CHAPTER 3

Matrix Algebra and Transformations

In this chapter, you will learn about some mathematical objects—vectors and matrices—that are essential to rendering three-dimensional scenes. After learning some theoretical background, you will apply this knowledge to create a **`Matrix`** class that will be fundamental in manipulating the position and orientation of geometric objects. Finally, you will learn how to incorporate matrix objects into the rendering of an interactive 3D scene.

3.1 INTRODUCTION TO VECTORS AND MATRICES

When creating animated and interactive graphics applications, you will frequently want to transform sets of points defining the shape of geometric objects. You may want to translate the object to a new position, rotate the object to a new orientation, or scale the object to a new size. Additionally, you will want to project the viewable region of the scene into the space that is rendered by OpenGL. This will frequently be a perspective projection, where objects appear smaller the further away they are from the virtual camera. These ideas were first introduced in Chapter 1, in the discussion of the graphics pipeline and geometry processing; these calculations take place in a vertex shader. In Chapter 2, you learned how to work with points (using the vector data types **`vec3`** and **`vec4`**), and you implemented a transformation (two-dimensional translation). In this chapter, you will learn about a data structure called a *matrix*—a rectangular or two-dimensional

DOI: 10.1201/9781003181378-3

array of numbers—that is capable of representing all of the different types of transformations you will need in three-dimensional graphics. For these matrix-based transformations, you will learn how to

- apply a transformation to a point
- combine multiple transformations into a single transformation
- create a matrix corresponding to a given description of a transformation

3.1.1 Vector Definitions and Operations

In the previous chapter, you worked with vector data types, such as **`vec3`**, which are data structures whose components are floating-point numbers. In this section, you will learn about vectors from a mathematical point of view. For simplicity, the topics in this section will be introduced in a two-dimensional context. In a later section, you will learn how each of the concepts is generalized to higher-dimensional settings.

A *coordinate system* is defined by an origin point and the orientation and scale of a set of coordinate axes. A *point* $P = (x, y)$ refers to a location in space, specified relative to a coordinate system. A point is typically drawn as a dot, as illustrated by Figure 3.1. Note that in the diagram, the axes are oriented so that the x-axis is pointing to the right, and the y-axis is pointing upward; the direction of the arrow represents the direction in which the values increase. The orientation of the coordinate axes is somewhat arbitrary, although having the x-axis point to the right is a standard choice. In most two-dimensional mathematics diagrams, as well as OpenGL, the y-axis points upward. However, in many two-dimensional

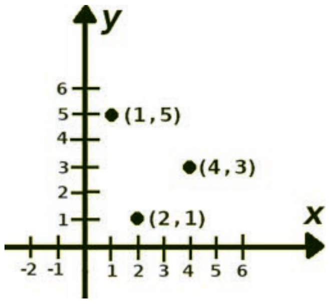

FIGURE 3.1 A collection of points.

computer graphics applications, the origin is placed in the top-left corner of the screen or drawing area, and the y-axis points downward. It is always important to know the orientation of the coordinate system you are working in!

A *vector* $\boldsymbol{v} = \langle m,n \rangle$ refers to a *displacement*—an amount of change in each coordinate—and is typically drawn as an arrow pointing along the direction of displacement, as illustrated by Figure 3.2. The point where the arrow begins is called the *initial point* or *tail*; the point where the arrow ends is called the *terminal point* or *head*, and indicates the result when the displacement has been applied to the initial point. The distance between the initial and terminal points of the vector is called its *length* or *magnitude*, and can be calculated from the components of the vector. Vectors are not associated with any particular location in space; the same vector may exist at different locations. A vector whose initial point is located at the origin (when a coordinate system is specified) is said to be in *standard position*.

To further emphasize the difference between location and displacement, consider navigating in a city whose roads are arranged as lines in a grid. If you were to ask someone to meet you at the intersection of 42nd Street and 5th Avenue, this refers to a particular location and corresponds to a point. If you were to ask someone to travel five blocks north and three blocks east from their current position, this refers to a displacement (a change in their position) and corresponds to a vector.

Each of these mathematical objects—points and vectors—are represented by a list of numbers, but as explained above, they have different geometric interpretations. In this book, notation will be used to quickly distinguish these objects. When referring to vectors, bold lowercase letters will be used for variables, and angle brackets will be used when listing components, as in $\boldsymbol{v} = \langle 1,\ 4 \rangle$. In contrast, when referring to points, regular (that is, non-bold) uppercase letters will be used for variables,

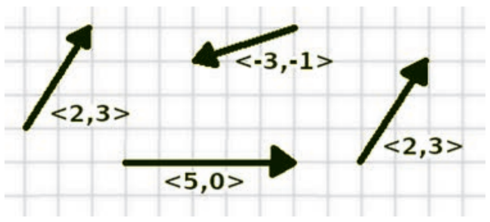

FIGURE 3.2 A collection of vectors.

and standard parentheses will be used when listing components, as in $P = (3, 2)$. Individual numbers (that are not part of a point or vector) are often called *scalars* in this context (to clearly distinguish them from points and vectors) and will be represented with regular lowercase letters, as in $x=5$. Additionally, subscripted variables will sometimes be used when writing the components of a point or vector, and these subscripts may be letters or numbers, as in $P=\left(p_x, p_y\right)$ or $P=\left(p_1, p_2\right)$ for points, and $\boldsymbol{v}=\left\langle v_x, v_y\right\rangle$ or $\boldsymbol{v}=\langle v_1, v_2\rangle$ for vectors.

While many algebraic operations can be defined on combinations of points and vectors, those with a geometric interpretation will be most significant in what follows. The first such operation is vector addition, which combines two vectors $\boldsymbol{v}=\langle v_1, v_2\rangle$ and $\boldsymbol{w}=\langle w_1, w_2\rangle$ and produces a new vector according to the following formula:

$$\boldsymbol{v}+\boldsymbol{w}=\langle v_1, v_2\rangle+\langle w_1, w_2\rangle=\langle v_1+w_1, v_2+w_2\rangle$$

For example, $\langle 2, 5\rangle+\langle 4,-2\rangle=\langle 6, 3\rangle$. Geometrically, this corresponds to displacement along the vector $\boldsymbol{v}$, followed by displacement along the vector $\boldsymbol{w}$; this can be visualized by aligning the terminal point of $\boldsymbol{v}$ with the initial point of $\boldsymbol{w}$. The result is a new vector $\boldsymbol{u}=\boldsymbol{v}+\boldsymbol{w}$ that shares the initial point of $\boldsymbol{v}$ and the terminal point of $\boldsymbol{w}$, as illustrated in Figure 3.3.

There is also a geometric interpretation for adding a vector $\boldsymbol{v}$ to a point P with the same algebraic formula; this corresponds to translating a point to a new location, which yields a new point. Align the vector $\boldsymbol{v}$ so its initial point is at location P, and the result is the point Q located at the terminal point of $\boldsymbol{v}$; this is expressed by the equation $P+\boldsymbol{v}=Q$. For example, $(2, 2)+\langle 1, 3\rangle=(3, 5)$, as illustrated in Figure 3.4.

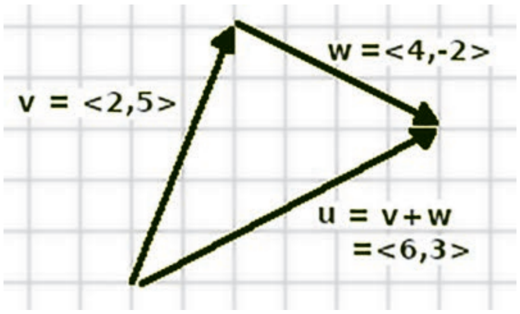

FIGURE 3.3 Vector addition.

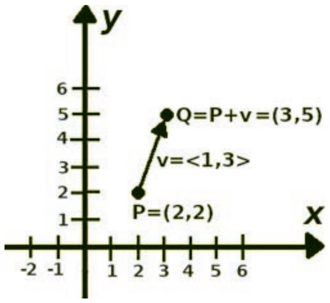

FIGURE 3.4 Adding a vector to a point.

The sum of two points does not have any clear geometric interpretation, but the difference of two points does. Rearranging the equation $P+\boldsymbol{v}=Q$ yields the equation $\boldsymbol{v}=Q-P$, which can be thought of as calculating the displacement vector between two points by subtracting their coordinates.

The componentwise product of two points or two vectors does not have any clear geometric interpretation. However, multiplying a vector $\boldsymbol{v}$ by a scalar c does. This operation is called *scalar multiplication* and is defined by

$$c\cdot\boldsymbol{v}=c\cdot\langle v_1,v_2\rangle=\langle c\cdot v_1,c\cdot v_2\rangle$$

For example, $2\cdot\langle 3,2\rangle=\langle 6,4\rangle$. Geometrically, this corresponds to scaling the vector; the length of $\boldsymbol{v}$ is multiplied by a factor of $|c|$ (the absolute value of c), and the direction is reversed when $c<0$. Figure 3.5 illustrates scaling a vector $\boldsymbol{v}$ by various amounts.

The operations of vector addition and scalar multiplication will be particularly important in what follows. Along these lines, consider the vectors

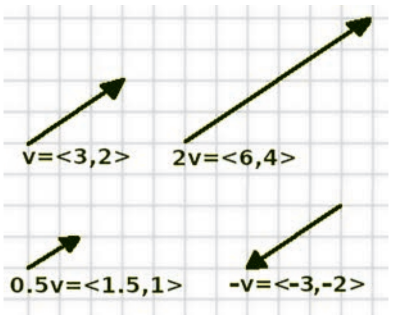

FIGURE 3.5 Scalar multiplication.

$\boldsymbol{i} = \langle 1, 0 \rangle$ and $\boldsymbol{j} = \langle 0, 1 \rangle$. Any other vector $\boldsymbol{v} = \langle x, y \rangle$ can be written in terms of $\boldsymbol{i}$ and $\boldsymbol{j}$ using vector multiplication and scalar multiplication in exactly one way, as follows:

$$\boldsymbol{v} = \langle x, y \rangle = \langle x, 0 \rangle + \langle 0, y \rangle = x \cdot \langle 1, 0 \rangle + y \cdot \langle 0, 1 \rangle = x \cdot \boldsymbol{i} + y \cdot \boldsymbol{j}$$

Any set of vectors with these properties is called a *basis*. There are many sets of basis vectors, but since $\boldsymbol{i}$ and $\boldsymbol{j}$ are in a sense the simplest such vectors, they are called the *standard basis* (for two-dimensional space).

3.1.2 Linear Transformations and Matrices

The main goal of this chapter is to design and create functions that transform sets of points or vectors in certain geometric ways. These are often called vector functions, to distinguish them from functions that have scalar valued input and output. They may also be called transformations to emphasize their geometric interpretation. These functions may be written as $F\big((p_1, p_2) \big) = (q_1, q_2)$, or $F\big(\langle v_1, v_2 \rangle\big) = \langle w_1, w_2 \rangle$, or occasionally vectors will be written in column form, as

$$F\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right) = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

The latter of these three expressions is most commonly used in a traditional mathematical presentation, but the alternative expressions are often used for writing mathematics within sentences.

Vector functions with differing levels of complexity are easy to write. As simple examples, one may consider the *zero function*, where the output is always the zero vector:

$$F\big(\langle v_1, v_2 \rangle\big) = \langle 0, 0 \rangle$$

There is also the *identity function*, where the output is always equal to the input:

$$F\big(\langle v_1, v_2 \rangle\big) = \langle v_1, v_2 \rangle$$

At the other extreme, one could invent all manner of arbitrary complicated expressions, such as

$$F(\langle v_1, v_2 \rangle) = \langle v_1 - 3\cdot\cos(3v_2), (v_1)^7 + \ln(|v_2|) + \pi \rangle$$

Most of which lack any geometrical significance whatsoever.

The key is to find a relatively simple class of vector functions which can perform the types of geometric transformations described at the beginning of this chapter. It will be particularly helpful to choose to work with vector functions that can be simplified in some useful way. One such set of functions are *linear functions* or *linear transformations*, which are functions F that satisfy the following two equations relating to scalar multiplication and vector addition: for any scalar c and vectors $\boldsymbol{v}$ and $\boldsymbol{w}$,

$$F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$$

$$F(\boldsymbol{v} + \boldsymbol{w}) = F(\boldsymbol{v}) + F(\boldsymbol{w})$$

One advantage to working with such functions involves the standard basis vectors $\boldsymbol{i}$ and $\boldsymbol{j}$: if F is a linear function and the values of $F(\boldsymbol{i})$ and $F(\boldsymbol{j})$ are known, then it is possible to calculate the value of $F(\boldsymbol{v})$ for any vector $\boldsymbol{v}$, even when a general formula for the function F is not given. For example, assume that F is a linear function, $F(\boldsymbol{i}) = \langle 1,2 \rangle$ and $F(\boldsymbol{j}) = \langle 3,1 \rangle$. Then by using the equations that linear functions satisfy, the value of $F(\ \langle 4, 5 \rangle\ )$ can be calculated as follows:

$$\begin{aligned}
F(\ \langle 4,5 \rangle\ ) &= F(\ \langle 4,0 \rangle + \langle 0,5 \rangle\ ) \\
&= F(\ \langle 4,0 \rangle\ ) + F(\ \langle 0,5 \rangle\ ) \\
&= F(\ 4\cdot\langle 1,0 \rangle\ ) + F(\ 5\cdot\langle 0,1 \rangle\ ) \\
&= 4\cdot F(\ \langle 1,0 \rangle\ ) + 5\cdot F(\ \langle 0,1 \rangle\ ) \\
&= 4\cdot F(\boldsymbol{i}) + 5\cdot F(\boldsymbol{j}) \\
&= 4\cdot\langle 1,2 \rangle + 5\cdot\langle 3,1 \rangle \\
&= \langle 4,8 \rangle + \langle 15,5 \rangle \\
&= \langle 19,13 \rangle
\end{aligned}$$

In a similar way, the general formula for $F(\ \langle x\ \ y \rangle\ )$ for this example can be calculated as follows:

$$\begin{aligned}
F(\langle x, y\rangle) &= F(\langle x, 0\rangle) + (\langle 0, y\rangle) \\
&= F(\langle x, 0\rangle) + F(\langle 0, y\rangle) \\
&= F(x \cdot \langle 1, 0\rangle) + F(y \cdot \langle 0, 1\rangle) \\
&= x \cdot F(\langle 1, 0\rangle) + y \cdot F(\langle 0, 1\rangle) \\
&= x \cdot F(\boldsymbol{i}) + y \cdot F(\boldsymbol{j}) \\
&= x \cdot \langle 1, 2\rangle + y \cdot \langle 3, 1\rangle \\
&= \langle x, 2x\rangle + \langle 3y, y\rangle \\
&= \langle x + 3y, 2x + y\rangle
\end{aligned}$$

In fact, the same line of reasoning establishes the most general case: if *F* is a linear function, where $\text{F}(\boldsymbol{i}) = a, c$ and $\text{F}(\boldsymbol{j}) = b,\ d$, then the formula for the function *F* is $F(\langle x, y\rangle) = \langle a \cdot x + b \cdot y,\ c \cdot x + d \cdot y\rangle$, since

$$\begin{aligned}
F(\langle x, y\rangle) &= F(\langle x, 0\rangle + \langle 0, y\rangle) \\
&= F(\langle x, 0\rangle) + F(\langle 0, y\rangle) \\
&= F(x \cdot \langle 1, 0\rangle) + F(y \cdot \langle 0, 1\rangle) \\
&= x \cdot F(\langle 1, 0\rangle) + y \cdot F(\langle 0, 1\rangle) \\
&= x \cdot F(\boldsymbol{i}) + y \cdot F(\boldsymbol{j}) \\
&= x \cdot \langle a, c\rangle + y \cdot \langle b, d\rangle \\
&= \langle a \cdot x, c \cdot x\rangle + \langle b \cdot y, d \cdot y\rangle \\
&= \langle a \cdot x + b \cdot y, c \cdot x + d \cdot y\rangle
\end{aligned}$$

In addition to these useful algebraic properties, it is possible to visualize the geometric effect of a linear function on the entire space. To begin, consider a unit square consisting of the points $\langle u, v\rangle = u \cdot \boldsymbol{i} + v \cdot \boldsymbol{j}$, where $0 \le u \le 1$ and $0 \le v \le 1$, the dot-shaded square labeled as *S* on the left side of Figure 3.6.

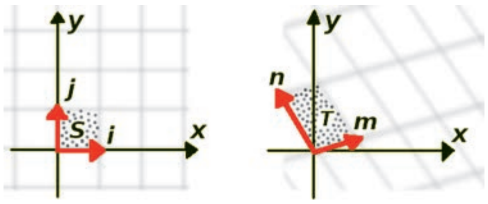

FIGURE 3.6 The geometric effects of a linear transformation.

Assume F is a linear function with $F(\boldsymbol{i}) = \boldsymbol{m}$ and $F(\boldsymbol{j}) = \boldsymbol{n}$. Then, the set of points in S is transformed to the set of points that can be written as

$$F(u \cdot \boldsymbol{i} + v \cdot \boldsymbol{j}) = u \cdot F(\boldsymbol{i}) + v \cdot F\langle \boldsymbol{j} \rangle = u \cdot \boldsymbol{m} + v \cdot \boldsymbol{n}$$

This area is indicated by the dot-shaded parallelogram labeled as T on the right side of Figure 3.6. Similarly, the function F transforms each square region on the left of Figure 3.6 into a parallelogram shaped region on the right of Figure 3.6.

The formula for a linear function F can be written in column form as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} a \cdot x + b \cdot y \\ c \cdot x + d \cdot y \end{bmatrix}$$

It is useful to think of the vector $\langle x, y \rangle$ as being operated on by the set of numbers a, b, c, d, which naturally leads us to a particular mathematical notation: these numbers can be grouped into a rectangular array of numbers called a *matrix*, typically enclosed within square brackets, and the function F can be rewritten as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

In accordance with this notation, the *product* of a matrix and a vector is defined by the following equation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \cdot x + b \cdot y \\ c \cdot x + d \cdot y \end{bmatrix}$$

For example, consider the following linear function:

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Then, $F(\langle 5,6 \rangle)$ can be calculated as follows:

$$F\left(\begin{bmatrix} 5 \\ 6 \end{bmatrix}\right) = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 2 \cdot 5 + 3 \cdot 6 \\ 4 \cdot 5 + 1 \cdot 6 \end{bmatrix} = \begin{bmatrix} 28 \\ 26 \end{bmatrix}$$

Similarly, to calculate $F(\langle -3,1 \rangle)$,

$$F\left(\begin{bmatrix} -3 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot (-3) + 3 \cdot 1 \\ 4 \cdot (-3) + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} -3 \\ -11 \end{bmatrix}$$

Once again, it will be helpful to use notation to distinguish between matrices and other types of mathematical objects (scalars, points, and vectors). When referring to a matrix, bold uppercase letters will be used for variables, and square brackets will be used to enclose the grid of numbers or variables. This notation can be used to briefly summarize the previous observation: if F is a linear function, then F can be written in the form $F(\boldsymbol{v}) = \boldsymbol{A} \cdot \boldsymbol{v}$ for some matrix $\boldsymbol{A}$. Conversely, one can also show that if F is a vector function defined by matrix multiplication, that is, if $F(\boldsymbol{v}) = \boldsymbol{A} \cdot \boldsymbol{v}$, then F also satisfies the equations that define a linear function; this can be verified by straightforward algebraic calculations. Therefore, these two descriptions of vector functions—those that are linear and those defined by matrix multiplication—are equivalent; they define precisely the same set of functions.

Two of the vector functions previously mentioned can be represented using matrix multiplication: the zero function can be written as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \cdot x + 0 \cdot y \\ 0 \cdot x + 0 \cdot y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

while the identity function can be written as

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \cdot x + 0 \cdot y \\ 0 \cdot x + 1 \cdot y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix in the definition of the identity function is called the *identity matrix* and appears in a variety of contexts, as you will see.

When introducing notation for vectors, it was indicated that subscripted variables will sometimes be used for the components of a vector. When working with a matrix, *double* subscripted variables will sometimes be used for its components; the subscripts will indicate the position (row and column) of the component in the matrix. For example, the variable a_{12} will refer to the entry in row 1 and column 2 of the matrix $\boldsymbol{A}$, and in general, a_{mn} will refer to the entry in row m and column n of the matrix $\boldsymbol{A}$. The contents of the entire matrix $\boldsymbol{A}$ can be written as

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

At this point, you know how to apply a linear function (written in matrix notation) to a point or a vector. In many contexts (and in computer graphics in particular), you will want to apply multiple functions to a set of points. Given two functions F and G, a new function H can be created by defining $H(\boldsymbol{v}) = F(G(\boldsymbol{v}))$; the function H is called the *composition* of F and G. As it turns out, if F and G are linear functions, then H will be a linear function as well. This can be verified by checking that the two linearity equations hold for H, making use of the fact that the linearity equations are true for the functions F and G by assumption. The derivation relating to vector addition is as follows:

$$\begin{aligned} H(\boldsymbol{v}+\boldsymbol{w}) &= F(G(\boldsymbol{v}+\boldsymbol{w})) \\ &= F(G(\boldsymbol{v})+G(\boldsymbol{w})) \\ &= F(G(\boldsymbol{v}))+F(G(\boldsymbol{w})) \\ &= H(\boldsymbol{v})+H(\boldsymbol{w}) \end{aligned}$$

The derivation relating to scalar multiplication is as follows:

$$\begin{aligned} H(c\cdot\boldsymbol{v}) &= F(G(c\cdot\boldsymbol{v})) \\ &= F(c\cdot G(\boldsymbol{v})) \\ &= c\cdot F(G(\boldsymbol{v})) \\ &= c\cdot H(\boldsymbol{v}) \end{aligned}$$

The reason this is significant is that given two linear functions—each of which can be represented by a matrix, as previously observed—their composition can be represented by a *single* matrix, since the composition is also a linear function. By repeatedly applying this reasoning, it follows that the composition of *any number* of linear functions can be represented by a *single* matrix. This immediately leads to the question: how can the matrix corresponding to the composition be calculated? Algebraically, given two transformations $F(\boldsymbol{v}) = \boldsymbol{A}\cdot\boldsymbol{v}$ and $G(\boldsymbol{v}) = \boldsymbol{B}\cdot\boldsymbol{v}$, the goal is to find a matrix $\boldsymbol{C}$ such that the transformation $H(\boldsymbol{v}) = \boldsymbol{C}\cdot\boldsymbol{v}$ is equal to $F(G(\boldsymbol{v})) = \boldsymbol{A}\cdot(\boldsymbol{B}\cdot\boldsymbol{v})$ for all vectors $\boldsymbol{v}$. In this case, one writes $\boldsymbol{C} = \boldsymbol{A}\cdot\boldsymbol{B}$. This operation is referred to as *matrix multiplication*, and $\boldsymbol{C}$ is called the *product* of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

The formula for matrix multiplication may be deduced by computing both sides of the equation $\boldsymbol{C}\cdot\boldsymbol{v} = \boldsymbol{A}\cdot(\boldsymbol{B}\cdot\boldsymbol{v})$ and equating the coefficients of x and y on either side of the equation. Expanding $\boldsymbol{C}\cdot\boldsymbol{v}$ yields

$$\boldsymbol{C}\cdot\boldsymbol{v} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_{11}\cdot x + c_{12}\cdot y \\ c_{21}\cdot x + c_{22}\cdot y \end{bmatrix}$$

Similarly, expanding $\boldsymbol{A}\cdot(\boldsymbol{B}\cdot\boldsymbol{v})$ yields

$$\begin{aligned}
\boldsymbol{A}\cdot(\boldsymbol{B}\cdot\boldsymbol{v}) &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\cdot\left(\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}\cdot\begin{bmatrix} x \\ y \end{bmatrix}\right) \\
&= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\cdot\begin{bmatrix} b_{11}\cdot x + b_{12}\cdot y \\ b_{21}\cdot x + b_{22}\cdot y \end{bmatrix} \\
&= \begin{bmatrix} a_{11}(b_{11}\cdot x + b_{12}\cdot y) + a_{12}(b_{21}\cdot x + b_{22}\cdot y) \\ a_{21}(b_{11}\cdot x + b_{12}\cdot y) + a_{22}(b_{21}\cdot x + b_{22}\cdot y) \end{bmatrix} \\
&= \begin{bmatrix} (a_{11}\cdot b_{11} + a_{12}\cdot b_{21})\cdot x + (a_{11}\cdot b_{12} + a_{12}\cdot b_{22})\cdot y \\ (a_{21}\cdot b_{11} + a_{22}\cdot b_{21})\cdot x + (a_{21}\cdot b_{12} + a_{22}\cdot b_{22})\cdot y \end{bmatrix}
\end{aligned}$$

Thus, matrix multiplication $\boldsymbol{C} = \boldsymbol{A}\cdot\boldsymbol{B}$ is defined as

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} (a_{11}\cdot b_{11} + a_{12}\cdot b_{21}) + (a_{11}\cdot b_{12} + a_{12}\cdot b_{22}) \\ (a_{21}\cdot b_{11} + a_{22}\cdot b_{21}) + (a_{21}\cdot b_{12} + a_{22}\cdot b_{22}) \end{bmatrix}$$

This formula can be written more simply using a vector operation called the *dot product*. Given vectors $\boldsymbol{v} = \langle v_1, v_2 \rangle$ and $\boldsymbol{w} = \langle w_1, w_2 \rangle$, the dot product $d = \boldsymbol{v} \bullet \boldsymbol{w}$ is a (scalar) number, defined by

$$d = \boldsymbol{v} \bullet \boldsymbol{w} = \langle v_1, v_2 \rangle \bullet \langle w_1, w_2 \rangle = v_1 \cdot w_1 + v_2 \cdot w_2$$

As an example of a dot product calculation, consider

$$\langle 3, 4 \rangle \bullet \langle 7, 5 \rangle = 3 \cdot 7 + 4 \cdot 5 = 41$$

To restate the definition of matrix multiplication: partition the matrix $\boldsymbol{A}$ into row vectors and the matrix $\boldsymbol{B}$ into column vectors. The entry c_{mn} of the product matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ is equal to the dot product of row vector m from matrix $\boldsymbol{A}$ (denoted by $\boldsymbol{a}_m$) and column vector n from matrix $\boldsymbol{B}$ (denoted by $\boldsymbol{b}_n$), as illustrated in the following formula, where partitions are indicated by dashed lines.

$$\begin{aligned}
\boldsymbol{A} \cdot \boldsymbol{B} &= \left[\begin{array}{cc} a_{11} & a_{12} \\ \hdashline a_{21} & a_{22} \end{array}\right] \cdot \left[\begin{array}{c:c} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array}\right] \\
&= \left[\begin{array}{c} \boldsymbol{a}_1 \\ \hdashline \boldsymbol{a}_2 \end{array}\right] \cdot \left[\begin{array}{c:c} \boldsymbol{b}_1 & \boldsymbol{b}_2 \end{array}\right] \\
&= \begin{bmatrix} \boldsymbol{a}_1 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_1 \bullet \boldsymbol{b}_2 \\ \boldsymbol{a}_2 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_2 \bullet \boldsymbol{b}_2 \end{bmatrix} \\
&= \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}
\end{aligned}$$

As an example of a matrix multiplication computation, consider

$$\begin{aligned}
\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix} &= \begin{bmatrix} \langle 2, 3 \rangle \bullet \langle 9, 7 \rangle & \langle 2, 3 \rangle \bullet \langle 8, 6 \rangle \\ \langle 4, 5 \rangle \bullet \langle 9, 7 \rangle & \langle 4, 5 \rangle \bullet \langle 8, 6 \rangle \end{bmatrix} \\
&= \begin{bmatrix} (2 \cdot 9 + 3 \cdot 7) & (2 \cdot 8 + 3 \cdot 6) \\ (4 \cdot 9 + 5 \cdot 7) & (4 \cdot 8 + 5 \cdot 6) \end{bmatrix} \\
&= \begin{bmatrix} 39 & 34 \\ 71 & 62 \end{bmatrix}
\end{aligned}$$

In general, matrix multiplication can quickly become tedious, and so a software package is typically used to handle these and other matrix-related calculations.

It is important to note that, in general, matrix multiplication is not a commutative operation. Given matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, the product $\boldsymbol{A}\cdot\boldsymbol{B}$ is usually not equal to the product $\boldsymbol{B}\cdot\boldsymbol{A}$. For example, calculating the product of the matrices from the previous example in the opposite order yields

$$\begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} \langle 9,8\rangle \bullet \langle 2,4\rangle & \langle 9,8\rangle \bullet \langle 2,5\rangle \\ \langle 7,2\rangle \bullet \langle 6,4\rangle & \langle 7,6\rangle \bullet \langle 3,5\rangle \end{bmatrix}$$

$$= \begin{bmatrix} (9\cdot 2+8\cdot 4) & (9\cdot 3+8\cdot 5) \\ (7\cdot 2+6\cdot 4) & (7\cdot 3+6\cdot 5) \end{bmatrix}$$

$$= \begin{bmatrix} 50 & 67 \\ 38 & 51 \end{bmatrix}$$

This fact has a corresponding geometric interpretation as well: the order in which geometric transformations are performed makes a difference. For example, let T represent translation by $\langle 1, 0\rangle$, and let R represent rotation around the origin by 90°. If P denotes the point $P=(2, 0)$, then $R(T(P))=R(3,0)=(0,3)$, while $T(R(P))=T(0,2)=(1,2)$, as illustrated in Figure 3.7. Thus, $R(T(P))$ does not equal $T(R(P))$.

The identity matrix $\boldsymbol{I}$, previously mentioned, is the matrix

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

The identity matrix has multiplication properties similar to those of the number 1 in ordinary multiplication. Just as for any number x, it is true that $1\cdot x = x$ and $x\cdot 1 = x$, for any matrix $\boldsymbol{A}$ it can be shown with algebra that $\boldsymbol{I}\cdot\boldsymbol{A} = \boldsymbol{A}$ and $\boldsymbol{A}\cdot\boldsymbol{I} = \boldsymbol{A}$. Because of these properties, both 1 and $\boldsymbol{I}$ are called *identity elements* in their corresponding mathematical contexts. Similarly, the identity function is the identity element in the context of function composition. Thinking of vector functions as geometric transformations, the identity function does not change the location of any points; the geometric transformations translation by ⟨0, 0⟩, rotation around the origin by

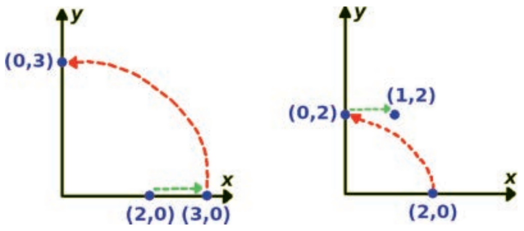

FIGURE 3.7 Geometric transformations (translation and rotation) are not commutative.

0 degrees, and scaling all components by a factor of 1 are all equivalent to the identity function.

The concept of identity elements leads to the concept of *inverse* elements. In a given mathematical context, combining an object with its inverse yields the identity element. For example, a number x multiplied by its inverse equals 1; a function composed with its inverse function yields the identity function. Analogously, a matrix multiplied by its inverse matrix results in the identity matrix. Symbolically, the inverse of a matrix $\boldsymbol{A}$ is a matrix $\boldsymbol{M}$ such that $\boldsymbol{A}\cdot\boldsymbol{M}=\boldsymbol{I}$ and $\boldsymbol{M}\cdot\boldsymbol{A}=\boldsymbol{I}$. The inverse of the matrix $\boldsymbol{A}$ is typically written using the notation $\boldsymbol{A}^{-1}$. Using a fair amount of algebra, one can find a formula for the inverse of the matrix $\boldsymbol{A}$ by solving the equation $\boldsymbol{A}\cdot\boldsymbol{M}=\boldsymbol{I}$ for the entries of $\boldsymbol{M}$:

$$\boldsymbol{A}\cdot\boldsymbol{M}=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\cdot\begin{bmatrix} m & n \\ p & q \end{bmatrix}=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}=\boldsymbol{I}$$

Solving this equation first involves calculating the product on the left-hand side of the equation and setting each entry of the resulting matrix equal to the corresponding entry in the identity matrix. This yields a system of four equations with four unknowns (the entries of $\boldsymbol{M}$). Solving these four equations yields the following formula for the inverse of a 2-by-2 matrix:

$$\boldsymbol{M}=\boldsymbol{A}^{-1}=\begin{bmatrix} d/(ad-bc) & -b/(ad-bc) \\ -c/(ad-bc) & a/(ad-bc) \end{bmatrix}$$

The value $(a\cdot d-b\cdot c)$ appearing in the denominator of each entry of the inverse matrix is called the *determinant* of the matrix $\boldsymbol{A}$. If this value is equal to 0, then the fractions are all undefined, and the inverse of the matrix $\boldsymbol{A}$ does not exist. Analogous situations, in which certain elements do not have inverse elements, arise in other mathematical contexts. For example, in ordinary arithmetic, the number $x = 0$ does not have a multiplicative inverse, as nothing times 0 equals the identity element 1.

As may be expected, if an invertible vector function F can be represented with matrix multiplication as $F(\boldsymbol{v}) = \boldsymbol{A} \cdot \boldsymbol{v}$, then the inverse function G can be represented with matrix multiplication by the inverse of $\boldsymbol{A}$, as $G(\boldsymbol{v}) = \boldsymbol{A}^{-1} \cdot \boldsymbol{v}$, since

$$F(G(v)) = \boldsymbol{A} \cdot \boldsymbol{A}^{-1} \cdot \boldsymbol{v} = \boldsymbol{I} \cdot \boldsymbol{v} = \boldsymbol{v}$$

$$G(F(v)) = \boldsymbol{A}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{v} = \boldsymbol{I} \cdot \boldsymbol{v} = \boldsymbol{v}$$

Once again, thinking of vector functions as geometric transformations, the inverse of a function performs a transformation that is in some sense the "opposite" or "reverse" transformation. For example, the inverse of translation by $\langle m, n \rangle$ is translation by $\langle -m, -n \rangle$; the inverse of clockwise rotation by an angle a is counterclockwise rotation by an angle a (which is equivalent to clockwise rotation by an angle $-a$); the inverse of scaling the components of a vector by the values r and s is scaling the components by the values $1/r$ and $1/s$.

3.1.3 Vectors and Matrices in Higher Dimensions

All of these vector and matrix concepts can be generalized to three, four, and even higher dimensions. In this section, these concepts will be restated in a three-dimensional context. The generalization to four-dimensional space follows the same algebraic pattern. Four-dimensional vectors and matrices are used quite frequently in computer graphics, for reasons discussed later in this chapter.

Three-dimensional coordinate systems are drawn using xyz-axes, where each axis is perpendicular to the other two. Assuming that the axes are oriented as in Figure 3.1, so that the plane spanned by the x and y axes are aligned with the window used to display graphics, there are two possible directions for the (positive) z-axis: either pointing towards the viewer or away from the viewer. These two systems are called *right-handed* and *left-handed coordinate systems*, respectively, so named due to the hand-based mnemonic rule used to remember the orientation. To visualize the relative orientation of the axes in a right-handed coordinate system, using

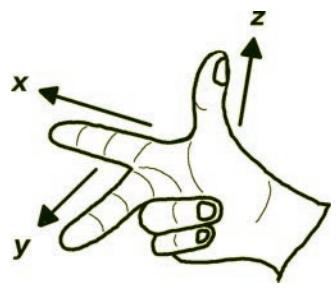

FIGURE 3.8 Using a right hand to determine the orientation of *xyz*-axes.

your right hand, imagine your index finger pointing along the *x*-axis and your middle finger perpendicular to this (in the direction the palm of your hand is facing) pointing along the *y*-axis. Then, your extended thumb will be pointing in the direction of the *z*-axis; this is illustrated in Figure 3.8. If the *z*-axis were pointing in the opposite direction, this would correspond to a left-handed coordinate system, and indeed, this would be the orientation indicated by carrying out the steps above with your left hand. Some descriptions of the right-hand rule, instead of indicating the directions of the *x* and *y* axes with extended fingers, will suggest curling the fingers of your hand in the direction from the *x*-axis to the *y*-axis; your extended thumb still indicates the direction of the *z*-axis, and the two descriptions have the same result.

In mathematics, physics, and computer graphics, it is standard practice to use a right-handed coordinate system, as shown on the left side of Figure 3.9. In computer graphics, the positive *z*-axis points directly at the viewer. Although the three axes are perpendicular to each other, when illustrated in this way, at first it may be difficult to see that the *z*-axis is perpendicular to the other axes. In this case, it may aid with visualization to imagine the *xyz*-axes as aligned with the edges of a cube hidden from view, illustrated with dashed lines as shown on the right side of Figure 3.9.

In three-dimensional space, points are written as $P = \left(p_x, p_y, p_z\right)$ or $P = \left(p_1, p_2, p_3\right)$, and vectors are written as $\boldsymbol{v} = \left\langle v_x, v_y, v_z \right\rangle$ or $\boldsymbol{v} = \left\langle v_1, v_2, v_3 \right\rangle$. Sometimes, for clarity, the components may be written as *x*, *y*, and *z* (and in four-dimensional space, the fourth component may be written as *w*). Vector addition is defined by

$$\boldsymbol{v} + \boldsymbol{w} = \left\langle v_1, v_2, v_3 \right\rangle + \left\langle w_1, w_2, w_3 \right\rangle = \langle v_1 + w_1, v_2 + w_2, v_3 + w_3 \rangle$$

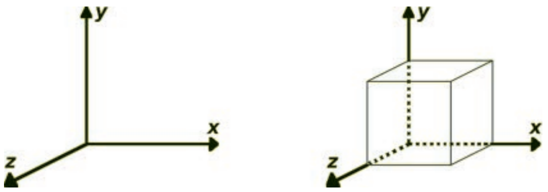

FIGURE 3.9 Coordinate axes in three dimensions.

while scalar multiplication is defined by

$$c \cdot \boldsymbol{v} = c \cdot \langle v_1, v_2, v_3 \rangle = \langle c \cdot v_1, c \cdot v_2, c \cdot v_3 \rangle$$

The standard basis for three-dimensional space consists of the vectors $\boldsymbol{i} = \langle 1, 0, 0 \rangle$, $\boldsymbol{j} = \langle 0, 1, 0 \rangle$, and $\boldsymbol{k} = \langle 0, 0, 1 \rangle$. Every vector $\boldsymbol{v} = \langle x, y, z \rangle$ can be written as a linear combination of these three vectors as follows:

$$\begin{aligned} v &= \langle x, y, z \rangle \\ &= \langle x, 0, 0 \rangle + \langle 0, y, 0 \rangle + \langle 0, 0, z \rangle \\ &= x \cdot \langle 1, 0, 0 \rangle + y \cdot \langle 0, 1, 0 \rangle + z \cdot \langle 0, 0, 1 \rangle \\ &= x \cdot \boldsymbol{i} + y \cdot \boldsymbol{j} + z \cdot \boldsymbol{k} \end{aligned}$$

The definition of a linear function is identical for vectors of any dimension, as it only involves the operations of vector addition and scalar multiplication; it does not reference the number of components of a vector at all:

$$F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$$

$$F(\boldsymbol{v} + \boldsymbol{w}) = F(\boldsymbol{v}) + F(\boldsymbol{w})$$

The values of a three-dimensional linear function can be calculated for any vector if the values of $F(\boldsymbol{i})$, $F(\boldsymbol{j})$, and $F(\boldsymbol{k})$ are known, and in general, such a function can be written in the following form:

$$F\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} a_{11} \cdot x + a_{12} \cdot y + a_{13} \cdot z \\ a_{21} \cdot x + a_{22} \cdot y + a_{23} \cdot z \\ a_{31} \cdot x + a_{32} \cdot y + a_{33} \cdot z \end{bmatrix}$$

As before, the coefficients of x, y, and z in the formula are typically grouped into a 3-by-3 matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The matrix-vector product $\boldsymbol{A}\cdot\boldsymbol{v}$ is then defined as

$$\boldsymbol{A}\cdot\boldsymbol{v} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11}\cdot x + a_{12}\cdot y + a_{13}\cdot z \\ a_{21}\cdot x + a_{22}\cdot y + a_{23}\cdot z \\ a_{31}\cdot x + a_{32}\cdot y + a_{33}\cdot z \end{bmatrix}$$

Matrix multiplication is most clearly described using the dot product, which for three-dimensional vectors is defined as

$$d = \boldsymbol{v} \bullet \boldsymbol{w} = \langle v_1, y_2 \;\; y_3 \rangle \cdot \langle w_1 \;\; w_2, w_3 \rangle = v_1\cdot w_1 + v_2\cdot w_2 + v_3\cdot w_3$$

Matrix multiplication $\boldsymbol{A}\cdot\boldsymbol{B}$ can be calculated from partitioning the entries of the two matrices into vectors: each row of the first matrix ($\boldsymbol{A}$) is written as a vector, and each column of the second matrix ($\boldsymbol{B}$) is written as a vector. Then, the value in row m and column n of the product is equal to the dot product of row vector m from matrix $\boldsymbol{A}$ (denoted by $\boldsymbol{a}_{\mathrm{m}}$) and column vector n from matrix $\boldsymbol{B}$ (denoted by $\boldsymbol{b}_n$), as illustrated below.

$$\boldsymbol{A}\cdot\boldsymbol{B} = \left[\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ \hdashline a_{21} & a_{22} & a_{23} \\ \hdashline a_{33} & a_{32} & a_{33} \end{array}\right] \cdot \left[\begin{array}{c:c:c} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{33} & b_{32} & b_{33} \end{array}\right]$$

$$= \left[\begin{array}{c} \boldsymbol{a}_1 \\ \hdashline \boldsymbol{a}_2 \\ \hdashline \boldsymbol{a}_3 \end{array}\right] \cdot \left[\begin{array}{c:c:c} \boldsymbol{b}_1 & \boldsymbol{b}_2 & \boldsymbol{b}_3 \end{array}\right]$$

$$= \begin{bmatrix} \boldsymbol{a}_1 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_1 \bullet \boldsymbol{b}_2 & \boldsymbol{a}_1 \bullet \boldsymbol{b}_3 \\ \boldsymbol{a}_2 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_2 \bullet \boldsymbol{b}_2 & \boldsymbol{a}_2 \bullet \boldsymbol{b}_3 \\ \boldsymbol{a}_3 \bullet \boldsymbol{b}_1 & \boldsymbol{a}_3 \bullet \boldsymbol{b}_2 & \boldsymbol{a}_3 \bullet \boldsymbol{b}_3 \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The 3-by-3 identity matrix $\boldsymbol{I}$ has the following form:

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

As before, this identity matrix is defined by the equations $\boldsymbol{I} \cdot \boldsymbol{A} = \boldsymbol{A}$ and $\boldsymbol{A} \cdot \boldsymbol{I} = \boldsymbol{A}$ for any matrix $\boldsymbol{A}$. Similarly, the inverse of a matrix $\boldsymbol{A}$ (if it exists) is a matrix denoted by $\boldsymbol{A}^{-1}$, defined by the equations $\boldsymbol{A} \cdot \boldsymbol{A}^{-1} = \boldsymbol{I}$ and $\boldsymbol{A}^{-1} \cdot \boldsymbol{A} = \boldsymbol{I}$. The formula for the inverse of a 3-by-3 matrix in terms of its entries is quite tedious to write down, and as mentioned previously, a software package will be used to handle these calculations.

3.2 GEOMETRIC TRANSFORMATIONS

The previous section introduced linear functions: vector functions that satisfy the linearity equations, functions which can be written with matrix multiplication. It remains to show that the geometric transformations needed in computer graphics (translation, rotation, scaling, and perspective projections) are linear functions, and then, formulas must be found for the corresponding matrices. In particular, it must be possible to determine the entries of a matrix corresponding to a description of a transformation, such as "translate along the x direction by 3 units" or "rotate around the z-axis by 45°." Formulas for each type of transformation (in both two and three dimensions) will be derived next, in increasing order of difficulty: scaling, rotation, translation, and perspective projection. In the following sections, vectors will be drawn in standard position (the initial point of each vector will be at the origin), and vectors can be identified with their terminal points.

3.2.1 Scaling

A scaling transformation multiplies each component of a vector by a constant. For two-dimensional vectors, this has the following form (where r and s are constants):

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} r\cdot x \\ s\cdot y \end{bmatrix}$$

It can quickly be deduced and verified that this transformation can be expressed with matrix multiplication as follows:

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)=\begin{bmatrix} r\cdot x \\ s\cdot y \end{bmatrix}=\begin{bmatrix} r & 0 \\ 0 & s \end{bmatrix}\cdot\begin{bmatrix} x \\ y \end{bmatrix}$$

Similarly, the three-dimensional version of this transformation, where the z-component of a vector is scaled by a constant value t, is:

$$F\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right)=\begin{bmatrix} r\cdot x \\ s\cdot y \\ t\cdot z \end{bmatrix}=\begin{bmatrix} r & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & t \end{bmatrix}\cdot\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Observe that if all the scaling constants are equal to 1, then the formula for the scaling matrix results in the identity matrix. This corresponds to the following pair of related statements: scaling the components of a vector by 1 does not change the value of the vector, just as multiplying a vector by the identity matrix does not change the value of the vector.

3.2.2 Rotation

In two dimensions, a rotation transformation rotates vectors by a constant amount around the origin point. Unlike the case with the scaling transformation, it is not immediately clear how to write a formula for a rotation function $F(\boldsymbol{v})$ or whether rotation transformations can even be calculated with matrix multiplication. To establish this fact, it suffices to show that rotation is a linear transformation that it satisfies the two linearity equations. An informal geometric argument will be presented for each equation.

To see that $F(c\cdot\boldsymbol{v})=c\cdot F(\boldsymbol{v})$, begin by considering the endpoint of the vector $\boldsymbol{v}$, and assume that this point is at a distance d from the origin. When multiplying $\boldsymbol{v}$ by c, the resulting vector has the same direction, and the endpoint is now at a distance $c\cdot d$ from the origin. Note that rotation transformations fix the origin point and do not change the distance of a point from the origin. Applying the rotation transformation F to the vectors $\boldsymbol{v}$ and $c \cdot \boldsymbol{v}$ yields the vectors $F(\boldsymbol{v})$ and $F(c \cdot \boldsymbol{v})$; these vectors have the same direction, and their endpoints have the same distances from the origin: d and $c \cdot d$, respectively. However, the vector $c \cdot F(\boldsymbol{v})$ is also aligned with $F(\boldsymbol{v})$ and its endpoint has distance $c \cdot d$ from the origin. Therefore, the endpoints of $F(c \cdot \boldsymbol{v})$ and $c \cdot F(\boldsymbol{v})$ must be at the same position, and thus, $F(c \cdot \boldsymbol{v}) = c \cdot F(\boldsymbol{v})$. This is illustrated in Figure 3.10.

To see that $F(\boldsymbol{v}+\boldsymbol{w}) = F(\boldsymbol{v}) + F(\boldsymbol{w})$, begin by defining $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{w}$, and let $\boldsymbol{o}$ represent the origin. Due to the nature of vector addition, the endpoints of $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, together with $\boldsymbol{o}$, form the vertices of a parallelogram. Applying the rotation transformation F to this parallelogram yields a parallelogram M whose vertices are $\boldsymbol{o}$ and the endpoints of the vectors $F(\boldsymbol{u})$, $F(\boldsymbol{v})$, and $F(\boldsymbol{w})$. Again, due to the nature of vector addition, the endpoints of $F(\boldsymbol{v})$, $F(\boldsymbol{w})$, and $F(\boldsymbol{v}) + F(\boldsymbol{w})$, together with $\boldsymbol{o}$, form the vertices of a parallelogram N. Since parallelograms M and N have three vertices in common, their fourth vertex must also coincide, from which it follows that $F(\boldsymbol{v}) + F(\boldsymbol{w}) = F(\boldsymbol{u}) = F(\boldsymbol{v}+\boldsymbol{w})$. This is illustrated in Figure 3.11.

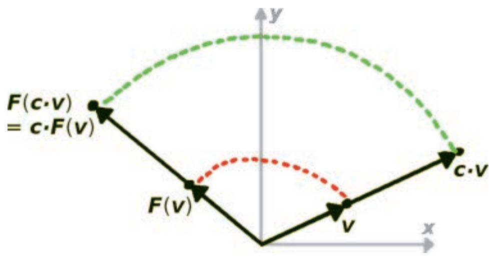

FIGURE 3.10 Illustrating that rotation transformations are linear (scalar multiplication).

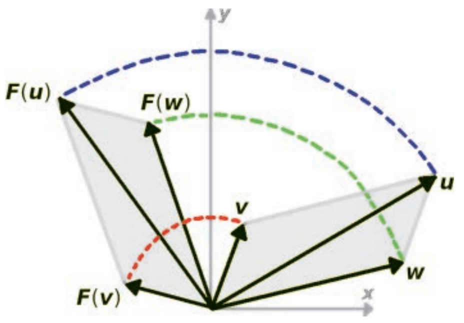

FIGURE 3.11 Illustrating that rotation transformations are linear (vector addition).

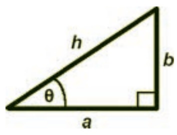

FIGURE 3.12 Right triangle with angle θ, indicating adjacent (a), opposite (b), and hypotenuse (h) side lengths.

Given that rotation is a linear transformation, the previous theoretical discussion of linear functions provides a practical method for calculating a formula for the associated matrix. The values of the function at the standard basis vectors—in two dimensions, $F(\boldsymbol{i})$ and $F(\boldsymbol{j})$—are the columns of the associated matrix. Thus, the next step is to calculate the result of rotating the vectors $\boldsymbol{i}=\langle 1, 0\rangle$ and $\boldsymbol{j}=\langle 0, 1\rangle$ around the origin (counterclockwise) by an angle θ.

This calculation requires basic knowledge of trigonometric functions. Given a right triangle with angle θ, adjacent side length a, opposite side length b, and hypotenuse length h, as illustrated in Figure 3.12, then the trigonometric functions are defined as ratios of these lengths: the sine function is defined by $\sin(\theta)=b/h$, the cosine function is defined by $\cos(\theta)=a/h$, and the tangent function is defined by $\tan(\theta)=b/a$.

As illustrated in Figure 3.13, rotating the vector $\boldsymbol{i}$ by an angle θ yields a new vector $F(\boldsymbol{i})$, which can be viewed as the hypotenuse of a right triangle. Since rotation does not change lengths of vectors, the hypotenuse has length $h=1$, which implies $\sin(\theta)=b$ and $\cos(\theta)=a$, from which it follows that $F(\boldsymbol{i})=\cos(\theta),\sin(\theta)$. This vector represents the first column of the rotation matrix.

As illustrated in Figure 3.14, rotating the vector $\boldsymbol{j}$ by an angle θ yields a new vector $F(\boldsymbol{j})$, which can once again be viewed as the hypotenuse of a right triangle with $h=1$, and as before, $\sin(\theta)=b$ and $\cos(\theta)=a$. Note that since the horizontal displacement of the vector $F(\boldsymbol{j})$ is towards the

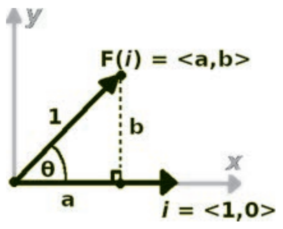

FIGURE 3.13 Rotating the basis vector $\boldsymbol{i}$ by an angle θ.

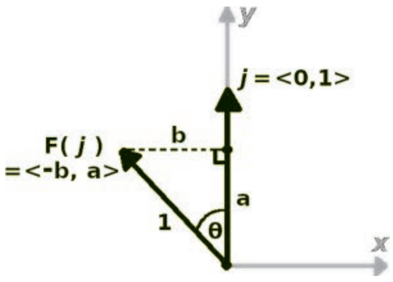

FIGURE 3.14 Rotating the basis vector $\boldsymbol{j}$ by an angle θ.

negative x direction, the value of the first vector component is $-b$, the negative of the length of the side opposite angle θ. From this, it follows that $F(\boldsymbol{j}) = \langle -\sin(\theta), \cos(\theta) \rangle$, yielding the second column of the rotation matrix.

Based on these calculations, the matrix corresponding to rotation around the origin by an angle θ in two-dimensional space is given by the matrix:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

To conclude the discussion of two-dimensional rotations, consider the following computational example. Assume that one wants to rotate the point (7, 5) around the origin by $\theta = 30°$ (or equivalently, $\theta = \pi/6$ radians). The new location of the point can be calculated as follows:

$$\begin{bmatrix} \cos(30°) & -\sin(30°) \\ \sin(30°) & \cos(30°) \end{bmatrix} \begin{bmatrix} 7 \\ 5 \end{bmatrix} = \begin{bmatrix} \sqrt{3/2} & -1/2 \\ 1/2 & \sqrt{3/2} \end{bmatrix} \begin{bmatrix} 7 \\ 5 \end{bmatrix}$$

$$= \begin{bmatrix} \left(7\sqrt{3} - 5\right)/2 \\ \left(7 + 5\sqrt{3}\right)/2 \end{bmatrix} \approx \begin{bmatrix} 3.56 \\ 7.83 \end{bmatrix}$$

In three dimensions, rotations are performed around a line, rather than a point. In theory, it is possible to rotate around any line in three-dimensional space. For simplicity, only the formulas for rotation around each of the three axes, as illustrated in Figure 3.15, will be derived in this section.

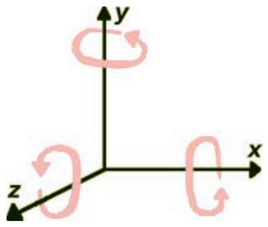

FIGURE 3.15 Rotations around the axes in three-dimensional space.

Note that the rotations appear counterclockwise when looking along each axis from positive values to the origin.

If the *xy*-plane of two-dimensional space is thought of as the set of points in three-dimensional space where $z = 0$, then the two-dimensional rotation previously discussed corresponds to rotation around the *z*-axis. Analogous to the observation that in two dimensions, rotating around a point does not move the point, in three dimensions, rotating around an axis does not move that axis. By the same reasoning as before, rotation (around an axis) is a linear transformation. Therefore, calculating the matrix corresponding to a rotation transformation *F* can be accomplished by finding the values of *F* at ***i***, ***j***, and ***k*** (the standard basis vectors in three dimensions); the vectors *F*(***i***), *F*(***j***), and *F*(***k***) the results will be the columns of the matrix.

To begin, let *F* denote rotation around the *z*-axis, a transformation which extends the previously discussed two-dimensional rotation around a point in the *xy*-plane to three-dimensional space. Since this transformation fixes the *z*-axis and therefore all *z* coordinates, based on previous work calculating *F*(***i***) and *F*(***j***), it follows that $F(\boldsymbol{i}) = \langle \cos(\theta), \sin(\theta), 0 \rangle$, $F(\boldsymbol{j}) = \langle -\sin(\theta), \cos(\theta), 0 \rangle$, and $F(\boldsymbol{k}) = \langle 0, 0, 1 \rangle$, and therefore, the matrix for this transformation is

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

Next, let F denote rotation around the *x*-axis. Evaluating the values of this function requires similar reasoning. This transformation fixes the *x*-axis, so $F(\boldsymbol{i}) = \langle 1, 0, 0 \rangle$ is the first column of the matrix. Since the transformation fixes all *x* coordinates, two-dimensional diagrams such as those in Figures 3.13 and 3.14, featuring the *yz*-axes, can be used to analyze $F(\boldsymbol{j})$ and $F(\boldsymbol{k})$, this is illustrated in Figure 3.16, where the *x*-coordinate is excluded for simplicity. One finds that $F(\boldsymbol{j}) = \langle 0, \cos(\theta), \sin(\theta) \rangle$ and $F(\boldsymbol{k}) = \langle 0, -\sin(\theta), \cos(\theta) \rangle$, and thus, the corresponding matrix is

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

Finally, let F denote the rotation around the *y*-axis. As before, the calculations use the same logic. However, the orientation of the axes illustrated in Figure 3.15 must be kept in mind. Drawing a diagram analogous to Figure 3.16, aligning the *x*-axis horizontally and the *z*-axis vertically (and excluding the *y*-coordinate), a counterclockwise rotation around the *y*-axis in three-dimensional space will appear as a clockwise rotation in the diagram; this is illustrated in Figure 3.17. One may then

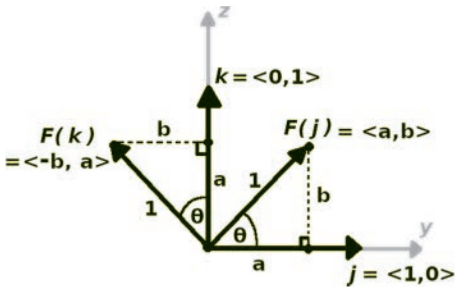

FIGURE 3.16 Calculating rotation around the *x*-axis.

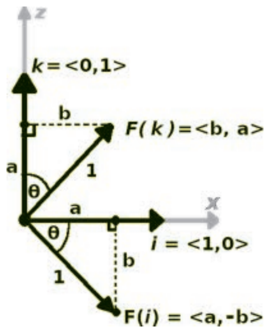

FIGURE 3.17 Calculating rotation around the *y*-axis.

calculate that $F(\boldsymbol{i}) = \langle \cos(\theta), 0, -\sin(\theta) \rangle$ and $F(\boldsymbol{k}) = \langle \sin(\theta), 0, \cos(\theta) \rangle$, and $F(\boldsymbol{j}) = \langle 0, 1, 0 \rangle$ since the *y*-axis is fixed. Therefore, the matrix for rotation around the *y*-axis is shown in Figure 3.17.

This completes the analysis of rotations in three-dimensional space; you now have formulas for generating a matrix corresponding to counterclockwise rotation by an angle θ around each of the axes. As a final note, observe that if the angle of rotation is $\theta = 0$, then since $\cos(0) = 1$ and $\sin(0) = 1$, each of the rotation matrix formulas yields an identity matrix.

3.2.3 Translation

A translation transformation adds constant values to each component of a vector. For two-dimensional vectors, this has the following form (where *m* and *n* are constants):

$$F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} x+m \\ y+n \end{bmatrix}$$

It can quickly be established that this transformation **cannot** be represented with a 2-by-2 matrix. For example, consider translation by ⟨2, 0⟩. If this could be represented as a matrix transformation, then it would be possible to solve the following equation for the constants *a*, *b*, *c*, and *d*:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \cdot x + b \cdot y \\ c \cdot x + d \cdot y \end{bmatrix} = \begin{bmatrix} x+2 \\ y \end{bmatrix}$$

Matching coefficients of *x* and *y* leads to $a = 1$, $c = 0$, $d = 1$, and the unavoidable expression $b = 2/y$, which is not a constant value for *b*. If the value $b = 2$ were chosen, the resulting matrix would not produce a translation—it would correspond to a *shear transformation*:

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + 2 \cdot y \\ y \end{bmatrix}$$

This particular shear transformation is illustrated in Figure 3.18, where the dot-shaded square on the left side is transformed into the dot-shaded parallelogram on the right side.

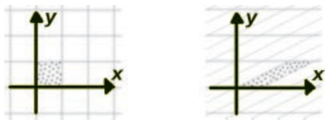

FIGURE 3.18 A shear transformation along the x direction.

Observe that, in Figure 3.18, the points along each horizontal line are being translated by a constant amount that depends on the y-coordinate of the points on the line; this is the defining characteristic of any shear transformation. In this particular example, the points along the line $y = 1$ are translated 2 units to the right, the points along $y = 2$ are translated 4 units to the right, and so forth; the points along $y = p$ are translated $2p$ units to the right.

The goal is to find a matrix that performs a constant translation on the *complete* set of points in a space; a shear transformation performs a constant translation on a *subset* of the points in a space. This observation is the key to finding the desired matrix and requires a new way of thinking about points. Consider a one-dimensional space, which would consist of only an x-axis. A translation on this space would consist of adding a constant number m to each x value. To realize this transformation as a matrix, consider a copy of the one-dimensional space embedded in two-dimensional space along the line $y = 1$; symbolically, identifying the one-dimensional point x with the two-dimensional point $(x, 1)$. Then, one-dimensional translation by m corresponds to the matrix calculation:

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ 1 \end{bmatrix}$$

Analogously, to perform a two-dimensional translation by $\langle m, n \rangle$, identify each point (x, y) with the point $(x, y, 1)$ and perform the following matrix calculation:

$$\begin{bmatrix} 1 & 0 & m \\ 0 & 1 & n \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ y+n \\ 1 \end{bmatrix}$$

Finally, to perform a three-dimensional translation by $\langle m, n, p \rangle$, identify each point (x, y, z) with the point $(x, y, z, 1)$ and perform the following matrix calculation:

$$\begin{bmatrix} 1 & 0 & 0 & m \\ 0 & 1 & 0 & n \\ 0 & 0 & 1 & p \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+m \\ y+n \\ z+p \\ 1 \end{bmatrix}$$

In other words, to represent translation as a matrix transformation, the space being translated is identified with a subset of a higher dimensional space with the additional coordinate set equal to 1. This is the reason that four-dimensional vectors and matrices are used in three-dimensional computer graphics. Even though there is no intuitive way to visualize four spatial dimensions, performing algebraic calculations on four-dimensional vectors is a straightforward process. This system of representing three-dimensional points with four-dimensional points (or representing n-dimensional points with $(n + 1)$-dimensional points in general) is called *homogeneous coordinates*. As previously mentioned, each point (x, y, z) is identified with $(x, y, z, 1)$; conversely, each four-dimensional point (x, y, z, w) is associated with the three-dimensional point $(x/w, y/w, z/w)$. This operation is called *perspective division* and aligns with the previous correspondence when $w=1$. There are additional uses for perspective division, which will be discussed further in Section 3.2.4.

It is also important to verify that the transformations previously discussed are compatible with the homogeneous coordinate system. In two dimensions, the transformation $F\left(\langle x,y\rangle\right) = \langle a{\cdot}x + b{\cdot}y, c{\cdot}x + d{\cdot}y\rangle$ becomes $F\left(\langle x, y, 1\rangle\right) = \langle a{\cdot}x + b{\cdot}y, c{\cdot}x + d{\cdot}y, 1\rangle$, which corresponds to the matrix multiplication:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a\cdot x+b\cdot y \\ c\cdot x+d\cdot y \\ 1 \end{bmatrix}$$

Therefore (when using homogeneous coordinates), all the geometric transformations of interest—translation, rotation, and scaling, collectively referred to as *affine transformations*—can be represented by multiplying by a matrix of the following form:

$$\begin{bmatrix} a_{11} & a_{12} & m_1 \\ a_{21} & a_{22} & m_2 \\ 0 & 0 & 1 \end{bmatrix}$$

where the 2-by-2 submatrix in the upper left represents the rotation and/or scaling part of the transformation (or is the identity matrix when there is no rotation or scaling), and the two-component vector $\langle m_1, m_2\rangle$ within the rightmost column represents the translation part of the transformation (or is the zero vector if there is no translation).

Similarly, in three dimensions (using homogeneous coordinates), affine transformations can be represented by multiplying by a matrix of the following form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & m_1 \\ a_{21} & a_{22} & a_{23} & m_2 \\ a_{31} & a_{32} & a_{33} & m_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the 3-by-3 submatrix in the upper left represents the rotation and/or scaling part of the transformation (or is the identity matrix if there is no rotation or scaling), and the three-component vector $\langle m_1, m_2, m_3\rangle$ within the rightmost column represents the translation part of the transformation (or is the zero vector if there is no translation).

3.2.4 Projections

In this section, the goal is to derive a formula for a perspective projection transformation. At the beginning of Chapter 1, and also in Section 1.2.2 on geometry processing, some of the core ideas were illustrated and informally introduced. To review, the viewable region in the scene needs to be mapped to the region rendered by OpenGL, a cube where the x, y, and z coordinates are all between −1 and +1, also referred to as *clip space*. In a perspective projection, the shape of the viewable region is a *frustum* or truncated pyramid. The pyramid is oriented so that it is "lying on its side": its central axis is aligned with the negative z-axis, as illustrated in Figure 3.19, and the viewer or virtual camera is positioned at the origin of the scene, which aligns with the point that was the tip of the original pyramid. The smaller rectangular end of the frustum is nearest to the origin, and the larger rectangular end is farthest from the origin. When the frustum is compressed into a cube, the larger end must be compressed more. This causes objects in the rendered image of the scene to appear smaller the farther they are from the viewer.

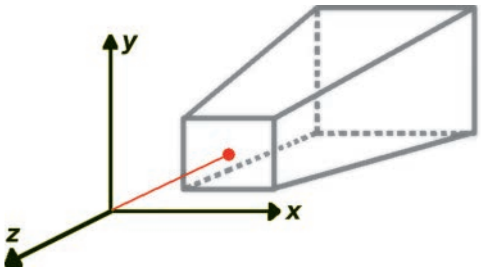

FIGURE 3.19 The frustum for a perspective transformation.

The shape of a frustum is defined by four parameters: the *near distance*, the *far distance*, the (vertical) *angle of view*, and the *aspect ratio*. The near and far distances are the most straightforward to explain: they refer to distances from the viewer (along the z-axis), and they set absolute bounds on what could potentially be seen by the viewer—any points outside of this range will not be rendered. However, not everything between these bounds will be visible. The *angle of view* is a measure of how much of the scene is visible to the viewer, and is defined as the angle between the top and bottom planes of the frustum (as oriented in Figure 3.19) if those planes were extended to the origin. Figure 3.20 shows two different frustums (shaded regions) as viewed from the side (along the x-axis). The figure also illustrates the fact that for fixed near and far distances, larger angles of view correspond to larger frustums.

In order for the dimensions of the visible part of the near plane to be proportional to the dimensions of the rendered image, the *aspect ratio* (defined as width divided by height) of the rendered image is the final

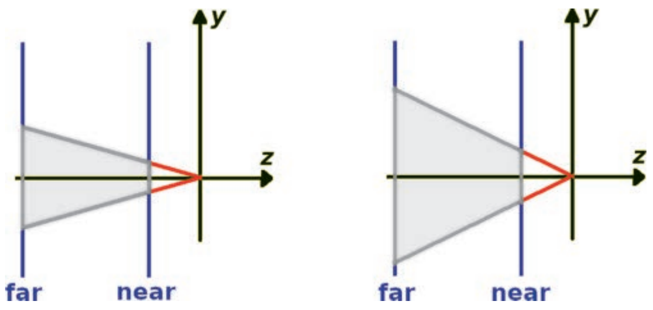

FIGURE 3.20 The effect of the angle of view on the size of a frustum.

value used to specify the shape of the frustum, illustrated in Figure 3.21, which depicts the frustum as viewed from the front (along the negative z-axis). In theory, a horizontal angle of view could be used to specify the size of the frustum instead of the aspect ratio, but in practice, determining the aspect ratio is simpler.

In a perspective projection, points in space (within the frustum) are mapped to points in the *projection window*: a flat rectangular region in space corresponding to the rendered image that will be displayed on the computer screen. The projection window corresponds to the smaller rectangular side of the frustum, the side nearest to the origin. To visualize how a point P is transformed in a perspective projection, draw a line from P to the origin, and the intersection of the line with the projection window is the result. Figure 3.22 illustrates the results of projecting three different points within the frustum onto the projection window.

To derive the formula for a perspective projection, the first step is to adjust the position of the projection window so that points in the frustum are projected to y-coordinates in the range from −1 to 1. Then, the formula for projecting y-coordinates will be derived, incorporating both matrix multiplication and the perspective division that occurs with homogeneous coordinates:

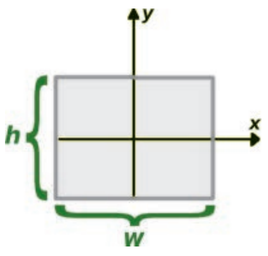

FIGURE 3.21 The aspect ratio $r = w/h$ of the frustum.

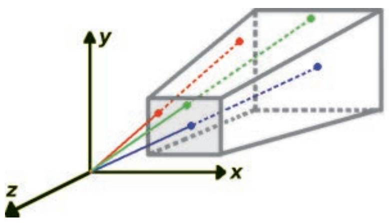

FIGURE 3.22 Projecting points from the frustum to the projection window.

converting (x, y, z, w) to $(x/w, y/w, z/w)$. Next, the formula for projecting x-coordinates will be derived, which is completely analogous to the formula for y-coordinates except that the aspect ratio needs to be taken into consideration. Finally, the z-coordinates of points in the frustum, which are bounded by the near distance and far distance, will be converted into the range from –1 to 1 and once again will require taking perspective division into account.

To begin, it will help to represent the parameters that define the shape of the frustum—the near distance, far distance, (vertical) angle of view, and aspect ratio—by n, f, a, and r, respectively. Consider adjusting the projection window so that the y-coordinates range from –1 to 1, while preserving the angle of view a, as illustrated on the left side of Figure 3.23 (viewed from the side, along the x-axis). This will change the distance d from the origin to the projection window. The value of d can be calculated using trigonometry on the corresponding right triangle, illustrated on the right side of Figure 3.23. By the definition of the tangent function, $\tan(a/2)=1/d$, from which it follows that $d=1/\tan(a/2)$. Therefore, all points on the projection window have their z coordinate equal to $-d$.

Next, ignoring x-coordinates for a moment, consider a point $P=(P_y, P_z)$ in the frustum that will be projected onto this new projection window. Drawing a line from P to the origin, let $Q=(Q_y, Q_z)$ be the intersection of the line with the projection window, as illustrated in Figure 3.24. Adding a perpendicular line from P to the z-axis, it becomes clear that we have two similar right triangles. Note that since the bases of the triangles are located on the negative z-axis, but lengths of sides are positive, the lengths of the sides are the negatives of the z-coordinates. The sides of similar triangles are proportional to each other, so we know that $P_y/(-P_z)=Q_y/(\;Q_z)$.

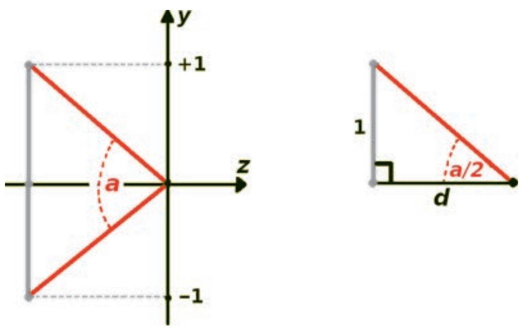

FIGURE 3.23 Adjusting the projection window.

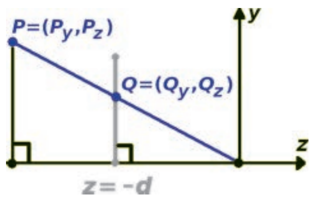

FIGURE 3.24 Calculating y-components of projected points.

Since Q is a point on the adjusted projection window, we also know that $Q_z = -d$. This allows us to write a formula for the y-coordinate of the projection: $Q_y = d \cdot P_y / (-P_z)$.

At first glance, this may appear to be incompatible with our matrix-based approach, as this formula is not a linear transformation, due to the division by the z coordinate. Fortunately, the fact that we are working in homogeneous coordinates (x, y, z, w) will enable us to resolve this problem. Since this point will be converted to (x/w, y/w, z/w) by perspective division (which is automatically performed by the GPU after the vertex shader is complete), the "trick" is to use a matrix to change the value of the w-component (which is typically equal to 1) to the value of the z-component (or more precisely, the negative of the z-component). This can be accomplished with the following matrix transformation (where $*$ indicates an as-yet unknown value):

$$\begin{bmatrix} * & * & * & * \\ 0 & d & 0 & 0 \\ * & * & * & * \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} * \\ d \cdot P_y \\ * \\ -P_z \end{bmatrix}$$

Then, after performing the division transformation, the homogeneous point $\left(*, d \cdot P_y, *, -P_z\right)$ is transformed into $\left(*, d \cdot P_y / (-P_z), *\right)$, as desired.

The next step is to derive a formula for the x-component of projected points; the calculations are similar to those for the y-component. The corresponding diagram is illustrated in Figure 3.25, viewed from along the y-axis. Again there is a pair of similar triangles, and therefore, the ratios of corresponding sides are equal, from which we obtain the equation $P_x / (-P_z) = Q_x / (-Q_z)$, and therefore, $Q_x = d \cdot P_x / (-Q_z)$.

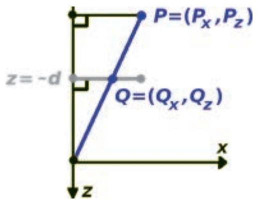

FIGURE 3.25 Calculating x-components of projected points.

However, one additional factor must be taken into account: the aspect ratio r. We have previously considered the y values to be in the range from –1 to 1; in accordance with the aspect ratio, the set of points that should be included in the rendered image have x values in the range from $-r$ to r. This range needs to be scaled into clip space, from –1 to 1, and therefore, the formula for the x coordinate must also be divided by r. This leads us to the formula $Q_x = (d/r)\cdot P_x/(-P_z)$, which can be accomplished with the following matrix transformation (again, $*$ indicating undetermined values):

$$\begin{bmatrix} d/r & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ * & * & * & * \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} (d/r)\cdot P_x \\ d\cdot P_y \\ * \\ -P_z \end{bmatrix}$$

The values in the third row of the matrix remain to be determined and will affect the z component of the point. The z value is used in depth calculations to determine which points will be visible, as specified by the near distance and far distance values. The values of the x and y components of the point are not needed for this calculation, and so the first two values in the row should be 0; refer to the remaining unknown values as b and c. Then, we have the following matrix transformation:

$$\begin{bmatrix} d/r & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & b & c \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} (d/r)\cdot P_x \\ d\cdot P_y \\ b\cdot P_z + c \\ -P_z \end{bmatrix}$$

After perspective division, the third coordinate $b \cdot P_z + c$ becomes $(b \cdot P_z + c)/(-P_z) = -b - c/P_z$. The values of the constants b and c will be determined soon, after two important points are clarified. First, the near distance n and the far distance f are typically given as positive values, even though the visible region frustum lies along the negative z-axis in world space, and thus, the nearest visible point P satisfies $P_z = -n$, while the farthest visible point P satisfies $P_z = -f$. Second, we know that we must convert the z coordinates of visible points into clip space coordinates (the range from −1 to 1), and it might seem as though the z coordinates of the nearest points to the viewer should be mapped to the value 1, as the positive z axis points directly at the viewer in our coordinate system. However, this is not the case! When OpenGL performs depth testing, it determines if one point is closer (to the viewer) than another by comparing their z coordinates. The type of comparison can be specified by the OpenGL function **`glDepthFunc`**, which has the default value GL_LESS, an OpenGL constant which indicates that a point should be considered closer to the viewer if its z coordinate is *less*. This means that in clip space, the *negative* z axis points directly at the viewer; this space uses a *left-handed* coordinate system. Combining these two points, we now know that if $P_z = -n$, then $-b - c/P_z$ should equal −1, and if $P_z = -f$, then $-b - c/P_z$ should equal 1. This corresponds to the following system of equations:

$$-b + \frac{c}{n} = -1$$

$$-b + \frac{c}{f} = 1$$

There are a variety of approaches to solve this system; one of the simplest is to multiply the first equation by −1 and then add it to the second equation. This eliminates b; solving for c yields $c = 2 \cdot n \cdot f / (n - f)$. Substituting this value for c into the first equation and solving for b yield $b = (n + f)/(n - f)$.

With this calculation finished, the matrix is completely determined. To summarize, the perspective projection transformation for a frustum-shaped region defined by near distance n, far distance f, (vertical) angle of view a, and aspect ratio r, when working with homogeneous coordinates, can be achieved with the following matrix:

$$\begin{bmatrix} \dfrac{1}{r \cdot \tan(a/2)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan(a/2)} & 0 & 0 \\ 0 & 0 & \dfrac{n+f}{n-f} & \dfrac{2 \cdot n \cdot f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

3.2.5 Local Transformations

At this point, you are able to produce the matrices corresponding to translation, rotation, and scaling transformations. For example, consider an object in two-dimensional space, such as the turtle on the left side of Figure 3.26, whose shape is defined by some set of points S. Let $\boldsymbol{T}$ be the matrix corresponding to translation by $\langle 1, 0 \rangle$, and let $\boldsymbol{R}$ be the matrix corresponding to rotation around the origin by 45°. To move the turtle around you could, for example, multiply all the points in the set S by $\boldsymbol{T}$, and then by $\boldsymbol{R}$, and then by $\boldsymbol{T}$ again, which would result in the sequence of images illustrated in the remaining parts of Figure 3.26. All these transformations are relative to an external, fixed coordinate system called *world coordinates* or *global coordinates*, and this aspect is emphasized by the more specific term *global transformations*.

The internal or local coordinate system used to define the vertices of a geometric object is somewhat arbitrary—the origin point and the orientation and scale of the local coordinate axes are typically chosen for convenience, without reference to the global coordinate system. For example, the local origin is frequently chosen to correspond to a location on the object that is in some sense the "center" of the object. Locations specified

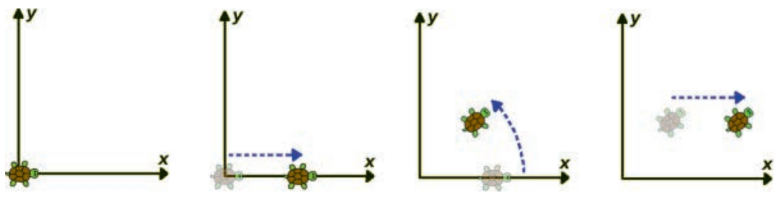

FIGURE 3.26 A sequence of global transformations.

relative to this coordinate system are called *object coordinates* or *local coordinates*. After an object is added to a three-dimensional scene, the object can then be repositioned, oriented, and resized as needed using geometric transformations.

Of particular interest in this section are *local transformations*: transformations relative to the local coordinate system of an object, and how they may be implemented with matrix multiplication. Initially, the local coordinate axes of an object are aligned with the global coordinate axes. As an object is transformed, its local coordinate axes undergo the same transformations. Figure 3.27 illustrates multiple copies of the turtle object together with their local coordinate axes after various transformations have been applied.

Figure 3.28 illustrates several examples of local transformations. Assuming the turtle starts at the state with position and orientation shown

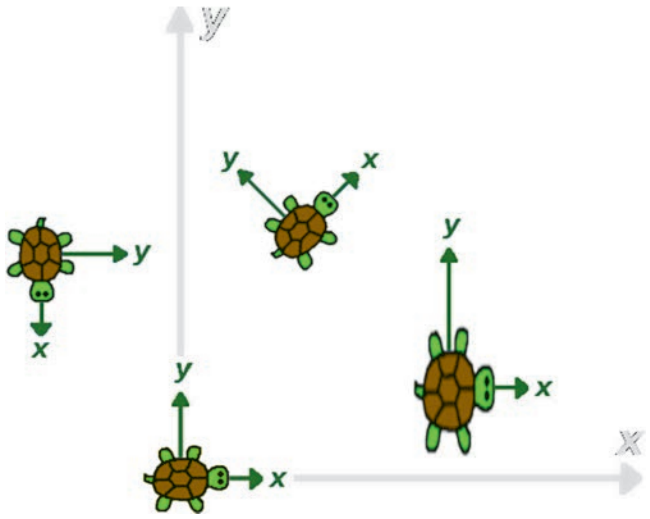

FIGURE 3.27 Transforming local coordinate axes.

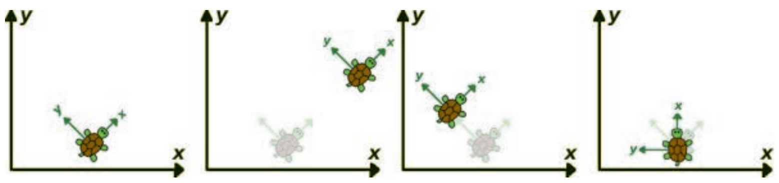

FIGURE 3.28 Various local transformations.

in the leftmost image, the remaining images show local translation by ⟨2, 0⟩, local translation by ⟨0, 1⟩, and local rotation by 45°, each applied to the starting state. Observe in particular that a local rotation takes place around the center of an object (the origin of its local coordinate system), rather than around the world or global origin.

The concepts of local and global transformation are reflected in everyday language. For example, walking forwards, backwards, left, or right, are examples of local translations that depend on the current orientation of a person's coordinate system: in other words, it matters which way the person is currently facing. If two people are facing in different directions, and they are both asked to step forward, they will move in different directions. If a global translation is specified, which is typically done by referencing the compass directions (north, south, east, and west), then you can be assured that people will walk in the same direction, regardless of which way they may have been facing at first.

The question remains: how can matrix multiplication be used to perform local transformations (assuming that it is possible)? Before addressing this question, it will help to introduce a new concept. When transforming a set of points with a matrix, the points are multiplied by the matrix in the vertex shader and the new coordinates are passed along to the fragment shader, but the new coordinates of the points are not permanently stored. Instead, the accumulated transformations that have been applied to an object are stored as the product of the corresponding matrices, which is a single matrix called the *model matrix* of the object. The model matrix effectively stores the current location, orientation, and scale of an object (although it is slightly complicated to extract some of the information from the entries of the matrix).

Given an object whose shape is defined by a set of points S, assume that a sequence of transformations have been applied and let $\boldsymbol{M}$ denote the current model matrix of the object. Thus, the current location of the points of the object can be calculated by $\boldsymbol{M}\cdot P$, where P ranges over the points in the set S. Let $\boldsymbol{T}$ be the matrix corresponding to a transformation. If you were to apply this matrix as a global transformation, as described in previous sections, the new model matrix would be $\boldsymbol{T}\cdot\boldsymbol{M}$, since each new matrix that is applied becomes the leftmost element in the product (just as functions are ordered in function composition). In order for the matrix $\boldsymbol{T}$ to have the effect of a local transformation on an object, the local coordinate axes of the object would have to be aligned with the global coordinate axes, which suggests the following three-step strategy:

1. Align the two sets of axes by applying $\boldsymbol{M}^{-1}$, the inverse of the model matrix.
2. Apply $\boldsymbol{T}$, since local and global transformations are the same when the axes are aligned.
3. Apply $\boldsymbol{M}$, the original model matrix, which returns the object to its previous state while taking the transformation $\boldsymbol{T}$ into account. (In a sense, it is as if the matrix $\boldsymbol{M}$ has been applied to transformation $\boldsymbol{T}$, converting it from a global transformation to a local transformation.)

This sequence of transformations is illustrated in Figure 3.29, where the images show an object with model matrix $\boldsymbol{M}$ (in this example, translation and then rotation by 45° was applied), the result of applying $\boldsymbol{M}^{-1}$ (reversing the rotation and then reversing the translation), the result of applying $\boldsymbol{T}$ (in this example, translation), and the result of applying $\boldsymbol{M}$ again (translating and rotating again). The last image also shows the outline of the object in its original placement for comparison.

At the end of this process, combining all these transformations, the model matrix has become $\boldsymbol{M} \cdot \boldsymbol{T} \cdot \boldsymbol{M}^{-1} \cdot \boldsymbol{M}$ (recall that matrices are applied to a point from the right to the left). Since $\boldsymbol{M}^{-1} \cdot \boldsymbol{M} = \boldsymbol{I}$ (the identity matrix), this expression simplifies to $\boldsymbol{M} \cdot \boldsymbol{T}$: the original model matrix $\boldsymbol{M}$ multiplied on the *right* by $\boldsymbol{T}$. This is the answer to the question of interest. To summarize, given an object with model matrix $\boldsymbol{M}$ and a transformation with matrix $\boldsymbol{T}$:

- To apply $\boldsymbol{T}$ as a global transformation, let the new model matrix equal $\boldsymbol{T} \cdot \boldsymbol{M}$.
- To apply $\boldsymbol{T}$ as a local transformation, let the new model matrix equal $\boldsymbol{M} \cdot \boldsymbol{T}$.

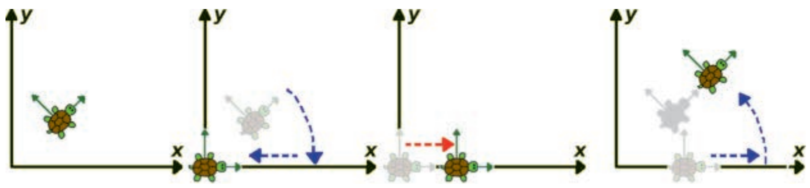

FIGURE 3.29 A sequence of global transformations equivalent to a local translation.

As you will see in later sections, being able to use local transformations will be useful when creating interactive three-dimensional scenes. For example, navigating within a scene feels more intuitive and immersive if one is able to move the virtual camera with local transformations. The viewer might not always be aware of the direction of the *z*-axis, which could lead to unexpected motions when moving in global directions, but the viewer is always able to see what is in front of them, which makes moving forward more natural.

3.3 A MATRIX CLASS

Now that you have learned how to derive the various types of matrices that will be needed, the next step will be to create a **`Matrix`** class containing static methods to generate matrices (with the numpy library) corresponding to each of the previously discussed geometric transformations: identity, translation, rotation (around each axis), scaling, and projection. To proceed, in the **`core`** folder, create a new file named **`matrix.py`** containing the following code:

```
import numpy
from math import sin, cos, tan, pi

class Matrix(object):

    @staticmethod
    def makeIdentity():
        return numpy.array( [[1, 0, 0, 0],
                              [0, 1, 0, 0],
                              [0, 0, 1, 0],
                              [0, 0, 0, 1]] ).
                             astype(float)

    @staticmethod
    def makeTranslation(x, y, z):
        return numpy.array([[1, 0, 0, x],
                             [0, 1, 0, y],
                             [0, 0, 1, z],
                             [0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationX(angle):
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[1, 0,  0, 0],
                            [0, c, -s, 0],
                            [0, s,  c, 0],
                            [0, 0,  0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationY(angle):
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[ c, 0, s, 0],
                            [ 0, 1, 0, 0],
                            [-s, 0, c, 0],
                            [ 0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeRotationZ(angle):
        c = cos(angle)
        s = sin(angle)
        return numpy.array([[c, -s, 0, 0],
                            [s,  c, 0, 0],
                            [0,  0, 1, 0],
                            [0,  0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makeScale(s):
        return numpy.array([[s, 0, 0, 0],
                            [0, s, 0, 0],
                            [0, 0, s, 0],
                            [0, 0, 0, 1]]).
                            astype(float)

    @staticmethod
    def makePerspective(angleOfView=60,
aspectRatio=1, near=0.1, far=1000):
        a = angleOfView * pi/180.0
        d = 1.0 / tan(a/2)
        r = aspectRatio
        b = (far + near) / (near - far)
        c = 2*far*near / (near - far)
        return numpy.array([[d/r, 0,  0, 0],
                            [0,   d,  0, 0],
                            [0,   0,  b, c],
                            [0,   0, -1, 0]]).
                            astype(float)
```

With this class completed, you are nearly ready to incorporate matrix-based transformations into your scenes.

3.4 INCORPORATING WITH GRAPHICS PROGRAMS

Before creating the main application, the **`Uniform`** class needs to be updated to be able to store the matrices generated by the **`Matrix`** class. In GLSL, 4-by-4 matrices correspond to the data type **`mat4`**, and the corresponding uniform data can be sent to the GPU with the following OpenGL command:

glUniformMatrix4fv(*variableRef*, *matrixCount*, *transpose*, *value*)

Specify the value of the uniform variable referenced by the parameter *variableRef* in the currently bound program. The number of matrices is specified by the parameter *matrixCount*. The matrix data is stored as an array of vectors in the parameter *value*. OpenGL expects matrix data to be stored as an array of column vectors; if this is not the case (if data is stored as an array of row vectors), then the boolean parameter *transpose* should be set to the OpenGL constant GL_TRUE (which causes the data to be re-interpreted as rows) and GL_FALSE otherwise.

In the file **`uniform.py`** located in the **`core`** folder, add the following else-if condition at the end of the block of **`elif`** statements in the **`uploadData`** function:

```
elif self.dataType == "mat4":
    glUniformMatrix4fv(self.variableRef, 1, GL_TRUE,
      self.data)
```

Since you will now be creating three-dimensional scenes, you will activate a render setting that performs depth testing (in case you later choose to add objects which may obscure other objects). This (and other render settings) can be configured by using the following two OpenGL functions:

glEnable(*setting*)

Enables an OpenGL capability specified by an OpenGL constant specified by the parameter *setting*. For example, possible constants include GL_DEPTH_TEST to activate depth testing, GL_POINT_SMOOTH to draw rounded points instead of square points, or GL_BLEND to enable blending colors in the color buffer based on alpha values.

glDisable(*setting*)

Disables an OpenGL capability specified by an OpenGL constant specified by the parameter *setting*.

Finally, for completeness, we include the OpenGL function that allows you to configure depth testing, previously mentioned in Section 3.2.4 on perspective projection. However, you will not change the function from its default setting, and this function will not be used in what follows.

glDepthFunc(compareFunction)

Specify the function used to compare each pixel depth with the depth value present in the depth buffer. If a pixel passes the comparison test, it is considered to be currently the closest to the viewer, and its values overwrite the current values in the color and depth buffers. The function used is specified by the OpenGL constant *compareFunction*, some of whose possible values include the default setting GL_LESS (indicating that a pixel is closer to the viewer if the depth value is less) and GL_GREATER (indicating that a pixel is closer if the depth value is greater). If depth testing has not been enabled, the depth test always passes, and pixels are rendered on top of each other according to the order in which they are processed.

Now you are ready to create an interactive scene. The first new component will be the vertex shader code: there will be two uniform **`mat4`** variables. One will store the model transformation matrix, which will be used to translate and rotate a geometric object. The other will store the perspective transformation matrix, which will make objects appear smaller as they move further away. To begin creating this scene, in your main directory, create a file named **`test-3.py`**, containing the following code (the **`import`** statements will be needed later on):

```
from core.base import Base
from core.openGLUtils import OpenGLUtils
from core.attribute import Attribute
from core.uniform import Uniform
from core.matrix import Matrix
from OpenGL.GL import *
from math import pi


# move a triangle around the screen
class Test(Base):

    def initialize(self):
        print("Initializing program...")


        ### initialize program ###
        vsCode = """
        in vec3 position;
        uniform mat4 projectionMatrix;
        uniform mat4 modelMatrix;
        void main()
        {
            gl_Position = projectionMatrix *
                  modelMatrix * vec4(position, 1.0);
        }
        """


        fsCode = """
        out vec4 fragColor;
        void main()
        {
            fragColor = vec4(1.0, 1.0, 0.0, 1.0);
        }
        """


        self.programRef = OpenGLUtils.initializeProgram(
            vsCode, fsCode)
```

Next, you will initialize attribute data for the vertices of an isosceles triangle, uniforms for the model and projection matrices, and variables to store the movement and rotation speed that will be applied to the triangle in the **update** function. To continue, return to the file named **test-3.py**, and add the following code to the **initialize** function:

```
### render settings ###
glClearColor(0.0, 0.0, 0.0, 1.0)
glEnable(GL_DEPTH_TEST)

### set up vertex array object ###
vaoRef = glGenVertexArrays(1)
glBindVertexArray(vaoRef)

### set up vertex attribute ###
positionData = [ [0.0, 0.2, 0.0], [0.1, -0.2, 0.0],
   [-0.1, -0.2, 0.0] ]
self.vertexCount = len(positionData)
positionAttribute = Attribute("vec3", positionData)
positionAttribute.associateVariable( self.programRef,
   "position" )

### set up uniforms ###
mMatrix = Matrix.makeTranslation(0, 0, -1)
self.modelMatrix = Uniform("mat4", mMatrix)
self.modelMatrix.locateVariable( self.programRef,
   "modelMatrix" )

pMatrix = Matrix.makePerspective()
self.projectionMatrix = Uniform("mat4", pMatrix)
self.projectionMatrix.locateVariable( self.programRef,
   "projectionMatrix" )

# movement speed, units per second
self.moveSpeed = 0.5
# rotation speed, radians per second
self.turnSpeed = 90 * (pi / 180)
```

Next, you will turn your attention to creating an **update** function. The first step will be to calculate the actual amounts of movement that may be applied, based on the previously set base speed and the time elapsed since the last frame (which is stored in the **Base** class variable **deltaTime**). In the file **test-3.py**, add the following code:

```
def update(self):
    # update data
    moveAmount = self.moveSpeed * self.deltaTime
    turnAmount = self.turnSpeed * self.deltaTime
```

To illustrate the versatility of using matrices to transform objects, both global and local movement will be implemented. Next, there will be a large number of conditional statements, each of which follow the same pattern: check if a particular key is being pressed, and if so, create the corresponding matrix **m** and multiply the model matrix by **m** in the correct order (**m** on the left for global transformations and **m** on the right for local transformations). Note that while shaders use the operator '*' for matrix multiplication, numpy uses the operator '**@**' for matrix multiplication. For global translations, you will use the keys W/A/S/D for the up/left/down/right directions and the keys Z/X for the forward/backward directions. In the file **test-3.py**, in the **update** function, add the following code:

```
# global translation
if self.input.isKeyPressed("w"):
    m = Matrix.makeTranslation(0, moveAmount, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("s"):
    m = Matrix.makeTranslation(0, -moveAmount, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("a"):
    m = Matrix.makeTranslation(-moveAmount, 0, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("d"):
    m = Matrix.makeTranslation(moveAmount, 0, 0)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("z"):
    m = Matrix.makeTranslation(0, 0, moveAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("x"):
    m = Matrix.makeTranslation(0, 0, -moveAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
```

Rotation that makes the object appear to rotate left and right from the viewer's perspective is really a rotation around the *z*-axis in three-dimensional space. Since the keys A/D move the object left/right, you will use the keys Q/E to rotate the object left/right, as they lay in the row directly above A/D. Since these keys will be used for a global rotation, they will cause the triangle to rotate around the origin (0,0,0) of the three-dimensional world. Continuing on in the file **`test-3.py`**, in the **`update`** function, add the following code:

```
# global rotation (around the origin)
if self.input.isKeyPressed("q"):
    m = Matrix.makeRotationZ(turnAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
if self.input.isKeyPressed("e"):
    m = Matrix.makeRotationZ(-turnAmount)
    self.modelMatrix.data = m @ self.modelMatrix.data
```

Next, to incorporate local translation, you will use the keys I/J/K/L for the directions up/left/down/right, as they are arranged in a similar layout to the W/A/S/D keys. Continue by adding the following code to the **`update`** function:

```
# local translation
if self.input.isKeyPressed("i"):
    m = Matrix.makeTranslation(0, moveAmount, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("k"):
    m = Matrix.makeTranslation(0, -moveAmount, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("j"):
    m = Matrix.makeTranslation(-moveAmount, 0, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("l"):
    m = Matrix.makeTranslation(moveAmount, 0, 0)
    self.modelMatrix.data = self.modelMatrix.data @ m
```

You will use the keys U/O for local rotation left/right, as they are in the row above the keys used local movement left/right (J/L), analogous to the key layout for global transformations. Since these keys will refer to a local rotation, they will rotate the triangle around its center (where the world origin was located when the triangle was in its original position).

```
# local rotation (around object center)
if self.input.isKeyPressed("u"):
    m = Matrix.makeRotationZ(turnAmount)
    self.modelMatrix.data = self.modelMatrix.data @ m
if self.input.isKeyPressed("o"):
    m = Matrix.makeRotationZ(-turnAmount)
    self.modelMatrix.data = self.modelMatrix.data @ m
```

After processing user input, the buffers need to be cleared before the image is rendered. In addition to clearing the color buffer, since depth testing is now being performed, the depth buffer should also be cleared. Uniform values need to be stored in their corresponding variables, and the **glDrawArrays** function needs to be called to render the triangle. To accomplish these tasks, at the end of the **update** function, add the following code:

```
### render scene ###
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
glUseProgram( self.programRef )
self.projectionMatrix.uploadData()
self.modelMatrix.uploadData()
glDrawArrays( GL_TRIANGLES , 0 , self.vertexCount )
```

Finally, to run this application, the last lines of code need to instantiate the **Test** class and call the **run** function. At the end of the **test-3.py** file, add the following code (with no indentation, as it is not part of the class or the **update** function):

```
# instantiate this class and run the program
Test().run()
```

At this point, the application class is complete! Run the file and you should see an isosceles triangle in the center of your screen. Press the keyboard keys as described previously to experience the difference between global and local transformations of an object. While the object is located at the origin, local and global rotations will appear identical, but when the object is located far away from the origin, the difference in rotation is more easily seen. Similarly, while the object is in its original orientation (its local right direction aligned with the positive *x*-axis), local and global translations will appear identical, but after a rotation, the difference in the translations becomes apparent.

3.5 SUMMARY AND NEXT STEPS

In this chapter, you learned about the mathematical foundations involved in geometric transformations. You learned about vectors and the operations of vector addition and scalar multiplication, and how any vector can be written as a combination of standard basis vectors using these operations. Then, you learned about a particular type of vector function called a linear transformation, which naturally led to the definition of a matrix and matrix multiplication. Then, you learned how to create matrices representing the different types of geometric transformations (scaling, rotation, translation, and perspective projection) used in creating animated, interactive three-dimensional graphics applications. You also learned how a model matrix stores the accumulated transformations that have been applied to an object, and how this structure enables you to use matrices for both global and local transformations. Finally, all of this was incorporated into the framework being developed in this book.

In the next chapter, you will turn your attention to automating many of these steps as you begin to create the graphics framework classes in earnest.