Assignment 2. The Graphics Pipeline

Total of Points of the Assignment: 20

For your first programming assignment, you have implemented a simple viewer for polygonal models using OpenGL. Your program could accept a triangle mesh file containing the description of the model and use OpenGL API to render the triangle mesh. By using the OpenGL fixed pipeline, your program supports a lot of features while rendering.

In this second assignment, you will be implementing some important tasks to replace the work of a subset of the OpenGL fixed rendering pipeline. Before you start your own implementation, you have to take a close look at the OpenGL pipeline. Basically, your work in this assignment should mimic OpenGL in the following tasks:

1. Matrix manipulation: when you translate or rotate the camera along its own axis, you should calculate the model-view matrix by yourself. Also, when you do the vertex projection from Camera Coordinate System (CCS) to image plane, you should implement your own projection matrix. They should be exactly the same as the matrices when you use gluLookat(), gluPerspective(), or glFrustum().
2. Shader rendering: instead of using the OpenGL fixed rendering pipeline, you are required to use GLSL (OpenGL Shading Language) to write your own vertex shader and fragment shader. In the vertex shader, you should write code for vertex position transformation by passing your own model-view and projection matrices. In the fragment shader, proper color should be set. There is one thing you should always keep in mind: when a shader is being used, the corresponding functionality of the fixed pipeline at this stage is disabled.

Goals

By successfully completing this assignment, you should feel very comfortable with concepts such as geometrical transformations, perspective projection, OpenGL pipeline, and basic GLSL grammar.

Assignment Description

Your viewer should provide at least the following features:

a) Calculate the model-view and projection matrices by yourself. **(6 points)**
b) Write code for vertex position transformation by passing your own model-view and projection matrix to the vertex shader. **(3 points)**
c) Set proper color in the fragment shader. **(3 points)**
d) Support translations and rotations of the virtual camera by modifying the model-view matrix. *Translations and rotations should be defined with respect to the coordinate system of the camera (CCS).* **(4 points)**

e) Render the models using points, wireframe and solid representations. **(1 points)**
f) Render the objects at the center of the windows. **(1 points)**
g) Reset the camera to its original position. **(0.5 points)**
h) Support for changing the values of the near and far clipping planes. **(0.5 points)**
i) Support for interactive change of colors (R, G, B) for the models, making sure that the color change is apparent under all rendering modes. **(0.5 points)**
j) Support for reading a new model file through the user interface. **(0.5 points)**

Note: Remember that writing clean code is important for reusability as well as for easy understanding from other people. This project involves a fair amount of complexity and I advise you to design your project before your start coding. This will allow you to clearly identify the necessary modules and the relationship among them. The use C++ can help you produce a clean implementation of your design.

Start to work on this assignment as early as possible. It might be harder than what you think. Good luck!

Tips on how to complete the assignment

In order for you to be able to take the most from these tips, **you should study Chapter 4 of Angel's book and the corresponding Course Slides, as well as Chapter 15 (GLSL) of the Red book**.

a) You will need to define a camera object (a good idea is to create a class camera) with at least the following attributes:
   i. *Position* (x, y, z). This defines the position of the camera in 3D. The camera position is also known as its center of projection (COP), viewpoint or eye point.
   ii. A coordinate system associated with the camera (CCS). This will define the camera's orientation in 3D. In our general derivation of change of basis, we represented the CCS using three 3D vectors: *u*, *v* and *n*.
   iii. Vertical fields of view (vfov) and Horizontal fields of view (hfov = vfov * aspect_ratio) of the view volume.
   iv. Distances associated with the near and far clipping planes.

   Initialize your camera with the following parameters (remember that in order for the rendering of your program to match OpenGL's, you should use a right-hand coordinate system):

   Position = <position that allows you to see the whole object as computed in assignment #1>
   u = (1, 0, 0), v = (0, 1, 0), n = (0, 0, 1)
   hfov = vfov = 60 degrees

b) All transformations and projections are performed via matrix multiplication. Since matrices play such an important role in this project, you should spend some time designing a good and flexible matrix class. Remember that since we are using

homogeneous representations of points in 3-D, you will need 4 by 4 matrices. Please, write your own code instead of copying a matrix library from the internet.

The matrix class should provide at least the following methods:

i) Multiplication between two matrices
ii) Set a ModelView Matrix
iii) Set a Projection Matrix

With these methods you are ready to complete the assignment.

c) There are two important matrices involved in the transformation from world coordinates to screen coordinates:

i) **Model/View Matrix (M)**: It represents the change of coordinate system (from WCS to CCS). If c = (c_x, c_y, c_z) is the position or center of projection of the camera and the CCS is defined by the vectors u, v, and n, then

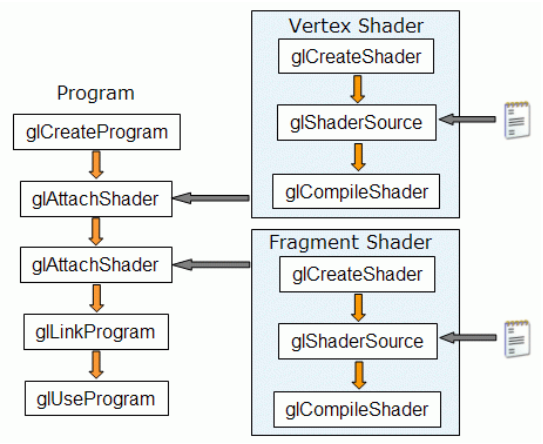
$$M = \begin{bmatrix} u_x & u_y & u_z & dx \\ v_x & v_y & v_z & dy \\ n_x & n_y & n_z & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where *dx* = -c dot u, *dy* = -c dot v and *dz* = -c dot n.

ii) **Projection Matrix (P)**: Projects points defined with respect to the CCS onto the image plane of the normalized perspective view volume.

d) Once you have these important matrices, the rendering of the scene/model can be described by: (But all of these should be done in the vertex shader)
   i) Compute PM = P*M. This composite matrix embodies both the geometric and projection transformations.
   ii) For each triangle (of the model) with vertices v1, v2 and v3, compute v1', v2' and v3', where vi' = PM*vi.

e) The figure bellow shows the necessary steps (in OpenGL 2.0 syntax) to create the shaders. Before writing your code, make sure you have read Chapter 15 of the OpenGL Red Book.



f) If you want to use your shader in OpenGL application, then OpenGL 2.0 or above will be required. You'd better check it – you can take a look at GLEW. GLEW simplifies the

usage of extensions and newer versions of OpenGL to a great deal since the new functions can be used right away.

Good Luck!