3D PROGRAMMING

Distribution Ray Tracing-
Assignment 1 and Exercises 3 and 4 - Report

Group 1

Inês Lacerda [86436]

Jin Xin [86438]

Lucas Soares [86463]

Overview

For assignment 1, we implemented the techniques of Distribution Ray Tracing. We also extended our ray tracing application by using a grid-based acceleration structure.

T. Whitted Ray-Tracer

Ray tracing is a technique that creates images by shooting rays. For each pixel in the viewport, a **primary ray** is shot. The origin of this ray is the position of the camera (i.e., the eye) and the direction can be computed using the following equation:

$d = normalize\left(w\left(\frac{x}{ResX} - 0.5\right)\widehat{X_e} + h\left(\frac{y}{ResY} - 0.5\right)\widehat{Y_e} - d_f\widehat{z_e}\right)$, where $\widehat{X_e}$, $\widehat{Y_e}$ and $\widehat{Z_e}$ are the Camera Frame axes.

The idea behind ray tracing is to compute the intersections of the rays with various objects (e.g., Spheres, Planes and Triangles).

Intersections

For each object in the scene, we are interested in two components: the normal and the intersection point. The **normal** is important as it defines the orientation of the object. The **Triangle's** and **Plane's normal** is calculated the same way: the cross product between the two edges. The **normal of a Sphere** can be computed using different methods, but in the project, we computed it as the point position minus the sphere center.

The **Ray-Plane Intersection** is computed using the ray parametric equation:

$$t = -\frac{(0 - a) \cdot n}{n \cdot d}$$

The **Ray-Triangle Intersection** is computed using the *Tomas Möller-Ben Trumbore* algorithm where the arbitrary point on the plane can be written by using its Barycentric Coordinates (α, β, γ). Only two Barycentric Coordinates are independent as we only need two coordinates to specify a point in the plane and three conditions must be meet:

1. $0 \leq \beta \leq 1$
2. $0 \leq \gamma \leq 1$
3. $0 \leq \beta + \gamma \leq 1$

The **Ray-Sphere Intersection** is computed using the quadratic function and the roots are calculated. If the roots are both positive: we pick the smallest; if both roots are the same: the intersection is tangent to the sphere; if there is one positive root and one negative: the ray originates inside the sphere, thus, we pick the positive root; if both roots are negative: we reject the intersections; if the roots are complex: there is no intersection.

However, there are some **optimizations** such as: if b is less or equal to 0, there is no intersection as the sphere is "behind" the ray and so we can detect intersections before calculating the discriminant which reduces computation time. b is the coefficient that represent the inner product between the ray direction and the vector OC (i.e., difference between the center of the sphere and the direction of the ray). Furthermore, if the discriminant is less or equal to 0, there is also no intersection.

Local Color Component

If the ray hits an object and that intersection point is not a shadow, the ray tracer checks each light source in the scene and determines the local color component of the pixel using the ***Blinn-Phong*** **model**. Only the diffuse and specular components were taken into consideration in the project.

Global Color Component

When indirect illumination is considered, incident radiance can arrive at a surface point of the object from any direction. When a ray hits an object whose material is reflective, we calculate a **reflected ray** in the direction of the mirror reflection and determine the color of the pixel. When a ray hits an object whose material is refracted, we calculate a **refracted ray** using the Snell's law and determine the color of the pixel.

Stochastic Sampling Techniques

Anti-aliasing

When looking at a lower resolution image, it is often to be noticed some distortion artifacts, commonly known as aliasing. In order to minimize these distortions, a technique called anti-aliasing is applied. The main essence of an anti-aliasing technique is that, the final color of each pixel does not depend on one single ray that hits it. For our project, we used the **Jittering** technique. The process is quite simple, when going through every pixel in the image, instead of shooting one single ray at that pixel and assigning the color

immediately to it, it separates the pixel into multiple stratified samples(SPP_N * SPP_N samples) and shoots a ray for each. However, a pixel cannot assume multiple colors, nor the sum of all colors from the sample, so the arithmetic average of the sample colors is calculated and assigned to the pixel.

$$
\begin{aligned}
&\textbf{for } each\ pixel\ (i,j) do\\
&\quad c = 0\\
&\quad \textbf{for } p = 0\ to\ n-1\ do\\
&\qquad \textbf{for } q = 0\ to\ n-1\ do\\
&\qquad\quad c = c + raycolor\left(i + \frac{p+\varepsilon}{n}, j + \frac{q+\varepsilon}{n}\right)\\
&\quad c_{ij} = c/n^2
\end{aligned}
$$

The pseudo code on the right explains step by step the Jittering process, with n * n being the number of stratified samples mentioned above.

Soft shadows

Before implementing this, the output images contained sharp edged shadows called hard shadows. To make these hard shadows smoother we must create a few penumbra zones so that the resulting images are more realistic. In our project, there are 2 different soft shadow implementations. Soft shadows can be present with and without anti-aliasing.

When there is anti-aliasing, for each light, the **random method** is applied. This method involves computing a random point in a sphere of diameter = 1 and the light position as center so that the point is not located too far from the light position. Then, the color of this new point light is calculated using the *Blinn-Phong* model. Without anti-aliasing, there are no random points, instead, for each light, a set of **N light source points are selected from an area light** (e.g., a sphere). Then, the colors of these new point lights are calculated using the *Blinn-Phong* model. However, since the number of light sources is 8 times the original set, then the intensity also increases. To even out the intensity, the final value for the color of each point is divided by the total number of light sources.

Depth of field

The depth of field of a camera is the range of distances over which objects appear to be in and out of focus. In a ray tracing algorithm, it is necessary for us to create the optical properties. For that, we require some crucial variables: the lens sample, the focal distance and the aperture of the lens. An important note is that the depth of field only exists when there is anti-aliasing and a new *PrimaryRay* function must be used. This new function receives the *lens_sample* as an argument and returns a ray with the direction and the origin now starting from the *lens_sample*. The *lens_sample* is the resulting vector from product of the *sample_unit_disk* and the aperture of the camera.

To simulate the depth of field, three tasks must be performed:

1. Compute *p* which is the point that hits the focal plane.
2. Compute the direction of the primary ray that goes through *p* using the *lens_sample* and the pixel sample.
3. In the *render scene* function, separate each pixel into multiple stratified samples (SPP_N * SPP_N samples) and shoot a primary ray for each.

Acceleration structures

In ray tracing, each ray must intersect all the objects in the scene which makes it an expensive process. Using an acceleration structure such as a regular grid saves computation time as it significantly reduces the number of ray intersections.

Grid

A regular grid has a uniform box shape. The box is divided into several constant size cells and each cell contains a group of objects that are in it or partly in it.

Construction

The construction of the grid involves many steps: adding the objects, computing the bounding box of the grid, and setting up the cells.

The grid's bounding box is computed based on the objects' bounding boxes which prevents the grid to be constructed in places where there are no objects. We compute the **maximum boundary** of the grid based on the object with the highest maximum bounding box and we compute the **minimum boundary** based on the object with the lowest minimum bounding box. These boundaries are slightly enlarged by subtracting/adding *EPSILON*. The grid's bounding box is, essentially, slightly larger than the union of every object's bounding box and the computed maximum and minimum boundaries represent the **dimension** of the grid.

The next step is to set up the cells which should be cubical for intersection efficiency. The **number of cells** is represented by $\boldsymbol{n_x}$in the x_w direction, $\boldsymbol{n_y}$ in the y_w direction and $\boldsymbol{n_z}$ in the z_w direction, where w represents the dimensions of the grid in x, y and z directions. n_x, n_y and n_z are computed by multiplying a multiplying factor by the

dimension of the grid and by S, where $S = \left(\frac{N}{w}\right)^{1/3}$ and N represents the number of objects in the grid.

The multiplying factor allows the adjustment of the number of cells and the computation of n_x, n_y and n_z has a +1 in it to guarantee that the number of cells is never zero. The total number of cells is, therefore, $n_x * n_y * n_z$. If $n_x = n_y = n_z = 1$, there is no speed-up as we do not essentially have a grid. The number of cells is important, because if there are too few cells, the ray intersects too many objects, whereas if there are too many cells, the ray wastes time testing for intersections in empty cells. The grid seems to be most efficient when the number of cells is 8-10 times greater than the number of objects. In our project, we defined 2 as our multiplying factor since it corresponds to 8 as factor.

The **indexing** of the cells is done by finding which cell an object must be placed in. This is done by computing the cells that contain the maximum and minimum corners of the objects' bounding boxes. For each direction, we must compute the ratio of the distances between the bounding boxes of the grid and the object, multiply it by the corresponding number of cells direction (e.g., n_x) and divide it by the dimension of the grid. This way we obtain i_x, i_y and $i_z$which are used to calculate the index of each cell by doing:

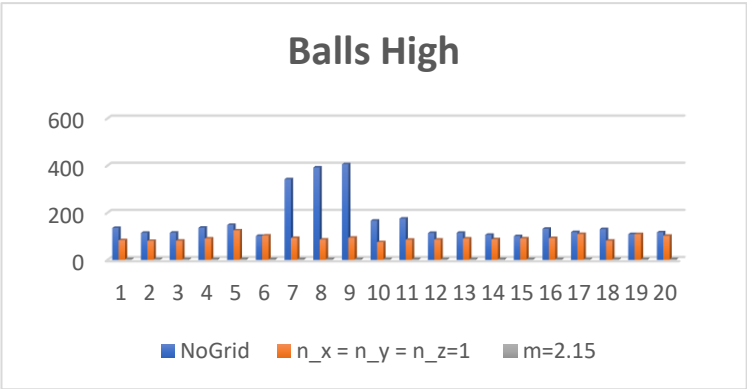
$$index = i_x + n_x * i_y + n_x * n_y * i_z$$

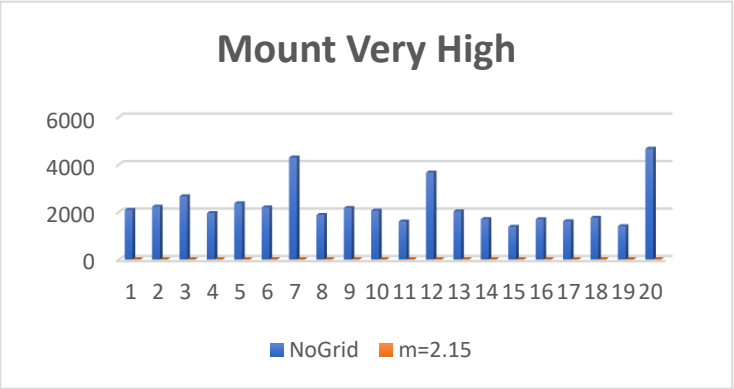
However, there is a problem when indexing the cells, because the bounding box of an object can be located on the positive face of the grid's bounding box which will result in an out of range error. To solve this, the clamp function is used in the computations of i_x, i_y and i_z where it joins the minimum and maximum values by using linear interpolation.

Traversal

When a ray is shot, it is tested for an intersection with the bounding box of the grid. This saves computation time since it is useless to test the intersection of the ray with an object if the ray does not intersect the grid. If the ray does not start inside the grid, we compute its initial hit point at with the ray crosses a cell of the grid's bounding box. The testing of the intersection with objects is based on that cell's index. Then, we go across every cell the ray passes through, until the ray intersects with the closest object or we reach the end of the grid. This can save a lot of time, particularly when there are a large number of objects in the grid, as the ray will only intersect a small fraction of objects.

Execution Times





We decided to execute our project 20 times for each parameter to get more accurate results and conclusions. We executed with grid but $n_x = n_y = n_z = 1$ to prove the statement made above where there is no speed-up. We also decided to execute with different values for the multiplying factor to compare the render speeds.

For the *balls high* scene, the execution time without grid was on average 163.48 seconds, with grid but $n_x = n_y = n_z = 1$ was on average 92.09 seconds, with grid but multiplying factor equal to 2 was on average 3.34 seconds and with grid but multiplying factor equal to 2.15 was on average 3.05 seconds.

For the *mount very high* scene, the execution time without grid was on average 2271.97 seconds, and with grid but multiplying factor equal to 2.15 was on average 9.061 seconds.

We concluded, that the grid significantly reduced our computation time by approximately 98% and we also proved that when having $n_x = n_y = n_z = 1$ there is no significant speed-up.

Extras

We implemented the Box-Ray intersection using the *Kay and Kajiya* algorithm which can be tested running the *balls box* scene. For a ray that is both refracted and reflected we also computed the mirror reflection and refraction attenuations using the *Fresnel* Equations. We also created three additional scenes called *new scene1*, *new scene2* and *new scene3*. *New scene 1* is good for testing reflections and refractions, *new scene2* is good for testing refractions and soft shadows as it has intense sharp shadows and *new scene3* is good for testing the depth of field.