Viewing and Camera Control in OpenGL

Niels Joubert & James Andrews *

2008-10-28

1 Basic OpenGL Transformations

Whenever we work in OpenGL, we have access to its API that defines functions for basic transformations. We use the following functions to create a transform matrix and push it onto the current transform stack[1]:

```
glScalef(x, y, z)           - scale object by x,y,z scalars along the current axes
glTranslatef(x, y, z)       - move an object by x,y,z along the current axes.
glRotatef(x, y, z, angle);  - rotate an object by angle around vector [x,y,z]
```

Using techniques from linear algebra, any transform can be decomposed into a combination of these simple transforms. Luckily, we do not have to resort to always using only these transforms - OpenGL defines an interface to intuitively work with the camera's orientation and perspective, or we can define arbitrary transforms and load this into OpenGL.

2 OpenGL Transformation Stacks

We think of viewing in OpenGL as specifying properties about a hypothetical camera inside our scene. To do this, we will be specifying transformations that must be applied to our world during the rendering process. OpenGL supports this by storing transformations in a user-controllable stack. OpenGL has 3 different transformation stacks which are applied at different times of the rendering pipeline. We are concerned with two of these stacks: the *Modelview* stack and the *Projection* stack.[2] We will explore the relation between our hypothetical camera and these transformations stacks. Each of these OpenGL transformation stacks specify the following information about the camera:

- **GL_MODELVIEW** - The position and orientation of the camera and the world.
- **GL_PROJECTION** - How the camera sees the world

3 Projection transformation

The projection transform defines how a point (or vertex) is transformed from world space to the 2D plane of the screen (screen space). This is part of what we studied when we discussed perspective transforms. OpenGL gives you functions to define an *orthographic* or a *perspective* projection **relative to the camera**:

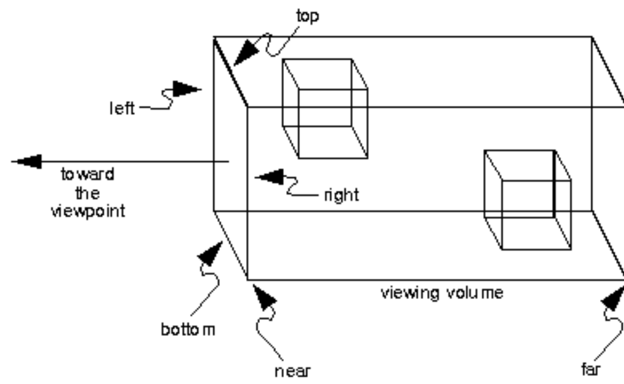
```
glFustrum (left, right, bottom, top, near, far);
gluPerspective(fovy, aspect, near, far);
glOrtho(left, right, bottom, top, near, far);
```

The following sketches should explain each of these functions:

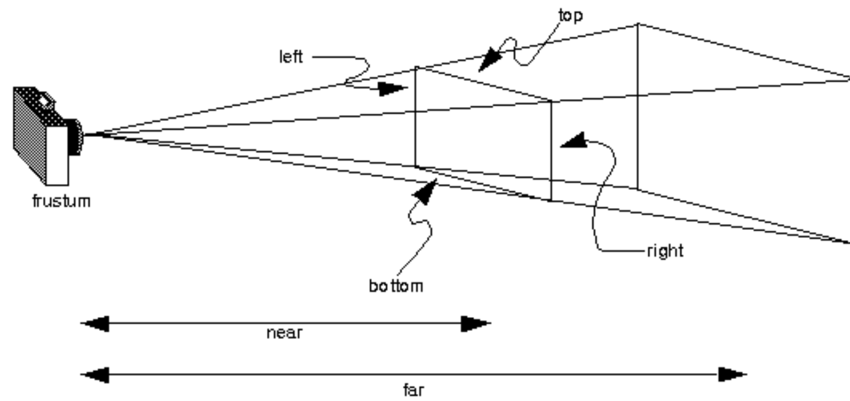
*Much of this handout was adapted from Stu Pomerantz at the Pittsburgh Supercomputing Center. http://www.psc.edu/

[1] See section 2

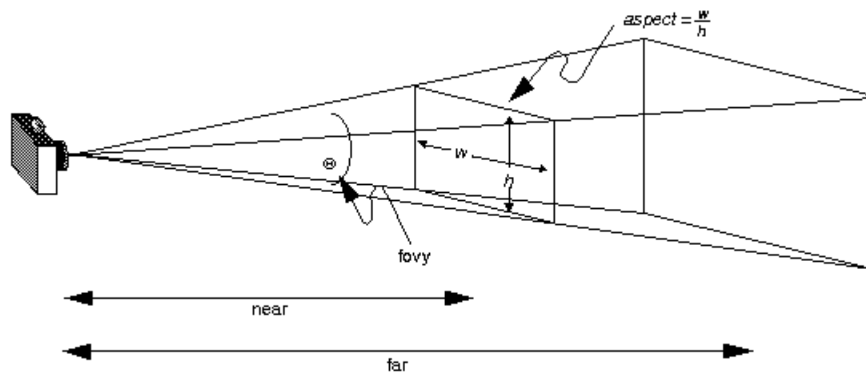
[2] There is also a Texture transform stack.



(a) glOrtho given left, right, bottom, top, near and far



(b) glFustrum give left, right, bottom, top, near and far



(c) glPerspective given field-of-view, aspect-ratio, near and far

Figure 1: OpenGL perspective functions

4 Modelview transformation

The Modelview transformation specifies both the position and orientation of the camera and the objects in the world. Why don't we have a model transform and a view transform? Because translating the world and translating the camera has exactly the same effect - thus we combine it into one Modelview transform. We tend to specify a viewing transform to place the camera, followed by transformations on the objects. Note that placing the camera using the viewing transform is exactly the same as applying the rotations and translations to place your camera using OpenGL's transformation functions.

```
Define the viewing transform:
  void gluLookAt(eyeX,eyeY,eyeZ, centerX,centerY,centerZ, upX,upY,upZ)

PARAMETERS
  eyeX,eyeY, eyeZ  - Specifies the position of the eye point.
  centerX, centerY, centerZ - Specifies the position of the reference point.
  upX, upY, upZ - Specifies the direction of the up vector.
```

Since the default transformation on any stack is the identity, this translates into a default camera with eye at $(0, 0, 1)$ and center at $(0, 0, 0)$ with an up direction of $(0, 1, 0)$ along the y axis. In other words, gluLookAt(0,0,1,0,0,0,0,1,0) is the default view.

5 Managing the Transformation Stacks

Today's graphics programs demand complex scenes with many objects, each rendered under its own transform. To facilitate this, as mentioned, OpenGL stores transformations in a stack. We control this stack through the following four methods:

- **glMatrixMode(**$STACK$**)** - Selects the stack to affect.
- **glLoadIdentity()** - Resets the selected stack to the Identity transform.
- **glPushMatrix()** - Duplicates the top transform of the current stack.
- **glPopMatrix()** - Deletes the top transform of the current stack.

We tend to initialize the ModelView stack to the identity, then apply our viewing transform (positioning the camera). Once this has occurred, for each object we want to render, we push a duplicate matrix onto the modelview stack, apply the transformations for the given object, draw the object, and pop the top matrix off the stack, thereby returning to the original view transform, ready to draw the next object.

6 Putting it all together - Specify the Camera

In order to specify the view in OpenGL:

- Set the viewport
- Set the projection transformation
- Set the modelview transformation

If you specify your viewport and projection on initialization, you need to specify only your modelview transformation every time you render. OpenGL's stack-based approach then allows you to apply different transformations to each object. Thus, a possible Reshape callback function (also called on initialization) would look like the following:

Listing 1: Specifying the view

```
void reshape(int w, int h) {
    //Set the viewport
    glViewport(0,0,w,h);

    //Set the projection transform
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45,1,5,100);
}
```

And a possible display function would look like the following:

Listing 2: Rendering the scene

```
float zoom, rotx, roty, tx, ty;

void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //Clear Z-Buffer

    //Set the camera orientation:
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,-1, 0,0,0, 0,1,0);

    //Rotate and zoom the camera. This order gives you maya-like control.
    glTranslatef(0,0,-zoom);
    glTranslatef(tx,ty,0);
    glRotatef(rotx,1,0,0);
    glRotatef(roty,0,1,0);

    //FOR EACH OBJECT:
        glPushMatrix(); //Save the current view matrix.
        //HERE YOU CAN APPLY TRANSFORMATIONS TO EACH OBJECT BEFORE DRAWING IT
        //AND HERE YOU CAN DRAW IT!
        glPopMatrix(); //Restore to the view matrix

    glutSwapBuffers();
}
```

For more information, consult either the Red Book, or the following sites:

http://www.newcyber3d.com/selfstudy/tips/camera_analogy.htm
http://www.robthebloke.org/opengl_programming.html#4
http://www.morrowland.com/apron/tut_gl.php

7 Assorted OpenGL notes - OpenGL 101

We highly recommend a copy of the so-called Blue[3] and Red[4] books for all graphics programmers. OpenGL is your friend. The cake is not a lie.

NOTE: We suggest googling for the MAN pages of the functions we mention here!

7.1 Drawing Objects

You already know how to do this, but as a brief overview, objects are drawn by issuing `glVertex()` and `glNormal()` calls between `glBegin(TYPE)` and `glEnd()` commands. For the normals to matter, you also want to enable lighting and define both material properties and lights, as covered in section 7.6. If you scale your objects, you need to have OpenGL renormalize your normals. You can enable this behavior with `glEnable(GL_NORMALIZE)`.

7.2 Shading

OpenGL supports flat and smooth (gouraud) shading as part of the hardware pipeline. This is toggled between with `glShadeModel(GL_SMOOTH)` and `glShadeModel(GL_FLAT)`. The shading model uses colors at vertices which are either computed by the lighting model or, if lighting is not enabled, specified directly with `glColor()`.

7.3 Wireframes

OpenGL can draw polygons in one of several modes, controlled through the `glPolygonMode()` function. You can switch between filled polygons and outlined polygons with `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)` and `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. Keep in mind that lighting and shading occurs in the same fashion for both of these, so it is often wise to disable lighting when drawing in `GL_LINE` mode, which you can do with `glDisable(GL_LIGHTING)`.

7.4 The Z-Buffer and Depth Tests

OpenGL implements a Z-Buffering algorithm to calculate the visibility of polygons. The Z-Buffer is initialized by passing the `GLUT_DEPTH` flag to `glutInitDisplayMode()`, and enabled with `glEnable(GL_DEPTH_TEST)`. Once the Z-Buffer has been enabled, you can clear it by calling `glClear(GL_DEPTH_BUFFER_BIT)`, something you normally want to do at the start of each frame's rendering cycle. If you want greater control of how the depth test is performed, you can use `glDepthFunc(FUNC)`. By default, a pixel is compared with the current Z-Buffer value using `GL_LESS`.

7.4.1 Hidden Line Removal

We can use a trick to draw wireframes but remove hidden lines by employing a two-pass scheme. Render the scene with filled polygons without populating your color buffer (call `glColorMask()` with all `GL_FALSE`) to calculate the depth buffer values. Then rerender the scene with wireframed polygons without clearing the depth buffer, but switching the depth test from `GL_LESS` to `GL_LEQUAL`.

7.5 Display Lists

Display lists provide an easy way to speed things up, by letting OpenGL remember a list of rendering instructions. It can optimize the instructions for rendering, and store them so you don't constantly need to send them to the card yourself. See `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=12`

[3] http://www.opengl.org/documentation/blue_book/
[4] http://www.opengl.org/documentation/red_book/

7.6 Lighting Example

Listing 3: Lighting Example

```
void initLights() {
    glEnable(GL_LIGHTING);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

    GLfloat global_ambient[] = { .1f, .1f, .1f };
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);

    //define and enable light0. You have 8 lights in total.
    GLfloat ambient[] = { .1f, .1f, .1f };
    GLfloat diffuse[] = { .6f, .5f, .5f };
    GLfloat specular[]={0.0, 0.0, 0.0, 1.0};
    GLfloat pos[] = { -3, 0, 2, 1 };
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, pos);
    glEnable(GL_LIGHT0);

    //define material properties:
    //You probably don't want to use the emission term for this class.
    GLfloat mat_specular[]={1.0, 0.0, 0.0, 1.0};
    GLfloat mat_diffuse[]={0.0, 1.0, 0.0, 1.0};
    GLfloat mat_ambient[]={0.0, 0.1, 0.1, 1.0};
    GLfloat mat_emission[]={0.2, 0.0, 0.0, 1.0};
    GLfloat mat_shininess={10.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emission);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);

    glShadeModel(GL_SMOOTH); // GL_FLAT gives flat shading

    //Allows you to scale objects at the cost of some performance.
    glEnable(GL_NORMALIZE);
}

int main(int argc,char** argv) {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);
        glutInitWindowSize(640,480);
        glutCreateWindow("Lighting Test");
        ...
        initLights();
        glEnable(GL_DEPTH_TEST);
        glutMainLoop();
}
```