Assignment 1 : Raytracer Project

Garance Perrot

May 2024

The goal of this assignment is to create an image from a scene composed of objects. The final image is a result of the environment's parameters like lighting or camera angle and the variables of the objects such as their shape and texture.

The Scene & its Elements

For the whole assignment, the width W and height H of the screen are fixed to 512 x 512. We start by defining the parameters of the environment contained in a scene (the light source's position and its intensity) in which we can add objects (only spheres for now). Spheres of big size are added to the scene to create the room enclosures, namely a red ceiling, a blue back wall, a green floor and a pink front wall (not seen from the camera's point of view). Their color is given by their argument *albedo*. The object in the middle of the scene, a simple sphere for now, is the body of interest.

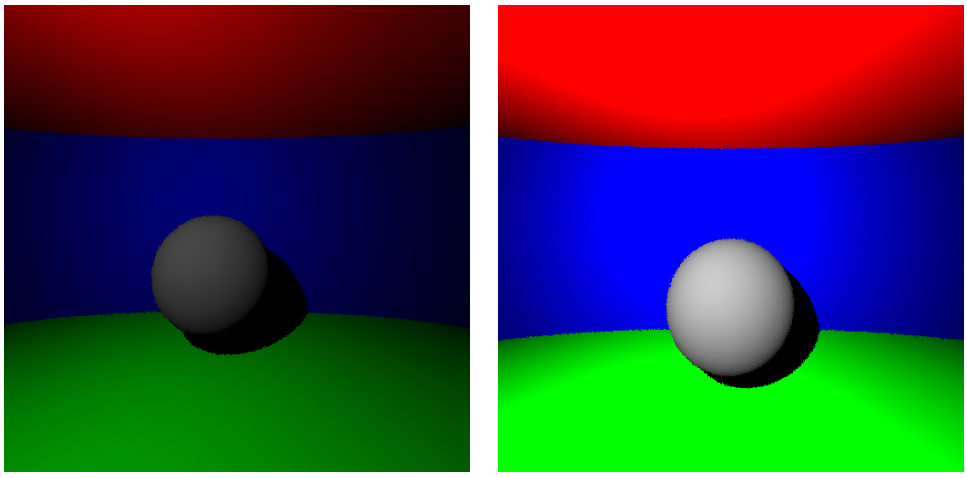

Figure 1: *Runtime : both 144ms. Left : light intensity at 2E10 and camera position at (10,-10,60). Right : light intensity at 2E11 and camera position at (0,0,55).*

Ray-Sphere Intersection

From the coordinate of each pixel and the camera settings, we compute a ray direction and check if it intersects with an object from the scene, keeping only the relevant intersection (closer to the ray origin), using the formula p.16. An intersection is defined in a separate class and consists of the

object of interest, P the point of intersection, N the unit normal at this point, t the distance and a bool *intersect()* taking as argument the ray. If there is an object-ray intersection, the texture of the object's surface determines the pixel color. During pixel color computation, it is important to offset the starting point of a reflected, refracted or shadow ray off the surface to avoid numerical precision issues.

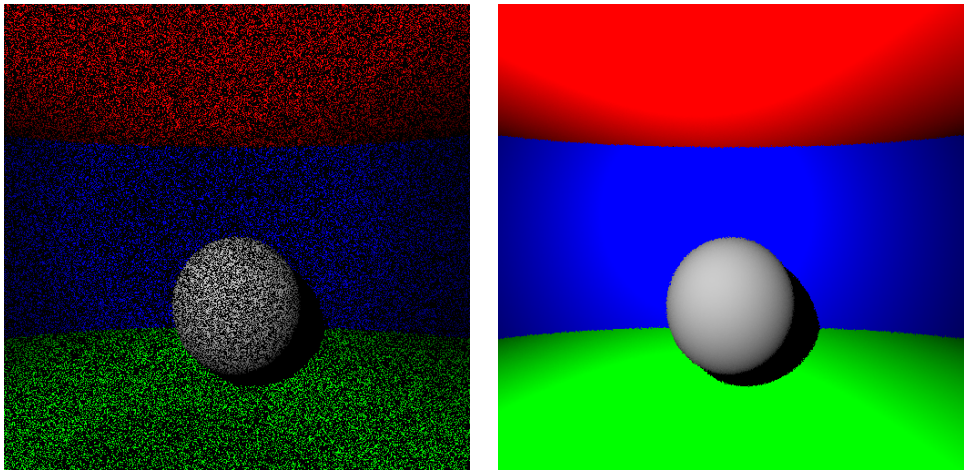

Figure 2: *Runtime : both 144ms. Left : No offset results in very noisy images. Right : The offset* ϵ *is set to 0.01.*

Shading & Shadow Computation

The pixel color is computed under the Lambertian model to handle shading and shadows using the formula from the lecture notes p.17. For instance, we introduce a visibility term that is equal to 0 if we reach a certain threshold, setting the pixel color to black to create a shadow pixel. However, shadow computation makes the scene appear too contrasted. Applying gamma correction (which consists in elevating RGB pixel values at $\frac{1}{\gamma}$ where $\gamma = 2.2$) and limiting the pixel values to the range [0..255] results in a more natural aspect of the scene.

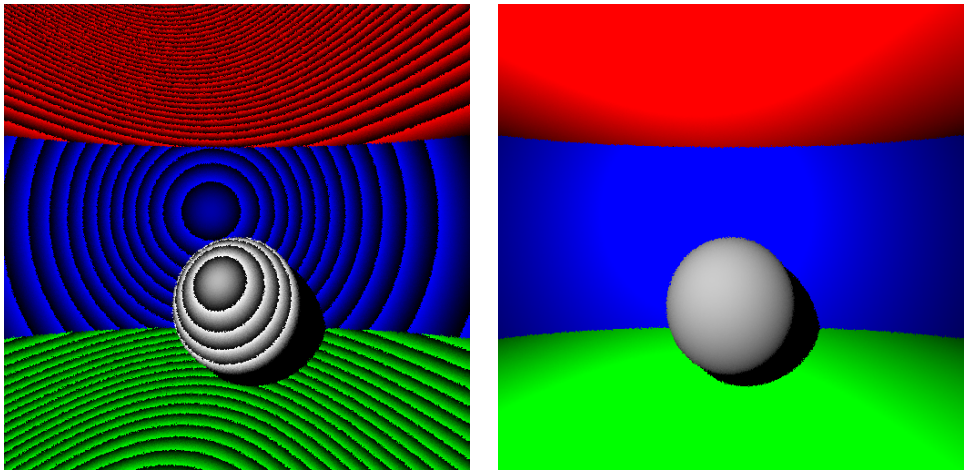

Figure 3: *Runtime : 116 VS 149ms.Left : Without gamma correction (I = 2.10E9), the scene appears too contrasted.Right : With gamma correction.*

Different Types of Surfaces

The previous pixel color computation is sufficient for a diffuse surface, in contrast light rays behave differently on other types of surfaces (diffuse, mirror or transparent surfaces). A mirror surface transfers light energy from the incident direction to a new reflected ray direction, while on a transparent surface, rays bounce off passing through it. For the latter, we use the Fresnel law to randomly launch either a reflection ray, or a refraction ray, using the formula p.22. The number of recursive calls allowed for refraction and reflection computation is controlled by the parameter *raydepth* and set to 5. As a side note, I also wanted to implement a hollow sphere using two spheres endowed with refraction, but did not obtain conclusive results.

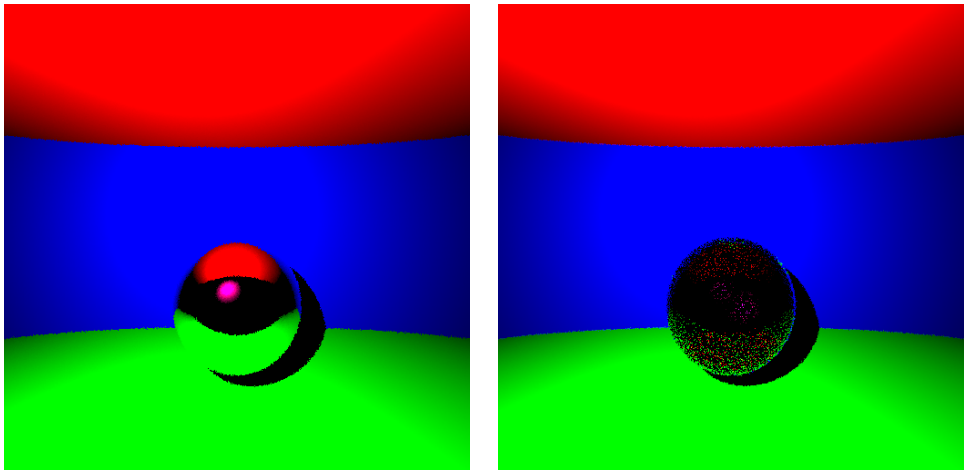

Figure 4: *Runtime : 141ms and 282ms. Left : A sphere with a mirror surface. Right : A sphere with a transparent surface.*

Indirect Lighting

To implement indirect lighting, we compute pixel color as described before, to which we add a random contribution. We obtain it from retrieving the unit normal at the intersection point and the two orthogonal tangent vectors, along with the formula p.29.

From now on, it is wise to implement parallelization in order to improve runtime. By simply adding a few lines of code (using OpenMP), we gain half the runtime.

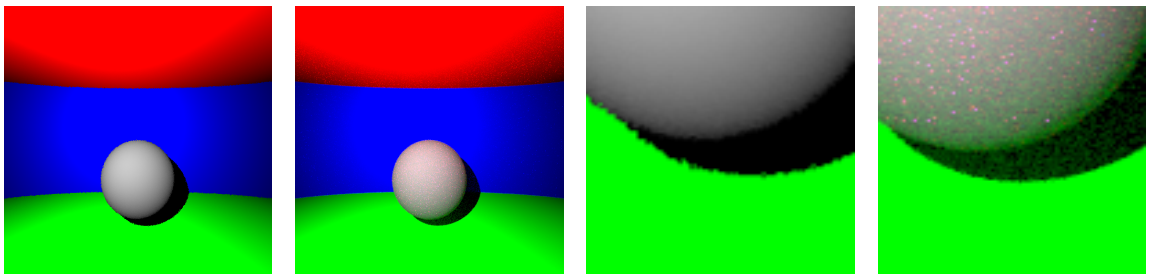

Figure 5: *Runtime : 141ms and 527ms, runtime is divided by 2 with parallelization. Left : Without indirect lighting, shadows are all black. Right : With indirect lighting.*

Antialiasing

The next step is to implement antialiasing in order to introduce more smoothness in the rendering. Now, we can launch multiple rays per pixel (controlled by the parameter *nbpaths*), and for each ray we combine the usual color computation with add a random contribution obtained from the Box Muller formula found at p.27. As a result, the pixel color is simply the mean of those color combinations for all of its rays.

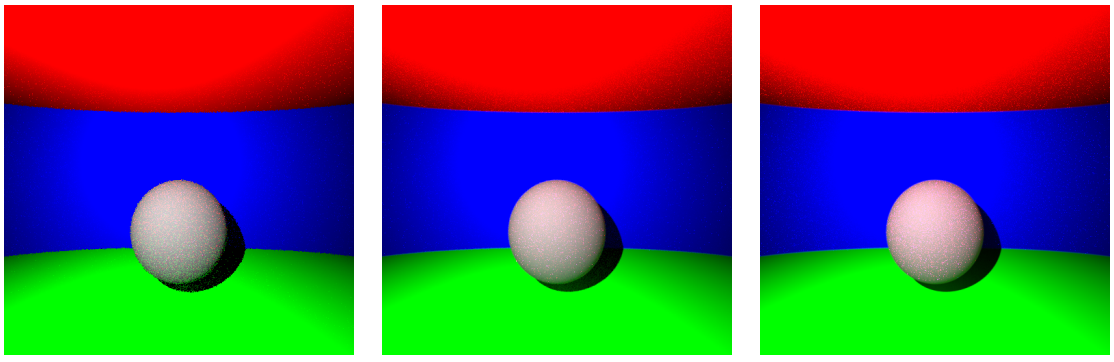

Figure 6: *Runtime : 233ms, 6917ms, . From left to right, 1, 32 and 1000 rays per pixel.*

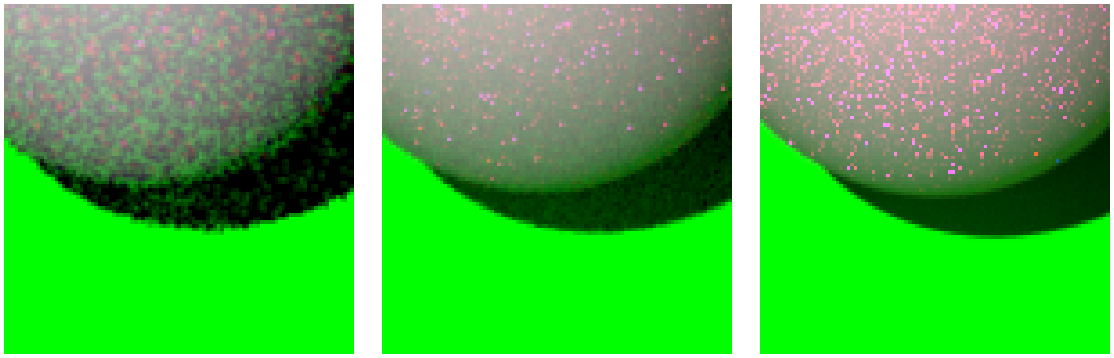

Figure 7: *Runtime : 233ms, 6917ms, 262 202ms. Zoom on the previous images.*

Ray-Mesh Intersection

Now, we add the possibility to place different types of objects to the scene. We create a class *Geometry* from which an object of the class Sphere or of the new class *TriangleMesh* inherits. The *intersect()* method of the class Scene is modified accordingly, looking at the type of object before checking its intersection with a ray. Of course, a *TriangleMesh* is also endowed with an intersection class that uses the Möller Trumbore intersection algorithm (p.42). More specifically, for all the triangles of the mesh, we use Cramer's formula to compute the distance t and the barycentric coordinates of the (possible) point of intersection, then use these results to determine if we keep the intersection or not for this triangle. It is important to translate and scale the triangle mesh before adding it to the scene so that its position and size are coherent with the other elements.

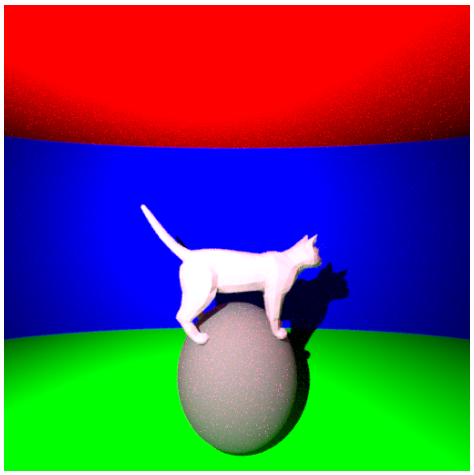

Figure 8: *Runtime : 1 910 000ms. 32 rays per pixel. With a bounding box, the runtime falls to 31 974ms and to 8 467 ms with BVH.*

- With Bounding Box

A first optimization of the ray-mesh intersection method consists in creating a bounding box around the triangle mesh. To do so, we compute the global minimum and maximum of the triangle mesh by going through all the vertices of the mesh. Now the intersect() method of a mesh first checks if the ray is within the boundaries defined by the pairs of planes of the bounding box. If not, the computations to determine if there is an intersection are avoided. **- With Bounding Volume Hierarchies**

We can further optimize the method of triangle meshes by implementing a recursive version of bounding boxes called Bounding Volume Hierarchies. We need to add a class representing the node of a BVH tree, objects of the class *bvhNode* are endowed with two children, indices to give a range of the corresponding triangle indices, and the bounding box around this node. In fact, we change the way we compute bounding boxes for triangle meshes: instead of having only one as before, now we create several ones for the same mesh, each time taking into account only a selection of vertices. So, in the main function we build the root of the BVH tree before adding the mesh to the scene.

References

I got inspired a lot from live coding during the tds, lecture notes, slides, but also did personal contributions and modifications. My goal was to successfully understand the concepts of the course as well as the role of each line of code in the overall process.