CPSC 314
Assignment 3: MyOpenGL Viewer

Due 4PM, Nov 9, 2012

In this assignment you will create a basic 3D geometry viewer using your own implementation of OpenGL. The goal of the assignment is to test your knowledge of the rendering pipeline. You will implement your own version of the geometric transformations involved in the graphics pipeline, clipping, scan-conversion, and z-buffer. You can earn bonus points for implementing other features such as lighting, smooth shading, etc....

Use the template code given online as a starting point for your code. You will not be using any OpenGL functions – all image pixels can be set using the `setPixel(x,y,r,g,b)` function call. The functions you will be implementing are mostly replacements for the equivalent OpenGL functions. For example, `myBegin()` can be thought of as a replacement for `glBegin()`.

The file `linalg.cpp` provides basic vector and matrix classes, along with methods to operate on them such as cross-products and matrix-vector multiplications. See `linalg.hpp` for a listing of the methods and some examples.

The following is a suggested order for implementing and testing your code. The code can be tested either via the list of scenarios detailed in the README or by loading a real 3D model (using scenario 'H' and a command line argument to your program specifying the name of the 'obj' format file to load the model from). The template contains a few example obj files. You can find more online, however note that some files can be of dubious quality. Your OpenGL implementation is likely to be relatively slow, so don't try to render very large models. **See the README for a full listing of key bindings and camera controls.**

After completing each part, ensure that the prior parts still work. The markers will be testing your code by running scenarios A through I without restarting your program. For the required parts of the assignment you should only make changes to the file `mygl.cpp`. A reference solution Linux executable `a3_sol` is provided for comparison. The reference solution implements all the required components as well as the optional polygon and texturing components. Note that it does not implement lighting.

(a) (15 points) Implement `myBegin()`, `myColor()`, `myVertex()`, and `myEnd()` functions. You will be testing these functions using **scenario A**, which uses these functions to draw a few points on the screen. The template code runs scenario A when "a" is typed on the keyboard. You will find it useful to implement a vertex data structure and create an array of these to hold all required information about vertices. In order

to make things simple, you may assume that there are never more than 50 vertices specified between a `glBegin()` and a `glEnd()`. Implement myVertex() so that it stores the untransformed coordinates as well as the current color. In your `myBegin()` function, you will need to remember the current type of primitive being drawn. Your code only needs to handle `GL_LINES` and `GL_TRIANGLES` (you may also add support for `GL_POLYGON` as an optional exercise at the end of the assignment). Your `myEnd()` function should call other functions of your own creation to transform all the points in the vertex list to viewport coordinates and then draw them as lines or triangles depending on the mode.

To help you debug your code, you should start by just rendering each vertex as a single point on the screen. Do this when the global variable `drawAsPoints` is set to `true` (this will allow you to test your implementation even without scan conversion fully implemented). The `drawAsPoints` flag can be toggled while running the program by pressing the "p" key. Test your code with scenario A. This sets the Model/View and Projection matrices to be identity matrices, and so object coordinates will effectively be the same as the normalized-device coordinates.

(b) (10 points) Implement `myTranslate()`, `myRotate()`, and `myScale()` functions. Test this with Scenario B.

(c) (10 points) Implement the `myLookAt()` and `myFrustum()` functions, which will alter the Model/View and Projection matrices, respectively. Test this with Scenario C. You should now also be able to use the mouse to move around in the scene and display models loaded from an input file (scenario 'H'). Note that depending on the view some points may be outside the viewport, hence you now need to implement clipping in the point drawing function to prevent your code from "drawing" them into a random location in memory.

(d) (10 points) Implement line scan conversion using solid shading and no Z-buffer. This code should be called when `GL_LINES` mode is set and the variable `drawAsPoints` is `false`. Use the color assigned to the first vertex as the color for the line segment. Use the basic line-drawing algorithm shown in class to fill in the appropriate pixels while moving between the line's endpoints. Be careful to handle lines with both positive and negative slope, as well as lines with slopes greater than 1 (or less than -1). Furthermore, the two endpoints of a line may be encountered in any order (ie: you will not always see the left-most vertex before the right-most vertex). Test this with Scenario A with `drawAsPoints` set to `false` (remember to press "p" to toggle this flag).

(e) (15 points) Implement triangle scan conversion using solid shading and no Z-buffer. The scan conversion should be called when the `GL_TRIANGLES` mode is set and the variable `drawAsPoints` is `false`. As with lines, the color assigned to the first vertex as being the color used for the triangle. Begin by computing the bounding box and making sure that it scan-converts correctly. Then make use of the implicit line equations of the triangle to only set those pixels that are interior to the triangle. Note that the

bounding box should be correctly clipped to the window before scan conversion. Test this with Scenario D. Run the previous scenarios to test for bugs.

(f) (15 points) Implement smooth shading by linearly interpolating the colors for each pixel from the colors given for the vertices. For triangles, do this by computing barycentric coordinates for each rendered pixel. Test this with Scenario E. When loading real models you should now get a smooth shading effect.

(g) (10 points) Implement a Z-buffer by interpolating the Z-values at the vertices and by doing a Z-buffer test before setting each pixel. Test this with Scenario F. Note that a Z-buffer has already been declared for you in the template code, and it is cleared for you every time a redraw is started.

(h) (15 points) To get the remaining marks you can implement one of the following options (or alternatively one of the options in (i)).

- Implement support for `GL_POLYGON`. This mode should draw a polygon with an arbitrary number of vertices. You may assume that the polygon passed into your rendering pipeline will always be convex. You will need to decompose the given polygon into a set of triangles, and then use your existing triangle drawing code to draw them. Test this with Scenario I.
- Implement basic lighting using only diffuse materials. Specify an ambient light source and a directional light source and compute the per-triangle (flat) shading based on those. You will need to implement the `myNormal` function as well. Note that while normals are transformed differently than positions for many transformations, you do not need to implement this for scenario 'H'; evaluate your lighting in world coordinates. To highlight the effect of the lighting you may want to override the color provided for the models loaded in scenario 'H' with a solid color, and by changing `Vertex::draw()` in object.cpp to not call myColor() for every vertex.

(i) (Bonus) If you implement all the required options till now you will get full marks for the assignment. To get extra marks you can implement one or more of the following options:

- Implement texture mapping by scan-converting the texture coordinates. Note that the perspective-correct scan-conversion of texture coordinates is slightly more complex than simply using the barycentric coordinates to produce a weighted combination of the vertex texture coordinates. When the `perspectiveCorrectTextures` boolean flag is true then perspective-correct texture coordinate interpolation should be applied. Otherwise, linear texture coordinate interpolation should be applied. The `Image *currentTexture` has a `lookup(u, v)` method, which you can use to get back the RGB color of point (u, v), where $u, v \in [0, 1]$ in the image that serves as the current texture map. Test this with Scenario G.

- Implement more complex lighting. For example, add point light sources with falloff, or implement the Gouraud or Phong illumination model.
- Implement the maximally efficient versions of all the algorithms above, as described in class. To showcase this option add a 'fast' toggle to your code allowing to switch between the basic and fast versions, highlighting the difference in speed. You may also consider adding a timer which will evaluate the rendering time done using both options.

Note that you should modify Scenario J to demonstrate any features that you add that are not demonstrated by one of the provided Scenarios.

Hand-in Instructions

You do not have to hand in any printed code. Create a README.txt file that includes your name, student number, and login ID for yourself, and any information you would like to pass on the marker. Create a folder called "assn3" under your cs314 directory and put all the source files, your makefile, and your README.txt file there. Also include any images that are used as texture maps. Do not use further sub-directories. The assignment should be handed in with the exact command:

```
handin cs314 assn3
```