

6. Matrices: Tools for Manipulating Space

Matrices in Computer Graphics

Matrices are used in 2D and 3D computer graphic systems to perform the standard affine transformations of translation, scaling and rotation that form part of the modelling, viewing and coordinate changing processes. Perspective viewing can also be described by a matrix method ('matrix' is the singular of 'matrices'). These different operations can be dealt with in a unified way through their representation as matrix operations, simplifying the code used for these purposes. Matrices, used in conjunction with homogeneous coordinate methods, also enable sequences of transformations to be 'concatenated' into a single operation, thus offering considerable time saving. A matrix maps directly into the two-dimensional array data structure that is readily available in most computer languages. It is a compact method of storing several numbers. These are all reasons for matrix methods to lie at the heart of most computer graphics systems.

The following sections define and develop laws controlling matrices and explain their properties before describing explicitly their main applications to computer graphics. It is hoped that this justification will give readers enough motivation to pursue the chapter through the properties and definitions until the importance of the use of matrices for computer graphics is made clear. Only those properties that are useful in the context of computer graphics are explained.

Definition and Notation

A matrix is a rectangular array of elements. For our purposes, the elements are real numbers, although there is no reason why they cannot be other entities, even matrices themselves. The array is a structure that holds elements of interest. The word 'matrix' can mean the womb or a cavity in which things are embedded. These include the structure of claws which hold jewels in a ring or brooch, the base material within which nuggets of ore or precious materials are located in geology or the mould from which type is cast. The rectangular form of a mathematical matrix means that elements appear in complete rows and columns.

Most of the notation concerning matrices is made clear by a single statement.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}.$$

The matrix itself is denoted by an upper case (capital) character, with its elements, enclosed within square brackets, carrying the same lower case character. Some texts

use round brackets '(' and ')'. This is not an important issue; curly braces '{' and '}' or plain vertical lines should, however, not be used as they can have different meanings. Each element has a pair of suffixes, indicating its row and column position. The (horizontal) row is always denoted before the (vertical) column. The general example given above has m rows and n columns and is said to have order $(m \times n)$ – pronounced 'm by n'. Again, rows before columns is the rule. For example, if we have

$$B = \begin{bmatrix} 3 & 8 & -4 & 0 \\ 2 & -8 & 8 & 5 \\ 1 & 1 & 4 & 11 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 4 & 1 & 7 \\ -3 & 1 & -5 \\ 2 & -4 & 1 \end{bmatrix},$$

matrix B has order (3×4) , with $b_{13} = -4$ and $b_{31} = 1$, and C has order (3×3) , with $c_{13} = 7$ and $c_{31} = 2$.

The simplest matrix operation is to find a transpose. The transpose A^T of a matrix A is found by writing the rows of A as the columns of A^T . Some texts use \bar{A} or A' for transpose. If matrix P has order $(m \times n)$ and $Q = P^T$, then Q has order $(n \times m)$ and $q_{ij} = p_{ji}$ for all i and j (that is, for $i = 1$ to n and $j = 1$ to m). For example, the transposes of the example matrices B and C given above are

$$B^T = \begin{bmatrix} 3 & 2 & 1 \\ 8 & -8 & 1 \\ -4 & 8 & 4 \\ 0 & 5 & 11 \end{bmatrix} \quad \text{and} \quad C^T = \begin{bmatrix} 4 & -3 & 2 \\ 1 & 1 & -4 \\ 7 & -5 & 1 \end{bmatrix}.$$

Forms of Matrices

Many of the matrices used in computer graphics have the same number of rows as columns. Such matrices are known as square matrices. C above is an example of a 'square matrix of order 3', which means the same as a 'matrix of order (3×3) '.

Particularly for a square matrix, the set of elements whose row and column numbers are equal is known as the leading diagonal. For our example matrix C , the elements c_{11} , c_{22} and c_{33} are emboldened to show the leading diagonal below.

$$C = \begin{bmatrix} \mathbf{4} & 1 & 7 \\ -3 & \mathbf{1} & -5 \\ 2 & -4 & \mathbf{1} \end{bmatrix}.$$

For reasons that should be obvious on inspection, the matrices D , E , F and G below are known as diagonal, upper triangular, lower triangular and symmetric respectively.

$$D = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 5 \end{bmatrix}, \quad E = \begin{bmatrix} -2 & 2 & -1 \\ 0 & -4 & 3 \\ 0 & 0 & 5 \end{bmatrix},$$

$$F = \begin{bmatrix} 4 & 0 & 0 \\ -3 & -3 & 0 \\ 2 & -4 & -1 \end{bmatrix}, \quad G = \begin{bmatrix} 3 & 1 & -2 \\ 1 & -5 & -5 \\ -2 & -5 & 1 \end{bmatrix}.$$

More formally stated, the rules for such square matrices are

- D is diagonal if and only if: $d_{ij} = 0$ whenever $i \neq j$;
- E is upper triangular if and only if: $e_{ij} = 0$ whenever $i > j$;
- F is lower triangular if and only if: $f_{ij} = 0$ whenever $i < j$;
- G is symmetric if and only if: $g_{ij} = g_{ji}$ for all i and j .

Operations on Matrices: Addition

There is no point in merely defining and describing several forms of matrices and looking at them in wonder, as one might admire the jewels held in a brooch. Matrices were developed in the mid to late nineteenth century, particularly by Arthur Cayley and James Joseph Sylvester, because they were useful in a number of topical problems. Werner Heisenberg developed some methods in the mid 1920s in advancing the topic of quantum mechanics; he was advised by Max Born that he had redeveloped some results from matrix theory. We now look at laws for comparing, manipulating and combining matrices that are useful in the context of computer graphics.

For two matrices to be equal, they have to be identical in all aspects. Equality of two matrices implies that they have the same orders and all pairs of corresponding elements are equal. Matrices also have to be of the same form before they can be added. Two matrices are conformable for addition if and only if they have the same orders. If this is the case, their sum has the same order and is formed by filling its elements with the sums of corresponding elements from the matrices to be added. Addition of matrices is a binary operator, as it takes in *two* matrices and blends them to give one as a result. Both addition and equality are illustrated in the statement

$$\begin{bmatrix} 4 & -2 & 8 & 1 \\ -8 & 0 & -4 & -5 \\ 5 & 3 & 0 & 3 \end{bmatrix} + \begin{bmatrix} -2 & 3 & -1 & 0 \\ -4 & -4 & 4 & 4 \\ 5 & 1 & 3 & -3 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 7 & 1 \\ -12 & -4 & 0 & -1 \\ 10 & 4 & 3 & 0 \end{bmatrix},$$

but no sum can be formed from two matrices that are not compatible for addition, such as

$$\begin{bmatrix} 4 & -2 & 8 & 1 \\ -8 & 0 & -4 & -5 \\ 5 & 3 & 0 & 3 \end{bmatrix} \text{ and } \begin{bmatrix} -2 & 3 & -1 \\ -4 & -4 & 4 \\ 5 & 1 & 3 \end{bmatrix}.$$

If a matrix is added to itself (this is always possible as the order must be the same), the effect is to multiply each element by 2. Similar repeated addition gives

multiplies by 3, 4, and so on. This suggests the rule for multiplying a matrix by a scalar (a scalar is simply a single real valued number; this term is used to distinguish it from other mathematical 'objects' such as matrices and vectors). To multiply a matrix A by a scalar value λ , simply multiply all elements of A by λ . For example,

$$-4 \begin{bmatrix} -5 & -2 \\ 3 & 1 \\ -4 & 3 \end{bmatrix} = \begin{bmatrix} 20 & 8 \\ -12 & -4 \\ 16 & -12 \end{bmatrix}.$$

Operations on Matrices: Multiplication

We have seen how to multiply a matrix by a scalar; now we consider the binary operation that multiplies a matrix by another matrix. This is not as straightforward as the operations described above. It may seem bewildering why a relatively complex operation is performed, when an easier one could have been defined. The reason is its usefulness. Defining an easy operation that has no obvious uses serves no purpose. A relatively complicated definition is used because it is useful across a wide range of potential applications. It is worth persevering to master the skill of multiplying matrices.

As in the case of addition, it is not always possible to multiply a pair of matrices. Matrices A and B can be multiplied only if they are conformable for multiplication, which occurs when the number of columns of A is equal to the number of rows of B . Thus, if A has order $(m \times p)$ and B has order $(p \times n)$, multiplication is allowed. Note that the row length of A is the same as the column length of B ; they are both equal to p . This is essential for the multiplication process described below. For example, when

$$A = \begin{bmatrix} 2 & -2 & 1 \\ 4 & 4 & -1 \\ -2 & 0 & 5 \end{bmatrix} \text{ and } B = \begin{bmatrix} -3 & 3 \\ -1 & -5 \\ 0 & 4 \end{bmatrix},$$

the order of A is (3×3) and that of B is (3×2) , so A and B are compatible for multiplication. The rows of A and the columns of B contain three elements. To decide on compatibility for multiplication, write down the orders of both matrices next to each other, as ' $(3 \times 3) (3 \times 2)$ '. If the central numbers can be equally 'linked', as in

$$(3 \times \boxed{3}) (\boxed{3}) \times 2)$$

then the matrices are conformable for multiplication. The numbers outside the linking box are also significant. They give the order of the matrix that results from the multiplication, (3×2) in this example. Thus, in general, if A has order $(m \times p)$ and B has order $(p \times n)$, multiplication is allowed and the resulting product matrix, C say, has order $(m \times n)$.

Now we need to find the elements of C . The i^{th} row of A and j^{th} column of B have elements

$$\begin{array}{cccccc}
 a_{i1} & a_{i2} & \dots & a_{ip} & \text{and} & b_{1j} \\
 & & & & & b_{2j} \\
 & & & & & \dots \\
 & & & & & b_{pj}
 \end{array}$$

respectively (remember that the first suffix indicates the row position and the second indicates the column position). To find the element c_{ij} of C , these elements are multiplied together pair by pair, the resulting values being summed to give

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj}.$$

The ‘...’ can be avoided using more formal mathematical notation as

$$c_{ij} = \sum_{k=1}^p a_{ik}b_{kj}.$$

This is be interpreted as ‘the sum of all values of $a_{ik}b_{kj}$ when k changes from 1 to p ’. The Greek equivalent of capital S, ‘ Σ ’, is used to denote that a sum is being found. This form of sum of repeated products is sometimes called an ‘inner product’ in mathematical texts.

For example, to find element c_{32} of the product C of our example matrices A and B ,

$$A = \begin{bmatrix} 2 & -2 & 1 \\ 4 & 4 & -1 \\ -2 & 0 & 5 \end{bmatrix} \text{ and } B = \begin{bmatrix} -3 & 3 \\ -1 & -5 \\ 0 & 4 \end{bmatrix},$$

we take the third row of A and the second column of B ,

$$\begin{array}{ccc}
 -2 & 0 & 5 & \text{and} & \begin{array}{c} 3 \\ -5 \\ 4, \end{array}
 \end{array}$$

multiply pair-wise as

$$\begin{array}{l}
 -2 \times 3 = -6 \\
 0 \times -5 = 0 \\
 5 \times 4 = 20
 \end{array}$$

and add to give

$$c_{32} = -6 + 0 + 20 = 14.$$

This is done for all six elements of C to give

$$\begin{bmatrix} 2 & -2 & 1 \\ 4 & 4 & -1 \\ -2 & 0 & 5 \end{bmatrix} \begin{bmatrix} -3 & 3 \\ -1 & -5 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} -4 & 20 \\ -16 & -12 \\ 6 & 14 \end{bmatrix}.$$

This rather unpleasant, but useful, process has a number of consequences. Matrix multiplication is not commutative; in general $AB \neq BA$. Even if AB can be formed, it may not be possible to find the product BA . This is the case in the example above, where A has order (3×3) and B has order (3×2) . A and B are compatible for multiplication, but B and A are not. In a product AB , A is said to pre-multiply B and B is said to post-multiply A .

Given matrices A order $(m \times p)$, B order $(p \times q)$ and C order $(q \times n)$, the product AB can be formed, having order $(m \times q)$. This is conformable for multiplication with C , and the permitted product $(AB)C$ has order $(m \times n)$. In $(AB)C$, the brackets indicate that the product AB is the first to be performed in time. Alternatively, we could form the product BC with order $(p \times n)$ and then pre-multiply by A to give $A(BC)$ with order $(m \times n)$. Both $(AB)C$ and $A(BC)$ are valid sequences of multiplication; both give matrices of the same order $(m \times n)$ and both end up giving exactly the same result. Thus, we can write both forms as ABC , the order of performing the multiplication being irrelevant. No proof is offered, but readers can justify this by trying a few examples. This appears to imply that the order of multiplication does not matter, whereas the last paragraph stressed that it does. The apparent paradox is due to different interpretations of the word 'order'. It is important to maintain *positional* order when multiplying matrices, but change of *time* order does not affect the outcome. This result will be useful later when we consider matrix operations to perform sequences of transformations on points.

The product AB involves the rows of A associated with the columns of B . In forming a transpose as defined above, rows and columns are exchanged, leading to the following rule for transpose of a product:

$$(AB)^T = B^T A^T.$$

This, and the multiple product rule described above, can be verified by trying out a few examples.

The Identity Matrix

In number theory, the identity element for any binary operation leaves the other value concerned unchanged. For example, zero has this role for addition (adding zero to any value leaves it unchanged) and 1 has this role for multiplication. An identity element for matrix multiplication is known as the identity matrix or unit matrix, designated by the symbol I . There is more than one form of identity matrix, depending on the order of the matrix needed to perform the multiplication. I is always a square diagonal matrix. If required, it can be designated as I_n , where n indicates its order. It is easy to verify that

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix},$$

with all elements zero except the unit values in the leading diagonal, satisfies these requirements. If I pre-multiplies or post-multiplies a matrix, the product leaves the other matrix unchanged.

The identity matrix is closely tied to the idea of an inverse matrix. In number theory, the inverse of the number x for addition is $-x$, as

$$x + (-x) = 0,$$

the identity for addition. The inverse of the number x for multiplication is $1/x$, as

$$x \cdot (1/x) = 1,$$

the identity for multiplication. Note that we must exclude $x = 0$ from this definition, it has no inverse as we are not allowed to divide by zero. In terms of function theory, we must exclude the value zero from the domain of the multiplicative inverse function. In general, an inverse ‘undoes’ the effect of an operation. If A is a square matrix, it *may* have an inverse – some matrices are excluded, as for multiplication of numbers. If such an inverse exists, it is designated as A^{-1} and has the property

$$AA^{-1} = A^{-1}A = I.$$

For square matrices of order two, the inverse can be easily stated. If

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix},$$

$$\text{then } A^{-1} = \frac{1}{(a_{11}a_{22} - a_{12}a_{21})} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix},$$

provided $(a_{11}a_{22} - a_{12}a_{21})$, known as the determinant of A , is not zero. This can be checked by multiplying A and A^{-1} . All square matrices have a determinant, a single value calculated from the elements of the matrix. For square matrices of order 2, the determinant is formed as shown here, multiplying the elements on the leading diagonal and subtracting the product of the two remaining elements. For square matrices of order 3 or more, the calculation of determinant and inverse is more complicated. This will be revisited later. The value of the determinant indicates whether a square matrix has an inverse. If the determinant is non-zero, an inverse exists and the matrix is said to be non-singular. A square matrix with no inverse (in other words with zero determinant) is singular, it has no ‘partner’.

Matrices and Equations

One of the major uses of matrices in general applications is in the solution of systems of equations. This is not the main use of matrices in computer graphics, but can be useful in some circumstances, for example in solving the systems of equations produced in the radiosity shading method (chapter 10) or solving some intersection problems in vector work (chapter 7). This section and the section following on general methods for calculating matrix inverses can therefore be considered as optional – they can be returned to if needed but may be skipped at first reading. For those who take this option, it would be sensible to restart, without penalty, at the section on ‘Matrices, transformations and homogeneous coordinates’ (page 160).

The ability to solve simple systems of linear equations (containing only single powers and no products of unknown quantities) is a skill learned at secondary school. For example, consider the two-variable system

$$\begin{aligned} 2x + 5y &= -1 & (1) \\ 3x - 4y &= 10 & (2). \end{aligned}$$

The system contains two equations with two unknowns. Generally, if we have the same number of distinct equations as unknown values, the system can be solved. If plotted as graphs in a Cartesian plane, equations (1) and (2) represent straight lines (hence the description as linear equations). Many points separately satisfy each of the equations, for example equation (1) is satisfied by (x, y) pairs $(-0.5, 0)$, $(2, -1)$ and $(-8, 3)$. A similar list of values can be generated to satisfy (2). The problem is to find the one point that satisfies both equations, if such a value exists. Interpreting the problem graphically, this involves finding the point at which the lines intersect. Some pairs of lines may have no intersection (they may be parallel), others may have an infinite number of common points (the two equations represent the same line). *One* way (not the only one) of approaching this problem is as follows.

Multiply both sides of equation (1) by 3 and both sides of equation (2) by 2 (these are the ‘coefficients’ of x in the alternate equation). Such multiplication involves multiplying all parts of the equation, both left-hand and right-hand sides, by the required constant. If the same operation is performed on both sides of an equation, the balance of the equation is unchanged. This gives

$$\begin{aligned} 6x + 15y &= -3 & (3) \\ 6x - 8y &= 20 & (4). \end{aligned}$$

Note that (3) and (4) are simply disguised versions of (1) and (2), with the coefficients of x made the same in both equations; the multiplying values for the original equations were chosen so that this should occur. The x terms can now be removed by subtracting equation (4) from equation (3). This involves subtracting left-hand side from left-hand side and right-hand side from right-hand side to maintain the balance, to give

$$23y = -23 \quad (5).$$

The number of unknown values and the number of equations is reduced by one, leaving one equation for one unknown, y . Note that if the equations represented parallel or coincident lines, the y term would also have disappeared, indicating that no single solution is possible. Equation (5) is solved by dividing both sides by 23, the coefficient of y , to give

$$y = -1.$$

We can substitute this value – this means replacing y wherever it occurs by the particular value -1 , into either of the original equations to find x . Using (1) for this purpose gives

$$2x - 5 = -1,$$

which can be rearranged by adding 5 to both sides, to give

$$2x = 4.$$

Dividing both sides by 2 (equivalent to the operation used to solve equation (5)),

$$x = 2.$$

Thus our final solution is $x = 2, y = -1$. This is the only pair of values of x and y that simultaneously satisfy both equations (1) and (2) and represent the intersection point of the two lines represented by these equations.

This relatively simple problem has been dealt with in some detail to illustrate some of the processes used in solving systems of equations. The validity of an equation is maintained if the same operation is performed on both sides of the equation. An exception to this rule is division by zero, which must be checked for in any computer routine. Useful operations on a system of equations, used in the general algorithm described later, are:

- multiplication of any equation by a constant;
- addition of a multiple of any equation to any other equation;
- swapping the positions of any two equations within the system.

The second of these could be replaced by the two operations of multiplying one equation by a constant and then adding it to another equation, which is effectively what was done in the above example.

Matrices have not yet been mentioned in this section. Systems of linear equations can be represented in matrix form as

$$AX = B.$$

For equations (1) and (2), A , X and B take the form

$$A = \begin{bmatrix} 2 & 5 \\ 3 & -4 \end{bmatrix}, \quad X = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -1 \\ 10 \end{bmatrix}.$$

Multiplying out AX , this gives

$$\begin{bmatrix} 2x + 5y \\ 3x - 4y \end{bmatrix} = \begin{bmatrix} -1 \\ 10 \end{bmatrix}.$$

Equating individual elements of these two matrices gives us the original equations (1) and (2). The structure $AX = B$ suggests a solution using the inverse of matrix A . If that inverse exists, we can pre-multiply both sides of this by A^{-1} to give

$$A^{-1}AX = A^{-1}B.$$

$A^{-1}AX$ can be 'grouped' as $(A^{-1}A)X = IX = X$, so the left-hand side is simply the matrix of unknowns. Thus, a complete solution is given as

$$X = A^{-1}B.$$

If the inverse A^{-1} does not exist (the determinant of A is zero), then there is no single solution of the system of equations. This method seems attractive, but the cost in evaluating A^{-1} can be prohibitively expensive, particularly for large systems of equations with several unknowns, so we investigate another method below.

All values involved in the problem can be placed in a single matrix M , combining the elements of A and B ,

$$M = \begin{bmatrix} 2 & 5 & -1 \\ 3 & -4 & 10 \end{bmatrix},$$

where each row of M contains all information from one of the original equations. The process of solving the system can be performed using the three equation operations defined above on rows of the matrix. The method described here, known as 'Gaussian elimination with partial pivoting', was developed by Gauss¹ and uses 'partial pivoting' to ensure rounding errors do not make computer-generated solutions unreliable. The procedure, which gives a standard method for dealing with any set of equations, is demonstrated on this limited system before a general algorithm is stated. It is important to keep in mind that we have a system of two equations for two unknown quantities (there are two rows of M). Pivoting is the first stage of this process, which starts by searching column 1 of M for its numerically largest element. This is 3, in row 2. (Note that in finding a 'numerically largest' element, the sign is ignored. For this purpose, -3 would be selected before 2, although 2 is strictly larger than -3.) The 'pivot' for column 1 is found in row 2, so we swap rows 1 and 2 to place the pivot in position m_{11} .

$$R_1 \leftrightarrow R_2: \quad M' = \begin{bmatrix} 3 & -4 & 10 \\ 2 & 5 & -1 \end{bmatrix}.$$

¹Carl Friedrich Gauss (1777–1855) was a prolific German mathematician, astronomer and physicist.

Now divide row 1 by m_{11} (the 'pivot'). This is later used to clear zeros in the remainder of its column (only one element in this case). Working to the very limited precision of one decimal place, we get the matrix,

$$R_1 \leftarrow R_1/m_{11}: \quad M'' = \begin{bmatrix} 1 & -1.3 & 3.3 \\ 2 & 5 & -1 \end{bmatrix}.$$

We continue to work with limited precision for this section as an extreme example of dealing with rounding errors. Now produce zeros in column 1 below row 1 by subtracting relevant multiples of row 1 from the remaining rows (this statement may sound artificial as there is only one such other row in this case, but this more general form is valid for any size of problem). The correct multiples for each row are found by taking all elements m_{i1} , where $i > 1$. Note that these are all numerically less than the original pivot value of m_{11} , so the net effect of the last two operations is to multiply elements of column 1 by a value numerically less than one, thus reducing any rounding error. For this example, we only have $m_{21} = 2$ to deal with (so the net effect of the last two operations is to multiply by $2/3$, which is less than one). $m_{21}R_1$ is subtracted from R_2 .

$$R_2 \leftarrow R_2 - m_{21}R_1: \quad M''' = \begin{bmatrix} 1 & -1.3 & 3.3 \\ 0 & 7.6 & -7.6 \end{bmatrix}.$$

We have now reduced column 1 to a simple form. We should now search through column 2 from m_{22} through m_{32} and so on for its pivot element and exchange rows where necessary, but only one such element remains, m_{22} with value 7.6 (this would be $7^2/3$ working precisely). We divide row 2 by this (effectively a default pivot).

$$R_2 \leftarrow R_2/m_{22}: \quad M'''' = \begin{bmatrix} 1 & -1.3 & 3.3 \\ 0 & 1 & -1 \end{bmatrix}.$$

As we have reached the bottom row, the elimination phase is over.

Remember that the three terms in a row of the equation in this case represent the multiple of x , multiple of y and the constant on the right-hand side of the equation. Thus, a row with 0, 1 and -1 in these positions represents equation

$$0x + 1y = -1,$$

or $y = -1.$

The first row of M'''' can also be represented as an equation,

$$1x - 1.3y = 3.3.$$

The value of y can be substituted into this, to give the result $x = 2$, $y = -1$, as before. Although there were intermediate rounding errors, the final solution turns out to be exactly correct – this does not always happen, but the pivoting does give some protection against rounding errors when many variables are involved. In small-scale problems, human ingenuity can often produce short-cut solutions. Computers

can be programmed more easily to follow a standard method such as that described here.

Suppose we have a general system of n simultaneous equations for n unknowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ \dots & \dots \dots \dots \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n. \end{aligned}$$

The unknowns are designated as x_1, x_2, \dots, x_n rather than the x, y used above. This is generally enough information to solve for the n unknowns, provided the equations are linearly independent. This means that none of the equations can be derived from the others using the standard equation manipulation rules. If the equations should be linearly dependent, the fact that there is no single solution will become clear during the process, as at some stage there will be no available non-zero value to act as pivot. An intuitive interpretation of this is that in the n -dimensional space of the variables (x_1, x_2, \dots, x_n), some of the 'surfaces' represented by the equations are parallel. This concept relates to the 'degrees of freedom' concept introduced earlier. If one or more of the equations can be derived from others, it acts as a constraint as its constants cannot be freely determined. Thus the dimensionality of the system of equations is reduced accordingly, and less information is carried.

We start by setting up a single matrix from the coefficients of the equations,

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{bmatrix}.$$

Within a loop, from $i = 1$ to n , we start by searching for the numerically largest of values in or below the leading diagonal in column i , a_{i1} to a_{in} . 'Numerically largest' means the element with largest modulus; the negative signs are ignored. Suppose this is in row k . If k and i are not equal, row k is exchanged with row i to place the pivotal element in position a_{ii} . Row i is then divided by the current pivot value in a_{ii} . Looping from $j = i + 1$ to n when $i < n$ (this stage is not needed for the final row when $i = n$), values in column i below a_{ii} are then reduced to zero by subtracting $a_{ji}R_i$ from R_j . When this loop is exhausted, M will usually have the form

$$M' = \begin{bmatrix} 1 & a_{12} & \dots & a_{1n} & b_1 \\ 0 & 1 & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & b_n \end{bmatrix}.$$

The values of a_{12}, \dots, b_n in M' are not the same as those in M above. If at any stage during this loop, the search for the pivot (the numerically largest of the values a_{i1} to a_{in}) shows all these values to be zero, it is clear that there is no distinct solution to the set of equations. The search for a solution can be abandoned at this stage.

Now the process of ‘back substitution’ starts. $y_n = b_n$ can be immediately allocated. Looping backwards for i from $n - 1$ to 1, we can define

$$x_i = b_i - \sum_{j=i+1}^n a_{ij}x_j,$$

to give the full set of values of x_1, x_2, \dots, x_n .

A routine based on this method can be built into a suite of graphics functions. Alternatively, a similar routine can be used from a commercially marketed numerical analysis system; this kind of numerical equation solver is properly a part of the subject known as ‘numerical analysis’.

The Inverse of a Square Matrix

The method given above for solving equations can be adapted to find the inverse of a square matrix. This is only one of several possible methods that could be used, but is relatively efficient and safe in its avoidance of rounding error. The square matrix A to be inverted is augmented by placing an identity (or unit) matrix to its right, as

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & 1 & \dots & 0 \\ \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & 0 & 0 & \dots & 1 \end{bmatrix}.$$

M is operated on very much as in the Gaussian elimination method with partial pivoting, except that, in reducing elements in a particular column to zero, this is performed for *all* elements in the column (those above the pivot as well as those below) except for the pivot a_{ii} itself. We should replace the statement ‘values in column i below a_{ii} ...’ by ‘all values in column i except for a_{ii} itself are then reduced to zero by looping from $j = 1$ to n (excluding the case $j = i$), subtracting $a_{ji}R_i$ from R_j ’. If at any stage it is impossible to find a non-zero pivot, this indicates that the determinant of the original matrix A is zero, so no inverse exists. Otherwise, M will reduce to the form

$$M' = \begin{bmatrix} 1 & 0 & \dots & 0 & b_{11} & b_{12} & \dots & b_{1n} \\ 0 & 1 & \dots & 0 & b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots \\ 0 & 0 & \dots & 1 & b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}.$$

Now the unit matrix is to the left, with a new square matrix to its right. This new square matrix is extracted as A^{-1} , the inverse of A ,

$$A^{-1} = B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix}.$$

No attempt is made to justify this method, other than to state that it works. The matrices produced form true inverses.

As a side issue, the determinant of matrix A can be evaluated from this process. During each stage of the original i loop, 'row i is divided by the current pivot value in a_{ii} '. As the loops proceed, these pivot values should be multiplied together and the number of row swaps performed, m , counted (if a row is swapped with itself, this does not count). The determinant of A is the product of the pivot multiplied by -1^m . It is sometimes of interest in computer graphics systems to find if the determinant of a square matrix is numerically less than one. This can be found directly from the pivotal product.

Matrices, Transformations and Homogeneous Coordinates: Two Dimensions

We now return to a major issue of importance to computer graphics. In discussing coordinate systems earlier, the transformations of translation, scaling and rotation were defined and equations for these developed. In this section, we show how matrix methods enable all these transformations to be performed by a standard method, giving a unified way of treated them in a computer graphics system. This technique has the added bonus of enabling sequences of transformations to be merged or 'concatenated' into one operation, speeding computer graphics routines. We consider first the 2D transformations (fig 6.1).

Translation $x' = x + t_x,$
 $y' = y + t_y.$

Scaling $x' = s_x x,$
 $y' = s_y y.$

Rotation $x' = x \cos(\theta) - y \sin(\theta),$
 $y' = x \sin(\theta) + y \cos(\theta).$

Shear in x $x' = x + k_x y,$
 $y' = y.$

Shear in y $x' = x,$
 $y' = k_y x + y.$

The shear operations are not given explicitly in many texts as they can be performed by combinations of rotations and scalings, but they can be more efficiently

performed if separately defined. The vertices of a 2D object are represented in Cartesian form as (x, y) . The formulae above transform (x, y) into an 'image' point or vertex (x', y') . When applied to all vertices defining an original object, such as the unit square of fig 6.1, the method changes the object into a transformed version. The effect of the translation defined here is to move an object t_x units horizontally and t_y units vertically. Scaling expands or contracts by factors s_x horizontally and s_y vertically, the origin $(0, 0)$ remaining 'pinned down'. A negative scale factor gives a reflection in the relevant axis. Rotation again pins down the origin $(0, 0)$, rotating the object through an angle θ in the positive anti-clockwise sense.

These sets of equations are directly expressed as matrix operations.

Translation
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix};$$

Scaling
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix};$$

Rotation
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Shear in x
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & k_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix};$$

Shear in y
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ k_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

The point (x, y) is represented in matrix form as $\begin{bmatrix} x \\ y \end{bmatrix}$. A perfectly valid alternative is to use $[x \ y]$ for (x, y) . In this case, the transformations would involve the transposes of the versions given above, with the order of multiplication for scaling and rotation reversed. The choice of method is a matter of personal choice – they are both equivalent in power – but once chosen, the method should not be changed as inconsistency could lead to error.

This is a matrix method, but it holds no advantage over the direct use of equations. A

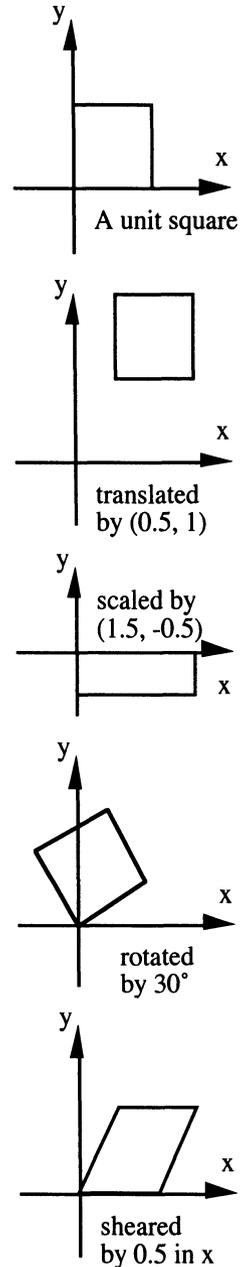


Figure 6.1 Effects of 2D transformations on a unit square

drawback is that the matrix methods used are not the same for each type of operation. Translation involves addition, whereas scaling and rotation use multiplication. Matrix methods become useful when all three basic transformations can be performed in the same way. Shear is not counted as one of these basic transformations as it can be performed by combining scaling and rotations. It is not possible to reduce all to the simpler form of matrix addition; some multiplication is involved. They can all be performed by matrix multiplication using the device of homogeneous coordinates.

The point (x, y) is represented in homogeneous coordinate form as a matrix $\begin{bmatrix} wx \\ wy \\ w \end{bmatrix}$, where w is any arbitrary non-zero constant, called the 'weight'. Thus, there are many possible interpretations for a single point. For example, $(3, -4)$ can be

represented as $\begin{bmatrix} 3 \\ -4 \\ 1 \end{bmatrix}$, $\begin{bmatrix} 6 \\ -8 \\ 2 \end{bmatrix}$ or $\begin{bmatrix} -9 \\ 12 \\ -3 \end{bmatrix}$ with weights 1, 2 and -3 respectively. Given a

point in homogeneous coordinate form, its Cartesian (x, y) form can easily be extracted by dividing the first two row values by the third. In the homogeneous

form $\begin{bmatrix} 11 \\ -8 \\ -2 \end{bmatrix}$ both 11 and -8 are divided by -2 to give the Cartesian point $(-5.5, 4)$. As

the choice of weight w is arbitrary, it makes sense to choose the easiest possible value to work with. Whenever possible, which is almost always in computer graphics, we choose $w = 1$. A general point (x, y) can be represented in the simplest

homogeneous form as $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. The exception to this in computer graphics occurs in

performing perspective projection, when use of a non-unit weight can simplify other calculations. This will be considered in a later section on 3D work.

We are now ready to devise the unified method for dealing with our three transformations. The required transformations are simply stated – their validity can be checked by multiplying out the matrices concerned and comparing results with the equations given above.

$$\text{Translation} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = T(t_x, t_y) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix};$$

$$\text{Scaling} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = S(s_x, s_y) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix};$$

$$\text{Rotation} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix};$$

$$\text{Shear in } x \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & k_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \text{Sh}_y(k_x) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

(the notation indicates that y is unchanged);

$$\text{Shear in } y \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \text{Sh}_x(k_y) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

(the notation indicates that x is unchanged).

The six matrices $T(t_x, t_y)$, $S(s_x, s_y)$, $R(\theta)$, $\text{Sh}_y(k_x)$ and $\text{Sh}_x(k_y)$ represent the operations of translation, scaling, rotation, shearing in the x direction and shearing in the y direction respectively. The clumsy notations used for shearing matrices indicate by suffixes which coordinate is unchanged by the shear. This method is used for consistency with the 3D forms introduced below. It is worth reiterating that some texts represent their homogeneous coordinates as a row matrix $[x \ y \ 1]$ rather than the column matrices used here. If this is done, the transformation matrices are the transposes of those given above and the order of multiplication is reversed. Care should be taken when interpreting texts to understand which method is being used. In a particular work, consistency is important. It does not matter which method is used, as long as the same one is used throughout.

Now we can illustrate the benefits of this technique. In a naïve application of multiple transformations, all transformations would be applied successively to every point of an object to be transformed. Using the matrix method for homogeneous coordinates, sequences of transformations can be reduced to a single matrix. As an example, suppose we wish to rotate a figure through 90° clockwise about the point $(3, -2)$. This can be performed by the sequence of 3 standard transformations shown in fig 6.2).

- translate the whole figure so that $(3, -2)$ is placed at the origin using a translation of $(-3, 2)$;
- rotate through the required angle of -90° (remember that a clockwise rotation is in the negative sense);
- return the figure to the correct location by moving its origin to $(3, -2)$ using a translation of $(3, -2)$.

The first translation stage is defined by matrix

$$T(-3, 2) = \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

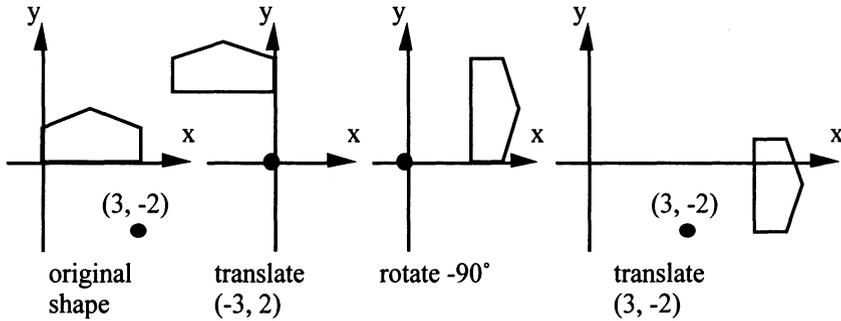


Figure 6.2 Rotation by 90° clockwise about $(3, -2)$

The standard form of rotation allows us to rotate by any angle about the origin. This initial translation rearranges the figure so that the required centre of rotation is now located at the origin, enabling us to rotate by -90° . From chapter 3, we use the trigonometric values $\sin(-90^\circ) = -1$ and $\cos(-90^\circ) = 0$. The required rotation matrix is

$$R(-90^\circ) = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The last translation, replacing the centre of rotation in its original position, has matrix

$$T(3, -2) = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Using matrix multiplication, these three operations can be combined into one matrix before processing the figure itself, so only one operation needs to be applied to each of the object's points. Figure 6.2 shows only a simple object defined by five points. When the method is applied to objects defined by several thousands of points, considerable time saving can be achieved. It is important to combine the matrices in the correct order as matrix multiplication is not commutative.

The transformation matrices given in this text operate on points by pre-multiplication. The point representation appears to the right, with the operator matrix to its left. Thus the first operation must be placed to the left of the point to be transformed, the next operation appearing to the left of the previous one, and so on. In general if we have to apply several transformation matrices M_1, M_2, M_3, \dots in that time order to a point represented in homogeneous form as matrix X , the composite operation is given by

$$X' = \dots (M_3(M_2(M_1X))),$$

where X' is the homogeneous representation of the final point reached. A discussion above showed that the time order (but not the space order) of matrix multiplication could be changed, so the above line can be restated generally as

$$X' = \dots M_3 M_2 M_1 X,$$

and finally rearranged as

$$X' = (\dots M_3 M_2 M_1) X,$$

where the matrix product $(\dots M_3 M_2 M_1)$ represents the complete sequence of transformations. By multiplying these together, the sequence is reduced to a single matrix that can be applied to each of the points of the figure in turn. For example, if the figure has 1000 points, and the transformation comprises five standard transformations, the naïve approach would involve 5000 multiplications of a matrix by a point. By combining the matrices into one, at an initial cost of four matrix multiplications, there would subsequently be only 1000 multiplications of a matrix by a point. The 'concatenation' of all operations into a single matrix has enabled considerable time saving. This is the real reason for the use of this technique in computer graphics.

Returning to our specific example, the sequence of individual matrices gives the single matrix

$$M = T(3, -2)R(-90^\circ)T(-3, 2) = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Remembering that we cannot change the positional order of these matrices, but we may multiply them in any time order; multiplying the right-hand pair first, we get

$$M = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 3 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 5 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

This single matrix can now be applied to all points of the figure, effectively cutting out the middle two phases of fig 6.2.

The determinant of a transformation matrix (which can be found by a computer routine as discussed above) indicates the change of area effected by that transformation. If we denote the determinant of matrix A as $\det[A]$, we have

$$\det[T(t_x, t_y)] = 1,$$

$$\det[S(s_x, s_y)] = s_x s_y,$$

$$\det[R(\theta)] = \cos^2(\theta) + \sin^2(\theta) = 1.$$

The last case uses a standard trigonometric identity that can be easily established from Pythagoras' theorem. These values clearly indicate the change of area given

when these transformations are applied to a two-dimensional shape. Scaling and rotation do not affect area, but the scaling factors s_x and s_y stretch or squash an object linearly, so their combined effect on area is represented by their product. A theorem on determinants of square matrices tells us that when matrices are multiplied, the determinant of the result is the product of the determinants of the original matrices. From this, it is clear that the determinant of a combination of affine transformations gives the overall area change effected by that combination.

Matrices, Transformations and Homogeneous Coordinates: Three Dimensions

Similar methods are used in the three-dimensional world. We move directly to the homogeneous representation of a three-dimensional point (x, y, z) in Cartesian

coordinates as $\begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$ when the 'weight' $w \neq 0$. For purposes other than performing the

perspective projection transformation (discussed later), we take $w = 1$ for simplicity,

representing the point (x, y, z) as $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$. Matrices for the operations of translation and

scaling are obvious extensions of the 2D case. Rotation must now be defined about the three coordinate axes as shown below.

Translation

$$\begin{aligned} x' &= x + t_x, \\ y' &= y + t_y, \\ z' &= z + t_z. \end{aligned}$$

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Scaling

$$\begin{aligned} x' &= s_x x, \\ y' &= s_y y, \\ z' &= s_z z. \end{aligned}$$

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotation about the x-axis

$$\begin{aligned} x' &= x, \\ y' &= y \cos(\theta) - z \sin(\theta), \\ z' &= y \sin(\theta) + z \cos(\theta). \end{aligned}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotation about the y-axis

$$\begin{aligned}x' &= x \cos(\theta) + z \sin(\theta), \\y' &= y, \\z' &= -x \sin(\theta) + z \cos(\theta).\end{aligned}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotation about the z-axis

$$\begin{aligned}x' &= x \cos(\theta) - y \sin(\theta), \\y' &= x \sin(\theta) + y \cos(\theta), \\z' &= z.\end{aligned}$$

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

Shear with z unchanged

$$\begin{aligned}x' &= x + k_x z, \\y' &= y + k_y z, \\z' &= z.\end{aligned}$$

$$Sh_z(k_x, k_y) = \begin{bmatrix} 1 & 0 & k_x & 0 \\ 0 & 1 & k_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The first two methods, for translation and scaling, are very similar to their 2D equivalents with the addition of an extra z coordinate, which needs an extra row and column for the transformation matrix.

Rotations are almost always described in 3D as being ‘about an axis’, invoking the idea of a ‘right-hand screw’ for positive rotation, in the sense of tightening a standard screw with a screwdriver pointing in the positive direction of the axis of rotation. If the right thumb points in the positive direction of this axis, the fingers curl naturally in the sense of rotation. This form of definition is specific to three-dimensional cases. No such axis of rotation exists in 2D, unless we artificially create an axis outside the 2D universe, and in four- and higher-dimensional systems there is more than one ‘axis’ used for each rotation. At first reading, the concept of rotation in 4D space may seem a bit bewildering, but there is little difficulty in extending mathematical laws to cope with extra dimensions. We have just extended a 2D transformation system to one in 3D, and we discussed in chapter 3 how ‘objects’ in four and higher dimensions can be created. These are sometimes useful for computer graphics as well as in other disciplines. In 4D systems, it is far easier to consider rotation as ‘from one axis towards another’. For example, in a 4D system with axes (x, y, z, u) , rotation ‘from x towards y ’ leaves z and u unchanged, so the rotation is conceptually ‘about z and u ’. The description of rotation in the sense ‘from x towards y ’ can be used in all coordinate systems from 2D upwards. It represents the only form of rotation in 2D systems, rotation about z in 3D, rotation ‘about z and u ’ in 4D, and so on, giving a uniform way of describing all cases.

We are particularly concerned with the 3D case here, so it is worth describing the three possible forms of rotation explicitly (fig 6.3). Rotation about the x -axis turns y towards z ; rotation about the y -axis turns z towards x ; rotation about the z -axis turns x towards y . Once the alphabetic x to y to z to x ... cycle is identified, this seems a natural system to adopt. As in other texts, the more convenient form of ‘ R_x ’ is used here rather than the cumbersome ‘ $R_{y \text{ to } z}$ ’, but knowledge of this cyclic effect may help readers to understand the process. This was discussed in chapter 4, but it does no harm to revisit the concept.

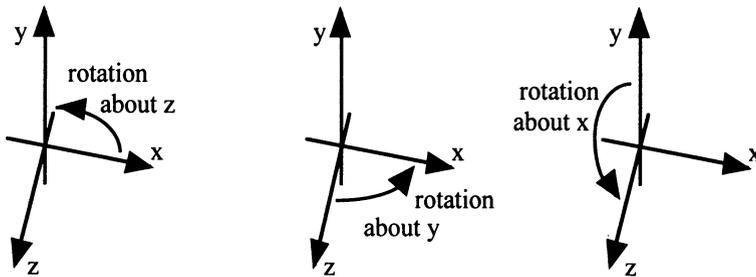


Figure 6.3 Rotations about the three coordinate axes

As we have to consider three forms of rotation in a 3D system, we must also take into account three forms of shear, each of which leaves one of the three spatial coordinates unchanged. The single example given above leaves the z coordinate unchanged, shearing in the x and y directions. The shear matrices

$$\text{Sh}_x(k_y, k_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ k_y & 1 & 0 & 0 \\ k_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \text{Sh}_y(k_z, k_x) = \begin{bmatrix} 1 & k_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & k_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

leave the x coordinate and the y coordinate unchanged respectively, shearing in the two altered coordinate directions. To visualize this, imagine having a stacked pack of cards on a table, then drag a finger across the edge of the pack to make it non-vertical. The vertical location of cards has not changed, but position in both horizontal directions can be changed. Suffixes in the above formulae are consistently given in the now familiar alphabetical cycle. Only one shear matrix was given in the main list above, for brevity. As for 2D systems, many texts do not mention shearing explicitly, as it can be performed by a combination of rotation and scaling operations, but having the shear operation directly available can add to the usability of modelling and computer graphics systems.

If we have a sequence of transformations, M_1, M_2, M_3, \dots in matrix form, to be performed on the homogeneous point $X = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, the complete operation can be

represented as

$$X' = \dots M_3 M_2 M_1 X = MX, \text{ say,}$$

where the single matrix $M = \dots M_3 M_2 M_1$ represents the complete sequence of transformations. As in the two-dimensional case, pre-computation of M can lead to considerable time savings in performing such a sequence on an object defined by many vertices in three dimensions. As in the 2D case, this is illustrated with a specific example.

Suppose we want to rescale an object by a factor of 2 in the x direction about the origin, and subsequently rotate it by 45° positively about the x-axis, keeping the point (1, 1, 1) fixed. Remembering that $\cos(45^\circ) = \sin(45^\circ) = 1/\sqrt{2}$, the required matrices for the composite operation can be written down from the standard formulae given above. The composite matrix representing the complete set of operations required is found by repeated pre-multiplication as the process develops, rather than just identifying the matrices to be used and then multiplying the stored values at the end. In a computer context, this is more memory efficient as just the one intermediate result is carried forward each stage, and no more costly in terms of time taken.

• Scale by 2 in the x direction: $S(2, 1, 1) = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

• Place (1, 1, 1) at the origin: $T(-1,-1,-1) = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Composite matrix:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• Rotate about x-axis by 45°: $R_x(45^\circ) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Composite matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & -\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(in this stage, we note that $2/\sqrt{2} = \sqrt{2}$, as $2 = \sqrt{2} * \sqrt{2}$).

• Replace the origin to (1, 1, 1): $T(1, 1, 1) = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Composite matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & -1 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & -\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 1 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 1-\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final matrix generated,

$$M = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 1 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 1-\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

represents all operations. Applying this overall transformation to a cube, say, with eight vertices, at the initial cost of three matrix multiplications, we now only have to perform eight direct matrix by point multiplications. Had we not 'concatenated' the series of operations into one matrix in this way, four such operations would have been performed on each point – a total of 24 matrix by point multiplications. With more complex objects, the savings are more dramatic.

As a check on our calculations, it is illustrative to apply the resulting matrix to a few points. We consider the point (0.5, 1, 1). The scaling in x will place this point at (1, 1, 1), the centre of the subsequent rotation, so it will not be moved further by

the rotation. Applying our transformation M to the point $X = \begin{bmatrix} 0.5 \\ 1 \\ 1 \\ 1 \end{bmatrix}$, we have

$$MX = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 1 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 1-\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

This can be interpreted as the Cartesian point (1, 1, 1) as required. As another example check, the point (1, 2, 2) will be moved to (2, 2, 2) by the x-scaling. A little thought indicates that rotation of 45° about the x-axis with centre of rotation (1, 1, 1) will place this resulting point on the plane $y = 1$, distant $1 + \sqrt{2}$ from the

z-axis. Applying matrix M to the representation $\begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix}$ of the point (1, 2, 2), we get

$$MX = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 1 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 1-\sqrt{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 1+\sqrt{2} \\ 1 \end{bmatrix}.$$

This gives the correct location for the final point as (2, 1, $1 + \sqrt{2}$).

Inverse of a Transformation Matrix

A relatively complicated routine for finding an inverse of a general square matrix was given above. For affine transformation matrices, inverses are easy to find. An inverse matrix 'undoes' the effect of the original matrix, so we can write down

$$\mathbf{2D} \quad T^{-1}(t_x, t_y) = T(-t_x, -t_y),$$

$$S^{-1}(s_x, s_y) = S(1/s_x, 1/s_y), \quad \text{provided } s_x \neq 0, s_y \neq 0,$$

$$R^{-1}(\theta) = R(-\theta),$$

$$\text{Sh}_y^{-1}(k_x) = \text{Sh}_y(-k_x), \quad \text{Sh}_x^{-1}(k_y) = \text{Sh}_x(-k_y).$$

$$\mathbf{3D} \quad T^{-1}(t_x, t_y, t_z) = T(-t_x, -t_y, -t_z),$$

$$S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z), \quad \text{provided } s_x \neq 0, s_y \neq 0, s_z \neq 0,$$

$$R_x^{-1}(\theta) = R_x(-\theta), \quad R_y^{-1}(\theta) = R_y(-\theta), \quad R_z^{-1}(\theta) = R_z(-\theta).$$

$$\text{Sh}_z^{-1}(k_x, k_y) = \text{Sh}_z(-k_x, -k_y), \quad \text{Sh}_x^{-1}(k_y, k_z) = \text{Sh}_x(-k_y, -k_z),$$

$$\text{Sh}_y^{-1}(k_z, k_x) = \text{Sh}_y(-k_z, -k_x).$$

Rotation of $-\theta$ undoes a rotation of θ . This result can also be checked by multiplying original matrices by their inverses. Each gives the unit matrix I .

In multiplying together sequences of transformation matrices, we have

$$M^{-1} = (\dots M_3 M_2 M_1)^{-1} = M_1^{-1} M_2^{-1} M_3^{-1} \dots$$

This can be justified by considering the product

$$\begin{aligned} (\dots M_3 M_2 M_1)(M_1^{-1} M_2^{-1} M_3^{-1} \dots) &= \dots M_3 M_2 (M_1 M_1^{-1}) M_2^{-1} M_3^{-1} \dots \\ &= \dots M_3 M_2 I M_2^{-1} M_3^{-1} \dots \\ &= \dots M_3 M_2 M_2^{-1} M_3^{-1} \dots \\ &= \dots = I. \end{aligned}$$

Central matrices are repeatedly paired off, until the whole sequence is reduced to I , the unit matrix. This gives an easy way of creating the inverse of a sequence of transformation matrices at the same time as creating the product of the sequence. When a new transformation is absorbed into the sequence, its matrix pre-multiplies the existing transformation matrix and its inverse post-multiplies the composite inverse. Both processes go hand in hand; there is no need to use a difficult matrix inversion routine.

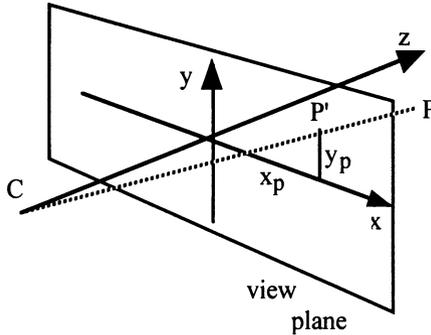


Figure 6.4 Perspective projection of a general point P onto a view plane using centre of projection C

Perspective Projection

Part of the process of viewing objects in 3D is to project them onto a 2D surface that will be mapped onto the viewing surface, such as a VDU screen or plotter. ‘Realistic’ effects are achieved by using perspective projection. We assume that a special view coordinate system has been set up for this purpose, with origin on this view plane and a pseudo observer (or centre of projection, C) at a point $(0, 0, -d)$ in this system, as shown in fig 6.4. These systems are often set up as left-handed sets of coordinates, as shown in this figure. The z-axis points away from the centre of projection, measuring the depth of an object into the scene. The other direction would give the more orthodox right-handed set of axes.

Given the location of a point P in space, the perspective projection calculates the position P' where the line CP intersects the view plane, the plane of $z = 0$ in this method. An alternative method places the centre of projection at the origin, with the view plane having equation $z = d$. The result for this alternative formulation is given later.

If we ‘look down’ upon fig 6.4 from the positive y-axis, fig 6.5 results. From this view, it is more easy to see how the x coordinate of P' can be found. The tan of

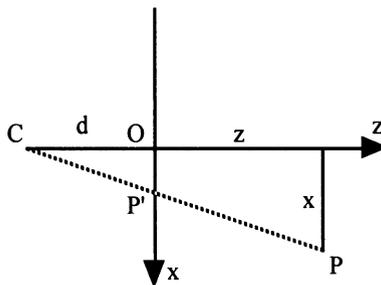


Figure 6.5 Vertical view of fig 6.4

angle $\angle OCP'$ can be found from two triangles as OP'/d or as $x/(d+z)$. As $OP' = x_p$, we have

$$\frac{x_p}{d} = \frac{x}{d+z},$$

so
$$x_p = \frac{dx}{d+z} = \frac{d}{d+z}x.$$

Similarly, the y coordinate of P' is given by

$$y_p = \frac{d}{d+z}y = \frac{dy}{d+z}.$$

Trivially, P' has z coordinate $z_p = 0$. Note that this method gives a zero divide error if $z = -d$. We cannot view a point that is alongside the observer. This makes physical sense. Most viewing systems get around this problem by including a 'near plane' to exclude any point with z less than some small positive value from the potentially visible region. This also avoids the situation of points behind the observer being projected in a negative sense onto the view plane. It could be useful in real life to have 'eyes in the back of one's head' but this would be most confusing in a computer graphics system. A 'perspective transformation matrix' to

perform this operation, given a homogeneous representation of P as $P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, is

$$\text{Per}(d) = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{bmatrix}.$$

This requires some explanation. P' is given by

$$P' = \text{Per}(d)P = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ 0 \\ d+z \end{bmatrix}.$$

The result is a homogeneous matrix that does not have weight $w = 1$. Above, the convention has been to hold $w = 1$ for simplicity, as in most computer graphic texts; this is the single case when the more general form is required. The Cartesian representation of P' is found by dividing its first three elements by the fourth, its 'weight' w . This gives the required result,

$$(x_p, y_p, z_p) = \left(\frac{dx}{d+z}, \frac{dy}{d+z}, 0 \right).$$

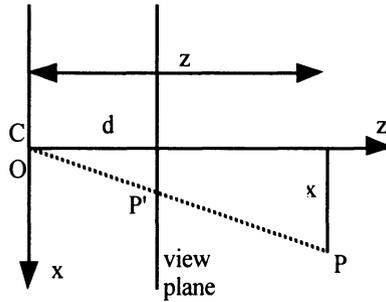


Figure 6.6 Alternative perspective set-up with origin at the centre of projection

In many algorithms, for example hidden surface algorithms that determine which of several objects is visible from the point of view of an observer, it is necessary to pass on the ‘depth information’ z . In such cases, the matrix above can be used to evaluate x_p and y_p , but a special addition to the algorithm can carry the value of z through unchanged as $z_p = z$.

An alternative method sets up the view coordinate system with its origin at the centre of projection C and the view plane passing through $(0, 0, d)$ (fig 6.6 has O and C coincident). The equations for the position of P' (x'_p, y'_p, z'_p) are

$$x'_p = \frac{dx}{z}, y'_p = \frac{dy}{z} \text{ and } z'_p = d.$$

The matrix $Per'(d) = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$ can be used to perform this version of

perspective projection. As in the previous method, points alongside the centre of projection (alongside the ears of an observer, with $z = 0$ in this formulation) would give a ‘zero divide’ error and cannot be viewed by this method. Once more, use of a ‘near plane’ elsewhere in the system prevents this anomalous situation and avoids the projection of points from behind the centre of projection onto the view plane.

Applying $Per'(d)$ to the homogeneous representation of P as $P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$, using the

general form of homogeneous point representation with weight w , we get

$$\begin{bmatrix} wx'_p \\ wy'_p \\ wz'_p \\ w \end{bmatrix} = Per'(d)P = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix}.$$

The result can be reduced to its Cartesian equivalent by dividing other terms by the ‘weight’ z , to give $x'_p = \frac{dx}{z}$, $y'_p = \frac{dy}{z}$ and $z'_p = d$, as required. As above, the value

of z'_p produced may be ignored, the previous value of z being carried through for use in algorithms that need depth information.

The different structures of these matrices compared with the affine transformation matrices (those for translation, scaling, rotation and all combinations of them have bottom row $[0\ 0\ 0\ 1]$) pose difficulties with computer implementation. Space and time-saving considerations make it attractive to eliminate this bottom row in storage of standard affine transformations, giving special concatenation routines that take this into account rather than use 'off the shelf' matrix multiplication. The creation of a perspective view in computer graphics comprises a modelling stage (when objects are composed to create the scene, as a photographer would arrange objects to be photographed), a viewing stage (the view parameters are set up, equivalent to the photographer locating the tripod, choosing a lens) and a projection and rendering stage (as in the actual exposure of the film).

The use of a 'near plane' to avoid division by zero in perspective projection has been introduced. Other forms of clipping are used in computer graphics, to eliminate all objects or parts of objects outside the field of view, or to cut out objects so far away as not to contribute meaningfully to an image. This has to intrude somewhere within this three-stage image synthesis process, so the smooth use of matrix concatenation has to be interrupted into two stages. Modelling and viewing can be concatenated into one matrix operation, although it may be useful to store intermediate data on the modelling stage if several views of the same scene are to be taken. It is more difficult to include the projection transformation as part of the same process, as the abbreviated form of matrix convenient for modelling transformations is no longer suitable. These two forms of process may be treated separately if specialized matrix operators are used for affine transformations. It is worth noting that some sophisticated modellers allow non-affine forms of space warping such as tapering or twisting as modelling transformations. Thus, the smooth concatenation of matrices must be interrupted at one or more stages in the process of image synthesis, sometimes called the 'computer graphics pipeline'. Many texts omit to mention this. Although this means that two or three matrix operations per point are needed, the use of matrix concatenation can still cut down considerably the amount of calculation needed to view a complex object; it is still a time-saving method. Some implementation issues are discussed later in the chapter.

Parallel projection is a simple alternative to perspective projection. It is equivalent to viewing a point from a centre of projection placed an enormous distance away from the view plane (technically, a parallel projection is the limit of a perspective projection as d approaches infinity). When all created points have been converted to a view coordinate system (as in figs 6.4 and 6.5), the parallel projection simply takes the x and y coordinates from the view projection. The equations for the 'view plane origin' form of perspective projection are

$$x_p = \frac{dx}{d+z} \quad \text{and} \quad y_p = \frac{dy}{d+z}.$$

If we divide top and bottom of the fractions by d , this can be rearranged as

$$x_p = \frac{x}{1+z/d} \quad \text{and} \quad y_p = \frac{y}{1+z/d}.$$

It is clear from this form that as d becomes very large the term z/d gets smaller because the denominator of z/d increases. In the limit, as d approaches infinity, we take $z/d = 0$, so the limiting form of the perspective projection gives the expected parallel projection formulae

$$x_p = x \quad \text{and} \quad y_p = y.$$

We have here taken a ‘limit as d approaches infinity’; this is similar to the ‘limit as δx approaches zero’ that we used in developing the differential and integral calculus in chapter 5.

Perspective projection could also be used to distort geometric models adding a non-affine option to the modelling process. It should be clear that perspective projection is a non-affine process from the presence of ‘vanishing points’ in classical perspective images or the visual effect of straight railway lines converging upon a distant point. A defining property of affine transformations is the preservation of parallel lines. Under perspective transformation, parallel lines are made to converge to a vanishing point, and are therefore non-parallel. The effect is equivalent to distorting an oval to an egg shape, broadening some parts and narrowing others. Distortion using centre of projection $(0, 0, -d)$ can be applied to both x and y variables of an object’s vertices, or either x or y alone. Care must be taken in carrying the z coordinate through this process – it can be retrieved through the ‘weight’. Equivalent formulae can be devised for perspective transformation with the centre of projection on the x or y coordinate axes. If allowed, this process makes concatenation of modelling operations very difficult. For example, direct matrix implementation of a sequence of perspective modelling transformations, one along each coordinate axis, without extraction of the perspective axis coordinate would reduce all vertices to the origin $(0, 0, 0)$. A way to safeguard against this in sequences of perspective modelling operations, perhaps with different axes, is to evaluate vertices after every perspective operation, losing the benefit of matrix concatenation. If required, perspective transformations could be incorporated into modelling as an initial stage, before passing object vertices on to the affine modelling and viewing stage.

Computer Implementation of Matrix Methods

Matrices are excellent conceptual devices for *devising* ways of performing affine and projective transformations. They are often described as methods for *performing* such transformations, but, once the concepts from matrix methods are developed, better time performance is achieved by specially tailored routines. A similar situation was outlined in chapter 2; ways of storing and manipulating numbers in a computer are based on the concepts of binary numbers, but the implementation uses a number of convenient divergences. Many texts hint at potential efficiencies leading from the peculiar structures of transformation matrices, but few give details of what they mean; fewer discuss limitations introduced by such efficiencies. Concatenation of two 3D transformation matrices (order 4×4) takes 4^3 or 64 multiplications using a

standard matrix multiplication routine. Ignoring, for the time being, perspective transformations, the standard modelling and viewing matrices all have fourth rows [0 0 0 1]. If this is taken into account, concatenation can be performed using only 36 multiplications. Further savings are possible using specific routines for concatenation of different forms of transformations. As shown in the methods suggested below, in 3D translations take only 3 additions, scalings use 12 multiplications, rotations involve 16 multiplications and 8 additions or subtractions after the necessary calculations of cosine and sine terms. Such savings are desirable in the time-consuming activity of computer graphics. The ‘working parts’ of such transformations fit only three rows of four elements, compared to the 16 elements using direct matrix methods, so space saving is also possible, although this is less of an issue as relatively few such matrices need to be stored.

A further divergence from the mathematical matrix model gives potential memory and time benefits. Accessing an element of a two-dimensional array is more time consuming than accessing an element of a one-dimensional array, so it is more efficient to code 3D transformations as 12 elements of a one-dimensional array, ensuring through the code that elements are properly matched. This can be done by storing a general 3D transformation matrix B as a 12 element one-dimensional array A,

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where a_i is equivalent to the C language array element $A[i]$. A similar device can hold a general 2D transformation in a six-element array. Whilst this method does give some memory and time savings, some developers may not find these factors important and may prefer to implement more standard direct matrix methods requiring fewer functions to be developed. Some sample manipulations of the ‘pared down’ 3D version are given below, on the understanding that this is just *a* method, not the only one or necessarily the ‘best’ for all circumstances. Similar methods are available for 2D manipulation; these will not suffer the problems caused by the special case of perspective transformations.

The fourth row of a perspective transformation is not of the form [0 0 0 1]. Ingenuity in interpretation of the array structure that holds transformations can allow this to be accommodated without defining a different data type. Direct application of the perspective transformation converts the ‘z’ depth coordinate uniformly to the constant z value of the view plane for all vertices, so this can be ignored. However, the value of z used in hidden-surface or line calculations can be recovered from the non-zero weight (directly when the origin is at the centre of projection, or by subtracting d if the origin is on the view plane), which is calculated from the fourth row of the overall matrix. This row can be held in place of the unneeded values in locations 8 to 11 of the transformation array. Care must then be taken in the correct interpretation of these array elements. This form of operation is shown below.

Following is a typical computer graphics sequence of transformations used in rendering a 3D scene, with the type of coordinate generated by each shown:

- modelling transformations used to define objects (3D world coordinates);
- viewing transformations (3D view coordinates);
- perspective transformation on all vertices (2D window coordinates and z);
- window to viewport transformation (2D device coordinates).

These could all be concatenated to one operation, preferably using ‘pseudo-matrix’ methods. Many texts state this blandly, but do not indicate the special treatment that must be given to the perspective transformation. This process has to be broken into at some stage in classical image synthesis algorithms to clip vertices, edges, faces and parts of edges and faces that lie outside the visible region defined in the viewing transformations. Clipping after perspective transformation is an easier process, involving simple comparisons of coordinates against fixed limits, compared to the more difficult testing of points against general planes used in clipping before perspective projection. The disadvantage of the ‘project first, clip last’ strategy is that all vertices in the data set must be projected; this is unsuitable when a large number of objects lie outside the potentially visible region. Sophisticated pre-selection strategies are often used in such cases to eliminate obviously external objects before all vertices that define objects (as described in chapter 8) pass through the modelling and viewing transformation stages, so the extra cost of performing perspective projection as part of this process is relatively small. Systems designers must make a strategic decision as to the positioning of their clipping routine in the general 3D viewing pipeline. The ordering adopted below is to clip after the perspective transformations has been performed, allowing concatenation of all 3D transformations into one operation. This neatly separates the 2D operations of window to viewport mapping from the 3D routines.

Examples are given below to show how such a matrix-operated affine transformation system could operate in a 3D computer graphics system. Routines set up initial matrices of the three standard transformations; these are concatenated with an existing transformation matrix using a multiplication routine to build a composite transformation matrix. (Routines for concatenating rotation about x and y and shear are not shown for brevity; the concept of how to develop these routines should be clear.) Initialization using a ‘near unit’ matrix (an identity matrix of order 4 with its bottom row sliced off) would avoid the need for separate translation, scaling and rotation matrix creation routines, but having such routines does eliminate one matrix multiplication. Suffixes indicating the content of a one-dimensional array are used, so that they can be directly coded, but operations are shown below in matrix formation to indicate their relationships with their matrix derivations. If speed is the major consideration in generating routines based on these formulae, inelegant but fast direct allocation of all elements is the best method. This can increase speed by avoiding the index checks used in neater looping methods.

3D Initial Matrices

$$\text{With } A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

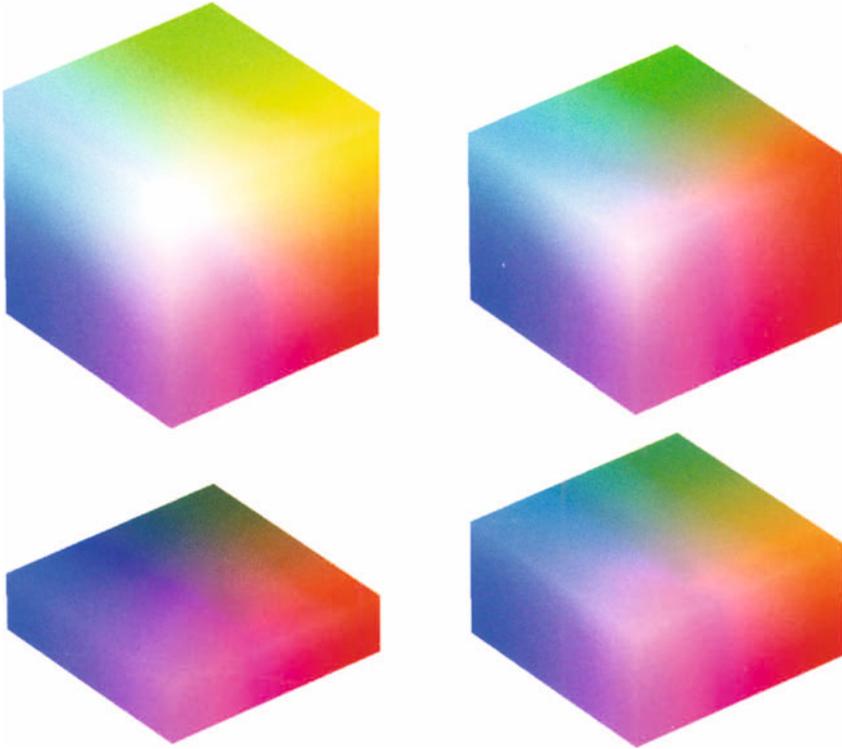


Plate 3.1 A (r, g, b) cube (top left) and, clockwise, slices through it at 0.75, 0.50 and 0.25 respectively of the green axis



Plate 3.2 Additive colour – combinations of red, green and blue – produce cyan, magenta, yellow and white



Plate 3.3 The (r, g, b) colour (0.2, 0.8, 0.6); its colour may be distorted due to its printed representation in (c, m, y)



Plate 3.4 A (h, s, v) cone, lines (anticlockwise) show the hues of red, magenta, blue, cyan, green, and yellow

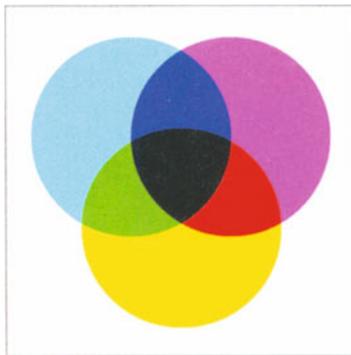


Plate 3.5 Subtractive colour – combinations of cyan, magenta and yellow – produce red, green, blue and black



Plate 5.1 A mathematical cone with two nappes; it extends to infinity beyond the bounds of the image

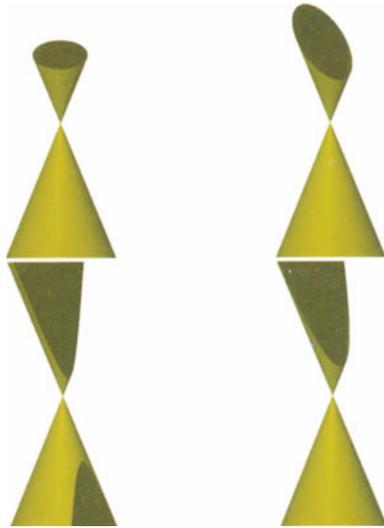


Plate 5.2 Slices at different angles through the cone revealing (clockwise from top left) a circle, ellipse, parabola and hyperbola

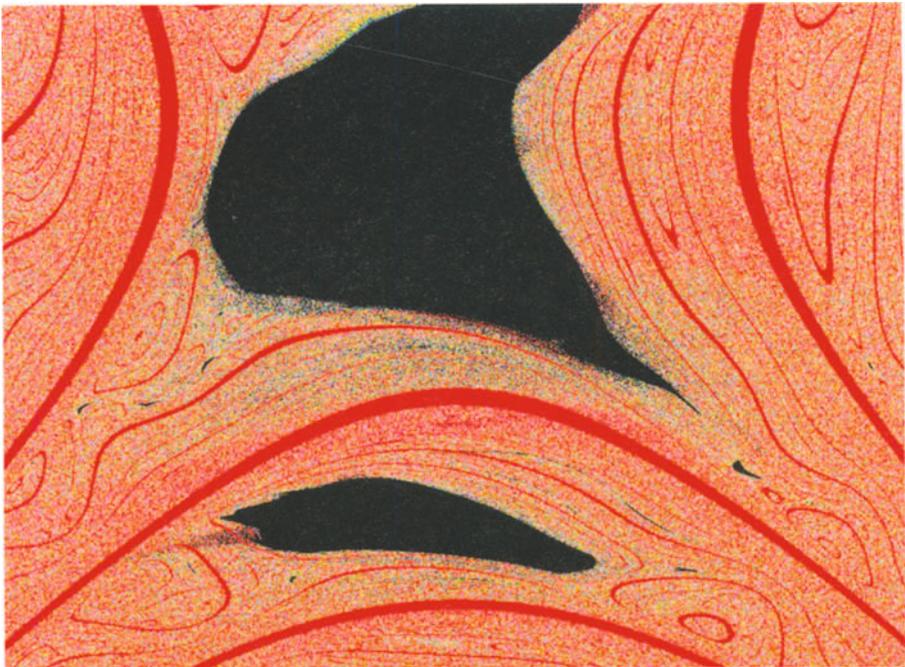


Plate 8.1 A chaotic pattern produced using a Mandelbrot type iteration in $-2 < x < 2$, $-1.5 < y < 1.5$ for the function $x' = \tan(x^2 - y^2) + c_x$, $y' = \cos(x^2 + y^2) - c_y$. Many functions produce interesting patterns



Plate 8.2 A tree generated by Paul Briggs using parametric L-systems

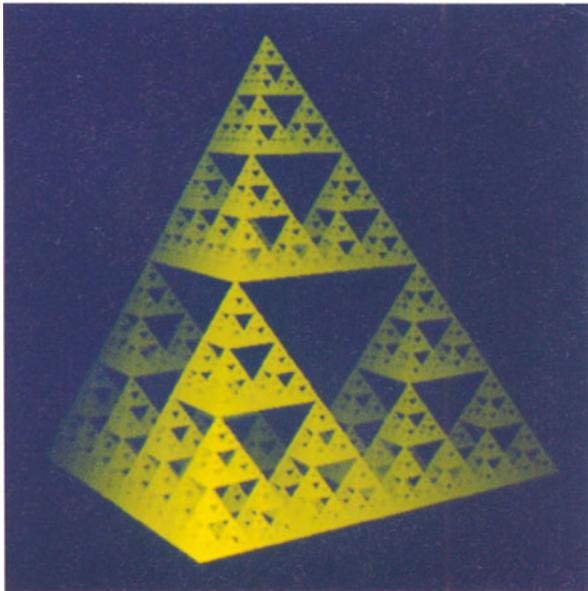


Plate 8.3 A Sierpinski tetrahedron generated by Aurelio Campa using 3D IFS



Plate 8.4 A fractal cove of 10 IFS trees using some non-affine transformations

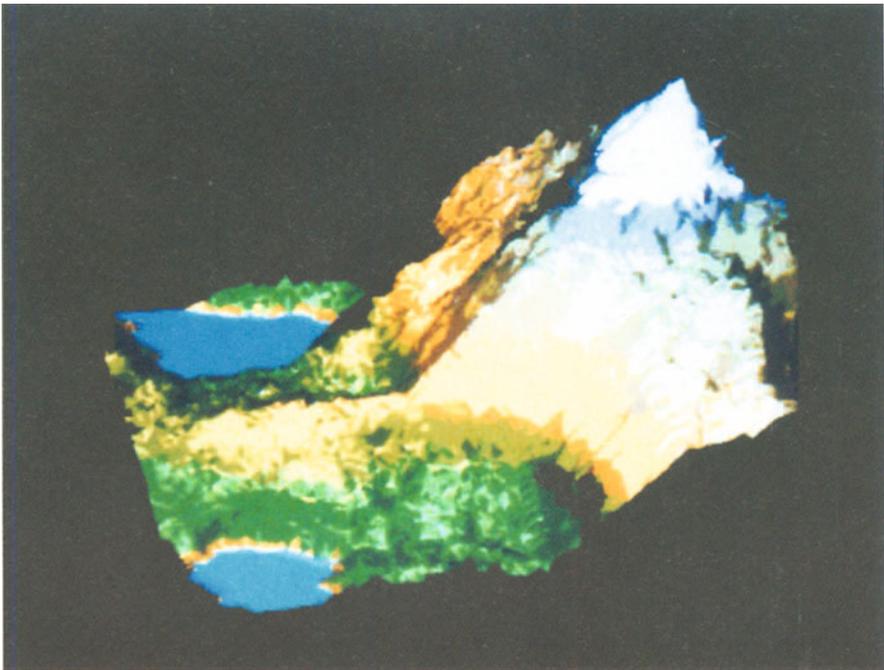


Plate 8.5 A terrain model by Denis Crampton using functional variation of height to produce flatter valley features without making mountain peaks over-pointed

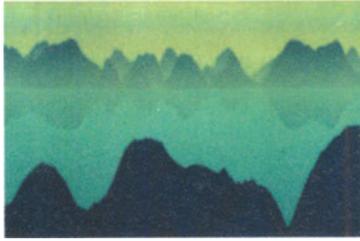


Plate 8.6 A landscape created by Semannia Luk Cheung with the support of John Vince

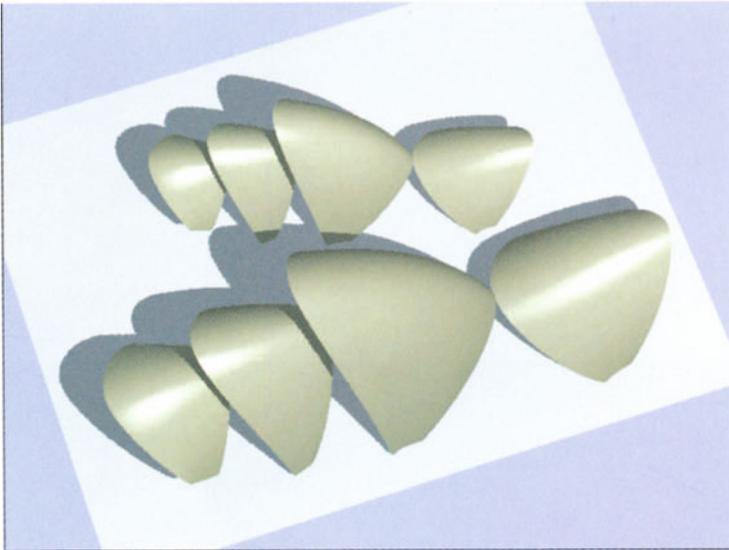


Plate 9.1 Eight 'shells', each created as a pair of cubic Bézier spline patches, joined at the central 'ridge' of symmetry. They are depicted using the exact object method (chapter 10) with Phong shading, highlights and shadows. Some pixellation is seen due to the random nature of the image generation, in which points are scattered across the two Bézier patch parameters

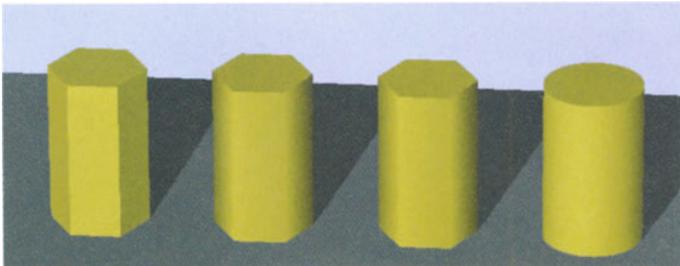


Plate 10.1 Four 'cylinders' displayed using, from the left, Lambert (flat) shading, Gouraud shading, Phong shading and exact object rendering. The first three examples are created as octagonal based prisms; there is little observed difference between Gouraud and Phong examples, as no specular highlights are seen



Plate 10.2 An abstract construction of cones, a cylinder and spheres. This shows how the z-buffer copes with interpenetration, the shadow buffer copes with complex cast shadows and how gloss factors may vary (higher for the rear sphere than the other two glossy objects)



Plate 10.3 Part of a maple tree modelled by John Thum. Texture mapping is used to create surface detail

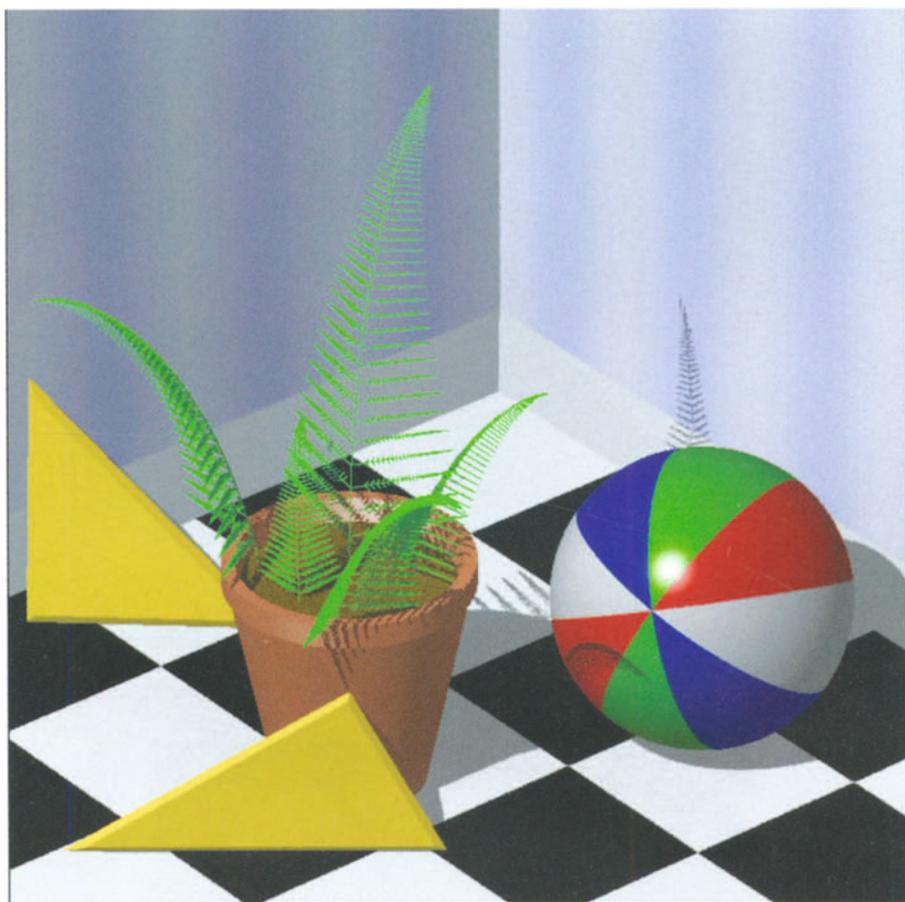


Plate 10.4 This plate shows many of the effects described in the book. Curved surfaces use exact object rendering; the walls, ball and floor are procedurally texture mapped and stochastically anti-aliased although some 'jaggies' still appear; the plants are generated by 3D IFS; a z-buffer and shadow buffer was used and the ball has a specular highlight

- translation: $a_i = 0$ for $i = 0$ to 11 , except for $a_0 = a_5 = a_{10} = 1$, $a_3 = t_x$, $a_7 = t_y$, $a_{11} = t_z$;
- scaling: $a_i = 0$ for $i = 0$ to 11 , except for $a_0 = s_x$, $a_5 = s_y$, $a_{10} = s_z$;
- rotation(x): $a_i = 0$ for $i = 0$ to 11 , except for $a_5 = \cos(\theta)$, $a_9 = \sin(\theta)$, $a_6 = -a_9$, $a_{10} = a_5$, $a_0 = 1$;
- rotation(y): $a_i = 0$ for $i = 0$ to 11 , except for $a_{10} = \cos(\theta)$, $a_2 = \sin(\theta)$, $a_8 = -a_2$, $a_0 = a_{10}$, $a_5 = 1$;
- rotation(z): $a_i = 0$ for $i = 0$ to 11 , except for $a_0 = \cos(\theta)$, $a_4 = \sin(\theta)$, $a_1 = -a_4$, $a_5 = a_0$, $a_{10} = 1$;
- shear(x same): $a_i = 0$ for $i = 0$ to 11 except for $a_0 = a_5 = a_{10} = 1$, $a_4 = k_y$, $a_8 = k_z$;
- shear(y same): $a_i = 0$ for $i = 0$ to 11 except for $a_0 = a_5 = a_{10} = 1$, $a_9 = k_z$, $a_1 = k_x$;
- shear(z same): $a_i = 0$ for $i = 0$ to 11 except for $a_0 = a_5 = a_{10} = 1$, $a_2 = k_x$, $a_6 = k_y$.

3D Concatenation Routines

The following assume that the specified transformation is performed after a general

transformation $A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$ and use $c = \cos(\theta)$ and $s = \sin(\theta)$ in the

rotation matrices:

• translation:
$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & (a_3 + t_x) \\ a_4 & a_5 & a_6 & (a_7 + t_y) \\ a_8 & a_9 & a_{10} & (a_{11} + t_z) \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

• scaling:
$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x a_0 & s_x a_1 & s_x a_2 & s_x a_3 \\ s_y a_4 & s_y a_5 & s_y a_6 & s_y a_7 \\ s_z a_8 & s_z a_9 & s_z a_{10} & s_z a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$\bullet \text{ rotation}(z): \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (ca_0 - sa_4) & (ca_1 - sa_5) & (ca_2 - sa_6) & (ca_3 - sa_7) \\ (sa_0 + ca_4) & (sa_1 + ca_5) & (sa_2 + ca_6) & (sa_3 + ca_7) \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

3D Transformation Applied to a Vertex

$$\bullet \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} (a_0x_0 + a_1x_1 + a_2x_2 + a_3) \\ (a_4x_0 + a_5x_1 + a_6x_2 + a_7) \\ (a_8x_0 + a_9x_1 + a_{10}x_2 + a_{11}) \\ 1 \end{bmatrix}.$$

Concatenation of perspective transformation

If the **origin is on the view plane**, with centre of projection at $(0, 0, 0, -d)$,

$$\bullet \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} da_0 & da_1 & da_2 & da_3 \\ da_4 & da_5 & da_6 & da_7 \\ 0 & 0 & 0 & 0 \\ a_8 & a_9 & a_{10} & (d + a_{11}) \end{bmatrix}.$$

All useful information in the latter matrix can be held efficiently by reinterpreting the 12-element array transformation data structure as its first, second and *fourth* rows. If array B represents the result of applying this perspective transformation to a previous transformation (possibly composite) represented by array A, we can interpret B as

$$\bullet \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ * & * & * & * \\ b_8 & b_9 & b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} da_0 & da_1 & da_2 & da_3 \\ da_4 & da_5 & da_6 & da_7 \\ 0 & 0 & 0 & 0 \\ a_8 & a_9 & a_{10} & (d + a_{11}) \end{bmatrix}.$$

Applying this matrix to a general point gives

$$\cdot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ * & * & * & * \\ b_8 & b_9 & b_{10} & b_{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} (b_0x_0 + b_1x_1 + b_2x_2 + b_3) \\ (b_4x_0 + b_5x_1 + b_6x_2 + b_7) \\ * \\ (b_8x_0 + b_9x_1 + b_{10}x_2 + b_{11}) \end{bmatrix},$$

enabling the extraction of

$$\begin{aligned} w &= (b_8x_0 + b_9x_1 + b_{10}x_2 + b_{11}), \\ x'_0 &= (b_0x_0 + b_1x_1 + b_2x_2 + b_3)/w, \\ x'_1 &= (b_4x_0 + b_5x_1 + b_6x_2 + b_7)/w, \end{aligned}$$

and $x'_2 = w - d$

as the coordinates of the resulting point. x'_0 and x'_1 are the (x, y) coordinates of the original point (x_0, x_1, x_2) after transformation into the view window coordinates, x'_2 is the z 'depth' of (x_0, x_1, x_2) from the window after it has undergone modelling and viewing transformations. The unchanged value of w would be just as useful for hidden surface/line and depth cueing purposes, giving the 'depth' of the model vertex from the centre of projection, which represents the location of an observer of the model. This is described in more detail in chapter 10.

If the origin is at the centre of projection,

$$\cdot \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} da_0 & da_1 & da_2 & da_3 \\ da_4 & da_5 & da_6 & da_7 \\ da_8 & da_9 & da_{10} & da_{11} \\ a_8 & a_9 & a_{10} & a_{11} \end{bmatrix}.$$

Again, if B is the overall result of applying this perspective transformation after a transformation (possibly composite) represented by A, B can be encoded as

$$\cdot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ * & * & * & * \\ b_8 & b_9 & b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} da_0 & da_1 & da_2 & da_3 \\ da_4 & da_5 & da_6 & da_7 \\ da_8 & da_9 & da_{10} & da_{11} \\ a_8 & a_9 & a_{10} & a_{11} \end{bmatrix},$$

as information in the third row (replaced by asterisks “*”) is only a multiple of the fourth by d . Applying this matrix to a point gives

$$\cdot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ * & * & * & * \\ b_8 & b_9 & b_{10} & b_{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} (b_0x_0 + b_1x_1 + b_2x_2 + b_3) \\ (b_4x_0 + b_5x_1 + b_6x_2 + b_7) \\ * \\ (b_8x_0 + b_9x_1 + b_{10}x_2 + b_{11}) \end{bmatrix},$$

enabling the extraction of

$$x'_2 = (b_8x_0 + b_9x_1 + b_{10}x_2 + b_{11}),$$

$$\begin{aligned} x'_0 &= (b_0x_0 + b_1x_1 + b_2x_2 + b_3)/x'_2, \\ \text{and } x'_1 &= (b_4x_0 + b_5x_1 + b_6x_2 + b_7)/x'_2 \end{aligned}$$

as the coordinates of the resulting point. x'_0 and x'_1 are the (x, y) coordinates of the original point (x_0, x_1, x_2) after transformation into the view window coordinates, x'_2 is the z 'depth' of (x_0, x_1, x_2) after it has undergone modelling and viewing transformations.

Perspective Transformation Applied Directly to a Vertex (Usually after Clipping)

If the origin is placed on the view plane, with centre of projection $(0, 0, 0, -d)$,

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} dx_0 \\ dx_1 \\ 0 \\ d + x_2 \end{bmatrix};$$

division by the weight $w = (d + x_2)$ gives the transformed point

$$\begin{bmatrix} dx_0 / (d + x_2) \\ dx_1 / (d + x_2) \\ 0 \\ 1 \end{bmatrix}.$$

The value of $(w - d) = x_2$ should be associated in a data structure with the 2D view window coordinates $(dx_0/(d + x_2), dx_1/(d + x_2))$ as it is used to give depth priorities for hidden surface or line removal.

If the origin is at the centre of projection,

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} dx_0 \\ dx_1 \\ dx_2 \\ x_2 \end{bmatrix};$$

division by the 'weight' $w = x_2$ gives the transformed point

$$\begin{bmatrix} dx_0 / x_2 \\ dx_1 / x_2 \\ d \\ 1 \end{bmatrix}.$$

The value of $w = x_2$ should be associated in a data structure with the 2D view window coordinates $(dx_0/x_2, dx_1/x_2)$ as it is used to give depth priorities for hidden surface or line removal.

Summary

Users of computer graphics systems can continue blissfully unaware of the use of matrices within the routines that enable their creations. This chapter has developed the concept of matrices, with particular reference to the needs of computer graphics based on the transformations discussed in chapter 4, but with a sideways look at the historically important use of matrices in solving sets of linear equations. This is not wasted; it, too, is used in specialized computer graphics methods, such as rendering using radiosity (chapter 10). We have discussed how purely mathematical matrix methods can be subverted to produce more efficient computer usage.

Does this help the casual user? As a car driver may drive more efficiently with some basic knowledge of the workings of a car, the intention is that computer graphics creators may create better images and do so more efficiently if they have at least a vague knowledge of how their routines work. At least, they should understand why some things take a relatively long time to perform, even with continuing advances in the speed and memory capacity of modern computers.