3D OpenGL

The View Pipeline.

Basic 3D programming in OpenGL is just as easy as 2D. We use 2 functions to setup the display pipeline: gluPerspective( ) and gluLookAt( ). Like the call to gluOrtho2D, the gluPerspective call should modify the PROJECTION matrix. The gluLookAt( ) call sets up the view matrix and should be the first thing in the MODELVIEW matrix. So we set up the pipeline with the following sequence of calls:

```
gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();
glu.gluPerspective( ... );
gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();
glu.gluLookAt( ... )
```

Here are the arguments for these calls.

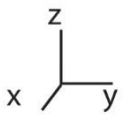
gluPerspective(fov, aspect, hither, yon)

fov (field of view) is the vertical angle of view (twice our theta).
aspect is the aspect ratio (width to heigh) of the viewport
hither and yon are our clipping planes

gluLookAt(vx, vy, vz, atx, aty, atz, upx, upy, upz)

(vx,vy, vz) is the viewer's position
(atx, aty, atz) is the lookat point
<upx, upy, upz> is the up-vector

This is easy to use and quite similar to what you did for HW2. Be careful with the up-vector. Unless you are doing a flight simulation, you probably want the up-vector to point in the same direction as the vertical world-coordinate axis. I like to imagine the world like this:



and so I almost always use <0,0,1> as the up-vector.

If you want an otthogonal projection rather than a perspective projection, there is a 3D analog to the gluOrtho2d( ) method we used before. This time it is just called glOrtho( ). The arguments are

gl.glOrtho(left, right, bottom, top, hither,yon)

This goes in the PROJECTION matrix, so you do it after a calls to

```
gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();
```

We won't be using glOrtho( ) much;

Here is a portion of a typical init( ) method:

```
gl = drawable.getGL().getGL2();
glu = new GLU();
gl.glMatrixMode(GL2.GL_PROJECTION);
gl.glLoadIdentity();
glu.gluPerspective(60, 1, 0.5f, 200);
gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();
glu.gluLookAt(10, 2, 0, 0, 0, 0, 0, 0, 1);
```

This has the viewer at (10, 2, 0), looking at the origin, with a 60 degree aperture of vision

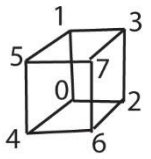
Drawing Commands

Drawing works the same in 3D as in 2D; we just need to be sure to give 3D coordinates for polygon vertices, by using glVertex3f( ) and glVertex3fv( ) in place of glVertex2f( ) and glVertex2fv( ). glVertex3f( ) takes 3 arguments for the x, y, and z coordinates. glVertex3fv( ) takes 2 arguments: an array of floats with the x, y, and z coordinates, and an "offset" for where to start in this array, which is usually 0. Here is a chunk of code that draws a rectangle in the y-z place (all points have x=0) with 0 <= y <= 10, 0<=z<=10:

```
gl.glBegin(GL2.GL_POLYGON);
        gl.glVertex3f(0, 0, 0);
        gl.glVertex3f(0, 10, 0);
        gl.glVertex3f(0, 10, 10);
        gl.glVertex3f(0, 0, 10);
gl.glEnd();
```

If you want to draw a polyhedron you might start by make an array of all of the vertices, then using indexes into this array to refer to vertices of particular faces. For example, here are the vertices of a cube:

```
float vertices[][] = new float [][] {
        {-1,-1,-1}, {-1, -1, 1}, {-1, 1, -1}, {-1, 1, 1},
        {1, -1, -1},{1, -1,1}, {1, 1, -1}, {1, 1,1}
};
```



and here is a pair of drawing methods that make use of this array

```
private void polygon( int a, int b, int c, int d) {
        gl.glBegin(GL2.GL_POLYGON);
                gl.glVertex3fv(vertices[a], 0);
                gl.glVertex3fv(vertices[b], 0);
                gl.glVertex3fv(vertices[c], 0);
                gl.glVertex3fv(vertices[d], 0);
        gl.glEnd();
}

private void drawCube( ):
        gl.glColor3f(1, 0, 0); //RED
        polygon(4,6,7,5);
        gl.glColor3f(0, 1, 0);    //GREEN
        polygon(6,2,3,7);
        gl.glColor3f(0, 0, 1);   //BLUE
        polygon(5, 7, 3, 1);
        gl.glColor3f(0, 1, 1);   //CYAN
        polygon(1,3,2,0);
        gl.glColor3f(1, 0, 1); // MAGENTA
        polygon(0, 4, 5, 1)
        gl.glColor3f(1, 1, 0);   // YELLOW
        polygon(4,0,2,6);
}
```

Of course, once you start drawing more than a single polygon, you need to consider the hidden surface problem. You can tell openGL to use a Z-buffer with

```
gl.glEnable(GL2.GL_DEPTH_TEST);
```

This usually goes into the init( ) method. In addition, you need to clear the Z-buffer each time you redraw the scene. The following goes at the start of your display( ) method, or whatever this method calls to draw the scene:

gl.glClear(GL2.*GL_COLOR_BUFFER_BIT* | GL2.*GL_DEPTH_BUFFER_BIT*);

There is one additional drawing feature. We can define various quadric surfaces (solutions to $Ax^2+By^2+Cz^2+D=0$) For this you need to make a quadric object:

GLUquadric quad = glu.gluNewQuadric( );

We can then use quad to draw various quadric surfaces

gluSphere(quadricObject, radius, slices, stacks);

This makes a sphere centered at the origin. "Slices" are lines of longitude, from pole to pole. "Stacks" are lines of latitude, like the equataor.

gluCylinder(quadricObject, baseRadius, topRadius, height, slices, stacks);

This makes a cylinder aligned with the z-axis, whose base is in the x-y plane. The base and top radii can be different, so this can make the frustum of a cone. "Stacks" are circles around the z-axis, slices are vertical lines down the surface.

gluDisk(quadricObjecdt, innerRadius, outerRadius, slices, rings);

This makes a disk with a hole in it (like a washer) unless you set the innerRadius to 0. It sits in the x-y plane centered on the z-axis. "slices" are vertical slices through it, "rings" are circles around it.

gluPartialDisk(quadricObjecdt, innerRadius, outerRadius, slices, rings, start, angle);

This is just like gluDisk, only the last two arguments specify a pie-shaped wedge (starting at angle start, extending angle degrees), that is removed fromthe disk. This is here to make for an easy implementation of Pacman.

Until we talk about lighting models you can't really use quadric objects, because when flat-shaded they look like blobs. This seemed like a convenient place to introduce them.

Transformations

OpenGL coding makes a lot of use of transformations, for both design and evaluation. All of the work we will do with transformations will affect the MODELVIEW matrix, so they should come after you have executed

gl.glMatrixMode(GL2.*GL_MODELVIEW*);

Every time you call one of openGL's transfomation methods, such as glTranslatef( ) or glRotatef( ), the appropriate matrix is generated and multiplied times the current matrix (which should be the MODELVIEW matrix). This multiplication is done so that when we create geometry (perhaps with glVertex3f) the most recent transformation called is the one first applied to the new geometry. The MODELVIEW matrix should always start with the View transformation generated by the call to gluLookAt( ), and then it is built up by successive transformations. You may want to have some transformations that only apply to specific objects. Because first multiplying the MODELVIEW matrix by T, generating the appropriate geomery, and then multiplying by T^{-1} to remove T and then proceeding is both slow and awkward, OpenGL provides another way to accomplish the same thing. The system maintains a stack of the current (MODELVIEW or PROJECTION) matrix values. If you are in MODELVIEW matrix mode, as you should be when generating geometry,

gl.glPushMatrix( )

pushes a copy of the current MODELVIEW matrix onto the stack; similarly

gl.glPopMatrix( )

pops the top of the current matrix stack into the MODELVIEW matrix. Thus, to apply a transformation to some new geometry we do the following

```
gl.glPushMatrix( );
<code to generate T>
<code for the new geometry to be affected by T>
gl.glPopMatrix();
```

The three transformations we will use are

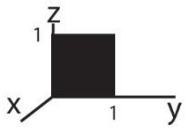
gl.glTranslatef(dx, dy, dz)
: which generates a matrix for translating by <dx, dy, dz> and multiplies it times the current matrix.

gl.glRotatef(angle, dx, dy, dz)
: which generates a matrix for rotating about the line through the origin parallel to <dx, dy, dz>, rotating around angle degrees. As with the other transformations, this matrix is multiplied times the current matrix.

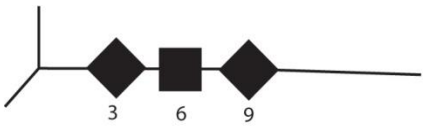
gl.gScale(dx, dy, dz)

which scales by <dx, dy, dz> and multiplies this matrix times the current matrix

For example, suppose we have a function box( ) which draws the following box in the y-z plane:



I want to make the following figure:



We get this with

```
gl.glPushMatrix();
        gl.glTranslatef(0, 3, 0);
        gl.glRotatef(45, 1,0, 0);
        gl.glTranslatef(0, -0.5f, -0.5f);
        box( )
gl.glPopMatrix( )

gl.glPushMatrix();
        gl.glTranslatef(0, 6, 0);
        gl.glTranslatef(0, -0.5f, -0.5f);
        box( )
gl.glPopMatrix( )

gl.glPushMatrix();
        gl.glTranslatef(0, 9, 0);
        gl.glRotatef(45, 1,0, 0);
        gl.glTranslatef(0, -0.5f, -0.5f);
        box( )
gl.glPopMatrix( )
```

Hierarchical Modeling

Many modeling situations have parts made up of sub-parts that move somewhat independently, while remaining attached. I can bend my arm at the wrist, elbow or shoulder and all of the pieces stay together. OpenGL's transformations are designed to make this kind of modeling easy. Consider the following picture:

Initially this consists of three horizontal bars, the red bar defined from 1 to 2 on the y axis, green bar from 2 to 3, and the blue bar from 3 to 4. Here is the code that displays it:

```
gl.glColor3f( 1,0,0);            // RED
gl.glTranslatef(0, 1, 0.05f);   // back to where the bar started
gl.glRotatef(redTheta, 1, 0, 0);
gl.glTranslatef(0, 1, -0.05f);  //puts the left edge at origin
<draw the red bar>

gl.glColor3f( 0, 1, 0);          //GREEN
gl.glTranslatef(0, 2, 0.05f);    // back to where the bar started
gl.glRotatef(greenTheta, 1, 0, 0);
gl.glTranslatef(0, -2, -0.05f);   // puts the left edge at origin
<draw the green bar>

gl.glColor3f(0, 0, 1);                  // BLUE
gl.glTranslatef(0, 3, 0.05f);  // back to where the bar started
gl.glRotatef(blueTheta, 1, 0, 0);
gl.glTranslatef(0, -3, -0.05f);   // puts the left edge at the origin
<draw the blue bar>
```

Note that there are no pushes and pops in this code. For the green bar, we translate it to the origin, rotate it, then translate it back, and then we apply the same transformations that are applied to the red bar. This is why it stays connected – the left edge of the green bar ends up at the same location as the right edge of the red bar. The same analysis can be applied to the blue and green bars.