3D PROGRAMING

Assignment 2 - Report

Group 4

Catarina Gonçalves 90709
Guilherme Monteiro 90724
Pedro Leitão 90764

For this assignment, we had to implement a Progressive Ray Tracer using the GLSL language. To do so, we had to implement some functions in the two files provided, the *P3D_RT.glsl* and the *common.glsl*, regarding the ray, the intersections with objects and the scattering for different types of object materials. We also implemented the camera rotation and created two new boxes, one metallic and one diffuse, as extras.

1. Ray intersections with objects

We started by implementing the *getRay()* function, in the common.glsl file, based on the notion of Depth of Field. To do so, we had to calculate the eye offset as well as ray direction in order to create the ray.

Then, we implemented the ray-object intersection functions, also in the common.glsl file, to calculate if the object is hit by a certain ray. The *hit_triangle()* function is based on the Tomas Moller-Ben Trumbore algorithm and uses the barycentric coordinates to check its intersections. The *hit_sphere()* uses the sphere's center and radius, while the *hit_movingSphere()* also uses another center and the time values.

All of these functions are called by the hit_world function, in the P3D_RT.glsl file.

2. Local color, multiple source lights, hard Shadows

To achieve these tasks, we started by implementing the *directlighting()* function, in the P3D_RT.glsl file, based on the Blinn-Phong color model. We calculated the light vector, the halfway vector and the dot product between the normal and the light vector. Afterwards, and since our materials didn't have the specular, diffuse and shine components, we decided to assign them according to their type and to the material properties of the objects in our last assignment. Consequently, if we were dealing with diffuse materials, their diffuse color would be equal to its albedo and its shininess would be equal to 10. The same applies to metallic materials and their specular color but its shininess would be equal to 500. When it comes to dielectric materials, we assign its diffuse color to vec3(0.0), its specular color to a vec3(0.04) and its shininess to 220. Then we calculated the color. Afterwards, we added multiple light sources to our color and multiplied each one with the throughput.

Regarding the implementation of hard shadows, we added a verification in our directlightning function, that calls the *hit_world* function to verify the intersection of the shadow ray with all the objects in the scene, in order to only calculate the color when the ray is not in shadow.

3. Global color

3.1. Diffuse Reflections for color bleeding

In order to achieve these reflections to make our objects take on color of their surroundings and modulate that with their own intrinsic diffuse color, we calculated the reflected ray direction considering a point on a non-existent sphere in order to allow a deviation in the direction. Then, we created the ray taking in consideration self-intersections (acne spots).

3.2. Mirror and Fuzzy Specular Reflections for metallic objects

To achieve these reflections, we calculated the reflected ray direction considering a point on a non-existent sphere in order to allow a deviation in the direction. Then, we multiply it with the roughness of the material in order to allow fuzzy reflection. Since the roughness is one of the attributes of metal materials and some metallic objects don't have roughness, this implementation allows, simultaneously, mirror and fuzzy reflections. .

3.3. Mirror Reflection and Refraction for dielectric objects

For dielectric objects, the common.glsl file already contained some verifications and calculations regarding refraction. In order to decide if it will scatter a reflected or a refracted ray, we had to use probabilistic math. To do so, we implemented a bool function that calculates the refracted ray direction. If it is a refracted ray, we apply the Schlick approximation of Fresnel Equations to calculate the reflection probability. Otherwise, we are in the case of total reflection and this probability is equal to 1. Then, according to this calculated probability, we scatter the reflected or refracted ray.

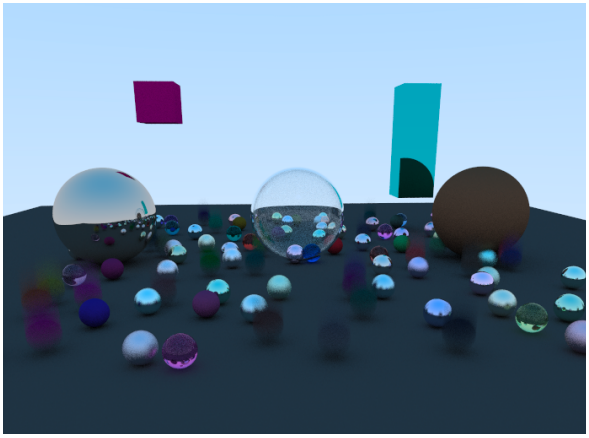

Figure 1 - No light sources

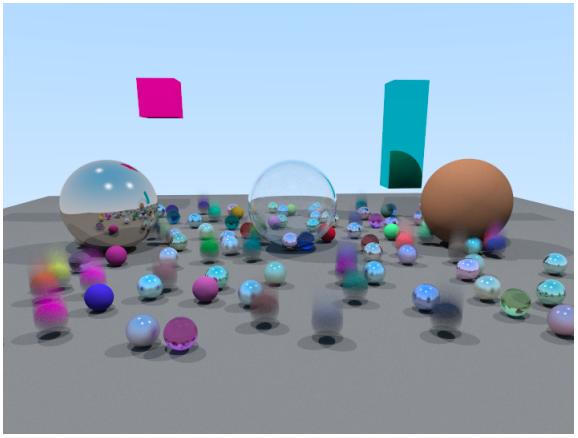

Figure 2 - Multiple light sources

4. Motion blur

Then we implemented Motion Blur in moving spheres. This notion exists to simulate the rapid movement of the objects during a certain period of time.

To do so, we had to implement the function that retrieves the center of these spheres as well as the hit_movingSphere(). The latter is equal to the hit_sphere with a slight change, since moving spheres have two different centers.

Then we added the variable time to the creation of our rays.

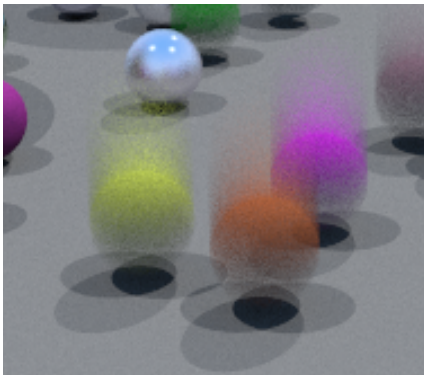

Figure 3 - Motion Blur

5. Depth-of-Field

Since the getRay function was already implemented based on the notion of Depth-of-Field, we only had to change the values of the aperture and distToFocus in order to see its effect. These two variables were defined in the main function from the P3D_RT.glsl file with values 0.0 and 1.0 respectively, which means the Depth-of-Field is initially deactivated. We activate it by changing the aperture to 12.0.

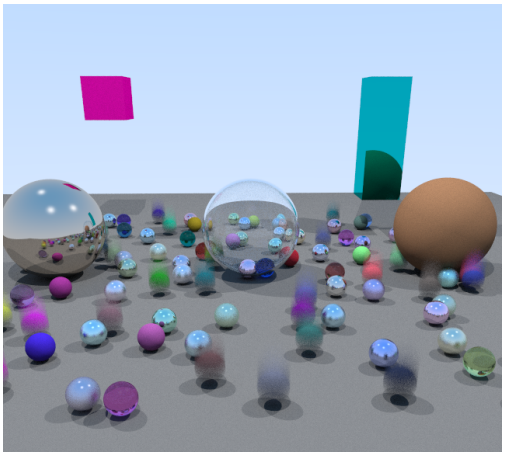

Figure 4 - Scene without Depth-of-Field

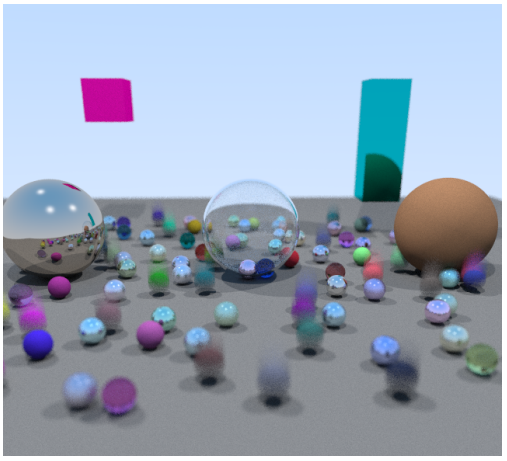

Figure 5 - Scene with Depth-of-Field

6. Extras

For the extras, we implemented the camera rotation which allows the user to move the camera around the scene. We achieved this by defining the mouse variable as a vector of two elements in the P3D_RT.glsl main function, which includes the coordinates x and y of the cursor position, divided by the scene resolution. We used this mouse variable to define the camera position variable.

In addition, we decided to create new geometric objects: boxes. To do so, we implemented the *hit_box()* function using the Kay and Kajiya Algorithm and created two different boxes in our scene: one turquoise and metallic and one pink and diffuse.

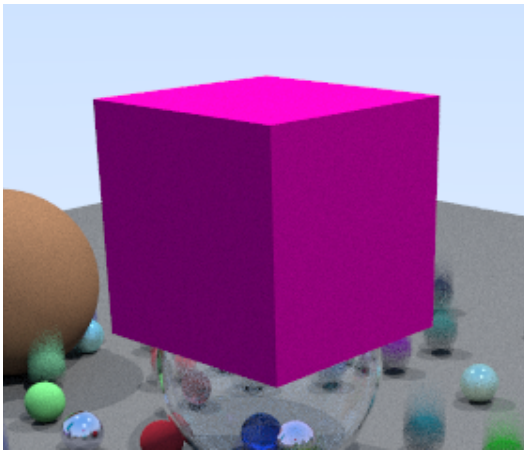

Figure 6 - Pink diffuse box

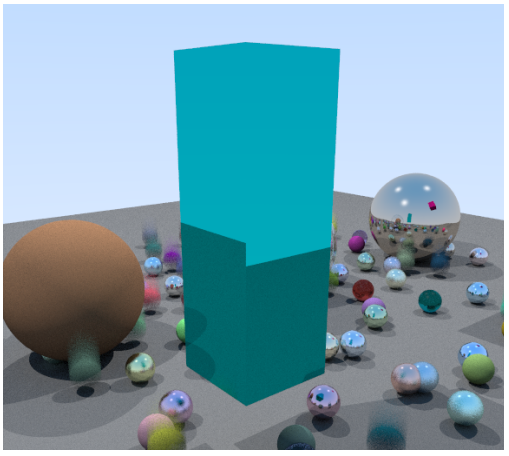

Figure 7 - Turquoise metallic box