CSE306 Project I - Ray tracer

Yinfeng Yu, May 2020

Introduction to the architecture

This project implements a preliminary ray tracer from scratch which is able to visualize some certain kinds of geometric objects. The scene is shown by physically-based rendering and step by step.

The project consists of a few directories: class, function, include, image, and *main.cpp* with a *makefile*. It is not necessary to have so many files and directories, as I did at the beginning, but it raises extra problems and poses extra difficulties when everything is packed together.

The class directory. It possesses all the definitions of the objects in the scene and their functions. *ray.h, geometry.h, sphere.h, mesh.h, scene.h* have the class of their name and *structs.h* defines Intersection, Camera, Image structs. A different design from the instruction of the lecture notes is that the construction of the scene and the scanning of the image is not directly coded in main function, but enclosed in the Scene class.

The function directory. It possesses only one header file which defines some useful but independent functions, `boxMuller()` and `randon_cos()`.

The include directory. It possesses the very important *stb_image_write.h* and an *include.h* which packs all the header files together to be imported in the main function.

The image directory. All the images generated will be put here.

The *main.cpp* does very simple job since the complex scanning part is defined within the Scene class. It only sets up a Scene objects and call the `scan()` function, save the picture in the right place and tell us how much time it takes to do all these things. The *makefile* has the compiling command of *main.cpp*, options for enabling parallelization are specified here.

All the computations all done by a MacBook Pro 2016 model, intel i5 2.9 GHz with 4 cores (it is really slow as the cases where the number of rays per pixel raises and it is one of the main obstacles).

Build the scene and implement the intersection algorithm

The scene consists of 6 gigantic spheres as walls and has some objects in the middle. This very first task is to have a sphere visible in the middle of the picture. Along with the definitions of the objects, such as ray, sphere, the intersection algorithm between sphere and ray is implemented and tested. To well visualize the scene, the algorithm of detecting the closest intersected sphere is also necessary. All the objects and camera are placed and set as described in the lecture notes.

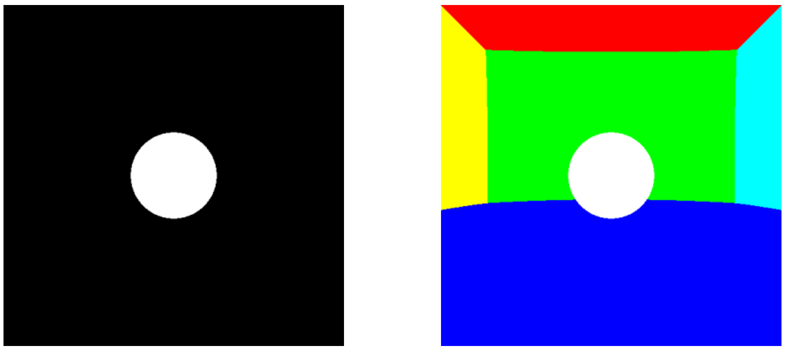
Figure 1: Left: sphere intersection algorithm test. Right: closest intersection test. With the detection of the closest intersection, we can preliminarily show the scene.

Direct lighting (diffuse material)

The next thing is to add direct light source to the scene. Implementing the given formula to realize the direct lighting and the test of shaded area yields the following images which has a diffuse sphere in the middle of the scene, where the walls are all diffuse. To silence the noise, a very small offset has to be added to the origin of the ray testing the accessibility of the light source. Also, gamma correction allows us to have a more brightened picture which is more realistic (otherwise the image would be too contrasted). I built the light source as a white sphere but it is not implemented as an area light source.

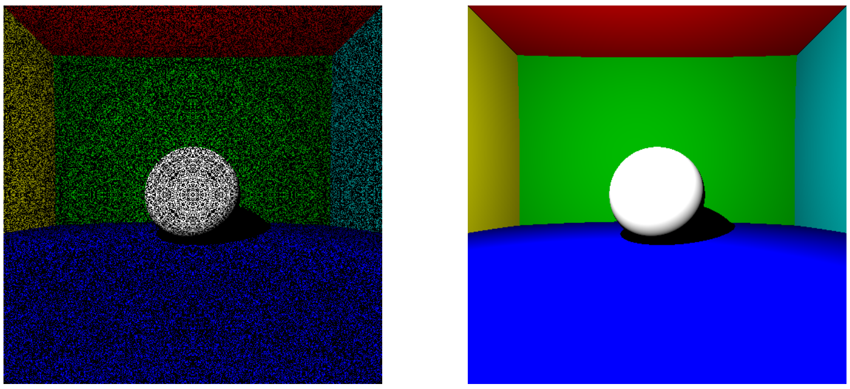
Figure 2: Left: without offset. Right: with offset.

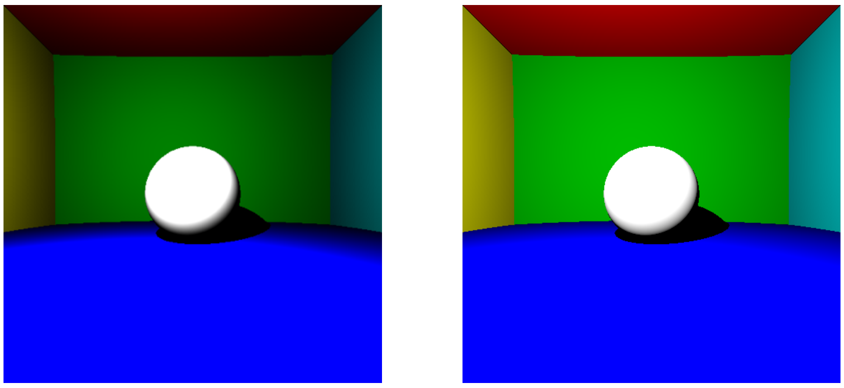
Figure 3: Left: without gamma correction. Right: with gamma correction.

Reflection (mirror material)

As to this point, to distinguish the diffuse material and mirror material for the rays, a new attribute called "type" is added to the sphere class. The reflection formula is simple to implement.

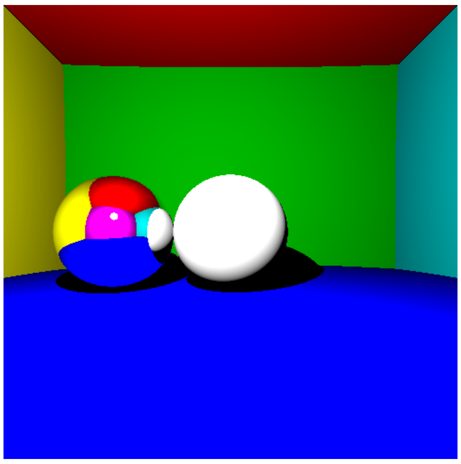
Figure 4: Mirror material sphere and diffuse material sphere.

Refraction (transparent material) and Fresnel law

Implementing transparent objects is similar to mirror objects but is slightly more involved. The direction of the norm vector has to be paid extra attention to make the rendering successful (I spent very much time on stopping the transparent material from showing in black). Also, the depth of the ray has to be set above 2 to assure the light can go through the sphere.

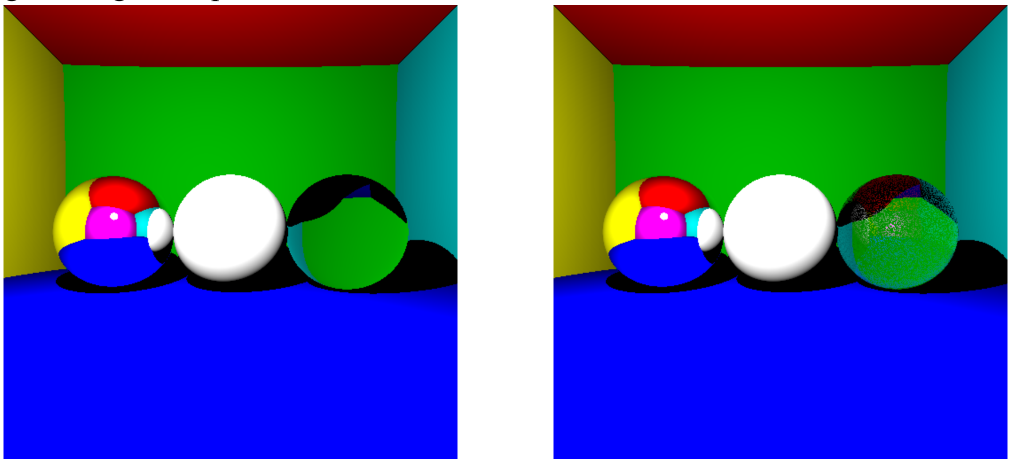
Figure 5: Mirror sphere, diffuse sphere and transparent sphere. Left: without Fresnel law. Right: with Fresnel law. The refraction coefficients are set to 1 and 1.5 so that the transparent sphere looks like glass.

Adding Fresnel law makes the transparent sphere reflect rays. But the result has too much noise thus multiple rays per pixel is needed, as we will implement for the indirect lighting.

Indirect lighting

Adding indirect lighting is basically making the diffuse material reflect light (By common sense, the shadow within a closed space should not be totally black). A function for generating randomly directed vectors is coded so that we can simulate the reflection on the diffuse surface by sampling multiple rays per pixel.

Before adding the indirect lighting, the executing time is far below 1 second (the very first pictures) or around 1 second (with direct lighting). But with indirect lighting, the program becomes very slow. Parallelization is thus applied and the executing time is significantly shortened (although the CPU of my computer reaches its limit, but the speed is not good enough to perform more delicate simulations).

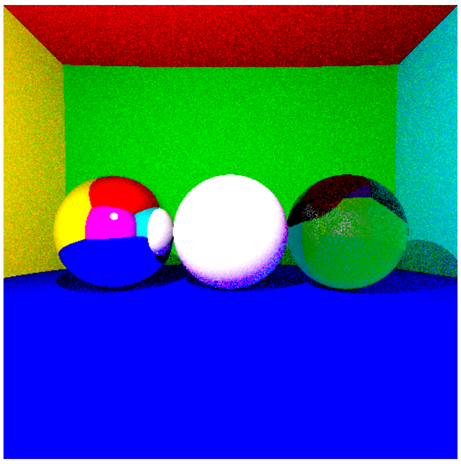

Figure 6: Image of size 512 × 512, 10 rays per pixel, with indirect lighting, takes 68.9 s without parallelization, takes 26.4 s with parallelization.

Anti-aliasing

A very tiny random offset is added to the ray emitted from the camera to achieve anti-aliasing. This makes the picture a little bit more blurred but with less zigzags.

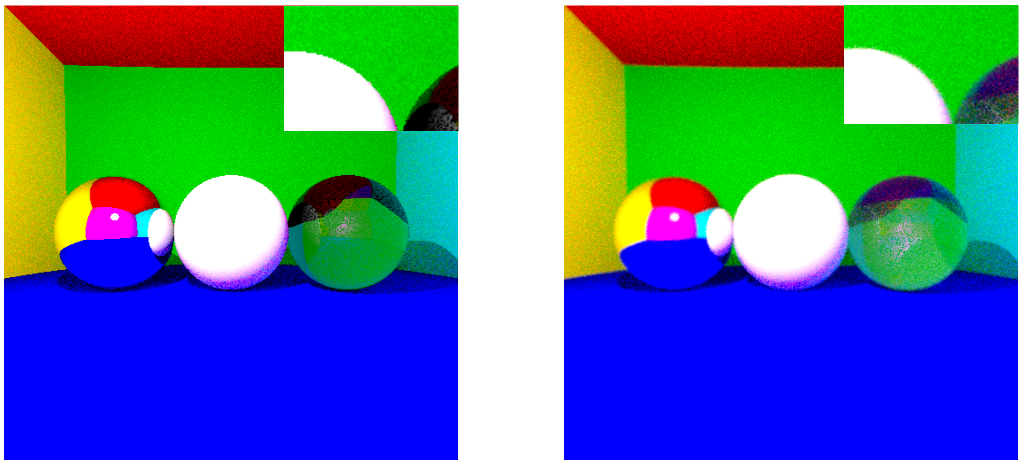

Figure 6: 32 rays per pixel, with indirect lighting. Left: without anti-aliasing. Right: with anti-aliasing.

Implement the intersection algorithm for triangle mesh

The Moller-Trumbore algorithm is implemented and the cat model is imported. Unfortunately, although the shape of the diffuse cat model is revealed, it mainly consists of indirect light, only some parts that are not straightly facing the camera show up with the desired color. Moreover, half of the tail of the cat is missing. By changing the position of the cat, we can see the miss tail shows up but part of the forelegs are missing again. This may due to the imperfection of the implementation of the algorithm. Performing this simulation is too time-costly thus no more progress is made at this stage.

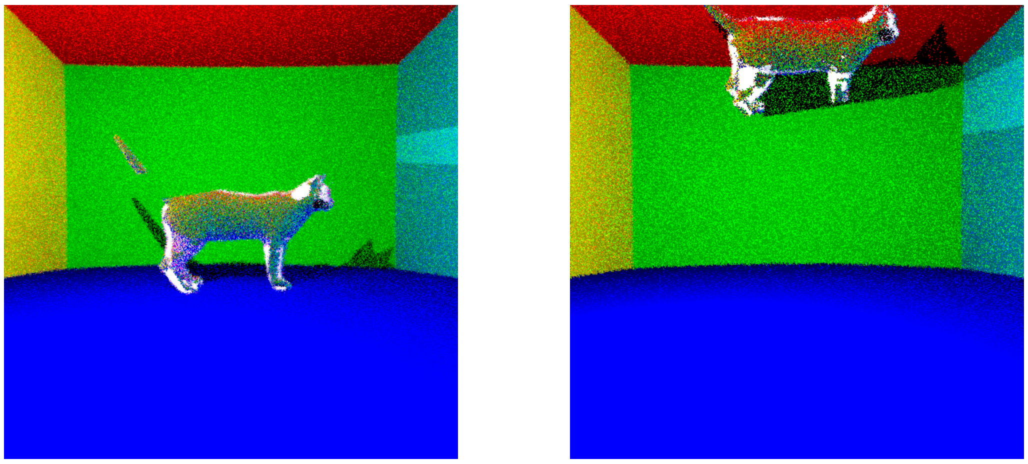
Figure 7: Left: cat model translated by (0, -10, 0), 5 rays per pixel, takes 5683.6 s (about 1 hour and a half). Right: cat model translated by (0, 20, 0), 2 rays per pixel, takes 2401.8 s (about 40 minutes).