CHAPTER

# Technical Background 2

This chapter serves as a prelude to the computer animation techniques presented in the remaining chapters. It is divided into two sections. The first serves as a quick review of the basics of the computer graphics display pipeline and discusses potential sources of error when dealing with graphical data. It is assumed that the reader has already been exposed to transformation matrices, homogeneous coordinates, and the display pipeline, including the perspective projection; this section concisely reviews these topics. The second section covers various orientation representations that are important for the discussion of orientation interpolation in Chapter 3.3.

## 2.1 Spaces and transformations

Much of computer graphics and computer animation involves transforming data (e.g., [2] [7]). Object data are transformed from a defining space into a world space in order to build a synthetic environment. Object data are transformed as a function of time in order to produce animation. Finally, object data are transformed in order to view the object on a screen. The workhorse transformational representation of graphics is the $4 \times 4$ transformation matrix, which can be used to represent combinations of three-dimensional rotations, translations, and scales as well as perspective projection.

A coordinate space can be defined by using a left- or a right-handed coordinate system (see Figure 2.1a,b). Left-handed coordinate systems have the $x$-, $y$-, and $z$-coordinate axes aligned as the thumb, index finger, and middle finger of the left hand are arranged when held at right angles to each other in a natural pose: extending the thumb out to the side of the hand, extending the index finger coplanar with the palm, and extending the middle finger perpendicular to the palm. The right-handed coordinate system is organized similarly with respect to the right hand. These configurations are inherently different; there is no series of pure rotations that transforms a left-handed configuration of axes into a right-handed configuration. Which configuration to use is a matter of convention. It makes no difference as long as everyone knows and understands the implications. Another arbitrary convention is the axis to use as the up vector. Some application areas assume that the $y$-axis is "up." Other applications assume that the $z$-axis is "up." As with handedness, it makes no difference as long as everyone is aware of the assumption being made. In this book, the $y$-axis is considered "up."

This section first reviews the transformational spaces through which object data pass as they are massaged into a form suitable for display. Then, the use of homogeneous representations of points and the $4 \times 4$ transformation matrix representation of three-dimensional rotations, translation, and scale are reviewed. Next come discussions of representing arbitrary position and orientation by a series

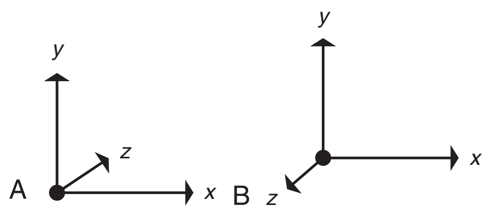


**FIGURE 2.1**

(a) Left-handed and (b) right-handed coordinate systems.

of matrices, representing compound transformations in a matrix, and extracting a series of basic transformations from a compound matrix. The display pipeline is then described in terms of the transformation matrices used to affect it; the discussion is focused on transforming a point in space. In the case of transforming vectors, the computation is slightly different (see Appendix B.3.2 for details). This section concludes with a discussion of error considerations, including orthonormalization of a rigid transformation matrix. Unless stated otherwise, space is assumed to be three-dimensional and right-handed.

### 2.1.1 The display pipeline

The *display pipeline* refers to the transformation of object data from its original defining space through a series of intermediate spaces until its final mapping onto the screen. The object data are transformed into different spaces in order to efficiently compute illumination, clip the data to the view volume, and perform the perspective transformation. This section reviews these spaces, their properties, the transformations that map data from one space to the next, and the parameters used to specify the transformations. The names used for these spaces vary from text to text, so they will be reviewed here to establish a consistent naming convention for the rest of the book. While an important process that eliminates lines and parts of lines that are not within the viewable space, clipping is not relevant to motion control and therefore is not covered.

The space in which an object is originally defined is referred to as *object space*. The data in object space are usually centered at the origin and often are created to lie within some limited standard range such as $-1$ to $+1$. The object, as defined by its data points (which are also referred to as its *vertices*), is transformed, usually by a series of rotations, translations, and scales, into *world space*, the space in which objects are assembled to create the environment to be viewed. Object space and world space are commonly right-handed spaces.

World space is also the space in which light sources and the observer are placed. For purposes of this discussion, *observer position* is used synonymously and interchangeably with *camera position* and *eye position*. The observer parameters include its *position* and its *orientation*. The orientation is fully specified by the *view direction* and the *up vector*. There are various ways to specify these orientation vectors. Sometimes the view direction is specified by giving a *center of interest* (COI), in which case the view direction is the vector from the observer or eye position (EYE) to the center of interest. The eye position is also known as the *look-from point*, and the COI is also known as the *look-to point*. The default orientation of “straight up” is defined as the observer’s up vector being perpendicular to the view direction and in the plane defined by the view direction and the global $y$-axis. A rotation away from this up direction will effect a tilt of the observer's head.

In order to efficiently project the data onto a view plane, the data must be defined relative to the camera, in a camera-centric coordinate system $(u, v, w)$; the $v$-axis is the observer's $y$-axis, or *up vector*, and the $w$-axis is the observer's $z$-axis, or *view vector*. The $u$-axis completes the local coordinate system of the observer. For this discussion, a left-handed coordinate system for the camera is assumed. These vectors can be computed in the right-handed world space by first taking the cross-product of the view direction vector and the $y$-axis, forming the $u$-vector, and then taking the cross-product of the $u$-vector and the view direction vector to form $v$ (Eq. 2.1).

$$\begin{array}{ll} w = COI - EYE & \text{view direction vector} \\ u = w \times (0,1,0) & \text{cross product with } y\text{-axis} \\ v = u \times w & \end{array} \tag{2.1}$$

After computing these vectors, they should be normalized in order to form a unit coordinate system at the eye position. A world space data point can be defined in this coordinate system by, for example, taking the dot product of the vector from the eye to the data point with each of the three coordinate system vectors.

Head-tilt information can be provided in one of two ways. It can be given by specifying an angle deviation from the straight-up direction. In this case, a head-tilt rotation matrix can be formed and incorporated in the world-to-eye-space transformation or can be applied directly to the observer's default $u$-vector and $v$-vector.

Alternatively, head-tilt information can be given by specifying an up-direction vector. The user-supplied up-direction vector is typically not required to be perpendicular to the view direction as that would require too much work on the part of the user. Instead, the vector supplied by the user, together with the view direction vector, defines the plane in which the up vector lies. The difference between the user-supplied up-direction vector and the up vector is that the up vector by definition is perpendicular to the view direction vector. The computation of the perpendicular up vector, $v$, is the same as that outlined in Equation 2.1, with the user-supplied up direction vector, $UP$, replacing the $y$-axis (Eq. 2.2).

$$\begin{array}{ll} w = COI - EYE & \text{view direction vector} \\ u = w \times UP & \text{cross product with user's up vector} \\ v = u \times w & \end{array} \tag{2.2}$$

Care must be taken when using a default up vector. Defined as perpendicular to the view vector and in the plane of the view vector and global $y$-axis, it is undefined for straight-up and straight-down views. These situations must be dealt with as special cases or simply avoided. In addition to the undefined cases, some observer motions can result in unanticipated effects. For example, the default head-up orientation means that if the observer has a fixed center of interest and the observer's position arcs directly, or almost so, over the center of interest, then just before and just after being directly overhead, the observer's up vector will instantaneously rotate by up to 180 degrees (see Figure 2.2).

In addition to the observer's position and orientation, the *field of view* (fov) has to be specified to fully define a viewable volume of world space. This includes an *angle of view* (or the equally useful *half angle of view*), *near clipping distance*, and *far clipping distance* (sometimes the terms *hither* and *yon* are used instead of *near* and *far*). The fov information is used to set up the *perspective projection*.

The visible area of world space is formed by the observer position and orientation, angle of view, near clipping distance, and far clipping distance (Figure 2.3). The angle of view defines the angle made between the upper and lower clipping planes, symmetric around the view direction. If this angle is

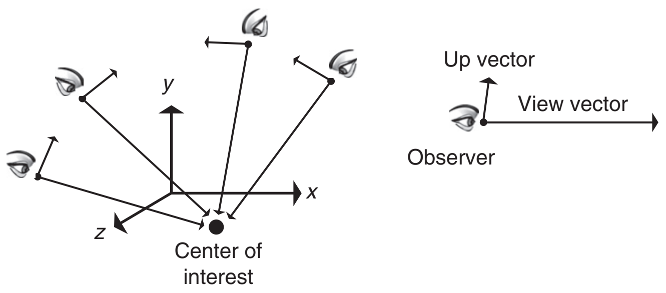


**FIGURE 2.2**

The up vector flips as the observer's position passes straight over the center of interest.

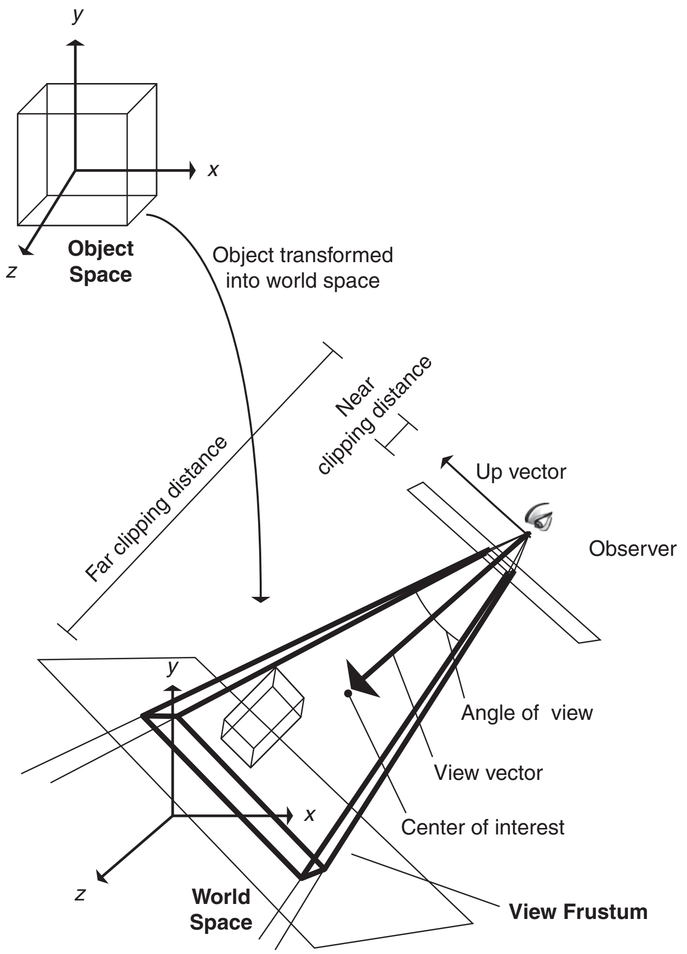


**FIGURE 2.3**

Object-space to world-space transformation and the view frustum in world space.

different than the angle between the left and right side clipping planes, then the two angles are identified as the *vertical angle of view* and the *horizontal angle of view*. The far clipping distance sets the distance beyond which data are not viewed. This is used, for example, to avoid processing arbitrarily complex data that is too far away to have much, if any, impact on the final image. The near clipping distance similarly sets the distance before which data are not viewed. This is primarily used to avoid division by zero in the perspective projection. These define the *view frustum*, the six-sided volume of world space containing data that need to be considered for display.

The view specification discussed above is somewhat simplified. Other view specifications use an additional vector to indicate the orientation of the projection plane, allow an arbitrary viewport to be specified on the plane of projection that is not symmetrical about the view direction to allow for off-center projections, and allow for a parallel projection. The reader should refer to standard graphics texts such as the one by Foley et al. [2] for an in-depth discussion of such view specifications.

In preparation for the perspective transformation, the data points defining the objects are usually transformed from world space to *eye space*. In *eye space*, the observer is positioned along the $z$-axis with the line of sight made to coincide with the $z$-axis. This allows the depth of a point, and therefore perspective scaling, to be dependent only on the point's $z$-coordinate. The exact position of the observer along the $z$-axis and whether the eye space coordinate system is left- or right-handed vary from text to text. For this discussion, the observer is positioned at the origin looking down the positive $z$-axis in left-handed space. In eye space as in world space, lines of sight emanate from the observer position and diverge as they expand into the visible view frustum, whose shape is often referred to as a *truncated pyramid*.

The *perspective transformation* transforms the objects' data points from eye space to *image space*. The perspective transformation can be considered as taking the observer back to negative infinity in $z$ and, in doing so, makes the lines of sight parallel to each other and to the (eye space) $z$-axis. The pyramid-shaped view frustum becomes a rectangular solid, or cuboid, whose opposite sides are parallel. Thus, points that are farther away from the observer in eye space have their $x$- and $y$-coordinates scaled down more than points that are closer to the observer. This is sometimes referred to as *perspective foreshortening*. This is accomplished by dividing a point's $x$- and $y$-coordinates by the point's $z$-coordinate. Visible extents in image space are usually standardized into the $-1$ to $+1$ range in $x$ and $y$ and from 0 to 1 in $z$ (although in some texts, visible $z$ is mapped into the $-1$ to $+1$ range). Image space points are then scaled and translated (and possibly rotated) into *screen space* by mapping the visible ranges in $x$ and $y$ ($-1$ to $+1$) into ranges that coincide with the viewing area defined in the coordinate system of the window or screen; the $z$-coordinates can be left alone. The resulting series of spaces is shown in Figure 2.4.

Ray casting (ray tracing without generating secondary rays) differs from the above sequence of transformations in that the act of tracing rays from the observer's position out into world space implicitly accomplishes the perspective transformation. If the rays are constructed in world space based on pixel coordinates of a virtual frame buffer positioned in front of the observer, then the progression through spaces for ray casting reduces to the transformations shown in Figure 2.5. Alternatively, data can be transformed to eye space and, through a virtual frame buffer, the rays can be formed in eye space.

In any case, animation is typically produced by one or more of the following: modifying the position and orientation of objects in world space over time, modifying the shape of objects over time, modifying display attributes of objects over time, transforming the observer position and orientation in world space over time, or some combination of these transformations.

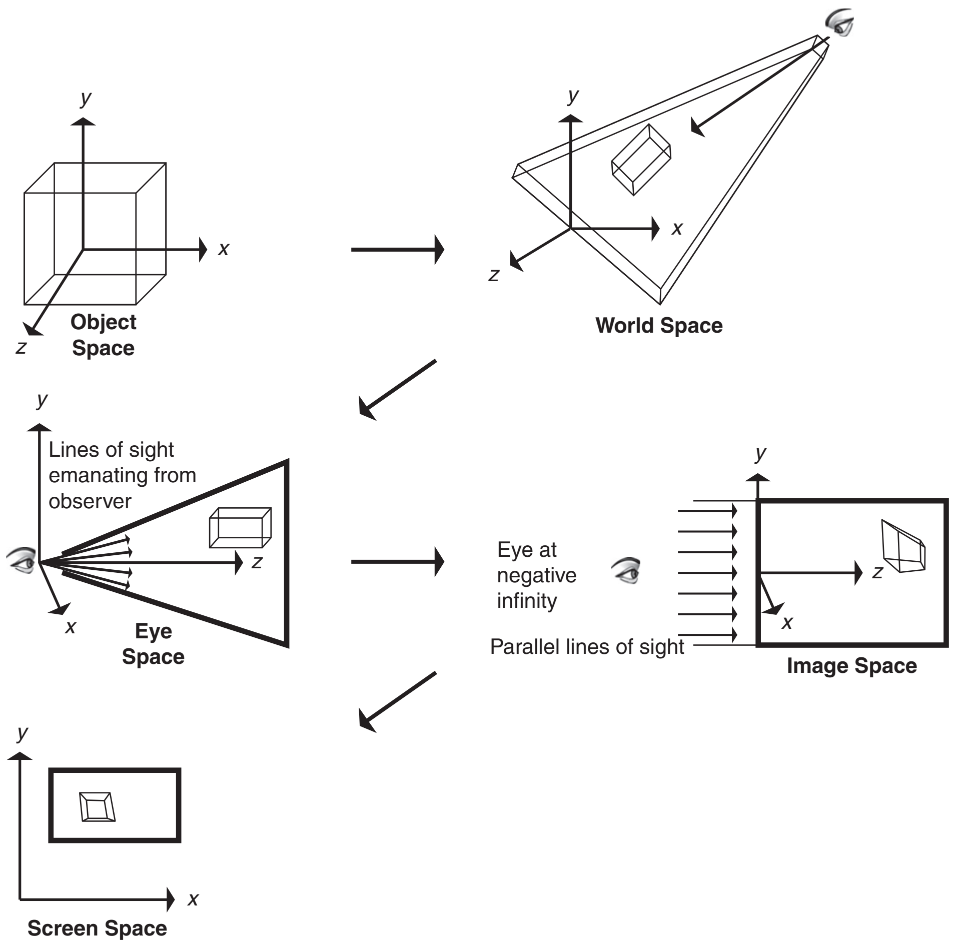


**FIGURE 2.4**

Display pipeline showing transformation between spaces.

## 2.1.2 Homogeneous coordinates and the transformation matrix

Computer graphics often uses a *homogeneous representation* of a point in space. This means that a three-dimensional point is represented by a four-element vector.[1] The coordinates of the represented point are determined by dividing the fourth component into the first three (Eq. 2.3).

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \equiv [x, y, z, w] \tag{2.3}$$

[1]Note the potential source of confusion in the use of the term *vector* to mean (1) a direction in space or (2) a $1 \times n$ or $n \times 1$ matrix. The context in which *vector* is used should make its meaning clear.

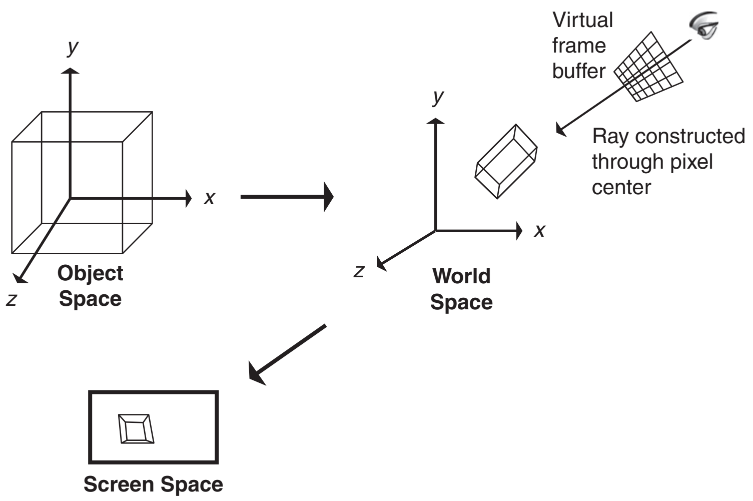


**FIGURE 2.5**

Transformation through spaces using ray casting.

Typically, when transforming a point in world space, the fourth component will be one. This means a point in space has a very simple homogeneous representation (Eq. 2.4).

$$(x, y, z) \equiv [x, y, z, 1] \tag{2.4}$$

The basic transformations of rotate, translate, and scale can be kept in 4 × 4 transformation matrices. The 4 × 4 matrix is the smallest matrix that can represent all of the basic transformations. Because it is a square matrix, it has the potential for having a computable inverse, which is important for texture mapping and illumination calculations. In the case of rotation, translation, and nonzero scale transformations, the matrix always has a computable inverse. It can be multiplied with other transformation matrices to produce compound transformations while still maintaining 4 × 4-ness. The 4 × 4 identity matrix has zeros everywhere except along its diagonal; the diagonal elements all equal one (Eq. 2.5).

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.5}$$

Typically in the literature, a point is represented as a 4 × 1 column matrix (also known as a *column vector*) and is transformed by multiplying by a 4 × 4 matrix on the left (also known as *premultiplying* the column vector by the matrix), as shown in Equation 2.5 in the case of the identity matrix. However, some texts use a 1 × 4 matrix (also known as a *row vector*) to represent a point and transform it by multiplying it by a matrix on its right (the matrix *postmultiplies* the row vector). For example, postmultiplying a point by the identity transformation would appear as in Equation 2.6.

$$[x \quad y \quad z \quad 1] = [x \quad y \quad z \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Because the conventions are equivalent, it is immaterial which is used as long as consistency is maintained. The $4 \times 4$ transformation matrix used in one of the notations is the transpose of the $4 \times 4$ transformation matrix used in the other notation.

## 2.1.3 Concatenating transformations: multiplying transformation matrices

One of the main advantages of representing basic transformations as square matrices is that they can be multiplied together, which concatenates the transformations and produces a compound transformation. This enables a series of transformations, $M_i$, to be premultiplied so that a single compound transformation matrix, $M$, can be applied to a point $P$ (see Eq. 2.7). This is especially useful (i.e., computationally efficient) when applying the same series of transformations to a multitude of points. Note that matrix multiplication is associative ($(AB)C = A(BC)$) but not commutative ($AB \neq BA$).

$$\begin{aligned} P' &= M_1 \ M_2 \ M_3 \ M_4 \ M_5 \ M_6 \ P \\ M &= M_1 \ M_2 \ M_3 \ M_4 \ M_5 \ M_6 \\ P' &= MP \end{aligned} \tag{2.7}$$

When using the convention of postmultiplying a point represented by a row vector by the same series of transformations used when premultiplying a column vector, the matrices will appear in reverse order in addition to being the transposition of the matrices used in the premultiplication. Equation 2.8 shows the same computation as Equation 2.7, except in Equation 2.8, a row vector is postmultiplied by the transformation matrices. The matrices in Equation 2.8 are the same as those in Equation 2.7 but are now transposed and in reverse order. The transformed point is the same in both equations, with the exception that it appears as a column vector in Equation 2.7 and as a row vector in Equation 2.8. In the remainder of this book, such equations will be in the form shown in Equation 2.7.

$$\begin{aligned} P'^T &= P^T \ M_6^T \ M_5^T \ M_4^T \ M_3^T \ M_2^T \ M_1^T \\ M^T &= M_6^T \ M_5^T \ M_4^T \ M_3^T \ M_2^T \ M_1^T \\ P'^T &= P^T \ M^T \end{aligned} \tag{2.8}$$

## 2.1.4 Basic transformations

For now, only the basic transformations rotate, translate, and scale (uniform scale as well as nonuniform scale) will be considered. These transformations, and any combination of these, are *affine transformations* [4]. It should be noted that the transformation matrices are the same whether the space is left- or right-handed. The perspective transformation is discussed later. Restricting discussion to the basic transformations allows the fourth element of each point vector to be assigned the value one and the last row of the transformation matrix to be assigned the value [0 0 0 1] (Eq. 2.9).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.9}$$

The $x$, $y$, and $z$ translation values of the transformation are the first three values of the fourth column ($d$, $h$, and $m$ in Eq. 2.9). The upper left $3 \times 3$ submatrix represents rotation and scaling. Setting the upper left $3 \times 3$ submatrix to an identity transformation and specifying only translation produces Equation 2.10.

$$\begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.10}$$

A transformation consisting of only uniform scale is represented by the identity matrix with a scale factor, $S$, replacing the first three elements along the diagonal ($a$, $f$, and $k$ in Eq. 2.9). Nonuniform scale allows for independent scale factors to be applied to the $x$-, $y$-, and $z$-coordinates of a point and is formed by placing $S_x$, $S_y$, and $S_z$ along the diagonal as shown in Equation 2.11.

$$\begin{bmatrix} S_x x \\ S_y y \\ S_z z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.11}$$

Uniform scale can also be represented by setting the lowest rightmost value to $1/S$, as in Equation 2.12. In the homogeneous representation, the coordinates of the point represented are determined by dividing the first three elements of the vector by the fourth, thus scaling up the values by the scale factor $S$. This technique invalidates the assumption that the only time the lowest rightmost element is not one is during perspective and therefore should be used with care or avoided altogether.

$$\begin{bmatrix} S\,x \\ S\,y \\ S\,z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{1}{S} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{S} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.12}$$

Values to represent rotation are set in the upper left $3 \times 3$ submatrix ($a$, $b$, $c$, $e$, $f$, $g$, $i$, $j$, and $k$ of Eq. 2.9). Rotation matrices around the $x$-, $y$-, and $z$-axis are shown in Equations 2.13–2.15, respectively. In a right-handed coordinate system, a positive angle of rotation produces a counterclockwise rotation as viewed from the positive end of the axis looking toward the origin (the right-hand rule). In a left-handed (right-handed) coordinate system, a positive angle of rotation produces a clockwise (counterclockwise) rotation as viewed from the positive end of an axis. This can be remembered by noting that when pointing the thumb of the left (right) hand in the direction of the positive axis, the fingers wrap clockwise (counterclockwise) around the closed hand when viewed from the end of the thumb.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.13}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.14}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.15}$$

Combinations of rotations and translations are usually referred to as *rigid transformations* because distance is preserved and the spatial extent of the object does not change; only its position and orientation in space are changed. *Similarity transformations* also allow uniform scale in addition to rotation and translation. These transformations preserve the object's intrinsic properties[2] (e.g., dihedral angles[3]) and relative distances but not absolute distances. Nonuniform scale, however, is usually not considered a similarity transformation because object properties such as dihedral angles are changed. A *shear* transformation is a combination of rotation and nonuniform scale and creates columns (rows) that might not be orthogonal to each other. Any combination of rotations, translations, and (uniform or nonuniform) scales still retains the last row of three zeros followed by a one. Notice that any affine transformation can be represented by a multiplicative $3 \times 3$ matrix (representing rotations, scales, and shears) followed by an additive three-element vector (translation).

## 2.1.5 Representing an arbitrary orientation

Rigid transformations (consisting of only rotations and translations) are very useful for moving objects around a scene without disturbing their geometry. These rigid transformations can be represented by a (possibly compound) rotation followed by a translation. The rotation transformation represents the object's orientation relative to its definition in object space. This section considers a particular way to represent an object's orientation.

### Fixed-angle representation

One way to represent an orientation is as a series of rotations around the principal axes (the *fixed-angle representation*). When illustrating the relationship between orientation and a fixed order of rotations around the principal axes, consider the problem of determining the transformations that would produce a given geometric configuration. For example, consider that an aircraft is originally defined at the origin of a right-handed coordinate system with its nose pointed down the $z$-axis and its up vector in the positive $y$-axis direction (i.e., its object space representation). Now, imagine that the objective is to position the aircraft in world space so that its center is at $(20, -10, 35)$, its nose is oriented toward the point $(23, -14, 40)$, and its up vector is pointed in the general direction of the $y$-axis (or, mathematically, so that its up vector lies in the plane defined by the aircraft's center, the point the plane is oriented toward, and the global $y$-axis) (see Figure 2.6).

[2]An object's intrinsic properties are those that are measured irrespective of an external coordinate system.
[3]The dihedral angle is the interior angle between adjacent polygons measured at the common edge.

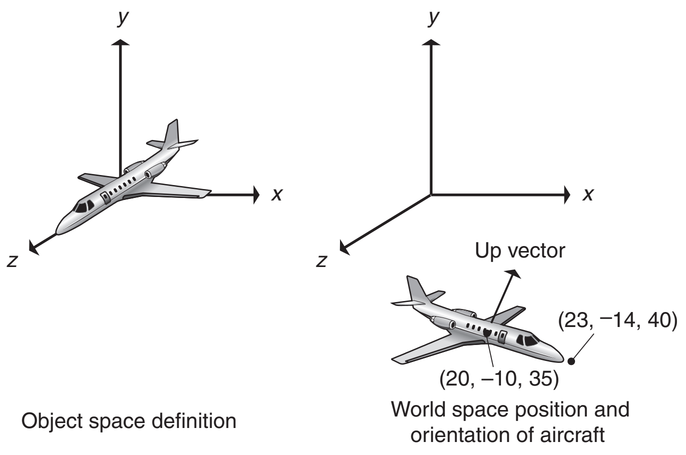


**FIGURE 2.6**

Desired position and orientation.

The task is to determine the series of transformations that takes the aircraft from its original object space definition to its desired position and orientation in world space. This series of transformations will be one or more rotations about the principal axes followed by a translation of $(20, -10, 35)$. The rotations will transform the aircraft to an orientation so that, with its center at the origin, its nose is oriented toward $(23-20, -14 + 10, 40-35) = (3, -4, 5)$; this will be referred to as the aircraft's desired orientation vector.

In general, any such orientation can be effected by a rotation about the $z$-axis to tilt the object, followed by a rotation about the $x$-axis to tip the nose up or down, followed by a rotation about the $y$-axis to swing the plane around to point to the correct direction. This sequence is not unique; others could be constructed as well.

In this particular example, there is no tilt necessary because the desired up vector is already in the plane of the $y$-axis and orientation vector. We need to determine the $x$-axis rotation that will dip the nose down the right amount and the $y$-axis rotation that will swing it around the right amount. We do this by looking at the transformations needed to take the plane's initial orientation in object space aligned with the $z$-axis to its desired orientation.

The transformation that takes the aircraft to its desired orientation can be formed by determining the sines and cosines necessary for the $x$-axis and $y$-axis rotation matrices. Note that the length of the orientation vector is $\sqrt{3^2 + (-4)^2 + 5^2} = \sqrt{50}$. In first considering the $x$-axis rotation, initially position the orientation vector along the $z$-axis so that its endpoint is at $(0, 0, \sqrt{50})$. The $x$-axis rotation must rotate the endpoint of the orientation vector so that it is $-4$ in $y$. By the Pythagorean Rule, the $z$-coordinate of the endpoint would then be $\sqrt{50 - 4^2} = \sqrt{34}$ after the rotation. The sines and cosines can be read from the triangle formed by the rotated orientation vector, the vertical line segment that extends from the end of the orientation vector up to intersect the $x$-$z$ plane, and the line segment from that intersection point to the origin (Figure 2.7). Observing that a positive $x$-axis rotation will

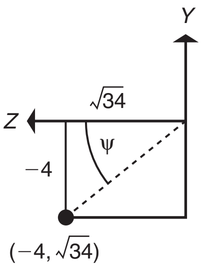


**FIGURE 2.7**

Projection of desired orientation vector onto *y-z* plane.

rotate the orientation vector down in $y$, we have $\sin\psi = 4/\sqrt{50}$ and $\cos\psi = \sqrt{34}/\sqrt{50}$. The $x$-axis rotation matrix looks like this:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \dfrac{\sqrt{34}}{\sqrt{50}} & \dfrac{-4}{\sqrt{50}} \\ 0 & \dfrac{4}{\sqrt{50}} & \dfrac{\sqrt{34}}{\sqrt{50}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \dfrac{\sqrt{17}}{5} & \dfrac{-2\sqrt{2}}{5} \\ 0 & \dfrac{2\sqrt{2}}{5} & \dfrac{\sqrt{17}}{5} \end{bmatrix} \tag{2.16}$$

After the pitch rotation has been applied, a $y$-axis rotation is required to spin the aircraft around (yaw) to its desired orientation. The sine and cosine of the $y$-axis rotation can be determined by looking at the projection of the desired orientation vector in the $x$-$z$ plane. This projection is (3, 0, 5). Thus, a positive $y$-axis rotation with $\sin\varphi = 3/\sqrt{34}$ and $\cos\varphi = 5/\sqrt{34}$ is required (Figure 2.8). The $y$-axis rotation matrix looks like this:

$$R_y = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{bmatrix} = \begin{bmatrix} \dfrac{5}{\sqrt{34}} & 0 & \dfrac{3}{\sqrt{34}} \\ 0 & 1 & 0 \\ \dfrac{-3}{\sqrt{34}} & 0 & \dfrac{5}{\sqrt{34}} \end{bmatrix} \tag{2.17}$$

The final transformation of a point $P$ would be $P' = R_y R_x P$.

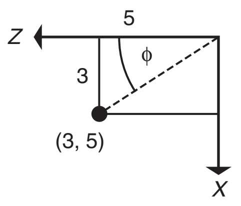


**FIGURE 2.8**

Projection of desired orientation vector onto *x-z* plane.

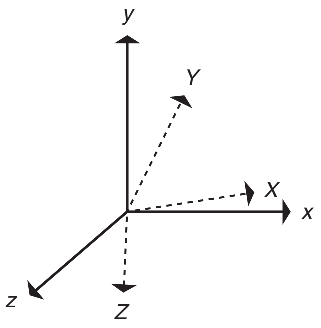


**FIGURE 2.9**

Global coordinate system and unit coordinate system to be transformed.

An alternative way to represent a transformation to a desired orientation is to construct what is known as the *matrix of direction cosines*. Consider transforming a copy of the global coordinate system so that it coincides with a desired orientation defined by a unit coordinate system (see Figure 2.9). To construct this matrix, note that the transformation matrix, $M$, should do the following: map a unit $x$-axis vector into the $X$-axis of the desired orientation, map a unit $y$-axis vector into the $Y$-axis of the desired orientation, and map a unit $z$-axis vector into the $Z$-axis of the desired orientation (see Eq. 2.18). These three mappings can be assembled into one matrix expression that defines the matrix $M$ (Eq. 2.19).

$$\begin{array}{ccc} X = Mx & Y = My & Z = Mz \\ \begin{bmatrix} X_x \\ X_y \\ X_z \end{bmatrix} = M\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & \begin{bmatrix} Y_x \\ Y_y \\ Y_z \end{bmatrix} = M\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} & \begin{bmatrix} Z_x \\ Z_y \\ Z_z \end{bmatrix} = M\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{array} \tag{2.18}$$

$$\begin{bmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{bmatrix} = M\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = M \tag{2.19}$$

Since a unit $x$-vector ($y$-vector, $z$-vector) multiplied by a transformation matrix will replicate the values in the first (second, third) column of the transformation matrix, the columns of the transformation matrix can be filled with the coordinates of the desired transformed coordinate system. Thus, the first column of the transformation matrix becomes the desired $X$-axis as described by its $x$-, $y$-, and $z$-coordinates in the global space, call it $u$; the second column becomes the desired $Y$-axis, call it $v$; and the third column becomes the desired $Z$-axis, call it $w$ (Eq. 2.20). The name *matrix of direction cosines* is derived from the fact that the coordinates of a desired axis in terms of the global coordinate system are the cosines of the angles made by the desired axis with each of the global axes.

In the example of transforming the aircraft, the desired $Z$-axis is the desired orientation vector, $w = [3, -4, 5]$. With the assumption that there is no longitudinal rotation (roll), the desired $X$-axis can be formed by taking the cross-product of the original $y$-axis and the desired $Z$-axis, $u = [5, 0, -3]$. The desired $Y$-axis can then be formed by taking the cross-product of the desired $Z$-axis and the desired $X$-axis, $v = [12, 34, 20]$. Each of these is normalized by dividing by its length to form unit vectors. This results in the matrix of Equation 2.20, which is the same matrix formed by multiplying the $x$-rotation matrix, $R_x$ of Equation 2.16, by the $y$-rotation matrix, $R_y$ of Equation 2.17.

$$M = \begin{bmatrix} u^T & v^T & w^T \end{bmatrix} = \begin{bmatrix} \frac{5}{\sqrt{34}} & \frac{6}{5\sqrt{17}} & \frac{3}{5\sqrt{2}} \\ 0 & \frac{\sqrt{17}}{5} & -2\frac{\sqrt{2}}{5} \\ -\frac{3}{\sqrt{34}} & \frac{2}{\sqrt{17}} & \frac{1}{\sqrt{2}} \end{bmatrix} = R_y R_x \tag{2.20}$$

### 2.1.6 Extracting transformations from a matrix

For a compound transformation matrix that represents a series of rotations and translations, a set of individual transformations can be extracted from the matrix, which, when multiplied together, produce the original compound transformation matrix. Notice that the series of transformations to produce a compound transformation is not unique, so there is no guarantee that the series of transformations so extracted will be exactly the ones that produced the compound transformation (unless something is known about the process that produced the compound matrix).

An arbitrary rigid transformation can easily be formed by up to three rotations about the principal axes (or one compound rotation represented by the direction cosine matrix) followed by a translation.

The last row of a $4 \times 4$ transformation matrix, if the matrix does not include a perspective transformation, will have zero in the first three entries and one as the fourth entry (ignoring the use of that element to represent uniform scale). As shown in Equation 2.21, the first three elements of the last column of the matrix, $A_{14}$, $A_{24}$, and $A_{34}$, represent a translation. The upper left $3 \times 3$ submatrix of the original $4 \times 4$ matrix can be viewed as the definition of the transformed unit coordinate system. It can be decomposed into three rotations around principal axes by arbitrarily choosing an ordered sequence of three axes (such as $x$ followed by $y$ followed by $z$). By using the projection of the transformed unit coordinate system to determine the sines and cosines, the appropriate rotation matrices can be formed in much the same way that transformations were determined in Section 2.1.5.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.21}$$

If the compound transformation matrix includes a uniform scale factor, the rows of the $3 \times 3$ submatrix will form orthogonal vectors of uniform length. The length will be the scale factor, which, when followed by the rotations and translations, forms the decomposition of the compound transformation. If the rows of the $3 \times 3$ submatrix form orthogonal vectors of unequal length, then their lengths represent nonuniform scale factors that precede any rotations.

### 2.1.7 Description of transformations in the display pipeline

Now that the basic transformations have been discussed in some detail, the previously described transformations of the display pipeline can be explained in terms of concatenating these basic transformations. It should be noted that the descriptions of eye space and the corresponding perspective transformation are not unique. They vary among the introductory graphics texts depending on where the observer is placed along the $z$-axis to define eye space, whether the eye space coordinate system is left- or right-handed, exactly what information is required from the user in describing the perspective transformation, and the range of visible $z$-values in image space. While functionally equivalent, the various approaches produce transformation matrices that differ in the values of the individual elements.

#### *Object space to world space transformation*

In a simple implementation, the transformation of an object from its object space into world space is a series of rotations, translations, and scales (i.e., an affine transformation) that are specified by the user (by explicit numeric input or by some interactive technique) to place a transformed copy of the object data into a world space data structure. In some systems, the user is required to specify this transformation in terms of a predefined order of basic transformations such as scale, rotation around the $x$-axis, rotation around the $y$-axis, rotation around the $z$-axis, and translation. In other systems, the user may be able to specify an arbitrarily ordered sequence of basic transformations. In either case, the series of transformations can be compounded into a single object space to world space transformation matrix.

The object space to world space transformation is usually the transformation that is modified over time to produce motion. In more complex animation systems, this transformation may include manipulations of arbitrary complexity not suitable for representation in a matrix, such as nonlinear shape deformations.

#### *World space to eye space transformation*

In preparation for the perspective transformation, a rigid transformation is performed on all of the object data in world space. The transformation is designed so that, in eye space, the observer is positioned at the origin, the view vector aligns with the positive $z$-axis in left-handed space, and the up vector aligns with the positive $y$-axis. The transformation is formed as a series of basic transformations.

First, the data is translated so that the observer is moved to the origin. Then, the observer's coordinate system (view vector, up vector, and the third vector required to complete a left-handed coordinate system) is transformed by up to three rotations so as to align the view vector with the global negative $z$-axis and the up vector with the global $y$-axis. Finally, the $z$-axis is flipped by negating the $z$-coordinate. All of the individual transformations can be represented by $4 \times 4$ transformation matrices, which are multiplied together to produce a single compound world space to eye space transformation matrix. This transformation prepares the data for the perspective transformation by putting it in a form in which the perspective divide is simply dividing by the point's $z$-coordinate.

#### *Perspective matrix multiply*

The perspective matrix multiplication is the first part of the perspective transformation. The fundamental computation performed by the perspective transformation is that of dividing the $x$- and $y$-coordinates by their $z$-coordinate and normalizing the visible range in $x$ and $y$ to $[-1, +1]$. This is accomplished by using a homogeneous representation of a point and, as a result of the perspective matrix multiplication, producing a representation in which the fourth element is $Z_e \tan \varphi$. $Z_e$ is the point's $z$-coordinate in eye space and φ is the half angle of view in the vertical or horizontal direction (assuming a square viewport for now and, therefore, the vertical and horizontal angles are equal). The $z$-coordinate is transformed so that planarity is preserved and the visible range in $z$ is mapped into $[0, +1]$. (These ranges are arbitrary and can be set to anything by appropriately forming the perspective matrix. For example, sometimes the visible range in $z$ is set to $[-1, +1]$.) In addition, the aspect ratio of the viewport can be used in the matrix to modify the horizontal or vertical half angle of view so that no distortion results in the viewed data.

### *Perspective divide*

Each point produced by the perspective matrix multiplication has a nonunitary fourth component that represents the perspective divide by $z$. Dividing each point by its fourth component completes the perspective transformation. This is considered a separate step from the perspective matrix multiply because a commonly used clipping procedure operates on the homogeneous representation of points produced by the perspective matrix multiplication but before perspective divide.

Clipping, the process of removing data that are outside the view frustum, can be implemented in a variety of ways. It is computationally simpler if clipping is performed after the world space to eye space transformation. It is important to perform clipping in $z$ using the near clipping distance before perspective divide to prevent divide by zero and to avoid projecting objects behind the observer onto the picture plane. However, the details of clipping are not relevant to the discussion here. Interested readers should refer to one of the standard computer graphics texts (e.g., [2]) for the details of clipping procedures.

### *Image to screen space mapping*

The result of the perspective transformation (the perspective matrix multiply followed by perspective divide) maps visible elements into the range of minus one to plus one ($[-1, +1]$) in $x$ and $y$. This range is now mapped into the user-specified viewing area of the screen-based pixel coordinate system. This is a simple linear transformation represented by a scale and a translation and thus can be easily represented in a $4 \times 4$ transformation matrix.

## 2.1.8 Error considerations

### *Accumulated round-off error*

Once the object space to world space transformation matrix has been formed for an object, the object is transformed into world space by simply multiplying all of the object's object space points by the transformation matrix. When an object's position and orientation are animated, its points will be repeatedly transformed over time—as a function of time. One way to do this is to repeatedly modify the object's world space points. However, incremental transformation of world space points can lead to the accumulation of round-off errors. For this reason, it is almost always better to modify the transformation from object to world space and reapply the transformation to the object space points rather than to repeatedly transform the world space coordinates. To further transform an object that already has a transformation matrix associated with it, one simply has to form a transformation matrix and premultiply it by the existing transformation matrix to produce a new one. However, round-off errors can also accumulate when one repeatedly modifies a transformation matrix. The best way is to build the transformation matrix anew each time it is to be applied.

An affine transformation matrix can be viewed as a 3 × 3 rotation/scale submatrix followed by a translation. Most of the error accumulation occurs because of the operations resulting from multiplying the *x*-, *y*-, and *z*-coordinates of the point by the 3 × 3 submatrix. Therefore, the following round-off error example will focus on the errors that accumulate as a result of rotations.

Consider the case of the moon orbiting the earth. For the sake of simplicity, the assumption is that the center of the earth is at the origin and, initially, the moon data are defined with the moon's center at the origin. The moon data are first transformed to an initial position relative to the earth, for example (*r*, 0, 0) (see Figure 2.10). There are three approaches that could be taken to animate the rotation of the moon around the earth, and these will be used to illustrate various effects of round-off error.

The first approach is, for each frame of the animation, to apply a delta *z*-axis transformation matrix to the moon's points, in which each delta represents the angle it moves in one frame time (see Figure 2.11). Round-off errors will accumulate in the world space object points. Points that began as coplanar will no longer be coplanar. This can have undesirable effects, especially in display algorithms that linearly interpolate values to render a surface.

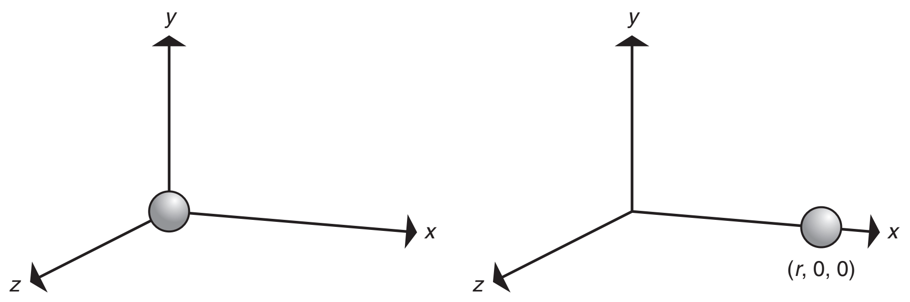
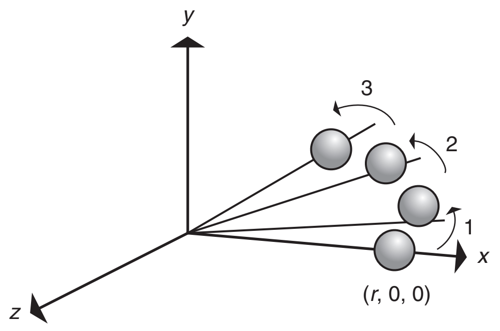


**FIGURE 2.10**

Translation of moon to its initial position on the *x*-axis.



```
for each point P of the moon {
  P′ = P
}
R_dz = y-axis rotation of 5 degrees
repeat until (done) {
  for each point P′ of the moon {
    P′ = R_dz*P′
  }
  record a frame of the animation
}
```

**FIGURE 2.11**

Rotation by applying incremental rotation matrices to points.

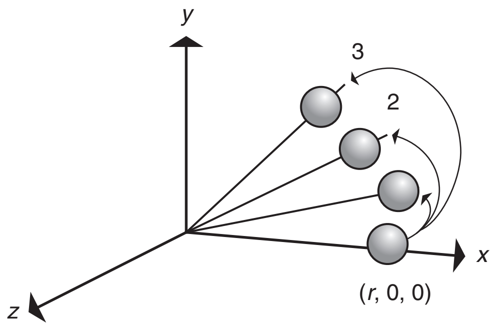
The second approach is, for each frame, to incrementally modify the transformation matrix that takes the object space points into the world space positions. In the example of the moon, the transformation matrix is initialized with the *x*-axis translation matrix. For each frame, a delta *z*-axis transformation matrix multiplies the current transformation matrix and then that resultant matrix is applied to the moon's object space points (see Figure 2.12). Round-off error will accumulate in the transformation matrix. Over time, the matrix will deviate from representing a rigid transformation. Shearing effects will begin to creep into the transformation and angles will cease to be preserved. While a square may begin to look like something other than a square, coplanarity will be preserved (because any matrix multiplication is, by definition, a linear transformation that preserves planarity), so that rendering results will not be compromised.

The third approach is to add the delta value to an accumulating angle variable and then build the *z*-axis rotation matrix from that angle parameter. This would then be multiplied with the *x*-axis translation matrix, and the resultant matrix would be applied to the original moon points in object space (see Figure 2.13). In this case, any round-off error will accumulate in the angle variable so that, over time, it may begin to deviate from what is desired. This may have unwanted effects when one tries to coordinate motions, but the transformation matrix, which is built anew every frame, will not accumulate any errors itself. The transformation will always represent a valid rigid transformation with both planarity and angles being preserved.

## *Orthonormalization*

The rows of a matrix that represent a rigid transformation are perpendicular to each other and are of unit length (orthonormal). The same can be said of the matrix columns. If values in a rigid transformation matrix have accumulated errors, then the rows cease to be orthonormal and the matrix ceases to represent a rigid transformation; it will have the effect of introducing shear into the transformation. However, if it is known that the matrix is supposed to represent a rigid transformation, it can be massaged back into a rigid transformation matrix.

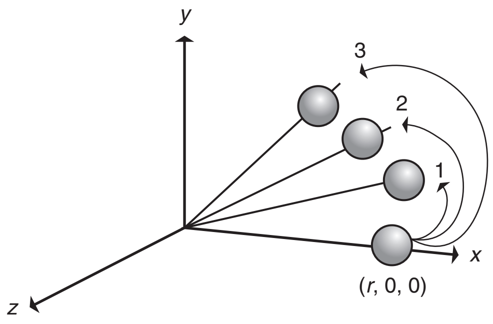
A rigid transformation matrix has an upper $3 \times 3$ submatrix with specific properties: the rows (columns) are unit vectors orthogonal to each other. A simple procedure to reformulate the transformation



```
R = identity matrix
R_dz = z-axis rotation of 5 degrees
repeat until (done) {
  for each point P of the moon {
    P' = R*P
  }
  record a frame of the animation
  R = R*R_dz
}
```

**FIGURE 2.12**

Rotation by incrementally updating the rotation matrix.



```
z = 0
repeat until (done) {
 R = z-axis rotation matrix of ‘z’ degrees
 for each point P of the moon {
  P′ = R*P
 }
 record a frame of the animation
 z = z+5
}
```

**FIGURE 2.13**

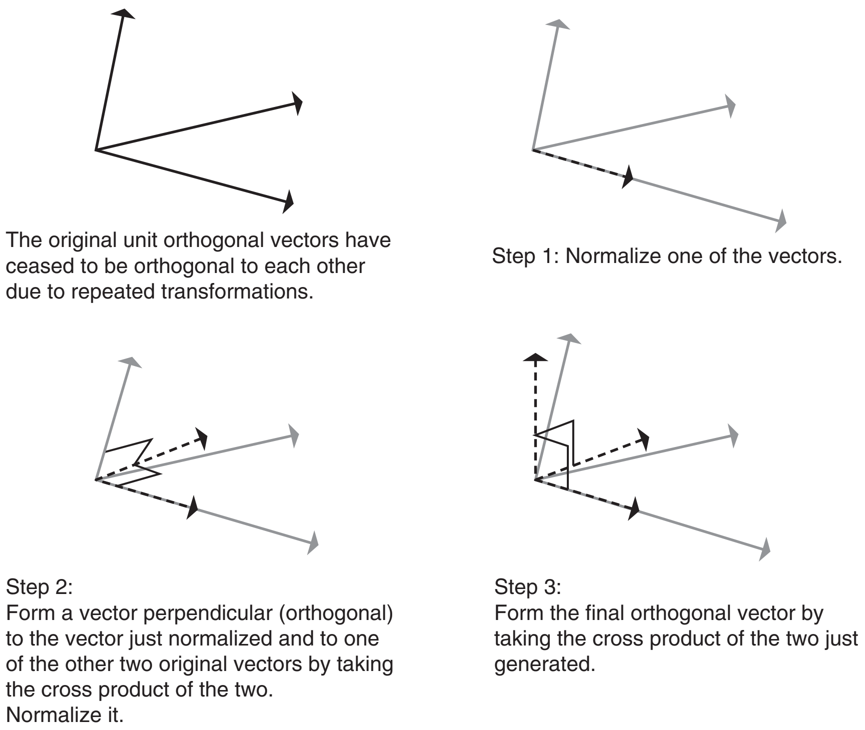
Rotation by forming the rotation matrix anew for each frame.

matrix to represent a rigid transformation is to take the first row (column) and normalize it. Take the second row (column), compute the cross-product of this row (column) and the first row (column), normalize it, and place it in the third row (column). Take the cross-product of the third row (column) and the first row (column) and put it in the second row (column) (see Figure 2.14). Notice that this does not necessarily produce the correct transformation; it merely forces the matrix to represent a rigid transformation. The error has just been shifted around so that the columns of the matrix are orthonormal and the error may be less noticeable.

If the transformation might contain a uniform scale, then take the length of one of the rows, or the average length of the three rows, and, instead of normalizing the vectors by the steps described above, make them equal to this length. If the transformation might include nonuniform scale, then the difference between shear and error accumulation cannot be determined unless something more is known about the transformations represented. For example, if it is known that nonuniform scale was applied before any rotation (i.e., no shear), then Gram-Schmidt Orthonormalization [6] can be performed, without the normalization step, to force orthogonality among the vectors. Gram-Schmidt processes the vectors in any order. To process a vector, project it onto each previously processed vector. Subtract the projections from the vector currently being processed, then process the next vector. When all vectors have been processed, they are orthogonal to each other.

### *Considerations of scale*

When constructing a large database (e.g., flight simulator), there may be such variations in the magnitude of various measures as to create precision problems. For example, you may require detail on the order of a fraction of an inch may be required for some objects, while the entire database may span thousands of miles. The scale of values would range from to $10^{-1}$ inches to 5000 miles$\cdot$5280 feet/mile$\cdot$12 inches/foot $= 3.168 \cdot 10^8$ inches. This exceeds the precision of 32-bit single-precision representations. Using double-precision will help eliminate the problem. However, using double-precision representations may also increase storage space requirements and decrease the speed of the computations. Alternatively, subdividing the database into local data, such as airports in the example of a flight simulator, and switching between these localized databases might provide an acceptable solution.



The original unit orthogonal vectors have ceased to be orthogonal to each other due to repeated transformations.

Step 1: Normalize one of the vectors.

Step 2:
Form a vector perpendicular (orthogonal) to the vector just normalized and to one of the other two original vectors by taking the cross product of the two.
Normalize it.

Step 3:
Form the final orthogonal vector by taking the cross product of the two just generated.

**FIGURE 2.14**

Orthonormalization of a set of three vectors.

## 2.2 Orientation representation

A common issue that arises in computer animation is deciding the best way to represent the position and orientation of an object in space and how to interpolate the represented transformations over time to produce motion. A typical scenario is one in which the user specifies an object in two transformed states and the computer is used to interpolate intermediate states, thus producing animated key-frame motion. Another scenario is when an object is to undergo two or more successive transformations and it would be efficient to concatenate these transformations into a single representation before applying it to a multitude of object vertices. This section discusses possible orientation representations and identifies strengths and weaknesses; the next chapter addresses the best way to interpolate orientations using these representations. In this discussion, it is assumed that the final transformation applied to the object is a result of rotations and translations only, so that there is no scaling involved, nonuniform or otherwise; that is, the transformations considered are *rigid body*.

The first obvious choice for representing the orientation and position of an object is by a $4 \times 4$ transformation matrix. For example, a user may specify a series of rotations and translations to apply to an object. This series of transformations is compiled into 4 × 4 matrices and is multiplied together to produce a compound 4 × 4 transformation matrix. In such a matrix, the upper left 3 × 3 submatrix represents a rotation to apply to the object, while the first three elements of the fourth column represent the translation (assuming points are represented by column vectors that are premultiplied by the transformation matrix). No matter how the 4 × 4 transformation matrix was formed (no matter in what order the transformations were given by the user, such as "rotate about $x$, translate, rotate about $x$, rotate about $y$, translate, rotate about $y$"), the final 4 × 4 transformation matrix produced by multiplying all of the individual transformation matrices in the specified order will result in a matrix that specifies the final position of the object by a 3 × 3 rotation matrix followed by a translation. The conclusion is that the rotation can be interpolated independently from the translation. (For now, consider that the interpolations are linear, although higher order interpolations are possible; see Appendix B.5.)

Consider two such transformations that the user has specified as key states with the intention of generating intermediate transformations by interpolation. While it should be obvious that interpolating the translations is straightforward, it is not at all clear how to go about interpolating the rotations. In fact, it is the objective of this discussion to show that interpolation of orientations is not nearly as straightforward as interpolation of translation. A property of 3 × 3 rotation matrices is that the rows and columns are orthonormal (unit length and perpendicular to each other). Simple linear interpolation between the nine pairs of numbers that make up the two 3 × 3 rotation matrices to be interpolated will not produce intermediate 3 × 3 matrices that are orthonormal and are therefore not rigid body rotations. It should be easy to see that interpolating from a rotation of +90 degrees about the $y$-axis to a rotation of −90 degrees about the $y$-axis results in intermediate transformations that are nonsense (Figure 2.15).

So, direct interpolation of transformation matrices is not acceptable. There are alternative representations that are more useful than transformation matrices in performing such interpolations including fixed angle, Euler angle, axis–angle, quaternions, and exponential maps.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Positive 90-degree $y$-axis rotation     Negative 90-degree $y$-axis rotation

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Interpolated matrix halfway between the orientation representations above

**FIGURE 2.15**

Direct interpolation of transformation matrix values can result in nonsense—transformations.

### 2.2.1 Fixed-angle representation

A *fixed-angle* representation[4] really refers to "angles used to rotate about fixed axes." A fixed order of three rotations is implied, such as *x*-*y*-*z*. This means that orientation is given by a set of three ordered parameters that represent three ordered rotations about fixed axes: first around *x*, then around *y*, and then around *z*. There are many possible orderings of the rotations, and, in fact, it is not necessary to use all three coordinate axes. For example, *x*-*y*-*x* is a feasible set of rotations. The only orderings that do not make sense are those in which an axis immediately follows itself, such as in *x*-*x*-*y*. In any case, the main point is that the orientation of an object is given by three angles, such as (10, 45, 90). In this example, the orientation represented is obtained by rotating the object first about the *x*-axis by 10 degrees, then about the *y*-axis by 45 degrees, and then about the *z*-axis by 90 degrees. In Figure 2.16, the aircraft is shown in its initial orientation and in the orientation represented by the values of (10, 45, 90).

The following notation will be used to represent such a sequence of rotations: $R_z(90)R_y(45)R_x(10)$ (in this text, transformations are implemented by premultiplying column vectors by transformation matrices; thus, the rotation matrices appear in right to left order).

From this orientation, changing the *x*-axis rotation value, which is applied first to the data points, will make the aircraft's nose dip more or less in the *y*-*z* plane. Changing the *y*-axis rotation will change the amount the aircraft, which has been rotated around the *x*-axis, rotates out of the *y*-*z* plane. Changing the *z*-axis rotation value, the rotation applied last, will change how much the twice-rotated aircraft will rotate about the *z*-axis.

The problem with using this scheme is that two of the axes of rotation can effectively line up on top of each other when an object can rotate freely in space (or around a 3 degree of freedom[5] joint). Consider an object in an orientation represented by (0, 90, 0), as shown in Figure 2.17. Examine the effect a slight change in the first and third parametric values has on the object in that orientation. A slight change of the third parameter will rotate the object slightly about the global *z*-axis because that is the rotation applied last to the data points. However, note that the effect of a slight change of the first

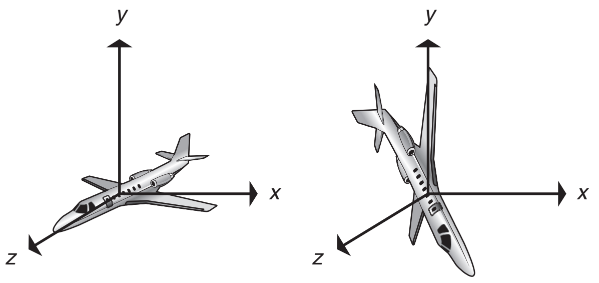


**FIGURE 2.16**

Fixed-angle representation.

[4]Terms referring to rotational representations are not used consistently in the literature. This book follows the usage found in *Robotics* [1], where fixed angle refers to rotation about the fixed (global) axes and Euler angle refers to rotation about the rotating (local) axes.

[5]The degrees of freedom that an object possesses is the number of independent variables that have to be specified to completely locate that object (and all of its parts).

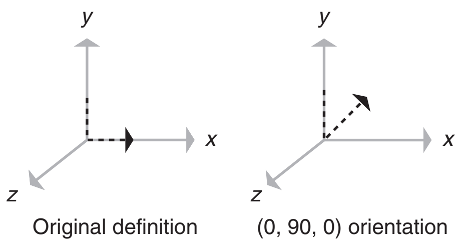


**FIGURE 2.17**

Fixed-angle representation of (0, 90, 0).

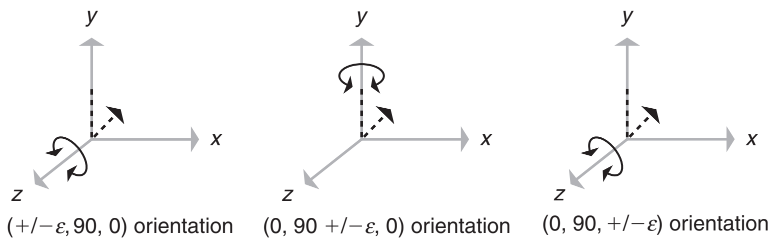

y
x
z
(+/−ε, 90, 0) orientation
y
x
z
(0, 90 +/−ε, 0) orientation
y
x
z
(0, 90, +/−ε) orientation

**FIGURE 2.18**

Effect of slightly altering values of fixed-angle representation (0, 90, 0).

parameter, which rotates the original data points around the *x*-axis, will also have the effect of rotating the transformed object slightly about the *z*-axis (Figure 2.18). This results because the 90-degree *y*-axis rotation has essentially made the first axis of rotation align with the third axis of rotation. The effect is called *gimbal lock*. From the orientation (0, 90, 0), the object can no longer be rotated about the global *x*-axis by a small change in its orientation representation. Actually, the representation that will perform an incremental rotation about the *x*-axis from the (0, 90, 0) orientation is (90, 90 + ε, 90), which is not very intuitive.

The cause of this problem can often make interpolation between key positions problematic. Consider the key orientations (0, 90, 0) and (90, 45, 90), as shown in Figure 2.19. The second orientation is a

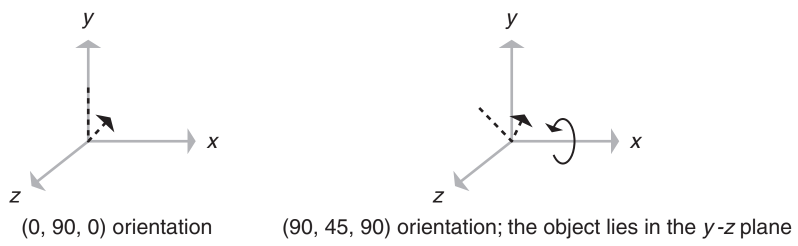


**FIGURE 2.19**

Example orientations to interpolate.

45-degree $x$-axis rotation from the first position. However, as discussed above, the object can no longer directly rotate about the $x$-axis from the first key orientation because of the 90-degree $y$-axis rotation. Direct interpolation of the key orientation representations would produce (45, 67.5, 45) as the halfway orientation, which is very different from the (90, 22.5, 90) orientation that is desired (because that is the representation of the orientation that is intuitively halfway between the two given orientations). The result is that the object will swing out of the $y$-$z$ plane during the interpolation, which is not the behavior one would expect.

In its favor, the fixed-angle representation is compact, fairly intuitive, and easy to work with because the implied operations correspond to what we know how to do mathematically—rotate data around the global axes. However, it is often not the most desirable representation to use because of the gimbal lock problem.

## 2.2.2 Euler angle representation

In a *Euler angle* representation, the axes of rotation are the axes of the local coordinate system that rotate with the object, as opposed to the fixed global axes. A typical example of using Euler angles is found in the roll, pitch, and yaw of an aircraft (Figure 2.20).

As with the fixed-angle representation, the Euler angle representation can use any of various orderings of three axes of rotation as its representation scheme. Consider a Euler angle representation that uses an $x$-$y$-$z$ ordering and is specified as $(\alpha, \beta, \gamma)$. The $x$-axis rotation, represented by the transformation matrix $R_x(\alpha)$, is followed by the $y$-axis rotation, represented by the transformation matrix $R_y(\beta)$, around the $y$-axis of the local, rotated coordinate system. Using a prime symbol to represent rotation about a rotated frame and remembering that points are represented as column vectors and are premultiplied by transformation matrices, one achieves a result of $R_y{}'(\beta)R_x(\alpha)$. Using global axis rotation matrices to implement the transformations, the $y$-axis rotation around the rotated frame can be effected by $R_x(\alpha)R_y(\beta)R_x(-\alpha)$. Thus, the result after the first two rotations is shown in Equation 2.22.

$$R_y{}'(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_x(-\alpha)R_x(\alpha) = R_x(\alpha)R_y(\beta) \tag{2.22}$$

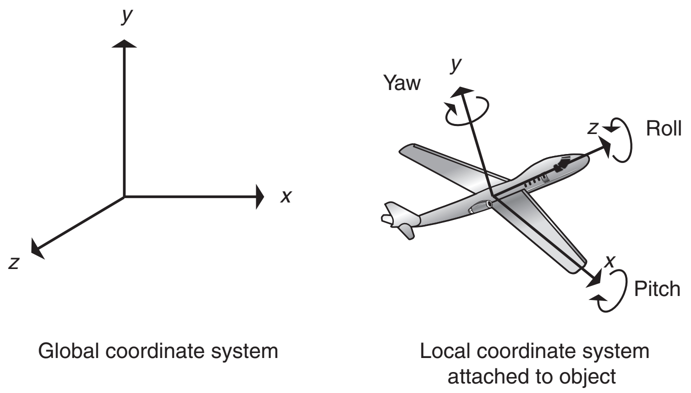


**FIGURE 2.20**

Euler angle representation.

The third rotation, $R_z(\gamma)$, is around the now twice-rotated frame. This rotation can be effected by undoing the previous rotations with $R_x(-\alpha)$ followed by $R_y(-\beta)$, then rotating around the global $z$-axis by $R_z(\gamma)$, and then reapplying the previous rotations. Putting all three rotations together, and using a double prime to denote rotation about a twice-rotated frame, results in Equation 2.23.

$$\begin{aligned} R_z''(\gamma)R_y'(\beta)R_x(\alpha) &= R_x(\alpha)R_y(\beta)R_z(\gamma)R_y(-\beta)R_x(-\alpha)R_x(\alpha)R_y(\beta) \\ &= R_x(\alpha)R_y(\beta)R_z(\gamma) \end{aligned} \tag{2.23}$$

Thus, this system of Euler angles is precisely equivalent to the fixed-angle system in reverse order. This is true for any system of Euler angles. For example, $z$-$y$-$x$ Euler angles are equivalent to $x$-$y$-$z$ fixed angles. Therefore, the Euler angle representation has exactly the same advantages and disadvantages (i.e., gimbal lock) as those of the fixed-angle representation.

### 2.2.3 Angle and axis representation

In the mid-1700s, Leonhard Euler showed that one orientation can be derived from another by a single rotation about an axis. This is known as the Euler Rotation Theorem [1]. Thus, any orientation can be represented by three numbers: two for the axis, such as longitude and latitude, and one for the angle (Figure 2.21). The axis can also be represented (somewhat inefficiently) by a three-dimensional vector. This can be a useful representation. Interpolation between representations $(A_1, \theta_1)$ and $(A_2, \theta_2)$, where A is the axis of rotation and $\theta$ is the angle, can be implemented by interpolating the axes of rotation and the angles separately (Figure 2.22). An intermediate axis can be determined by rotating one axis part-way toward the other. The axis for this rotation is formed by taking the cross product of two axes, $A_1$ and $A_2$. The angle between the two axes is determined by taking the inverse cosine of the dot product of normalized versions of the axes. An interpolant, $k$, can then be used to form an intermediate axis and angle pair. Note that the axis–angle representation does not lend itself to easily concatenating a series of rotations. However, the information contained in this representation can be put in a form in which these operations are easily implemented: quaternions.

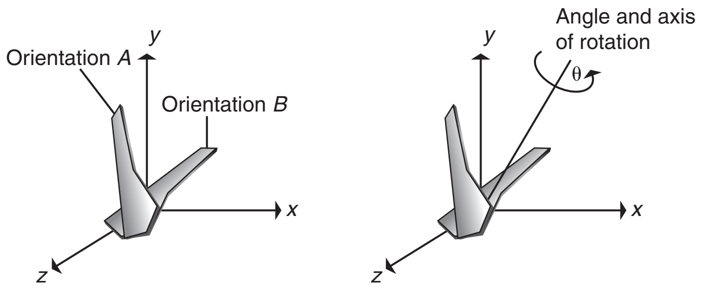
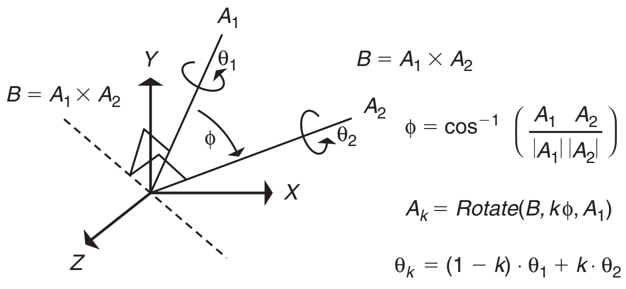


**FIGURE 2.21**

Euler's rotation theorem implies that for any two orientations of an object, one can be produced from the other by a single rotation about an arbitrary axis.



$A_1$
$Y$
$\theta_1$
$B = A_1 \times A_2$
$B = A_1 \times A_2$
$A_2$
$\phi$
$\theta_2$
$X$
$Z$
$\phi = \cos^{-1}\left(\frac{A_1 \quad A_2}{|A_1|\,|A_2|}\right)$
$A_k = Rotate(B, k\phi, A_1)$
$\theta_k = (1 - k)\cdot\theta_1 + k\cdot\theta_2$

**FIGURE 2.22**

Interpolating axis-angle representations of ($A_1$, $\theta_1$) and ($A_2$, $\theta_2$) by $k$ to get ($A_k$, $\theta_k$), where 'Rotate(a,b,c)' rotates 'c' around 'a' by 'b' degrees.

## 2.2.4 Quaternion representation

As discussed earlier, the representations covered so far have drawbacks when interpolating intermediate orientations when an object or joint has three degrees of rotational freedom. A better approach is to use *quaternions* to represent orientation [5]. A quaternion is a four-tuple of real numbers, $[s, x, y, z]$, or, equivalently, $[s, v]$, consisting of a scalar, $s$, and a three-dimensional vector, $v$.

The quaternion is an alternative to the axis and angle representation in that it contains the same information in a different, but mathematically convenient, form. Importantly, it is in a form that can be interpolated as well as used in concatenating a series of rotations into a single representation. The axis and angle information of a quaternion can be viewed as an orientation of an object relative to its initial object space definition, or it can be considered as the representation of a rotation to apply to an object definition. In the former view, being able to interpolate between represented orientations is important in generating key-frame animation. In the latter view, concatenating a series of rotations into a simple representation is a common and useful operation to perform to apply a single, compound transformation to an object definition.

### *Basic quaternion math*

Before interpolation can be explained, some basic quaternion math must be understood. In the equations that follow, a bullet operator represents dot product, and "×" denotes cross-product. *Quaternion addition* is simply the four-tuple addition of quaternion representations, $[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2]$. *Quaternion multiplication* is defined as Equation 2.24. Notice that quaternion multiplication is associative, $(q_1\, q_2)q_3 = q_1(q_2\, q_3)$, but is not commutative, $q_1\, q_2 \neq q_2\, q_1$.

$$[s_1, v_1]\,[s_2, v_2] = [s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2] \tag{2.24}$$

A point in space, $v$, or, equivalently, the vector from the origin to the point, is represented as $[0, v]$. It is easy to see that quaternion multiplication of two orthogonal vectors ($v_1 \bullet v_2 = 0$) computes the cross-product of those vectors (Eq. 2.25).

$$[0, v_1]\,[0, v_2] = [0, v_1 \times v_2] \;\text{ if }\; v_1 \cdot v_2 = 0 \tag{2.25}$$

The quaternion $[1, (0, 0, 0)]$ is the multiplicative identity; that is,

$$[s,\ \ v]\,[1, (0, 0, 0)] = [s, v] \tag{2.26}$$

The *inverse of a quaternion*, $[s, v]^{-1}$, is obtained by negating its vector part and dividing both parts by the magnitude squared (the sum of the squares of the four components), as shown in Equation 2.27.

$$q^{-1} = (1/||q||)^2[s, -v] \qquad \text{where } ||q|| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag{2.27}$$

Multiplication of a quaternion, q, by its inverse, $q^{-1}$, results in the multiplicative identity $[1, (0, 0, 0)]$. A unit-length quaternion (also referred to here as a unit quaternion), $\hat{q}$, is created by dividing each of the four components by the square root of the sum of the squares of those components (Eq. 2.28).

$$\hat{q} = q/(||q||) \tag{2.28}$$

### *Representing rotations using quaternions*

A rotation is represented in a quaternion form by encoding axis–angle information. Equation 2.29 shows a unit quaternion representation of a rotation of an angle, *u*, about a unit axis of rotation (*x*, *y*, *z*).

$$QuatRot(\theta, (x, y, z)) \equiv [\cos(\theta/2), \sin(\theta/2)(x, y, z)] \tag{2.29}$$

Notice that rotating some angle around an axis is the same as rotating the negative angle around the negated axis. In quaternions, this is manifested by the fact that a quaternion, $q = [s, v]$, and its negation, $-q = [-s, -v]$, represent the same rotation. The two negatives in this case cancel each other out and produce the same rotation. In Equation 2.30, the quaternion *q* represents a rotation of *u* about a unit axis of rotation (*x*, *y*, *z*), i.e.,

$$\begin{aligned} -q &= [-\cos(\theta/2), -\sin(\theta/2)(x, y, z)] \\ &= [\cos(180-(\theta/2)), \sin(\theta/2)(-(x, y, z))] \\ &= [\cos((360-\theta)/2), \sin(180-\theta/2)(-(x, y, z))] \\ &= [\cos((360-\theta)/2), \sin(360-\theta/2)(-(x, y, z))] \\ &\equiv QuatRot(-\theta, -(x, y, z)) \\ &\equiv QuatRot(\theta, (x, y, z)) \end{aligned} \tag{2.30}$$

Negating *q* results in a negative rotation around the negative of the axis of rotation, which is the same rotation represented by *q* (Eq. 2.30).

### *Rotating vectors using quaternions*

To rotate a vector, *v*, using quaternion math, represent the vector as $[0, v]$ and represent the rotation by a quaternion, *q*. The vector is rotated according to Equation 2.31.

$$Rot_q(v) \equiv qvq^{-1} = v' \tag{2.31}$$

A series of rotations can be concatenated into a single representation by quaternion multiplication. Consider a rotation represented by a quaternion, *p*, followed by a rotation represented by a quaternion, *q*, on a vector, *v* (Eq. 2.32).

$$Rot_q(Rot_p(v)) = q(pvp^{-1})q^{-1} = (qp)v(qp)^{-1} = Rot_{qp}(v) \tag{2.32}$$

The inverse of a quaternion represents rotation about the same axis by the same amount but in the reverse direction. Equation 2.33 shows that rotating a vector by a quaternion, *q*, followed by rotating the result by the inverse of that same quaternion produces the original vector.

$$Rot_{q^{-1}}(Rot_q(v)) = q^{-1}(qvp^{-1})q = v \tag{2.33}$$

Also, notice that in performing rotation, $qvq^{-1}$, all effects of magnitude are divided out due to the multiplication by the inverse of the quaternion. Thus, any scalar multiple of a quaternion represents the same rotation as the corresponding unit quaternion (similar to how the homogeneous representation of points is scale invariant).

A concise list of quaternion arithmetic and conversions to and from other representations can be found in Appendix B.3.4.

### 2.2.5 Exponential map representation

Exponential maps, similar to quaternions, represent an orientation as an axis of rotation and an associated angle of rotation as a single vector [3]. The direction of the vector is the axis of rotation and the magnitude is the amount of rotation. In addition, a zero rotation is assigned to the zero vector, making the representation continuous at the origin. Notice that an exponential map uses three parameters instead of the quaternion's four. The main advantage is that it has well-formed derivatives. These are important, for example, when dealing with angular velocity.

This representation does have some drawbacks. Similar to Euler angles, it has singularities. However, in practice, these can be avoided. Also, it is difficult to concatenate rotations using exponential maps and is best done by converting to rotation matrices.

## 2.3 Summary

Linear transformations represented by $4 \times 4$ matrices are a fundamental operation in computer graphics and animation. Understanding their use, how to manipulate them, and how to control round-off error is an important first step in mastering graphics and animation techniques.

There are several orientation representations to choose from. The most robust representation of orientation is quaternions, but fixed angle, Euler angle, and axis–angle are more intuitive and easier to implement. Fixed angles and Euler angles suffer from gimbal lock and axis–angle is not easy to composite, but they are useful in some situations. Exponential maps also do not concatenate well but offer some advantages when working with derivatives of orientation. Appendix B.3.4 contains useful conversions between quaternions and other representations.

## References

[1] Craig J. Robotics. New York: Addison-Wesley; 1989.
[2] Foley J, van Dam A, Feiner S, Hughes J. Computer Graphics: Principles and Practice. 2nd ed. New York: Addison-Wesley; 1990.
[3] Grassia FS. Practical Parameterization of Rotations Using the Exponential Map. The Journal of Graphics Tools 1998;3.3.
[4] Mortenson M. Geometric Modeling. New York: John Wiley & Sons; 1997.
[5] Shoemake K. Animating Rotation with Quaternion Curves. In: Barsky BA, editor. Computer Graphics. Proceedings of SIGGRAPH 85, vol. 19(3). San Francisco, Calif; August 1985. p. 143–52.
[6] Strong G. Linear Algebra and Its Applications. New York: Academic Press; 1980.
[7] Watt A, Watt M. Advanced Animation and Rendering Techniques. New York: Addison-Wesley; 1992.