CMSC 435 / 634 Introduction to Computer Graphics

Project Assignment 2: Viewing (Due Oct 1st 11 pm)

Goals of this project: Understand viewing matrix setup in computer graphics. Once this project is completed, you will possess all the tools needed to handle displaying 3D objects oriented in any way and viewed from any position.

Introduction

In this assignment you will be writing the tools necessary to move, rotate and scale your camera and objects using the robot scene from project 1. You will:

- Write camera handling routines to represent a perspective transformation.

We want to be clear up front, however, that this assignment has the potential of being quite rough, so take it seriously and start early!

Reading Chapter 3 in the OpenGL Redbook will help this assignment tremendously.

Motivations:

When rendering a 3D scene, you need to go through several stages. One of the first steps is to take the objects you have in the scene and break them into triangles. You did that in project 1. The next step is to place those triangles in their proper position in the scene. Not all objects will be at the "standard" object position, and you need some way of resizing, moving, and orienting them so that they are where they belong. Then you need to set up the appropriate camera locations such that the geometries will be shown on the 2D screen.

This assignment is for starting to handle the problem either with OpenGL **matrix manipulation** routines (for 435 students) or your own linear algebra package (everything from scratch for 634 students).

This programming assignment is to define how the triangulated objects in the three-dimensional scene are displayed on the 2D screen. This is accomplished through the use of camera transformation, a matrix that you apply to a point in three-dimensional space to find its projection on a 2D plane. While it is possible to position everything in the scene so that all the camera matrix needs to do is flatten the scene, the camera transformation usually incorporates handling where the camera is located and how it is oriented as well.

Now it is entirely possible to simply throw together a camera matrix which you can use as your all-purpose transformation for any 3D scene you may wish to view. All that you would have to do is make sure you position your objects such that they fall within the standard view volume. But this is tedious and inflexible. What happens if you create a scene, and decide that you want to look at it from a slightly different angle or position? You would have to go through and reposition everything to fit

your generic camera transformation. Do this often enough and before long you would really get bored. What we really want to do is to follow the lectures and define several relatively independent transformations.

Requirements:

--- Camera handling routines (this part is more complex)**:**

Implement functions to set ***four*** classical camera's position and orientation relative to the visible axes (the red X, green Y, and blue Z ones): along the three "FOO-axis" and a "birds' eye" view. For the "FOO-axis" locations, position the camera so that it is located two units along the FOO (x or y or z) axis, pointing towards the origin. The "birds' eye" will positions the camera such that it is located at the point (2,2,2) and is pointing towards the origin. This view is similar to the viewing example I showed in our lecture). The supporting code provides a key input "c" for you to shuffle among these options.

The last two parameters of the camera, "Height angle" and "Aspect ratio," control the shape of the viewing frustum. Your job is to allow the adjustment of these parameters. The "Height angle" key (h/H) should interactively adjust the height angle of the camera, where pressing h or H will reduce or increase the "Height angle" accordingly; similarly for the "Aspect ratio" (recall that width = height ∗ Aspect ratio), where keys a / A will increase or decrease the ratio.

It is worth noting that the two controls of "Height angle" and "Aspect ratio" are interdependent: certain combinations of Height angle and Aspect ratio are illegal in the sense that they result in angles larger than 180◦. However, your code does not need to handle the illegal conditions.

The supporting code provides some key inputs. Functions of these key inputs need to be implemented: "a" / "A" for increasing / decreasing the aspect ratio; "w, v, u" / "W, V, U" for increasing the decreasing the angles of the camera in its local coordinate system.

Other implementation in this assignment involves the following:

- Maintaining a world-to-viewing plane matrix that implements the perspective transformation.
- Setting the camera's absolute position/orientation given an eye point, look vector, and up vector.
- Setting the camera's height angle and aspect ratio.
- Translating the camera in world space.
- Rotating the camera about one of the axes in its own virtual coordinate system.
- Setting the near and far clipping planes.
- And having the ability to, at any point, spit out the current eye point, look vector, up vector, height angle, aspect ratio, or world to film matrix, as if I can manipulate each of these individually.

Supporting code:

The example code uses the following defaults for the camera's initial stage. The near clip plane is 1 and the far clip plane is 100. The vertical view angle is 60-degree and the screen aspect ratio is 1:1. The eye of the camera is at (2, 2, 2) and the camera is looking at the origin, and its up vector is (0, 1, 0).

The translation and rotation are enabled through keyboard to modify the camera's position, though relative to a virtual set of axes (a set of axes different from the ones shown in the scene), which represent the camera's "local" coordinate space. In the camera's coordinate space, the camera is located at (0,0,0), the camera is looking along the negative Z axis, the Y axis is pointing upwards, and the X axis is pointing to the right. To emphasize the difference between the normal coordinate space and the camera's "local" coordinate space, these axes are usually referred to as *W*, *V*, and *U*, respectively. When you move the camera, this set of axes move with it.

A Makefile is provided and all the modifications that need to be made to it are for you to add the name of your object files and source files. The support code comes prepackaged to draw a single triangle, which should help you get started on the right track.

You will need to insert the World to ViewPlane matrix yourselves using a few OpenGL commands. OpenGL requires two separate transformation matrices to place objects in their correct locations. The first, the ModelView matrix, positions and orients your camera relative to the scene, or if you prefer, orients the world relative to your camera. In any case, it is taken care of by support code in this assignment. The second, the Projection matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In this assignment, this is your responsibility. More specifically, in your putWorldToViewplane function in the Camera, you must make the appropriate calls to configure the Projection Matrix.

The method putWorldToViewplane() is called frequently by the support code, so you should cache the matrix rather than recomputing it needlessly every time it is called. Here is some sample code to help you out. . .

```
// tell GL that future matrix operations should be applied to the projection
//matrix
glMatrixMode(GL_PROJECTION);

// give a pointer to an array of doubles containing the matrix to load. The
//matrix pointed to will become the new projection matrix
glLoadMatrixd(pointer to first elt of double[16] array);
```

For more information, *man glMatrixMode* and *man glLoadMatrix* in a shell. Some other useful OpenGL functions include *glMatrixMode, glPushMatrix/glPopMatrix, glLoadIdentity, glLoadMatrix, glMultMatrix, glRotatef, glScalef, and glTranslatef*.

Using OpenGL viewing routines such as *glLookAt*, *gluPerspective*, is not allowed. You will have to use OpenGL matrix manipulation routines to support the calculations yourself.

643 only:

Implement the camera routines using your OWN linear algebra package, to perform these matrix manipulations. The package should be capable of operations on matrices, vectors, and points.

Please do not use any other library camera functions, we want to see your camera and math functions in action. Remember, GL_PROJECTION is only for the camera transformation matrix. Other parts of an object's transformation do not belong here.

Extra credits

For 5 points, try implementing an orthographic camera (this is relatively simple).

For 8 points, write geometry handling routines that the robot can walk (arms and legs are moving in the appropriate reference frames). You will want to figure out the scene graph and how different parts of the body move together. For the transformation, refer to our lecture and the OpenGL Redbook Chapter 3 and example code for details. It is okay to let the robot walk in place (do not move in the world coordinate) or walk along the "FOO-axis" (x, y, or z).

For 10 points for 435 students, implement the 634 part of the assignment.

What to turn in

Source code only by email to TA. Please do not include any .o files. Please include:

- A README with your handin containing basic information about your design decisions and any known bugs or extra credit;
- How to compile and run your code as if you are telling a colleague that is to continue the development. This means you will need to submit **all code**.
- Please name your project directory as **02viewing_<your umbc name>** and put everything in that directory.

You only need to submit ONE tarball that compresses all files in the ***02viewing_<your umbc name>*** directory. To create the tarball, on the gl server, go one level above the 02viewing_<your umbc name> directory and then type in the following line.

tar cfv 02viewing_<your umbc name>.tar 02viewing_<your umbc name>

Email 02viewing_<your umbc name>.tar to the TA Alisa.