CPSC 314
Assignment 3: MyOpenGL Viewer

Due 4PM, Nov 1, 2013

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, line rendering, texture-mapping, 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 must be set using the `setPixel(x,y,r,g,b)` function call, that draws one pixel in the provided "virtual screen". 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 G** 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`. Reference solution executables are provided for comparison (`a3_sol` for Linux, and `a3_sol.exe` for Windows). The reference solution implements all the required components as well as the optional polygon, basic lighting, wireframe and texturing components.

(a) (20 points) Implement the `myBegin()`, `myColor()`, `myVertex()`, and `myEnd()` functions. For this question, you are only required to draw the vertices as *points* on the screen (no scan-conversion). To help you debug your code, this behaviour must work when `myBegin(GL_POINTS)` is used, but also when `myBegin(GL_TRIANGLE)` is used and `currentPolygonMode == GL_POINT` (this will allow you to test your implementation even without scan conversion fully implemented). You will be testing these functions using **scenario A** (type "a" on the keyboard), which call them to draw a few points on the screen. 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.

You will find it useful to define your own data structure `struct Vertex {/*todo*/};`, and create a vector of these `std::vector<Vertex> vertices;` (see page 5 for more info) to hold all required information about vertices. 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 `myEnd()` function should call other functions of your own creation to transform all the points in the vertex list to viewport coordinates and then to draw them as points.

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

(c) (15 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 G**). 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) (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 `currentPolygonMode == GL_FILL`. Use 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.

(e) (15 points) Implement smooth shading by linearly interpolating the colors for each pixel from the colors given for the vertices. 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.

(f) (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.

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

- Implement support for `GL_POLYGON`, that you will test using **scenario H**. 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.
- Implement basic lighting using only diffuse materials, that you can toggle using `myEnableLighting()` and `myDisableLighting()`. You will test it using **scenario I**. 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 I**: evaluate your lighting in world coordinates.

  To highlight the effect of the lighting, you may ignore the existing color of vertices and use a solid predefined color instead.
- Implement a wire-frame mode where when displaying a triangle you draw its edges only, using line scan conversion, that you will test it using **scenario J**.

  This must be done when the variable `currentPolygonMode == GL_LINE`.

(h) (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 of the following options:

- 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.
- Implement more complex lighting. For example, add point light sources with falloff, or implement the Gouraud or Phong illumination model.
- Implement texture mapping by scan-converting the texture coordinates, that you will test it using **scenario K**. This has not yet been covered in class, so it is really not expected that any of you implement it. However, it may be left as a good exercise for after the due date. 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.

Hand-in Instructions

- Create a folder called `assn3` under your cs314 directory and put all the source files and the `README.txt` file there. Also include any images that are used as texture maps.
- In the `README.txt` file, please describe what functionalities you have implemented, as well as any kind of information you would like to give us for getting credit for partial implementation. If you dont complete all the requirements, please state clearly what you have tried, what problems you are having and what you think might be promising solutions. If you are using external sources, provide clear attribution.
- The assignment should be handed in with the exact command:

  ```
  handin cs314 assn3
  ```

  This will handin your entire assn3 directory tree by making a copy of your assn3 directory, and deleting all subdirectories! (If you want to know more about this handin command, use: `man handin`)

Assignment grading

The assignment will be graded with face-to-face demos: you will demo your program for the TA. If your assignment is incomplete, you can concisely summarize your explanations and what you were trying to do in your README. You can also explain any extra credit features you implemented.

- We will circulate a sign-up sheet for demo slots in class. Each slot will be 7 minutes. The demo sessions times will be determined closer to submission date.
- You must ensure that your program compiles and runs on whatever medium you intend to demo on (laptop, lab computer, etc.). The face to face grading time slots are short, you will not have time to do any quick fixes ! If your code as handed in does not run during the grading session, you will fail the assignment.
- The code that you demo must match exactly what you submitted electronically: you will show the TA a long listing of the files that youre using, so that he can quickly verify that the file timestamps are before the submission deadline.
- Arrive at ICICS/CS 005 at least 10 minutes before your scheduled session. Double-check that your code compiles and runs properly.
- When the TA comes to your computer you will run your assignment and go through the different provided scenarios. If you are shooting for the bonus points, show the grader the extra features and/or scenarios you created.

Very quick intro to std::vector

The C++ Standard Library (built-in with any C++ compiler) provides `std::vector`, a very useful dynamic array of variable size. Here is how to use them (more info on Google):

```
// First, don't forget to include the header
#include <vector>

int main()
{
    // Declare an empty vector of 'int'.
    // You can use any struct/class instead of 'int' (e.g., 'Vertex')
    std::vector<int> v;

    // Add elements to the vector
    v.push_back(5);                // now the vector is [ 5 ]
    v.push_back(3);                // now the vector is [ 5, 3 ]

    // Access and/or modify elements of the vector
    v[1] = v[0] * 2;               // now the vector is [ 5, 10 ]

    // Iterate through all elements of the vector
    for(int i=0; i<v.size(); ++i)
    {
        v[i]++;                    // increments the i-th element
    }
                                   // now (end of loop) the vector is [ 6, 11 ]
    // Reset/empty the vector
    v.clear();                     // now the vector is [ ]
}
```