Basics of Open GL

Hans de Nivelle

November 10, 2010

You should read the Red Book, `http://wwww.glprogramming.com/red/`, but it is very imprecise about the basic principles, so I explain these in this text.

1 Basic Drawing

1.1 Visibility Cube

The internal coordinate system of openGL is specified as follows:

X to the right.

Y in upward direction.

Z away from the viewer.

This means that the default coordinate system of openGL is *left handed*! It is possible to draw directly in this coordinate system, but you should not do it, unless for 2D graphics.

A point (x, y, z) is visible if it lies inside the cube defined by:

$$-1 < x < 1, \quad -1 < y < 1, \quad -1 < z < 1.$$

Lines and surfaces that are partially outside of this cube, are *clipped* in a nice way. For example, if you draw a line from $(0.8; 0.8; -1.1)$ to $(-1.2; 0.9; 0.5)$, the part that is inside the cube will be drawn.

If a point lies within the cube, then its position on the screen is obtained by throwing away the Z-coordinate. The point $(0, 0)$ is in the middle of the window. The point $(1, 0)$ is to the right of the window (just out of scope). The point $(0, 1)$ is on the top of the window (just out of scope). If you want to project objects with different coordinates, (which you usually want), you will have to define translations, rotations, and projections that bring the objects into the cube.

1.2 Depth Test

In real life, things are often in front of each other, and objects that are behind another object cannot be seen at all, or are partially hidden. In OpenGL, this is

modelled through the *depth test*. The graphic memory does not only remember the color of a point, but it also remembers the depth of this point. The depth is obtained by rounding the Z-coordinate, when a point (x, y, z) is written. (The precision depends on the graphical card, and the settings.) By default, the depth test is switched off. In order to switch it on, give the command

```
glEnable( GL_DEPTH_TEST );
```

When it is switched on, a write at (x, y, z) will only take place, if the current remembered depth z' has $z' > z$. This behaviour can be modified by using `glDepthFunc( )`, but you should not do that.

It should be noted that the default depth test, (smaller Z-coordinate is closer to the viewer) implies that the default coordinate system of OpenGL is left handed.

If the depth test is enabled, then it is important that the graphic buffer is cleared at the right depth before writing begins.

```
glClearColor( 0.0, 0.0, 0.0, 0.0 );   // Black
glClearDepth( 1.0 );                  // Backside of cube.
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

1.3 Polygon Offset

Suppose that you want to draw a polygon on top of another polygon, for example a decoration on a wall. One can first draw the wall, and after that the calculation, and set the depth test to `GL_LEQUAL`. Unfortunately, there can be small inaccuracies in the calculation of the Z-coordinate, and it can happen that the polygon that is supposed to be in front, is still partially hidden by the other polygon. In order to avoid that, one can artificially move one of the polygons a little bit forward:

```
(Draw the first polygon, which will be partially
 covered by the second.)
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset( -1.0, -1.0 );
   // Works in most cases. If it doesn't work, see
   // Chapter 6 of the Red Book.

   (Draw the polygon that has to be front)
glDisable(GL_POLYGON_OFFSET_FILL);
```

1.4 Culling

Culling is a convenient technique for removing hidden surfaces. The back side of a house is invisible because it is behind the front of the house. This implies that the depth test will remove the back of the house when it is drawn. But there exists an easier way: If one assigns an orientation to each surface, (call

them *front* and *back*), then one can make the surface transparant from the back side, and visible from the front side. If one specifies for each surface of the house that the front is outside, and the back is inside, then the hidden surfaces of the house will be invisible because they turn their back side to the user. This test is very easy to implement, and it is a lot cheaper than the depth test. The depth test has to be done for each point separately, while the culling test needs to be done only once for each polygon. Culling is selected as follows:

```
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );  // Specifies that back facing is invisible.
```

By default, a polygon is front facing if it can be drawn counter clock wise. This setting can be changed, but you should not do that. Technically, it can be tested very easy (we assume that the polygon is convex.) from which side we see the polygon. The polygon has form $(p1, p_2, \ldots, p_n)$, where $p_1, p_2, \ldots, p_n$ are the points in the polygon. First find three consecutive points that are not on the same line. If no such points exist, then the polygon is a line and invisible. Without loss of generality, assume the points are p_1, p_2, p_3. Then compute the dot product $(p_2 - p_1) \times (p_3 - p_2)$. If it has positive Z-coordinate, then the polygon is front facing. (This test is done after all transformations are completed, in cube coordinates.)

2 Translations, Rotations and Scalings

Translations, rotations and scalings are obtained as follows:

```
glTranslatef( x,y, z)
   // Add a translation to the selected matrix.
glRotatef( angle, x, y, z )
   // Add a rotation to the selected matrix.
glScalef( x, y, z )
   // Add a scaling to the selected matrix.
```

The selected matrix should be normally the `MODELVIEW` matrix.

3 Projections

There are two types of projections in openGL, *orthogonal projections* and *frustum projections*.

Orthogonal Projection An orthogonal projection is set by the command `glOrtho( )`. The default behaviour of openGL is to simply throw away the Z-coordinate, which is already an orthogonal projection. Therefore, the `glOrtho( )` command does almost nothing.

```
glOrtho( X0, X1, Y0, Y1, Z0, Z1 )
```

Normally, $X0 < X1$, $Y0 < Y1$, and $Z0 < Z1$. The projection applies a linear transformation, such that point (x_0, y_0, z_0) is mapped to $(-1, -1, 1)$, and the point (x_1, y_1, z_1) is mapped to $(1, 1, -1)$. Note that $Z0$ is mapped to 1, and $Z1$ is mapped to -1. This means that `glOrtho` will normally mirror the orientation of the coordinate system. This is very important because the user uses (should use!) a right handed coordinate system, but the depth test assumes that a smaller Z coordinate is closer to the viewer, which requires a left handed coordinate system.

Frustum Projection A frustum projection defines a perspective projection. It is set by the command `glFrustum( X0, X1, Y0, Y1, Z0, Z1 )`. Normally $X0 < X1$, $Y0 < Y1$, and $Z0 < Z1$. It maps as follows:

```
( -X0, -Y0, -Z0 )  ==>  ( -1, -1, 1 ),
( X0, -Y0, -Z0 )   ==>  ( 1, -1, 1 ),
( X0, Y0, -Z0 )    ==> ( 1, 1, 1 ),
( -X0, Y0, -Z0 )   ==> ( -1, 1, 1 ),

Let F = Z1 / Z0.

( -X0 * F, -Y0 * F, -Z1 )   ==>  ( -1, -1, -1 )
( X0 * F, -Y0 * F, -Z1 )    ==>  ( 1, -1, -1 )
( X0 * F, Y0 * F, -Z1 )   ==>  ( 1, 1, -1 )
( -X0 * F, Y0 * F, -Z1 )  ==>  ( -1, 1, -1 )
```

It is pretty hard to understand what `glFrustum( )` actually does:

- Like `glOrtho( )`, it mirrors the orientation of the coordinate system.
- It carries out a perspective projection, assuming that the viewer is situated in $(0, 0, 0)$. Points (x_0, y_0, z_0) and (x_1, y_1, z_1), for which

  $$x_0 z_1 = x_1 z_0 \text{ and } y_0 z_1 = y_1 z_0,$$

  receive the same Z-coordinate after the projection, so that that they will be drawn at the same position. (If they will be drawn.) The values of `Z0` and `Z1` have no influence on the perspective projection, but Z0 has influence on the scaling, because X0,X1,Y0,Y1 define a rectangle at distance -Z0.

  They only have influence on the front and backside of the visibility cube.

In OpenGL, all operations (translations, rotations, projections, mirrorings) are represented by 4×4-matrices. For practical usage, there is no need to understand this, but it is very nice, and I will explain it in class.

The system maintains two transformations, called `MODELVIEW` and `PROJECTION`. If you write a point p somewhere, it will be always transformed into $P(M(p))$,

where M is `MODELVIEW`, and P `PROJECTION`. Mathematically, there is no distinction between P and M, and you can put any transformation into either of them. In practice, one puts the perpective projection in P, and the rest in M, because of practical considerations: One often wants to change M, (in order to see the object from different sides), without changing P. Defining a transformation always means that the existing transformation is multiplied with the new transformation. If sets P to T, then what actually happens is $P := P \times T$, so that the transformation $P(M(p))$ is replaced by $P(T(M(p)))$.

```
glMatrixMode( GL_MODELVIEW ); // Specifies the transformation
                              // we want to change.
glLoadIdentity( );            // Sets selected transformation
                              // to Identity.
glFrustum( -1,1,-1,1,1,2 );   // Right multiplies MODELVIEW.
                              // (normally, this would be GL_PROJECTION)
                              // with Frustum projection.
```

4 Complete Drawing Procedure

As far as I can see, the drawing algorithm of openGL works as follows:

1. Apply the the MODELVIEW matrix.
2. Now the object is assumed to be in *eye coordinates*. This means that

   X is to the right of the viewer.

   Y is upwards, seen from the viewer.

   Z is towards and behind the viewer.

   The eye coordinate system is still right handed. If we want to do a Frustum projection, then the object should be placed in such a way that all vertices have negative Z-coordinate. (Otherwise, the perpective projection will swap orientations, or divide by zero.)
3. Apply the PROJECTION matrix. The projection matrix transforms the visible part of the world (expressed in eye coordinates) into the cube mentioned before. Because the projection has mirrored Z-coordinates, the coordinate system is now left handed.
4. If Culling is selected, then this is the moment at which it is decided which polygons are culled, by doing the cross product test.
5. Object that are outside of the visibility cube are clipped. When a polygon offet is set, it is added. The object is drawn, and the depth test is used to decide which pixels of the surface being drawn can overwrite existing pixels.

5 Lights and Colors

Lighting in openGL is in principle easy. The main reason why it is difficult to understand, is the fact that it is not in correspondence with physical reality. Once you have accepted this idea, the model is easy.

5.1 Light

In physical reality, light comes out of a light source, after which it bounces against several objects, until it eventually reaches your eye. Only light that has travelled such a trajectory is visible. In openGL, things are different:

ambient light When a light source is switched on, all objects in the room get a little bit lighter. In physical reality, all extra light can be traced back to the light source, but openGL does not do that. The ambient light of a light source, is the light that makes all objects a little bit lighter.

diffuse light When light falls on an object, it gets reflected in different directions. Some objects reflect very accurately in one direction (mirrors, clean cars) other objects don't. (the ground, textile) In physical reality, the object determines how the light is reflected. In OpenGL, a light source has a special kind of light, the diffuse light, that will be reflected equally in all directions, once it has hit the object. Precisely: The diffuse light shines on the object. Dependent on the angle, the object gets a little bit lighter. After that, the object looks equally bright (and with the same colour) in all directions.

specular light This is a special kind of light that gets reflected very accurately by the object. It will create shiny spots on the object, that are the reflections of the light source.

5.2 Material

In consistence with the lighting model, materials have three colours, and in addition, they can emit light by themselves. This results material having four colours:

ambient colour The ambient colour specifies how the object reacts to ambient light.

diffuse colour The diffuse colour specifies how the object gets brighter (in all directions) when a light shines on it directly. How much the object gets brighter depends on the strength of the light source, its distance, the angle under which the light strikes the object, and the diffuse colour of the object. Once its brightness is determined, it shines equally in all directions.

specular colour The specular colour determines how the material can produce mirror images of the light sources that shine on it. In a clean red car, you can see a reflection of the sun, but the reflection looks red.

emitted colour Some objects emit light by themselves. The viewer can see this light, but this light does not make other objects brighter. (Think of a far away street lamp, or a star. The moon is already too bright for this.)

I will not list all the commands related to lights and colour. They are described in Chapter 5 of the Red Book.