Catarina Sousa 93695 Maria Ribeiro 93735 Patrícia Vilão 93604
3D Programming - Assignment 1

Group 1: Assignment 1

Introduction:

In this report, we explain the steps we had to implement in order to develop a Raytracer.

Intersections:

The first thing to develop in order to create a Raytracer is the intersections. Having this in mind, we are going to clarify how we deal with the intersections of different polygons.

Sphere:

To calculate if there is an intersection between a ray and a sphere we start by checking if the ray's origin is outside the sphere. If it is, then we check the sign of the dot product between the ray's direction and the vector between the center of the sphere and the origin of the ray to understand if the sphere is "behind" the ray (dot product with a negative sign).

- If the sphere is behind the origin of the ray we can discard it right away and we avoid unnecessary calculations.
- Otherwise, we calculate the discriminant, which indicates if there's a hit or not.
    - If the discriminant is less or equal to zero there is no hit.
    - Otherwise, there is an intersection between a ray and a sphere. If the ray's origin is outside the sphere, we calculate the smallest root and if it is inside we calculate the positive one, which correspond to the hit points.

Plane:

The first thing to check, in order to avoid unnecessary calculations, is the dot product between the normal of the plane and the ray's direction. If the result is zero, it means that the plane and the ray are parallel, hence there is no intersection.

To determine if there is an intersection we have to substitute the ray's equation into the plane to obtain a new equation: $t = -\frac{(origin - p0) \cdot N}{N \cdot ray.direction}$. By solving it and checking the sign of it, we can conclude if there is an intersection or not:

- If the value is less than zero, the intersection is behind the ray's origin and we can discard it.
- Otherwise, there's a valid intersection.

Triangle:

For the triangle intersection, we implemented the Moller Trumbore Intersection Algorithm because it is an efficient algorithm since it doesn't require the precomputation of the plane equation of the plane containing the triangle. In order to complement the slides, we also viewed this video which explains the algorithm https://www.youtube.com/watch?v=fK1RPmF_zjQ.

Axis Aligned Bounding Box:

To implement the ray intersections with Axis Aligned Bounding Boxes, we used, as suggested in the course slides, the Kay and Kajiya algorithm. This algorithm, also known as the Slabs algorithm, has the objective of for each spacial direction, finding the boundaries of the box,

in order to compute all the areas intersected. We implemented it in our project accordingly to slides 35 and 36 of the “Ray-Geometry intersections” PowerPoint provided on the course page.

Reflection and Refraction:

The second steps were to develop the reflection and refraction:

To have reflection, we use the reflection law to compute the **Mirror Reflection Vector**. Having the reflection ray’s direction, we can trace a ray from the hit point taking into account the new calculated direction.

In order to have refraction, we first need to compute the refraction ray’s direction. This can be accomplished by using the **Snell Law** (which takes into account the refraction indexes of both materials and the angle of incidence). $\frac{\eta i}{\eta t} = \frac{sin\ \theta t}{sin\ \theta i}$. Since there are materials that reflect and refract at the same time, we also need to use the **Fresnel Equations** to compute the amount of reflected vs refracted light. The light is composed of two perpendicular waves (perpendicular and parallel polarised light).

$$R_{perpendicular} = \left|\frac{\eta i\ cos\theta i - \eta t\ cos\theta t}{\eta i\ cos\theta i + \eta t\ cos\theta t}\right|^2 \qquad R_{parallel} = \left|\frac{\eta i\ cos\theta t - \eta t\ cos\theta i}{\eta i\ cos\theta t + \eta t\ cos\theta i}\right|^2$$

Averaging the computed values, we obtain *Kr*, the ratio of reflected light.

We also apply the Schlick’s approximation when *schlick_approximation* is enabled, which is a formula to approximate the contribution of the Fresnel factor in the reflection of light. We start by calculating the Fresnel reflectance at 0º: $R(0) = \frac{\eta i - \eta t^2}{\eta i + \eta t}$ and then we calculate the *Kr*: $Kr = R(0) + (1 - R(0))(1 - cos\theta i)^5$.

In the images below, it is possible to see the results of the sphere, plane, triangle and Axis Aligned Bounding Box intersections and also the reflection and refraction.

Image 1 - balls.low_p3f

Image 2 - balls_medium.p3f

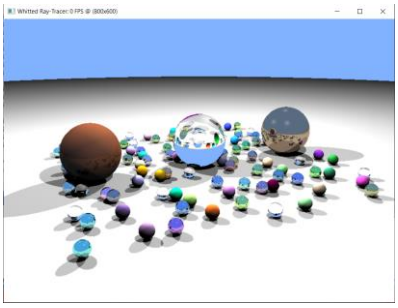

Image 3 - random scene

Image 4 - mount_low.p3f

Image 5 – balls_box.p3f

Depth of field, Soft Shadows and Antialiasing:

Next, we developed the depth of field, soft shadows and anti-aliasing techniques.

Depth of field:

In order to simulate depth of field, it is necessary to distribute rays across the eye.

By default, the lens is infinitely small and real cameras have a finite-aperture lens with focal distance. To compute the depth of field effect, we have to simulate a thin lens. When the *depth_of_field* is enabled we build the lens sample, by creating a unit disk and multiplying it by the camera's aperture (a wide aperture allows a shallow depth of field, while a narrow one allows a deeper depth of field).

Having the lens sample created, we call the function *PrimaryRay* from the file *camera.h*, with a little variation. In this function, we compute the point **p** where the center ray hits the focal plane, and by using the point **p** and the sample point on the lens we compute the direction of the primary ray so that this ray also goes through **p**. To allow antialiasing, we offset the origin of the ray.

Anti-aliasing:

We implemented anti-aliasing using the *jittered* method, a hybrid strategy that randomly perturbs a regular grid.

We can find the implementation in the *renderScene* function in the *main.cpp* file. If the *antialiasing* flag is set to true, we will supersample the pixel with the value of the spp variable. This method works by iterating over each cell and picking a random position within it, assuring that the samples are fairly evenly spread out.

Soft Shadows:

In our project, we implemented soft shadows with and without antialiasing. For both techniques, instead of using the lights as a single light point, the idea to develop the soft shadows is to have instead of one light, have multiple ones around the original light point in order to scatter the shadows, making them look softer and not as harsh as with only a single light point.

The implementation without the antialiasing technique focuses on defining a sort of grid/matrix of lights around the original single light point, dividing the intensity/color of the original light by the number of lights in the grid/matrix. Then we proceed to iterate over each light of the matrix, and for each one, we apply the function *applyLights*, to calculate the color components for each of the lights of the grid and apply them.

Anti-aliasing with Soft Shadows:

When using anti-aliasing with soft shadows, we consider an area of light that contains infinite light points from which we need to choose. To implement a similar technique to the jittered method, we use the current iteration of the antialiasing loop (in the *renderScene* function) to pick a random point within the light area.

In the images below, it is possible to see the results of the depth of field, anti-aliasing and soft shadows.

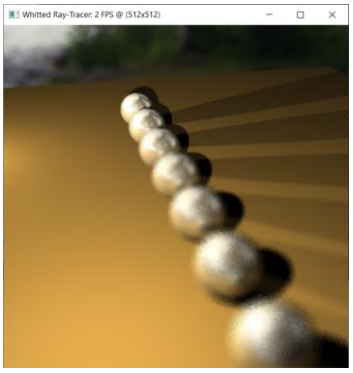

Image 6 - dof.p3f with anti-aliasing

Image 7 - balls_low.p3f with soft shadows and antialiasing

Acceleration structures:

We implemented two acceleration structures, the *uniform grid* and *BVH*.

Uniform grid:

To implement the uniform grid we used the functions provided by the teachers in the file *grid.cpp*.

In the file *main.cpp* we had to make sure that the implementation and the calls to the functions were correct. Having this in mind, if the *acceleration struct* is set to *GRID_ACC* and we want to compute the closest object in the function *rayTracing*, we call the method Traverse (implemented in the *grid.cpp* file) to compute whether there is an intersection, and if so, calculate the closest object and the hit point.

To compute if the object is in the shadow (if there is another object between the original one and the light) it is called the method Traverse (also implemented in the *grid.cpp* file). If it finds an intersection then the object is in the shadow and we don't apply the light to process the color of the object. Otherwise, we compute the color taking into account the light applied.

BVH:

For BVH, the first function to be called is Build, which will do the preparation necessary for the Build_recursive function to be called to construct the tree.

In the build_recursive, the first thing we do is check if the number of objects is fewer than the threshold.

If it is, we initiate the current node as a leaf with primitives from objects[left_index] to objects[right_index] (these indexes correspond to the start and end indexes of the objects vector subset we are looking at).

If it isn't, we need to find the AABB longest axis and sort each object along this direction. Then, we need to find a midpoint that divides the bounding box. To not have any empty sides, we verify if the first object (left index) has a centroid larger than the midpoint or if the last object (right index) has a centroid that is smaller than the midpoint. If any of these conditions are true, we

recalculate the midpoint based on the mean of the object's centroids and use it instead of the node's bounding box's actual middle.

After this, we need to find our split index. To do it, we recurrently divide in half the objects' vector, from the left index until the right index and compute the coordinates of the centroid at the middle index. If these coordinates are smaller than the previously computed mid point, it means that our split index is in the upper half of the array so the start index of the next iteration is the mid index + 1. If the coordinates are bigger, then the split index is in the bottom half, so the end index is set to the middle index. When none of the conditions are true, we have found the split index and can break out of the while cycle.

Then, we divide the scene using the split index into left and right sides. For each side, an AABB bounding volume containing their respective objects is created. Finally, left and right nodes are created in the tree, and their respective bounding boxes are set.

The build_recursive function is called again with the updated indexes to continue to build the tree.

For the *Traverse* function:

First, we need to fetch the root node, which corresponds to the world bounding box. Then we need to see if there is a hit, in case this doesn't happen, we can return false.
If there is a hit, we enter a while cycle.

The first thing to be checked inside the cycle is if the node is an inner node, and if it is we perform an intersection test with both child nodes. If both nodes hit (and they are both closer than the current minimum intersection distance), we put the one furthest away on the stack and set the *currentNode* equal to the closest node. If there was only one hit, we set the current node to the hit node.

If the node is a leaf, we need to check the intersection with each object within that node. If there is an intersection, we store the minimum intersection distance, and the closest object and also update the hit flag to true.

After this, we pop the stack until we find a node with a distance smaller than the current closest intersection distance. If we find a node that fits, we set the current node to the popped one and set the node flag to true.

In the image below it is possible to see the dragon scene rendered with the BVH acceleration structure (the result is the same for the Grid acceleration structure).

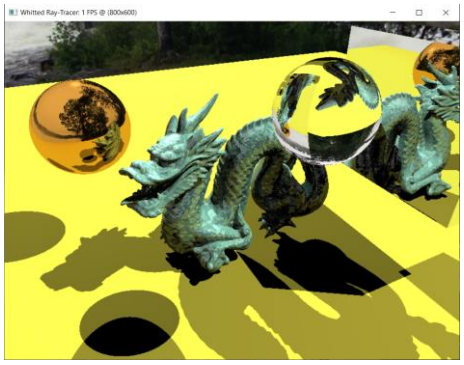

Image 8 - dragon.p3f

Extra work:

For the extra work, we developed fuzzy reflections and motion blur.

Fuzzy Reflections

The fuzzy reflections were implemented based on the course's slides, and the theory behind the implementation is to change randomly the direction of the reflected ray, by choosing a random point in a unit sphere and a roughness factor, this is, by adding the multiplication between this random point of the unit sphere by the roughness factor to the reflected ray, to change his direction and calculate the color based on the new ray calculated. The bigger this factor is, the fuzziest the reflection is. In the project, we can change the variable roughness (between 0 and 1) and see how different values of roughness affect the effect of the fuzzy reflections.

Motion Blur

To implement the motion blur, our approach was also based on the course slides, and to implement it changes were made in the classes Ray and Camera, and a new object called Moving Sphere was created and the implementations for the intersection are the same as for the sphere. To simulate the shutter speed, we added two parameters to the camera object: the initial time and the final time. The interval between the new times added to the camera object was also added to the ray object, in order for it to have the time the ray needs to be shot. As for the moving sphere, to simulate its movement, two centers are needed, an initial and a final one, and a moving center is calculated taking into account the two center positions and the two times, which represent the two positions between each the balls move, and the instants in which the ball is in which position. To use the motion blur, we had to create a new p3f file: motion.p3f in which the moving sphere is used with the tag ms.

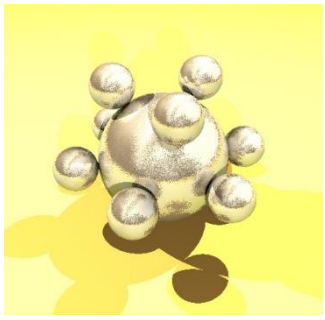

Image 9 – balls_low.p3f with fuzzy reflections

Image 10 - motion.p3f with motion blur