Assignment 1 - 3D Programming

Rita Mota (ist1103509), Beatriz Neto (ist1103149), Catarina Costa (ist1112377)

Group 2

1 T. Whitted Ray-Tracer

1.1 Ray-Geometry Intersections

1.1.1 Sphere Intersections

For the sphere intersection, we will calculate the coefficients, a 1, b 2, and c 3, of the intersection of the ray with the sphere, and using the discriminant 4, we check if the ray intersects the sphere or not. If there is an intersection we calculate the two intersection distances and choose the correct one, meaning the one in front of the ray. If none fit the criteria, the default HitRecord is returned. If a valid intersection is found, the function sets isHit to true and computes the surface normal at the intersection point by normalizing the vector from the sphere center to the hit location on the ray.

$$a = \vec{d} \cdot \vec{d} \quad (1) \qquad b = 2(o - c) \cdot \vec{d} \quad (2)$$

$$c = (o - c) \cdot (o - c) - r^2 \quad (3) \qquad \text{discriminant} = b^2 - 4ac \quad (4)$$

1.1.2 Triangles Intersections

For the triangle intersection, the surface normal is computed via the cross product of two of its edges and normalized. We define two triangle edges and compute the vector h, the cross product of the ray direction and one edge. If the dot product of h and the other edge (a) is near zero, the ray is parallel to the triangle plane, and no intersection is possible. Otherwise, barycentric coordinates u and v are calculated to determine whether the intersection lies within the triangle bounds. If both are within valid ranges, the algorithm computes the intersection distance t from the ray origin. If t is positive and above a small epsilon threshold, a valid hit is recorded. If any of the conditions fail, the function returns a default record indicating no intersection.

1.1.3 Axis-Aligned Boxes Intersections

For each slab we calculate the `t_min` and `t_max` of the intersection, and compare that with the values from the other slabs' intersection to get the largest entering `t_value` and smallest exiting `t_value`. The closest intersection is then checked to see if it's ahead of the ray, which means it's a valid value. In that case we calculate the hitpoint and the normal according to the intersected face.

1.2 Local Color Component

To compute the local color contribution of a given object, both the diffuse and specular reflection components must be considered for each source light. The diffuse component is calculated using Equation 5, and the local specular component using Equation 6.

$$C_{dif_\lambda} = C_{light_\lambda}, k_{dif_\lambda}, (\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}) \quad (5)$$

$$C_{s_\lambda} = C_{light_\lambda}, k_{s_\lambda}, (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^n \quad (6)$$

However, since computing the reflection vector $\hat{\mathbf{r}}$ can be computationally intensive, the Blinn-Phong approximation is adopted. This approach replaces $\hat{\mathbf{r}}$ with the halfway vector $\hat{\mathbf{h}}$, defined in Equation 7.

$$\hat{\mathbf{h}} = \frac{\hat{\mathbf{l}} + \hat{\mathbf{v}}}{|\hat{\mathbf{l}} + \hat{\mathbf{v}}|} \quad (7)$$

Using this approximation, the local color component is computed according to the Blinn-Phong reflection model, as expressed in Equation 8.

$$C_\lambda = C_{light_\lambda}, k_{dif_\lambda}, (\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}) + C_{light_\lambda}, k_{s_\lambda}, (\hat{\mathbf{h}} \cdot \hat{\mathbf{n}})^n \tag{8}$$

1.3 Hard Shadows

Shadow rays were defined according to Expression 9.

$$\mathbf{p}(t) = \mathbf{p}i + t\hat{\mathbf{r}}s \tag{9}$$

To avoid self-intersections due to floating-point precision errors (commonly known as "shadow acne"), all shadow rays—and secondary rays in general—are offset by a small amount, $\epsilon = 1 \cdot 10^{-4}$, along the geometric normal.

For each intersection, a shadow ray is cast toward the light source. If the ray intersects another object before reaching the light, the point is considered to be in shadow, and its local illumination is disregarded.

This approach to computing local illumination and hard shadows produced the rendering shown in Figure 1, using the `balls_low.p3f` scene.

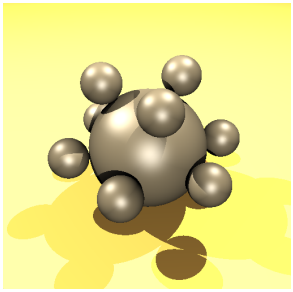


Figure 1: Scene `balls_low.p3f` with Direct Illumination and Hard Shadows

1.4 Indirect Illumination

To account for indirect illumination—such as the transmission of light through transparent media and interactions with reflective surfaces—Snell's Law and Schlick's approximation of the Fresnel equations were employed. These models enable the computation of the proportions of light that are reflected and refracted at the interface between two materials.

The direction of the refracted ray, derived from Snell's Law, is given by Equation 10. Figure 2 illustrates the components involved, including the incident and refracted rays, the surface normal $\hat{\mathbf{n}}$, its negation $-\hat{\mathbf{n}}$, and the angles of incidence θ_i and refraction θ_t.

$$\mathbf{r}t = \sin\theta t, \hat{\mathbf{t}} + \cos\theta_t, (-\hat{\mathbf{n}}) \tag{10}$$

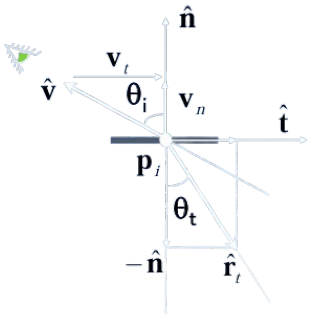


Figure 2: Schematic representation of the refracted ray, showing the incident and transmitted rays, the normal $\hat{\mathbf{n}}$, its negation $-\hat{\mathbf{n}}$, and the angles θ_i and θ_t.

The Fresnel reflectance coefficient was computed using Schlick's approximation, as expressed in Equation 11:

$$K_r = R(\theta_i) = R(0) + (1 - R(0))(1 - \cos\theta_i)^5 \; R(0) \quad = \left(\frac{\eta_i - \eta_t}{\eta_i + \eta_t}\right)^2 \quad \text{(Fresnel reflectance at } 0^\circ\text{)} \tag{11}$$

For rays transmitted through colored transparent materials, Beer's Law was applied to model the attenuation of light due to absorption. The transmitted intensity I is computed using the exponential decay expression shown in Equation 12:

$$I = I_0 \cdot e^{-C_{\text{refracted}} \cdot t} \tag{12}$$

where I_0 is the original intensity, $C_{refracted}$ is the absorption coefficient of the medium, and t is the distance traveled by the refracted ray within the material.

Since the geometric normal always points outward from the object, special care was required when rays originated from inside the object. In such cases, the normal used in reflection and refraction calculations had to be inverted, which was handled by explicitly reversing the geometric normal stored in the child classes of the `Object` class.

The implementation of direct and indirect illumination allowed for the correct rendering of the `teste.p3f` scene, as shown in Figure 3.

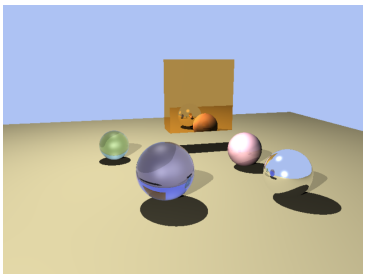


Figure 3: `teste.p3f` scene that shows Refraction, Reflection, Local Color and Hard Shadows

2 Stochastic Sampling Techniques

2.1 Anti-Aliasing with the Jittered Method

For each pixel, it were generated two 2D arrays for pixel and light samples. The arrays contain the random coordinates between 0 and 1 for each ray in a cell of the $n \times n$ grid. Afterwards, the light samples array was then shuffled to avoid a correlation with the pixel samples.

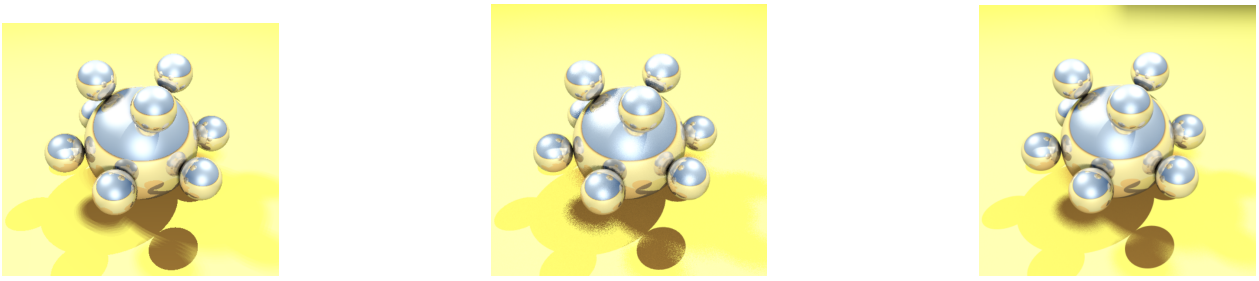
The color of each ray in the pixel is summed and the final value of the color is obtained by dividing the sum by the number of light samples (average).

2.2 Soft Shadows

Soft shadows result from, instead of a punctual light, using area light and distributing shadow rays over the light source area.

In the `ray_tracing` function, for each quad light, the light sample is passed to the `getAreaLightPoint` function, which receives the light sample (coordinates are random numbers between 0 and 1), to the world coordinates, in the quad light perimeter. This ray is then traced.

In Figure 4, it were used two quad lights and one point light.



(a) with spp=0 (b) with spp=4 (c) with spp=64

Figure 4: Scene `balls_low.p3f`

2.2.1 Regular Sampling with no Anti-Aliasing (spp=0)

For $spp = 0$, Figure 4a shows the artifacts that result from the regular sampling method.

When the variable `AA` is false (spp=0), it is implemented the regular sampling method that samples rays on the center of each cell of a $n \times n$ grid. n is given by the square root of the number of light samples, the last value in the light quad description in the $.p3f$ scene file (`gridRes`).

2.2.2 Jittered Method with Anti-Aliasing

With the jittered method, $spp > 0$, the rays passed to the `ray_tracing` , and, consequently, to `getAreaLightPoint`, are jittered samples, obtained as explained in 2.1.

The effect is visible in 4b and 4c. These have softer shadows without artifacts. The higher the number of samples per pixel, the more blended is the shadow (more realistic effect), but also the more computationally costly and time-consuming.

2.3 Depth of Field

To implement Depth of Field, rays are no longer cast from a single camera point but from random positions on a simulated lens. A random lens sample offsets the ray's origin, while the pixel sample defines its target on the image plane. Using the focal distance and focal ratio, the algorithm computes a point on the focal plane that the ray should pass through. The ray direction is then calculated from the offset origin to this focal point. This setup causes rays to converge sharply at the focal plane and blur for objects outside it, producing a realistic depth of field effect when multiple rays are averaged per pixel.

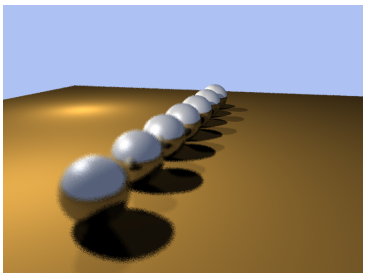


Figure 5: Depth of field

2.4 Fuzzy Reflections

Fuzzy reflections simulate the behavior of rough surfaces by scattering reflection rays. The degree of scattering depends on the roughness of the material. In our implementation, we use a fixed roughness value of 0.3. We start with the perfect reflection direction R to compute the fuzzy reflection. We then offset this direction by adding a random vector from inside a unit sphere, scaled by the roughness parameter.

$$S = p + R + \text{roughness\_param} \cdot \text{rand\_in\_unit\_sphere}()$$

$$\text{ray.direction} = S - p = \text{normalize}(R + \text{roughness\_param} \cdot \text{rand\_in\_unit\_sphere}())$$

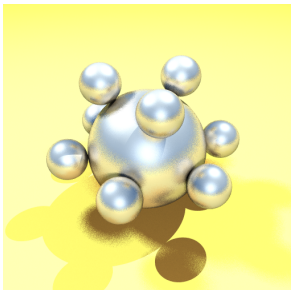


Figure 6: Fuzzy reflections with spp = 16 and roughness = 0.3

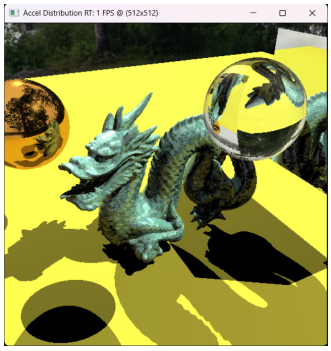
3 Acceleration Structure

To improve the frame per second rendering of complex scenes accelaration structures were implemented, namely Uniform Grid Integration and BOunding Volume Hierarchy.

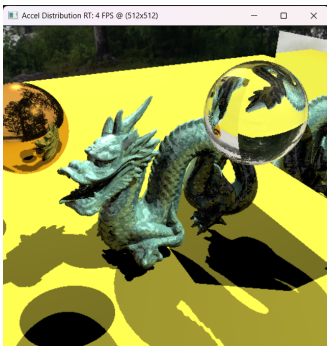
3.1 Uniform Grid Integration

Since the Grid Construction and Traversal methods were already in place, the `Sphere::GetBoundingBox()` method was implemented to provide an AABB enclosing the sphere. This was done by extending the center by the radius in all directions. The traversal used non-normalized shadow rays to function correctly. However, the grid acceleration did not significantly improve performance for `teste.p3f` at `spp = 16`, indicating that a more efficient structure like BVH is needed.

3.2 Bounding Volume Hierarchy (BVH)



(a) with midpoint strategy



(b) with Binned SAH strategy

Figure 7: Scene `dragon_assignment1`

3.2.1 `build_recursive`

In an initial implementation, a simple recursive splitting method was used to construct the Bounding Volume Hierarchy (BVH). The set of intersectable objects was divided into two halves based on a centroid midpoint strategy along the dominant axis of the bounding box. The objects were sorted along that axis, and a `split_index` was chosen such that the left and right subsets could each be enclosed in their own bounding boxes. New BVH nodes were created for each subset, and the process was recursively repeated. Although this approach generated a valid BVH structure, it did not account for the spatial distribution or clustering of primitives, which resulted in inefficient ray traversal. When tested on the `dragon_assignment1` scene, it achieved a performance of only 1 frame per second (fps).

To optimise this method, a Binned Surface Area Heuristic (SAH) method was implemented. Instead of simply splitting at the midpoint, this approach evaluated multiple potential splits across each of the three axes using a fixed number of spatial buckets (in this case, 12). Each object was assigned to a bucket based on the position of its centroid. For every possible split, a cost function was computed as shown in Equation 13:

$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i \tag{13}$$

where c_t is the traversal cost, c_i is the cost of a ray-primitive intersection, $S(B_p)$ is the surface area of the parent bounding box, and $S(B_l)$, $S(B_r)$, n_l, and n_r are the surface areas and primitive counts of the left and right child bounding boxes, respectively. The algorithm selects the axis and partition point that minimizes this cost. This results in a more optimal tree structure that reduces the number of ray-object intersection tests during rendering. Applying this technique to the `dragon_assignment1` scene improved performance to 4 fps, which led to much more responsive movements in the scene.

3.2.2 Traverse

The traversal algorithm has two main functions: one for shadow rays that returns a boolean indicating if any object is hit, and another for determining the closest intersection point along the ray.

Closest-Hit Intersection: This function starts by testing the ray against the root node, which represents the bounding box of the entire scene. If there's no hit, it returns false immediately. Otherwise, the algorithm enters a loop. Next, it checks whether the current node is a leaf. If it is not a leaf node, the ray is tested against both child nodes. If only one child is hit, that child becomes the new current node, and the loop continues. If both children are hit, the algorithm compares their intersection distances, sets the closer one as the current node, and pushes the other onto the stack for later processing. If the current node is a leaf, the algorithm tests the ray against all primitives in that node, looking for any intersection that is closer than the current closest hit. After processing a node, if the stack is not empty, a node is popped and examined. If its intersection distance is closer than the current closest hit, it becomes the new current node. This process repeats until an intersection is found or the stack is empty. If no intersections occur, the function returns false.

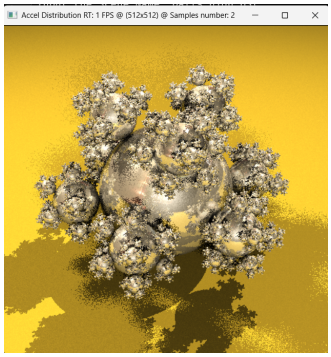
Boolean Intersection for Shadow Ray: The Traversal for the shadow ray works much like the Closest-Hit traversal. Some differences are when the node is not a leaf and both children are intersected by the ray, instead of choosing the closest one for the next current node, we use the left and push the right child node. Then, when analysing the primitives, we also don't need to know the closest one, but only if any is intersected. In that case the algorithm returns true. Otherwise, we pop the stack and continue with that node. If the stack is empty, false is returned since no object was intersected.

3.3 Progressive Mode

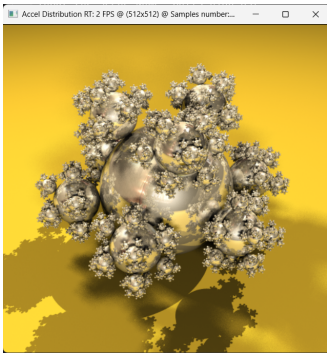
By clicking the `p` key, the progressive mode is enabled. For each iteration of the progressive mode, it is calculated the color of the pixel, using the `lerp` function (linear interpolation). It receives the color of the current iteration (col_n), the color calculated in the previous iteration (col_{n-1}) and a constant ($\frac{1}{n}$) and obtains the pixel final color in the current iteration, given by the Equation 14.

$$col = col_{n-1} + \frac{col_n - col_{n-1}}{N} \tag{14}$$

This mode leads to smooth images, as illustrated in Figure 8b.



(a) 2 samples



(b) 100 samples

Figure 8: Scene `balls_high`