# Raytracer - Project Report

**Alexandru Cristache(mc14468), Sebastian-Stefan Stamen(ss15807)**

## Introduction

The purpose of this report is to describe the extra features and improvements we implemented to a basic raytracer renderer. Since, when it comes to speed in computer graphics, ray tracing is far from the best option, we chose to trade off rendering time and focus our effort towards improving the quality of the output images and making them resemble real life scenes.

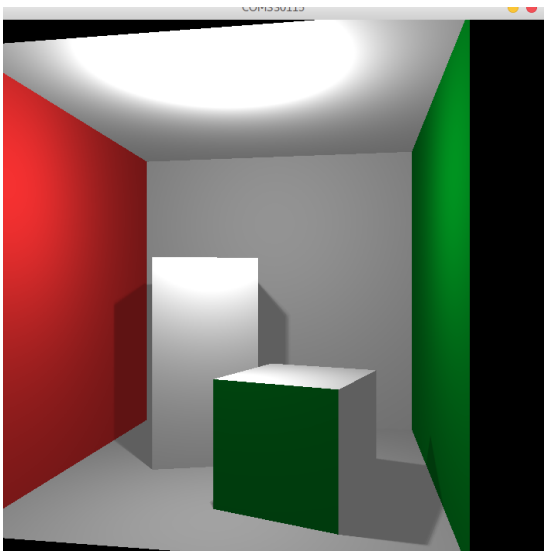


Fig.1: Basic raytracer

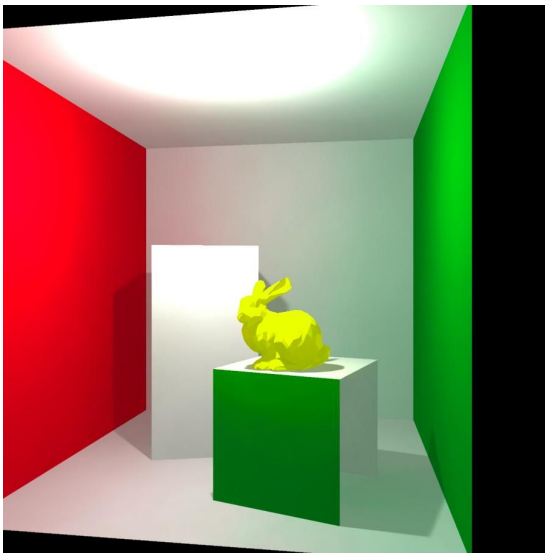
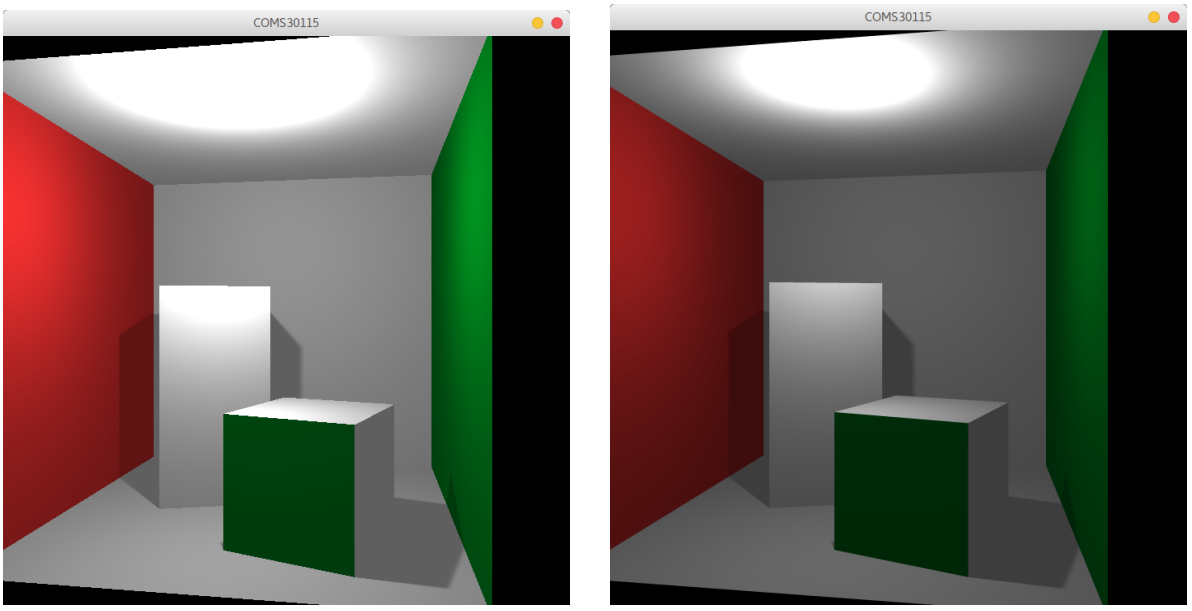


Fig.2: Final rendered image

## 1. Anti-Aliasing - SSAA

After finishing the basic raytracer, we first looked into possible ways of smoothening the jagged edges from the scene. With the same quality-vs-time mindset as before, we chose to implement supersampling instead of multisampling anti-aliasing, purely because the end result is superior.

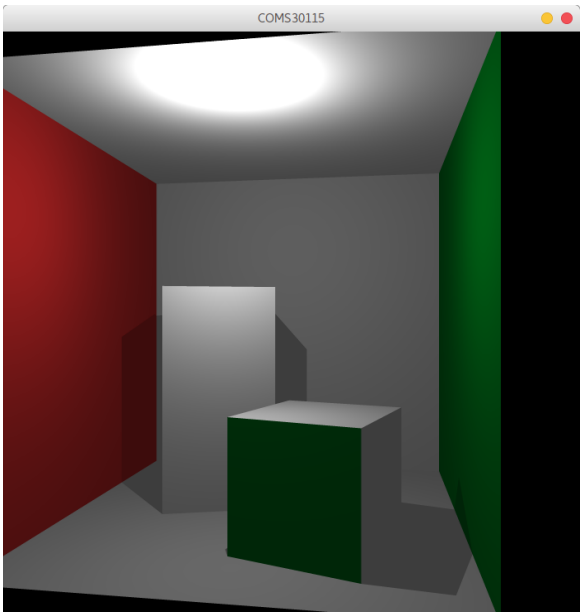
The idea behind SSAA is to “shoot” slightly disturbed rays for each pixel, therefore rendering the scene at a much higher quality. By taking an average of the samples around each original pixel, what we get is a smooth “down-sampled” image at the original resolution. Having carried out some tests, we figured out that 16 samples per pixel are more than enough to produce quality anti-aliasing regardless of image resolution.

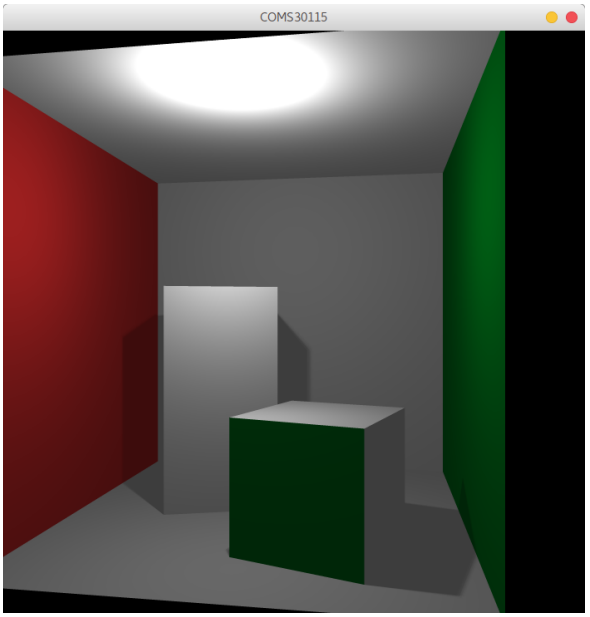



## 2. Soft Shadows

The next upgrade to our raytracer was improving the shadows we originally had. Even though the shaded areas looked decent, them having sharp edges made the shadows seem unrealistic (unless, of course, we assume the light source is infinitely small, which would not be the case in real life).

The way around this issue was, instead of having one single point emitting light, to randomly simulate multiple identical ones around a fixed 4D coordinate. This way, with enough such light sources, we can simulate the effect of a small area where the light comes from. The end result of this stage is a scene where the shadows are slightly faded at the edges, just as they would look in the real world.



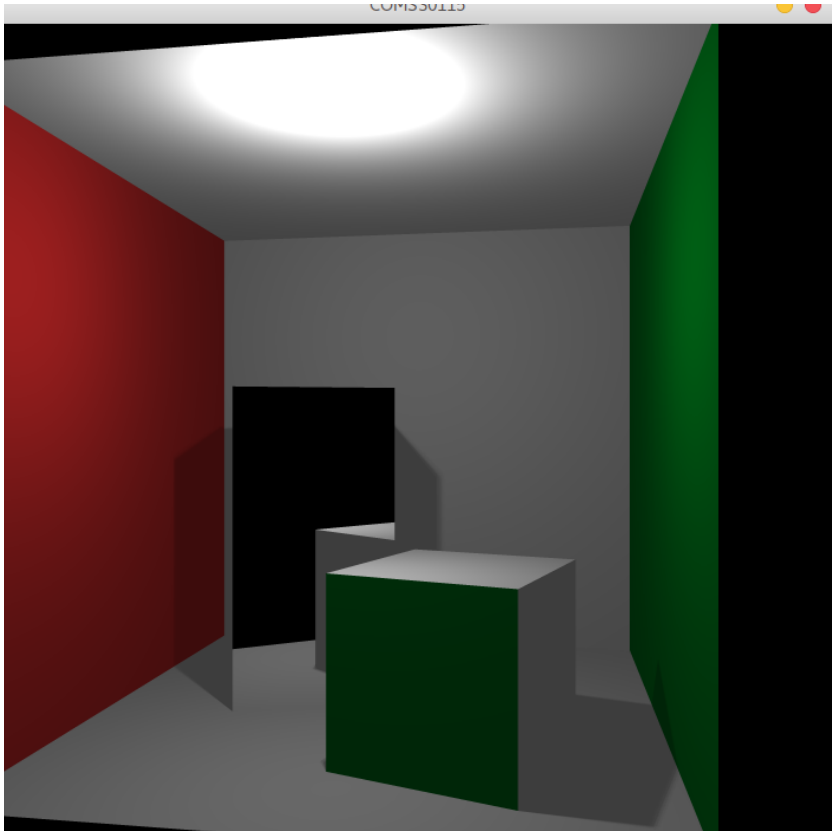



### 3. Reflection and Refraction

The following step we made was to give the input surfaces the ability to reflect and refract incident light. This would allow us to simulate phenomena like perfect and partial reflective mirrors, both of which are very common in the real world.

The idea behind perfect reflection is that, when a ray of light reaches a surface, it would bounce off following an angle congruent to the incident one. Since in ray tracing we assume the camera is the one that “shoots” rays at the world, then we follow its “reflected” ray towards the light source, we used a similar approach for mirrors. This time, if a ray hits a mirror, we consider that point to be a new “camera pixel” that shoots a new ray forwards. If that new ray reaches a part the scene, then that’s what the camera sees inside the mirror.

However, that is far from being feasible in a real-life environment. Mirrors are not perfect, however clear they might be. In reality, while a big part of the incident light bounces off the mirror, some of it goes inside the mirror.
Refraction is equivalent to the light transmitted to the object. The ratio between reflected and transmitted light is given by the Fresnel effect.



**4. Global Illumination - Photon Mapping**

In real world, any surface reflects a portion of the light it receives. This is the reason why even if we don't see the light source directly, we are still able to distinguish colours and see objects. In Computer Graphics there are three main techinques to achieve the effect of global illumination: Monte Carlo, Radiosity and Photon Mapping. Each of the three has pros and cons. Monte Carlo gives highly correct results at the expense of being slow. Radiosity does colour bleeding really well, but requires discretisation.

Photon Mapping, our weapon of choice, has low frequent noise and enables us to implement caustics reasonably easy (even though we have not) with the mention it is not exactly “correct”.

In order to achieve global illumination, we sampled nearby photon bounces at certain ray intersection points. Two popular choices to do this are radius search and K nearest neighbour. In our implementation we used radius search because K nearest neighbour might provide inaccurate results if there are not enough samples around a certain pixel.

## 5. Parallelising the Task

Even though we expected adding new features to the renderer to increase its running time drastically, the global illumination alone made it skyrocket. Rendering medium sized images then took several hours with a reasonable number of photons in the scene. Our first strategy to tackle this was to optimise the code serially in order to gain speed. However, the only optimisation that made a visible difference was implementing Cramer’s rule during the calculation for the closest ray-surface intersection. Even after this, though, a massive speedup was still needed to render decently sized images.

Consequently, we wanted to split the task among the cores of the host machine and parallelize it to increase the running speed. In particular, we aimed to use OpenMP to run all the “heavyweight” loops of the program. After this stage, the running time reduced considerably, allowing for rendering bigger sized and more complex images in reasonable amounts of time.

## 6. Overview and Possible Improvements

The end result of these stages is a Raytracer renderer that can produce rather realistic scenes and simulate many of the visual effects we see in real life. However, this comes at a considerable cost. As we stated

before, we designed those features only having quality in mind, up to the point where the lack of speed became an immense drawback.

Using OpenMP to parallelize the task did help with this regard, in that we were able to compute a rather impressive globally illuminated scene (see below), but only for the simple model. A more complex scene (>10.000 triangles) would take the current version days to compute at a medium resolution. One possible solution to this issue is to port the task onto the GPU instead of the CPU and render it there.

The program can be even further optimised by implementing more efficient data structures to facilitate searching for ray-surface intersections. Range trees or KD-trees are both valid options in the sense that they would both, on average, perform much better than a linear search does. Since this is where most of the ray tracing computations happen, these optimisations could probably reduce the task to a matter of hours.

Overall, ray tracing is a very powerful tool for computer graphics (and incredibly fun to implement), as it mimics the exact phenomena that happen in the real world. Our implementation covers many features on top of the basic functionalities, and the results far exceeded our initial expectations. Even though it is not suitable for real-time graphic applications, ray tracer rendering is able to produce outstanding images of a virtual scene.