Ray Tracer Assignment

1. What is expected by this assignment?

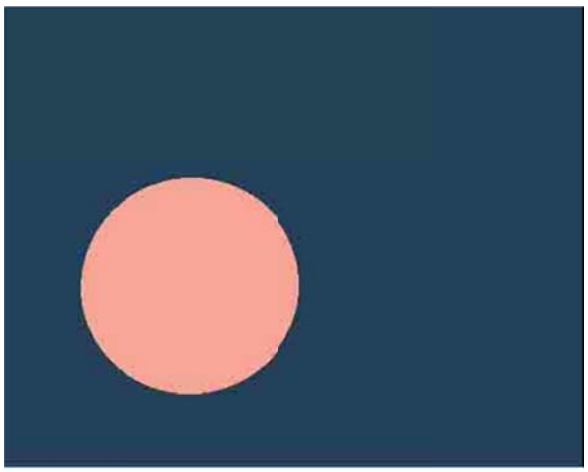
In this assignment you will modify the “toytracer" (toy ray tracer) skeleton that has been supplied for you in the EEE dropbox. The basic job of the “toytracer” is doing ray-sphere intersection, which is considered as an example for you to complete your implementation. By adding codes in the proper way at the proper position, you should be able to provide the following functionalities: (1) ray-box intersection; (2) Phong shading; (3) point light source shadowing; (4) reflection. Together, they make the “toytracer” a reasonable offline rendering tool featured with ray tracing.

2. What do you need for this assignment?

a. A C++ programming IDE, since the “toytracer” is fully implemented in C++;
b. The skeleton program, included by the package from EEE;
c. The *.sdf scene files, included by the package from EEE, and possibly set by yourself;
d. A software can view *.ppm image files, IrfanViewer is one of them for Windows, download from:
   http://irfanview.tuwien.ac.at/
   Toy Viewer for Mac users, download from:
   http://www7a.biglobe.ne.jp/~ogihara/software/OSX/toyv-eng.html

3. What do you need to do for this assignment?

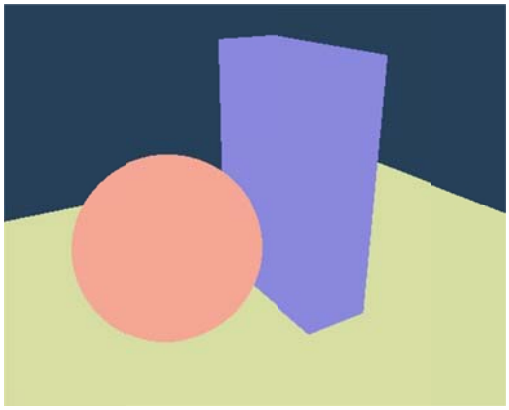
a. Download the basic “toytracer”, compile it and run it with the scene1.sdf file. Use your image viewer to check the generated *.ppm file, it should be a circle in a dark background, like this:



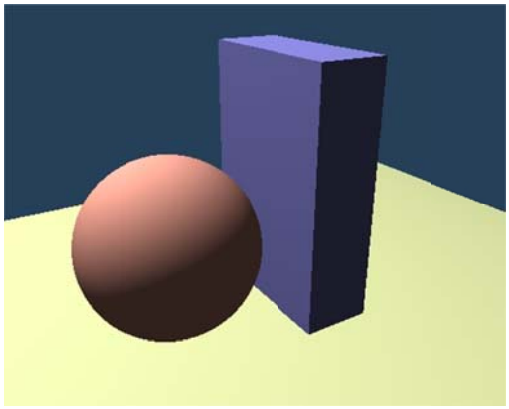
b. Have an overview of the program. The program is fairly well documented internally, so you should not have too much trouble to find out what needs to be added and where. Note that although there are a lot of source files there, not all of them are involved in this assignment. Some of them you should definitely understand are included but not limited by the following list:

   - Main.cpp, where you set your console prompts, load files, and run the whole program;
   - Toytracer.h, where you understand the HitInfo structure;
   - Ray.h, where you understand how a ray works;
   - Sphere.cpp, which works as an example for this assignment. By understanding it, you should be able to have a basic idea that what functions are necessary for a shape object, and how you should add codes for other shapes by yourself;
   - Block.cpp, which is the shape you should implement by yourself;
   - Basic_shader.cpp, which is the key part of this assignment. Most of your efforts will be put on it to make it render the scene in the way you want.

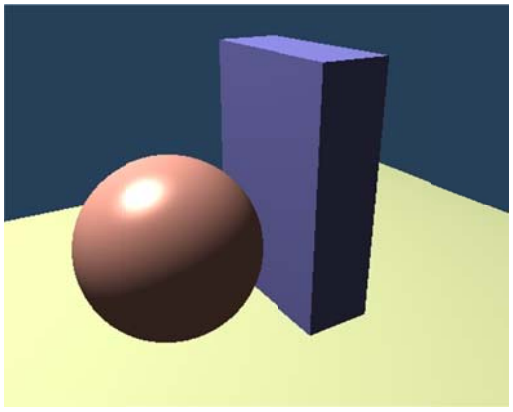
c. Fill in the **Intersect** method of the Block primitive object. All of the other methods have been provided for you, but you should understand what they are doing both in the **block.cpp** module and the **sphere.cpp** module. The basic approach for intersecting a ray with an axis-aligned box, using three orthogonal “slabs," will be outlined in class. After you have the **Intersect** method works for the Block primitive object, you should have one box and a ground appear in your scene image, like this:



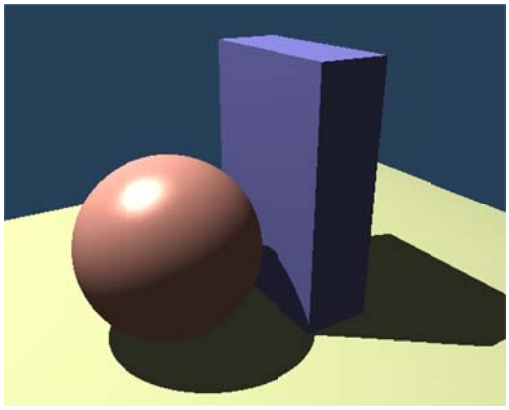
d. Fill in the **Shade** method of the **basic_shader.cpp**. Implement the diffuse shading based on what you have learned from the class. After that, your objects should be shaded, like this:



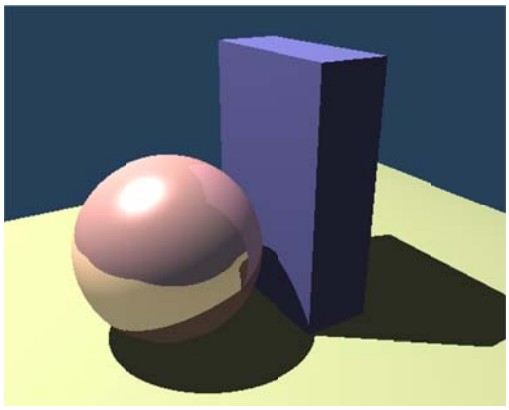
e. Continue to fill in the **Shade** method of the **basic_shader.cpp**. Implement the specular shading based on what you have learned from the class. So far, you have completed the Phong shading, which you usually do by OpenGL. After that, your sphere objects should be highlighted, like this:



f. Continue to fill in the **Shade** method of the **basic_shader.cpp**. Implement the point light source shadow effect. Remember shadow is nothing but a mask which indicating pixels cannot be lit because of occlusion. To create this mask, you should study the **toytracer.h** and the **ray.h**. After that, your scene should have shadow now, like this:



g. Continue to fill in the **Shade** method of the **basic_shader.cpp**. Implement the reflection effect. Reflection is achieved by ray bouncing. Here, utilize the **generation** property of the **ray** object in **ray.h.** Finally, your scene looks different from a simple OpenGL-rendered scene. Here is an example:



h. Keep two things in mind:

- Always completely finish a stage before you move to the next one, and always back up your bug-free code before you do further modification. You will not be able to debug if you lose track with such a complex program.
- Images you generated may not be exactly same as the examples above. However, if a major feature is missed or color pattern is totally different, then you must have done something wrong.

i. Feel free to try new scenes with different shapes and different physical phenomena. It is always your own wealth if you learn something new.