CSC 473
Program 1 – file parsing and ray cast (planes and spheres)

Due Friday April 8th, at 11:55pm

Overview:
Throughout this quarter you will implement the basic functions of a distributed ray tracer. This software will be enhanced throughout the quarter, thus this initial assignment will serve as the base for following assignments. Since you will be adding and reusing your code, it is advised that you write your code in a clean, structured object-oriented fashion. All code must be written in C++.

Very loosely, there are four steps to your final ray tracer:

- Parsing the scene description file.
- Computing ray-object intersections
- Shading
- Recursive Tracing (reflection, refraction and shadows)
- Write out resulting image

This initial assignment if focused on the first two of these stages.

Goal for assignment 1: Write a ray caster that can intersect rays with spheres and planes from a pinhole camera model as specified in the povray file. You do not need to compute lighting, just color the pixels the pigment of the closest intersections with a sphere or plane. Your code will need to

- Parse a subset of the scene description file, specially, simple.pov which is linked on the website.
- Compute ray-sphere and ray-plane intersections
- Color pixels based the pigment of the closest intersection
- Write out resulting image

Software engineering considerations:
As the initial assignment for your ray tracer, you need to think about designing and implementing the abstract object(s) to represent geometry in your scene and write the parsing code to read in the scene description file (see below). In general, all geometric objects in your world will need to be able to be parsed, intersected (by rays), and shaded.

In addition, your ray tracer will need to support specific unit tests throughout the quarter. In general, this will require that for a specific ray (relative to the camera – i.e. listed in pixel space, e.g. [Xi, Yi]), can be tested – for example, having various values returned throughout the ray tracing process. This will include values such as the ray's point,and direction, distance to closest object, color of intersected object, and then later derived ray's from the original ray.

Scene Description Language

The scene description language that we will be using will be based (loosely) on a subset of the Povray format.
The big difference is that we will be using a right-handed coordinate system for our world, object and camera coordinates.

Your raytracer must be able to parse the following types:

- **// comments**
- **camera**
- **light_source**
- translate, scale, rotate
- box
- **sphere**
- **plane**
- triangle
- **pigment**
    - **color**
        - **rgb**
        - **rgbf**
- finish
    - ambient
    - diffuse
    - specular
    - roughness
    - reflection
    - refraction
    - ior

The bold elements are the types that MUST be parsed for this first assignment (you may ignore the zero translate at this time if you would like). You should create an abstract object from which all of the ray tracer objects will be derived. Each derived object should have its own parse function (read) that takes the filestream in, processes the data for that object, and returns the altered file pointer (for example).

Ray Object Intersections

After parsing the scene file and creating the necessary data structures, your program should begin casting rays. The camera object should, with knowledge of the output image size and a given pixel, be able to cast the necessary rays and return the appropriate rgb color value for the pixel. In the assignment you will only cast one ray per pixel (this will change in later assignments however).
The rays should be represented by an object. To cast a ray, simply traverse the scene object list (also represented by C++ objects) testing for intersection with each object.

Each derived geometry object should have its own intersection routine that takes a ray, performs an intersection test and returns the closest intersection (if one exists).
The closest intersection will be shaded using the model described in the next section. However, in the first pass of the algorithm you may wish to color every intersection a constant color to test for correct intersection.

Notes for the camera for this assignment
For this assignment, you may work under the assumption that the camera is positioned down the positive Z axis, looking down the negative Z axis (we will next implement a complete virtual camera). Note this is how the camera values are specified in the sample povray file. To compute the value of the rays, note that the limits of the near plane (i.e. the left, right, top and bottom) are defined by the camera up and left vectors. Assuming the 'eye' represents the center of projection of the camera, and the near plane is one unit in front of the camera, divide the world space defined by these extents by the number of pixels specified as input to the program (see 'program execution below') to generate sample points 'per' pixel in world space. Use this point to compute a vector for the ray's direction.

Program execution:
Your program should have the following syntax:
raytrace <width> <height> <input_filename>
where the options are:
width = the image width
height = the image height
input_filename = the name of the povray file to read and render
Thus:
raytrace 640 480 sample.pov
will render a 640x480 image file, "sample.tga" consisting of the scene defined in "sample.pov".
Image files should be output as tga files
Sample input files and images are given on the class webpage. Some of these pictures may be generated by Povray and will not look identical to your output (due to differences in the shading model, etc.). For later assignments, you will need to also submit rendered images. Please submit a README.txt file that contains a description of which parts of the ray tracer that you believe are working, partially working, and not implemented. This will assist the grader in determining what is causing potential errors in your output and help in assigning partial credit.

Grading breakdown:
40 points file parsing (evidenced via ray cast working)
30 points sphere intersections (working ray cast)
30 points plane intersection (working ray cast)