3D PROGRAMING

Assignment 1 - Report

Group 4	
Catarina Gonçalves	90709
Guilherme Monteiro	90724
Pedro Leitão	90764

For this assignment we had to implement the Turner Whitted Ray Tracer and extended it using the techniques of Distribution Ray Tracing covered in theoretical classes. Then we extended our ray tracing application by using both BVH and a Grid acceleration structures. In addition, we also implemented an **extra**: the fuzzy reflections.

1. Turner Whitted Ray Tracer

We started by implementing the *primaryRay()* (camera.h) function to create a ray for each individual pixel in the viewport. Then, we being to implemented the *rayTracing()* (main.cpp) function where we shoot all the primary rays.

Afterwards, the result of those implementations was just the background color, we implemented the **intersection** of the rays with the objects. At this initial stage, we only implemented the intersection with spheres and, as you can see in Fig.1, the spheres only had a solid color. Throughout the implementation of the assignment, we eventually implemented all the necessary geometric intersections: for triangles, planes and axis-aligned boxes – where we used the Kay and Kajiya algorithm.

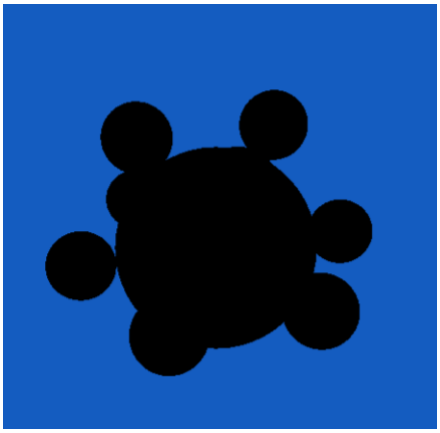

Figure 1- Sphere Intersection before Blinn-Phong illumination model.

To calculate the color of the objects, we implemented the **Blinn-Phong illumination model** that considers both the Specular as well as the Diffuse colors. Afterwards, we implemented **Hard Shadows** where we have to cast a secondary ray from the hit point in the direction of all light sources. If there is an intersection, it is in shadow. Otherwise, it is not. During this implementation, we came across *acne spots* that are the result of self-intersections. To fix this, we displaced the hit points to ensure they were outside the object. This was the result of these implementations (Fig. 2):

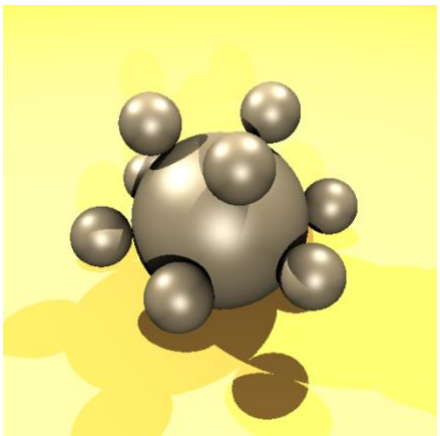

Figure 2 - Blinn-Phong illumination model and Hard Shadows

Our next task was to deal with global color illumination, starting with implementing the mirror **Reflection**. To do so, we calculated the reflected ray's direction and traced the ray, displacing the hit point to avoid acne spots as referred above. Afterwards, we moved on to **Refraction** using the Snell's Law and the Schlick approximation of Fresnel Equations for dielectric materials. During this implementation, we had to fix some issues such as using positive dot products (by checking the max between our dot product and 0) as well as checking if the rays hit from inside or outside.

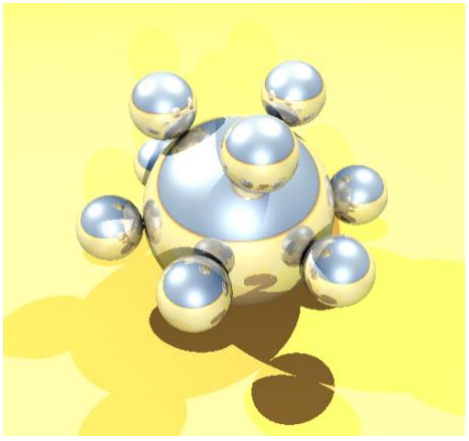

Figure 3 – Mirror reflection

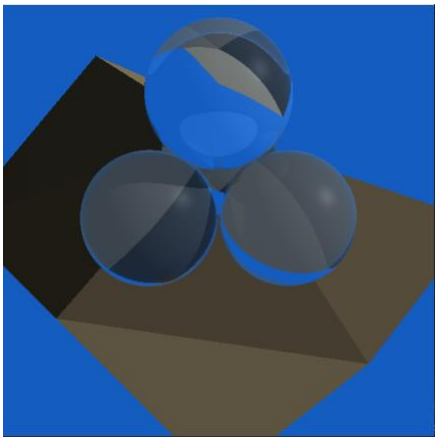

Figure 4 – Refraction and Ray-Triangle intersections

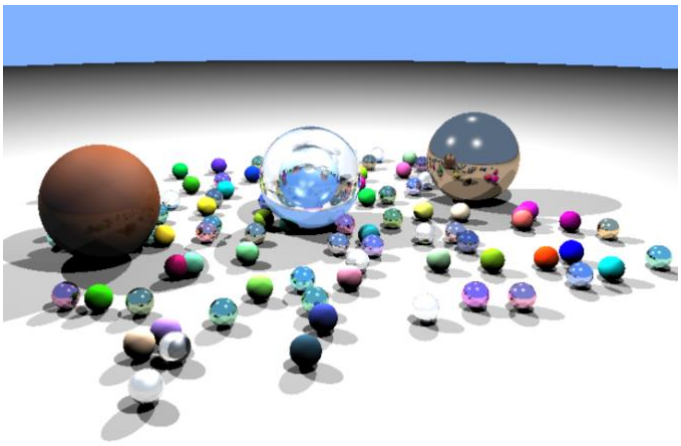

Figure 5 - Reflection and refraction

2. Stochastic Sampling Techniques

Then we moved on to Distributed Ray Tracing, which allows smother scenes. We started with the implementation of **Anti-aliasing** using *jittering*. With this technique, we divide our pixel into a grid n*n, shoot several rays per pixel, each from a random position in that pixel's cell and then we average each ray's color contribution. As you can see in Fig. 6, all the edges of the spheres as well as the shadows became smoother in comparison with Fig. 3.

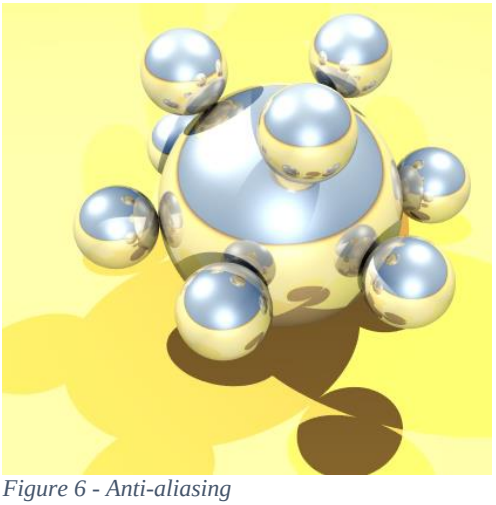

Figure 6 - Anti-aliasing

Afterwards, we implemented **Soft Shadows** which consists in distributing our rays over light source areas as oppose to light points. There are two cases of soft shadows: with or without antialiasing.

With anti-aliasing (Fig. 7), we get a smoother transition between the shadows and the light since we represent the area light as an infinite number of point lights and choose one at random for each primary ray. Since we are using the jittering technique, we shuffled our array in order to avoid any correlation between the light and pixel array's. Then, we trace our shadow ray.

Without anti-aliasing (Fig. 8): we represent the area light as a distributed set of N point lights, each with one Nth of the intensify of the base light. Then, we trace our shadow ray as usual.

These were the results of both cases using 16spp:

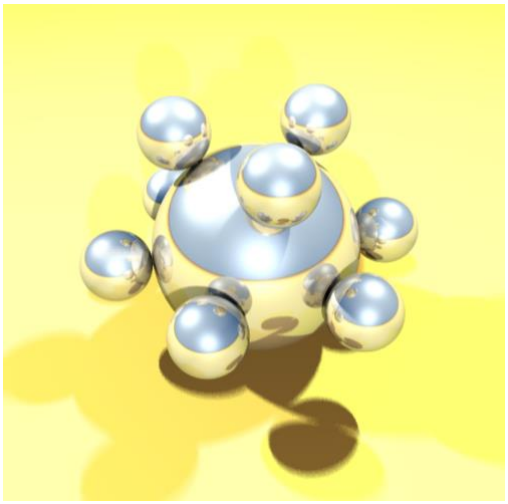

Figure 7 - Soft shadows with anti-aliasing

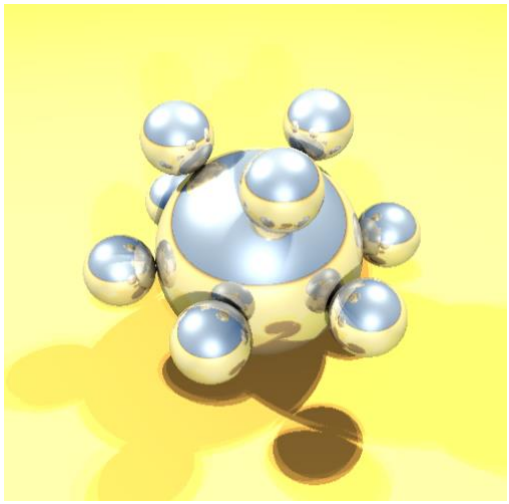

Figure 8 - Soft shadows without anti-aliasing

To end this section, we implemented the **Depth of Field** where we had to simulate the optical properties of a thins lens. This notion comes with two parameters: *aperture*, that determines the blurriness of the objects out of focus (the bigger the aperture, the more blurred the objects will be), and *focal length*, the distance along the view direction at which things will be in focus. We had to compute the point where the center ray hits the focal plane as well as obtain a sample point in our pixel to calculate the ray's direction. To do so, we implemented the other *primaryRay()* function to shoot a primary ray from a random position in our lens in the typical direction of that ray. This were our results for different apertures:

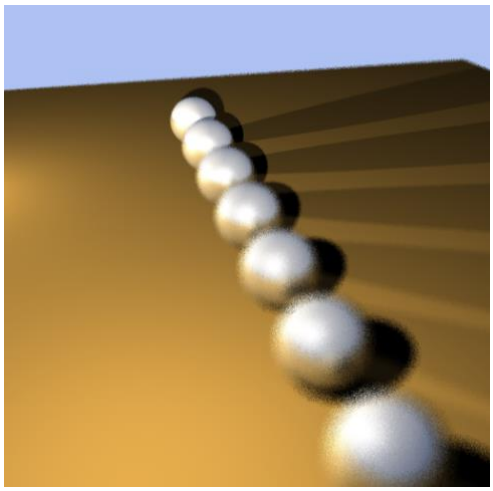

Figure 9 - Depth of Field, lens aperture 12

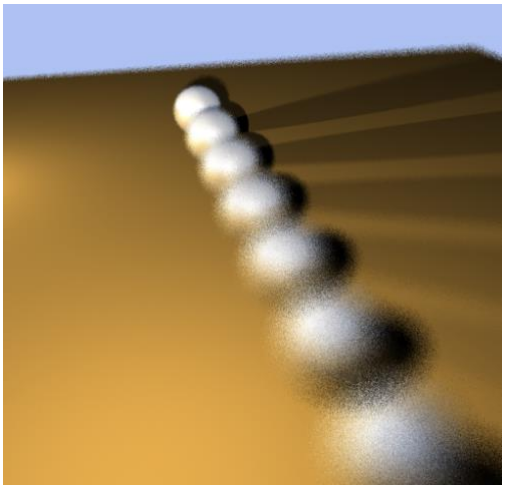

Figure 10 - Depth of Field, lens aperture 30

3. Acceleration Structures

In order to improve the performance of our ray tracing, we implemented two different acceleration structures whose goal is to reduce the rendering times for large number of objects by reducing the number of ray-object intersections.

3.1 Uniform Grid

With the uniform grid, our scene is divided into several equal-sized cells that will contain the objects (Fig 11). Note that to discover the cells that contain a certain object, we check the overlapping of the objects bounding box with the cells (Fig. 12). This way, the ray that hits the grid only passes through certain cells which means that it only intersects the objects in those cells. So, once the grid is built, we cast a ray, check the cells it traverses and test the intersections with the respective objects.

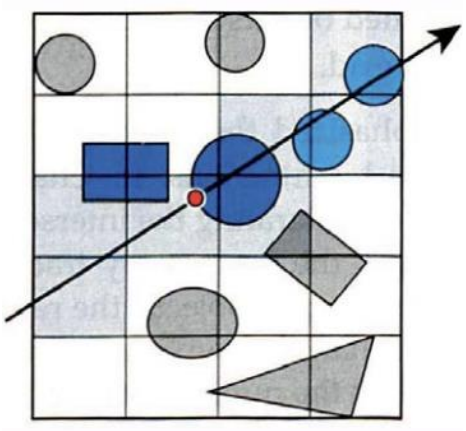

Figure 11 - Ray traversing the grid

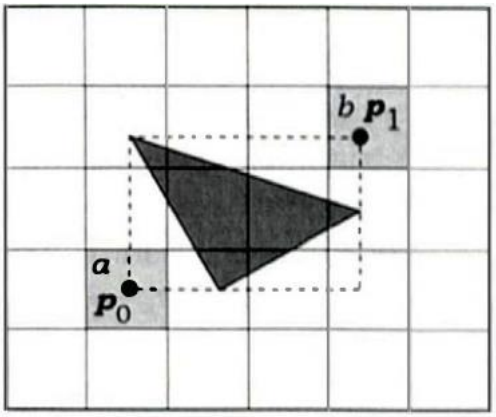

Figure 12 - Object bounding box overlapping grid cells

The code for the grid was provided so we only had to adapt it to our existent code. To do so, we called the grid build function in the *init_scene*() and checked for the intersection of the primary rays as well as for the shadow rays in our ray tracing function.

3.2 Bounding Volume Hierarchy

With the Bounding Volume Hierarchy, we organize the objects in a hierarchy of disjoint sets, i.e., a tree. This means that we subdivide our initial node that contains all the scene's objects into two different child nodes, left and right, each containing a certain number of objects. This is done recursively for each child node until a certain threshold is met, becoming a leaf node. Since the build function, that creates the root node, was already implemented, we had to implement the build_recursive() and traverse() *functions*. Let us explain our implementation in more detail:

3.2.1 Build recursive function:

In order to build our hierarchy, we start by finding the largest axis which is the one with the largest range of centroids and sort the elements in that dimension. Then we need to find the midpoint that divides our node in left and right side. To do so, we simply find the centroid of the largest axis. However,

since this may lead to child nodes with no objects in them, we check if any of the sides created by our initial midpoint are empty. If they are, we find a new midpoint by using the **mean splitting** discussed in class and repeat the verification. If even after this we haven't obtained a decent midpoint, our split_index will be equal to the sum of the left_index with the established threshold. Otherwise, it will be equal to the index of the first object at the right of our previously calculated midpoint.

Then, we create the child nodes, their respective bounding boxes and call the build_recursive() function for each node.

3.2.2 **Traverse** functions:

It was time to implement the traverse functions: one for primary rays and the other for shadow rays. The only difference in these two functions is that for shadow rays we don't need to check and store the closest intersection and hit object.

To check if a primary ray traverses, we start by checking if it intercepts the world bounding box. Then, if it does, we check the intersection with its children inside a while(true) loop until we reach a leaf. If the ray hits any of the child nodes, the current node will be updated and become the intersected one. If it hits both, the current node will become the one that is closest and the other will be added to the stack. Once we reach a leaf node, we test the intersections with the objects contained by that node, storing the closest one. Afterwards, if our node stack is not empty, we recursively pop the top node in our stack and check if it is closest than our current node and, if it is, we update it.

For shadow rays, we implemented a very similar function without the closest intersection's verifications. However, our shadows had some noise in them that we weren't able to fix in time for the demonstration as you can see below:

Figure 13 - Dragon scene with Uniform Grid

Figure 14 - Dragon scene with BVH

Here is a table containing the rendering times in seconds for the mount_high.p3f and mount_very_high.p3f scenes of each acceleration structure since there is no visual difference:

Acceleration Structure	mount_high.p3f	mount_very_high.p3f
None	87.30 s	1200.00 s *
Uniform Grid	2.39 s	2.77 s
Bounding Volume Hierarchy	1.74 s	2.16 s

*we stopped the timer since it was taking a lot of time to render the scene.

4. Extra – Fuzzy Reflection

Since ray tracing only supports perfectly sharp mirror reflections, we implemented the **fuzzy reflections** as an extra. To do so, we have to shoot the reflected ray in the direction of a random point in a small sphere, whose centre is where the normal reflection ray hits and radius is the roughness parameter that we establish. Then we checked the product between the normal and the fuzzy direction to avoid the ray being below the surface. This was the result:

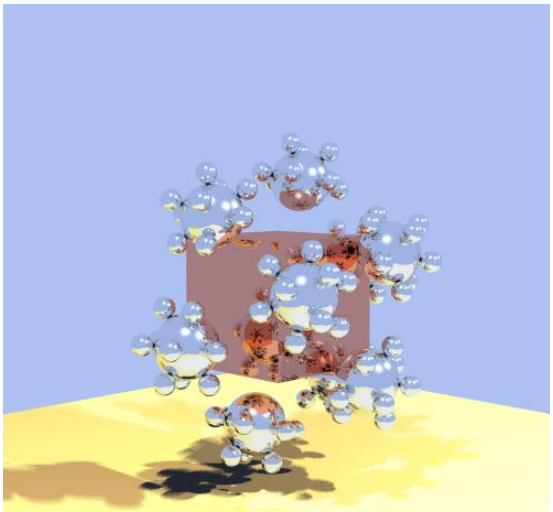

Figure 15 - Without fuzzy reflection

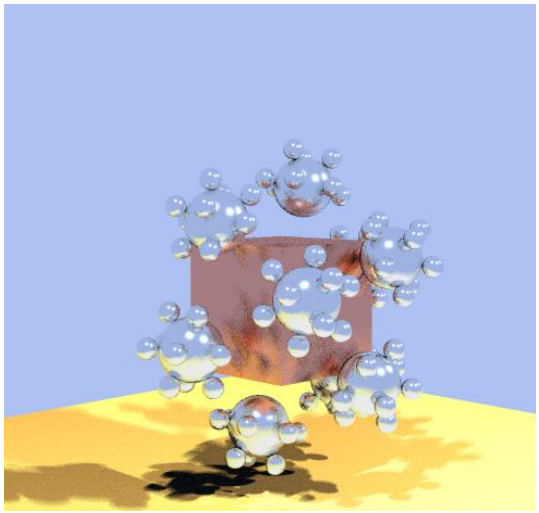

Figure 16 - With fuzzy reflection