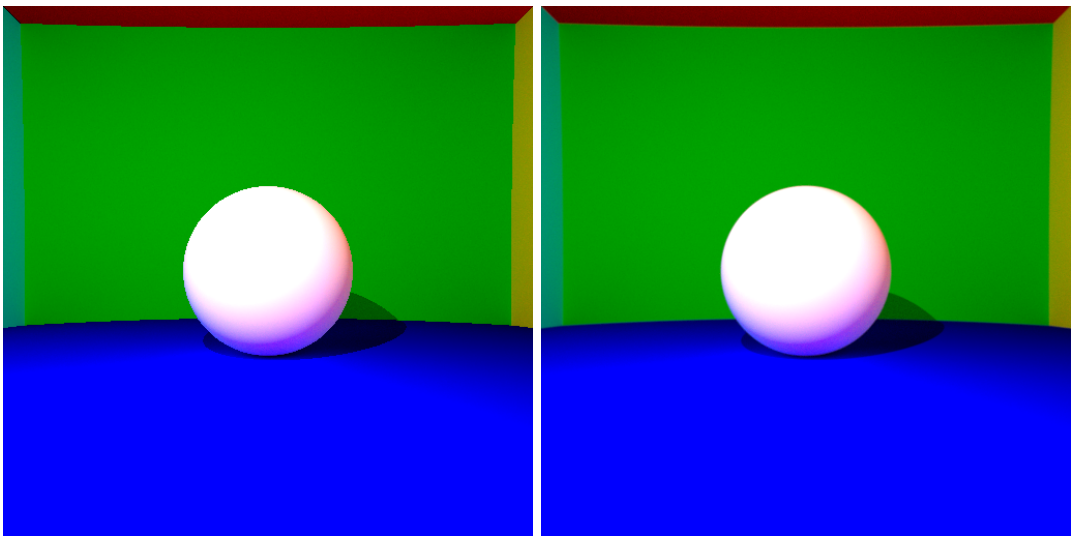

Figure 5: Comparison of image with(left) and without(right) Anti-aliasing. The image rendered in 2 min 15s.

The differences are subtle, but we can see that the right image is smoother than the left one. Mainly focus on the edge of the floor and walls, and the contour of the white sphere.

5 Spherical lights

In real life we usually are not dealing with point light sources. Spherical light sources were also implemented. (See Figure 6).

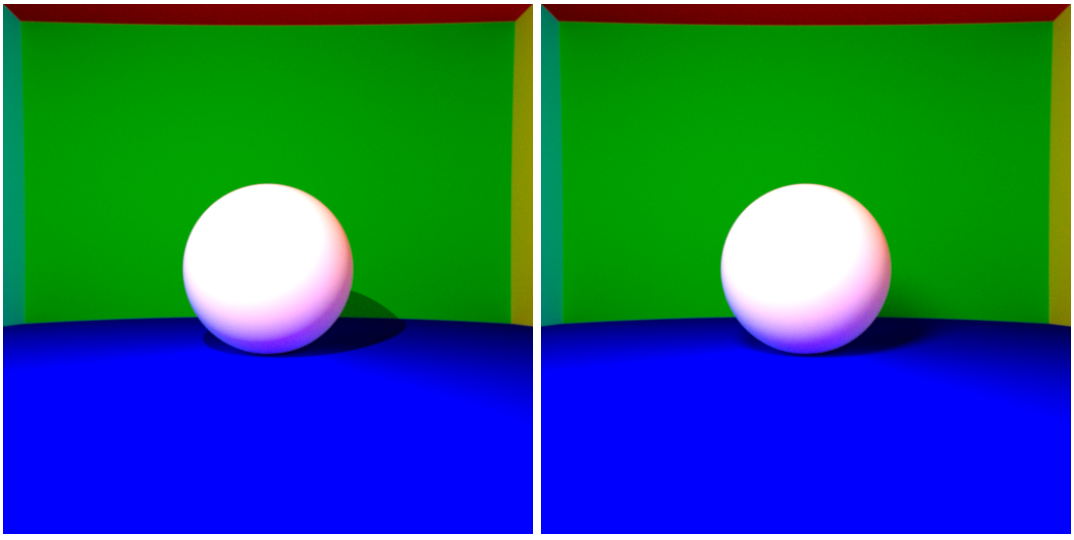

Figure 6: Comparison of image with(left) and without(right) Spherical lights. The image rendered in 2 min 50s.

We can see now that these leads to much more softer shadows. Even more impressive, we can now

simulate caustics. (See Figure 7).

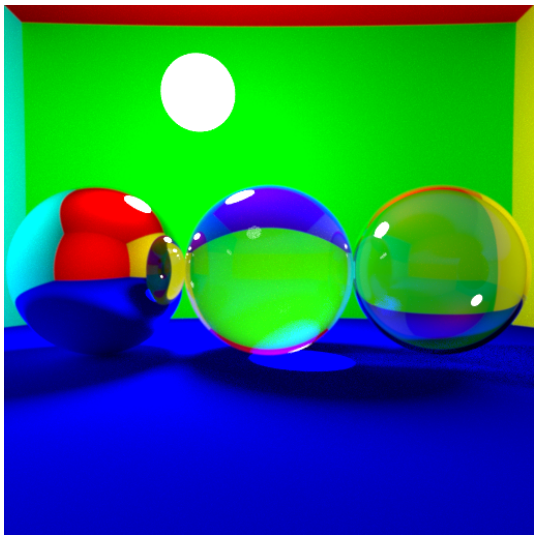

Figure 7: We moved the light a bit to position (-10,25,-10) and gave it a radius of 5. These indirect specular bounces are hard to capture and thus produce much higher levels of noise, we therefore used 5000 rays for this rendering. This image took over 14 min to render.

6 Depth of Field and Motion blur

With the current environment it is now easy to implement depth of field and motion blur to the Ray Tracer (See Figure 8).

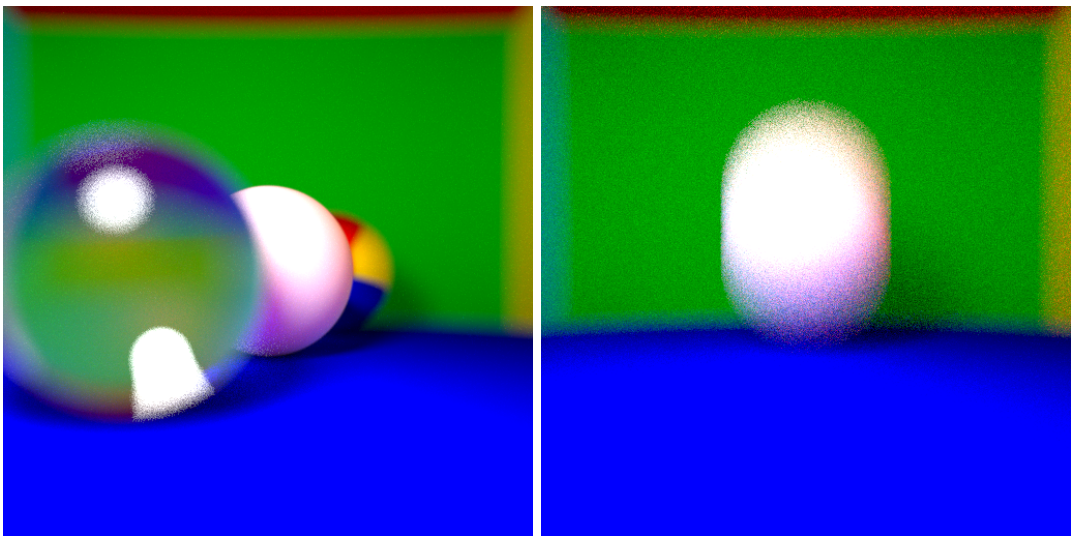

Figure 8: Image with Depth of field focused on the white sphere (left,rendered in 6min with 2000 sampled rays), and image with blurred motion(right, rendered in 4s with 32 sampled rays) using a simple linear motion.

7 Meshes

In order to render more general shapes, a way to render meshes was implemented.

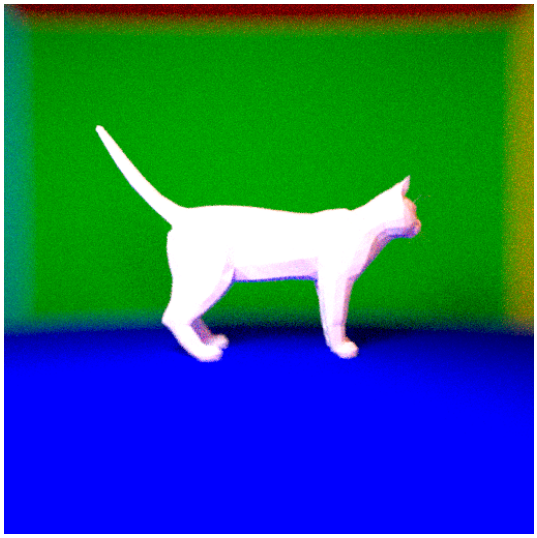

Figure 9: Rendering a cat mesh with a depth of 5 rays, and sampling 32 rays per pixel. Using the naive algorithm our Mesh is rendered in 25 min, using one bounding box it takes 3 min and finally with the BVH it takes 21s. This mesh was downloaded from https://perso.liris.cnrs.fr/nbonneel/cat.rar

8 Smoothing and texture

More information can be retrieved from the files we used to render the above mesh. Such as the normal vectors of each triangle composing the mesh to make the shape smoother. And add texture to our Mesh thanks to a mapping included by the artist in the `cat_diff.png` file. (See figure 10)

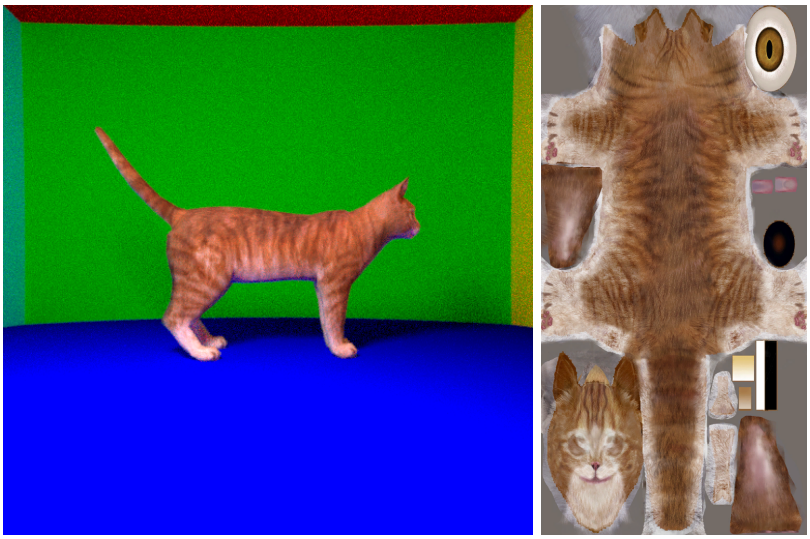

Figure 10: Image rendered in 20s. Right image is the texture of the mesh.

9 Other nice pictures

Finally here is a compilation of other images we can render with the Ray Tracer.

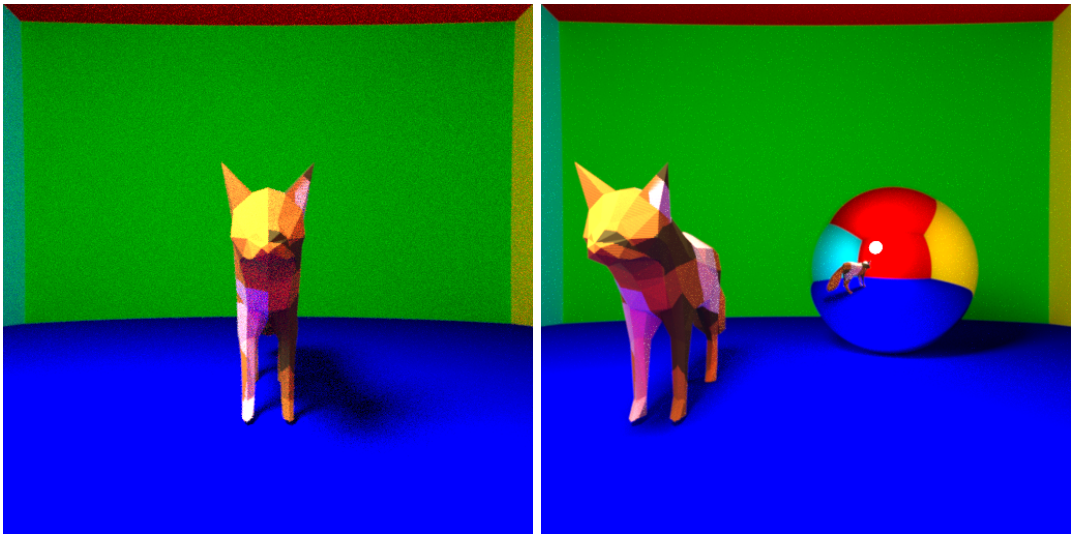

Figure 11: Downloaded new fox mesh from https://clara.io/view/1a03ac6b-d6b5-4c2d-9f1a-c80068311396. The fox mesh is composed of 290 triangles.

We obtained these images by tweaking the number of samples per ray (1000 for the mirror and fox picture, it took 10 min to render). And we also tweaked the DoF setting for the fox picture by choosing an aperture of 0.2 (it takes 10s to render). We turned it off for the second picture.