Rotation, Zoom and Pan in OpenGL

Peter Salvi

June 29, 2008

1 Introduction

This is a memo that explains *one* way to do standard camera movement in a perspective OpenGL window. This method depends exclusively on the GLU functions `gluPerspective` and `gluLookAt`, which means that every movement is given as a change of the following data:

- The eye position in object space (eye)
- The central position (where the eye looks at) in object space (center)
- The (unit) direction of going *up* on the screen in object space (up)
- The clipping planes (zNear and zFar)

There are also a few parameters these movements depend on:

- The window geometry (width $\times$ height)
- The field-of-view angle in the y direction (FoV_y)

Although the FoV_y is fixed in most applications (generally at a value between 45° and 60°), the window geometry can change in the course of the program. Since OpenGL needs to be informed about this, there should be a reshaping function that is called every time a change in the window geometry occurs. As this function is responsible for setting the correct perspective, it should also be called when initializing the OpenGL window.

Another important function is the display function that is called every time the program has to show the OpenGL window. This will take care of the modelview matrix transformations. Section 2 explains these functions and Section 3 and 4 introduces the computations required for the various movements. Calculation of the correct clipping planes is shown in Section 5, and an appendix contains the details of non-trivial computations.

All program snippets and computations are based on the program *SFView* I've written a few days ago.

2 Reshaping and displaying

The reshaping function needs to tell OpenGL the new size of the window and also set the correct perspective transformation. This can be achieved by the following piece of code:

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(FoV_y, width/height, zNear, zFar);
glMatrixMode(GL_MODELVIEW);
```

Here zNear and zFar are computed as shown in Section 5.

The displaying function has to clear the screen, set the modelview transformation, display the objects and update the screen. We can do this with these commands:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z, center_x, center_y, center_z, up_x, up_y,
    up_z);
// Display the objects...
glFlush();
// Swap buffers...
```

3 Movement

The movements treated here are generally connected to mouse drag events. For our needs, this means that every time the mouse moves, we get the relative distance from its last position (Δx and Δy, both in the range $[-1, 1]$). This can be done, for instance, by saving the "old" mouse position on a click event, and resetting it with the new coordinates in the *end* of the mouse motion function. Then

$$\begin{aligned} \Delta x &= \frac{x_{\text{new}} - x_{\text{old}}}{\text{width}}, \\ \Delta y &= \frac{y_{\text{new}} - y_{\text{old}}}{\text{height}}. \end{aligned}$$

3.1 Rotation

Rotation in fact means two rotations: one in the direction corresponding to both of the window axes. We need a function that rotates a point around an axis that is fixed in space (see the Appendix), let's call it `rotatePoint(`*point*`,` *origin*`,`

direction, *angle*). Both axes have their origins at center, and their directions are

$$\begin{aligned} \mathrm{dir}_x &= \mathrm{up}, \\ \mathrm{dir}_y &= \frac{\mathrm{up} \times (\mathrm{center} - \mathrm{eye})}{\|\mathrm{up} \times (\mathrm{center} - \mathrm{eye})\|}. \end{aligned}$$

Then the rotation is done with the following three commands:

```
eye = rotatePoint(eye, center, dir_x, -Δx · π);
eye = rotatePoint(eye, center, dir_y, Δy · π);
up = rotatePoint(center + up, center, dir_y, Δy · π) - center;
```

The vector up should still be a unit vector after this transformation, but due to floating point errors, it is better to normalize it again after the above commands.

3.2 Zoom

We just have to move the eye closer to (further from) the center:

$$\mathrm{eye} = \mathrm{center} + (\mathrm{eye} - \mathrm{center}) \cdot (\Delta y + 1).$$

3.3 Pan

The height of the window in object space is given by

$$\mathrm{length} = 2 \, \|\mathrm{eye} - \mathrm{center}\| \cdot \tan \tfrac{\mathrm{FoV}_y}{2},$$

so the new position of center is:

$$\begin{aligned} \mathrm{center} = \mathrm{center} \quad &+ \quad \mathrm{dir}_y \cdot \Delta x \cdot \mathrm{length} \cdot \frac{\mathrm{width}}{\mathrm{height}} \\ &+ \quad \mathrm{dir}_x \cdot \Delta y \cdot \mathrm{length}, \end{aligned}$$

using the vectors dir_x and dir_y as defined in 3.1. This formula (reasonably) supposes that the width and height of the window do not differ too much.

4 Zoom to a bounding box

Another often required feature is to zoom the view to a given bounding box. We will set up our environment such that eye direction will point along the z axis and the up vector along the y axis.

Let the bounding box given by its two opposing corners, b^1 and b^2, with b^2 having the larger z coordinate. The new center of the screen will be $\mathrm{b}^1 + \frac{1}{2}(\mathrm{b}^2 - \mathrm{b}^1)$, and we set the up vector to the y direction, i.e. (0, 1, 0). Then we take the

larger of the two visible directions of the bounding box, and multiply by some factor (e.g. 1.2) to give some space on the sides:

$$\text{length} = 1.2 \cdot \max(\left|\mathrm{b}_x^2 - \mathrm{b}_x^1\right|,\ \left|\mathrm{b}_y^2 - \mathrm{b}_y^1\right|).$$

Now we can set the eye position:

$$\text{eye} = \left(\text{center}_x,\ \text{center}_y,\ \mathrm{b}_z^2 + \frac{\text{length}}{2\tan(\text{FoV}_y/2)}\right).$$

In fact, this is only precise in the y direction and could be refined, but it is adequate in most applications.

5 Clipping

With the exception of rotation, all of these movements affect the clipping planes, which can be set using the `gluPerspective` command, as seen in Section 2. We can calculate zNear and zFar as the minimal and maximal distance of the bounding box vertices from the eye position in the eye direction. It also advised to multiply these values by some factors for security's sake.

$$\begin{aligned} \text{zNear} &= 0.4 \cdot \min_i(\mathrm{b}^i - \text{eye}) \cdot \mathrm{dir}_{\text{eye}}, \\ \text{zFar} &= 1.7 \cdot \max_i(\mathrm{b}^i - \text{eye}) \cdot \mathrm{dir}_{\text{eye}}, \end{aligned}$$

where $\mathrm{dir}_{\text{eye}} = \frac{\text{center}-\text{eye}}{\|\text{center}-\text{eye}\|}$.

A problem arises, when the eye gets *inside* the bounding box — this means that zNear becomes negative. Although the OpenGL documentation says zNear should always be positive, setting zNear to 0 is an effective countermeasure.

Appendix

Rotation around a vector fixed in space

Rotating point p by α degrees around a vector v originated from o can be done as follows:

$$\hat{p} = o + \begin{bmatrix} v_x v_x C + c & v_x v_y C - v_z s & v_x v_z C + v_y s \\ v_y v_x C + v_z s & v_y v_y C + c & v_y v_z C - v_x s \\ v_z v_x C - v_y s & v_z v_y C + v_x s & v_z v_z C + c \end{bmatrix} \cdot (p - o),$$

where $c = \cos\alpha$, $s = \sin\alpha$ and $C = 1 - \cos\alpha$.