

COMP3201 – Computer Graphics

2.3 Modelling Transformations in 3D

2.3.1 The Right-hand Rule

OpenGL uses a right-handed coordinate system. The right-hand rule makes it easy to remember the direction of positive and negative rotations.

To apply the right-hand rule, open your right hand and position it so that the thumb points along the positive part of the axis. Then curl your fingers to close over the axis. The direction of curl of your fingers is the direction of a positive rotation around that axis.

2.3.2 Transformation Matrices

The following transformation matrices can be applied either to all the vertices of the objects in the scene or to the coordinate system, before the objects are drawn. The following notes define the form that each of the different transformation matrices takes.

2.3.3 Translation in 3D

This is a straightforward extension of the 2D case. Now the point P has coordinates (P_x, P_y, P_z) and is to be translated to $Q = (Q_x, Q_y, Q_z)$. In homogeneous coordinates, we get

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}.$$

The OpenGL command for translation in 3D is **glTranslatef(T_x, T_y, T_z)** where T_x, T_y, T_z are floating point numbers.

2.3.4 Scaling in 3D

This is also straightforward, with just the inclusion of the scaling factor in the z -direction.

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}.$$

The OpenGL command for scaling in 3D is **glScalef(S_x, S_y, S_z)** where S_x, S_y, S_z are floating point numbers.

2.3.5 Rotation in 3D

When working in 3D, we must now specify the axis of rotation. As before, positive values of the rotation angle β correspond to rotation in a counter-clockwise direction, when looking inwards along the rotation axis.

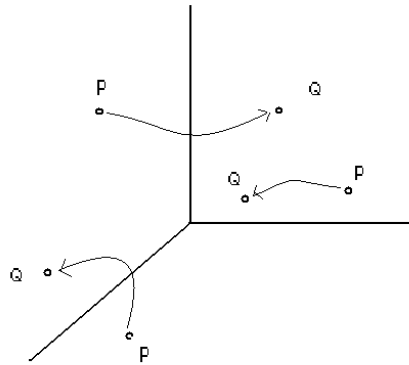


Figure 2.12

Using the notation R_x , R_y , and R_z to represent rotations about the x -, y - and z -axis respectively, the rotation transformation matrices are

$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \beta & -\sin \beta & 0 \\ 0 & \sin \beta & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad R_y(\beta) = \begin{pmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$R_z(\beta) = \begin{pmatrix} \cos \beta & -\sin \beta & 0 & 0 \\ \sin \beta & \cos \beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Note the position of the 1's and 0's – for R_x , these are in row 1 and column 1, while for R_y , row 2 and column 2 are zero except for a 1 in the (2,2) position; for R_z , row 3 and column 3 have 0's and 1's.

Also note the positioning of $-\sin \beta$ in R_y .

If a 90 degree rotation were performed about the z -axis, then the x -axis is mapped to the y -axis. Similarly, a 90 degree rotation about the y -axis maps the z -axis onto the x -axis.

The OpenGL command for rotation in 3D is **glRotatef(angle, x, y, z)** where x , y and z are floating point numbers representing the axis of rotation, and **angle** is specified in degrees.

2.3.6 Shear in 3D

There are several different shear matrices for applying the shear transformation in 3D. This is because it is possible to use different combinations of shear parameters and shear directions. The ones we will usually come across are:

shear in x and y along the z -axis

$$SH_{xy} = \begin{pmatrix} 1 & 0 & S_{xz} & 0 \\ 0 & 1 & S_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

shear in x and z along the y -axis

$$SH_{xz} = \begin{pmatrix} 1 & S_{xy} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & S_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

shear in y and z along the x -axis

$$SH_{yz} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ S_{yx} & 1 & 0 & 0 \\ S_{zx} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For example, we can see that a shear in x and y along the z -axis updates the x and y components with an amount proportional to the z -component while leaving z unchanged:

$$SH_{xy} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & S_{xz} & 0 \\ 0 & 1 & S_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + S_{xz}z \\ y + S_{yz}z \\ z \\ 1 \end{pmatrix}$$

It is also possible to construct shear matrices to shear the x component in both the y and z directions, the y component in both the x and z directions, and the z component in both the x and y directions.

2.3.7 Concatenation of 3D Affine Transformations

Similarly to the 2D case, multiple transformations can be concatenated into a single transformation matrix.

This technique is required, for example, if you wish to rotate an object about an arbitrary axis. The first step requires two rotations to align the arbitrary axis with the z -axis; then rotation about the z -axis is applied, and finally the first two rotations must be un-done, so that the points are in the correct positions.

2.3.8 Coordinate Transformations

So far we have considered transformations as applying to the points that define an object.

An alternative approach is to consider that the transformations are applied to the coordinate system and that the object is then re-drawn using its original coordinates but now with respect to the transformed coordinate system.

The following 2D example illustrates what is happening with these two approaches.

Consider the square with coordinates $(0, 0)$, $(0, 2)$, $(2, 2)$, $(2, 0)$ and suppose it is mapped according to Figure 2.13:

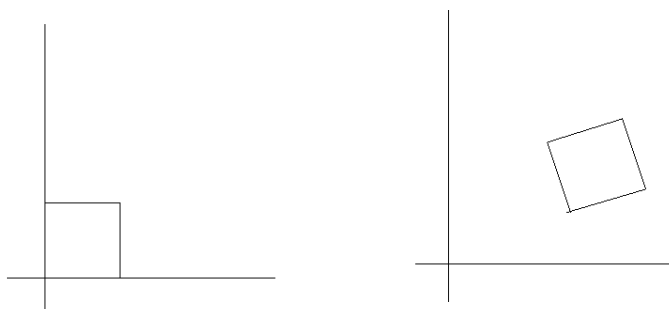


Figure 2.13

The new coordinates of the (mapped) square are

$$(4, 4), (3, 4 + \sqrt{3}), (3 + \sqrt{3}, 5 + \sqrt{3}), (4 + \sqrt{3}, 5)$$

and the required transformations were

- (i) rotate 30 degrees about the origin
- (ii) translate 4 in each of the x - and y - directions.

Applying the transformations to the coordinates of the square requires the application of the rotation matrix

$$R = \begin{pmatrix} \cos 30 & -\sin 30 & 0 \\ \sin 30 & \cos 30 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

followed by the translation

$$T = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix}$$

Mathematically, the new coordinates are

$$\begin{pmatrix} 4 & 3 & 3+\sqrt{3} & 4+\sqrt{3} \\ 4 & 4+\sqrt{3} & 5+\sqrt{3} & 5 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 2 & 2 \\ 0 & 2 & 2 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

In other words, Q_{new} (the new set of vertices for the transformed square) is obtained from Q_{orig} by pre-multiplication of the vertices:

$$Q_{new} = (T(R Q_{orig})).$$

Alternatively, if we transform the coordinate system so that we can then draw the square using vertices Q_{orig} , then effectively we translate the coordinate system first and then rotate it, and then do the drawing. So the original coordinate system is post-multiplied by T and then R before drawing takes place (using Q_{orig}). Thus, as above, $Q_{new} = (TR) Q_{orig}$.

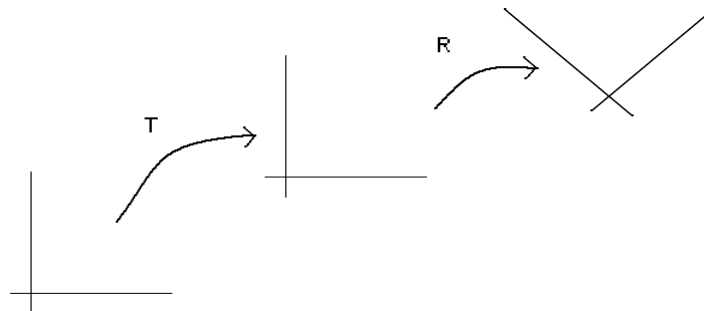


Figure 2.14

2.3.9 OpenGL Transformation Matrices (Model-View)

There are two transformation matrices – one for the model-view and one for the projection. The projection transformation matrix will be discussed later.

OpenGL builds up a “current transformation” matrix which effectively constructs a transformed coordinate system relative to which the drawing is then carried out.

The current transformation matrix starts as the identity matrix, which is then post-multiplied by rotation, translation or scaling matrices as required.

In OpenGL, first specify model-view mode (as opposed to defining the type of projection transformation being used for the scene) and then initialise the current transformation matrix:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity( );
```

The basic transformations are defined in OpenGL as follows:

```
glTranslatef(x, y, z);  
glScalef(sx, sy, sz);  
glRotatef(angle, x, y, z);
```

As these functions are called, the current transformation matrix is updated and thus the next point to be drawn is relative to the current coordinate system as represented by the current transformation matrix.

2.3.10 Matrix Stacks

While constructing a scene in OpenGL, it is often useful to go back to a previous coordinate system. OpenGL provides a facility to do this – the matrix stack. There are two matrix stacks. One is for projection transformations and the other is for modelling transformations.

These matrix stacks work as all stacks work; that is, OpenGL provides push and pop functions

```
glPushMatrix( );  
glPopMatrix( );
```

It is important to understand the action of **glPushMatrix**. It simply pushes a copy of the current transformation onto the stack. This means that the current transformation matrix is left unchanged, i.e. the current transformation matrix is NOT reset to the identity. It is sometimes necessary to reset the coordinate system to the identity matrix or to an intermediate state, and then to build a new sequence of transformations, as otherwise the transformations would keep compounding.

The function **glPushMatrix** puts a copy of the current transformation matrix on top of the stack. This copy can then be manipulated, updated and used, while the saved copy can be restored by popping the current transformation matrix off the stack.

For example, consider the following sequence:

Load identity	Identity
Push/Save copy	Identity Identity
Apply transformation	T1 Identity
Apply transformation	T1 T2 Identity

Push/Save copy	T1 T2 T1 T2 Identity
Apply rotation	T1 T2 R1 T1 T2 Identity
Draw box	
Pop/Restore	T1 T2 Identity
Apply rotation	T1 T2 R2 Identity
Draw box	

This example has applied two translations, then a rotation and then drawn a box. It then returned to the state represented by the two translations, applied a different rotation and then drew another box. This demonstrates how the stack is built up and used. Note though, that by not saving the “T1 T2” state prior to applying the second rotation (that is, by not pushing a copy of the current transformation matrix on the top of the stack), we cannot now return to that state. We would usually use “push” after a “pop” to avoid this situation.

You should make sure that you have the same number of pushes and pops in your code.