Dire Dawa Institute of Technology
Department of Computer Science
Chapter 4 Handout – Transformations
Compiled by Gaddisa Olani

1. Introduction

In this handout we will introduce some important theoretical foundation for the next chapter on the viewing pipeline. We will review the mathematics of matrix transformations, and see how matrices can be used to perform different types of transformation: translations, rotations and scalings. We will introduce the important topic of homogeneous coordinates – a variation on the standard Cartesian coordinate system that is widely employed in computer graphics. Finally we will consider how transformations can be defined in OpenGL.

2. 2-D Matrix Transformations

First of all let us review some basics of matrices. 2x2 matrices can be multiplied according to the following equation.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix} \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

For example,

$$\begin{pmatrix} 3 & -1 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 3\times1-1\times2 & 3\times2-1\times0 \\ 2\times1+1\times2 & 2\times2+1\times0 \end{pmatrix} = \begin{pmatrix} 1 & 6 \\ 4 & 4 \end{pmatrix}$$

Matrices of other sizes can be multiplied in a similar way, provided the number of columns of the first matrix is equal to the number of rows of the second.

Matrix multiplication is <u>not</u> commutative. In other words, for two matrices *A* and *B*, $AB \neq BA$. We can see this from the following example.

$$\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}\begin{pmatrix} 3 & -1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 5 & -1 \\ 6 & -2 \end{pmatrix}$$

$$\begin{pmatrix} 3 & -1 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 6 \\ 4 & 4 \end{pmatrix}$$

However, matrix multiplication <u>is</u> associative. This means that if we have three matrices *A*, *B* and *C*, then $(AB)C = A(BC)$. We can see this from the following example.

$$\left[\begin{pmatrix}3 & -1\\ 2 & 1\end{pmatrix}\begin{pmatrix}1 & 2\\ 2 & 0\end{pmatrix}\right]\begin{pmatrix}1 & 0\\ 2 & 1\end{pmatrix} = \begin{pmatrix}1 & 6\\ 4 & 4\end{pmatrix}\begin{pmatrix}1 & 0\\ 2 & 1\end{pmatrix} = \begin{pmatrix}13 & 6\\ 12 & 4\end{pmatrix}$$

$$\begin{pmatrix}3 & -1\\ 2 & 1\end{pmatrix}\left[\begin{pmatrix}1 & 2\\ 2 & 0\end{pmatrix}\begin{pmatrix}1 & 0\\ 2 & 1\end{pmatrix}\right] = \begin{pmatrix}3 & -1\\ 2 & 1\end{pmatrix}\begin{pmatrix}5 & 2\\ 2 & 0\end{pmatrix} = \begin{pmatrix}13 & 6\\ 12 & 4\end{pmatrix}$$

In the following sections we will consider how to perform certain common types of coordinate transformations using matrices. We will start off by looking only at 2-D points, i.e. points that have an x and a y coordinate. Later in this chapter we will extend our discussion to 3-D points.

2.1. 2-D Translation

The translation transformation shifts all points by the same amount. Therefore, in 2-D, we must define two translation parameters: the x-translation t_x and the y-translation t_y. A sample translation is illustrated in Figure 1.

To translate a point P to P' we add on a vector T:

$$P = \begin{pmatrix}p_x\\ p_y\end{pmatrix} \qquad \text{(2)}$$

$$P' = \begin{pmatrix}p'_x\\ p'_y\end{pmatrix} \qquad \text{(3)}$$

$$T = \begin{pmatrix}t_x\\ t_y\end{pmatrix} \qquad \text{(4)}$$

$$\begin{pmatrix}p'_x\\ p'_y\end{pmatrix} = \begin{pmatrix}p_x\\ p_y\end{pmatrix} + \begin{pmatrix}t_x\\ t_y\end{pmatrix} \qquad \text{(5)}$$

Therefore, from Eq. (5) we can see that the relationship between points before and after the translation is:

$$p'_x = p_x + t_x \qquad \text{(6)}$$

$$p'_y = p_y + t_y \qquad \text{(7)}$$

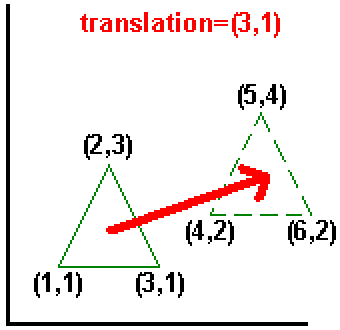

Figure 1 - A 2-D Translation

2.2. 2-D Rotation

The rotation transformation rotates all points about a centre of rotation. Normally this centre of rotation is assumed to be at the origin (0,0), although as we will see later on it is possible to rotate about any point. The rotation transformation has a single parameter: the angle of rotation, θ. A sample rotation in 2-D about the origin is illustrated in Figure 2.

To rotate a point P anti-clockwise by θ^o, we apply the rotation matrix R:

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \quad \ldots \quad (8)$$

$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad \ldots \quad (9)$$

Therefore, from Eq. (9) we can see that the relationship between points before and after the rotation is:

$$p'_x = p_x \cos\theta - p_y \sin\theta \quad \ldots \quad (10)$$

$$p'_y = p_y \cos\theta + p_x \sin\theta \quad \ldots \quad (11)$$

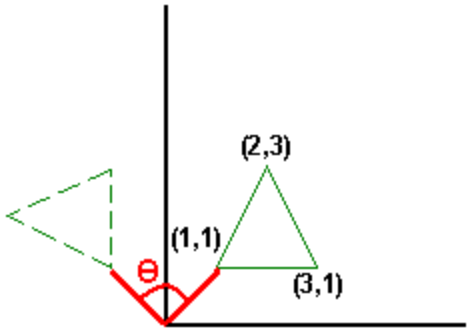

Figure 2 - A 2-D Rotation about the Origin

2.3. 2-D Scaling

The scaling transformation multiplies each coordinate of each point by a *scale factor*. The scale factor can be different for each coordinate (e.g. for the *x* and *y* coordinates). If all scale factors are equal we call it *uniform scaling*, whereas if they are different we call it *differential scaling*. A sample scaling is shown in Figure 3.

To scale a point P by scale factors S_x and S_y we apply the scaling matrix S:

$$S = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (12)$$

$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (13)$$

Therefore, from Eq. (13) we can see that the relationship between points before and after the scaling is:

$$p'_x = S_x p_x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (14)$$

$$p'_y = S_y p_y \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (15)$$

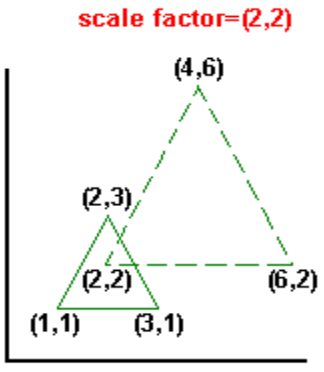

Figure 3 - A 2-D Scaling

3. Homogeneous Coordinates

In the next chapter we will look at the viewing pipeline for computer graphics: a sequence of transformations that every primitive undergoes before being displayed. As we must perform a sequence of transformations in this pipeline, it is essential that we have an efficient way to execute these transformations.

One answer is to *compose* the series of transformations into a single matrix, and then apply the composed matrix to every primitive. This would be efficient because we perform the composition once only (the transformation will be the same for all primitives), leaving only a single matrix multiplication for each primitive. Since matrix multiplications are often executed in dedicated hardware on the video card this will be very fast.

There is a problem with this solution, however. We can illustrate this problem by looking at the following two examples.

Example 1

We want to transform a large number of points by the same sequence of three matrix transformations: a rotation R_1, followed by a scaling S_1 and finally another rotation R_2. In this case, the overall transformation is $P' = R_2 S_1 R_1 P$. Therefore we can implement this by composing R_2, S_1 and R_1 into a single composite matrix C, and then multiplying each point by C.

Example 2

We want to transform a large number of points by the same sequence of four matrix transformations: a translation T_1, a rotation R_1, another translation T_2 and finally a scaling S_1. In this case, the overall transformation can be expressed as $P' = S_1(R_1(P + T_1) + T_2) = S_1 R_1 P + S_1 R_1 T_1 + S_1 T_2$. Clearly this is significantly more complex than the previous example. Even if we combined S_1R_1, $S_1R_1T_1$ and S_1T_2 into composite matrices we would still have to apply two extra matrix additions for every point we wanted to transform. Therefore the operation of the graphics pipeline would be much slower in this second example compared to the first. The reason is that this second sequence of transformations included translations, whereas the first sequence consisted of only rotations and scaling. Using the definitions we have seen so far rotations and scaling are performed using matrix multiplications, whereas translations are performed using matrix additions. We could improve the efficiency of the graphics pipeline if we could find a way to express translations as matrix multiplications. Homogeneous coordinates allow us to do just this. With homogeneous coordinates we add an extra coordinate, the *homogenous parameter*, to each point in Cartesian coordinates. So 2-D points are stored as three values: the *x*-coordinate, the *y*-coordinate and the homogeneous parameter. The relationship between homogeneous points and their corresponding Cartesian points is:

Homogeneous point = $\begin{pmatrix} x \\ y \\ h \end{pmatrix}$, Cartesian point = $\begin{pmatrix} x/h \\ y/h \\ 1 \end{pmatrix}$

Normally the homogenous parameter is given the value 1, in which case homogenous coordinates are the same as Cartesian coordinates but with an extra value which is always 1. In the following sections we will see how adding this extra homogeneous parameter helps us to express translation transformations using matrix multiplications.

3.1. 2-D Translation with Homogenous Coordinates

Now we can express a translation transformation using a single matrix multiplication, as shown below.

$$P = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (16)$$

$$P' = \begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (17)$$

$$T = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (18)$$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (19)$$

Therefore $p'_x = p_x + t_x$, $p'_y = p_y + t_y$, exactly the same as before, but we used a matrix multiplication instead of an addition.

3.2. 2-D Rotation with Homogenous Coordinates

Rotations can also be expressed using homogenous coordinates. The following equations are similar to the form of the 2-D rotation given in Eqs. (8)-(11), with the exception that the rotation matrix *R* has an extra row and and extra column.

$$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{...} \quad (20)$$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad \text{..} \quad (21)$$

Therefore $p'_x = p_x \cos\theta - p_y \sin\theta$ and $p'_y = p_y \cos\theta + p_y \sin\theta$, which is the same outcome as before.

3.3. 2-D Scaling with Homogenous Coordinates

Finally, we can also express scalings using homogeneous coordinates, as shown by the following equations.

$$S = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{...} \quad (22)$$

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix} \quad \text{...} \quad (23)$$

Therefore $p'_x = S_x p_x$ and $p'_y = S_y p_y$, exactly the same as before.

4. Matrix Composition

As we saw in Section 3, the use of homogenous coordinates allows us to compose a sequence of transformations into a single matrix. This can be very useful in the graphics viewing pipeline (see next chapter), but also allows us to define different types of transformation from those we have already seen. For example, the rotation matrix we introduced in Section 3.2 only rotates about the origin, but often we may want to apply a rotation about a different point (a *pivot point*). Using matrix composition, we can achieve this using the following sequence of transformations:

- Translate from pivot point to origin
- Rotate about origin
- Translate from origin back to pivot point

An example of this sequence of transformations is shown in Figure 4. Here we perform a rotation about the pivot point (2,2) by translating by (-2,-2) to the origin, rotating about the origin and then translating by (2,2) back to the pivot point. Let us denote our transformations as follows:

- T_1 is a matrix translation by (-2,-2)
- R is a matrix rotation by θ^o about the origin
- T_2 is a matrix translation by (2,2)

Therefore, using homogenous coordinates we can compose all three matrices into one composite transformation, C:

$$C = T_2 R T_1 \quad \text{(24)}$$

The composite matrix C can now be computed from the three constituent matrices T_2, R and T_1, and represents a rotation about the pivot point (2,2) by θ^o. Note from Eq. (24) that T_1 is applied first, followed by R and then T_2. For instance, if we were to apply the three transformations to a point P the result would be

$$P' = T_2 R T_1 P.$$

Therefore because T_1 is right next to the point P it gets applied first, followed by the next transformation to the left, R, and so on.

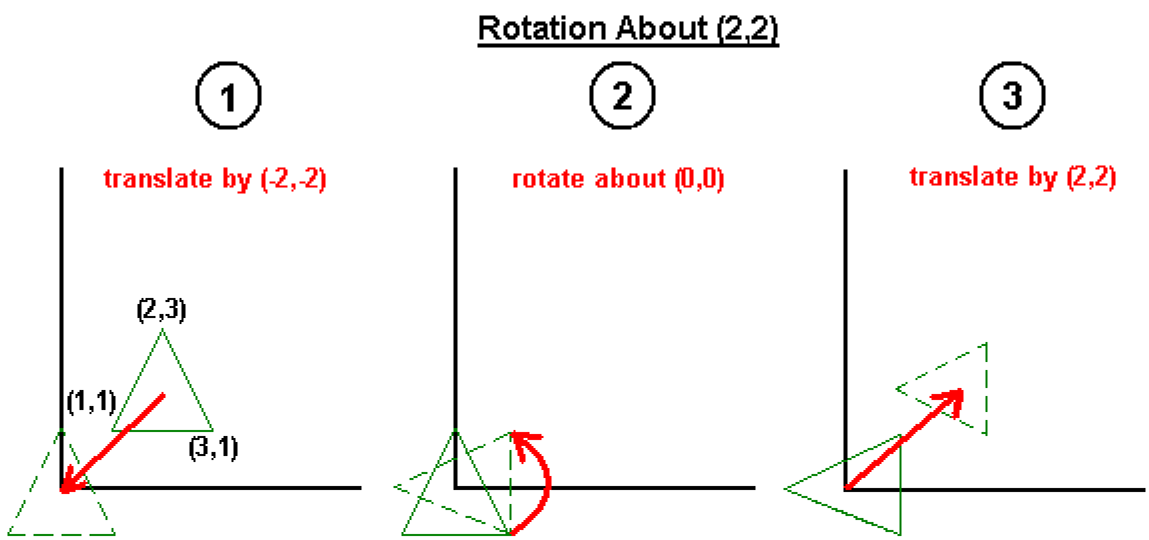

Figure 4 - Composing Transformations to Achieve Rotation about an Arbitrary Pivot Point

5. 3-D Matrix Transformations

The concept of homogenous coordinates is easily extended into 3-D: we just introduce a fourth coordinate in addition to the x, y and z-coordinates. In this section we review the forms of 3-D translation, rotation and scaling matrices using homogeneous coordinates.

5.1. 3-D Translation with Homogenous Coordinates

The 3-D homogeneous coordinate's translation matrix is similar in form to the 2-D matrix, and is given by:

$$T = \begin{pmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (25)$$

We can see that 3-D translations are defined by three translation parameters: t_x, t_y and t_z. We apply this transformation as follows:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (26)$$

Therefore $p'_x = p_x + t_x$, $p'_y = p_y + t_y$ and $p'_z = p_z + t_z$.

5.2. 3-D Scaling with Homogeneous Coordinates

Similarly, 3-D scalings are defined by three scaling parameters, S_x, S_y and S_z. The matrix is:

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We apply this transformation as follows:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Therefore $p'_x = S_x p_x$, $p'_y = S_y p_y$ and $p'_z = S_z p_z$.

5.3. 3-D Rotation with Homogenous Coordinates

For rotations in 3-D we have three possible *axes of rotation*: the *x*, *y* and *z* axes. (Actually we can rotate about any axis, but the matrices are simpler for *x*, *y* and *z* axes.) Therefore the form of the rotation matrix depends on which type of rotation we want to perform.

For a rotation about the *x*-axis the matrix is:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For a rotation about the *y*-axis we use:

$$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And for a rotation about the *z*-axis we have:

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5.4. Right-Handed vs. Left-Handed Coordinate Systems

When referring to 3-D coordinate systems, we can distinguish between *right-handed* and *left-handed* coordinate systems. This concept is illustrated in Figure 5. For a right-handed coordinate system, if we extend the thumb and first two fingers of our right-hand so that they are perpendicular to each other, then the first finger represents the direction of the *x*-axis, the second finger the *y*-axis and then thumb points in the direction of the *z*-axis. Contrast this to a left-handed coordinate system in which we do the same thing with our left hand. In this case, if we align the *x* and *y* axes of the right-handed and left-handed coordinate systems, the *z*-axes will point in opposite directions. The most common type of coordinate system used in computer graphics is the right-handed coordinate system, but when using a general purpose graphics package it is important to know which type of coordinate system it uses.

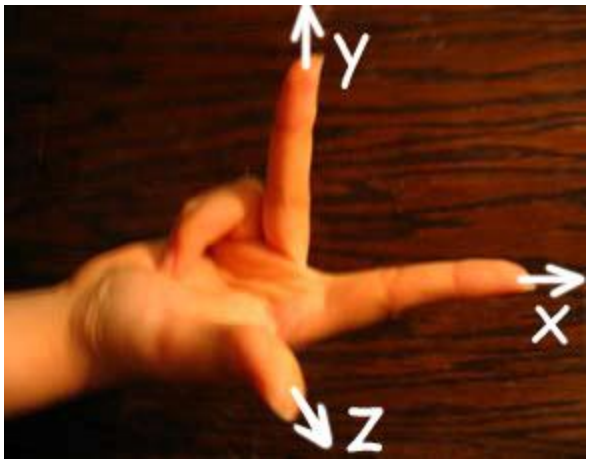

Figure 5 - A Right-Handed Coordinate System

6. Defining Transformations in OpenGL

Before we look at how to define matrix transformations in OpenGL we must introduce the concepts of *premultiplying* and *postmultiplying*. Whenever a matrix is multiplied by another existing matrix we can either *premultiply* or *postmultiply*. For example, if we premultiply matrix *A* by matrix *B*, the result will be *BA*. If we postmultiply matrix *A* by matrix *B* the result will be *AB*. Often, when using a general purpose graphics package we need to specify a sequence of transformations, so we need to know whether the package will compose these transformations by premultiplying or postmultiplying. This is very important because which of these two techniques the package uses determines the order in which we should specify our transformations. For example, suppose we specify a sequence of matrix transformations *A*, *B* and *C* (in this order). Using premultiplying, the composite transformation will be *CBA*, whereas using postmultiplying it will be *ABC*. We have already seen in Section 2 that matrix multiplication is not *commutative*, so these two results will be different. We can see from this example that when postmultiplying we must define our sequence of transformations in the reverse order to that in which we want them to be applied. The result of postmultiplying the matrices *A*, *B* and *C* is *ABC*, so *C* is applied first, followed by *B* and then *A*.

Whenever we apply a transformation in OpenGL it is applied to a *current matrix*. In fact, as we will see in the next chapter, in OpenGL we have several current matrices, but for the moment just remember that there is a current matrix. Almost all transformations in OpenGL *postmultiply* by this current matrix. Therefore when applying a sequence of transformations we must define them in reverse order. We will see an example of this later. OpenGL always uses a *right-handed* coordinate system

Now let us look at OpenGL functions for defining transformations. In total, there are six functions that affect the current matrix:

- *glTranslate*(*t_x*,*t_y*,*t_z*)*: Postmultiply the current matrix by a translation matrix formed from the translation parameters t_x, t_y and t_z.

- *glRotate*(θ,vx,vy,vz)*: Postmultiply the current matrix by a rotation matrix that rotates by θ^o about the axis defined by the direction of the vector (*vx*,*vy*,*vz*).
- *glScale*(S_x,S_y,S_z)*: Postmultiply the current matrix by a scaling matrix formed from the scale factors S_x, S_y and S_z.
- *glLoadMatrix*(elements16)*: Replaces the current matrix with the 16-element array *element16*. The array should be defined in *column-major* order (i.e. the first four elements represent the first column; the next four represent the second column, etc.).
- *glMultMatrix*(elements16)*: Postmultiplies the current matrix with the 16-element array *element16*. The array should be defined in *column-major*.
- *glLoadIdentity(elements16)*: Replaces the current matrix with a 4x4 identity matrix.

Each current matrix in OpenGL has an associated *matrix stack*. This is a standard FIFO stack that can be used to 'remember' different transformations. In fact, the current matrix is actually just the top matrix on the matrix stack. We will see in the next chapter why matrix stacks can be useful, but for the moment let us introduce the two functions for manipulating the stack:

- *glPushMatrix*: Copy the current matrix to the next position down in the stack, push all other matrices down one position. The current matrix (i.e. the top matrix on the stack) is left unchanged.
- *glPopMatrix*: Destroy the current matrix, and move all other matrices on the stack up one position.

To finish this chapter, let us look at an example of using these OpenGL routines to define a composite transformation. Consider the following code:

```
glLoadIdentity();
glTranslated(2.0, 2.0, 0.0);
glRotated(90.0, 0.0, 0.0, 1.0);
glTranslated(-2.0, -2.0, 0.0);
```

This is actually the same example as we saw above in Figure 4: a rotation about the pivot point (2,2). Note from this example that we define the transformations in reverse order (because OpenGL always postmultiplies). This example uses 2-D graphics so the rotation is performed about the *z*-axis.

Summary

The following points summarise the key concepts of this chapter:

- Matrix multiplication is not commutative, i.e. $AB \neq BA$.
- Matrix multiplication is associative, i.e. $(AB)C = A(BC)$.
- Using normal Cartesian coordinates, 2-D translation, rotation and scaling transformations are defined as follows:
  - Translation: $\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$
  - Rotation: $\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$
  - Scaling: $\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$
- *Homogenous coordinates* introduce an extra homogeneous parameter to standard Cartesian coordinates. The homogenous parameter normally has the value 1.
- Using homogenous coordinates, 2-D translation, rotation and scaling transformations are defined as follows:
  - Translation: $\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$
  - Rotation: $\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$
  - Scaling: $\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$
- *Matrix composition* refers to combining (i.e. multiplying together) a sequence of transformations to produce a single transformation matrix.
- Using homogeneous coordinates, 3-D translation, rotation and scaling transformations are defined as follows:
  - Translation: $\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$
  - Scaling: $\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$

- Rotation about the *x*-axis: $\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$
- Rotation about the *y*-axis: $\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$
- Rotation about the *z*-axis: $\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$

- Most graphics packages use a right-handed coordinate system: we can visualise the axes of a right-handed coordinate system by extending the thumb and first two fingers of the right hand so that they are perpendicular: the first finger is the *x*-axis, the second finger is the *y*-axis and then thumb is the *z*-axis.
- If we *premultiply* matrix *A* by matrix *B* the result is *BA*. If we *postmultiply A* by *B* the result is *AB*.
- The following routines can be used in OpenGL to define transformations:
  - *glTranslate*(t_x,t_y,t_z)*
  - *glRotate*(θ,vx,vy,vz)*
  - *glScale*(S_x,S_y,S_z)*
  - *glLoadMatrix*(elements16)*
  - *glMultMatrix*(elements16)*
  - *glLoadIdentity(elements16)*
- OpenGL always postmultiplies by the current matrix. This means that the sequence of transformations must be specified in the reverse order to that which we want them to be applied.
- A *matrix stack* can be used in OpenGL to remember previous transformations. The following routines can be used to manipulate the matrix stack:
  - *glPushMatrix*
  - *glPopMatrix*

Source: Computer Graphics with OpenGL,
Donald Hearn and M. Pauline Baker,
Prentice-Hall, 2004 (Third Edition)