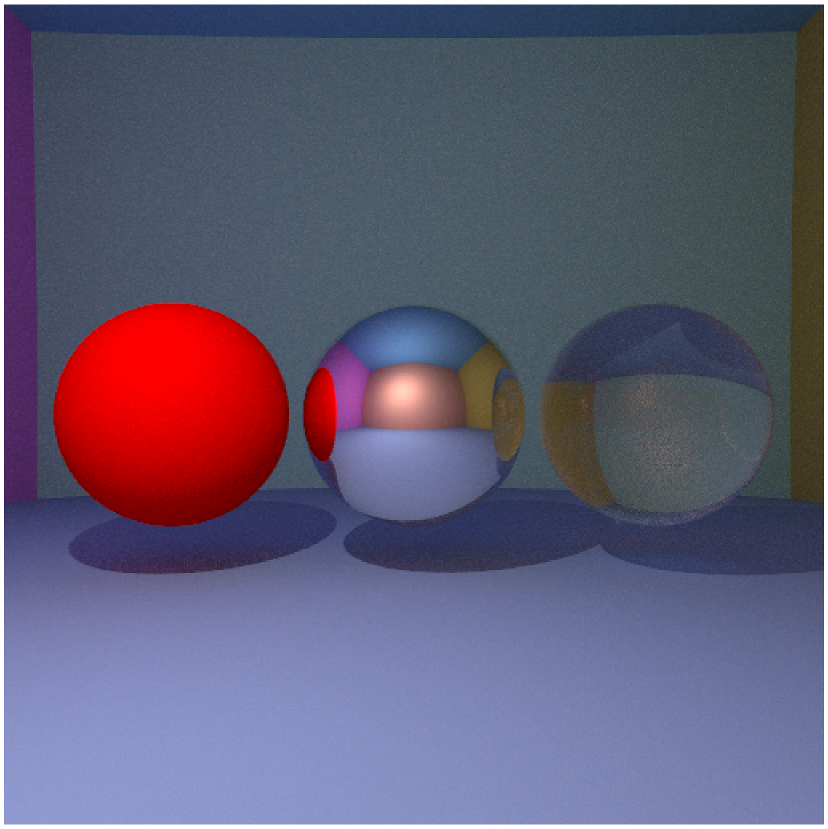
Computer Graphics - Raytracer project

Lucas Massot



I. Basics of Raytracing and Spheres

Disclaimer : Because of the way the computers work through ssh it is hard to have a consistent estimate of the time it takes to render images.
In this section, we will look into the founding principles and mechanisms of our raytracer. The rendering set up for this part consists of a light source and a camera, given as positions in space, and a scene object that contains all the objects to render.

I.1. Class outline

The raytracer relies on several classes of objects and their interactions to function. In our implementation, we use the following :

- **Vector** : an easier way to represent three-dimensional vectors, with a member variable that contains the coordinates. Vectors also provide methods for each vector operation that we use (e.g. addition, dot product).
- **Ray** : an object that contains both the origin of the ray and its direction.
- **Sphere** : the main objects we want to render (and the only ones in the first part). It contains an origin, a radius and an albedo (to get the color), as well as an intersect method.
- **Scene** : it contains all the objects we want to render in the form of a vector, as well as the light source and the camera. It also offers a method for computing the color of a given pixel for rendering
- **Properties** : an object to contain all the properties of the object (e.g. ismirror)

I.2. Ray intersection and color

The principle of ray tracing is to emit ray objects that have as origin our camera and pointing towards each pixel. From there we check whether this ray intersects a sphere through the intersect method and return the color of the closest sphere if intersected. We also check that the ray doesn't intersect another sphere before reaching the one we care about, otherwise we simply return a black color to represent the shadow.

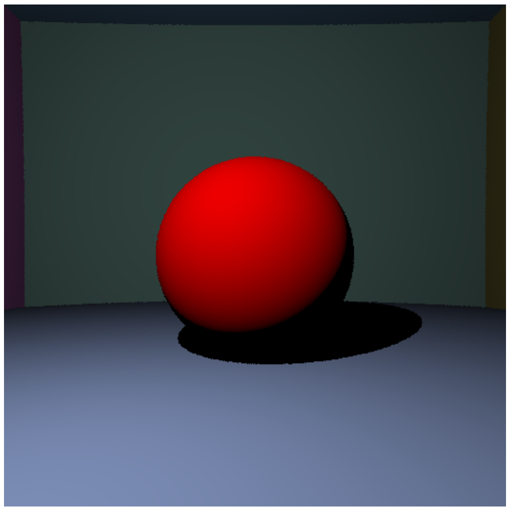


Figure I: rendered with H = 512, W = 512 in <1s

I.3. Mirror surfaces

Another feature of this raytracer is the possibility to make mirror surfaces. Instead of returning the color of the sphere the ray hits first, we emit another ray from the intersection point, and continue this logic until a non-mirror surface is hit or until the "maxdepth" is reached.

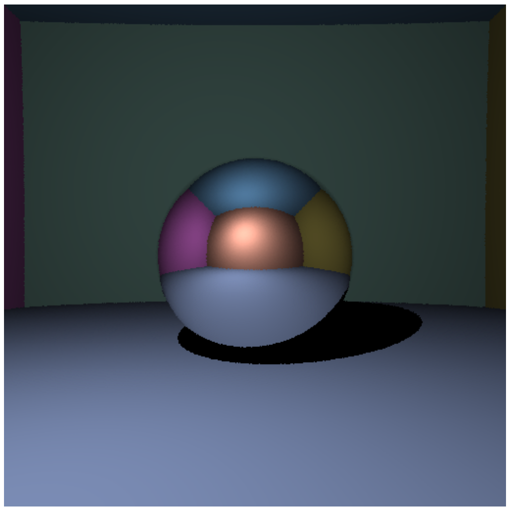


Figure II: rendered with H = 512, W = 512 in <1s

I.4. Transparent surfaces

The last feature related to properties of the object that we implemented is the possibility to render transparent objects. The mechanism relies on Snell's law to emit another ray from the intersection that is deviated. This new ray goes though the object until it intersects the surface of the object again from which it emits a new deviated ray. We can also choose to use the Fresnel law to compute additional light on the surface to get a more realistic rendering of the transparent object. It especially look good when combined with indirect lighting (see fig IV).

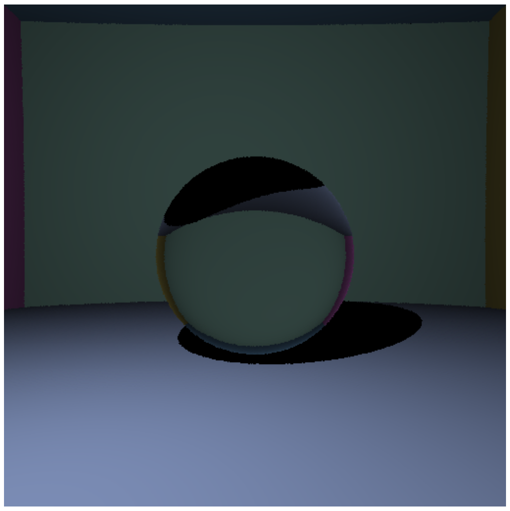


Figure III: rendered with H = 512, W = 512 in <1s

I.5. Indirect lighting

One of the biggest issues with our raytracer at this point is the unrealistic approach to lighting, which creates uncanny shadows. A way to fix this is to introduce indirect lighting. Instead of simply returning the color of the sphere directly, we also emit another ray (randomly) and add the color of this ray to the previous one until we reach depth. This mechanism, combined with the emission of not only one but N rays per pixel (randomly spread around the pixel) and taking the average color, allows for a smoother shadows and more realistic rendering

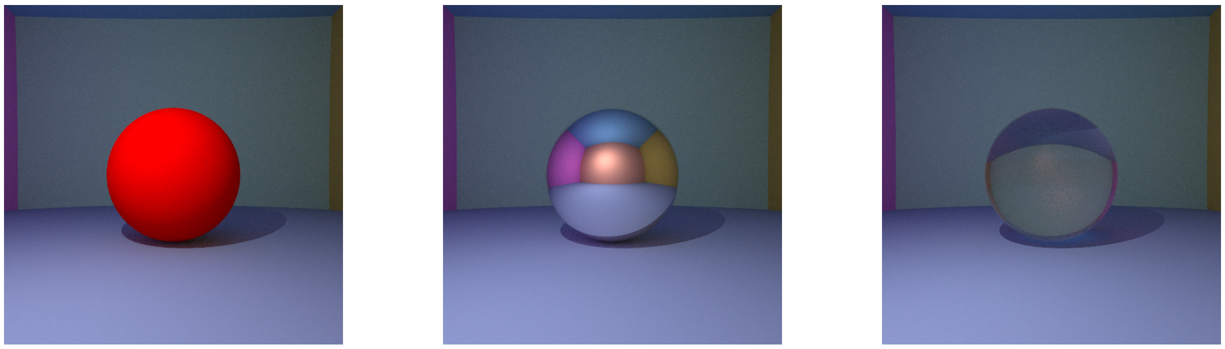


Figure IV: rendered with H = 512, W = 512, depth = 5, N = 64 in 15s

II. Raytracing of Meshes

Now that we have a fully working and realistic raytracer rendering, we want to introduce new objects to render beyond just spheres. We use meshes, which are collections of triangles that put together create the outline of any shape.

II.1. Mesh handling

This introduction of new objects calls for a rework of the class systems :

- **Objects** : an abstract class used in Scene to represent any object to render
- **Mesh** : an object that contains all the information about the mesh, as well as methods like intersect

II.2. Ray intersection with meshes

Keeping the same principle of ray emission, we can introduce an intersection method for meshes that iterates through the triangles of the mesh which gives us information about the intersection if it occurs. This way we can use the same function to get the color of the pixel by replacing the sphere intersection with the mesh intersection

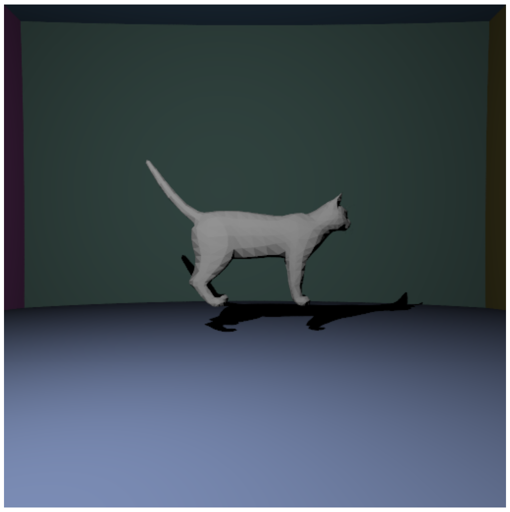


Figure V: rendered with H = 512, W = 512, depth = 5, N = 64 in 15s

We can also use weighted sums of the normal vectors of the vertices of the triangle to get a smoother rendering and avoid the sight of the triangles on the image

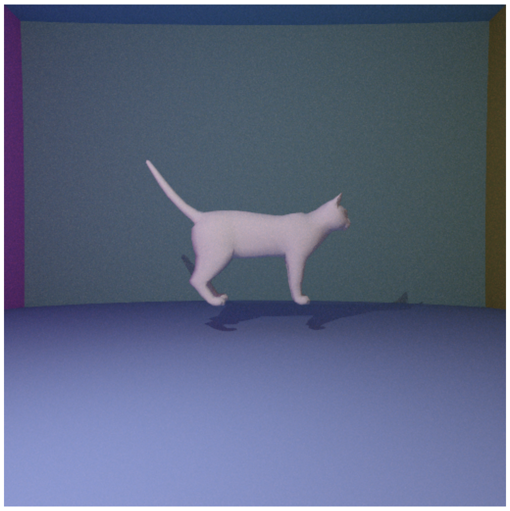


Figure VI: rendered with H = 512, W = 512, depth = 5, N = 64 in 15s

II.3. Optimizing the rendering of meshes

So far, our ray intersection with the meshes is done by iterating over all the triangle of the mesh for each ray that we emit. A more optimized way to handle this is to introduce bounding boxes. If a ray intersects the bounding box, then we iterates through the triangles contained within the box. In particular, starting from the root box (the smallest box containing the whole mesh) we introduce a Bounding Volume Hierarchy, aka BVH, to recursively split in two the boxes. If a box is intersected, we check which of its children box is also intersected and recursively check for this one. This allows us to reduce the amount of triangles we iterate over, hence a better speed performance overall. This mechanism was used in the two previous mesh rendering.