CMPUT 411/511—Computer Graphics

Assignment 1

Fall 2013
Department of Computing Science
University of Alberta

Due: 23:59:59 *Sunday, October 6*
Worth: 20% of final grade
Instructor: Dale Schuurmans, Ath409, x2-4806, dale@cs.ualberta.ca

---

In this assignment you will implement a 3D model viewer (in C++/OpenGL) that can read in a description of a 3D model represented as a polygonal mesh, display the model, orient and move the model, and orient and move a camera that is viewing the model.

Note When submitting your program, you need to include a `Makefile` that allows the TA to simply run "`make`" and have your program compile properly on the lab machines. The TA will then run the executable "`./modelviewer` ⟨filename.*obj*⟩" to test your program on the model stored in ⟨filename.obj⟩.

When finished, please submit a *single* .zip archive containing your files on **eclass**.

Note A set of .obj files containing examples of models is posted on the course website.

Implementing a 3D model viewer

You will implement a basic model viewer, with executable called `modelviewer`, that reads in a Wavefront .obj file describing a model represented as a polygonal mesh.

Marks are given for the functionalities achieved by your viewer.

1. (1%) **Read in a model from a .obj file**

   Your program needs to accept a single argument that names an input file to read. In particular, your program will be invoked with a command "`./modelviewer` ⟨filename.obj⟩", where ⟨filename.obj⟩ is the name of the file to be read.

   The input file ⟨filename.obj⟩ is assumed to be in .obj format, described on the Wikipedia page http://en.wikipedia.org/wiki/Wavefront_.obj_file. For this assignment you will only process the vertex and face lines, tagged by `v` and `f` respectively.

   Note Vertices should be loaded into a *vertex array*, and the faces should all be coded in a *display list* in the initialization portion of your program called from `main`. The drawing portion of your program (probably called something like *drawScene* and registered in `glutDisplayFunc`), should only invoke the display list.

2. (1%) **Write a model out an .obj file**

   It is very useful to have a simple output function for debugging and marking purposes. Therefore, when the user types the character 'w', your program should write to the output file `output.obj`, recording the list of vertices, one per line, each line tagged by `v`, followed by the list of faces, one per line, each line tagged by `f`. That is, your program should be able to write out a valid .obj file that represents the current model configuration.

3. (1%) **Center and scale the model, place in default position**

   Calculate the *center* of the model simply by computing the mean vertex location. Translate the vertices so that the center is at the origin. Then calculate the maximum and minimum x values, the maximum and minimum y values, and the maximum and minimum z values, respectively. Let $scale = \max\{x_{\max} - x_{\min}, y_{\max} - y_{\min}, z_{\max} - z_{\min}\}$. Divide the vertex coordinates by *scale* to normalize the size of the model. Finally, translate the model so that the center is moved to $(0, 0, -2)^\top$.

4. (3%) **Place camera in default position and orientation, display the model**

   Assume the camera is initially centered at the origin, looking in the $-z$ direction, with *camera up* in the y direction. Define the viewing region by placing the *front* viewing face at position $z = -1$ with x ranging between $[-1, +1]$ and y ranging between $[-1, +1]$. Place the back face of the viewing region at $z = -100$. (That is, $near = 1$ and $far = 100$ respectively.) Initially, use *orthographic* projection. (This is usually specified in the *resize* function registered in `glutReshapeFunc`.)

Display the *wireframe* image of the model. (This would usually be done in a *drawScene* function registered in `glutDisplayFunc`.) Make sure the background (clearing) color is set to black and the wireframe drawing color is set to white.

5. (1%) **Switch between orthographic and perspective projection**

   When the user types the character 'v' the viewing projection should be set to *orthographic*. When the user types 'V' the viewing projection should be set to *perspective* (with the same front viewing face as for orthographic projection).

6. (2%) **Translate the model**

   When the user types:
   ⟨leftarrow⟩ or ⟨rightarrow⟩: translate the model -0.1 or 0.1 units along the x axis, respectively,
   ⟨downarrow⟩ or ⟨uparrow⟩: translate the model -0.1 or 0.1 units along the y axis, respectively,
   'n' or 'N': translate the model -0.1 or 0.1 units along the z axis, respectively.

   Redisplay after each keystroke.

7. (3%) **Rotate the model (about its center)**

   When the user types:
   'p' or 'P' (pitch): rotate the model -10 or 10 degrees counterclockwise around the x axis (centered at the model's *center*), respectively
   'y' or 'Y' (yaw): rotate the model -10 or 10 degrees counterclockwise around the y axis (centered at the model's *center*), respectively
   'r' or 'R' (roll): rotate the model -10 or 10 degrees counterclockwise around the z axis (centered at the model's *center*), respectively

   Redisplay after each keystroke.

8. (2%) **Translate the camera**

   When the user types:
   'd' or 'D' (dolly): translate the camera -0.1 or 0.1 units along the x axis, respectively,
   'c' or 'C' (crane): translate the camera -0.1 or 0.1 units along the y axis, respectively,
   'i' or 'I' (zoom): translate the camera -0.1 or 0.1 units along the z axis, respectively.

   Redisplay after each keystroke.

9. (3%) **Pivot the camera (about its origin)**

   When the user types:
   't' or 'T' (tilt): rotate the camera -10 or 10 degrees counterclockwise around its x axis (centered at the camera's origin), respectively,
   'a' or 'A' (pan): rotate the camera -10 or 10 degrees counterclockwise around its y axis (centered at the camera's origin), respectively,

'l' or 'L' (roll): rotate the camera -10 or 10 degrees counterclockwise around its z axis (centered at the camera's origin), respectively,

Redisplay after each keystroke.

10. (1%) **Reset or quit**

    When the user types:
    'x': reset the model and camera to their default position and orientation
    'q': exit the program

11. (1%) **Distance fading**

    Wireframe models are hard to visualize if they get too complicated. Visualization can be aided by a "fog effect" where more distant points and vertices are progressively faded. For this part, you will add a fog effect that the user can turn on and off.

    When the user types 'f' there should be no fog effect, but when the user types 'F' the fog effect should be turned on. Use the LINEAR fog model described on P.502 of the course textbook. Set FOG_START to 1 and FOG_END to 5. Set FOG_COLOR to the background (clearing) color. (Make sure you turn depth testing on.)

12. (1%) **Use double buffering to improve animation quality**

    Displaying changing objects or views (i.e. animation) looks much better when the user does not see the objects being drawn. (This is especially true when the models are large and take a non-trivial amount of time to draw.) To cope with this problem, OpenGL provides a double buffering model where the drawing can occur in a drawable back buffer, after which it can be swapped with a viewable front buffer—preventing the user from seeing any of the drawing operations being executed.

    Improve the quality of your viewer by enabling double buffering (for example, as explained in Section 4.5 of the textbook).