2.4 Vectors

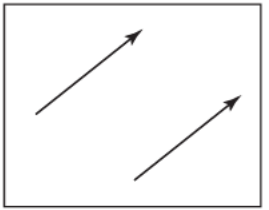

Figure 2.11. These two vectors are the same because they have the same length and direction.

A *vector* describes a length and a direction. It can be usefully represented by an arrow. Two vectors are equal if they have the same length and direction even if we think of them as being located in different places (Figure 2.11). As much as possible, you should think of a vector as an arrow and not as coordinates or numbers. At some point, we will have to represent vectors as numbers in our programs, but even in code, they should be manipulated as objects and only the low-level vector operations should know about their numeric representation (DeRose, 1989). Vectors will be represented as bold characters, e.g., **a**. A vector's length is denoted

$\|\mathbf{a}\|$. A *unit vector* is any vector whose length is one. The *zero vector* is the vector of zero length. The direction of the zero vector is undefined.

Vectors can be used to represent many different things. For example, they can be used to store an *offset*, also called a *displacement*. If we know "the treasure is buried two paces east and three paces north of the secret meeting place," then we know the offset, but we don't know where to start. Vectors can also be used to store a *location*, another word for *position* or *point*. Locations can be represented as a displacement from another location. Usually, there is some understood *origin* location from which all other locations are stored as offsets. Note that locations are not vectors. As we shall discuss, you can add two vectors. However, it usually does not make sense to add two locations unless it is an intermediate operation when computing weighted averages of a location (Goldman, 1985). Adding two offsets does make sense, so that is one reason why offsets are vectors. But this emphasizes that a location is not an offset; it is an offset from a specific origin location. The offset by itself is not the location.

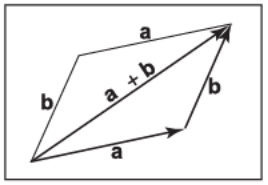

Figure 2.12. Two vectors are added by arranging them head to tail. This can be done in either order.

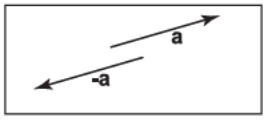

Figure 2.13. The vector –a has the same length but opposite direction of the vector a.

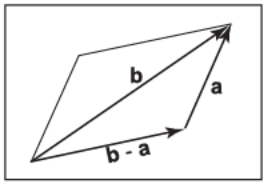

Figure 2.14. Vector subtraction is just vector addition with a reversal of the second argument.

2.4.1 Vector Operations

Vectors have most of the usual arithmetic operations that we associate with real numbers. Two vectors are equal if and only if they have the same length and direction. Two vectors are added according to the *parallelogram rule*. This rule states that the sum of two vectors is found by placing the tail of either vector against the head of the other (Figure 2.12). The sum vector is the vector that "completes the triangle" started by the two vectors. The parallelogram is formed by taking the sum in either order. This emphasizes that vector addition is commutative:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}.$$

Note that the parallelogram rule just formalizes our intuition about displacements. Think of walking along one vector, tail to head, and then walking along the other. The net displacement is just the parallelogram diagonal. You can also create a *unary minus* for a vector: $-\mathbf{a}$ (Figure 2.13) is a vector with the same length as a but opposite direction. This allows us to also define subtraction:

$$\mathbf{b} - \mathbf{a} \equiv -\mathbf{a} + \mathbf{b}.$$

You can visualize vector subtraction with a parallelogram (Figure 2.14). We can write

$$\mathbf{a} + (\mathbf{b} - \mathbf{a}) = \mathbf{b}.$$

Vectors can also be multiplied. In fact, there are several kinds of products involving vectors. First, we can *scale* the vector by multiplying it by a real number k.

This just multiplies the vector's length without changing its direction. For example, 3.5a is a vector in the same direction as a, but it is 3.5 times as long as a. We discuss two products involving two vectors, the dot product and the cross product, later in this section, and a product involving three vectors, the determinant, in Chapter 6.

2.4.2 Cartesian Coordinates of a Vector

A 2D vector can be written as a combination of any two nonzero vectors which are not parallel. This property of the two vectors is called *linear independence*. Two linearly independent vectors form a 2D *basis*, and the vectors are thus referred to as *basis vectors*. For example, a vector c may be expressed as a combination of two basis vectors a and b (Figure 2.15):

$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}. \tag{2.3}$$

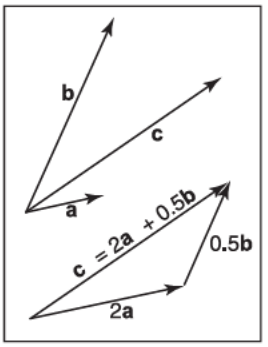

Figure 2.15. Any 2D vector c is a weighted sum of any two nonparallel 2D vectors a and b.

Note that the weights a_c and b_c are unique. Bases are especially useful if the two vectors are *orthogonal*; i.e., they are at right angles to each other. It is even more useful if they are also unit vectors in which case they are *orthonormal*. If we assume two such "special" vectors x and y are known to us, then we can use them to represent all other vectors in a *Cartesian* coordinate system, where each vector is represented as two real numbers. For example, a vector a might be represented as

$$\mathbf{a} = x_a\mathbf{x} + y_a\mathbf{y},$$

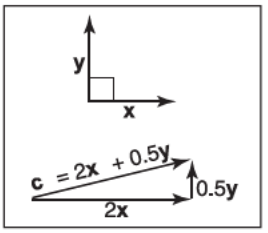

Figure 2.16. A 2D Cartesian basis for vectors.

where x_a and y_a are the real Cartesian coordinates of the 2D vector a (Figure 2.16). Note that this is not really any different conceptually from Equation (2.3), where the basis vectors were not orthonormal. But there are several advantages to a Cartesian coordinate system. For instance, by the Pythagorean theorem, the length of a is

$$\|\mathbf{a}\| = \sqrt{x_a^2 + y_a^2}.$$

It is also simple to compute dot products, cross products, and coordinates of vectors in Cartesian systems, as we'll see in the following sections.

By convention, we write the coordinates of a either as an ordered pair (x_a, y_a) or a column matrix:

$$\mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}.$$

The form we use will depend on typographic convenience. We will also occasionally write the vector as a row matrix, which we will indicate as $\mathbf{a}^{\mathrm{T}}$:

$$\mathbf{a}^{\mathrm{T}} = \begin{bmatrix} x_a & y_a \end{bmatrix}.$$

We can also represent 3D, 4D, etc., vectors in Cartesian coordinates. For the 3D case, we use a basis vector $\mathbf{z}$ that is orthogonal to both $\mathbf{x}$ and $\mathbf{y}$.

2.4.3 Dot Product

The simplest way to multiply two vectors is the *dot* product. The dot product of $\mathbf{a}$ and $\mathbf{b}$ is denoted $\mathbf{a} \cdot \mathbf{b}$ and is often called the *scalar product* because it returns a scalar. The dot product returns a value related to its arguments' lengths and the angle ϕ between them (Figure 2.17):

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \, \cos\phi, \tag{2.4}$$

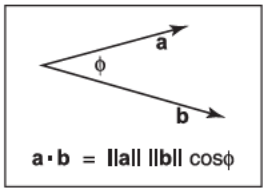

Figure 2.17. The dot product is related to length and angle and is one of the most important formulas in graphics.

The most common use of the dot product in graphics programs is to compute the cosine of the angle between two vectors.

The dot product can also be used to find the *projection* of one vector onto another. This is the length $\mathbf{a}{\rightarrow}\mathbf{b}$ of a vector $\mathbf{a}$ that is projected at right angles onto a vector $\mathbf{b}$ (Figure 2.18):

$$\mathbf{a}{\rightarrow}\mathbf{b} = \|\mathbf{a}\| \, \cos\phi = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}. \tag{2.5}$$

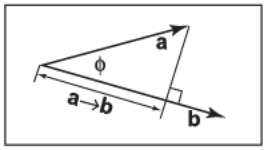

Figure 2.18. The projection of $\mathbf{a}$ onto $\mathbf{b}$ is a length found by Equation (2.5).

The dot product obeys the familiar associative and distributive properties we have in real arithmetic:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \mathbf{b} \cdot \mathbf{a}, \\ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}, \\ (k\mathbf{a}) \cdot \mathbf{b} &= \mathbf{a} \cdot (k\mathbf{b}) = k\mathbf{a} \cdot \mathbf{b}. \end{aligned} \tag{2.6}$$

If 2D vectors $\mathbf{a}$ and $\mathbf{b}$ are expressed in Cartesian coordinates, we can take advantage of $\mathbf{x} \cdot \mathbf{x} = \mathbf{y} \cdot \mathbf{y} = 1$ and $\mathbf{x} \cdot \mathbf{y} = 0$ to derive that their dot product is

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= (x_a\mathbf{x} + y_a\mathbf{y}) \cdot (x_b\mathbf{x} + y_b\mathbf{y}) \\ &= x_a x_b(\mathbf{x} \cdot \mathbf{x}) + x_a y_b(\mathbf{x} \cdot \mathbf{y}) + x_b y_a(\mathbf{y} \cdot \mathbf{x}) + y_a y_b(\mathbf{y} \cdot \mathbf{y}) \\ &= x_a x_b + y_a y_b. \end{aligned}$$

Similarly in 3D we can find

$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b + z_a z_b.$$

2.4.4 Cross Product

The cross product $\mathbf{a} \times \mathbf{b}$ is usually used only for three-dimensional vectors; generalized cross products are discussed in references given in the chapter notes. The cross product returns a 3D vector that is perpendicular to the two arguments of the cross product. The length of the resulting vector is related to $\sin\phi$:

$$\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin\phi.$$

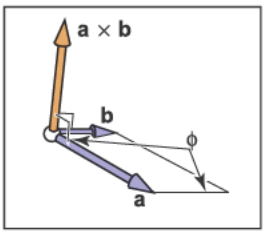

Figure 2.19. The cross product $\mathbf{a} \times \mathbf{b}$ is a 3D vector perpendicular to both 3D vectors $\mathbf{a}$ and $\mathbf{b}$, and its length is equal to the area of the parallelogram shown.

The magnitude $\|\mathbf{a} \times \mathbf{b}\|$ is equal to the area of the parallelogram formed by vectors $\mathbf{a}$ and $\mathbf{b}$. In addition, $\mathbf{a} \times \mathbf{b}$ is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$ (Figure 2.19). Note that there are only two possible directions for such a vector. By definition, the vectors in the direction of the x-, y- and z-axes are given by

$$\begin{aligned} \mathbf{x} &= (1,0,0), \\ \mathbf{y} &= (0,1,0), \\ \mathbf{z} &= (0,0,1), \end{aligned}$$

and we set as a convention that $\mathbf{x} \times \mathbf{y}$ must be in the plus or minus $\mathbf{z}$ direction. The choice is somewhat arbitrary, but it is standard to assume that

$$\mathbf{z} = \mathbf{x} \times \mathbf{y}.$$

All possible permutations of the three Cartesian unit vectors are

$$\begin{aligned} \mathbf{x} \times \mathbf{y} &= +\mathbf{z}, \\ \mathbf{y} \times \mathbf{x} &= -\mathbf{z}, \\ \mathbf{y} \times \mathbf{z} &= +\mathbf{x}, \\ \mathbf{z} \times \mathbf{y} &= -\mathbf{x}, \\ \mathbf{z} \times \mathbf{x} &= +\mathbf{y}, \\ \mathbf{x} \times \mathbf{z} &= -\mathbf{y}. \end{aligned}$$

Because of the $\sin\phi$ property, we also know that a vector cross itself is the zero vector, so $\mathbf{x} \times \mathbf{x} = \mathbf{0}$ and so on. Note that the cross product is *not* commutative, i.e., $\mathbf{x} \times \mathbf{y} \neq \mathbf{y} \times \mathbf{x}$. The careful observer will note that the above discussion does not allow us to draw an unambiguous picture of how the Cartesian axes relate. More specifically, if we put $\mathbf{x}$ and $\mathbf{y}$ on a sidewalk, with $\mathbf{x}$ pointing east and $\mathbf{y}$ pointing north, then does $\mathbf{z}$ point up to the sky or into the ground? The usual convention is to have $\mathbf{z}$ point to the sky. This is known as a *right-handed* coordinate system. This name comes from the memory scheme of "grabbing" $\mathbf{x}$

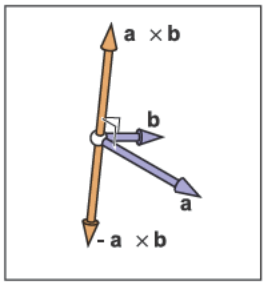

Figure 2.20. The "right-hand rule" for cross products. Imagine placing the base of your right palm where **a** and **b** join at their tails, and pushing the arrow of **a** toward **b**. Your extended right thumb should point toward $\mathbf{a} \times \mathbf{b}$.

with your *right* palm and fingers and rotating it toward $\mathbf{y}$. The vector $\mathbf{z}$ should align with your thumb. This is illustrated in Figure 2.20.

The cross product has the nice property that

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c},$$

and

$$\mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b}).$$

However, a consequence of the right-hand rule is

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

In Cartesian coordinates, we can use an explicit expansion to compute the cross product:

$$\begin{aligned}
\mathbf{a} \times \mathbf{b} &= (x_a\mathbf{x} + y_a\mathbf{y} + z_a\mathbf{z}) \times (x_b\mathbf{x} + y_b\mathbf{y} + z_b\mathbf{z}) \\
&= x_a x_b \mathbf{x} \times \mathbf{x} + x_a y_b \mathbf{x} \times \mathbf{y} + x_a z_b \mathbf{x} \times \mathbf{z} \\
&\quad + y_a x_b \mathbf{y} \times \mathbf{x} + y_a y_b \mathbf{y} \times \mathbf{y} + y_a z_b \mathbf{y} \times \mathbf{z} \\
&\quad + z_a x_b \mathbf{z} \times \mathbf{x} + z_a y_b \mathbf{z} \times \mathbf{y} + z_a z_b \mathbf{z} \times \mathbf{z} \\
&= (y_a z_b - z_a y_b)\mathbf{x} + (z_a x_b - x_a z_b)\mathbf{y} + (x_a y_b - y_a x_b)\mathbf{z}. \qquad (2.7)
\end{aligned}$$

So, in coordinate form,

$$\mathbf{a} \times \mathbf{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b). \qquad (2.8)$$

2.4.5 Orthonormal Bases and Coordinate Frames

Managing coordinate systems is one of the core tasks of almost any graphics program; the key to this is managing *orthonormal bases*. Any set of two 2D vectors $\mathbf{u}$ and $\mathbf{v}$ form an orthonormal basis provided that they are orthogonal (at right angles) and are each of unit length. Thus,

$$\|\mathbf{u}\| = \|\mathbf{v}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = 0.$$

In 3D, three vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ form an orthonormal basis if

$$\|\mathbf{u}\| = \|\mathbf{v}\| = \|\mathbf{w}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0.$$

This orthonormal basis is *right-handed* provided

$$\mathbf{w} = \mathbf{u} \times \mathbf{v},$$

and otherwise, it is left-handed.

Note that the Cartesian canonical orthonormal basis is just one of infinitely many possible orthonormal bases. What makes it special is that it and its implicit origin location are used for low-level representation within a program. Thus, the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are never explicitly stored and neither is the canonical origin location $\mathbf{o}$. The global model is typically stored in this canonical coordinate system, and it is thus often called the *global coordinate system*. However, if we want to use another coordinate system with origin $\mathbf{p}$ and orthonormal basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, then we *do* store those vectors explicitly. Such a system is called a *frame of reference* or *coordinate frame*. For example, in a flight simulator, we might want to maintain a coordinate system with the origin at the nose of the plane, and the orthonormal basis aligned with the airplane. Simultaneously, we would have the master canonical coordinate system (Figure 2.21). The coordinate system associated with a particular object, such as the plane, is usually called a *local coordinate system*.

At a low level, the local frame is stored in canonical coordinates. For example, if $\mathbf{u}$ has coordinates (x_u, y_u, z_u),

$$\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}.$$

A location implicitly includes an offset from the canonical origin:

$$\mathbf{p} = \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y} + z_p\mathbf{z},$$

where (x_p, y_p, z_p) are the coordinates of $\mathbf{p}$.

Note that if we store a vector $\mathbf{a}$ with respect to the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ frame, we store a triple (u_a, v_a, w_a) which we can interpret geometrically as

$$\mathbf{a} = u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}.$$

To get the canonical coordinates of a vector $\mathbf{a}$ stored in the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ coordinate system, simply recall that $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are themselves stored in terms of Cartesian coordinates, so the expression $u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}$ is already in Cartesian coordinates if evaluated explicitly. To get the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ coordinates of a vector $\mathbf{b}$ stored in the canonical coordinate system, we can use dot products:

$$u_b = \mathbf{u} \cdot \mathbf{b}; \quad v_b = \mathbf{v} \cdot \mathbf{b}; \quad w_b = \mathbf{w} \cdot \mathbf{b}.$$

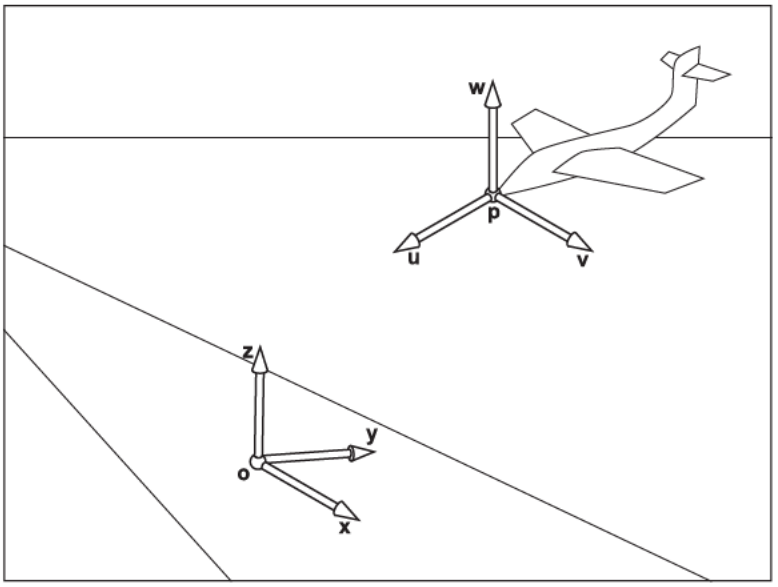

Figure 2.21. There is always a master or "canonical" coordinate system with origin **o** and orthonormal basis **x**, **y**, and **z**. This coordinate system is usually defined to be aligned to the global model and is thus often called the "global" or "world" coordinate system. This origin and basis vectors are never stored explicitly. All other vectors and locations are stored with coordinates that relate them to the global frame. The coordinate system associated with the plane is explicitly stored in terms of global coordinates.

This works because we know that for *some* u_b, v_b, and w_b,

$$u_b\mathbf{u} + v_b\mathbf{v} + w_b\mathbf{w} = \mathbf{b},$$

and the dot product isolates the u_b coordinate:

$$\begin{aligned}\mathbf{u}\cdot\mathbf{b} &= u_b(\mathbf{u}\cdot\mathbf{u}) + v_b(\mathbf{u}\cdot\mathbf{v}) + w_b(\mathbf{u}\cdot\mathbf{w}) \\ &= u_b.\end{aligned}$$

This works because **u**, **v**, and **w** are orthonormal.

Using matrices to manage changes of coordinate systems is discussed in Sections 7.2.1 and 7.5.

2.4.6 Constructing a Basis from a Single Vector

Often we need an orthonormal basis that is aligned with a given vector. That is, given a vector **a**, we want an orthonormal **u**, **v**, and **w** such that **w** points in the

same direction as $\mathbf{a}$ (Hughes & Möller, 1999), but we don't particularly care what $\mathbf{u}$ and $\mathbf{v}$ are. One vector isn't enough to uniquely determine the answer; we just need a robust procedure that will find any one of the possible bases.

This can be done using cross products as follows. First, make $\mathbf{w}$ a unit vector in the direction of $\mathbf{a}$:

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|}.$$

This same procedure can, of course, be used to construct the three vectors in any order; just pay attention to the order of the cross products to ensure the basis is right-handed.

Then, choose any vector $\mathbf{t}$ not collinear with $\mathbf{w}$, and use the cross product to build a unit vector $\mathbf{u}$ perpendicular to $\mathbf{w}$:

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}.$$

If $\mathbf{t}$ is collinear with $\mathbf{w}$, the denominator will vanish, and if they are nearly collinear, the results will have low precision. A simple procedure to find a vector sufficiently different from $\mathbf{w}$ is to start with $\mathbf{t}$ equal to $\mathbf{w}$ and change the smallest magnitude component of $\mathbf{t}$ to 1. For example, if $\mathbf{w} = (1/\sqrt{2}, -1/\sqrt{2}, 0)$, then $\mathbf{t} = (1/\sqrt{2}, -1/\sqrt{2}, 1)$. Once $\mathbf{w}$ and $\mathbf{u}$ are in hand, completing the basis is simple:

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

An example of a situation where this construction is used is surface shading, where a basis aligned to the surface normal is needed but the rotation around the normal is often unimportant.

For serious production code, recently researchers at Pixar have developed a rather remarkable method for constructing a vector from two vectors that is impressive in its compactness and efficiency (Duff et al., 2017). They provide battle-tested code, and readers are encouraged to use it as there are not "gotchas" that have emerged as it used throughout the industry.

2.4.7 Constructing a Basis from Two Vectors

The procedure in the previous section can also be used in situations where the rotation of the basis around the given vector is important. A common example is building a basis for a camera: it's important to have one vector aligned in the direction the camera is looking, but the orientation of the camera around that vector is *not* arbitrary, and it needs to be specified somehow. Once the orientation is pinned down, the basis is completely determined.

A common way to fully specify a frame is by providing two vectors $\mathbf{a}$ (which specifies $\mathbf{w}$) and $\mathbf{b}$ (which specifies $\mathbf{v}$). If the two vectors are known to be perpendicular, it is a simple matter to construct the third vector by $\mathbf{u} = \mathbf{b} \times \mathbf{a}$.

$\mathbf{u} = \mathbf{a} \times \mathbf{b}$ also produces an orthonormal basis, but it is left-handed.

To be sure that the resulting basis really is orthonormal, even if the input vectors weren't quite, a procedure much like the single-vector procedure is advisable:

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|},$$
$$\mathbf{u} = \frac{\mathbf{b} \times \mathbf{w}}{\|\mathbf{b} \times \mathbf{w}\|},$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

If you want me to set $\mathbf{w}$ and $\mathbf{v}$ to two non-perpendicular directions, something has to give—with this scheme, I'll set everything the way you want, except I'll make the smallest change to $\mathbf{v}$ so that it is in fact perpendicular to $\mathbf{w}$.

What will go wrong with the computation if $\mathbf{a}$ and $\mathbf{b}$ are parallel?

In fact, this procedure works just fine when $\mathbf{a}$ and $\mathbf{b}$ are not perpendicular. In this case, $\mathbf{w}$ will be constructed exactly in the direction of $\mathbf{a}$, and $\mathbf{v}$ is chosen to be the closest vector to $\mathbf{b}$ among all vectors perpendicular to $\mathbf{w}$.

This procedure *won't* work if $\mathbf{a}$ and $\mathbf{b}$ are collinear. In this case, $\mathbf{b}$ is of no help in choosing which of the directions perpendicular to $\mathbf{a}$ we should use: it is perpendicular to all of them.

In the example of specifying camera positions (Section 4.3), we want to construct a frame that has $\mathbf{w}$ parallel to the direction the camera is looking, and $\mathbf{v}$ should point out the top of the camera. To orient the camera upright, we build the basis around the view direction, using the straight-up direction as the reference vector to establish the camera's orientation around the view direction. Setting $\mathbf{v}$ as close as possible to straight up exactly matches the intuitive notion of "holding the camera straight."

2.4.8 Squaring Up a Basis

Occasionally, you may find problems caused in your computations by a basis that is supposed to be orthonormal but where error has crept in—due to rounding error in computation, or to the basis having been stored in a file with low precision, for instance.

The procedure of the previous section can be used; simply constructing the basis anew using the existing $\mathbf{w}$ and $\mathbf{v}$ vectors will produce a new basis that is orthonormal and is close to the old one.

This approach is good for many applications, but it is not the best available. It does produce accurately orthogonal vectors, and for nearly orthogonal starting bases, the result will not stray far from the starting point. However, it is asymmetric: it "favors" $\mathbf{w}$ over $\mathbf{v}$ and $\mathbf{v}$ over $\mathbf{u}$ (whose starting value is thrown away). It chooses a basis close to the starting basis but has no guarantee of choosing *the* closest orthonormal basis. When this is not good enough, the SVD (Section 6.4.1) can be used to compute an orthonormal basis that *is* guaranteed to be closest to the original basis.