6.837 Computer Graphics Fall 2012
Programming Assignment 0: OpenGL Mesh Viewer

1 Getting Started

Let's start off by looking at `main.cpp`. It contains a fully functional application that displays a teapot. Other than that, it's not very interesting. To compile this application, type `make` on an Athena Linux station. This should produce an executable called `a0`. If there are error messages, contact the TAs.

Once you've successfully built the executable, run it by typing "`./a0`" at the terminal. It should display a teapot. Yes, that's all it does. It's now your job to make this application a bit more interesting by modifying the code.

2 Requirements

2.1 Color Changes

Add the ability to change the color of the displayed model. Right now, the color is set to [0.5, 0.5,0.9] (RGB), which is a boring light blue. Your task is to wire the `c` key to toggle through several other colors (feel free to choose which colors you want). How do you handle keyboard events? Notice that, when you press keys while the application is running, the console says something like this:

```
Unhandled key press h.
Unhandled key press e.
Unhandled key press l.
Unhandled key press l.
Unhandled key press o.
```

The code that prints these messages is in the `keyboardFunc` function. Modify the code to handle the `c` key appropriately. A reasonable way to do this might be to have the `c` key increment some sort of global counter variable and then use that variable to select a color in the `drawScene` function. Note that GLUT will not immediately redraw the scene after it has called

`keyboardFunc`. The end of the function contains a call, `glutPostRedisplay()` that updates the display.

2.2 Light Position Changes

Add the ability to change the position of the light. In the code, the light is placed at [1.0,1.0,5.0]. Wire the arrow keys to change the position of the light. More specifically, the left/right arrow keys should decrement/increment the first value of the position by 0.5, and the up/down arrow keys should do the same for the second value. This can be done quite similarly to the suggested method for the previous requirement.

2.3 Mesh Loading and Display

Once you have completed the above requirements, we can move on to the tough part: loading new objects. In the sample code, we have provided several 3D meshes in OBJ format. It is your job to write teh code to load and display these files. OBJ files are a fairly standard format that can describe all sorts of shapes, and you'll be handling a subset of their functionality.

Let's look at `sphere.obj`. It's a big file, but it can be summarized as follows:

```
#This file uses ...
 ...
v 0.148778 -0.987688 -0.048341
v 0.126558 -0.987688 -0.091950
 ...
vn 0.252280 -0.951063 -0.178420
vn 0.295068 -0.951063 -0.091728
...
f 22/23/1 21/22/2 2/2/3
f 1/1/4 2/2/3 21/22/2
...
```

Each line of this file starts with a token followed by some arguments. The lines that start with v define *vertices*, the lines that start with `vn` define *normals*, and the lines that start with `f` define *faces*. There are other types of lines, and your code should ignore these.

Your first task is to read in all of the vertices ("v") into an array (`vecv`) (or any other data structure that allows you to quickly reference the ith element). Then, do the same for the normals ("`vn`), loading them into another array (`vecn`).

Understanding the faces ("`f`") is a little more difficult. Each face is defined using nine numbers in the following format: $a/b/c\ d/e/f\ g/h/i$. This defines a face with three vertices with indices a,d, g and respective normals c, f, and i (you can ignore b, e, and h for this assignment). The general OBJ format allows faces with an arbitrary number of vertices; you'll just have to handle triangles.

So let's say you have the vertices and normals stored in `vecv` and `vecn`. Then you'd draw the aforementioned triangle using the following code:

```
glBegin(GL_TRIANGLES);
glNormal3d(vecn[c-1][0], vecn[c-1][1], vecn[c-1][2]);
glVertex3d(vecv[a-1][0], vecv[a-1][1], vecv[a-1][2]);
glNormal3d(vecn[f-1][0], vecn[f-1][1], vecn[f-1][2]);
glVertex3d(vecv[d-1][0], vecv[d-1][1], vecv[d-1][2]);
glNormal3d(vecn[i-1][0], vecn[i-1][1], vecn[i-1][2]);
glVertex3d(vecv[g-1][0], vecv[g-1][1], vecv[g-1][2]);
glEnd();
```

You may be wondering why there are all those minus-ones. It's because the faces index vertices and normals from 1, and C/C++ indexes from 0. If you have this implemented, the rest is fairly straightforward: you just have to loop over all teh faces to draw the complete mesh.

In `main.cpp`, vecv is defined as an STL vector of Vectof3fs. An STL vector is simply a list of arbitrary objects. In this case, it is a list of Vector3f objects.

```
vector<Vector3f> vecv;
```

To add a new entry to this array, use push_back:

```
vecv.push_back(Vector3f(0,0,0));
```

There are several ways to iterate over an STL vector. Here's an example of using indices (if you're interested in learning more about STL, check out the documentation at http://www.sgi.com/tech/stl/):

```
for(unsigned int i=0; i < vecv.size(); i++) {
   Vec3d &v = vecv[i];
   //do something with v[0], v[1], v[2]
}
```

Please also keep in mind that you'll need another array to store the faces (perhaps `vecf`). It may be tempting to try to draw them as they are read from the OBJ, but OpenGL requires you to redraw the model whenever the window is obstructed or resized (and also when you change the color or lightning).

Your final executable should take the OBJ files via standard input:
`./a0 < sphere.obj`
The "<" operator will put the contents of sphere.obj into the "standard input" stream. This stream can be accessed using the cin object. For example, to read a single line of data from the stream (all characters up to the next newline):

```
char buffer[MAX_BUFFER_SIZE];
cin.getline(buffer, MAX_BUFFER_SIZE);
```

cin.getline will return zero at the end of the file. You can use this fact to step through each line in the file. Once you get have an array of characters (the text from a single line of the file), you can parse it using a stringstream object. Create a stringstream object from an array of chacters (buffer) as follows:
`stringstream ss(buffer);`

Now that you have a stringstream object, you can read tokens (separated by spaces) from the buffer in order by using the ">>" operator. For example, given the input string "v 1.0 1.1 1.2", in the following code:

```
Vector3f v;
string s;
```

```
ss >> s;
ss >> v[0] >> v[1] >> v[2];
```

will put the value "v" into s, and load the values 1.0, 1.1, and 1.2 into v[0], v[1], and v[2]. Note that you can compare the string objects to constant strings using the regular "==" operator.

```
if (s== "v") {
      //do something
}
```

Make sure that you're able to load and view the three provided files without crashing; these are the only three files we'll test your program on.

You may want to run the provided sample solution `a0soln` to get an idea of how your application should work (run `.a0soln < garg.obj` and read the console output for usage instructions).

3 Extra Credit

Here are some ideas (sorted roughly by increasing level of difficulty) that might spice up your project. The amount of extra credit given will depend on the difficultly of the task and the quality of your implementation. In addition, feel free to suggest your own extra credit ideas! Just because it's not on this list doesn't mean we won't give you some extra points (although if it's a big addition, make sure you run it by the course staff first just to make sure).

Easy

- The sample solution (`a0soln`) lets you hit `r` to spin the model. Implement this functionality in your code (look up `glutTimerFunc`).
- Display the model using OpenGL display lists or vertex buffer objects for higher performance rendering.
- Modify the code so that the `c` key smoothly transitions between different colors (rather than just toggling it).

Medium

- Implement a mouse-based camera control to allow the user to rotate and zoom in on the object. Credit will vary depending on the quality of the implementation.

Hard

- Large meshes are quite difficult to draw and process. For interactive applications, such as video games, it's often desirable to simplify meshes as much as possible without sacrificing too much quality. Implement a mesh simplification method, such as the one described in Surface Simplification Using Quadric Error Metrics (Garland and Heckbert, SIGGRAPH 97).

4 Submission

Zip your VStudio project folder (the entire thing) or Xcode project folder, together with a README file that tells us:
(a) What kind of arguments (if any) that your program needs
(b) What components of extra project that's been done
(c) Were there any references that you found particularly helpful in completing this assigment? If yes, list them
(d) Are there any known bugs for your code? If yes, provide a list, and if possible describe what do you think caused it. This is very important as we can give you partial credit if you help us understand your code better.

Upload your zipped assignment to e-dimension