Ray Tracing

Modern photorealistic rendering using an enhanced Ray Tracer

Fábio Cruz
96957
MEIC-T
fabio.rocha.cruz
@tecnico.pt

Luís Fernandes
97018
MEIC-T
luis.martins.fernandes
@tecnico.pt

Florian Lackner
101042
MEIC-A
florian.lackner
@tecnico.pt

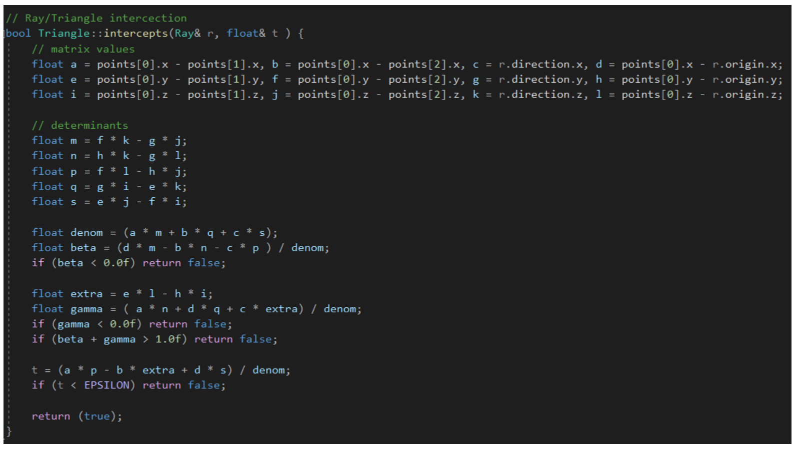
Abstract

In the context of the first assignment of the group project for the 3D Programming course, we implemented modern photorealistic rendering using an enhanced Ray Tracer that supports local color component (Blinn-Phong model illumination) with multiple lights and hard shadows, as well as global illumination by implementing mirror reflections and refractions with Schlick approximation of Fresnel equations for dielectric materials. To test this with some objects we also had to do ray intersections with spheres, triangles, planes and axis-aligned boxes. Besides all this, in order to add more realism to our program we improved it with some stochastic sampling techniques such as anti-aliasing, soft shadows, depth of field and fuzzy reflections. Finally, we implemented two acceleration structures to run our Ray Tracer more efficiently.
In this document, we briefly describe the mentioned techniques and present some details and results of our implementation of Ray Tracing.

1. Whitted's Ray Tracing

Ray Tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects, producing a high degree of realism but being computationally expensive. In practice, we do this by creating a primary ray that originates in the camera eye and is casted in the direction of each individual pixel of the image. If this ray is intersected by an object, we then calculate the color of that pixel using the properties of the object that was intersected, and we can create secondary rays from the intersection point to check if we are in a shadow or even to calculate reflection and refraction effects, if applicable. Otherwise, if the ray does not hit an object, we use the color of the background or the skybox if we have one.

To check this, we first must implement the intersection between the ray and some geometry objects such as spheres, planes, triangles and axis-aligned bounding boxes (AABB), as well as calculate their normals and bounding boxes.



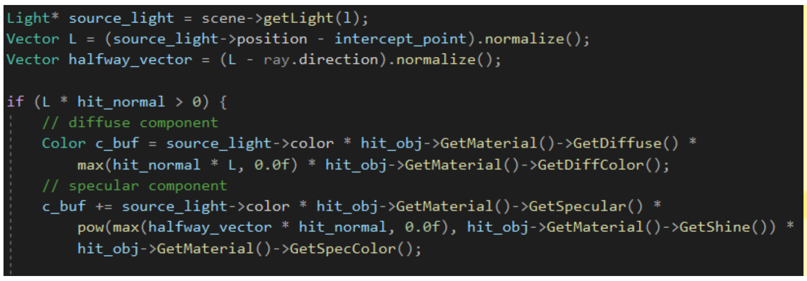
```
// Ray/Triangle intercection
bool Triangle::intercepts(Ray& r, float& t ) {
    // matrix values
    float a = points[0].x - points[1].x, b = points[0].x - points[2].x, c = r.direction.x, d = points[0].x - r.origin.x;
    float e = points[0].y - points[1].y, f = points[0].y - points[2].y, g = r.direction.y, h = points[0].y - r.origin.y;
    float i = points[0].z - points[1].z, j = points[0].z - points[2].z, k = r.direction.z, l = points[0].z - r.origin.z;

    // determinants
    float m = f * k - g * j;
    float n = h * k - g * l;
    float p = f * l - h * j;
    float q = g * i - e * k;
    float s = e * j - f * i;

    float denom = (a * m + b * q + c * s);
    float beta = (d * m - b * n - c * p ) / denom;
    if (beta < 0.0f) return false;

    float extra = e * l - h * i;
    float gamma = ( a * n + d * q + c * extra) / denom;
    if (gamma < 0.0f) return false;
    if (beta + gamma > 1.0f) return false;

    t = (a * p - b * extra + d * s) / denom;
    if (t < EPSILON) return false;

    return (true);
}
```

Figure 1: Ray/Triangle intersection implementation

Regarding planes and spheres, we use the ray equation and the geometry equation to find the parameter 't', which is the distance between the origin of the ray and the intersection point. This parameter is important because it tells us where the object is in the scene from the camera point of view. For example, if the 't' is negative that means the object is behind us, so we ignore that intersection. For the spheres we get 2 solutions for the parameter value because the intersection calculation is done by solving a quadratic formula, which gives the distances of the 2 points where the ray intersects the sphere. These two values again give us information of where we are regarding the objects position (the sphere can be behind us, we can be inside the sphere, or the sphere can be in front of us). The triangle and AABB intersections are slightly more complicated but the basic idea is similar. For triangles we use Barycentric Coordinates to check if the ray intersects them, and for AABB's we use the concept of slabs and check if they overlap to know if the ray hits the AABB.

Now that we know if our primary ray intersects an object when casted through a pixel, we need to calculate that pixel's color, which is made up of 3 contributions according to Whitted's Ray Tracing: if not in shadow, local color due to direct illumination; color from a ray coming from the reflection direction (reflected ray); color from a ray coming from the refraction direction (refracted ray). Regarding the first contribution, to calculate the color that results from direct illumination we will use the Blinn-Phong reflection model, which uses the contribution of the diffuse and specular components to calculate that color. The diffuse component is calculated by multiplying the light color, the object's diffuse component and color and the dot product between the normal on the hit point and the light ray that goes from the hit point to the light source. The specular component, that basically represents the shininess of the light on the object, is calculated by multiplying the specular component and color with the clamped dot product between the surface normal and the halfway vector to get the cosine angle between them that we again raise to a specular shininess exponent.

Creating hard shadows is simple, we just need to cast a ray from the hit point to the light source and check if that ray is intersected by an object, and if it is then there is no contribution from that light source to the pixel color, which gives a nice shadow effect even when using multiple light sources.



```
Light* source_light = scene->getLight(l);
Vector L = (source_light->position - intercept_point).normalize();
Vector halfway_vector = (L - ray.direction).normalize();

if (L * hit_normal > 0) {
    // diffuse component
    Color c_buf = source_light->color * hit_obj->GetMaterial()->GetDiffuse() *
        max(hit_normal * L, 0.0f) * hit_obj->GetMaterial()->GetDiffColor();
    // specular component
    c_buf += source_light->color * hit_obj->GetMaterial()->GetSpecular() *
        pow(max(halfway_vector * hit_normal, 0.0f), hit_obj->GetMaterial()->GetShine()) *
        hit_obj->GetMaterial()->GetSpecColor();
```

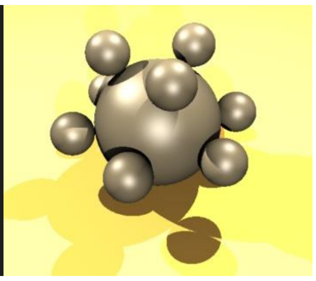


Figure 2: Blinn-Phong model implementation; direct illumination and hard shadows example

Regarding global illumination, reflections and refractions are very common in the real world so we will also implement them, and we will use the Fresnel effect to determine how much of the light is being reflected vs how much is being refracted. Starting with reflections, basically a refracted ray is originated in the hit point and has a symmetrical direction to the incidence ray direction, which means it is very easy to calculate using the law of reflection that uses the angles between the rays and the surface normal to obtain a simple equation to calculate the reflected ray direction. This reflected direction ends up being calculated with the following expression: R = I – 2(N.I)N (where 'N' is the surface normal and 'I' is the incidence ray direction). These reflection calculations are done recursively until we hit a max depth value defined in our program (the same happens with refractions).

Refractions are observed when the rays pass from one transparent medium to another, which alters the direction of the ray. This new direction depends on the angle of the incidence ray and the index of refraction of the mediums. Refractions are described by the Snell's law that relates the angles of incidence and refraction and the two indexes of refraction. Combining the equations that we get from this law and other geometry equations we can mathematically calculate the direction of the new refraction ray direction, which is then used to contribute to the color alongside the reflection component. Some things that need to be considered when implementing these two effects are the direction of the normal (depending on if the refracted ray hits from the outside or inside), the use of a bias to calculate the origin of the new secondary rays that are created, and total internal reflection in the refractions.

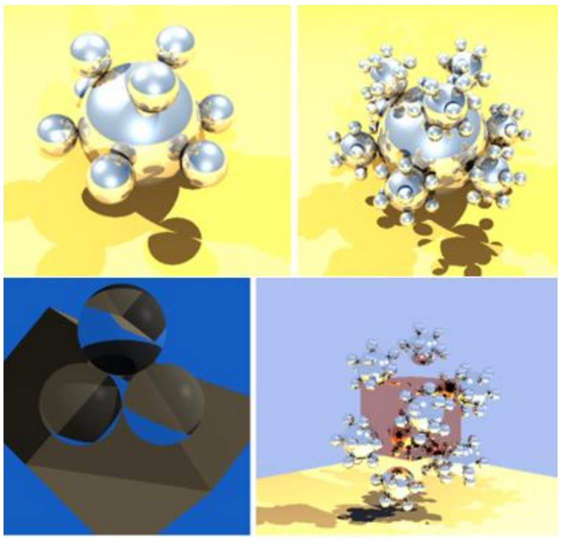
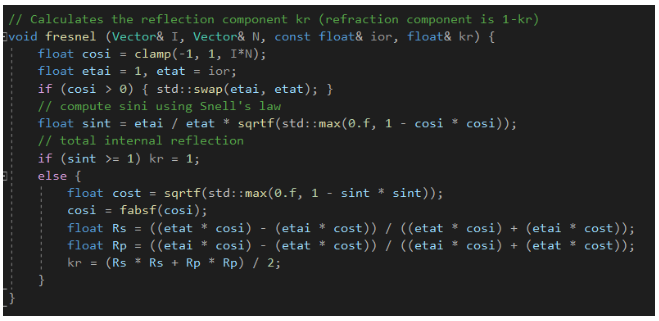


Figure 4: Reflection and refraction examples

The weight that each of them have in the final color of the pixel is calculated using the Schlick approximation of the Fresnel equations.



```
// Calculates the reflection component kr (refraction component is 1-kr)
void fresnel (Vector& I, Vector& N, const float& ior, float& kr) {
    float cosi = clamp(-1, 1, I*N);
    float etai = 1, etat = ior;
    if (cosi > 0) { std::swap(etai, etat); }
    // compute sini using Snell's law
    float sint = etai / etat * sqrtf(std::max(0.f, 1 - cosi * cosi));
    // total internal reflection
    if (sint >= 1) kr = 1;
    else {
        float cost = sqrtf(std::max(0.f, 1 - sint * sint));
        cosi = fabsf(cosi);
        float Rs = ((etat * cosi) - (etai * cost)) / ((etat * cosi) + (etai * cost));
        float Rp = ((etai * cosi) - (etat * cost)) / ((etai * cosi) + (etat * cost));
        kr = (Rs * Rs + Rp * Rp) / 2;
    }
}
```

Figure 5: Fresnel implementation

2. Distribution Ray Tracing

Our implementation of Whitted's Ray Tracing gives us somewhat realistic results, but we still have some problems related with the smoothness of the edges of the objects, the smoothness of shadows (hard shadows are too clean), perfect mirrors and pinhole cameras. To solve these problems, we implemented techniques such as anti-aliasing, soft shadows, depth of field and fuzzy reflections.

Distribution Ray Tracing essentially consists in using many rays to compute average values over pixel areas, area lights, reflected directions, etc. This is the main idea behind anti-aliasing. Instead of casting only 1 ray per pixel to calculate that pixel color, with anti-aliasing we want to cast several rays into one pixel and then average the obtained colors with each of the casted rays. The anti-aliasing algorithm we implemented was Jittering, which is a hybrid strategy that randomly perturbs a regular grid, combining regular grid sampling and stochastic (random) sampling. Regular sampling consists in dividing the pixel in a grid and casting a ray in the center of each grid cell, but this leads to artefacts like moiré patterns. To solve this, we inject some random noise in the solution, meaning the rays are still casted in each grid cell but not exactly in the center of those cells, thus adding a random aspect to our anti-aliasing and improving the smoothness in the edges of the geometry objects.
Regarding soft shadows the idea is somewhat similar to anti-aliasing. Instead of casting only 1 shadow ray to the light source we will cast several shadow rays to a light area around the light's center position. The resulting color percentage, between 0 (no light or full shadow) and 1(full light/no shadow), is the fraction between the number of shadow rays that do not intersect with other objects and the total of rays casted.

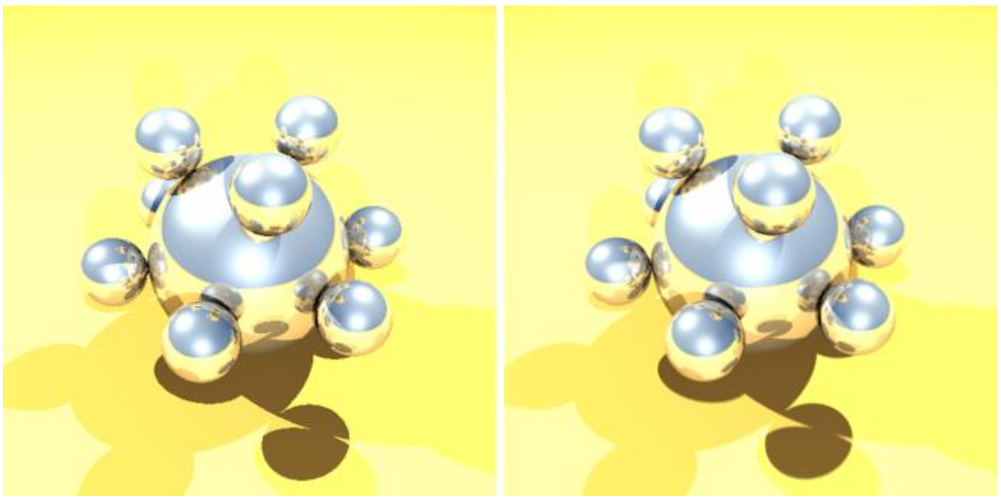


Figure 6: Aliasing and hard shadows vs Anti-aliasing and soft shadows

Depth of field is the distance between the nearest and the farthest objects that are in acceptably sharp focus in an image. The depth of field can be calculated based on focal length, distance to subject, the acceptable circle of confusion size, and aperture, and adds more realism to our program since it represents better the real-world cameras. For the implementation of depth of field, we had to calculate several things. First, we calculated the hit point in the focal plane (focal point) by projecting the primary ray to that plane, by multiplying the pixel sample coordinates with the focal ratio. Then, we calculated lens sample points for each pixel sample by using a unit disk around the camera eye/origin and multiplied those samples by the aperture of the camera. Finally, we created the new primary ray for depth of field scenes with the lens sample as the origin of the ray and subtracting the focal point by the lens sample to get the ray direction. We then ray traced this new ray to the scene and got the depth of field effect.
To finish this section, the fuzzy reflections technique was simple to implement. This effect consists in simulating the roughness of some materials, so we do not always have perfect mirror reflections in our objects. So, what we did was add to the reflected

ray direction a small random sample from a unit sphere multiplied by the roughness value of the material (for instance, 0.3). To facilitate the visualization of this last effect we decided to add skyboxes to our program, so we could easily see the differences in the skybox reflections when enabling fuzzy reflections.

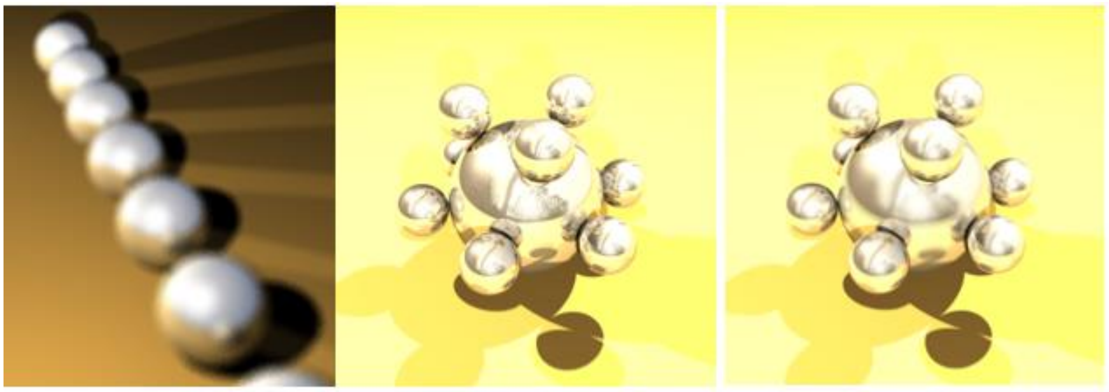


Figure 7: Depth of field; before and after fuzzy reflections

3. Acceleration Structures

One of the major issues with Ray Tracing is its performance, it can be very slow with complex scenes when compared to other rendering methods. To attenuate this, we implemented 2 different acceleration structures: Uniform Grid Acceleration and Bounding Volume Hierarchy Acceleration (BVH).

The first one consists in making a grid of cells from the scene and checking what cells the ray intersects so we can avoid making unnecessary calculations in all the cells that were not intersected, which saves us a lot of computational time. We then only check the closest intersection for the objects that are inside those hit cells. Another optimization in this acceleration structure is testing first the cells that are closest to the ray origin, so we avoid testing cells that are far away and have no chance of having the object that has the nearest intersection with the ray, and the algorithm stops as soon as the ray hits the nearest object.

The second one, BVH, consists in having a tree hierarchy of groups of object’s bounding boxes. To build this tree, we first need to calculate the bounding boxes of all the boxes and its centroids and create a root node with all of the boxes. Then we use a splitting criterion to divide the root node into two different nodes, with a similar number of objects inside each one, and we keep splitting these groups and adding them to the tree hierarchy until we reach a certain threshold. In the end we end up having the object’s bounding boxes in the leaf nodes of the tree, so when we cast our ray trough the scene, we only have to test the objects inside the bounding boxes that were hit by the ray. And since we test first the closest groups to the origin of the ray, just like in the previous algorithm, we also avoid testing a lot of the groups that are hit, saving a lot of computational time.

After running our program with the naïve approach and both of the acceleration structures we observed that for simple scenes the naïve approach has similar running times comparing to the Grid and BVH times, and the reason for that might be because building these structures can be a little expensive and might not be worth it for scenes with few objects. However, for more complex scenes the difference in the times that we observed were significant. For instance, when running the dragon file with 64spp the naïve approach takes approximately 245 thousand seconds, which is ridiculous, but the Grid approach takes only 269 seconds and the BVH a little more than the previous, 639 seconds. Here are some benchmark examples:

	Balls_low	**Balls_high**	**Balls_box**	**Dof**	**Mount_low**	**Mount_high**	**Dragon**
None	0,46	96,07	0,90	0,36	0,39	27,11	245296
Grid	0,60	2,08	0,52	0,48	0,50	0,76	269
BVH	0,56	2,46	0,42	0,39	0,44	0,95	639

4. Post-Mortem

We think that in general this first assignment of implementing Ray Tracing went really well, we managed to successfully implement all the required functionalities and even an extra one, the fuzzy reflections.
We can conclude that using a Ray Tracer to render photorealist image is a good option since the results we got were very satisfying, and that the acceleration structures were very effective when use in complex scenes, when compared to the naïve approach.
The only less positive things were that our refractions had a little bug that we could not fix on time, were the refractions always looked a little zoomed in, and also the implementation of the BVH was probably not the most efficient since its running times are slightly higher than with the Uniform Grid Acceleration.

References

Scratchapixel website:

- https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel
- https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell
- https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

Books:

- Kevin Suffern, “Ray Tracing from the Ground Up”, AK. Peters, 2007;

P3D course slides