CPSC 314
Assignment 4: Ray Tracer

Due 4PM, Dec 2, 2011

In this assignment you will implement a simple raytracer that supports spheres, planes, and optionally other types of surfaces. The raytracer should cast primary rays into the scene, which spawn secondary reflection/refraction and shadow rays. The goal of the assignment is to experiment with advanced rendering tools and to get hands-on experience with both lighting and geometry manipulation.

Extra credit points are available for extending your program to support additional features.

Template: The template code has three main subdirectories, `src`, `include`, and `scenes`. The `scenes` directory contains scene descriptions in the `.ray` format, describing the following scene parameters: Dimensions, Perspective, LookAt, Material, PushMatrix, PopMatrix, Translate, Rotate, Scale, Sphere, Plane, and PointLight. The comments in those files describe the format. The directory also contains reference images created by the solution code which you will find very useful for debugging.

You will be making additions to two of the template code files in the `src` directory: `object.cpp`, and `raytracer.cpp`. You may also change the `mesh.cpp` file if you choose to implement the optional rendering of meshes. You do not need to make any changes to the other source files.

The `README` contains instructions for compiling and running your raytracer. The `raytracer` binary takes two optional arguments: the name of the scene description, and the name of the output PPM image file. The defaults are `scenes/basic.ray` and `output.ppm`. The output of the program is two image files, a color image and a black-and-white depth map image that you might find useful for debugging. The name of the depth map image file is `filename_depth.ppm`, where `filename.ppm` is the specified output image file.

A reference solution Linux executable `raytracer_sol` is provided for comparison.

As usual, your assignment consists of several mandatory components and a number of optional ones. The mandatory components are as follows.

- **15 pts** Implement the missing parts of `Raytracer::render` and `Raytracer::trace` for basic ray casting for all pixels in the image, using the camera location and the coordinates of each pixel. You can test this code by re-computing the pixel as the intersection of the ray and the view plane and testing that you obtain the same coordinates back.

- **10 pts** Implement `Sphere::localIntersect`. Test your result by comparing the depth algorithm's outputs of your and solution methods on the provided scenes. You can also render the spheres using the diffuse coefficients provided.

- **10 pts** Implement `Plane::localIntersect` and test in a similar way (note that as you do it new objects will appear).

- **20 pts** Implement the missing part of `Raytracer::shade` that does a lighting calculation to find the color at a point. You should calculate the ambient, diffuse, and specular terms. Test your results by comparing to the ground truth ones.

- **15 pts** Implement the shadow ray calculation in `Raytracer::shade` and update the lighting computation accordingly.

- **20 pts** Implement the secondary ray recursion for reflection in `Raytracer::shade`, use the `rayDepth` recursion depth variable to stop the recursion process. (The default used in the solution is 10.) Update the lighting computation at each step to account for the secondary component.

The implementation so far gives you 85 points. To obtain the remaining 15 you should implement one of the three options below.

- Implement `Conic::localIntersect` to enable intersections between the rays and generalized conical surfaces (`http://en.wikipedia.org/wiki/Conical_surface`) . Note that this requires detecting the bounding circles of the conics and accurately handling those (to get finite cylinders/cones/ellipsoid parts).

- Implement `Mesh::localIntersect` to enable intersection between rays and meshes. You can reuse your plane intersection code, but need to handle interior tests for triangles.

- Implement secondary ray recursion for refraction rays. Use the same recursion depth variable `rayDepth` as for reflection to stop the recursion process. Update the lighting computation at each step to account for the secondary component

For those who want to further explore, up to 20 extra points will be given for implementing either several of the optional enhancements above or one of the following. To better demonstrate your add-ons, you may consider generating additional input scene files.

- Soft shadows - convert each point light source into an area one and use random samples on it as light sources.

- Texturing - use the provided `Image` class to import textures and access the texture during ray-tracing to get a local diffuse color.

- Speed - consider speeding up your method using any of the space-partitioning methods discussed in class. The template provides a timer which you can use to compare your result to those of others and the ones in the solution.
- Gloss - use randomized direction estimation to account not only for specular but also glossy surfaces.

The comments in the template code above each section where are you required to add code contain the details of the specification. They also contain many hints. The recommended order of implementation is exactly the order we list the items above.

Hand-in Instructions: You do not have to hand in any printed code. Create a README.txt file that includes your name, student number, and login ID for yourself, and any information you would like to pass on the marker. Create a folder called "assn4" under your "cs314" directory and put all the source fies, your makefle, and your README.txt file there. Do submit the images made by your program for the example scenes provided. If you design extra-credit scenes, also submit the .ray file for them. Also include any images that you used as texture maps. Do not use further sub-directories. The assignment should be handed in with the exact command:

```
handin cs314 assn4
```