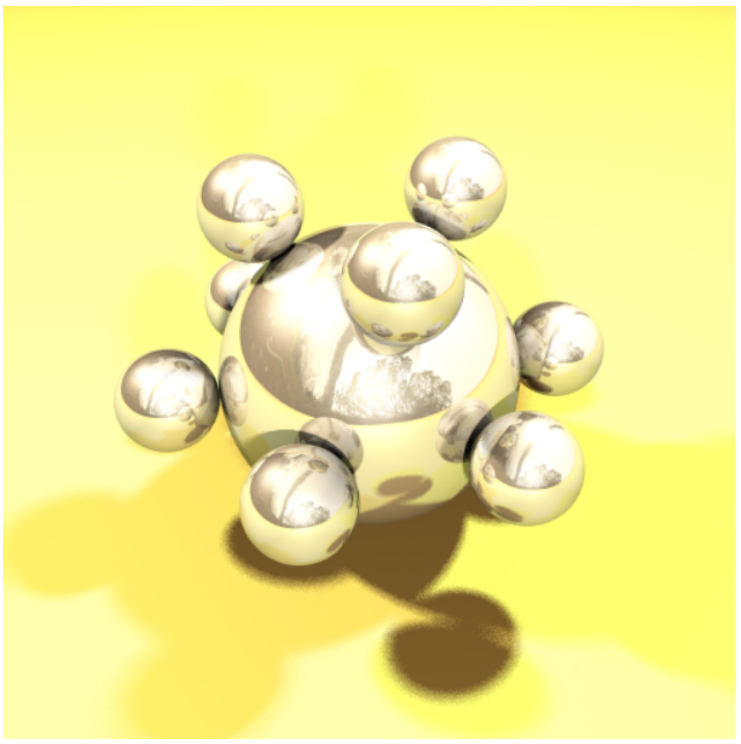

My Ray Tracer

P3D 1st Assignment - April 2021

Daniel Gonçalves [98845]

Diogo Silva [87652]

Henrique Gaspar [98879]

Table of Contents

1 Structure

This project's code was implemented in C++ and as such, is divided into several .cpp and .h files. Notably we have **main.cpp** which contains the main loop of the program (including the renderScene and RayTracing functions), **scene.cpp** which has our scene's object's creation and intersections, **bvh.cpp** which contains our BVH implementation and **camera.h** which defines our camera and the methods to get the primary rays used for the *Whitted Raytracer*.

2 Fresnel Equations & Schlick's Approximation

An essential part of every ray tracer is, of course, being able to simulate **reflections** and **refractions** (indirect illumination). To complement that we also use the Fresnel effect that defines how much the light is reflected versus how much it is refracted.
For the **reflection** we use the law of reflection to create a **mirror reflection vector** that, joined with the hit point found beforehand, allows us to shoot the new reflection ray.
For the **refraction,** in order to compute the respective ray, each material has a refraction index attributed to it. This value when joined with the **ray angle of incidence**, allows us to calculate the new refraction ray with the use of **Snell's Law** as the basis for the calculations.
But if we left it at that, we would be met with a problem when we ray traced a scene. The problem would be that, in the real world, objects can reflect as well as refract light at the same time (Dielectric Materials), and as it is that isn't happening. To compute the ratio of reflected vs refracted light, we use the **Fresnel equations**.
At a very basic level, taking the principle of the conservation of energy into account, we can surmise that the amount of reflected plus the amount of refracted light is equal to the total amount of incident light. Keeping that in mind, we also know that light is composed of parallel

(Rparal) and perpendicular (Rperp) light, these two values are calculated using the fresnel equations, and to compute the ratio (Kr), we only need to do the average of these two values to get the ratio of reflected light. With this value Kr we can surpass the problem we had with the previous implementation, and see reflections and refractions at the same time on the same object.
Now, when talking about Ray Tracing, the smaller the cost of each operation the better, so we also implemented the **Schlick's Approximation** of the Fresnel equations. In this case, we simply use the angle of refraction as 0. This of course changes the calculations that have to be made, but cheapens the overall cost of the process by a significant margin.

3 Intersection Detection

3.1 Triangle

For the Triangle Intersection, we followed the original Tomas Moller approach in what regards to defining each triangle by using the barycentric coordinates. This way of understanding and acquiring an intersection point between a Ray and a Triangle is much more efficient than the traditional geometric way, which uses the plane in which the triangle is set and then applies some bounds. Moller's approach only needs to store the vertices of each triangle, and not the whole plane, and this is the key point for its higher efficiency. Since this technique is theoretically extensive, the full walkthrough will not be displayed in this document but it is available at: https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf. This document not only is pretty complete and explains the whole purpose of using barycentric coordinates, but it was also our main reference for this task.

3.2 Plane

Our approach on the Plane Intersection method is pretty simple: We just take the equation of a plane, the parametric equation of a Ray, and the concept that a vector that represents going from a point P to P0 (the point that represents the "center" of the plane and with which the normal is traced) should be perpendicular to the plane's normal in order to assert that P is actually part of this plane, and merge them in order to find an equation that, once solved for t (the multiplier of the Ray's direction), can ultimately give us the point of interception.
It is important to notice that some cautions were taken, mainly in what regards to a possible infinite number of interception points (once the Ray perfectly aligns with the plane) and to not finding any interception at all (the Ray being parallel to the plane). These exceptions were attended to by dealing with the possible outcomes of the equation previously obtained.

3.3 Sphere

Just like the plane intersection method that we used, the way for us to calculate an intersection between a Ray and a Sphere was also based on implicit geometry (describing an object by mathematical equations). By mixing the ray parametric equation with the one of a sphere, and then solving it in order to t (the multiplier of the Ray's direction), we end up with a quadratic equation that we don't really need to solve. We just need to find this equation's discriminants and use its result to find the number of possible solutions, aka intersection points. There can be up to two solutions and some particular cases (when the ray origin is inside the sphere, for example), so we use the c parameter to understand where are located t0 and t1 (the t value for the intersection points) and how we should understand them.

3.4 Axis Aligned Bounding Box

For the AABB intersection test, we decided to use a straight line based approach. For this technique, we used the mathematical equation of a straight line and mixed it with the equation of a ray. Once again, the theoretical explanation of this technique is quite extensive, so we advise the analysis of this article: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection, which was actually the one that we used in order to fully understand this task and all its intricacies.

4 Ray Distribution

4.1 Antialiasing

Our **Antialiasing** was implemented using **Shirley's Jittering Super Sampling** method. On our renderScene method (within **main.cpp**), if the *ANTIALIASING* option is set to true, instead of just fetching the current pixel, we instead generate X new pixels within an area around the pixel. This X is defined by the square *SPP* (samples per pixel) variable. As such we are supersampling the pixel. In order to improve the spatial distribution of our new samples we generate them by iterating over two *for* loops. From a high level perspective we can think of this as the actual pixel being at the top left of a grid of *SPP x SPP* cells. We then iterate over each cell and pick a random position within it (pixel.x = x + (p + rand_float()) / SPP ; pixel.y = y + (q + rand_float()) / SPP;). This way we ensure that our new pixel samples won't be all clustered in close-by positions, and are instead relatively evenly spread out. The code that implements this is located in **main.cpp** starting at **line 709**.

4.2 Soft Shadows

Soft Shadows were implemented in two manners, depending on whether or not we are using **Antialiasing** and can be activated through the *SOFT_SHADOWS* boolean. The overall principle for both implementations is the same: Instead of considering the light's in our scene as a single point, we instead represent them as "area's of light', within which are several light sources. Both implementations lie within the RayTracing function. When **not using Antialiasing**, we simply take our current light and define that it as the center of a "grid of lights" of size *NO_LIGHTS x NO_LIGHTS* (note this value should be the same as SPP), where each light's colour is defined as the original's colour divided by the number of lights. We define the variable *size* as being the total size of this grid, and as such, we can compute the distance between each light simply by doing *dist* = *size/NO_LIGHTS*. With this, we can simply iterate over each light in the "grid" using a couple of *for* loops, and incrementing the position using our *dist*. We then call our *processLight* function (which handles checking for shadows/incrementing the color of the pixel according to the light source). When **using Antialiasing**, however, instead of a "grid of lights" we consider an area of light (of which the current light is the top left of) containing infinite light points and simply have to pick one. Originally, all we did was pick a random point that fell within the bounds of this area of light (whose size is again defined by the variable *size*), but, in order to improve the outcome of our algorithm, we instead opted to apply a similar technique to the **Jittering** used in our Antialiasing. We store the pixel's current offsets (i.e the current iteration of the Antialiasing's for loops) in a global variable, and utilize it to pick a random point in our area of light using a similar equation to the one we used to pick our pixel's random position (light->position.x + size * ((off_x + rand_float()) / SPP)). This way we can ensure the uniform spatial distribution of our random light points since what we did was basically divide our area of light into a grid, and then pick a random position within each cell. This code can be found in **main.cpp** starting at **line 282**.

4.3 **Depth Of Field**

By default, the lens of our camera is simulated as if it were a singular point, i.e, it's infinitely small (Pinhole camera model). To simulate DoF, we can use ray distribution to simulate a **thin lens** which collects light from a cone of directions (which has its apex at the distance where everything is in focus - *Focal Plane)*. When the variable *DEPTH_OF_FIELD* is enabled in **line 692** of our **main.cpp** we create this *thin lens* by first creating a unitary disk and multiplying it by the camera's aperture (the aperture defines the diameter of our lens, i.e the area within light is allowed to come in). We then call a variation of our camera's **PrimaryRay** function, passing it this lens. Within **camera.h** (**line 107**) we start by computing the point where the center ray hits the focal plane - **p**. With this computed point and our lens' sample we can then compute the direction of the primary ray so that it also passes through **p**. Additionally we also offset the ray's origin point, rather than always shooting them from the center of the camera's eye to allow for the Antialiasing of the points in our focal plane (i.e those that should be in focus).

4.4 **Fuzzy Reflections**

Fuzzy reflections were implemented by taking the reflected direction and randomizing it based on a unit sphere and a new endpoint for the ray. They can be enabled by setting the *FUZZY_REFLECTIONS* variable to true and tweaking the *ROUGHNESS* variable (between 0 and 1). In this case, the sphere size and fuzziness are directly proportional.Crossing over to our code implementation, we take the **ray direction** that was computed to get a given object's reflections. To that vector we had the normalization of the product of a **sample unit sphere**, and the **roughness** value, that is basically the radius of the sphere, so the bigger this value is, the fuzziest the reflection will be. Then when we get the new ray direction based on this, we just create the **new reflected ray** using it and the **precise hit point** computed before.

4.5 **Motion Blur**

For **Motion Blur**, we started by creating 2 new variables in our **camera.h** - time0 and time1 - which represent the camera's shutter speed. Additionally, we also added a new variable to our **ray.h** - time - which represents the instance of time at which the ray was shot (and falls between t0 and t1). So now, if the *MOTION_BLUR* variable in **main.cpp** is set to true, on our RenderScene function we simply set our camera's shutter speed to what's defined in the *t0* and *t1* variables. We modified our camera's GetPrimaryRay functions to also receive the *MOTION_BLUR* bool. If set to true, when computing the primary ray, the camera also sets it's *time* variable by picking a random instance between time0 and time1 using the formula *time0 + rand_double() * (time1 - time0)*. To see the effects of Motion Blur we created a new object in p3f - **Moving Sphere** - defined by the tag *sm* and a new scene - *ball_motion.p3f* - containing a single one of these objects. The way it differs from a regular sphere is the fact that rather than having a single center, it has two - center0 and center1 - representing the two positions between which the ball moves, and also has two new variables - t0 and t1 - with t0 representing the instant of time in which the ball is at center0, and t1 the instant of time in which the ball is at center1. When computing the intersection with this ball, we must first set it's "current center" by taking into account the *time* at which the ray was fired, center0, center1 and the corresponding t0 and t1 - *center0 + (center1 - center0) * ((time - time0) / (time1 - time0))*. We now have the ball's center position and can compute the intersection as if it were a normal sphere. The code for the Moving Sphere intersection can be found in **scene.cpp** at **line 866**.

5 Bounding Volume Hierarchy

5.1 Build

If *USE_ACCEL_STRUCT* is set to **2** in **main.cpp**, then at **line 662** we call the **bvh** build method (passing it a vector with our scene's objects). Within this method, we start by setting the root of our binary tree of nodes as being the scene's bounding box and start our **build_recursive** method which will carry on constructing this tree. Within build recursive we have the variables *left_index* and *right_index* which correspond to the start and end index of the subset of our object's vector that we're currently looking at. First thing we do in this method is to check whether *right_index - left_index* (i.e the current number of objects in our subset of the object's vector) is smaller than a given threshold (2). If so, we create a leaf node with these objects. If that's not the case then we must continue splitting. To do this we first find the largest axis of our current node's bounding box. Then we sort our object's vector from left index to our right index, in accordance to their centroid positions' value in the aforementioned largest axis. Then we find the middle coordinates of our current node's bounding box (according to the largest axis). Before continuing, we check whether our first object (the one at the left index) has a centroid that is larger than the computed middle coordinates, or if our last object (the one at the right index) has a centroid that is smaller than the computed middle coordinates. If any of that is true, then it means we either have an empty left or right half (respectively) and will use a mean of the object's centroid as mid coordinates instead of the node's bounding box's actual middle. For safety we do the same check again and if true, simply set our split_index (i.e the index in the object's array that splits the objects within our current node's bounds) as being our left_index plus our threshold. Else, we perform a *binary search* in order to find our split_index. The way this works is we're constantly splitting in half our object's vector starting at our left_index until our right_index, and getting the coordinates of the centroid at the middle index. If these coordinates are smaller or equal to the previously computed mid coordinates, then our split index is in the upper half of our array, so we set our starting index to be the middle index +1 and carry on to the next iteration. If these coordinates are bigger, than the split_index is in the left half, so we set our end index to be the mid index. If neither of this is true, we found our split_index and can break out. Now that we have the split_index all that's left is to create the left and right children. Their bounding boxes are done by taking all object's to the left of the split index for the left children's bounding box, and all of those to the right for the right children's. We then push back those nodes (being children of our current node) and call **build recursive** again to continue building starting from the left child and then from the right child.

5.2 Traversal

For traversing the **BVH** we start by fetching the root node - which corresponds to the "entire scenes" bounding box. If the ray doesn't intersect it, then we can immediately return false. Else, we enter a *while* cycle in which we check if our current node is a leaf. If that's the case then we must check the intersection of our ray with each of the objects stored in that node (and return true if we simply want to check for the existence of an intersection, or store the minimum intersection distance and closest object otherwise). If our current node isn't a leaf, then we must fetch it's left and right child and check our ray's intersection with their bounding boxes. If both hit (and they're both closer than our current minimum intersection distance), then we set the closest child as our current node and store the other one in a stack and go to the next iteration of the loop. If only one hits then we simply pick that one as current node and continue to the next iteration. If no new node was chosen from the children of the current node, or if our current node is a leaf, we try to fetch a new node from the stack (only picking it if it's distance is smaller than our current smallest intersection distance). If we managed to get a new node, we carry on to the next iteration, else we break out of the loop. If all we want is to check for an intersection, we simply return false after leaving the loop (no intersection was found otherwise we would've already left the function and returned true during the cycle). If we want to find the intersection point, we check whether there was a hit found or not during the loop, create the hit point if there was and return true, otherwise return false.

6 Uniform Grid

6.1 Implementation

The file **grid.cpp** contains an implementation of a Uniform Grid which was provided by the teacher. Nevertheless, before this happened, we implemented our own version of the Uniform Grid, which was still left in the file commented under the given implementation. It contains a method to compute the resolution of our grid and initialization of the grid's cells and their respective object arrays, a method to initialize the traversal parameters (i.e, the maximum and minimum x, y and z coordinates and how much we move in each of these coordinates at any point - step size) which utilizes the 3DDD - 3D Digital Differential Analyzer - Algorithm (which makes use of the similarity between triangles to compute these variables), and two traversal functions (one used for the shadow feelers, which seeks to find only whether there's an intersection or not, and one which seeks the closest intersection). Additionally, there are also some support functions for getting the maximum and minimum bounding boxes.

6.2 Mailboxing

In order to further improve the Uniform Grid's performance (which was given by the teacher and, due to space restraints on the report won't be further mentioned here) we implemented **MailBoxing** which allows us to avoid having to check a ray's intersection with a given object more than once (which may happen in Uniform Grid since objects may belong to several cells). The way we do this is pretty simple. We added two new variables to **ray.h** - *id*, which represents the ray's id and *next_id* which is static (i.e the same for every instance of the ray class) and initialized at 0. When we create a ray, we set it's *id* to be equal to *next_id* and increment *next_id* so that the next ray will have a different id. Additionally, each object (i.e sphere, plane, and so on) has a new internal variable - *mailbox_id*, initialized at -1 - When the boolean *USE_MAILBOX* in **scene.h** is set to true, on each of the object's intersections we check if it's mailbox_id is larger than the ray we're using for the intersection's id. If that's the case, this means that this object's intersection has already been tested for this ray and hence we can simply return false. Else, we set the object's mailbox_id to be equal to the ray's id, and carry on with testing the intersection.