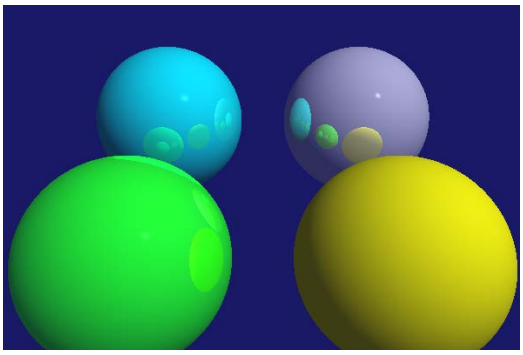
Lab 5: To Infinity and Beyond! (Ray Tracing)



In this assignment, you will program a ray tracer. Your ultimate goal is to generate a ray traced picture on the right. You are given a skeleton file with a lot of helps like vector operations. And we will also help you by breaking down this whole assignment into smaller steps.

Structure of the Given Skeleton

There are two cpp files given in the solution file. One is the `main.cpp` that contains all main operations for the ray tracer. The other is the `vector3D.cpp` (and `vector3D.h`) that contains a library for basic vector operations. For the usage of the vector package, you can get a kick start by looking at the function `renderScene()` in `main.cpp`.

Default Setting

In the beginning of main.cpp, you will see:

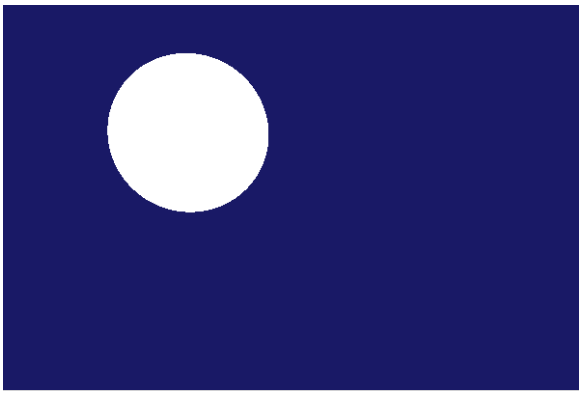
```
#define WINWIDTH 600
#define WINHEIGHT 400
#define NUM_OBJECTS 4
#define MAX_RT_LEVEL 50
#define NUM_SCENE 2

float* pixelBuffer = new float[WINWIDTH * WINHEIGHT * 3];
```

There are two scenes (= NUM_SCENE) and the number of objects in the scene is 4 (= NUM_OBJECTS). You will create two pictures (= NUM_SCENE) with the dimension of WINWIDTH x WINHEIGHT (600x400) and their RGB colors will be stored in the array **`pixel Buffer`**. The program will first use the ray tracer to generate the color of each pixel in the picture and store them in **`pixelBuffer`** first. And then the buffer will be copied to the screen and displayed by **`glDrawPixels`** in the function display. In order to store the color of a pixel into the pixel buffer, you should use the function **`drawInPixelBuffer()`** provided in the code. So your real job begins in the steps below. And you should be able to find the location of each step in the code by search for "step n".

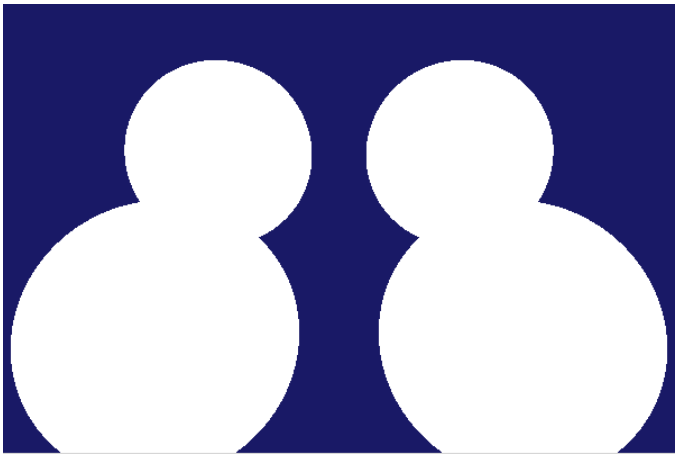
Step 1: Computing the intersection of the first object.

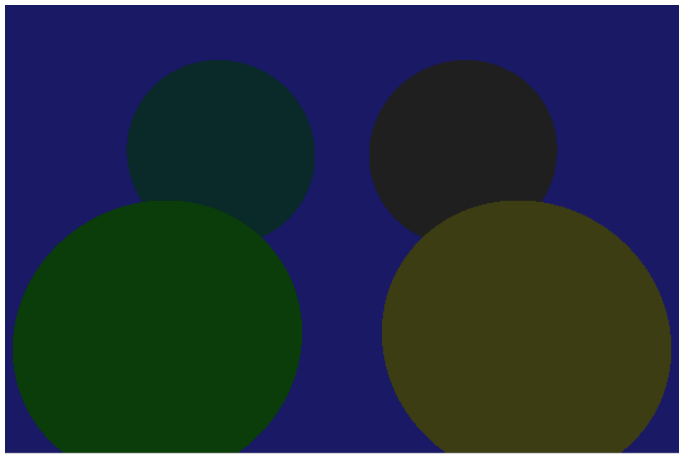
The function **`intersectionWithRay`** should compute the intersection between the ray and the sphere, and output the smallest positive value of t according to the lecture notes. The position of the intersection and the normal vector at the intersection should be returned through the parameters. If you do this part of code correctly, you should be able to find out all the intersection of the ray with the first sphere and colored the pixels with white (the coloring part is near the location of Step 2 in the code).



Step 2: Computing the intersection with all objects.

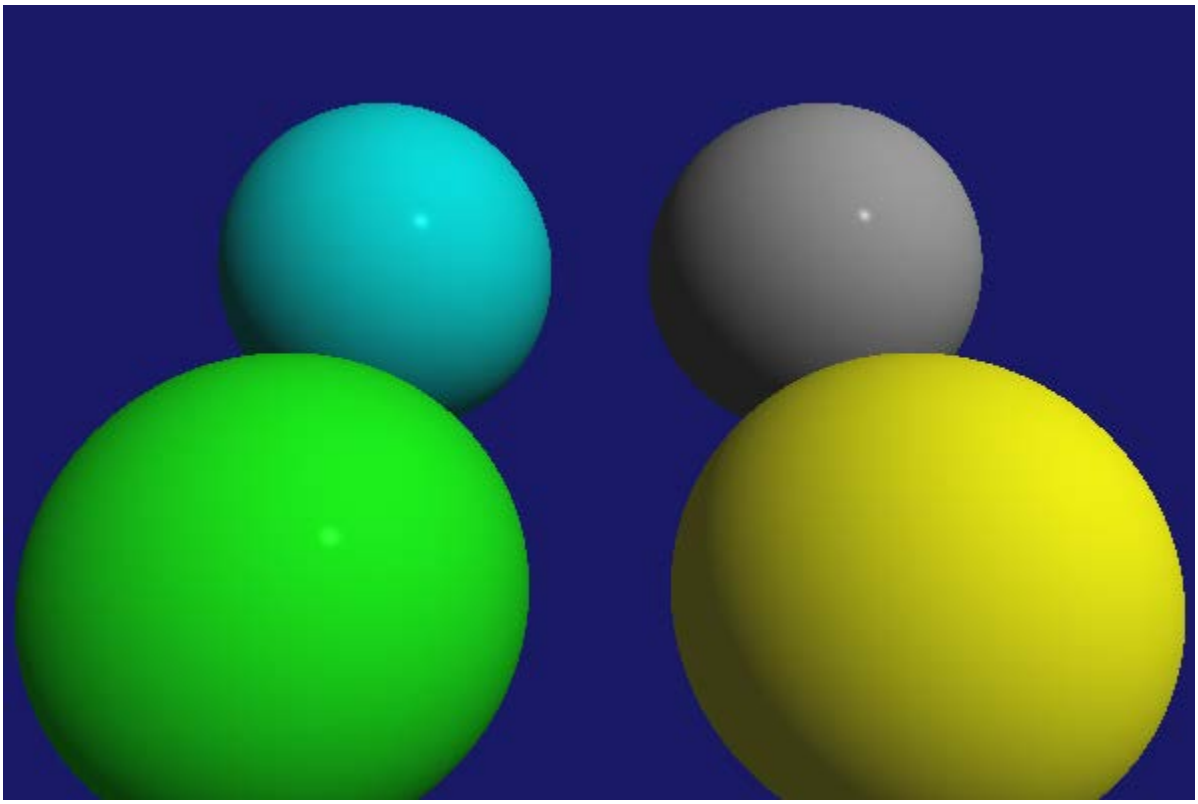
Uncomment the for loop in the code and it should compute the intersections for all the four spheres in the scene. And it should produce the four white spheres (Left). Next, assign each pixel of the spheres with the ambient color of the spheres. You can access the ambient colors by `objList[0]->ambiantReflection[i]` with i is the different RGB values. And you will produce the color balls like the ones (right).





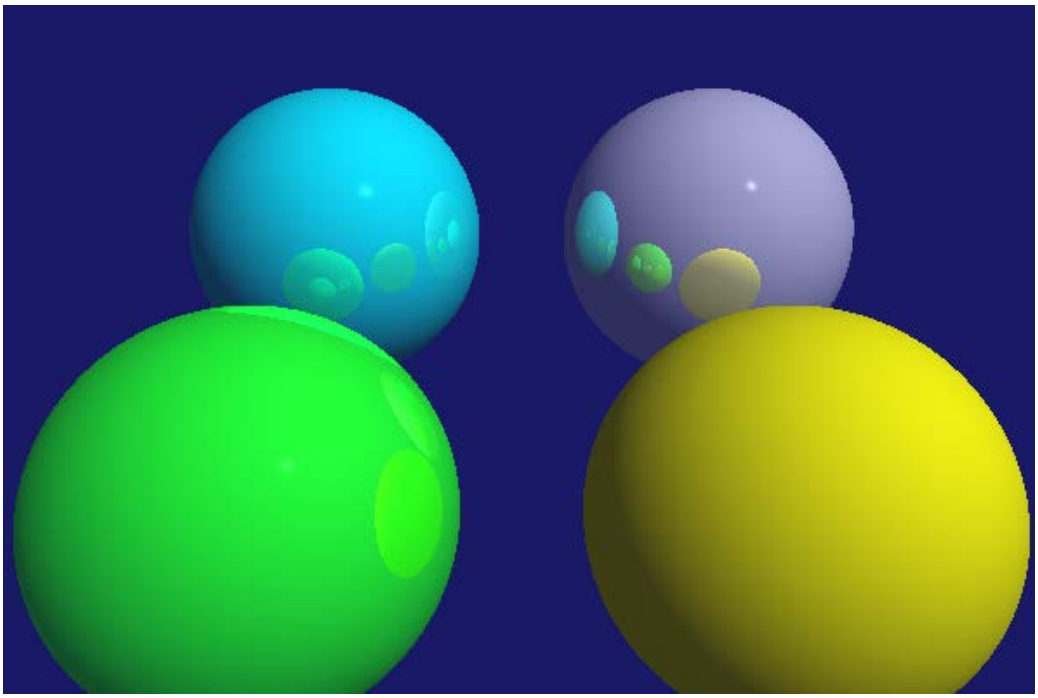
Step 3: Shading and Illumination

Compute Phong Illumination Equation for the points on the sphere according to Lecture 8.



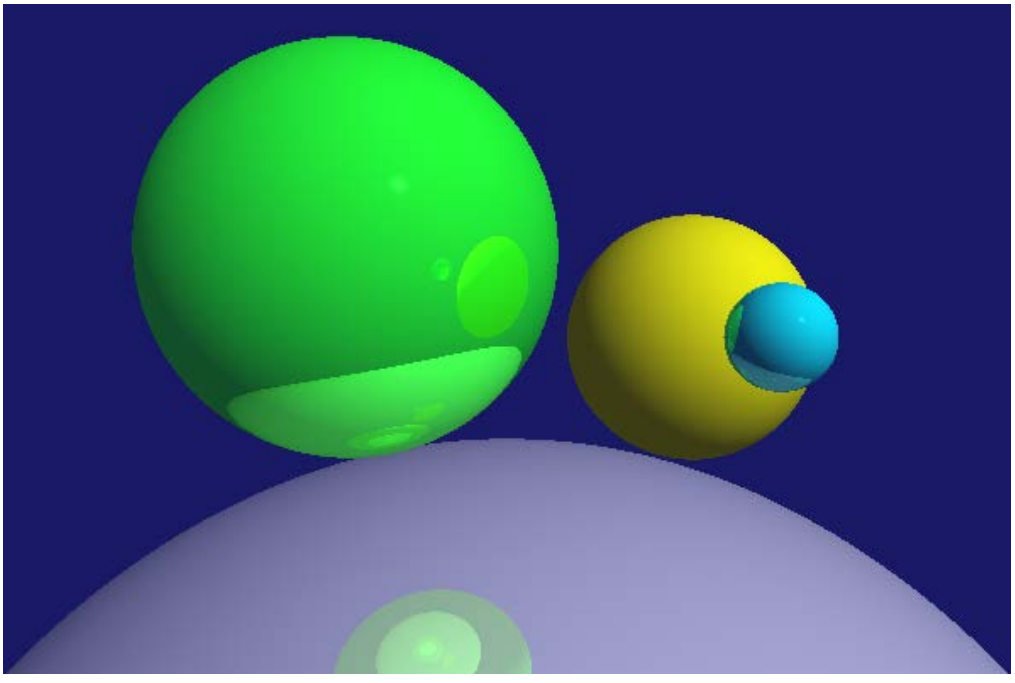
Step 4: Ray Tracing

Finished the code by recursively tracing the ray. The recursive level should not exceed `MAX_RT_LEVEL` . You can assume that a ray will not hit an object where it rebounds from. So the variable `fromObj` is for that purpose. If your reflection is done correctly, you should produce a picture like this. Note that the yellow sphere is supposed to be dull and have no reflection.



Step 5: Add another scene

By pressing the letter 's', another scene is rendered. Basically the algorithm should be the same and all you need to do is to change the settings. Your job is to create another scene by changing the positions and the material properties for the objects. Try to avoid having intersecting objects.

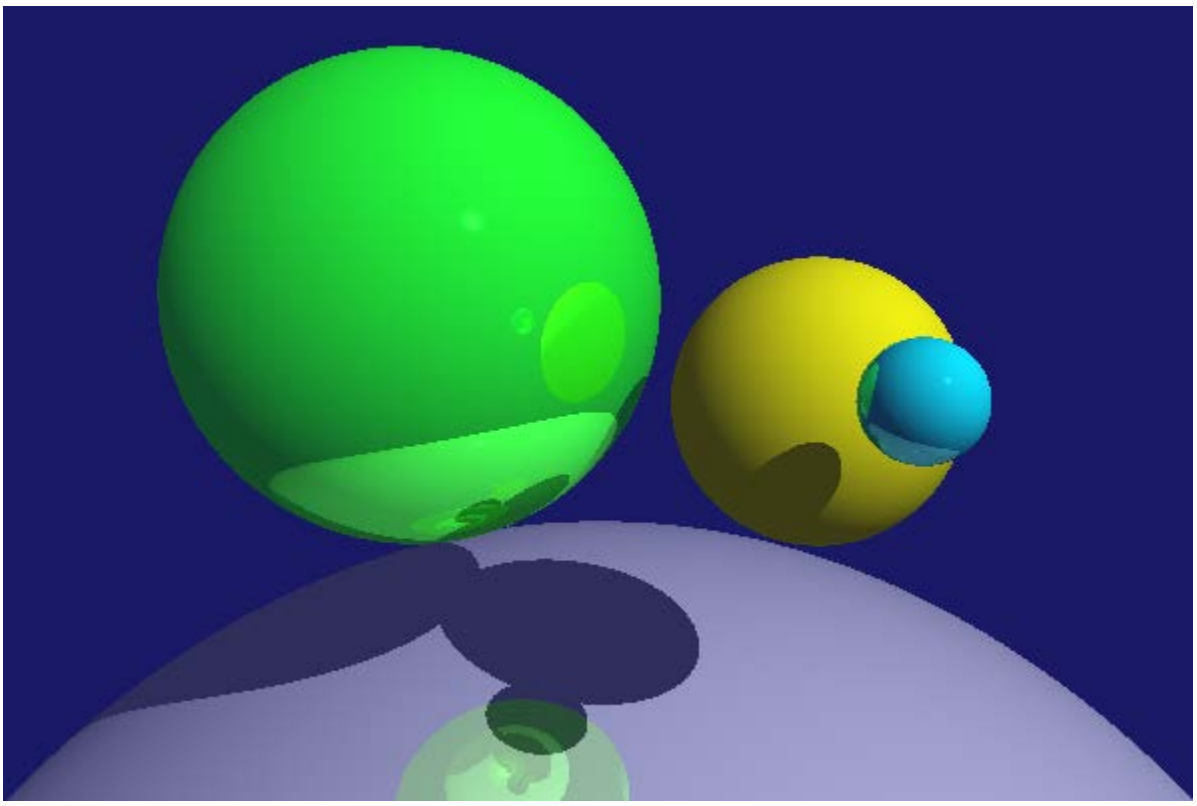


Submission

You should only modify and hand in the file main.cpp.

Extra features:

1. Add shadows



2. Add another type of objects, e.g. conics, polygons

3. Add transparency.

4. Texture map, or other mapping techniques such as bump map.