OpenGL Notes [a]

Stu Pomerantz
smp@psc.edu
`http://www.psc.edu/~smp`

June 6, 2007

[a] Most material is adapted from: OpenGL ARB, et. al, "The OpenGL Programming Guide", Third Ed., Reading: Addison-Wesley, 1999

OpenGL Buffers Revisited

- The OpenGL color buffer is where the final image resides. The contents of the color buffer are displayed on the screen.
- Recall that the animation cycle is:
  clear the color buffer $\rightarrow$ update $\rightarrow$ redraw
  (update and redraw may interleave)
- What happens if the animation cycle can't be completed in less than $\frac{1}{30}th$ second ? **flicker**.
- This problem is avoided by *double buffering*. That is, display the 'front' color buffer while drawing into the 'back' color buffer. When the drawing is complete, then swap the front and back color buffers.

OpenGL Buffers Revisited

- Using GLUT request a graphics context with a color buffer and double buffering:
  `glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)`
- At the end of Display() call:
  `glutSwapBuffers()`
- Since the buffer swap is fast, the result is flicker free animation.
- The buffer swap can also be synchronized to the vertical refresh of a monitor. This is generally available as a video driver option.

OpenGL Buffers Revisited

- The depth buffer, a.k.a the z-buffer, stores a depth value for each pixel.
- Depth is usually measured in terms of distance from the eye.
- Pixels with larger depth values are overwritten by pixels with smaller depth values.
- Depth buffering is how near surfaces occlude far surfaces.
- The depth buffer has finite precision. It has more precision near the eye than far from the eye. Setting the limits of the depth buffer excessively large will can cause *depth fighting* at distances far from the eye. That is, objects appear to pass through each other.

OpenGL Buffers Revisited

- Using GLUT request a graphics context with a color buffer, double buffering and a depth buffer:
  `glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH)`
- At the beginning of Display() call:
  `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`
- The function which sets the limits of the depth buffer will be discussed in in a moment.
- There are still more buffers that GL keeps. We may discuss them later.

Simple OpenGL Lighting

- `glEnable(GL_LIGHTING)` 'Turns on the power'.
  You will see the glEnable(*feature*) / glDisable(*feature*) pattern often.
- `glEnable(GL_LIGHT0)` Flips the light switch on light0.
- The number of lights available is implementation defined. Find out using:
  `glGetIntegerv(GL_MAX_LIGHTS, &maxlights)`
  Usually at least 8.
- Lights have many properties. A few are discussed here.

Simple OpenGL Lighting

This code:

```
float lightpos[4] ;
glLightfv(GL_LIGHT0, GL_POSITION, lightpos) ;
```

Sets the position of light0.

- If lightpos[3] == 0 then lightpos is interpreted as a direction and the light is assumed to be infinitely far away, like the sun. A ‘directional’ light.
- If lightpos[3] == 1 then lightpos is interpreted as a point and the position of the light is accounted for in the lighting calculation, like a desk lamp. A ’positional’ light.

Simple OpenGL Lighting

- glLightfv(*light*, *parameter*, *parameter value*) is used to specify many properties of a light.
- The `GL_AMBIENT` parameter sets the ambient RGBA intensity of the light.
- The `GL_DIFFUSE` parameter sets the diffuse RGBA intensity of the light.
- The `GL_SPECULAR` parameter sets the specular RGBA intensity of the light.

Simple OpenGL Lighting

- `glShadeModel(GL_FLAT)` uses a single surface normal for a primitive to perform lighting calculations for the whole primitive.
- `glShadeModel(GL_SMOOTH)` uses the vertex normals for a primitive to perform lighting calculations. The computed colors at each vertex are interpolated across the primitive.

Simple Material Properties

- Just as properties can be specified for lights, the can be specified for primitives.
- `glColorMaterial()` is fast and efficient way for quickly specifying material properties.
- glColorMaterial() will cause the ambient, diffuse, and specular components of a primitive (which are specified with `glMaterial()`) to track the current color.
- In general, glColorMaterial() 'does the right thing' .
- Remember to `glEnable(GL_COLOR_MATERIAL)`

OpenGL Matrix Modes

- Matrices are used to transform primitives or *models*.
- Without loss of generality, assuming a uniform background, transforming a primitive two units to the left is equivalent to transforming the eye, or camera, two units to the right.
- `glMatrixMode(GL_MODELVIEW)` is where programs spend their time. The camera viewpoint is specified in this mode and primitive transformations are specified while in this mode.

OpenGL Matrix Modes

- To specify the camera, 3 parameters are needed: A position for the camera, a point the camera is looking at, and up direction for the camera.

```
 // set the matrix mode to modelview
glMatrixMode(GL_MODELVIEW) ;
 // set the identity transformation for the modelview
glLoadIdentity() ;
// calculate a transformation matrix for this camera
// and multiply it into the modelview
gluLookAt(eyeX, eyeY, eyeZ,
          centerX, centerY, centerZ,
          upX, upY, upZ ) ;
```

OpenGL Matrix Modes

- Matrices are used to specify a projection transformation.
- Two kinds of projection transformations are directly supported by OpenGL, perspective and orthographic.
- `glMatrixMode(GL_PROJECTION)` is used to specify the projection, usually called only during a reshape event.

OpenGL Matrix Modes

- To specify a perspective projection, 4 parameters are needed: A field of view for the camera, an aspect ratio, the near and far clipping planes.

```
glMatrixMode(GL_PROJECTION) ;
glLoadIdentity() ;
gluPerspective( fovy, aspect, zNear, zFar ) ;
```

- aspect is usually width/height of the viewport.
- zNear & zFar are the limits of the near and far clipping planes and affect the precision of the depth buffer.

OpenGL Transformations

- The following transformation functions multiply the current transformation environment (**almost always `glMatrixMode(GL_MODELVIEW)`**) to transform primitives.

OpenGL Transformations

- `glTranslatef(x,y,z)` multiply in a translation matrix.
- `glRotatef(deg,x,y,z)` multiply in a rotation matrix that rotates *deg* degrees about the axis specified by x,y,z
- `glScalef(sx,sy,sz)` multiply in a scaling matrix that scales the x coordinates of vertices by sx, and similar for corresponding y and z coordinates.

OpenGL Transformations

- `glPushMatrix()` pushes the current matrix stack down by one, duplicating the current matrix.
- `glPopMatrix()` pops the current matrix stack, replacing the current matrix with the one below it on the stack.
- The matrix stack depth is at least 32.
- In practice this allows the creation of temporary transformation state on top of global transformation state.

OpenGL Transformations

For example, suppose a function that draws a point at the origin called draw_origin(). Then,

```
draw_origin() ; // point at 0,0,0
glPushMatrix() ;
  glTranslatef(1,0,0) ;
  draw_origin() ; // point at 1,0,0
  glPushMatrix() ;
    glTranslatef(1,0,0) ;
    draw_origin() ; // point at 2,0,0
  glPopMatrix() ;
glPopMatrix() ;
draw_origin() ; // point at 0,0,0
```

OpenGL Display Lists

- Display lists store primitive specifications in memory *on the graphics card*.
  - Large surfaces need only be sent across the bus once.
  - A small surface, drawn many times per frame does not have to be transmitted across the bus again and again.
- Once the display list is created, a single function call will draw all its primitives.
- Display lists are an efficiency mechanism, they save bus bandwidth and cpu time at the expense of graphics card memory.

OpenGL Display Lists

After the graphics context is initialized:

```
// variable to hold my display list identifier
GLuint mylist ;
// ask GL to create the display list identifier
mylist = glGenLists(1) ;
// tell GL you're starting to specify primitives
// for this list
glNewList(mylist, GL_COMPILE) ;
/* draw primitives here in the usual way */
// tell GL you're done specifying primitives
glEndList() ;
```

OpenGL Display Lists

for example:

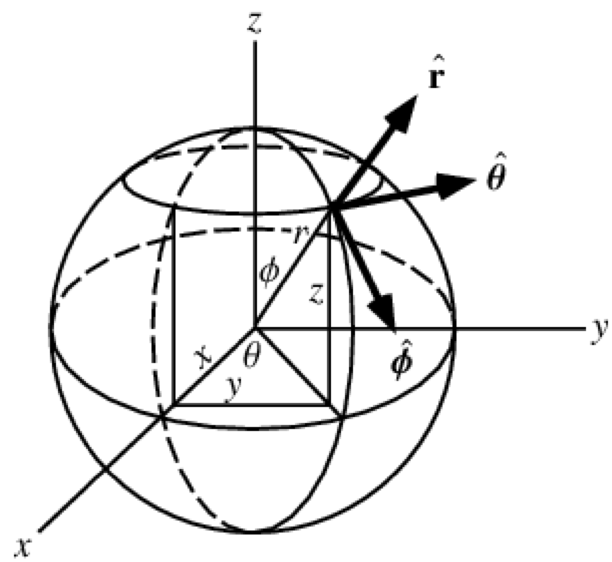
```
GLuint mylist ;
mylist = glGenLists(1) ;
glNewList(mylist, GL_COMPILE) ;
 glBegin(GL_LINES) ;
  glVertex3f(0,0,0) ;
  glVertex3f(1,1,1) ;
  .
  .
  .
 glEnd() ;
glEndList() ;
```

OpenGL Display Lists

Now, somewhere in display() do:

```
glCallList(mylist) ;
```

Spherical Coordinates



$$x = r\cos(\theta)\sin(\phi) \quad r = \sqrt{x^2 + y^2 + z^2}$$

$$y = r\sin(\theta)\sin(\phi) \quad \theta = \arctan(\tfrac{y}{x})$$

$$z = r\cos(\phi) \qquad \phi = \arccos(\tfrac{z}{r})$$