Raytracing project - CSE306

Alexandru-Nicolae Serban

1 Introduction

We will present in this report the construction of our raytracer and all the features that it supports, including diffuse, mirror and transparent surfaces, direct lighting, shadows and indirect lighting for point light sources, antialiasing and ray-mesh intersection, including BVH. Throughout the project we followed the directions from the Lecture Notes.

We will begin by introducing the major classes that we implemented in our project. Two classes of high importance are `Sphere` and `TriangleMesh`, which create the main objects we deal with, namely spheres and a cat given as a triangle mesh. Both of the classes possess a method called `intersect`, which checks if an object of the class is hit by a given ray or not and stores the information about the intersection, such as the point of contact, the normal in the point, the albedo of the object and properties related to the surface. Another relevant brick in our raytracer is represented by the class `Scene`, which deals with shadows, properties of the hitting surfaces, as well as direct and indirect lighting.

Now that we introduced the main components in the code, we can describe the scene that we used for emphasizing the features of our raytracer. The exterior consists of six big spheres of radii between 940 and 990, positioned along exactly one of the three axis at distance 1000 from the origin. These correspond to the four walls, the ceiling and the floor. The floor is blue (*i. e.* albedo $=(0, 0, 1)$), the ceiling is red (*i. e.* albedo $=(1, 0, 0)$), the wall in front is green (*i. e.* albedo $=(0, 1, 0)$), the left wall is turquoise (*i. e.* albedo $=(0, 1, 1)$), the right wall yellow (*i. e.* albedo $=(1, 1, 0)$) and the wall behind is purple (*i. e.* albedo $=(1, 0, 1)$). The camera was positioned at the coordinates $(0, 0, 55)$ and it sent rays from each pixel in the 512x512 grid. The light source that we used consisted of a point located at position $(-10, 20, 40)$, sending light in all directions. In order to test the bounces of light, we added first three small spheres in the center of the image, one acting as a mirror, one transparent and one hollow. In the end, we replaced the balls by a cat which was given as a collection of triangles.

2 Reflective, refractive and diffuse surfaces

The first feature that we added to the raytracer consisted in the introduction of the Lambertian model for casting the shadows and dealing with the diffuse surfaces, which represent materials that scatter light equally in all directions. In order to determine the points that lie in the shadow and which would have to be black (*i. e.* albedo $=(0, 0, 0)$), we checked if they are hit and they are obstructed by another object from the light source. This was done by sending a ray from the point towards the lighting source and checking if it intersects another object before reaching the light source. For all the points that were not in the shadow we used the equation of the Lambertian model, namely that the light reflected off the surface is $L = \frac{I}{4\pi d^2} \frac{\rho}{\pi} \langle N \omega_i \rangle$, where, as in [**?**], d is the distance between the light source and the intersection point, N is the normal, ρ is the albedo of the point and ω_i is the normalized direction of the ray. We also took care of the numerical precision errors by adding a very small portion of the normal to the intersection point for making sure that we do not find the same object in the intersection. In order to showcase the behaviour of the diffuse surface and of the shadows, we used just one diffuse sphere in the center of the image of albedo $(1, 1, 1)$ and we applied a gamma correction of 2.2 for enhancing the values of the pixels for a more colourful image. We obtained the image from Figure **??** for a light intensity of $2e10$.

We then introduced reflection and refraction to our raytracer by constructing the function `Scene::getColor`, in a recursive manner, which allows light to bounce off the objects, changing the color of the hitting points depending on the light received from their surroundings. We handled reflection by adding a bool attribute

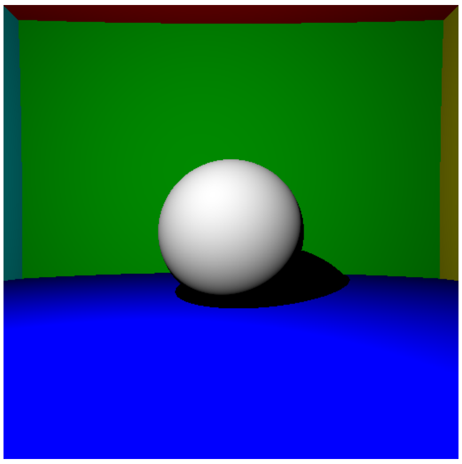

Figure 1: The image obtained by applying gamma correction of $\gamma = 2.2$, for a light intensity of $2e10$ in about 75 miliseconds.

to the `Sphere` class which stores whether the ball is a mirror or not. In the former case, we redirect the light in the direction obtained by the formula $\omega_i - 2\langle N, \omega_i \rangle N$.

For the refractive surfaces we initialized a bool attribute to the `Sphere` class which stores whether the surface is transparent or not. In the former case, we used Snell-Descartes law for computing the refracted array. A corner case that we had to study was the total internal reflection case, which melt down to just reflection. We treated all the transparent surfaces as glass with a refractive index of 1.5 and all the surfaces that were neither mirroring not transparent as diffuse.

The last kind of sphere that we introduced was the hollow sphere, which we simulated, as hinted in the lecture notes, by two transparent spheres, one contained in the other, by reverting the normal of the inside sphere. In order to keep track of whether a surface is inside another sphere, we used a bool attribute in the `Sphere` class. Every time this bool was true, we reverted the normal obtained in the intersection.

We also added Fresnel reflection for transparent surfaces in order confer our scene a more realistic characteristic, as , in practice, a ray is both partially reflected and refracted. In order to ameliorate the noise resulting in the randomization between the reflective and refractive behaviour of each pixel we send multiple rays from each pixel of the camera and average the results. A comparison between the image with and without Fresnel can be observed in Figure 2 and Figure 3

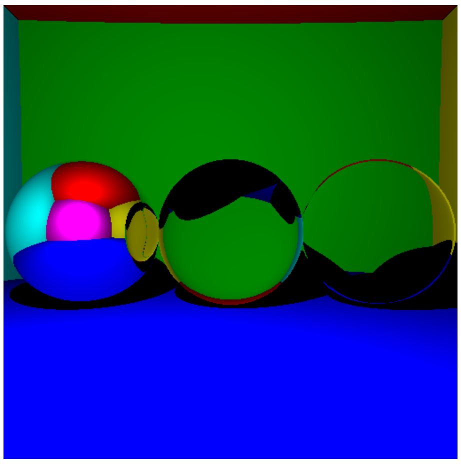

Figure 2: The left ball is fully reflective (mirror), the center one is fully refractive (lens) and the third one is hollow with refraction. The rendering took about 95 miliseconds.

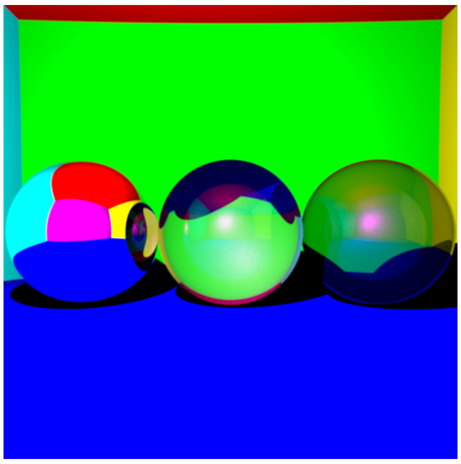

Figure 3: We added the Fresnel reflection and we sent 1024 rays per pixel. We notice the purple marks on the 2 spheres on the right, given by the partial reflection. The rendering took about 5 seconds.

3 Indirect lighting

In order to make our raytracer more realistic, we considered indirect lighting, as, in reality, not only the objects that are directly hit by a source of light have color. That's why we added the influence of indirect lighting from the rendering equation in our `Scene::getColor` function by approximating the integral via Monte Carlo integration. As noted in the lecture notes, the only problematic case is represented by diffuse surfaces since in the analysis of reflection and reflaction, we actually worked with indirect lighting. Thus, we used an importance sampling technique, derived from the Box-Muller algorithm, as can be seen in the function `Scene::random_cos`. Again, in order to ameliorate the randomness, we need to send multiple rays per pixel and average in order to get a good enough approximation. One can check the influence of the indirect lighting in the Figure 4.

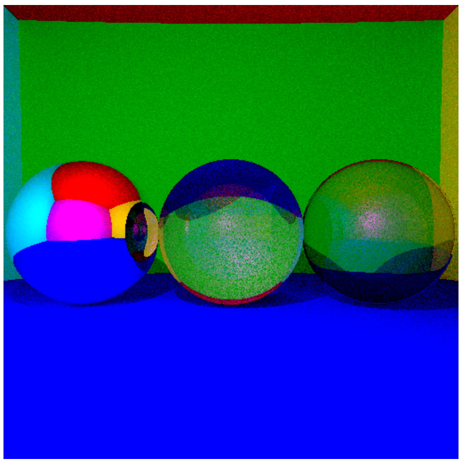

Figure 4: The image obtained by adding indirect lighting using 32 rays per pixel and a maximum ray depth of 5 (*i. e.* 5 bounces allowed). The rendering took 2.5 seconds. We used parallelization.

At this point, we realized the presence of aliasing given by the fact that we only consider rays from the center of each pixel. Thus, at every ray sampling we hit the same pixels, creating big differences in color between pixels found at the border between the small balls and the walls. Thus, in we order to reduce the noise, we utilize again importance sampling, using this time the actual Box Muller technique for adding a small Gaussian noise to the origin of the ray. This way, we obtained a smoother transition between surfaces, as can be seen in Figure 5 and Figure 6.

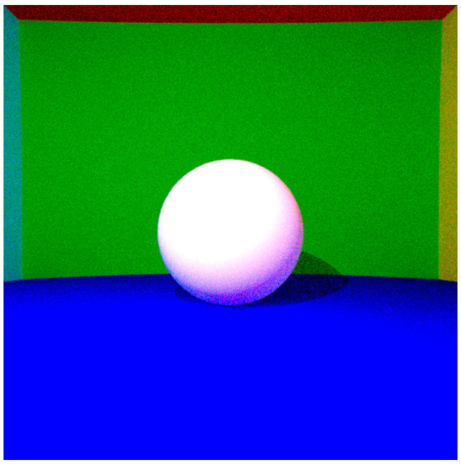

Figure 5: We simulated the image with just the diffuse white ball under the effect of antialiasing with 32 rays per pixel.

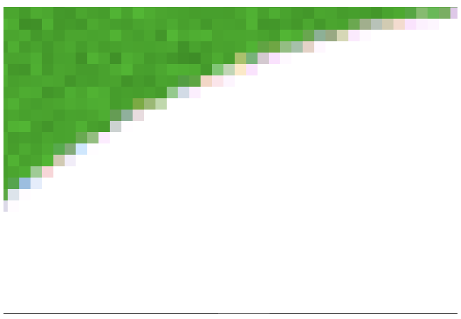

Figure 6: The effect of antialiasing for the pixels that determine the contour of the sphere.

4 Ray mesh intersection

Since in practice we often deal with objects that are not spherical, we moved on to working with a new class of objects, triangle meshes in the class `TriangleMesh`. The reason for considering such as class was based on the observation that we can approximate most objects by a large number of triangles sharing only the edges. Thus, it makes a lot of sense to build our ”star” object, in this case a cat, as a triangle mesh. For loading the cat, we used the mesh loader provided in the notes as the method `TriangleMesh::readOBJ`. In order to use the functions implemented in the first part, we initialized this class as derived (as well as the `Sphere` class) from the abstract class `Geometry`. So, the only issue that we encounter is the construction of a ray mesh intersection function.

The technique that we used is the variant of the Moller-Trumbore algorithm presented in the lecture notes. Namely, for each triangle in the mesh we compute the barycentric coordinates of the intersection point between the ray and the plane generating by the triangle. If all the coordinates are between 0 and 1, the intersection point lies exactly inside the triangle, so we can compute the distance between the origin of the ray and the point. After checking all the triangles in the mesh, we select the closest one as the intersection. With this algorithm we can easily obtain the intersection between the ray and the triangle mesh and we can just use the `Scene::getColor` as before. The result of this algorithm can be seen in Figure 7.

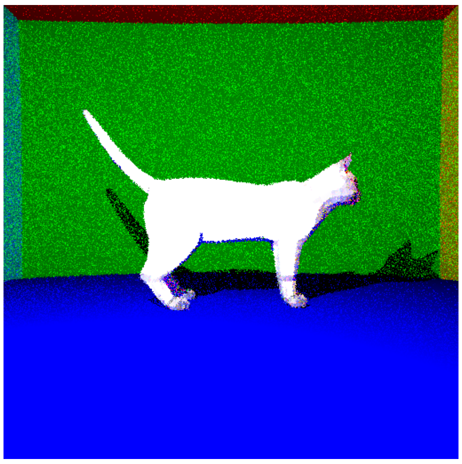

Figure 7: The image obtained by using the Moller-Trumbore algorithm with 1 ray per pixel and a maximum ray depth of 5. We scaled the cat by a factor of 0.6 and translated by the vector (0,-10,0). The rendering took 20 seconds. We used parallelization.

Since the rendering was very slow, we spent the rest of the project, bolstering the speed of our raytracer. For this, we first constructed a bounding box for the cat, given by the extremities of the cat (the minimal and, respectively maximal, vertex coordinates in each direction). So, now, instead of checking for each ray the intersection with all triangles we only apply the Moller-Trumbore algorithm to the rays that intersect the bounding box. This strategy yielded an almost 6 times speed up but it was still slower than what we wanted.

The last technique that we used, which improved massively the computational time, was the Bounding Volume Hierarchies(BVH). For this, we implemented a binary tree, in a recursive manner, whose nodes were bounding boxes obtained by splitting the parent's triangles according to the barycenter along the longest axis of the box. For constructing the BVH tree, we used the method akin to *QuickSort* that was presented in the lecture notes, in which we rearrange the order of the triangles in order to keep the ones that are alike together and to only keep track of the starting and ending indices of the triangles corresponding to each bounding box. Finally, for integrating the method in our computations, we implemented the DFS traversal presented in the lecture notes and obtained the final image, presented in Figure 8.

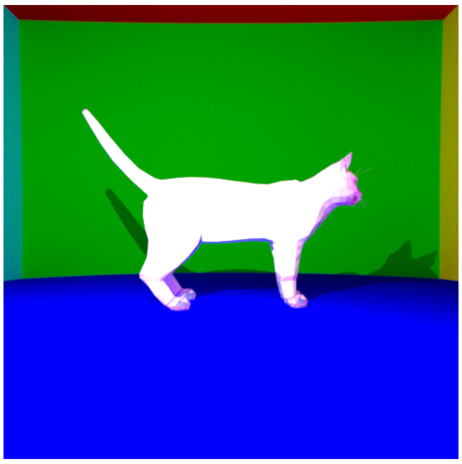

Figure 8: The image obtained by using BVH with 1024 rays per pixel and a maximum ray depth of 5. We scaled the cat by a factor of 0.6 and translated by the vector (0,-10,0). The rendering took around 8 minutes. We used parallelization.

5 Conclusions

In this report, we implemented a raytracer which handles diffuse, reflective and refractive surfaces. We also implemented the Fresnel reflection. Moreover, we are also able to handle indirect lighting and antialiasing. Lastly, we dealt with ray mesh intersection and we implemented BVH. The code can be found at https://github.com/alexserban2002/CSE306.git.

For elaborating of the project, I discussed some aspects of the implementation with my colleague Cezara Petrui.