Assignment 4: Intersecting shapes

Written Homework Due: Tuesday, March 31 (9:30 AM)
Programming Project Due: Wednesday, April 1 (5:00 PM)

1 Introduction

In this assignment you will learn how to intersect shapes with rays. Like the previous shapes assignment, you will be performing your intersection in object space — *i.e.,* the object will be in the coordinate system defined in the first shapes assignment. Obviously, it will be useful in the future to be able to put the shape anywhere in the scene and still be able to use the same intersection code. How to do this? Well, if you apply a matrix M to an object to move it somewhere, you can always apply the matrix M^{-1} to get it back to its initial starting place. Suppose you're given a ray r and a shape S that has been moved by the matrix M, and you want to find the intersection using the intersection code you're about to write. How can you get your ray into S's object space? Think about this; we'll come back to it again in later assignments.

You will be using this shape intersect code later in the ray tracing assignment. In ray tracing, you will be generating rays from the camera through every pixel on the screen, which you will then intersect with all of the objects in the scene. These rays will spawn new rays that bounce off of intersected objects and possibly intersect with other objects.

Now down to the core of this assignment. For each of the shapes (cone, sphere, cube, cylinder) you will write an intersection routine that takes in a ray (which is a point p plus a vector d) and returns a list of t_i values, intersection points q_i, and surface normals n_i. The t_i values describe where along the ray the intersection occurred:

$$p + t_i d = q_i$$

The t_i values should be sorted in ascending order, and you should only include those intersections that are in front of the ray. You will not be using the surface normals in this assignment, however, you will need them in a later assignment.

Technically, you really only need to return the first intersection encountered. However, you will find it easier to debug if you find all intersections.

1.1 Implicit Equations

One of the real advantages of ray intersection is that you don't have to work with approximations to the objects in your scenes (i.e., the triangles from the previous assignment). When your objects are defined by an implicit equation, you can mathematically calculate the actual intersection values.

1.2 Shell support

The shell will allow you to display an object and a ray, plus the intersection points returned by your ray intersection code. You can move the ray around and change its orientation.

I have provided a `HitRecord` class that manages the storing and sorting of hits for you. The (u, v) coordinates are texture coordinates; you do not need to worry about them now, just set them to something

reasonable like $(0, 0)$. I also provide a routine that will draw all the hits using OpenGL. Most of the methods inside of this class should be self-explanatory.

You will need to make some changes in `IntersectionUI.cpp` to keep track of the current shape, draw it (as in the shapes assigment), perform the intersection and draw the intersection points (I have provided a method to do this for you). As in the shapes assignment, remember to use good object oriented design principles. For example, you probably will want all your shapes to have a virtual intersect method that returns a hit record given an input ray.

In addition to this, we have a test script that generates a set of rays and writes out the intersection results. The seed parameter is an offset into the random sequence used. When I grade your assignment, I will choose a specific seed, and compare the result the demo with yours. Make sure this works before handing in. You will need to edit the code in `IntersectionUI::writeTest` to actually perform the intersection against your four base shapes.

1.3 Debugging

I recommend explicitly generating all possible intersection types, then sorting them. Also, it's a lot easier to keep the t value and a boolean to indicate if the t value is valid, rather than trying to use a special t value to indicate no intersection (it also makes your code a lot cleaner to read).

1.4 Extra Credit

Try intersecting the cow, or other mesh. The cow is composed of a lot of triangles, so you may want to think about some of the optimizations you can do. Effective optimizations will win you extra points. You can also try adding extra implicitly-defined shapes. The torus, for example, is described by a quartic on x, y, and z. Feel free to add a shape to your shape classes.