CS1566 Assignment 4
Camera Modeler

Algo Due:	Tue 10/16 5pm
Asgn Due:	Tue 10/23 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, in order to build and manipulate a complex scene consisting of several elementary shapes which may be manipulated arbitrarily. We will enable the user to walk through the scene via camera transformations. Oh, gorgeous scenery, here we come!

When rendering a three-dimensional scene, we need to go through several stages. One of the first steps is to take the objects in the scene and break them down into triangles (aka "shape tessellation"). The next step is to place those triangles in their proper position in the scene ("transform" them). In any given scene, not all objects will be located at the origin of the world coordinate system; transformations also provide some way of resizing, moving, and orienting the objects so that they are placed in the scene where they belong. (scene "modeling"). There remains but one important step: to define how the triangulated objects in the three-dimensional scene are displayed on the two-dimensional screen ("camera viewing"). This is accomplished through the use of a camera transformation, a matrix that we apply to a point in three-dimensional space to find its projection on a two-dimensional "film" plane. While it is possible to position everything in the scene so that all the camera matrix needs to do is flatten the scene[1], the camera transformation is usually significantly more flexible and versatile: it also allows us to position and orient the viewer in 3D space.

First, the program will read a description of the scene from a specification file. The description tells the program many things, including the types of various shapes, their properties, and camera settings (starting position, where it is looking, and which direction is up). In particular, each object will have certain properties needed for rendering (material color, for example), and a series of transformations that will move the vertices and normals from object space into world space for the specific object. We will store all this information into a list of objects.

Once we've parsed the file information, we will iterate through the list of objects, set up the appropriate information, and render each object to the screen. We'll be making good use of our code from Stitcher and Transformer. The main components of this assignment are, of course, parsing the file correctly in order to build the scene and manipulating the camera. For this assignment, you will also be implementing a virtual camera. It will be your job to implement almost all the adjustments that one could perform on a camera.

[1] Do this often enough and before long you would really wish you had decided to become a sailing instructor instead of studying computer science.

Keep your chin up! Writing a virtual camera is one of the easiest ways to start programming the GPU. Once the modeling and camera steps have been completed, you will possess all the tools needed to handle displaying three-dimensional objects oriented in any way and viewed from any position.

2. The Assignment

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)
- tori (object type ID #6)

Optionally, you may include support for any additional shapes you may have done (crests, hourglasses, meshes). 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.

For the camera part, your OpenGL program should have the following functionality:

- 'C'/'c' should switch to your own, parallel (orthographic) camera; you will need to maintain a world-to-film matrix that implements a parallel (orthographic) camera.

- Set your camera's absolute position/orientation given an eye point, look vector, and up vector (use the default values in the spec files).

- Translate the camera **in world space**.
  - When 'T'/'t' and 'X', respectively 'T'/'t' and 'x' are pressed together, the camera translates along the **world** X axis, in the positive, respectively negative direction (so the world translates in the opposite direction).
  - When 'T'/'t' and 'Y', respectively 'T'/'t' and 'y' are pressed together, the camera translates along the **world** Y axis, in the positive, respectively negative direction.
  - When 'T'/'t' and 'Z', respectively 'T'/'t' and 'z' are pressed together, the rendered object translates along the **world** Z axis, in the positive, respectively negative direction.

- Translate the camera **in its local space**.
  - When 'T'/'t' and 'U', respectively 'T'/'t' and 'u' are pressed together, the camera translates along its **local** u axis, in the positive, respectively negative direction (so the world translates to the left/right).
  - When 'T'/'t' and 'V', respectively 'T'/'t' and 'v' are pressed together, the camera translates along its **local** v axis, in the positive, respectively negative direction.
  - 'T'/'t' and 'W', respectively 'T'/'t' and 'w' pressed together, should move the camera closer/further to the point it is looking at (sort of a panning effect)

- Set the camera's height angle and aspect ratio.
  - When 'H/h' and '+'/'-' are pressed together, the height angle of the camera is increased, respectively, decreased.
  - When 'A/a' and '+'/'-' are pressed together, the aspect ratio of the camera is increased, respectively, decreased.

- Set the near and far clipping planes.
  - When 'N'/'n' and '+'/'-' are pressed together, the near plane should move closer to, respectively further from the camera
  - When 'F'/'f' and '+'/'-' are pressed together, the far plane should move closer to, respectively further from the camera

- 'L'/'l' and '+'/'-' should make the camera zoom in/out from the object (it's not quite the same thing as the panning above; in this case you'd have to modify the camera lens)

- 'P'/'p' should print out, at any point, the current eye point, look vector, up vector, height angle, aspect ratio, and world to film matrix.

3. An Example Specification File

What follows is an example specification file. It sets up one cube. It also sets up the camera. You will need to let the user decide[2] which specification file to parse and display. You will also need to parse the file they specify.

```
# object definition:
# rotation given in degrees
# object_type (tx ty tz 1) (sx sy sz 1) (rx ry rz 1) (r g b a)
# where object_type = 1 is a cube, 2 is a house etc etc
1 (2 0 10 1) (1 10 1 1) (0 0 0 1) (0.5 0.5 0.5 1)
#camera definitions:
#c (posx posy posz 1) (lookAtx lookAty lookAtz 1) (upx upy upz 1)
c (0 0 -20 1) (0 0 0 1) ( 0 1 0 1)
```

When reading the rotation/translation/scaling info from the spec file the order of operations should be interpreted like this: the object is first scaled in place at the origin, then rotated / oriented properly, and then translated to its final position. When specifying rotations, please assume that the numbers read in from the spec file are in degrees. Finally, (rx ry rz 1) means rotate first along the x-axis by rx degrees, then along the y-axis by ry degrees, and along the z-axis by rz degrees.

4. 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

[2] Ask the user at runtime (by reading from standard in?) or through command line parameters.

assignment. Begin by reading and solving the Assignment 4 (part 1): Camera Modeler Algorithm handout (available under Assignments on the course website). When done, please move on to coding.

Download the Camera Modeler 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 this code with your Transformer code.

You will need to use your C/C++ implementation of real translate, real scaling, and real rotate functions (developed in assignment 3) to automate the placement and manipulation of your shapes. You will have to generate the normals after the object has been transformed.

You will need to insert the World to Film matrix into the rendering pipeline yourselves by using a few OpenGL commands. It may make sense to write a putWorldToFilm() function for that. Please recall from class that OpenGL requires two separate transformation matrices in order to place objects in their correct locations. The first, the ModelView matrix, positions and orients your camera relative to the scene, or if you prefer, orients the world relative to your camera. The second, the Projection matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In Camera Modeler, this is your responsibility. More specifically, in your putWorldToFilm function (if you choose to write one), you must make the appropriate calls to configure the Projection Matrix. The function putWorldToFilm() is called frequently, so you should cache the matrix rather than recomputing it needlessly every time it is called. Here is some sample code to help you out…

```
// tell GL that future matrix operations should be applied to the projection matrix
glMatrixMode(GL_PROJECTION);

// give a pointer to an array of doubles containing the matrix to load
// the matrix pointed to will become the new projection matrix
glLoadMatrixd(pointer_to_first_element_of_double[16]_array);
```

For more information, look up glMatrixMode and glLoadMatrix in the Blue Book (linked from the course website). Please do not use any other library camera functions, we want to see your camera and math functions in action. Remember, GL_PROJECTION is only for the camera "lens" matrix. Other parts of an object's transformation do not belong here.

5. Keyboard Controls

Key:	**What it should do**
'C'/'c'	Switch to your own parallel (orthographic) camera
'T'/'t' and 'X'/'x'	The camera translates along the **world** X axis in the positive, respectively negative, direction

'T'/'t' and 'Y'/'y'	The camera translates along the **world** Y axis in the positive, respectively negative, direction
'T'/'t' and 'Z'/'z'	The camera translates along the **world** Z axis in the positive, respectively negative, direction
'T'/'t' and 'U'/'u'	The camera translates along its **local** u axis in the positive, respectively negative, direction
'T'/'t' and 'V'/'v'	The camera translates along its **local** v axis in the positive, respectively negative, direction
'T'/'t' and 'W'/'w'	The camera translates along its **local** w axis in the positive, respectively negative, direction
'H/h' and '+'/'-'	The height angle of the camera is increased, respectively, decreased
'A/a' and '+'/'-'	The aspect ratio of the camera is increased, respectively, decreased
'N'/'n' and '+'/'-'	The near plane moves closer to, respectively further from the camera
'F'/'f' and '+'/'-'	The far plane moves closer to, respectively further from the camera
'L'/'l' and '+'/'-'	The camera zoom in/out from the object
'P'/'p'	Prints out the current eye point, look vector, up vector, height angle, aspect ratio, and world to film matrix

6. Grading

Grading is as follows:

Task:	Point Value:
Algorithm Turn-In	30
Scene Parsed Correctly (aka, Things Display and Behave as Expected, including Normals)	10
T/t and X/x, Y/y, Z/z translate the camera along the **world** axes correctly	15
T/t and U/u, V/v, W/w translate the camera along the **local** axes correctly	15
H/h, A/a and +/- change the height angle and aspect ratio of the camera correctly	10
N/n, F/f and +/- move the near plane correctly	5
L/l and +/- displays correct zooming behavior	5
P/p prints out the current eye point, look vector, up vector, height angle, aspect ratio, and world to film matrix	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.

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

Here are just a few bonus ideas[3]

- Add support for additional shapes (moebius strip, etc)
- How about a 'jump' option, where the user/camera can jump in the air, or do a double jump, Ratchet and Clank style?
- Let the user use the mouse to "look" around the scene
- Add fog (look up glFog[f,i](...)) to create an eerie scene.
- Add collision detection so the user cannot walk through shapes (bounding boxes?)
- At the very least, make a neat spec file of your own (Snowman, anyone? Google Doodle Muppets?)
- Add more controls to your camera or implement a perspective camera, as well:
  - You can try rotating your camera around the world axes.
  - Try rotating your camera around its own local axes. (Hint: Rotation: How (mathematically) will you use the u, v, and w vectors, in conjunction with a rotation angle θ, to get new u, v, and w vectors when:
    - adjusting the "roll" in a clockwise direction by θ radians?
    - adjusting (rotating) the "pitch" to face upwards by θ radians?
    - -- adjusting the "yaw" to face right by θ radians?)
  - (This is fun) Make your camera bob up and down as the cameraman walks; you may want to use a sinus wave for the camera path.
  - Implement a perspective camera.

EC points are at the discretion of the grader, somewhere between 5 and 30 points are possible.

8. 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

A few words about pointers. It's kind of hard to add multiple shapes to our scenes without using pointers, so brace yourselves: in this assignment we'll use them. The cs1566 staff has gracefully provided a pointer infrastructure and a cube example to help those of us with little or no background in pointers. Please check out the Object array definition in the example code, the parse_obj and also the draw objects() functions, then emulate.

[3] Roughly ordered according to increasing difficulty

The Object array is filled with Object structs that will store many different objects in one array. Each Object is identified by its 'sid' (shape ID). The sid is read from the spec file and inserted into each Object as it is created. The Object structure may need to be modified to contain different storage methods for each shape, depending on how you are storing your vertex information, for example "GLfloat vertices cyl[50][50][4]; GLfloat normals cyl[50][50][4];" for a cylinder vertices and normals. Objects also contain information on material properties and all the transformations contained in the spec file, read using parse obj() function. Once your Object array is filled from the spec file, you use the Draw Objects() function to iterate over the array and call the drawing functions for each shape using the sid and a case statement.

9. FAQ

Many things may go wrong with this assignment. Here's a few tips:

```
1. Why are my houses tilted/skewed?
When reading from the rotation/translation/scaling from the spec file
the order of operations should be something like this

Scale(...)
Rotate(...)
Translate(...)

In other words, the object is first scaled in place at the origin, then
rotated / oriented properly, and then translated to its final position.

2. Why does a rotation by 1 do something really weird?
For sanity's sake, everybody just assume that the numbers read in from
the spec file are in degrees.

3. Nothing gets drawn, why?
Try drawing a Teapot. If you call glutSolidTeapot(size) and nothing
gets displayed, chances your camera is not pointing at the Teapot.
Either point the camera towards the origin (where the teapot is), or
move the teapot to wherever your camera is looking.

If you have no clue where your camera is or what it points at (?!),
increase the size of the teapot until some part of it shows in the
scene.

4. Nothing gets drawn, why (part 2)?
When debugging your program, it is often a good idea to use an uncommon
background color, like a random shade of pink, since black can arise as
a result of many different bugs. Who knows, maybe you're drawing black
on black.

5. Should I transform my normals?
Yes, if you are not using cross-products. If so, Recall that transforms
do not work on normals the way they work on points. If you scale the
object and try to apply the same scaling to its normals, your normals
will point in the wrong direction.
```

Hope this helps!

10. 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!