An Introduction to 2D OpenGL

January 23, 2015

OpenGL predates the common use of object-oriented programming. It is a mode-based system in which state information, modified by various function calls as the system runs, is preserved in the system. In modern object-oriented terms this translates into classes that preserve the state information and methods of these classes that modify it. There are two primary classes that we will use: GL2, which has most of the methods that directly modify the output, and GLU (short for GL Utilities), which has some shortcuts, particularly for setting up the camera (or view) model. We will generally construct objects of these two classes at the start of our program (i.e, in the constructor for the main class) and save them in class variables so they may be used throughout the program.

Imports. You need to import the following three libraries:

```
import javax.media.opengl.*;
import javax.media.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.*;
```

You may find that Eclipse automatically imports them when you use the OpenGL classes.

The main class constructor. We will set up OpenGL programs so that the main( ) method constructs an object of the primary class. The constructor for this class needs to start up OpenGL and create the appropriate variables. Here is what needs to happen

```
GLProfile glp=GLProfile.getDefault();
GLCapabilities caps = new GLCapabilities(glp);
canvas = new GLCanvas(caps);  // canvas is a class varible of type GLCanvas
// here the user interface should be created and
// the canvas added to the window frame.
canvas.addGLEventListener(this);
FPSAnimator animator = new FPSAnimator(canvas, 60);
animator.start();
```

The **canvas** is a window in which all of the output of your program will appear. It is possible to have multiple canvases open at once, but we won't do that. The canvas needs to add a **GLEventListener** or it won't be drawn. I generally have the main class implement the **GLEventListener** interface, hence the reference to ***this*** as the Listener object. This requires the main class to have **init( )**, **display( )**, and **reshape( )** methods, as noted below. Finally, the **animator** allows the canvas to be redrawn. You need to create the animator even if your program does not run a continuous animation but only changes the canvas as the result of user interactions. The number in the **FPSAnimator** constructor is the desired frame rate for the program.

The init( ) method. This has startup code for OpenGL. Note that you never call this method; it is called by the OpenGL system that you attached to your program in the main class constructor. At a minimum this method should create the gl and glu class variables. You probably also want it to create a coordinate system on the canvas. Here is a typical **init( )** method:

```
public void init(GLAutoDrawable drawable) {
        gl = drawable.getGL( ).getGL2( ); // gl is a class variable of tye GL2
        glu = new GLU( );  // glu is a class variable of type GLU

        gl.glMatrixMode(GL2.GL_PROJECTION);
        gl.glLoadIdentity();
        glu.gluOrtho2D(0f, 10f, 0f, 10f);
        gl.glClearColor(1,  1, 0, 1);
}
```

This needs some discussion. There are two matrices in OpenGL: the Projection matrix and the ModelView matrix. The Projection matrix projects the world onto the canvas, and the ModelView matrix is applied to each object in the scene as it is created. The call

gl.glMatrixMode(GL2.GL_PROJECTION);

makes the Projection matrix "current", in the sense that subsequent matrix operations are applied to the Projection matrix rather than the ModelView matrix.

The next line,

gl.glLoadIdentity( );

loads the identity matrix into the current matrix (i.e., it makes the Projection matrix be the identity matrix).

The next like is the key one:

glu.gluOrtho2D(xmin, xmax, ymin, ymax)

creates a matrix that transforms a rectangular regions of the world bounded by xmin, xmax, ymin and ymax, onto the canvas. The current matrix is multiplied times this transformation matrix. Since the current matrix is the Projection matrix and its current value is the identity, this sets the Projection matrix to the transformation that maps that rectangular region onto the canvas. Note that this is a very common sequence in OpenGL. Rather than giving us methods that set a matrix to what we want, it gives us methods for setting the matrix to the identity and methods for multiplying it by something. When you design your own graphics system you can do things differently, but for now that is the way OpenGL works.

The last line of the **init( )** method sets the background color (which OpenGL calls the "clear" color). The four arguments are floats for red, green, blue, and "alpha" or opaqueness. This call has full red and green channels, no blue and is completely opaque, which makes for a yellow background.

When we start using 3D OpenGL we will have more complex **init( )** methods, but while we are doing 2D work this is all there is to setting up the view transformation.

The display( ) method also takes an argument of type **GLAutoDrawable**. In this method you need to give all of the code for setting up the current appearance of the canvas. I usually divide this into two portions – a call to method **update( )**, which modifies variables in your program for running an animation, and **render( )**, which contains the actual drawing commands. This means the display method is usually

```
public void display(GLAutoDrawable drawable) {
        update( );
        render( );
}
```

For example, if your canvas is supposed to display a rotating square, you might represent the vertices of your square in terms of an angle **theta**, which will be a class variable. The **update( )** method will add a small increment to theta. Your **render( )** method will compute each of the four vertices in terms of **theta** and use them to draw the square. All of this happens 60 times a second, or according to the frame rate you request in the **FPSAnimator** in the main class constructor.

The **reshape( )** method is called whenever the window is resized. It is also called when the window is initially created. We can, if we wish, place the call to gluOrtho2D( ) in the reshape method rather than the init method. To see why we might want to do this, consider a canvas that is 200x200 pixels. If we create a coordinate system indexed 0 to 10 in each direction and draw a 5x5 square on this canvas, it will look like a square, for it is 100 pixels wide and 100 pixels high. Now suppose we resize the window so is 400 pixels wide and still 200 pixels high. If the coordinate system (0 to 10 in each direction) remains the same, a 5x5 square will be 100 pixels high and 200 pixels wide. In other words, if the coordinate system is uniform squares have the same shape as the overall canvas. That may be what we want, but often we want something drawn as a square to look like a square, regardless of the shape of the canvas.

The reshape( ) method has signature

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height);

The last four arguments are the position and the shape of the canvas. We can use the width and height to adjust the mapping system so that squares are drawn as squares. Imagine that the width is twice the height. We want the x-coordinates to have twice the range of the y-coordinates. If the x-range is R, we want the y-range to be R/2. For general shape(width, height), we want the y-range to be R*height/width. Here, for example, is a **reshape( )** method that has coordinates 0 to 10 in both x and y for square canvases, and for non-square canvases coordinates that preserve the aspect ratio of the canvas:

```
public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
        gl.glMatrixMode(GL2.GL_PROJECTION);
        gl.glLoadIdentity();
        glu.gluOrtho2D(0f, 10f, 0f, (10f*height)/width);
}
```

In fact, the **reshape( )** method is called when the canvas is created, so we don't need the gluOrtho2 code in the **init( )** method at all. I usually have **init(**) set up the **g**l and **glu** variables and the clearColor and leave the view transformation to the reshape( ) method.

In summary, here are the required methods for a typical 2D OpenGL program:

```
public void init(GLAutoDrawable drawable) {
        gl = drawable.getGL().getGL2();
        glu = new GLU();
        gl.glClearColor(1,  1, 0, 1);
}

public void display(GLAutoDrawable drawable) {
        update();
        render();
}

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
        // this is called when the window is resized
        gl.glMatrixMode(GL2.GL_PROJECTION);
        gl.glLoadIdentity();
        glu.gluOrtho2D(0f, 10f, 0f, (10f*height)/width);
}
```

2D Drawing Commands

Many OpenGL command names end with a digit and a letter, as in **gl.glColor3f( )**. The digit refers to the number of arguments and the letter indicates the type of the arguments. **gl.glColor3( )** expects 3 arguments (the red, green and blue channels) and expects them to be floats in the range from 0.0 to 1.0. while **gl.glColor4i( )** expects 4 integer arguments (red, green, blue and alpha). Sometimes there is a final flag v, which indicates that the arguments are to be packed into an array of the appropriate type and size: **gl/.glColor3fv( )** expects an array with 3 floats for its argument.

Color As noted above, **gl.glColor3f( )** and **gl.glColor3fv( )** set the current drawing color. For example, **gl.glColor3f(0, 1, 0)** sets the current color to green. This stays in effect until a new call to **gl.glColor...( )** is made. For example, if you set the current color to green and then draw a square, the entire square will be green. On the other hand, if you set the current color to green and specify two vertices of the square, then set the current color to red and draw the other two, the square will ramp from green to red. In general, whenever a vertex is specified the current color is attached to it; the colors of the interior points of a polygon are interpolated from the colors of the vertices.

Clear. You probably want to start your display( ) code by clearing the screen. This is done with

```
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
```

Lines. All geometrical objects are draw between **gl.glBegin( )** and **gl.glEnd( )** "parentheses". To enter line-drawing mode you issue the command

```
gl.glBegin( GL2.GL_LINES);
```

You then give a series of vertex declarations with **gl.glVertex2f( )**. Each successive pair of vertices creates a new line. You leave the drawing mode with **gl.glEnd( )**. For example, the following code draws a black line from point (-1, 3) to (2, 4):

```
gl.glBegin(GL2.GL_LINES);
gl.glColor3f(0, 0, 0);
gl.glVertex2f(-1, 3);
gl.glVertex2f(2, 4);
gl.glEnd( );
```

Polygons. There are several ways to draw polygons, which only differ in the initial call to glBegin. If you use **gl.glBegin(GL2.GL_TRIANGLES)** then each successive triple of vertices are used to define a triangle. If you use **gl.glBegin(GL2.GL_QUADS)** then each group of 4 vertices defines a quadrilateral. Finally, if you use **gl.glBegin(GL2.GL_POLYGON)** then the vertices define a single, hopefully convex, polygon. Here, for example, is a procedure that draws a yellow square

```
private void drawBox() {
        gl.glClear(GL.GL_COLOR_BUFFER_BIT);

        gl.glColor3f(1, 1, 0);
        gl.glBegin(GL2.GL_POLYGON);
                gl.glVertex2f(x, y);
                gl.glVertex2f(x, y+side);
                gl.glVertex2f(x+side, y+side);
                gl.glVertex2f(x+side, y);
        gl.glEnd();
}
```

Similarly, here is code that draws a box rotating by an angle theta:

```
private void drawBox() {
        gl.glClear(GL.GL_COLOR_BUFFER_BIT);

        gl.glColor3f(1, 1, 0);
        gl.glBegin(GL2.GL_POLYGON);
            gl.glVertex2f((float)(x+side*Math.cos(theta)), (float)(y+side*Math.sin(theta)));
            gl.glVertex2f((float)(x+side*Math.cos(theta+Math.PI/2)), (float)(y+side*Math.sin(theta+Math.PI/2)));
            gl.glVertex2f((float)(x+side*Math.cos(theta+Math.PI)), (float)(y+side*Math.sin(theta+Math.PI)));
            gl.glVertex2f((float)(x+side*Math.cos(theta+3*Math.PI/2)), (float)(y+side*Math.sin(theta+3*Math.PI/2)));
        gl.glEnd();
}
```

To make this box rotate we only need to add an increment to theta into the **display( )** method. The way I usually organize the code, **display( )** is

```
public void display(GLAutoDrawable drawable) {
        update();
        render();
}
```

update( ) is

```
public void update( ) {
        theta += increment;  // increment is adjusted by a speed slider
}
```

and **render( )** is

```
public void render( ) {
        drawBox( );
}
```