CSC 433 / 533: Spring 2015
Assignment 2

Assigned: Wednesday, Feb 25th
Due: 11:59pm Wednesday, Mar 11th

For this assignment you will write a geometry viewer which is an interactive program that reads geometry descriptions from files and uses OpenGL to render the objects in 3D. Programs such as these are also called mesh viewers. Your program will be graded on the linux workstations on the 9th floor of Gould-Simpson building.

For all programs this semester:

1. Provide a Makefile that builds the program(s). We will not grade programs that do not compile.
2. Your directory should be self-contained and not need any files from other directories. So, it will contain all source code for both the application on the CPU and shader files for the GPU, and it will include any required data files.
3. Report any errors to stderr and provide a usage message when the program is invoked with no parameters.
4. Include a readme.txt file and include any special instructions or assumptions.

Task:

1. viewer
    1. In your makefile create an executable named viewer.
    2. Input to the viewer:
        3. Usage statement for *viewer*

           -c controlFile
        4. The control file is text file with keyword-value lines
            - i. obj <name>
            - ii. rx <angle> ry <angle> rz <angle> (angle values are in degrees)
            - iii. t <x-amount> <y-amount> <z-amount>
            - iv. s <x-factor> <y-factor> <z-factor>
            - v. Apply the transformations in the order they are listed in the file
            - vi. There are example control files below.
    5. Read and display geometry described in the wavefront obj format. The geometry information is in *.obj files and the color information is in .mtl files. Some URLs for this information are (http://www.martinreddy.net/gfx/3d/OBJ.spec , http://www.fileformat.info/format/wavefrontobj/egff.htm , http://www.fileformat.info/format/material/ and http://paulbourke.net/dataformats/mtl/ . This is not a complete list. You need to be able to process only the ASCII format of an obj file and you only need to process polygonal descriptions.
        - i. In an obj file, the name of the material library, is a file name that is in the same directory as the obj file.

ii. Note: the indices used to describe vertex connectivity in the obj file start with 1 not 0.

iii. You can ignore many pieces of information in the obj file. You only need to process the following keywords of the obj file

1. mtllib and usemtl used to describe the color
2. v, vt and vn for vertex information
3. f for vertex connectivity that describes a triangle (or face)

iv. n-sided polygons. In this assignment, I'm only asking you to deal with obj files that describe objects with triangles. You will find many obj files that have more than three vertices in a face description. For those that are interested, I have read that blender, a free 3D modeling program, offers a triangulate feature that could be used to read an obj file with n-sided polygons and create an obj file with only triangles. This is **not** a required part of the assignment.

v. For an obj file that doesn't provide normals, compute face normals, by taking the cross product of two vectors along the sides of the triangle, and use that normal for all three vertices of a face

vi. There are many .obj files on the web, here is one repository: http://people.sc.fsu.edu/~jburkardt/data/obj/obj.html . Respect people's requests about using their geometry files.

6. To compute color for an object, you will be using a basic per-vertex lighting model implemented in your vertex shader. Use the following equation Ka + (**N dot L**)*Kd, applied to the red, green, and blue components individually. So the only property values from the material file needed for this project will be Ka and Kd.

   i. The default Ka (ambient color) is .1 .1 .1 and the default Kd (diffuse color) is .9 .9 .9 **N dot L** is the vector dot product of **N**, a **unit** length surface normal at the vertex and **L**, a **unit** length vector pointing toward the light. Use the dot function in GLSL.

   ii. To ensure these normals are unit length, use the normalize function in your shader. This simplified lighting calculation is to be done in the vertex shader, and the **L** vector will be static in the application.

   iii. Ensure that **N dot L** is between 0 and 1, which you can do with the GLSL function max.

   iv. The output of your vertex shader will be position and color, where the position has been processed by viewing transformations and the color is set as described above.

   v. These lighting calculations are to be done in world coordinates. It is also common to do lighting calculations in eye coordinates.

   vi. Remember to transform the normals by the modeling transform. You do not need to compute the complete solution to the normal transform matrix. As long as you do uniform scaling, you can use the upper 3x3 portion of the modeling matrix to transform normals. Remember, you do not want to translate the normal vectors. In GLSL, cast a mat4 to a mat3 to get the upper 3x3 matrix.

7. Specification and control of 3D viewing parameters.
    i. Height in world coordinates is the z axis.
    ii. default viewing parameters:
        1. camera location – x coordinate is along the +X axis and three times the largest dimension of all the objects, y coordinate is along the +Y axis and three times the largest dimension of all the objects, and z coordinate is along +Z axis and one time the largest dimension of all the objects
        2. focal point – is at the geometric center of all the object descriptions
        3. view up - +Z axis
    iii. use a perspective view
8. keyboard commands
    i. ESC or q – quit application
    ii. r – reset view to default values
    iii. w – wireframe rendering
    iv. s – solid surface rendering
    v. camera movement
        1. ←→ - rotate the eye position and focal point around the z axis, ← 1 degree clockwise, and → 1 degree counter-clockwise
        2. ↑↓ - move camera forward with ↑ and back with ↓. The motion is along the gaze vector (focal_point – eye_point). Move both eye and focal point by the same amounts. Use the bounds of the scene to scale the distance for each key press.
    vi. camera movement only
        1. c,v – move the camera along the view up vector. c moves the camera down (opposite direction of view up) and v moves the camera up (along direction of view up). Use the bounds of the scene to scale the distance for each key press.
    vii. Focal point movement only
        1. d,f – move the focal point along the view up vector. d moves the focal point in the opposite direction of the view up vector, and f moves the focal point in the direction of view up.
    viii. view up vector movement
        1. z - rotate view up vector by 1 degree counter-clockwise around gaze vector
        2. x – rotate view up vector by 1 degree clockwise around gaze vector
9. GLUT changes
    i. Add GLUT callback, glutReshapeFunc. This will allow the user to resize the OpenGL window. You'll want to keep track of the height and width of the window and call glViewport.
    ii. Add GLUT callback glutSpecialFunc to handle the arrow keys.

2. Overview of program

1. The overall, general framework of your program1 can likely be reused. This is one approach to completing this task, and you are not required to follow these steps.

2. One shader program is enough, and the fragment shader will be the simple approach of passing the input color on to the pipeline. The vertex shader will be much different as it will need to perform a modeling transformation, a viewing transformation, a perspective projection transformation and calculation of color. Recommendations:
    i. Use several uniform variables, for example three mat4 variables for the transformation matrices.
    ii. Input variables will include vertex position, vertex normal, and color in two parts: ambient color (r,g,b) and diffuse color (r,g,b)
3. To set up depth buffering (z buffering).
    i. Recommended: you modify the call to glutInitDisplayMode to add a depth buffer also called a z buffer. Use GLUT_RGBA | GLUT_DEPTH to set up the z buffer.
    ii. Mandatory: enable depth testing with glEnable(GL_DEPTH_TEST).
    iii. Recommended: enable depth mask (or writing to the depth buffer) with glDepthMask (GL_TRUE).
4. The display callback function needs many changes
    i. Provide for two display buffers, the color buffer as existed in program1 but add the depth buffer, used for hidden surface removal.
        1. Use glClear to clear both the color buffer and the depth buffer with glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ii. For each geometric object (mesh) set up the appropriate viewing matrices and draw.
    iii. The way vertices are indexed in obj files doesn't map easily to OpenGL indexed drawing, and you are not required to use indexed drawing.
5. Add a reshape method, set up with glutReshapeFunc, to support resizing the output window.
6. Reading an obj file and putting the vertex information into OpenGL buffers is a demanding part of this project. The obj format is complex but many of the capabilities you can ignore, for example defining a curved surface. There are some simple obj files provided for you to use during development. There are many obj files available on the internet, and you are welcome to use them. We will use a few obj files not provided for testing your program, but we will not use unusual obj files in an effort to break your program.
7. Setting up the viewing parameters is actually just a few function calls to set up a modeling matrix, a viewing transform, and a perspective projection. If you don't use C++ and the header file vmath.h, you may have to write a few functions to replace the functions such as lookat and frustum.

3. Hints
    1. You do not have to follow these suggestions, they are provided in an effort to help.
    2. An empty screen is frustrating. Separate reading and loading obj data into buffers for rendering from viewing. Get the 3D viewing working first. Create a simple static object description in a vertex array object and successfully view it. For example, a simple triangle in 3D with vertices at {1,0,0}, {0,1,0} and {0,0,1} will

work. Set up a simple view out the x or y axis 2 units looking back at the origin. A usable starting set of viewing parameters for this test triangle is eye {2,2,0}, focal point {0,0,0} and viewup {0,0,1} with a near distance of 1.5, and far distance of 20. Once you can see this triangle, then work on viewing obj file(s). You will want to experiment with different near and far distances. If the far distance is to close, many or all of your objects can be clipped away.

3. Use the vmath routines directly, if your program is in C++, or use some of those routines to write your own routines to set up the viewing matrices easier. You can use the vmath header and / or copy code from that file into your code. For example, routines vmath::lookat and vmath::frustum build viewing and perspective matrices.
4. The vmath header file also has methods for building transformation matrices.
5. Start with a static view and get that working, then add the ability to move the eye, focal point and view up vector.
    - i. To establish initial viewing parameters keep track of the x,y, and z bounds of all the objects. And remember to determine bounds using the modeling transform.
        1. A usable eye location is (xRange * 1.5, yRange * 1.5, zRange * 1.5).
        2. A usable starting focal point is (midpoint xRange, midpoint yRange, midpoint zRange)
6. The OpenGL Red Book example code of chapter 4 for the shadow map has some useful example code, but there is code that you won't be using for this assignment. Chapter 4 introduces shadow maps, which is more than we're doing for this assignment.

4. Grading rubric – 100 points total

1. [70 points] reading and displaying obj objects.
    - i. View one simple object.
    - ii. View one moderately complex object.
    - iii. Applying modeling transforms.
    - iv. Viewing multiple objects with individual modeling transforms.
    - v. Reshape display window.
2. [15 points] control of viewing parameters, including camera, look at point, and view up direction
    - i. Control of camera or eye position.
    - ii. Control of focal point position.
    - iii. Control of view up vector
3. [5 points] Other keyboard commands
    - i. r, w, and s
4. [10 points] For the written assignment answering the questions below.

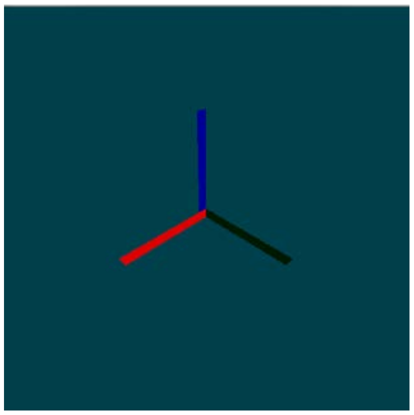
Questions

5. [3 points] Compute the viewing matrix for the following settings:
    i. Eye = 5,0,0
    ii. Focal point = 0,0,0
    iii. View up = 0,0,1

6. [4 points] For the world coordinate 1,0,0, what is its value in eye coordinates using the viewing matrix in problem 3?

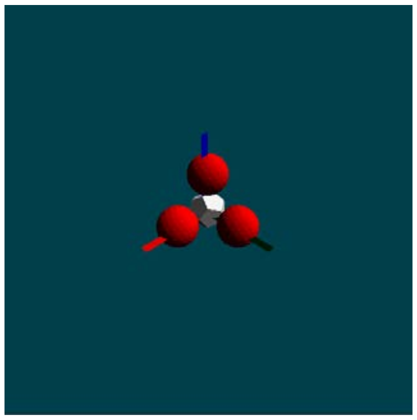
7. [3 points] Compute the face normal for a triangle described by these vertices (1,0,0), (0,1,0) and (0,0,1). Assume CCW ordering of these coordinates.

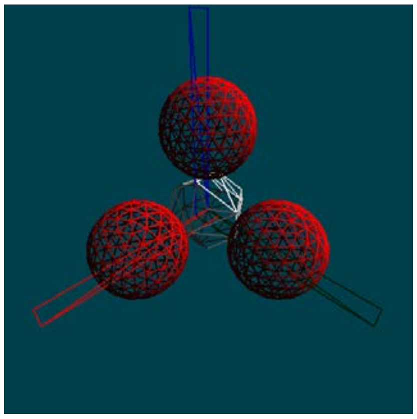
Sample output



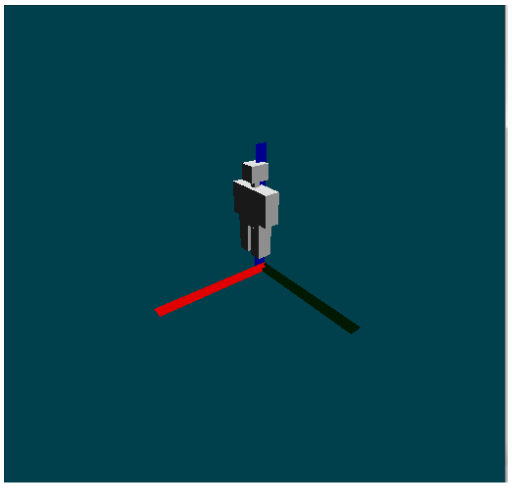
```
Control file

obj objFiles/axis.obj
```





```
obj  objFiles/dodecahedron.obj
obj objFiles/sphere.obj
t 2 0 0
obj objFiles/sphere.obj
t 0 0 2
obj objFiles/sphere.obj
t 0 2 0
obj objFiles/axis.obj
s 4 4 4
```



```
obj objFiles/humanoid_tri.obj
obj objFiles/axis.obj
s 20 20 20
```

Submission Instructions

Submit your files on the host **lectura.cs.arizona.edu** using the command
turnin cs433s15-assg2 [files] We will use your makefile to create your viewer for testing.

Assignment Advice

1. Start early.
2. Do your own work.
3. Check piazza regularly.