OpenGL Transformations

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

02/18/2014
The most essential aspect of OpenGL is the vertex pipeline described in Chapter 3 of the Redbook. Objects are represented by a set of vertices in user-defined object coordinates, and these vertices are converted to window coordinates by a sequence of four transformations (state information). The primary goal of this course is to convey an understanding of these transformations.

The transformations of interest in graphics are linear transformations (scaling, rotation, reflection, shear, and orthographic projection), affine transformations (linear transformations, along with translation), and projective transformations (central projection). These transformations can be understood at three levels.

1. The visual aspect of the geometry
2. The algebraic matrix representation and operations
3. The internal workings of the vertex pipeline
The focus of these notes is on the vertex pipeline:

Object Coordinates
  \[\rightarrow\] [Modelview Matrix] \[\rightarrow\]
Eye Coordinates
  \[\rightarrow\] [Projection Matrix] \[\rightarrow\]
Clip Coordinates
  \[\rightarrow\] [Perspective Division] \[\rightarrow\]
Normalized Device Coordinates
  \[\rightarrow\] [Viewport Transformation] \[\rightarrow\]
Window Coordinates and Depths
Figure 3-4 demonstrates that the order of operations is significant. We will use a right-handed coordinate system.
Figure 3-10 describes translation in a fixed coordinate system (left) versus a change of coordinates (right).
The modelview matrix represents a sequence of modeling transformations followed by a sequence of viewing transformations. The modeling transformations typically include scaling, rotation, and translation. A tricycle might be rendered by drawing a wheel (perhaps stored as a display list) three times with different scalings and translations. The parts of a robot arm might be constructed by scaling, rotating, and translating an axis-aligned cube centered at the origin.

In a 2-D application no viewing transformations are necessary. In a 3-D application with perspective projection, the eye position is at the origin, the viewing direction is the -z direction, and the projection plane is parallel to the x-y plane. Only vertices with negative z components will be visible. A typical viewing transformation therefore includes at least a translation in the negative z direction.
Figure 3-11 describes the default location and orientation of the eye position.
The projection plane is the near plane defined by \( z = -n \) for \( n > 0 \). There are two types of projection.

1. **Orthogonal projection** or (parallel) orthographic projection maps a point \( p \) to the nearest point in the projection plane by simply setting the z component to \( -n \).

2. **Perspective projection** involves central projection through the origin (the center of projection). A point \( p \) is mapped to the point of intersection of the line defined by \( p \) and the origin with the projection plane.

Perspective projection results in perspective foreshortening in which more distant lines project to shorter lines. Orthographic projection may be thought of as the limit of perspective projection as the eye position is moved to \((0, 0, \infty)\). This is inappropriate for rendering real (or realistic appearing) objects, but is cheaper and has the advantage that relative lengths are preserved — useful for CAD and architectural drawing.
Figure 3-13 depicts the six planes defining a frustum view volume.
Figure 3-14 depicts the parameters in a call to `gluPerspective(fovy, aspect, near, far)`. The aspect ratio is w/h.
Figure 3-19 shows that \( \tan(\theta/2) = (\text{size}/2)/\text{distance} \), where \( \theta = \text{fovy} \), \( \text{size} = t-b \), and \( \text{distance} = n \). Note that size does not scale linearly with \( \theta \), making zoom problematic. Also, the symmetry about the z axis eliminates the possibility of panning.
Figure 3-15 demonstrates that orthographic projection preserves relative sizes.
The term 'projection matrix' is a misnomer because no projection is actually applied. The z components (depths) must be retained in order to remove hidden lines and surfaces — determine which fragments are actually visible and may become pixels. (Fragment depths are obtained by linear interpolation of vertex depths.)

A view volume or viewing volume is a rectangular box in the case of orthographic projection, or a frustum (truncated pyramid) in the case of perspective projection. In either case, it is bounded by six planes which are defined by six numbers in eye coordinates: \( l, r, b, t, n, \) and \( f \), where the intersection of the view volume with the near plane is \([l, r] \times [b, t] \), and \( n \) and \( f \) are the distances in the negative z direction to the near and far planes. Thus, for example, the left plane of the frustum is defined by the three points \((0, 0, 0)\), \((l, b, -n)\), and \((l, t, -n)\). What about the right plane?
All geometric primitives are *clipped* against the boundary of the view volume, creating additional vertices in the case of lines, and additional vertices and lines in the case of polygons. Hence the term *clip coordinates*.

The projection matrix, along with perspective division in the case of a perspective projection, is the affine transformation that maps the view volume into the normalized device coordinate space $[-1, 1]^3$ (an axis-aligned box with side length 2 centered at the origin).
A *viewport* is a rectangular portion of a window (which may coincide with the entire display area of the screen — full-screen mode) in which all drawing occurs.

The *viewport transformation* is the unique affine transformation that maps the normalized device coordinate space to *window coordinates* and depths. Window coordinates are related to integer absolute device coordinates as follows: with resolution $n_x$ by $n_y$ the center of pixel $(i,j)$ has window coordinates $(i+.5,j+.5)$ for $0 \leq i \leq n_x - 1$ and $0 \leq j \leq n_y - 1$. Depths, by default, are in the interval $[0,1]$.

In OpenGL commands, the origin of the window coordinate system is at the lower left — the math convention. GLUT, however, uses the hardware-oriented convention in which the origin is at the upper left. Hence, mouse coordinates $(x, y)$ returned by a keyboard or mouse callback function must be converted to $(x + .5, h - y - .5)$, for window height $h$, for use by OpenGL.
Figure 3-18 depicts the z coordinate values associated with uniformly distributed depths. The depths associated with uniform z coordinates get closer together (and harder to distinguish) as distance increases \((z \text{ approaches } -f)\), especially when \(n\) is small.

\[
\text{depth} = \frac{fn}{f - nz} + \frac{f}{f - n} \quad \text{for } -f \leq z \leq -n.
\]
void glMatrixMode(GLenum mode); causes subsequent transformation commands to affect the matrix specified by mode, where mode = GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE.

void glLoadIdentity(void); initializes the currently modifiable matrix to the order-4 identity matrix.

void glTranslatef(TYPE x, TYPE y, TYPE z); right-multiplies the current matrix by a translation operator.

void glRotate(TYPE a, TYPE x, TYPE y, TYPE z); right-multiplies the current matrix by one that rotates through angle a (in degrees) CCW as viewed toward the orgin about the direction vector (x,y,z).

void glScale(TYPE x, TYPE y, TYPE z); right-multiplies the current matrix by one that scales and/or reflects in the axis directions.
void gluLookAt(GLdouble Ex, GLdouble Ey, GLdouble Ez, GLdouble Cx, GLdouble Cy, GLdouble Cz, GLdouble Ux, GLdouble Uy, GLdouble Uz);
right-multiplies the current matrix by a viewing transformation (translations and rotations) associated with eye position E, object point C, and up-vector (view-up direction) U. The default (no viewing transformation) is equivalent to calling gluLookAt with E = (0,0,0), C = (0,0,-100), and U = (0,1,0).

void glPushMatrix(void); duplicates the matrix at the top of the current stack (defined by glMatrixMode).

void glPopMatrix(void); pops the top of the current stack, discarding the matrix.
void glFrustum(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f); right-multiplies the current matrix by a 'projection' operator associated with a perspective projection.

void gluPerspective(GLdouble fovy, GLdouble a, GLdouble near, GLdouble far); right-multiplies the current matrix by a 'projection operator' associated with a symmetric frustum centered on the z axis with an angular field of view fovy in the y-z plane (0 to 180 degrees), and aspect ratio a.

void glOrtho(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f); right-multiplies the current matrix by a 'projection' operator associated with an orthographic projection.
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h); creates a viewport with lower left corner at window coordinates (x,y), width w, and height h. The default is the entire window.

int gluUnProject(GLdouble wx, GLdouble wy, GLdouble wz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *x, GLdouble *y, GLdouble *z); maps windows coordinates (wx,wy,wz) to object coordinates (x,y,z) by inverting the specified operators. The return value is GL_TRUE = 1 if successful, GL_FALSE = 0 if failure results from a non-invertible matrix or something.
Robot Arm code: robot.c

Function reshape:
```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -5);
```

```latex
\[ M = M_0 = T_{-5e3} \]
```

Function display:
```c
glPushMatrix();
glTranslatef(-1, 0, 0);
```

```latex
\[ M = T_{-5e3} T_{-e1} \]
```

```c
glRotatef(s, 0, 0, 1);
```

```latex
\[ M = T_{-5e3} T_{-e1} R_z(s) \]
```

```c
glTranslatef(1, 0, 0);
```

```latex
\[ M = M_1 = T_{-5e3} T_{-e1} R_z(s) T_{e1} \]
```

```c
glPushMatrix();
glScalef(2, .4, 1);
```

```latex
\[ M = M_1 S \]
```

```c
glutWireCube(1);
```

```latex
\text{Draw upper arm} \]
```

```c
glPopMatrix();
```

```latex
\text{Pop } M = M_1 \]
```

\[ M_0 \] is a viewing transformation applied to the entire arm.
\[ M_1 \] is a modeling transformation which rotates the entire arm through angle \( s \) about the shoulder.
Robot Arm continued

```c
// Robot arm transformations

glTranslatef(1, 0, 0);  // Translation in the x direction
M = M_1 T_{e_1}

glRotatef(e, 0, 0, 1);  // Rotation about the z-axis
M = M_1 T_{e_1} R_z(e)

M = M_2 = M_1 T_{e_1} R_z(e) T_{e_1}

Push M_2

M = M_2 S  // Scale the forearm

Draw forearm

Pop M = M_2

Pop M = M_0
```

`glutWireCube(1)` draws a wireframe cube with side length = 1 centered at the origin.

*M_2* is a modeling transformation which rotates the forearm through angle *e* about the elbow.

Push *M_2* and Pop *M_2* are only needed if a hand is added (after Pop *M_2*, before Pop *M_0*).

Additional viewing transformations (x and y axis rotations) should be placed before the modeling transformations but after Push *M_0*. 

---

R. J. Renka  OpenGL Transformations