CSE306: Ray Tracer Report

MAMOUN ELKHETTAR

May 3, 2020

This document reports the step taken and the work achieved to build a full **Ray Tracer** in *C++* for the class CSE306.

1 Path Tracing

1.1 Rendering Spheres

To start our **Ray Tracer** we many different objects to define in our order to construct our scene. We then define the following **struct** with their respective arguments :

- **Vector** : an array of 4 **doubles** x, y, z, w to represent the position of the points.
  We also define a instance of **Vector** with one argument to be used when we want to perform operations between scalars and vectors. Moreover, We define a set of operators to be used on **Vectors** (+, -, ×, /, dot product and norm).
- **Ray** with 2 **Vector** members, **o** for the origin vector and **u** for the unit direction vector.
- **Intersection** with 2 **Vector** members, **point** for the intersection point and **normal** for normal of the point, also a **double** **t** and **bool** **exist** to characterize the existence of the intersection of a **Ray** and an object.
- **Sphere** with 2 **Vector** members, **center** for the center of the sphere and **albed** for the color of the sphere, and **double** **radius** for the radius of the sphere. We also define an **intersect** method that inputs a ray and return a Intersection object.
- **Scene** with a **std::vector** of Sphere, **o** for the origin vector and **u** for the unit direction vector.

1.2 First sphere

In our first step, we want to render a white **Sphere** of center position (0, 0, 0) in a **Scene**.
In our **main**, we define the different components we need : a Camera Q of as a **Vector** for its position:

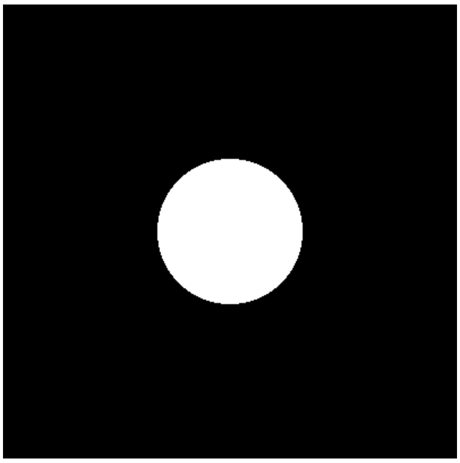
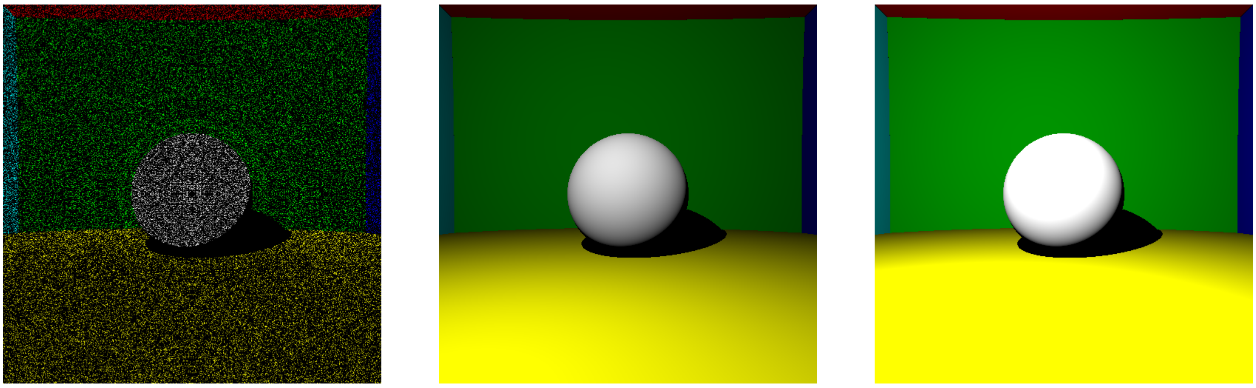


Figure 1: White sphere rendering

(0, 0, 55), a Light source, a **Scene** with 6 spheres (defined in the poly) as and the **Sphere** we want to render as object. We need to generate an image, we use the $stbi\_write\_png$ available in *stb image* that needs a vector of pixels as input. We then scan all pixels by projecting rays from the Camera point with direction to each pixels.
First, we only want to display the sphere without considering the other spheres composing the scene and the Light source. For each pixel, we check the intersection of the **Ray** with the our **Sphere** object and intersect function. If there is an intersection, i.e. *intersection.exist = true* , we project a white pixel and if there is no intersection we project a black pixel. We obtain the image in Figure 1.

1.3 Shadowing

As a second step, we want to compute the shadows using the Lambertian model. We add a method, **intensity**, to our **Scene** object which returns the intensity reflected of a surface as a **Vector**.
We add an other member to the object **Intersection**, **int index**, to store the index of the sphere with the closest intersection in the scene computed in the method **intersection** we add to **Sphere**. Now, rather than checking if there an intersection between the incoming rays from the camera and our sphere, we check for the intersection between those ray and the closest sphere in the scene with the method **intersection** from which we project a **Ray** from the intersection point toward the Light source and consider the intersection with the Light source or not to compute shadows.
We obtain the image **(a)** in Figure 2, it is quite noisy as we get black pixels as the results of the intersection from the intersection point is itself. We correct this using an offset to launch the ray from the intersection towards the Light source, see **(b)** in Figure 2. Final, in this part to get a clearer and more realistic image, we apply a gamma correction, result seen in **c** in Figure 2. We obtain the latest in **90 ms** approximately (using clock function) for a 512×512 image.



(a) First image we get

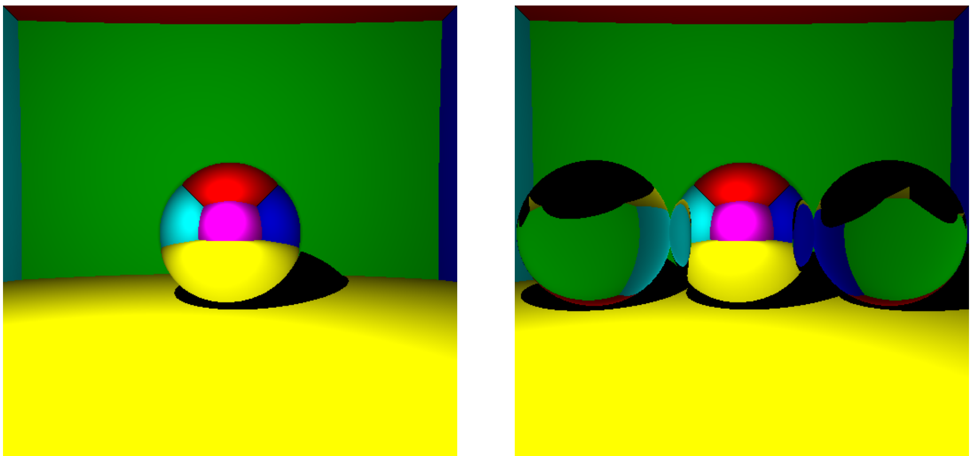
(b) Image with offset

(c) Image with gamma correction, I=1E5, 90ms, 512x512

Figure 2: Images with shadowing

1.4 Reflection

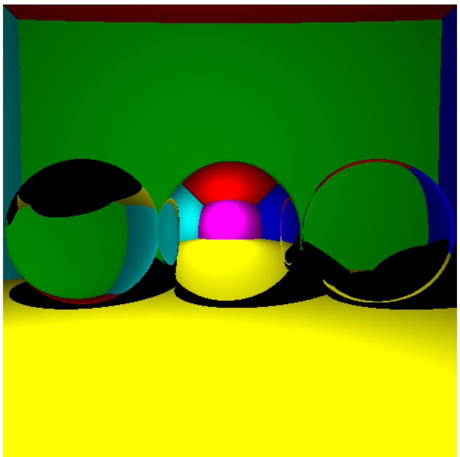
Now, we want to add the option reflection to our spheres. To do so, we add a **bool** member **mirror** to **Sphere** to store if we want the object to have reflection. To compute the reflection, we add a **getColor** method to **Scene** which will be called in **main** with input the incoming ray from the camera, ray depth and light source and return the color to project. In this method, we check if the sphere at the intersection point has a true mirror Boolean. We then recursively call the method with the ray define as in the poly. If intersection.mirror = false we use the **intersection** to compute the color. See the result in (a) in Figure 3. Here the execution time has slightly increased with the same parameters as in Figure 2 with shadowing.



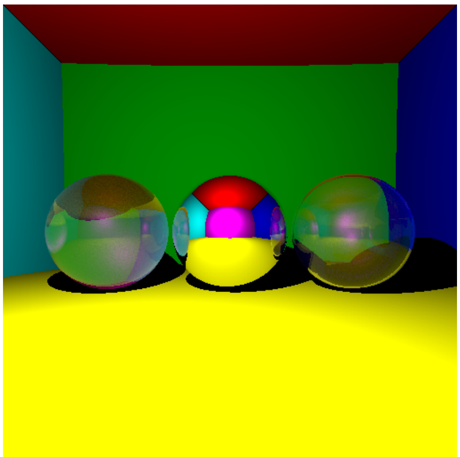
(a) Image with reflection, I=1E5, 100ms, 512x512

(b) Refraction, I=1E5, 100ms, 512x512

Figure 3: Image with reflection, I=1E5, 100ms, 512x512



(a) Refraction with hollow, I=1E5, 150ms, 512x512



(b) Fresnel Refraction, I=1E5, 34 sec, 512x512

Figure 4: Refraction

1.5 Refraction

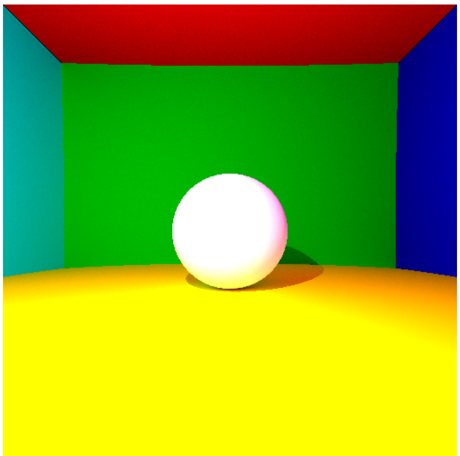
Now, this is part is very challenging. We add refraction and then use the Fresnel law to get a better refraction. We did some change in the Sphere class, we added a **enum** Property class to determined if the sphere has diffuse, mirror or transparent property instead of boolean members for each property. For the normal refraction, we use the method in the lecture and get (b) in Figure 3. Moreover, we added **double** ref_index (by default 1) and **bool** **hollow** (by default false) members to **Sphere** to input the refraction index of the sphere when needed and if the we have a hollow sphere. The hollow is in (b) Figure 4. Finally, we get the Fresnel Refraction in (c) in Figure 4 with a bigger field of view and more space spheres. As I struggled a lot with this part, I had help from Carolina for the Hollow Sphere and Maria for the Fresnel refraction.

1.6 Indirect Light and Anti-aliasing

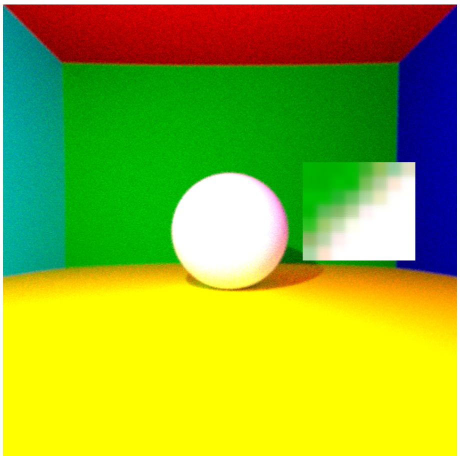
For this part, we just added the indirect part in the **getColor** function with the $random\_cos$ function. We get issue with parallelization on Mac OS X so the computation time is not optimal so we don't use a big K. For anti-aliasing we used the **boxMuller** The results can be found in Figure 5.

1.7 Ray-Mesh Intersection

For this part, what we did is integrate the given **file reader** to render objects such as a cat. We implemented the Ray-Mesh intersection function and used the parameter in the lecture notes. For the scale and transpose, I directly performed it on **readOBJ** function. Results in Figure 6.



(a) Indirect light, I=1E5, 7min, K=1000, 512x512



(b) Anti-aliasing, I=1E5, 30min, K=100, 512x512

Figure 5: Indirect Light and Anti-aliasing

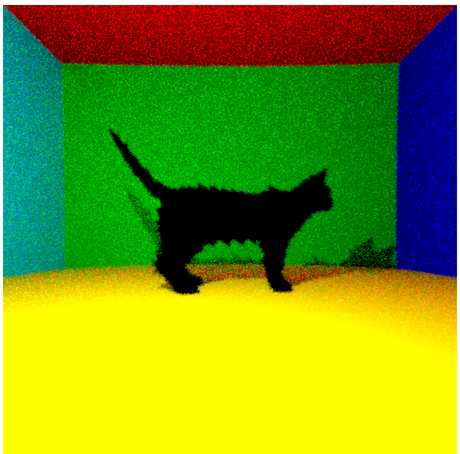


Figure 6: Cat rendering, 60 min, ray depth of 3, K = 15