CS1566 Assignment 5
Intersect

Algo Due: Wed 10/31 5pm
Asgn Due: Tue 11/5 11:59pm

1. Overview

In this assignment we will put to good use previously learned techniques (such as scenegraphs – flat as they may be; 3D tessellated objects; geometric transformations and camera viewing; and intersections), in order to create an (admittedly silly) first-person-shooter game. We will enable the user to walk through the scene and shoot (or pick, for the pacifists) objects in the scene. Basically, you will shoot out rays from the eye of a camera and calculate intersections with primitives in a scene using their implicit equations. These are the first steps in writing a full-fledged ray tracer. You'll also be on your way to better understanding collision detection. Oh, fun interactive games, here we come!

2. The Assignment

Once again you will use the spec files provided for the Camera Modeler assignment to load the scenes. Your program must be able to handle the following shapes:

- cubes (object type ID #1)
- house (object type ID #2)
- spheres (object type ID #3)
- cylinders (object type ID #4)
- cones (object type ID #5)

Optionally, you may include support for any additional shapes you may have done (crests, hourglasses, meshes, tori). This possibly can lead to extra credit points being assigned. However, please first add support for the elementary shapes and then worry about the additional shapes.

The user should also be able to click on the screen and pick the object under the mouse, and your program needs to indicate the exact location of the intersection on the object. At the very basic level, your program needs to handle intersecting spheres, cubes, cones and cylinders. Your intersection points need to be clipped against the finite shape (i.e., return intersections with the cube faces, not with the infinite planes to which the faces belong; similar for the finite cylinder), although we do not require you to detect intersections with the bottom and top caps of the cylinder or of the cone (you can do those for EC).

Start early. Start now. Start yesterday. This assignment will require you to work out some math and implement it as well; do not underestimate the workload required.

3. Approach

To encourage you to start working early, we broke the assignment into two parts. The first part is algorithmic and is due in a few days, in paper form, to the grad TA, to their mailbox or in their hand; it is worth 30% of your grade for this assignment, no late handins accepted. The second part is the actual program and is due in ten days, via electronic submission; the coding component is worth 70% of your grade for this assignment. Begin by reading and solving the Assignment 5 (part 1): Intersect Algorithm handout (available under Assignments on the course website). When done, please move on to coding.

Download the Intersect example code package (available under Assignments on the course website). It contains a glmain.c and a glmain.h file, a Readme.txt file, and several spec files. Make sure you can compile and run the support code. Read through the source code once, paying attention to comments. You might want to start by integrating in your Camera Modeler code.

For this assignment, you will also need to be able to maneuver the camera (P04). There's no need to use your own parallel orthographic camera; go ahead and use the OpenGL perspective one. It's probably worth trying to fix any surviving bugs related to camera motion from P04 first.

3.1. Ray-object Intersections

You will need to:

- **Generate a ray** (simplest case, just shoot a ray through the center of the clicked pixel) and display the ray as a vector
- **Find all the objects along the ray** and return the intersection point(s)
- **Indicate visually the intersection point(s)** by placing a marker (e.g., a really small sphere) at each intersection point.

For the first step, one way to generate a pick ray is to call gluUnProject() twice for the mouse location, first with *winz* of 0.0 (at the near plane), then with *winz* of 1.0 (at the far plane). Subtract the near plane call's results from the far plane call's results to obtain the XYZ direction vector of your ray. The ray origin is the view location, of course. You may use a different approach if you wish (e.g., for the direction, subtract the view location from the unprojected far plane point.) Either way, always remember to normalize the ray direction!

For the second step, you will need to use implicit equations. Basically, for each object type (cube, cylinder, sphere, cone, etc), you will need to define an Intersect function that takes as arguments the ray origin and direction and returns the first intersection point if the ray intersects the object, or some default value if the ray misses the object. This means that you will use math to solve exactly and elegantly where a ray intersects a cone etc. (solve a system of equations, compute the roots etc.), thus surpassing the precision and computational limitations of the polygon-based rendering pipeline.

For cube intersections, use the plane equations, and note that the point must be **inside** the faces of the cube. Please remember that such clipping operations are much easier to

compute in the cube local coordinate space. Proceed slowly and debug carefully! And keep in mind we **will** test your program on spec files that have rotated and scaled shapes, including the robot file.

Note: While OpenGL and GLU do have a basic implementation of a picking action for object-selection purposes, they do not support ray-object intersections. This is one of the reasons why we write our own. :-) (another reason might be achieving a mastery of alternative rendering pipelines, ray-tracing style; or understanding the basics of collision detection). The bottom line is: **To receive any credit for this part of the assignment, you need to implement ray-object intersections using the implicit representation of each shape. No per-triangle intersections, thank you very much**.

Finally, whenever a picking/shooting action takes place, you will need to generate the ray, then iterate over the objects in your object list/array and check whether the ray intersects any of them, and if so, where exactly. You need to show marker(s) at the intersection point(s) for each ray; you may remove the previous markers when processing the next ray. NOTE: please do not remove the objects as they get shot; we need a visual indication of where the intersection is, and for that we need to see the ray, the object, and the marker at the same time (two crossed line-segments, or a tiny sphere, or even a large point would do for the marker). For a bit of EC, you **could** modify the color/transparency of the shot object, but again, do not remove completely the shot object.

3.2. Finding Intersections in the Object Local Space

The easiest way to compute the intersection is in the object local coordinate space (alternatively, one could fudge the implicit equations to handle rotated and translated spheres, but such an approach would break down when handling cubes, cylinders, cones and other shapes; do not attempt). We strongly recommend you stick to the local-space approach, for your own good: you've been duly warned :-). This probably means you will have to, in this order:

1. transform the ray into the object coordinate space. We do this by applying the inverse of the current object transform to both the ray origin and to the ray direction (the object transform is the sequence of transforms you parsed from the spec file).
   Note 1: Keep in mind that computing the inverse of a geometric transform does not involve explicitly finding the inverse of a matrix; e.g., the inverse of a 30 deg rotation around *X* is simply a -30 degree rotation around *X*.
   Note 2: Please mind the transformation order. If the object is first scaled, rotated, and translated, in this order, then you will have to first (un)translate, then (un)rotate, then (un)scale your ray origin and direction. Also remember that in homogenous coordinates vectors (a.k.a. the ray direction) have a 0 as their 4th component, not a 1 as vertices do.
   Note 3: You've probably transformed points and vectors before, in P03, right? Not a biggie.
2. compute the intersection (first the *t* value, then the corresponding *x*, *y*, *z* coordinates)

3. transform the intersection point back into the world coordinate space (i.e., scale, rotate, translate the intersection point by the object transform).

For your sanity, please start by coding and testing the simplest case: intersections with a sphere located at the origin, with the camera at the default position given in the spec files. Do the same for a cylinder, then for a cone, then for a cube. For this simplest case, no transformation to the object space and back is required. Once you get this part working, move on to objects that have been transformed into space (e.g., a single transformed sphere, then a cylinder, then a cone etc). Finally, move on to handling multiple objects in the same scene (e.g., the robot spec file). Building and testing your code in stages will provide some satisfaction within minutes.

4. Keyboard Controls

Key:	**What it should do**
mouse left click	shoot and display ray and ray-intersections
camera controls from P04	Translate, pan, zoom etc the default OpenGL camera

Jumping, flying etc. will count as EC.

5. Grading

Grading is as follows:

Task:	Point Value:
Algorithm Turn-In	30
Camera Motion Controls	5
Correctly generated rays	10
Correct sphere intersection point (simplest case, at origin and with default camera is worth 5 points; the remaining 5 points are for intersections with transformed spheres, moving-camera intersections etc)	10
Correct cylinder intersection point (same as above)	10
Correct cone intersection point (same as above)	10
Correct cube intersection point (same as above)	15
Correct multiple-object intersection points	5
README turned in and quality of the code	5

Please note that although it is not explicitly listed, points will be taken off if an incomplete README file is submitted, if your program does not compile, if your code is messy (use comments if in doubt!), if your code is inefficient, or for any another miscellaneous reason.

6. Extra Credit

Bonus can also be attempted. Up to 20 points can be earned in this way. It is always recommended that everybody attempts to get a few bonus points. It is human nature to

make a mistake. By attempting bonus you can negate the loss of points from your mistake(s).

Here are just a few bonus ideas

- Make some fun spec files (a snowman? Any of the Google Doodle Muppets?)
- Make your shape distinguish between "wound shot" as opposed to a "kill shot".
- Shrink ray: Let the user shrink/expand the object she is looking at.
- Make your snowman or Muppet turn and look towards the mouse click location when the click falls outside the Muppet (not that hard; place your Muppet somewhere on the YOZ plane, and detect intersections between rays and the YOZ plane, then orient the Muppet towards the intersection point.)
- Make detecting exact object intersections *really fast* by using a smart space partitioning scheme (BSP, anyone?).

7. Example Code

- glmain.h - header file for the program
- glmain.c - main file for the program
- Readme.txt - text file containing your name, ID, description of Extra Credit work, and any additional comments.
- various example spec files

8. Handing In

To hand in this assignment, follow the Submit link on the course website and upload the following files:

- your modified source files (glmain.c(pp) and glmain.h, any additional h/c/cpp files)
- your Makefile (if any)
- filled in Readme.txt file (please save it as plain text)
- any interesting scene files you have created

Do NOT submit any executables. Use ftp only as an emergency backup plan (and notify the TAs immediately via email). Enjoy!