Chapter 4
Transforms

"What if angry vectors veer
Round your sleeping head, and form.
There's never need to fear
Violence of the poor world's abstract storm."
—Robert Penn Warren

A *transform* is an operation that takes entities such as points, vectors, or colors and converts them in some way. For the computer graphics practitioner, it is extremely important to master transforms. With them, you can position, reshape, and animate objects, lights, and cameras. You can also ensure that all computations are carried out in the same coordinate system, and project objects onto a plane in different ways. These are only a few of the operations that can be performed with transforms, but they are sufficient to demonstrate the importance of the transform's role in real-time graphics, or, for that matter, in any kind of computer graphics.

A *linear transform* is one that preserves vector addition and scalar multiplication. Specifically,

$$\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{y}) = \mathbf{f}(\mathbf{x} + \mathbf{y}), \tag{4.1}$$

$$k\mathbf{f}(\mathbf{x}) = \mathbf{f}(k\mathbf{x}). \tag{4.2}$$

As an example, $\mathbf{f}(\mathbf{x}) = 5\mathbf{x}$ is a transform that takes a vector and multiplies each element by five. To prove that this is linear, the two conditions (Equations 4.1 and 4.2) need to be fulfilled. The first condition holds since any two vectors multiplied by five and then added will be the same as adding the vectors and then multiplying. The scalar multiplication condition (Equation 4.2) is clearly fulfilled. This function is called a scaling transform, as it changes the scale (size) of an object. The rotation transform is another linear transform that rotates a vector about the origin. Scaling and rotation transforms, in fact all linear transforms for three-element vectors, can be represented using a 3×3 matrix.

However, this size of matrix is usually not large enough. A function for a three-element vector $\mathbf{x}$ such as $\mathbf{f}(\mathbf{x}) = \mathbf{x} + (7, 3, 2)$ is not linear. Performing this function on two separate vectors will add each value of $(7, 3, 2)$ twice to form the result. Adding a fixed vector to another vector performs a translation, e.g., it moves all locations by

the same amount. This is a useful type of transform, and we would like to combine various transforms, e.g., scale an object to be half as large, then move it to a different location. Keeping functions in the simple forms used so far makes it difficult to easily combine them.

Combining linear transforms and translations can be done using an *affine transform*, typically stored as a 4×4 matrix. An affine transform is one that performs a linear transform and then a translation. To represent four-element vectors we use *homogeneous notation*, denoting points and directions in the same way (using bold lowercase letters). A direction vector is represented as $\mathbf{v} = (v_x \quad v_y \quad v_z \quad 0)^T$ and a point as $\mathbf{v} = (v_x \quad v_y \quad v_z \quad 1)^T$. Throughout the chapter, we will make extensive use of the terminology and operations explained in the downloadable linear algebra appendix, found on realtimerendering.com.

All translation, rotation, scaling, reflection, and shearing matrices are affine. The main characteristic of an affine matrix is that it preserves the parallelism of lines, but not necessarily lengths and angles. An affine transform may also be any sequence of concatenations of individual affine transforms.

This chapter will begin with the most essential, basic affine transforms. This section can be seen as a "reference manual" for simple transforms. More specialized matrices are then described, followed by a discussion and description of quaternions, a powerful transform tool. Then follows vertex blending and morphing, which are two simple but effective ways of expressing animations of meshes. Finally, projection matrices are described. Most of these transforms, their notations, functions, and properties are summarized in Table 4.1, where an orthogonal matrix is one whose inverse is the transpose.

Transforms are a basic tool for manipulating geometry. Most graphics application programming interfaces let the user set arbitrary matrices, and sometimes a library may be used with matrix operations that implement many of the transforms discussed in this chapter. However, it is still worthwhile to understand the real matrices and their interaction behind the function calls. Knowing what the matrix does after such a function call is a start, but understanding the properties of the matrix itself will take you further. For example, such an understanding enables you to discern when you are dealing with an orthogonal matrix, whose inverse is its transpose, making for faster matrix inversions. Knowledge like this can lead to accelerated code.

4.1 Basic Transforms

This section describes the most basic transforms, such as translation, rotation, scaling, shearing, transform concatenation, the rigid-body transform, normal transform (which is not so normal), and computation of inverses. For the experienced reader, this can be used as a reference manual for simple transforms, and for the novice, it can serve as an introduction to the subject. This material is necessary background for the rest of this chapter and for other chapters in this book. We start with the simplest of transforms—the translation.

Notation	Name	Characteristics
$\mathbf{T}(\mathbf{t})$	translation matrix	Moves a point. Affine.
$\mathbf{R}_x(\rho)$	rotation matrix	Rotates ρ radians around the x-axis. Similar notation for the y- and z-axes. Orthogonal & affine.
$\mathbf{R}$	rotation matrix	Any rotation matrix. Orthogonal & affine.
$\mathbf{S}(\mathbf{s})$	scaling matrix	Scales along all x-, y-, and z-axes according to $\mathbf{s}$. Affine.
$\mathbf{H}_{ij}(s)$	shear matrix	Shears component i by a factor s, with respect to component j. $i, j \in \{x, y, z\}$. Affine.
$\mathbf{E}(h, p, r)$	Euler transform	Orientation matrix given by the Euler angles head (yaw), pitch, roll. Orthogonal & affine.
$\mathbf{P}_o(s)$	orthographic projection	Parallel projects onto some plane or to a volume. Affine.
$\mathbf{P}_p(s)$	perspective projection	Projects with perspective onto a plane or to a volume.
$\texttt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t)$	slerp transform	Creates an interpolated quaternion with respect to the quaternions $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, and the parameter t.

Table 4.1. Summary of most of the transforms discussed in this chapter.

4.1.1 Translation

A change from one location to another is represented by a translation matrix, $\mathbf{T}$. This matrix translates an entity by a vector $\mathbf{t} = (t_x, t_y, t_z)$. $\mathbf{T}$ is given below by Equation 4.3:

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.3}$$

An example of the effect of the translation transform is shown in Figure 4.1. It is easily shown that the multiplication of a point $\mathbf{p} = (p_x, p_y, p_z, 1)$ with $\mathbf{T}(\mathbf{t})$ yields a new point $\mathbf{p}' = (p_x + t_x, p_y + t_y, p_z + t_z, 1)$, which is clearly a translation. Notice that a vector $\mathbf{v} = (v_x, v_y, v_z, 0)$ is left unaffected by a multiplication by $\mathbf{T}$, because a direction vector cannot be translated. In contrast, both points and vectors are affected by the rest of the affine transforms. The inverse of a translation matrix is $\mathbf{T}^{-1}(\mathbf{t}) = \mathbf{T}(-\mathbf{t})$, that is, the vector $\mathbf{t}$ is negated.

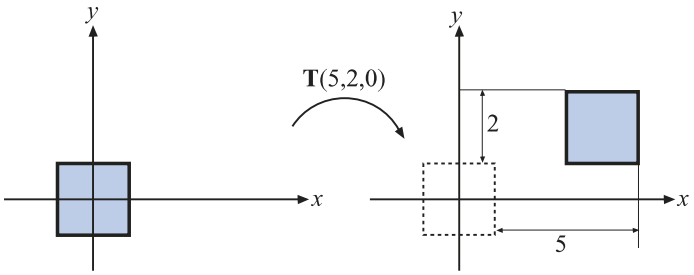

Figure 4.1. The square on the left is transformed with a translation matrix $\mathbf{T}(5, 2, 0)$, whereby the square is moved 5 distance units to the right and 2 upward.

We should mention at this point that another valid notational scheme sometimes seen in computer graphics uses matrices with translation vectors in the bottom row. For example, DirectX uses this form. In this scheme, the order of matrices would be reversed, i.e., the order of application would read from left to right. Vectors and matrices in this notation are said to be in *row-major* form since the vectors are rows. In this book, we use *column-major* form. Whichever is used, this is purely a notational difference. When the matrix is stored in memory, the last four values of the sixteen are the three translation values followed by a one.

4.1.2 Rotation

A rotation transform rotates a vector (position or direction) by a given angle around a given axis passing through the origin. Like a translation matrix, it is a *rigid-body transform*, i.e., it preserves the distances between points transformed, and preserves handedness (i.e., it never causes left and right to swap sides). These two types of transforms are clearly useful in computer graphics for positioning and orienting objects. An *orientation matrix* is a rotation matrix associated with a camera view or object that defines its orientation in space, i.e., its directions for up and forward.

In two dimensions, the rotation matrix is simple to derive. Assume that we have a vector, $\mathbf{v} = (v_x, v_y)$, which we parameterize as $\mathbf{v} = (v_x, v_y) = (r\cos\theta, r\sin\theta)$. If we were to rotate that vector by ϕ radians (counterclockwise), then we would get $\mathbf{u} = (r\cos(\theta+\phi), r\sin(\theta+\phi))$. This can be rewritten as

$$\begin{aligned}\mathbf{u} &= \begin{pmatrix} r\cos(\theta+\phi) \\ r\sin(\theta+\phi) \end{pmatrix} = \begin{pmatrix} r(\cos\theta\cos\phi - \sin\theta\sin\phi) \\ r(\sin\theta\cos\phi + \cos\theta\sin\phi) \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}}_{\mathbf{R}(\phi)} \underbrace{\begin{pmatrix} r\cos\theta \\ r\sin\theta \end{pmatrix}}_{\mathbf{v}} = \mathbf{R}(\phi)\mathbf{v},\end{aligned} \tag{4.4}$$

where we used the angle sum relation to expand $\cos(\theta+\phi)$ and $\sin(\theta+\phi)$. In three dimensions, commonly used rotation matrices are $\mathbf{R}_x(\phi)$, $\mathbf{R}_y(\phi)$, and $\mathbf{R}_z(\phi)$, which

rotate an entity ϕ radians around the x-, y-, and z-axes, respectively. They are given by Equations 4.5–4.7:

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.5}$$

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.6}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.7}$$

If the bottom row and rightmost column are deleted from a 4×4 matrix, a 3×3 matrix is obtained. For every 3×3 rotation matrix, $\mathbf{R}$, that rotates ϕ radians around any axis, the trace (which is the sum of the diagonal elements in a matrix) is constant independent of the axis, and is computed as [997]:

$$\mathrm{tr}(\mathbf{R}) = 1 + 2\cos\phi. \tag{4.8}$$

The effect of a rotation matrix may be seen in Figure 4.4 on page 65. What characterizes a rotation matrix, $\mathbf{R}_i(\phi)$, besides the fact that it rotates ϕ radians around axis i, is that it leaves all points on the rotation axis, i, unchanged. Note that $\mathbf{R}$ will also be used to denote a rotation matrix around any axis. The axis rotation matrices given above can be used in a series of three transforms to perform any arbitrary axis rotation. This procedure is discussed in Section 4.2.1. Performing a rotation around an arbitrary axis directly is covered in Section 4.2.4.

All rotation matrices have a determinant of one and are orthogonal. This also holds for concatenations of any number of these transforms. There is another way to obtain the inverse: $\mathbf{R}_i^{-1}(\phi) = \mathbf{R}_i(-\phi)$, i.e., rotate in the opposite direction around the same axis.

Example: Rotation Around a Point. Assume that we want to rotate an object by ϕ radians around the z-axis, with the center of rotation being a certain point, $\mathbf{p}$. What is the transform? This scenario is depicted in Figure 4.2. Since a rotation around a point is characterized by the fact that the point itself is unaffected by the rotation, the transform starts by translating the object so that $\mathbf{p}$ coincides with the origin, which is done with $\mathbf{T}(-\mathbf{p})$. Thereafter follows the actual rotation: $\mathbf{R}_z(\phi)$. Finally, the object has to be translated back to its original position using $\mathbf{T}(\mathbf{p})$. The resulting transform, $\mathbf{X}$, is then given by

$$\mathbf{X} = \mathbf{T}(\mathbf{p})\mathbf{R}_z(\phi)\mathbf{T}(-\mathbf{p}). \tag{4.9}$$

Note the order of the matrices above. □

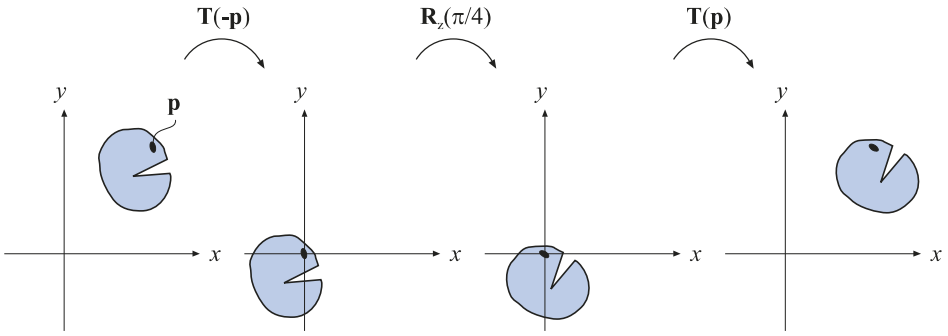

Figure 4.2. Example of rotation around a specific point $\mathbf{p}$.

4.1.3 Scaling

A scaling matrix, $\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z)$, scales an entity with factors s_x, s_y, and s_z along the x-, y-, and z-directions, respectively. This means that a scaling matrix can be used to enlarge or diminish an object. The larger the s_i, $i \in \{x, y, z\}$, the larger the scaled entity gets in that direction. Setting any of the components of $\mathbf{s}$ to 1 naturally avoids a change in scaling in that direction. Equation 4.10 shows $\mathbf{S}$:

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.10}$$

Figure 4.4 on page 65 illustrates the effect of a scaling matrix. The scaling operation is called *uniform* if $s_x = s_y = s_z$ and *nonuniform* otherwise. Sometimes the terms *isotropic* and *anisotropic* scaling are used instead of uniform and nonuniform. The inverse is $\mathbf{S}^{-1}(\mathbf{s}) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$.

Using homogeneous coordinates, another valid way to create a uniform scaling matrix is by manipulating matrix element at position $(3, 3)$, i.e., the element at the lower right corner. This value affects the w-component of the homogeneous coordinate, and so scales every coordinate of a point (not direction vectors) transformed by the matrix. For example, to scale uniformly by a factor of 5, the elements at $(0, 0)$, $(1, 1)$, and $(2, 2)$ in the scaling matrix can be set to 5, or the element at $(3, 3)$ can be set to $1/5$. The two different matrices for performing this are shown below:

$$\mathbf{S} = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad \mathbf{S}' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/5 \end{pmatrix}. \tag{4.11}$$

In contrast to using $\mathbf{S}$ for uniform scaling, using $\mathbf{S}'$ must always be followed by homogenization. This may be inefficient, since it involves divides in the homogenization

process; if the element at the lower right (position $(3,3)$) is 1, no divides are necessary. Of course, if the system always does this division without testing for 1, then there is no extra cost.

A negative value on one or three of the components of $\mathbf{s}$ gives a type of *reflection matrix*, also called a *mirror matrix*. If only two scale factors are -1, then we will rotate π radians. It should be noted that a rotation matrix concatenated with a reflection matrix is also a reflection matrix. Hence, the following is a reflection matrix:

$$\underbrace{\begin{pmatrix} \cos(\pi/2) & \sin(\pi/2) \\ -\sin(\pi/2) & \cos(\pi/2) \end{pmatrix}}_{\text{rotation}} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}}_{\text{reflection}} = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}. \tag{4.12}$$

Reflection matrices usually require special treatment when detected. For example, a triangle with vertices in a counterclockwise order will get a clockwise order when transformed by a reflection matrix. This order change can cause incorrect lighting and backface culling to occur. To detect whether a given matrix reflects in some manner, compute the determinant of the upper left 3×3 elements of the matrix. If the value is negative, the matrix is reflective. For example, the determinant of the matrix in Equation 4.12 is $0 \cdot 0 - (-1) \cdot (-1) = -1$.

Example: Scaling in a Certain Direction. The scaling matrix $\mathbf{S}$ scales along only the x-, y-, and z-axes. If scaling should be performed in other directions, a compound transform is needed. Assume that scaling should be done along the axes of the orthonormal, right-oriented vectors $\mathbf{f}^x$, $\mathbf{f}^y$, and $\mathbf{f}^z$. First, construct the matrix $\mathbf{F}$, to change the basis, as below:

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}^x & \mathbf{f}^y & \mathbf{f}^z & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.13}$$

The idea is to make the coordinate system given by the three axes coincide with the standard axes, then use the standard scaling matrix, and then transform back. The first step is carried out by multiplying with the transpose, i.e., the inverse, of $\mathbf{F}$. Then the actual scaling is done, followed by a transform back. The transform is shown in Equation 4.14:

$$\mathbf{X} = \mathbf{F}\mathbf{S}(\mathbf{s})\mathbf{F}^T. \tag{4.14}$$

□

4.1.4 Shearing

Another class of transforms is the set of shearing matrices. These can, for example, be used in games to distort an entire scene to create a psychedelic effect or otherwise warp a model's appearance. There are six basic shearing matrices, and they are denoted $\mathbf{H}_{xy}(s)$, $\mathbf{H}_{xz}(s)$, $\mathbf{H}_{yx}(s)$, $\mathbf{H}_{yz}(s)$, $\mathbf{H}_{zx}(s)$, and $\mathbf{H}_{zy}(s)$. The first subscript is used to denote which coordinate is being changed by the shear matrix, while the second

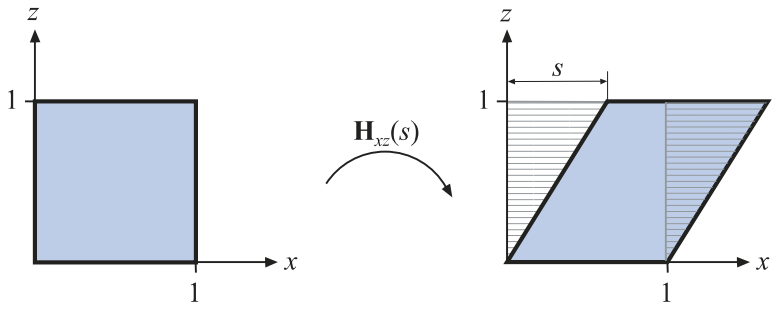

Figure 4.3. The effect of shearing the unit square with $\mathbf{H}_{xz}(s)$. Both the y- and z-values are unaffected by the transform, while the x-value is the sum of the old x-value and s multiplied by the z-value, causing the square to become slanted. This transform is area-preserving, which can be seen in that the dashed areas are the same.

subscript indicates the coordinate which does the shearing. An example of a shear matrix, $\mathbf{H}_{xz}(s)$, is shown in Equation 4.15. Observe that the subscript can be used to find the position of the parameter s in the matrix below; the x (whose numeric index is 0) identifies row zero, and the z (whose numeric index is 2) identifies column two, and so the s is located there:

$$\mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.15}$$

The effect of multiplying this matrix with a point $\mathbf{p}$ yields a point: $(p_x+sp_z \;\; p_y \;\; p_z)^T$. Graphically, this is shown for the unit square in Figure 4.3. The inverse of $\mathbf{H}_{ij}(s)$ (shearing the ith coordinate with respect to the jth coordinate, where $i \neq j$), is generated by shearing in the opposite direction, that is, $\mathbf{H}_{ij}^{-1}(s) = \mathbf{H}_{ij}(-s)$.

You can also use a slightly different kind of shear matrix:

$$\mathbf{H}'_{xy}(s,t) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.16}$$

Here, however, both subscripts are used to denote that these coordinates are to be sheared by the third coordinate. The connection between these two different kinds of descriptions is $\mathbf{H}'_{ij}(s,t) = \mathbf{H}_{ik}(s)\mathbf{H}_{jk}(t)$, where k is used as an index to the third coordinate. The right matrix to use is a matter of taste. Finally, it should be noted that since the determinant of any shear matrix $|\mathbf{H}| = 1$, this is a volume-preserving transformation, which also is illustrated in Figure 4.3.

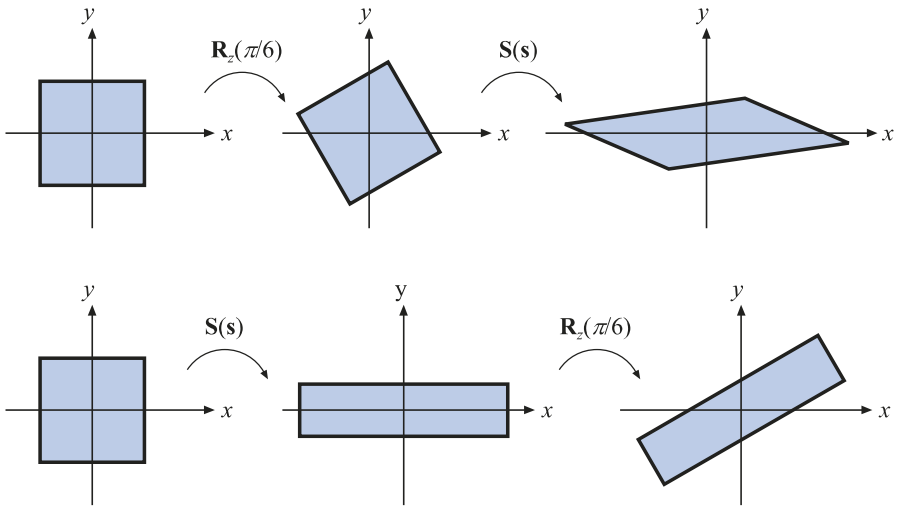

Figure 4.4. This illustrates the order dependency when multiplying matrices. In the top row, the rotation matrix $\mathbf{R}_z(\pi/6)$ is applied followed by a scaling, $\mathbf{S}(\mathbf{s})$, where $\mathbf{s} = (2, 0.5, 1)$. The composite matrix is then $\mathbf{S}(\mathbf{s})\mathbf{R}_z(\pi/6)$. In the bottom row, the matrices are applied in the reverse order, yielding $\mathbf{R}_z(\pi/6)\mathbf{S}(\mathbf{s})$. The results are clearly different. It generally holds that $\mathbf{MN} \neq \mathbf{NM}$, for arbitrary matrices $\mathbf{M}$ and $\mathbf{N}$.

4.1.5 Concatenation of Transforms

Due to the noncommutativity of the multiplication operation on matrices, the order in which the matrices occur matters. Concatenation of transforms is therefore said to be order-dependent.

As an example of order dependency, consider two matrices, $\mathbf{S}$ and $\mathbf{R}$. $\mathbf{S}(2, 0.5, 1)$ scales the x-component by a factor two and the y-component by a factor 0.5. $\mathbf{R}_z(\pi/6)$ rotates $\pi/6$ radians counterclockwise around the z-axis (which points outward from page of this book in a right-handed coordinate system). These matrices can be multiplied in two ways, with the results being entirely different. The two cases are shown in Figure 4.4.

The obvious reason to concatenate a sequence of matrices into a single one is to gain efficiency. For example, imagine that you have a game scene that has several million vertices, and that all objects in the scene must be scaled, rotated, and finally translated. Now, instead of multiplying all vertices with each of the three matrices, the three matrices are concatenated into a single matrix. This single matrix is then applied to the vertices. This composite matrix is $\mathbf{C} = \mathbf{TRS}$. Note the order here. The scaling matrix, $\mathbf{S}$, should be applied to the vertices first, and therefore appears to the right in the composition. This ordering implies that $\mathbf{TRSp} = (\mathbf{T}(\mathbf{R}(\mathbf{Sp})))$, where $\mathbf{p}$ is a point to be transformed. Incidentally, $\mathbf{TRS}$ is the order commonly used by scene graph systems.

It is worth noting that while matrix concatenation is order-dependent, the matrices can be grouped as desired. For example, say that with **TRSp** you would like to compute the rigid-body motion transform **TR** once. It is valid to group these two matrices together, (**TR**)(**Sp**), and replace with the intermediate result. Thus, matrix concatenation is *associative*.

4.1.6 The Rigid-Body Transform

When a person grabs a solid object, say a pen from a table, and moves it to another location, perhaps to a shirt pocket, only the object's orientation and location change, while the shape of the object generally is not affected. Such a transform, consisting of concatenations of only translations and rotations, is called a rigid-body transform. It has the characteristic of preserving lengths, angles, and handedness.

Any rigid-body matrix, $\mathbf{X}$, can be written as the concatenation of a translation matrix, $\mathbf{T}(\mathbf{t})$, and a rotation matrix, $\mathbf{R}$. Thus, $\mathbf{X}$ has the appearance of the matrix in Equation 4.17:

$$\mathbf{X} = \mathbf{T}(\mathbf{t})\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.17}$$

The inverse of $\mathbf{X}$ is computed as $\mathbf{X}^{-1} = (\mathbf{T}(\mathbf{t})\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{T}(\mathbf{t})^{-1} = \mathbf{R}^T\mathbf{T}(-\mathbf{t})$. Thus, to compute the inverse, the upper left 3×3 matrix of $\mathbf{R}$ is transposed, and the translation values of $\mathbf{T}$ change sign. These two new matrices are multiplied together in opposite order to obtain the inverse. Another way to compute the inverse of $\mathbf{X}$ is to consider $\mathbf{R}$ (making $\mathbf{R}$ appear as 3×3 matrix) and $\mathbf{X}$ in the following notation (notation described on page 6 with Equation 1.2):

$$\begin{aligned} \bar{\mathbf{R}} = \begin{pmatrix} \mathbf{r}_{,0} & \mathbf{r}_{,1} & \mathbf{r}_{,2} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_{0,}^T \\ \mathbf{r}_{1,}^T \\ \mathbf{r}_{2,}^T \end{pmatrix}, \\ \Downarrow \\ \mathbf{X} = \begin{pmatrix} \bar{\mathbf{R}} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix}, \end{aligned} \tag{4.18}$$

where $\mathbf{r}_{,0}$ means the first column of the rotation matrix (i.e., the comma indicates any value from 0 to 2, while the second subscript is 0) and $\mathbf{r}_{0,}^T$ is the first row of the column matrix. Note that $\mathbf{0}$ is a 3×1 column vector filled with zeros. Some calculations yield the inverse in the expression shown in Equation 4.19:

$$\mathbf{X}^{-1} = \begin{pmatrix} \mathbf{r}_{0,} & \mathbf{r}_{1,} & \mathbf{r}_{2,} & -\bar{\mathbf{R}}^T\mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.19}$$

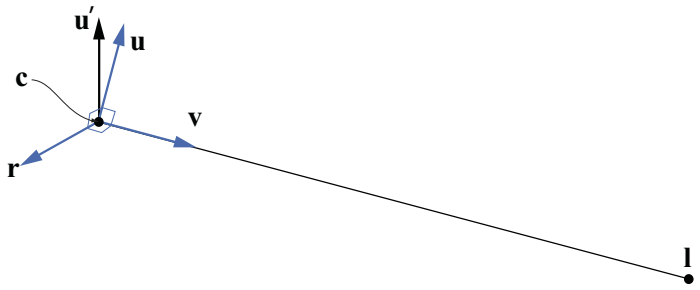

Figure 4.5. The geometry involved in computing a transform that orients the camera at $\mathbf{c}$, with up vector $\mathbf{u}'$, to look at the point $\mathbf{l}$. For that purpose, we need to compute $\mathbf{r}$, $\mathbf{u}$, and $\mathbf{v}$.

Example: Orienting the Camera. A common task in graphics is to orient the camera so that it looks at a certain position. Here we will present what `gluLookAt()` (from the OpenGL Utility Library, GLU for short) does. Even though this function call itself is not used much nowadays, the task remains common. Assume that the camera is located at $\mathbf{c}$, that we want the camera to look at a target $\mathbf{l}$, and that a given up direction of the camera is $\mathbf{u}'$, as illustrated in Figure 4.5. We want to compute a basis consisting of three vectors, $\{\mathbf{r}, \mathbf{u}, \mathbf{v}\}$. We start by computing the view vector as $\mathbf{v} = (\mathbf{c}-\mathbf{l})/||\mathbf{c}-\mathbf{l}||$, i.e., the normalized vector from the target to the camera position. A vector looking to the "right" can then be computed as $\mathbf{r} = -(\mathbf{v} \times \mathbf{u}')/||\mathbf{v} \times \mathbf{u}'||$. The $\mathbf{u}'$ vector is often not guaranteed to be pointing precisely up, so the final up vector is another cross product, $\mathbf{u} = \mathbf{v} \times \mathbf{r}$, which is guaranteed to be normalized since both $\mathbf{v}$ and $\mathbf{r}$ are normalized and perpendicular by construction. In the camera transform matrix, $\mathbf{M}$, that we will construct, the idea is to first translate everything so the camera position is at the origin, $(0,0,0)$, and then change the basis so that $\mathbf{r}$ is aligned with $(1,0,0)$, $\mathbf{u}$ with $(0,1,0)$, and $\mathbf{v}$ with $(0,0,1)$. This is done by

$$\mathbf{M} = \underbrace{\begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{change of basis}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{translation}} = \begin{pmatrix} r_x & r_y & r_z & -\mathbf{t}\cdot\mathbf{r} \\ u_x & u_y & u_z & -\mathbf{t}\cdot\mathbf{u} \\ v_x & v_y & v_z & -\mathbf{t}\cdot\mathbf{v} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.20}$$

Note that when concatenating the translation matrix with the change of basis matrix, the translation $-\mathbf{t}$ is to the right since it should be applied first. One way to remember where to put the components of $\mathbf{r}$, $\mathbf{u}$, and $\mathbf{v}$ is the following. We want $\mathbf{r}$ to become $(1,0,0)$, so when multiplying a change of basis matrix with $(1,0,0)$, we can see that the first row in the matrix must be the elements of $\mathbf{r}$, since $\mathbf{r}\cdot\mathbf{r} = 1$. Furthermore, the second row and the third row must consist of vectors that are perpendicular to $\mathbf{r}$, i.e., $\mathbf{r}\cdot\mathbf{x} = 0$. When applying the same thinking also to $\mathbf{u}$ and $\mathbf{v}$, we arrive at the change of basis matrix above. □

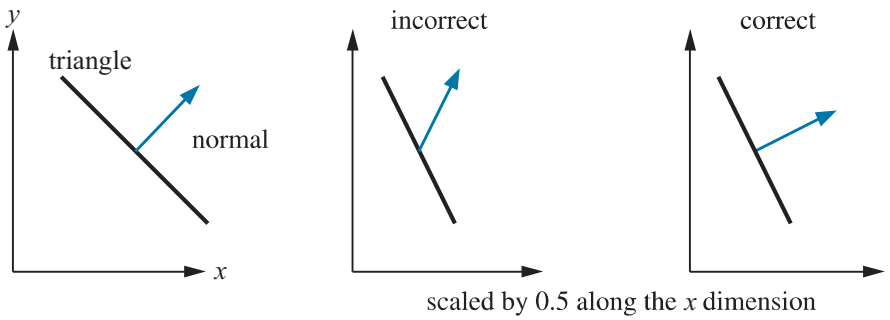

Figure 4.6. On the left is the original geometry, a triangle and its normal shown from the side. The middle illustration shows what happens if the model is scaled along the x- axis by 0.5 and the normal uses the same matrix. The right figure shows the proper transform of the normal.

4.1.7 Normal Transform

A single matrix can be used to consistently transform points, lines, triangles, and other geometry. The same matrix can also transform tangent vectors following along these lines or on the surfaces of triangles. However, this matrix cannot always be used to transform one important geometric property, the surface normal (and the vertex lighting normal). Figure 4.6 shows what can happen if this same matrix is used.

Instead of multiplying by the matrix itself, the proper method is to use the transpose of the matrix's adjoint [227]. Computation of the adjoint is described in our online linear algebra appendix. The adjoint is always guaranteed to exist. The normal is not guaranteed to be of unit length after being transformed, so typically needs to be normalized.

The traditional answer for transforming the normal is that the transpose of the inverse is computed [1794]. This method normally works. The full inverse is not necessary, however, and occasionally cannot be created. The inverse is the adjoint divided by the original matrix's determinant. If this determinant is zero, the matrix is singular, and the inverse does not exist.

Even computing just the adjoint for a full 4×4 matrix can be expensive, and is usually not necessary. Since the normal is a vector, translation will not affect it. Furthermore, most modeling transforms are affine. They do not change the w-component of the homogeneous coordinate passed in, i.e., they do not perform projection. Under these (common) circumstances, all that is needed for normal transformation is to compute the adjoint of the upper left 3×3 components.

Often even this adjoint computation is not needed. Say we know the transform matrix is composed entirely of a concatenation of translations, rotations, and uniform scaling operations (no stretching or squashing). Translations do not affect the normal. The uniform scaling factors simply change the length of the normal. What is left is a series of rotations, which always yields a net rotation of some sort, nothing more.

The transpose of the inverse can be used to transform normals. A rotation matrix is defined by the fact that its transpose is its inverse. Substituting to get the normal transform, two transposes (or two inverses) give the original rotation matrix. Putting it all together, the original transform itself can also be used directly to transform normals under these circumstances.

Finally, fully renormalizing the normal produced is not always necessary. If only translations and rotations are concatenated together, the normal will not change length when transformed by the matrix, so no renormalizing is needed. If uniform scalings are also concatenated, the overall scale factor (if known, or extracted—Section 4.2.3) can be used to directly normalize the normals produced. For example, if we know that a series of scalings were applied that makes the object 5.2 times larger, then normals transformed directly by this matrix are renormalized by dividing them by 5.2. Alternately, to create a normal transform matrix that would produce normalized results, the original matrix's 3×3 upper left could be divided by this scale factor once.

Note that normal transforms are not an issue in systems where, after transformation, the surface normal is derived from the triangle (e.g., using the cross product of the triangle's edges). Tangent vectors are different than normals in nature, and are always directly transformed by the original matrix.

4.1.8 Computation of Inverses

Inverses are needed in many cases, for example, when changing back and forth between coordinate systems. Depending on the available information about a transform, one of the following three methods of computing the inverse of a matrix can be used:

- If the matrix is a single transform or a sequence of simple transforms with given parameters, then the matrix can be computed easily by "inverting the parameters" and the matrix order. For example, if $\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(\phi)$, then $\mathbf{M}^{-1} = \mathbf{R}(-\phi)\mathbf{T}(-\mathbf{t})$. This is simple and preserves the accuracy of the transform, which is important when rendering huge worlds [1381].
- If the matrix is known to be orthogonal, then $\mathbf{M}^{-1} = \mathbf{M}^T$, i.e., the transpose is the inverse. Any sequence of rotations is a rotation, and so is orthogonal.
- If nothing is known, then the adjoint method, Cramer's rule, LU decomposition, or Gaussian elimination could be used to compute the inverse. Cramer's rule and the adjoint method are generally preferable, as they have fewer branch operations; "if" tests are good to avoid on modern architectures. See Section 4.1.7 on how to use the adjoint to invert transform normals.

The purpose of the inverse computation can also be taken into account when optimizing. For example, if the inverse is to be used for transforming vectors, then only the 3×3 upper left part of the matrix normally needs to be inverted (see the previous section).

4.2 Special Matrix Transforms and Operations

In this section, several matrix transforms and operations that are essential to real-time graphics will be introduced and derived. First, we present the Euler transform (along with its extraction of parameters), which is an intuitive way to describe orientations. Then we touch upon retrieving a set of basic transforms from a single matrix. Finally, a method is derived that rotates an entity around an arbitrary axis.

4.2.1 The Euler Transform

This transform is an intuitive way to construct a matrix to orient yourself (i.e., the camera) or any other entity in a certain direction. Its name comes from the great Swiss mathematician Leonhard Euler (1707–1783).

First, some kind of default view direction must be established. Most often it lies along the negative z-axis with the head oriented along the y-axis, as depicted in Figure 4.7. The Euler transform is the multiplication of three matrices, namely the rotations shown in the figure. More formally, the transform, denoted $\mathbf{E}$, is given by Equation 4.21:

$$\mathbf{E}(h,p,r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h). \tag{4.21}$$

The order of the matrices can be chosen in 24 different ways [1636]; we present this one because it is commonly used. Since $\mathbf{E}$ is a concatenation of rotations, it is also clearly orthogonal. Therefore its inverse can be expressed as $\mathbf{E}^{-1} = \mathbf{E}^T = (\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y)^T = \mathbf{R}_y^T\mathbf{R}_x^T\mathbf{R}_z^T$, although it is, of course, easier to use the transpose of $\mathbf{E}$ directly.

The Euler angles h, p, and r represent in which order and how much the head, pitch, and roll should rotate around their respective axes. Sometimes the angles are all called "rolls," e.g., our "head" is the "y-roll" and our "pitch" is the "x-roll." Also, "head" is sometimes known as "yaw," such as in flight simulation.

This transform is intuitive and therefore easy to discuss in layperson's language. For example, changing the head angle makes the viewer shake their head "no," changing the pitch makes them nod, and rolling makes them tilt their head sideways. Rather than talking about rotations around the x-, y-, and z-axes, we talk about altering the head, pitch, and roll. Note that this transform can orient not only the camera, but also any object or entity as well. These transforms can be performed using the global axes of the world space or relative to a local frame of reference.

It is important to note that some presentations of Euler angles give the z-axis as the initial up direction. This difference is purely a notational change, though a potentially confusing one. In computer graphics there is a division in how the world is regarded and thus how content is formed: y-up or z-up. Most manufacturing processes, including 3D printing, consider the z-direction to be up in world space; aviation and sea vehicles consider $-z$ to be up. Architecture and GIS normally use z-up, as a building plan or map is two-dimensional, x and y. Media-related modeling systems often consider the y-direction as up in world coordinates, matching how we always describe a camera's screen up direction in computer graphics. The difference between

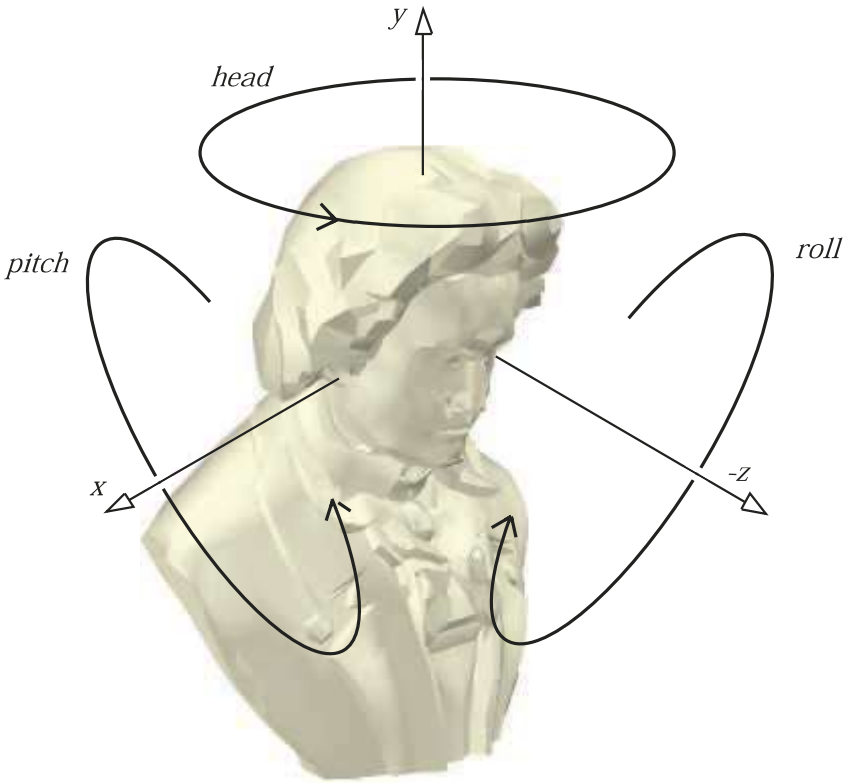

Figure 4.7. The Euler transform and how it relates to the way you change the *head*, *pitch*, and *roll* angles. The default view direction is shown, looking along the negative z-axis with the up direction along the y-axis.

these two world up vector choices is just a 90° rotation (and possibly a reflection) away, but not knowing which is assumed can lead to problems. In this volume we use a world direction of y-up unless otherwise noted.

We also want to point out that the camera's up direction in its view space has nothing in particular to do with the world's up direction. Roll your head and the view is tilted, with its world-space up direction differing from the world's. As another example, say the world uses y-up and our camera looks straight down at the terrain below, a bird's eye view. This orientation means the camera has pitched 90° forward, so that its up direction in world space is $(0, 0, -1)$. In this orientation the camera has no y-component and instead considers $-z$ to be up in world space, but "y is up" remains true in view space, by definition.

While useful for small angle changes or viewer orientation, Euler angles have some other serious limitations. It is difficult to work with two sets of Euler angles in combi-

nation. For example, interpolation between one set and another is not a simple matter of interpolating each angle. In fact, two different sets of Euler angles can give the same orientation, so any interpolation should not rotate the object at all. These are some of the reasons that using alternate orientation representations such as quaternions, discussed later in this chapter, are worth pursuing. With Euler angles, you can also get something called gimbal lock, which will be explained next in Section 4.2.2.

4.2.2 Extracting Parameters from the Euler Transform

In some situations, it is useful to have a procedure that extracts the Euler parameters, h, p, and r, from an orthogonal matrix. This procedure is shown in Equation 4.22:

$$\mathbf{E}(h,p,r) = \begin{pmatrix} e_{00} & e_{01} & e_{02} \\ e_{10} & e_{11} & e_{12} \\ e_{20} & e_{21} & e_{22} \end{pmatrix} = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h). \tag{4.22}$$

Here we abandoned the 4×4 matrices for 3×3 matrices, since the latter provide all the necessary information for a rotation matrix. That is, the rest of the equivalent 4×4 matrix always contains zeros and a one in the lower right position.

Concatenating the three rotation matrices in Equation 4.22 yields

$$\mathbf{E} = \begin{pmatrix} \cos r \cos h - \sin r \sin p \sin h & -\sin r \cos p & \cos r \sin h + \sin r \sin p \cos h \\ \sin r \cos h + \cos r \sin p \sin h & \cos r \cos p & \sin r \sin h - \cos r \sin p \cos h \\ -\cos p \sin h & \sin p & \cos p \cos h \end{pmatrix}. \tag{4.23}$$

From this it is apparent that the pitch parameter is given by $\sin p = e_{21}$. Also, dividing e_{01} by e_{11}, and similarly dividing e_{20} by e_{22}, gives rise to the following extraction equations for the head and roll parameters:

$$\frac{e_{01}}{e_{11}} = \frac{-\sin r}{\cos r} = -\tan r \quad \text{and} \quad \frac{e_{20}}{e_{22}} = \frac{-\sin h}{\cos h} = -\tan h. \tag{4.24}$$

Thus, the Euler parameters h (head), p (pitch), and r (roll) are extracted from a matrix $\mathbf{E}$ using the function `atan2(y,x)` (see page 8 in Chapter 1) as in Equation 4.25:

$$\begin{aligned} h &= \texttt{atan2}(-e_{20}, e_{22}), \\ p &= \arcsin(e_{21}), \\ r &= \texttt{atan2}(-e_{01}, e_{11}). \end{aligned} \tag{4.25}$$

However, there is a special case we need to handle. If $\cos p = 0$, we have gimbal lock (Section 4.2.2) and rotation angles r and h will rotate around the same axis (though possibly in different directions, depending on whether the p rotation angle was $-\pi/2$ or $\pi/2$), so only one angle needs to be derived. If we arbitrarily set $h = 0$ [1769], we get

$$\mathbf{E} = \begin{pmatrix} \cos r & \sin r \cos p & \sin r \sin p \\ \sin r & \cos r \cos p & -\cos r \sin p \\ 0 & \sin p & \cos p \end{pmatrix}. \tag{4.26}$$

Since p does not affect the values in the first column, when $\cos p = 0$ we can use $\sin r / \cos r = \tan r = e_{10}/e_{00}$, which gives $r = \texttt{atan2}(e_{10}, e_{00})$.

Note that from the definition of arcsin, $-\pi/2 \leq p \leq \pi/2$, which means that if $\mathbf{E}$ was created with a value of p outside this interval, the original parameter cannot be extracted. That h, p, and r are not unique means that more than one set of the Euler parameters can be used to yield the same transform. More about Euler angle conversion can be found in Shoemake's 1994 article [1636]. The simple method outlined above can result in problems with numerical instability, which is avoidable at some cost in speed [1362].

When you use Euler transforms, something called *gimbal lock* may occur [499, 1633]. This happens when rotations are made so that one degree of freedom is lost. For example, say the order of transforms is $x/y/z$. Consider a rotation of $\pi/2$ around just the y-axis, the second rotation performed. Doing so rotates the local z-axis to be aligned with the original x-axis, so that the final rotation around z is redundant.

Mathematically, we have already seen gimbal lock in Equation 4.26, where we assumed $\cos p = 0$, i.e., $p = \pm\pi/2 + 2\pi k$, where k is an integer. With such a value of p, we have lost one degree of freedom since the matrix only depends on one angle, $r + h$ or $r - h$ (but not both at the same time).

While Euler angles are commonly presented as being in $x/y/z$ order in modeling systems, a rotation around each local axis, other orderings are feasible. For example, $z/x/y$ is used in animation and $z/x/z$ in both animation and physics. All are valid ways of specifying three separate rotations. This last ordering, $z/x/z$, can be superior for some applications, as only when rotating π radians around x (a half-rotation) does gimbal lock occur. There is no perfect sequence that avoids gimbal lock. Euler angles nonetheless are commonly used, as animators prefer curve editors to specify how angles change over time [499].

Example: Constraining a Transform. Imagine you are holding a (virtual) wrench that is gripping a bolt. To get the bolt into place, you have to rotate the wrench around the x-axis. Now assume that your input device (mouse, VR gloves, space-ball, etc.) gives you a rotation matrix, i.e., a rotation, for the movement of the wrench. The problem is that it is likely to be wrong to apply this transform to the wrench, which should rotate around only the x-axis. To restrict the input transform, called $\mathbf{P}$, to be a rotation around the x-axis, simply extract the Euler angles, h, p, and r, using the method described in this section, and then create a new matrix $\mathbf{R}_x(p)$. This is then the sought-after transform that will rotate the wrench around the x-axis (if $\mathbf{P}$ now contains such a movement). □

4.2.3 Matrix Decomposition

Up to this point we have been working under the assumption that we know the origin and history of the transformation matrix we are using. This is often not the case.

For example, nothing more than a concatenated matrix may be associated with some transformed object. The task of retrieving various transforms from a concatenated matrix is called *matrix decomposition*.

There are many reasons to retrieve a set of transformations. Uses include:

- Extracting just the scaling factors for an object.
- Finding transforms needed by a particular system. (For example, some systems may not allow the use of an arbitrary 4×4 matrix.)
- Determining whether a model has undergone only rigid-body transforms.
- Interpolating between keyframes in an animation where only the matrix for the object is available.
- Removing shears from a rotation matrix.

We have already presented two decompositions, those of deriving the translation and rotation matrix for a rigid-body transformation (Section 4.1.6) and deriving the Euler angles from an orthogonal matrix (Section 4.2.2).

As we have seen, it is trivial to retrieve the translation matrix, as we simply need the elements in the last column of the 4×4 matrix. We can also determine if a reflection has occurred by checking whether the determinant of the matrix is negative. To separate out the rotation, scaling, and shears takes more determined effort.

Fortunately, there are several articles on this topic, as well as code available online. Thomas [1769] and Goldman [552, 553] each present somewhat different methods for various classes of transformations. Shoemake [1635] improves upon their techniques for affine matrices, as his algorithm is independent of frame of reference and attempts to decompose the matrix to obtain rigid-body transforms.

4.2.4 Rotation about an Arbitrary Axis

Sometimes it is convenient to have a procedure that rotates an entity by some angle around an arbitrary axis. Assume that the rotation axis, $\mathbf{r}$, is normalized and that a transform should be created that rotates α radians around $\mathbf{r}$.

To do this, we first transform to a space where the axis around which we want to rotate is the x-axis. This is done with a rotation matrix, called $\mathbf{M}$. Then the actual rotation is performed, and we transform back using $\mathbf{M}^{-1}$ [314]. This procedure is illustrated in Figure 4.8.

To compute $\mathbf{M}$, we need to find two axes that are orthonormal both to $\mathbf{r}$ and to each other. We concentrate on finding the second axis, $\mathbf{s}$, knowing that the third axis, $\mathbf{t}$, will be the cross product of the first and the second axis, $\mathbf{t} = \mathbf{r} \times \mathbf{s}$. A numerically stable way to do this is to find the smallest component (in absolute value) of $\mathbf{r}$, and set it to 0. Swap the two remaining components, and then negate the first of them

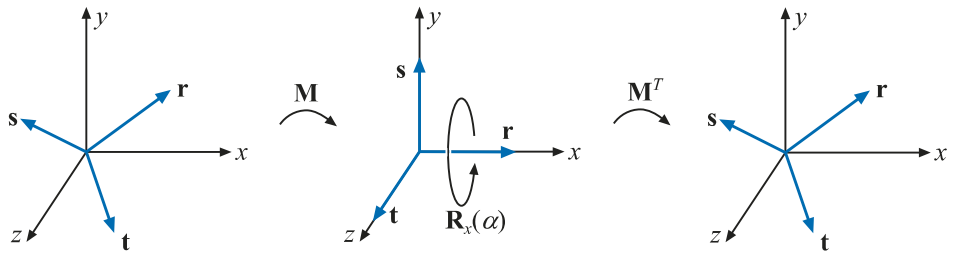

Figure 4.8. Rotation about an arbitrary axis, $\mathbf{r}$, is accomplished by finding an orthonormal basis formed by $\mathbf{r}$, $\mathbf{s}$, and $\mathbf{t}$. We then align this basis with the standard basis so that $\mathbf{r}$ is aligned with the x-axis. The rotation around the x-axis is performed there, and finally we transform back.

(in fact, either of the nonzero components could be negated). Mathematically, this is expressed as [784]:

$$\begin{aligned}
\bar{\mathbf{s}} &= \begin{cases} (0, -r_z, r_y), & \text{if } |r_x| \leq |r_y| \text{ and } |r_x| \leq |r_z|, \\ (-r_z, 0, r_x), & \text{if } |r_y| \leq |r_x| \text{ and } |r_y| \leq |r_z|, \\ (-r_y, r_x, 0), & \text{if } |r_z| \leq |r_x| \text{ and } |r_z| \leq |r_y|, \end{cases} \\
\mathbf{s} &= \bar{\mathbf{s}}/||\bar{\mathbf{s}}||, \\
\mathbf{t} &= \mathbf{r} \times \mathbf{s}.
\end{aligned} \tag{4.27}$$

This guarantees that $\bar{\mathbf{s}}$ is orthogonal (perpendicular) to $\mathbf{r}$, and that $(\mathbf{r}, \mathbf{s}, \mathbf{t})$ is an orthonormal basis. Frisvad [496] presents a method without any branches in the code, which is faster but has lower accuracy. Max [1147] and Duff et al. [388] improve the accuracy of Frisvad's method. Whichever technique is employed, these three vectors are used to create a rotation matrix:

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix}. \tag{4.28}$$

This matrix transforms the vector $\mathbf{r}$ into the x-axis, $\mathbf{s}$ into the y-axis, and $\mathbf{t}$ into the z-axis. So, the final transform for rotating α radians around the normalized vector $\mathbf{r}$ is then

$$\mathbf{X} = \mathbf{M}^T \mathbf{R}_x(\alpha) \mathbf{M}. \tag{4.29}$$

In words, this means that first we transform so that $\mathbf{r}$ is the x-axis (using $\mathbf{M}$), then we rotate α radians around this x-axis (using $\mathbf{R}_x(\alpha)$), and then we transform back using the inverse of $\mathbf{M}$, which in this case is $\mathbf{M}^T$ because $\mathbf{M}$ is orthogonal.

Another method for rotating around an arbitrary, normalized axis $\mathbf{r}$ by ϕ radians has been presented by Goldman [550]. Here, we simply present his transform:

$$\mathbf{R} = \begin{pmatrix} \cos\phi + (1-\cos\phi)r_x^2 & (1-\cos\phi)r_x r_y - r_z \sin\phi & (1-\cos\phi)r_x r_z + r_y \sin\phi \\ (1-\cos\phi)r_x r_y + r_z \sin\phi & \cos\phi + (1-\cos\phi)r_y^2 & (1-\cos\phi)r_y r_z - r_x \sin\phi \\ (1-\cos\phi)r_x r_z - r_y \sin\phi & (1-\cos\phi)r_y r_z + r_x \sin\phi & \cos\phi + (1-\cos\phi)r_z^2 \end{pmatrix}. \tag{4.30}$$

In Section 4.3.2, we present yet another method for solving this problem, using quaternions. Also in that section are more efficient algorithms for related problems, such as rotation from one vector to another.

4.3 Quaternions

Although quaternions were invented back in 1843 by Sir William Rowan Hamilton as an extension to the complex numbers, it was not until 1985 that Shoemake [1633] introduced them to the field of computer graphics.[1] Quaternions are used to represent rotations and orientations. They are superior to both Euler angles and matrices in several ways. Any three-dimensional orientation can be expressed as a single rotation around a particular axis. Given this axis & angle representation, translating to or from a quaternion is straightforward, while Euler angle conversion in either direction is challenging. Quaternions can be used for stable and constant interpolation of orientations, something that cannot be done well with Euler angles.

A complex number has a real and an imaginary part. Each is represented by two real numbers, the second real number being multiplied by $\sqrt{-1}$. Similarly, quaternions have four parts. The first three values are closely related to axis of rotation, with the angle of rotation affecting all four parts (more about this in Section 4.3.2). Each quaternion is represented by four real numbers, each associated with a different part. Since quaternions have four components, we choose to represent them as vectors, but to differentiate them, we put a hat on them: $\hat{\mathbf{q}}$. We begin with some mathematical background on quaternions, which is then used to construct a variety of useful transforms.

4.3.1 Mathematical Background

We start with the definition of a quaternion.

Definition. A quaternion $\hat{\mathbf{q}}$ can be defined in the following ways, all equivalent.

$$\begin{aligned} \hat{\mathbf{q}} &= (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w, \\ \mathbf{q}_v &= iq_x + jq_y + kq_z = (q_x, q_y, q_z), \\ i^2 &= j^2 = k^2 = -1, \; jk = -kj = i, \; ki = -ik = j, \; ij = -ji = k. \end{aligned} \tag{4.31}$$

The variable q_w is called the real part of a quaternion, $\hat{\mathbf{q}}$. The imaginary part is $\mathbf{q}_v$, and i, j, and k are called imaginary units. □

For the imaginary part, $\mathbf{q}_v$, we can use all the normal vector operations, such as addition, scaling, dot product, cross product, and more. Using the definition of the quaternion, the multiplication operation between two quaternions, $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, is derived

[1] In fairness, Robinson [1502] used quaternions in 1958 for rigid-body simulations.

as shown below. Note that the multiplication of the imaginary units is noncommutative.

$$\begin{aligned}\textbf{Multiplication:}\quad \hat{\mathbf{q}}\hat{\mathbf{r}} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w) \\ &= i(q_y r_z - q_z r_y + r_w q_x + q_w r_x) \\ &+\ j(q_z r_x - q_x r_z + r_w q_y + q_w r_y) \\ &+\ k(q_x r_y - q_y r_x + r_w q_z + q_w r_z) \\ &+\ q_w r_w - q_x r_x - q_y r_y - q_z r_z \\ &= (\mathbf{q}_v \times \mathbf{r}_v + r_w \mathbf{q}_v + q_w \mathbf{r}_v,\ q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v).\end{aligned} \tag{4.32}$$

As can be seen in this equation, we use both the cross product and the dot product to compute the multiplication of two quaternions.

Along with the definition of the quaternion, the definitions of addition, conjugate, norm, and an identity are needed:

Addition: $\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v, q_w) + (\mathbf{r}_v, r_w) = (\mathbf{q}_v + \mathbf{r}_v, q_w + r_w).$

Conjugate: $\hat{\mathbf{q}}^* = (\mathbf{q}_v, q_w)^* = (-\mathbf{q}_v, q_w).$

$$\begin{aligned}\textbf{Norm:}\quad n(\hat{\mathbf{q}}) &= \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*} = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \sqrt{\mathbf{q}_v \cdot \mathbf{q}_v + q_w^2} \\ &= \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}.\end{aligned} \tag{4.33}$$

Identity: $\hat{\mathbf{i}} = (\mathbf{0}, 1).$

When $n(\hat{\mathbf{q}}) = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}$ is simplified (result shown above), the imaginary parts cancel out and only a real part remains. The norm is sometimes denoted $||\hat{\mathbf{q}}|| = n(\hat{\mathbf{q}})$ [1105]. A consequence of the above is that a multiplicative inverse, denoted by $\hat{\mathbf{q}}^{-1}$, can be derived. The equation $\hat{\mathbf{q}}^{-1}\hat{\mathbf{q}} = \hat{\mathbf{q}}\hat{\mathbf{q}}^{-1} = 1$ must hold for the inverse (as is common for a multiplicative inverse). We derive a formula from the definition of the norm:

$$n(\hat{\mathbf{q}})^2 = \hat{\mathbf{q}}\hat{\mathbf{q}}^* \iff \frac{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})^2} = 1. \tag{4.34}$$

This gives the multiplicative inverse as shown below:

$$\textbf{Inverse:}\quad \hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})^2}\hat{\mathbf{q}}^*. \tag{4.35}$$

The formula for the inverse uses scalar multiplication, which is an operation derived from the multiplication seen in Equation 4.3.1: $s\hat{\mathbf{q}} = (\mathbf{0},\ s)(\mathbf{q}_v,\ q_w) = (s\mathbf{q}_v,\ sq_w)$, and $\hat{\mathbf{q}}s = (\mathbf{q}_v,\ q_w)(\mathbf{0},\ s) = (s\mathbf{q}_v,\ sq_w)$, which means that scalar multiplication is commutative: $s\hat{\mathbf{q}} = \hat{\mathbf{q}}s = (s\mathbf{q}_v,\ sq_w)$.

The following collection of rules are simple to derive from the definitions:

Conjugate rules:
$$\begin{aligned}(\hat{\mathbf{q}}^*)^* &= \hat{\mathbf{q}},\\ (\hat{\mathbf{q}}+\hat{\mathbf{r}})^* &= \hat{\mathbf{q}}^* + \hat{\mathbf{r}}^*,\\ (\hat{\mathbf{q}}\hat{\mathbf{r}})^* &= \hat{\mathbf{r}}^*\hat{\mathbf{q}}^*.\end{aligned} \tag{4.36}$$

Norm rules:
$$\begin{aligned}n(\hat{\mathbf{q}}^*) &= n(\hat{\mathbf{q}}),\\ n(\hat{\mathbf{q}}\hat{\mathbf{r}}) &= n(\hat{\mathbf{q}})n(\hat{\mathbf{r}}).\end{aligned} \tag{4.37}$$

Laws of Multiplication:

Linearity:
$$\begin{aligned}\hat{\mathbf{p}}(s\hat{\mathbf{q}}+t\hat{\mathbf{r}}) &= s\hat{\mathbf{p}}\hat{\mathbf{q}} + t\hat{\mathbf{p}}\hat{\mathbf{r}},\\ (s\hat{\mathbf{p}}+t\hat{\mathbf{q}})\hat{\mathbf{r}} &= s\hat{\mathbf{p}}\hat{\mathbf{r}} + t\hat{\mathbf{q}}\hat{\mathbf{r}}.\end{aligned} \tag{4.38}$$

Associativity:
$$\hat{\mathbf{p}}(\hat{\mathbf{q}}\hat{\mathbf{r}}) = (\hat{\mathbf{p}}\hat{\mathbf{q}})\hat{\mathbf{r}}.$$

A unit quaternion, $\hat{\mathbf{q}} = (\mathbf{q}_v,\ q_w)$, is such that $n(\hat{\mathbf{q}}) = 1$. From this it follows that $\hat{\mathbf{q}}$ may be written as

$$\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q,\ \cos\phi) = \sin\phi\mathbf{u}_q + \cos\phi, \tag{4.39}$$

for some three-dimensional vector $\mathbf{u}_q$, such that $||\mathbf{u}_q|| = 1$, because

$$\begin{aligned}n(\hat{\mathbf{q}}) = n(\sin\phi\mathbf{u}_q,\ \cos\phi) &= \sqrt{\sin^2\phi(\mathbf{u}_q\cdot\mathbf{u}_q) + \cos^2\phi}\\ &= \sqrt{\sin^2\phi + \cos^2\phi} = 1\end{aligned} \tag{4.40}$$

if and only if $\mathbf{u}_q\cdot\mathbf{u}_q = 1 = ||\mathbf{u}_q||^2$. As will be seen in the next section, unit quaternions are perfectly suited for creating rotations and orientations in a most efficient way. But before that, some extra operations will be introduced for unit quaternions.

For complex numbers, a two-dimensional unit vector can be written as $\cos\phi + i\sin\phi = e^{i\phi}$. The equivalent for quaternions is

$$\hat{\mathbf{q}} = \sin\phi\mathbf{u}_q + \cos\phi = e^{\phi\mathbf{u}_q}. \tag{4.41}$$

The log and the power functions for unit quaternions follow from Equation 4.41:

Logarithm:
$$\log(\hat{\mathbf{q}}) = \log(e^{\phi\mathbf{u}_q}) = \phi\mathbf{u}_q, \tag{4.42}$$

Power:
$$\hat{\mathbf{q}}^t = (\sin\phi\mathbf{u}_q + \cos\phi)^t = e^{\phi t\mathbf{u}_q} = \sin(\phi t)\mathbf{u}_q + \cos(\phi t).$$

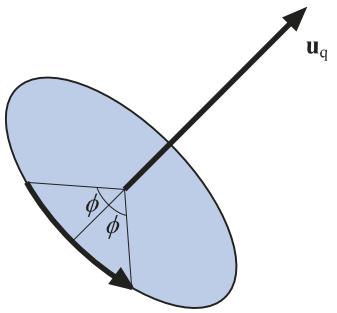

Figure 4.9. Illustration of the rotation transform represented by a unit quaternion, $\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q,\ \cos\phi)$. The transform rotates 2ϕ radians around the axis $\mathbf{u}_q$.

4.3.2 Quaternion Transforms

We will now study a subclass of the quaternion set, namely those of unit length, called *unit quaternions*. The most important fact about unit quaternions is that they can represent any three-dimensional rotation, and that this representation is extremely compact and simple.

Now we will describe what makes unit quaternions so useful for rotations and orientations. First, put the four coordinates of a point or vector $\mathbf{p} = (p_x\ \ p_y\ \ p_z\ \ p_w)^T$ into the components of a quaternion $\hat{\mathbf{p}}$, and assume that we have a unit quaternion $\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q,\ \cos\phi)$. One can prove that

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1} \tag{4.43}$$

rotates $\hat{\mathbf{p}}$ (and thus the point $\mathbf{p}$) around the axis $\mathbf{u}_q$ by an angle 2ϕ. Note that since $\hat{\mathbf{q}}$ is a unit quaternion, $\hat{\mathbf{q}}^{-1} = \hat{\mathbf{q}}^*$. See Figure 4.9.

Any nonzero real multiple of $\hat{\mathbf{q}}$ also represents the same transform, which means that $\hat{\mathbf{q}}$ and $-\hat{\mathbf{q}}$ represent the same rotation. That is, negating the axis, $\mathbf{u}_q$, and the real part, q_w, creates a quaternion that rotates exactly as the original quaternion does. It also means that the extraction of a quaternion from a matrix can return either $\hat{\mathbf{q}}$ or $-\hat{\mathbf{q}}$.

Given two unit quaternions, $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, the concatenation of first applying $\hat{\mathbf{q}}$ and then $\hat{\mathbf{r}}$ to a quaternion, $\hat{\mathbf{p}}$ (which can be interpreted as a point $\mathbf{p}$), is given by Equation 4.44:

$$\hat{\mathbf{r}}(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^* = \hat{\mathbf{c}}\hat{\mathbf{p}}\hat{\mathbf{c}}^*. \tag{4.44}$$

Here, $\hat{\mathbf{c}} = \hat{\mathbf{r}}\hat{\mathbf{q}}$ is the unit quaternion representing the concatenation of the unit quaternions $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$.

Matrix Conversion

Since one often needs to combine several different transforms, and most of them are in matrix form, a method is needed to convert Equation 4.43 into a matrix. A quaternion,

$\hat{\mathbf{q}}$, can be converted into a matrix $\mathbf{M}^q$, as expressed in Equation 4.45 [1633, 1634]:

$$\mathbf{M}^q = \begin{pmatrix} 1 - s(q_y^2 + q_z^2) & s(q_x q_y - q_w q_z) & s(q_x q_z + q_w q_y) & 0 \\ s(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & s(q_y q_z - q_w q_x) & 0 \\ s(q_x q_z - q_w q_y) & s(q_y q_z + q_w q_x) & 1 - s(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.45}$$

Here, the scalar is $s = 2/(n(\hat{\mathbf{q}}))^2$. For unit quaternions, this simplifies to

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.46}$$

Once the quaternion is constructed, *no* trigonometric functions need to be computed, so the conversion process is efficient in practice.

The reverse conversion, from an orthogonal matrix, $\mathbf{M}^q$, into a unit quaternion, $\hat{\mathbf{q}}$, is a bit more involved. Key to this process are the following differences made from the matrix in Equation 4.46:

$$\begin{aligned} m_{21}^q - m_{12}^q &= 4q_w q_x, \\ m_{02}^q - m_{20}^q &= 4q_w q_y, \\ m_{10}^q - m_{01}^q &= 4q_w q_z. \end{aligned} \tag{4.47}$$

The implication of these equations is that if q_w is known, the values of the vector $\mathbf{v}_q$ can be computed, and thus $\hat{\mathbf{q}}$ derived. The trace of $\mathbf{M}^q$ is calculated by

$$\begin{aligned} \operatorname{tr}(\mathbf{M}^q) &= 4 - 2s(q_x^2 + q_y^2 + q_z^2) = 4\left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2}\right) \\ &= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{(n(\hat{\mathbf{q}}))^2}. \end{aligned} \tag{4.48}$$

This result yields the following conversion for a unit quaternion:

$$\begin{aligned} q_w &= \frac{1}{2}\sqrt{\operatorname{tr}(\mathbf{M}^q)}, & q_x &= \frac{m_{21}^q - m_{12}^q}{4q_w}, \\ q_y &= \frac{m_{02}^q - m_{20}^q}{4q_w}, & q_z &= \frac{m_{10}^q - m_{01}^q}{4q_w}. \end{aligned} \tag{4.49}$$

To have a numerically stable routine [1634], divisions by small numbers should be avoided. Therefore, first set $t = q_w^2 - q_x^2 - q_y^2 - q_z^2$, from which it follows that

$$\begin{aligned} m_{00} &= t + 2q_x^2, \\ m_{11} &= t + 2q_y^2, \\ m_{22} &= t + 2q_z^2, \\ u &= m_{00} + m_{11} + m_{22} = t + 2q_w^2, \end{aligned} \tag{4.50}$$

which in turn implies that the largest of m_{00}, m_{11}, m_{22}, and u determine which of q_x, q_y, q_z, and q_w is largest. If q_w is largest, then Equation 4.49 is used to derive the quaternion. Otherwise, we note that the following holds:

$$\begin{aligned} 4q_x^2 &= +m_{00} - m_{11} - m_{22} + m_{33}, \\ 4q_y^2 &= -m_{00} + m_{11} - m_{22} + m_{33}, \\ 4q_z^2 &= -m_{00} - m_{11} + m_{22} + m_{33}, \\ 4q_w^2 &= \mathrm{tr}(\mathbf{M}^q). \end{aligned} \tag{4.51}$$

The appropriate equation of the ones above is then used to compute the largest of q_x, q_y, and q_z, after which Equation 4.47 is used to calculate the remaining components of $\hat{\mathbf{q}}$. Schüler [1588] presents a variant that is branchless but uses four square roots instead.

Spherical Linear Interpolation

Spherical linear interpolation is an operation that, given two unit quaternions, $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, and a parameter $t \in [0, 1]$, computes an interpolated quaternion. This is useful for animating objects, for example. It is not as useful for interpolating camera orientations, as the camera's "up" vector can become tilted during the interpolation, usually a disturbing effect.

The algebraic form of this operation is expressed by the composite quaternion, $\hat{\mathbf{s}}$, below:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t\hat{\mathbf{q}}. \tag{4.52}$$

However, for software implementations, the following form, where *slerp* stands for spherical linear interpolation, is much more appropriate:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \mathtt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin\phi}\hat{\mathbf{r}}. \tag{4.53}$$

To compute ϕ, which is needed in this equation, the following fact can be used: $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ [325]. For $t \in [0, 1]$, the slerp function computes (unique[2]) interpolated quaternions that together constitute the shortest arc on a four-dimensional unit sphere from $\hat{\mathbf{q}}$ ($t = 0$) to $\hat{\mathbf{r}}$ ($t = 1$). The arc is located on the circle that is formed from the intersection between the plane given by $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, and the origin, and the four-dimensional unit sphere. This is illustrated in Figure 4.10. The computed rotation quaternion rotates around a fixed axis at constant speed. A curve such as this, that has constant speed and thus zero acceleration, is called a *geodesic* curve [229]. A *great circle* on the sphere is generated as the intersection of a plane through the origin and the sphere, and part of such a circle is called a *great arc*.

The slerp function is perfectly suited for interpolating between two orientations and it behaves well (fixed axis, constant speed). This is not the case with when

[2] If and only if $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$ are not opposite.

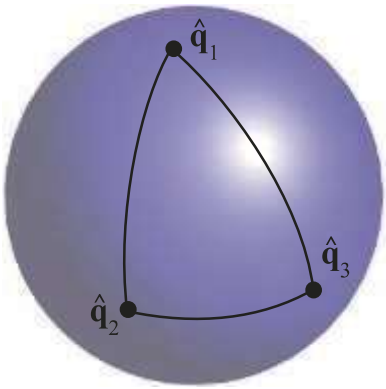

Figure 4.10. Unit quaternions are represented as points on the unit sphere. The function slerp is used to interpolate between the quaternions, and the interpolated path is a great arc on the sphere. Note that interpolating from $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_2$ and interpolating from $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_3$ to $\hat{\mathbf{q}}_2$ are not the same thing, even though they arrive at the same orientation.

interpolating using several Euler angles. In practice, computing a slerp directly is an expensive operation involving calling trigonometric functions. Malyshau [1114] discusses integrating quaternions into the rendering pipeline. He notes that the error in the orientation of a triangle is a maximum of 4 degrees for a 90 degree angle when, instead of using slerp, simply normalizing the quaternion in the pixel shader. This error rate can be acceptable when rasterizing a triangle. Li [1039, 1040] provides much faster incremental methods to compute slerps that do not sacrifice any accuracy. Eberly [406] presents a fast technique for computing slerps using just additions and multiplications.

When more than two orientations, say $\hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_{n-1}$, are available, and we want to interpolate from $\hat{\mathbf{q}}_0$ to $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_2$, and so on until $\hat{\mathbf{q}}_{n-1}$, slerp could be used in a straightforward fashion. Now, when we approach, say, $\hat{\mathbf{q}}_i$, we would use $\hat{\mathbf{q}}_{i-1}$ and $\hat{\mathbf{q}}_i$ as arguments to slerp. After passing through $\hat{\mathbf{q}}_i$, we would then use $\hat{\mathbf{q}}_i$ and $\hat{\mathbf{q}}_{i+1}$ as arguments to slerp. This will cause sudden jerks to appear in the orientation interpolation, which can be seen in Figure 4.10. This is similar to what happens when points are linearly interpolated; see the upper right part of Figure 17.3 on page 720. Some readers may wish to revisit the following paragraph after reading about splines in Chapter 17.

A better way to interpolate is to use some sort of spline. We introduce quaternions $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{a}}_{i+1}$ between $\hat{\mathbf{q}}_i$ and $\hat{\mathbf{q}}_{i+1}$. Spherical cubic interpolation can be defined within the set of quaternions $\hat{\mathbf{q}}_i$, $\hat{\mathbf{a}}_i$, $\hat{\mathbf{a}}_{i+1}$, and $\hat{\mathbf{q}}_{i+1}$. Surprisingly, these extra quaternions are computed as shown below [404][3]:

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \exp\left[-\frac{\log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i-1}) + \log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i+1})}{4}\right]. \tag{4.54}$$

[3]Shoemake [1633] gives another derivation.

The $\hat{\mathbf{q}}_i$, and $\hat{\mathbf{a}}_i$ will be used to spherically interpolate the quaternions using a smooth cubic spline, as shown in Equation 4.55:

$$\begin{aligned}&\texttt{squad}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t) = \\ &\quad \texttt{slerp}(\texttt{slerp}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t), \texttt{slerp}(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t), 2t(1-t)).\end{aligned} \tag{4.55}$$

As can be seen above, the `squad` function is constructed from repeated spherical interpolation using slerp (Section 17.1.1 for information on repeated linear interpolation for points). The interpolation will pass through the initial orientations $\hat{\mathbf{q}}_i$, $i \in [0, \ldots, n-1]$, but not through $\hat{\mathbf{a}}_i$—these are used to indicate the tangent orientations at the initial orientations.

Rotation from One Vector to Another

A common operation is transforming from one direction $\mathbf{s}$ to another direction $\mathbf{t}$ via the shortest path possible. The mathematics of quaternions simplifies this procedure greatly, and shows the close relationship the quaternion has with this representation. First, normalize $\mathbf{s}$ and $\mathbf{t}$. Then compute the unit rotation axis, called $\mathbf{u}$, which is computed as $\mathbf{u} = (\mathbf{s} \times \mathbf{t})/||\mathbf{s} \times \mathbf{t}||$. Next, $e = \mathbf{s} \cdot \mathbf{t} = \cos(2\phi)$ and $||\mathbf{s} \times \mathbf{t}|| = \sin(2\phi)$, where 2ϕ is the angle between $\mathbf{s}$ and $\mathbf{t}$. The quaternion that represents the rotation from $\mathbf{s}$ to $\mathbf{t}$ is then $\hat{\mathbf{q}} = (\sin\phi\mathbf{u},\ \cos\phi)$. In fact, simplifying $\hat{\mathbf{q}} = (\frac{\sin\phi}{\sin 2\phi}(\mathbf{s} \times \mathbf{t}),\ \cos\phi)$, using the half-angle relations and the trigonometric identity, gives [1197]

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s} \times \mathbf{t}), \frac{\sqrt{2(1+e)}}{2}\right). \tag{4.56}$$

Directly generating the quaternion in this fashion (versus normalizing the cross product $\mathbf{s} \times \mathbf{t}$) avoids numerical instability when $\mathbf{s}$ and $\mathbf{t}$ point in nearly the same direction [1197]. Stability problems appear for both methods when $\mathbf{s}$ and $\mathbf{t}$ point in opposite directions, as a division by zero occurs. When this special case is detected, any axis of rotation perpendicular to $\mathbf{s}$ can be used to rotate to $\mathbf{t}$.

Sometimes we need the matrix representation of a rotation from $\mathbf{s}$ to $\mathbf{t}$. After some algebraic and trigonometric simplification of Equation 4.46, the rotation matrix becomes [1233]

$$\mathbf{R}(\mathbf{s}, \mathbf{t}) = \begin{pmatrix} e + hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e + hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z - v_y & hv_yv_z + v_x & e + hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.57}$$

In this equation, we have used the following intermediate calculations:

$$\begin{aligned}\mathbf{v} &= \mathbf{s} \times \mathbf{t}, \\ e &= \cos(2\phi) = \mathbf{s} \cdot \mathbf{t}, \\ h &= \frac{1 - \cos(2\phi)}{\sin^2(2\phi)} = \frac{1-e}{\mathbf{v} \cdot \mathbf{v}} = \frac{1}{1+e}.\end{aligned} \tag{4.58}$$

As can be seen, all square roots and trigonometric functions have disappeared due to the simplifications, and so this is an efficient way to create the matrix. Note that the structure of Equation 4.57 is like that of Equation 4.30, and note how this latter form does not need trigonometric functions.

Note that care must be taken when $\mathbf{s}$ and $\mathbf{t}$ are parallel or near parallel, because then $||\mathbf{s} \times \mathbf{t}|| \approx 0$. If $\phi \approx 0$, then we can return the identity matrix. However, if $2\phi \approx \pi$, then we can rotate π radians around *any* axis. This axis can be found as the cross product between $\mathbf{s}$ and any other vector that is not parallel to $\mathbf{s}$ (Section 4.2.4). Möller and Hughes use Householder matrices to handle this special case in a different way [1233].

4.4 Vertex Blending

Imagine that an arm of a digital character is animated using two parts, a forearm and an upper arm, as shown to the left in Figure 4.11. This model could be animated using rigid-body transforms (Section 4.1.6). However, then the joint between these two parts will not resemble a real elbow. This is because two separate objects are used, and therefore, the joint consists of overlapping parts from these two separate objects. Clearly, it would be better to use just one single object. However, static model parts do not address the problem of making the joint flexible.

Vertex blending is one popular solution to this problem [1037, 1903]. This technique has several other names, such as *linear-blend skinning*, *enveloping*, or *skeleton-subspace*

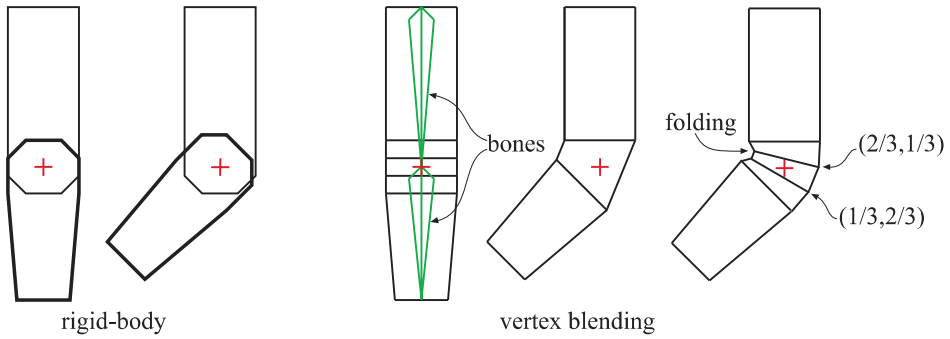

Figure 4.11. An arm consisting of a forearm and an upper arm is animated using rigid-body transforms of two separate objects to the left. The elbow does not appear realistic. To the right, vertex blending is used on one single object. The next-to-rightmost arm illustrates what happens when a simple skin directly joins the two parts to cover the elbow. The rightmost arm illustrates what happens when vertex blending is used, and some vertices are blended with different weights: $(2/3, 1/3)$ means that the vertex weighs the transform from the upper arm by $2/3$ and from the forearm by $1/3$. This figure also shows a drawback of vertex blending in the rightmost illustration. Here, folding in the inner part of the elbow is visible. Better results can be achieved with more bones, and with more carefully selected weights.

deformation. While the exact origin of the algorithm presented here is unclear, defining bones and having skin react to changes is an old concept in computer animation [1100]. In its simplest form, the forearm and the upper arm are animated separately as before, but at the joint, the two parts are connected through an elastic "skin." So, this elastic part will have one set of vertices that are transformed by the forearm matrix and another set that are transformed by the matrix of the upper arm. This results in triangles whose vertices may be transformed by different matrices, in contrast to using a single matrix per triangle. See Figure 4.11.

By taking this one step further, one can allow a single vertex to be transformed by several different matrices, with the resulting locations weighted and blended together. This is done by having a skeleton of bones for the animated object, where each bone's transform may influence each vertex by a user-defined weight. Since the entire arm may be "elastic," i.e., all vertices may be affected by more than one matrix, the entire mesh is often called a *skin* (over the bones). See Figure 4.12. Many commercial modeling systems have this same sort of skeleton-bone modeling feature. Despite their name, bones do not need to necessarily be rigid. For example, Mohr and Gleicher [1230] present the idea of adding additional joints to enable effects such as muscle bulge. James and Twigg [813] discuss animation skinning using bones that can squash and stretch.

Mathematically, this is expressed in Equation 4.59, where $\mathbf{p}$ is the original vertex, and $\mathbf{u}(t)$ is the transformed vertex whose position depends on time t:

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0. \tag{4.59}$$

There are n bones influencing the position of $\mathbf{p}$, which is expressed in world coordinates. The value w_i is the weight of bone i for vertex $\mathbf{p}$. The matrix $\mathbf{M}_i$ transforms from the initial bone's coordinate system to world coordinates. Typically a bone has its controlling joint at the origin of its coordinate system. For example, a forearm bone would move its elbow joint to the origin, with an animated rotation matrix moving this part of the arm around the joint. The $\mathbf{B}_i(t)$ matrix is the ith bone's world transform that changes with time to animate the object, and is typically a concatenation of several matrices, such as the hierarchy of previous bone transforms and the local animation matrix.

One method of maintaining and updating the $\mathbf{B}_i(t)$ matrix animation functions is discussed in depth by Woodland [1903]. Each bone transforms a vertex to a location with respect to its own frame of reference, and the final location is interpolated from the set of computed points. The matrix $\mathbf{M}_i$ is not explicitly shown in some discussions of skinning, but rather is considered as being a part of $\mathbf{B}_i(t)$. We present it here as it is a useful matrix that is almost always a part of the matrix concatenation process.

In practice, the matrices $\mathbf{B}_i(t)$ and $\mathbf{M}_i^{-1}$ are concatenated for each bone for each frame of animation, and each resulting matrix is used to transform the vertices. The vertex $\mathbf{p}$ is transformed by the different bones' concatenated matrices, and then

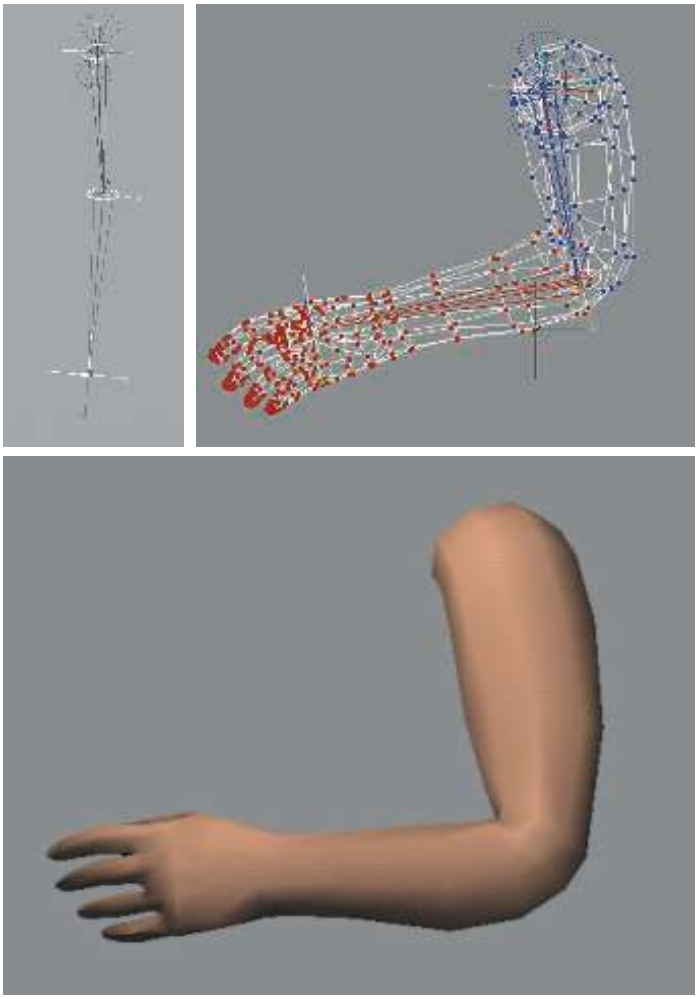

Figure 4.12. A real example of vertex blending. The top left image shows the two bones of an arm, in an extended position. On the top right, the mesh is shown, with color denoting which bone owns each vertex. Bottom: the shaded mesh of the arm in a slightly different position. *(Images courtesy of Jeff Lander [968].)*

blended using the weights w_i—thus the name *vertex blending*. The weights are non-negative and sum to one, so what is occurring is that the vertex is transformed to a few positions and then interpolated among them. As such, the transformed point $\mathbf{u}$ will lie in the convex hull of the set of points $\mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{p}$, for all $i = 0 \dots n-1$ (fixed t). The normals usually can also be transformed using Equation 4.59. Depending on the transforms used (e.g., if a bone is stretched or squished a considerable amount),

the transpose of the inverse of the $\mathbf{B}_i(t)\mathbf{M}_i^{-1}$ may be needed instead, as discussed in Section 4.1.7.

Vertex blending is well suited for use on the GPU. The set of vertices in the mesh can be placed in a static buffer that is sent to the GPU one time and reused. In each frame, only the bone matrices change, with a vertex shader computing their effect on the stored mesh. In this way, the amount of data processed on and transferred from the CPU is minimized, allowing the GPU to efficiently render the mesh. It is easiest if the model's whole set of bone matrices can be used together; otherwise the model must be split up and some bones replicated. Alternately the bone transforms can be stored in textures that the vertices access, which avoids hitting register storage limits. Each transform can be stored in just two textures by using quaternions to represent rotation [1639]. If available, unordered access view storage allows the reuse of skinning results [146].

It is possible to specify sets of weights that are outside the range $[0, 1]$ or do not sum to one. However, this makes sense only if some other blending algorithm, such as *morph targets* (Section 4.5), is being used.

One drawback of basic vertex blending is that unwanted folding, twisting, and self-intersection can occur [1037]. See Figure 4.13. A better solution is to use *dual quaternions* [872, 873]. This technique to perform skinning helps to preserve the rigidity of the original transforms, so avoiding "candy wrapper" twists in limbs. Computation is less than $1.5\times$ the cost for linear skin blending and the results are good, which has led to rapid adoption of this technique. However, dual quaternion skinning can lead to bulging effects, and Le and Hodgins [1001] present center-of-rotation skinning as a better alternative. They rely on the assumptions that local transforms should be rigid-body and that vertices with similar weights, w_i, should have similar transforms. Centers of rotation are precomputed for each vertex while orthogonal (rigid body) constraints are imposed to prevent elbow collapse and candy wrapper twist artifacts. At runtime, the algorithm is similar to linear blend skinning in that a GPU implementation performs linear blend skinning on the centers of rotation followed by a quaternion blending step.

4.5 Morphing

Morphing from one three-dimensional model to another can be useful when performing animations [28, 883, 1000, 1005]. Imagine that one model is displayed at time t_0 and we wish it to change into another model by time t_1. For all times between t_0 and t_1, a continuous "mixed" model is obtained, using some kind of interpolation. An example of morphing is shown in Figure 4.14.

Morphing involves solving two major problems, namely, the *vertex correspondence* problem and the *interpolation* problem. Given two arbitrary models, which may have different topologies, different number of vertices, and different mesh connectivity, one usually has to begin by setting up these vertex correspondences. This is a difficult

Figure 4.13. The left side shows problems at the joints when using linear blend skinning. On the right, blending using dual quaternions improves the appearance. *(Images courtesy of Ladislav Kavan et al., model by Paul Steed [1693].)*

problem, and there has been much research in this field. We refer the interested reader to Alexa's survey [28].

However, if there already is a one-to-one vertex correspondence between the two models, then interpolation can be done on a per-vertex basis. That is, for each vertex in the first model, there must exist only one vertex in the second model, and vice versa. This makes interpolation an easy task. For example, linear interpolation can be used directly on the vertices (Section 17.1 for other ways of doing interpolation). To

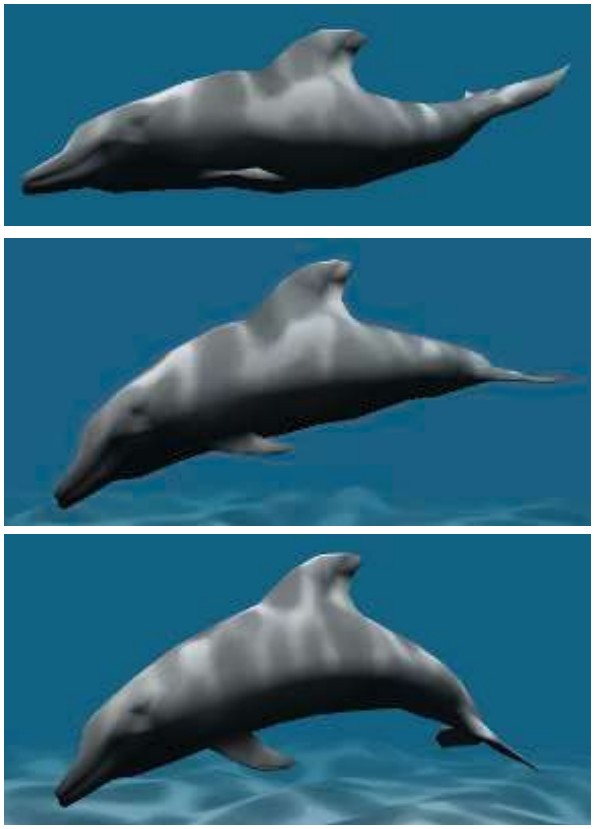

Figure 4.14. Vertex morphing. Two locations and normals are defined for every vertex. In each frame, the intermediate location and normal are linearly interpolated by the vertex shader. *(Images courtesy of NVIDIA Corporation.)*

compute a morphed vertex for time $t \in [t_0, t_1]$, we first compute $s = (t - t_0)/(t_1 - t_0)$, and then the linear vertex blend,

$$\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1, \tag{4.60}$$

where $\mathbf{p}_0$ and $\mathbf{p}_1$ correspond to the same vertex but at different times, t_0 and t_1.

A variant of morphing where the user has more intuitive control is referred to as *morph targets* or *blend shapes* [907]. The basic idea can be explained using Figure 4.15.

We start out with a neutral model, which in this case is a face. Let us denote this model by $\mathcal{N}$. In addition, we also have a set of different face poses. In the example illustration, there is only one pose, which is a smiling face. In general, we can allow $k \geq 1$ different poses, which are denoted $\mathcal{P}_i$, $i \in [1, \ldots, k]$. As a preprocess, the

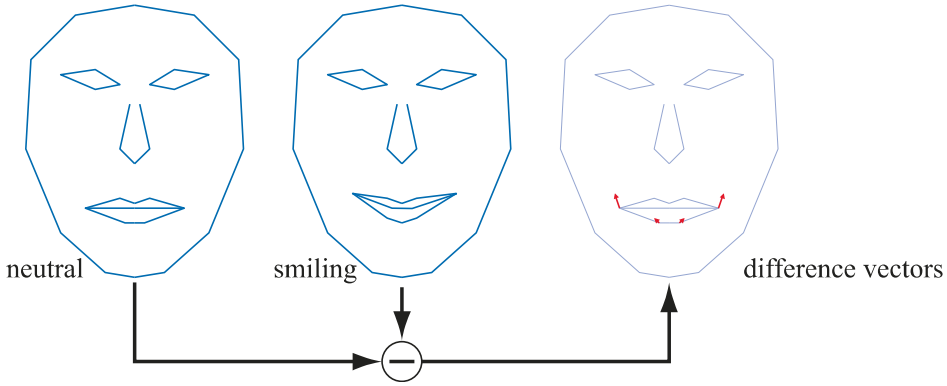

Figure 4.15. Given two mouth poses, a set of difference vectors is computed to control interpolation, or even extrapolation. In morph targets, the difference vectors are used to "add" movements onto the neutral face. With positive weights for the difference vectors, we get a smiling mouth, while negative weights can give the opposite effect.

"difference faces" are computed as: $\mathcal{D}_i = \mathcal{P}_i - \mathcal{N}$, i.e., the neutral model is subtracted from each pose.

At this point, we have a neutral model, $\mathcal{N}$, and a set of difference poses, $\mathcal{D}_i$. A morphed model $\mathcal{M}$ can then be obtained using the following formula:

$$\mathcal{M} = \mathcal{N} + \sum_{i=1}^{k} w_i \mathcal{D}_i. \tag{4.61}$$

This is the neutral model, and on top of that we add the features of the different poses as desired, using the weights, w_i. For Figure 4.15, setting $w_1 = 1$ gives us exactly the smiling face in the middle of the illustration. Using $w_1 = 0.5$ gives us a half-smiling face, and so on. One can also use negative weights and weights greater than one.

For this simple face model, we could add another face having "sad" eyebrows. Using a negative weight for the eyebrows could then create "happy" eyebrows. Since displacements are additive, this eyebrow pose could be used in conjunction with the pose for a smiling mouth.

Morph targets are a powerful technique that provides the animator with much control, since different features of a model can be manipulated independently of the others. Lewis et al. [1037] introduce *pose-space deformation*, which combines vertex blending and morph targets. Senior [1608] uses precomputed vertex textures to store and retrieve displacements between target poses. Hardware supporting stream-out and the ID of each vertex allow many more targets to be used in a single model and the effects to be computed exclusively on the GPU [841, 1074]. Using a low-resolution mesh and then generating a high-resolution mesh via the tessellation stage and displacement mapping avoids the cost of skinning every vertex in a highly detailed model [1971].

Figure 4.16. The Delsin character's face, in *inFAMOUS Second Son*, is animated using blend shapes. The same resting pose face is used for all of these shots, and then different weights are modified to make the face appear differently. *(Images provided courtesy of Naughty Dog LLC. inFAMOUS Second Son © 2014 Sony Interactive Entertainment LLC. inFAMOUS Second Son is a trademark of Sony Interactive Entertainment LLC. Developed by Sucker Punch Productions LLC.)*

A real example of using both skinning and morphing is shown in Figure 4.16. Weronko and Andreason [1872] used skinning and morphing in *The Order: 1886*.

4.6 Geometry Cache Playback

In cut scenes, it may be desirable to use extremely high-quality animations, e.g., for movements that cannot be represented using any of the methods above. A naive approach is to store all the vertices for all frames, reading them from disk and updating the mesh. However, this can amount to 50 MB/s for a simple model of 30,000 vertices used in a short animation. Gneiting [545] presents several ways to reduce memory costs down to about 10%.

First, quantization is used. For example, positions and texture coordinates are stored using 16-bit integers for each coordinate. This step is lossy in the sense that one cannot recover the original data after compression is performed. To reduce data further, spatial and temporal predictions are made and the differences encoded. For spatial compression, parallelogram prediction can be used [800]. For a triangle strip, the next vertex's predicted position is simply the current triangle reflected in the triangle's plane around the current triangle edge, which forms a parallelogram. The differences from this new position is then encoded. With good predictions, most values will be close to zero, which is ideal for many commonly used compression schemes. Similar to MPEG compression, prediction is also done in the temporal dimension. That is, every n frames, spatial compression is performed. In between, predictions are done in the temporal dimension, e.g., if a certain vertex moved by delta vector from frame $n-1$ to frame n, then it is likely to move by a similar amount to frame $n+1$. These techniques reduced storage sufficiently so this system could be used for streaming data in real time.

4.7 Projections

Before one can actually render a scene, all relevant objects in the scene must be projected onto some kind of plane or into some type of simple volume. After that, clipping and rendering are performed (Section 2.3).

The transforms seen so far in this chapter have left the fourth coordinate, the w-component, unaffected. That is, points and vectors have retained their types after the transform. Also, the bottom row in the 4×4 matrices has always been $(0 \;\; 0 \;\; 0 \;\; 1)$. *Perspective projection matrices* are exceptions to both of these properties: The bottom row contains vector and point manipulating numbers, and the homogenization process is often needed. That is, w is often not 1, so a division by w is needed to obtain the nonhomogeneous point. *Orthographic projection*, which is dealt with first in this section, is a simpler kind of projection that is also commonly used. It does not affect the w-component.

In this section, it is assumed that the viewer is looking along the camera's negative z-axis, with the y-axis pointing up and the x-axis to the right. This is a right-handed coordinate system. Some texts and software, e.g., DirectX, use a left-handed system in which the viewer looks along the camera's positive z-axis. Both systems are equally valid, and in the end, the same effect is achieved.

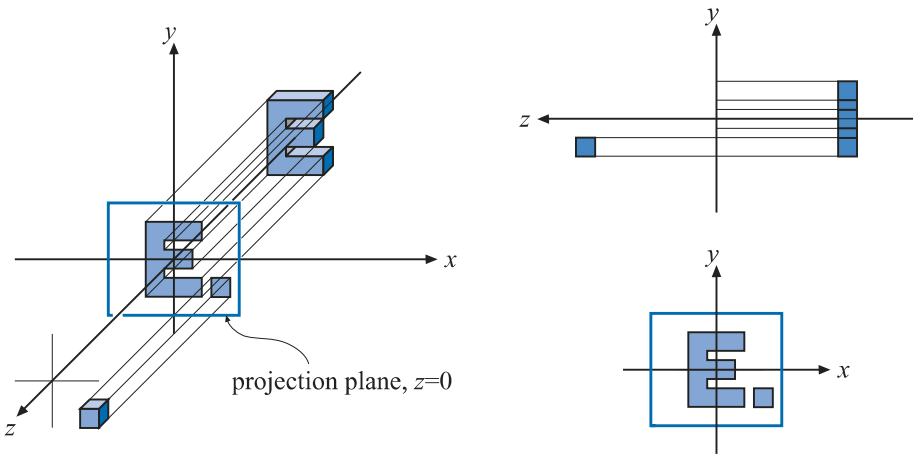

Figure 4.17. Three different views of the simple orthographic projection generated by Equation 4.62. This projection can be seen as the viewer is looking along the negative z-axis, which means that the projection simply skips (or sets to zero) the z-coordinate while keeping the x- and y-coordinates. Note that objects on both sides of $z = 0$ are projected onto the projection plane.

4.7.1 Orthographic Projection

A characteristic of an orthographic projection is that parallel lines remain parallel after the projection. When orthographic projection is used for viewing a scene, objects maintain the same size regardless of distance to the camera. Matrix $\mathbf{P}_o$, shown below, is a simple orthographic projection matrix that leaves the x- and y-components of a point unchanged, while setting the z-component to zero, i.e., it orthographically projects onto the plane $z = 0$:

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.62}$$

The effect of this projection is illustrated in Figure 4.17. Clearly, $\mathbf{P}_o$ is non-invertible, since its determinant $|\mathbf{P}_o| = 0$. In other words, the transform drops from three to two dimensions, and there is no way to retrieve the dropped dimension. A problem with using this kind of orthographic projection for viewing is that it projects both points with positive and points with negative z-values onto the projection plane. It is usually useful to restrict the z-values (and the x- and y-values) to a certain interval, from, say n (near plane) to f (far plane).[4] This is the purpose of the next transformation.

A more common matrix for performing orthographic projection is expressed by the six-tuple, (l, r, b, t, n, f), denoting the left, right, bottom, top, near, and far planes. This matrix scales and translates the *axis-aligned bounding box* (AABB; see the definition in Section 22.2) formed by these planes into an axis-aligned cube centered around

[4] The near plane is also called the *front plane* or *hither*; the far plane is also the *back plane* or *yon*.

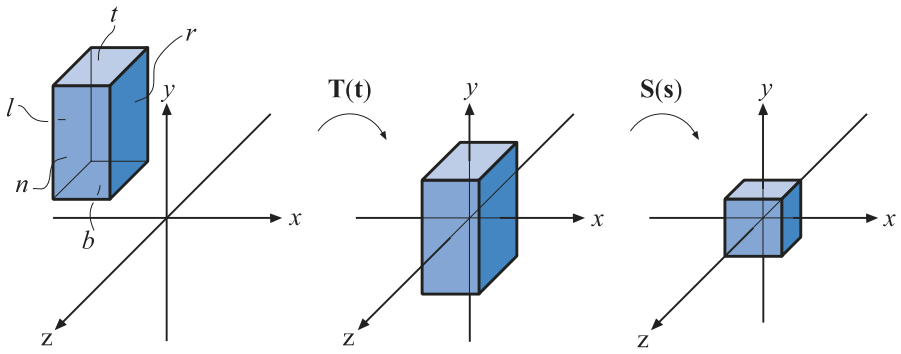

Figure 4.18. Transforming an axis-aligned box on the canonical view volume. The box on the left is first translated, making its center coincide with the origin. Then it is scaled to get the size of the canonical view volume, shown at the right.

the origin. The minimum corner of the AABB is (l, b, n) and the maximum corner is (r, t, f). It is important to realize that $n > f$, because we are looking down the negative z-axis at this volume of space. Our common sense says that the near value should be a lower number than the far, so one may let the user supply them as such, and then internally negate them.

In OpenGL the axis-aligned cube has a minimum corner of $(-1, -1, -1)$ and a maximum corner of $(1, 1, 1)$; in DirectX the bounds are $(-1, -1, 0)$ to $(1, 1, 1)$. This cube is called the *canonical view volume* and the coordinates in this volume are called *normalized device coordinates*. The transformation procedure is shown in Figure 4.18. The reason for transforming into the canonical view volume is that clipping is more efficiently performed there.

After the transformation into the canonical view volume, vertices of the geometry to be rendered are clipped against this cube. The geometry not outside the cube is finally rendered by mapping the remaining unit square to the screen. This orthographic transform is shown here:

$$\begin{aligned}\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.\end{aligned} \tag{4.63}$$

As suggested by this equation, $\mathbf{P}_o$ can be written as the concatenation of a translation, $\mathbf{T}(\mathbf{t})$, followed by a scaling matrix, $\mathbf{S}(\mathbf{s})$, where $\mathbf{s} = (2/(r-l), 2/(t-b), 2/(f-n))$, and $\mathbf{t} = (-(r+l)/2, -(t+b)/2, -(f+n)/2)$. This matrix is invertible,[5] i.e., $\mathbf{P}_o^{-1} = \mathbf{T}(-\mathbf{t})\mathbf{S}((r-l)/2, (t-b)/2, (f-n)/2)$.

In computer graphics, a left-hand coordinate system is most often used after projection—i.e., for the viewport, the x-axis goes to the right, y-axis goes up, and the z-axis goes into the viewport. Because the far value is less than the near value for the way we defined our AABB, the orthographic transform will always include a mirroring transform. To see this, say the original AABBs is the same size as the goal, the canonical view volume. Then the AABB's coordinates are $(-1, -1, 1)$ for (l, b, n) and $(1, 1, -1)$ for (r, t, f). Applying that to Equation 4.63 gives us

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.64}$$

which is a mirroring matrix. It is this mirroring that converts from the right-handed viewing coordinate system (looking down the negative z-axis) to left-handed normalized device coordinates.

DirectX maps the z-depths to the range $[0, 1]$ instead of OpenGL's $[-1, 1]$. This can be accomplished by applying a simple scaling and translation matrix applied after the orthographic matrix, that is,

$$\mathbf{M}_{st} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.65}$$

So, the orthographic matrix used in DirectX is

$$\mathbf{P}_{o[0,1]} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.66}$$

which is normally presented in transposed form, as DirectX uses a row-major form for writing matrices.

[5] If and only if $n \neq f$, $l \neq r$, and $t \neq b$; otherwise, no inverse exists.

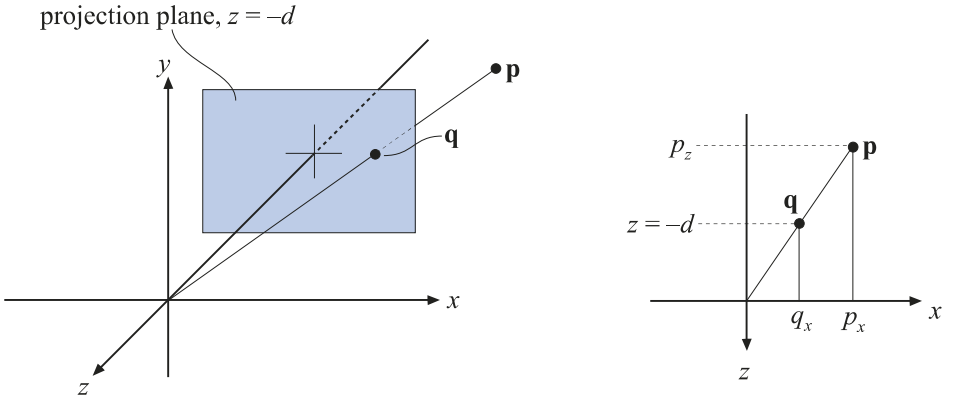

Figure 4.19. The notation used for deriving a perspective projection matrix. The point $\mathbf{p}$ is projected onto the plane $z = -d$, $d > 0$, which yields the projected point $\mathbf{q}$. The projection is performed from the perspective of the camera's location, which in this case is the origin. The similar triangle used in the derivation is shown for the x-component at the right.

4.7.2 Perspective Projection

A more complex transform than orthographic projection is perspective projection, which is commonly used in most computer graphics applications. Here, parallel lines are generally not parallel after projection; rather, they may converge to a single point at their extreme. Perspective more closely matches how we perceive the world, i.e., objects farther away are smaller.

First, we shall present an instructive derivation for a perspective projection matrix that projects onto a plane $z = -d$, $d > 0$. We derive from world space to simplify understanding of how the world-to-view conversion proceeds. This derivation is followed by the more conventional matrices used in, for example, OpenGL [885].

Assume that the camera (viewpoint) is located at the origin, and that we want to project a point, $\mathbf{p}$, onto the plane $z = -d$, $d > 0$, yielding a new point $\mathbf{q} = (q_x, q_y, -d)$. This scenario is depicted in Figure 4.19. From the similar triangles shown in this figure, the following derivation, for the x-component of $\mathbf{q}$, is obtained:

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \qquad \Longleftrightarrow \qquad q_x = -d\frac{p_x}{p_z}. \tag{4.67}$$

The expressions for the other components of $\mathbf{q}$ are $q_y = -dp_y/p_z$ (obtained similarly to q_x), and $q_z = -d$. Together with the above formula, these give us the perspective projection matrix, $\mathbf{P}_p$, as shown here:

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}. \tag{4.68}$$

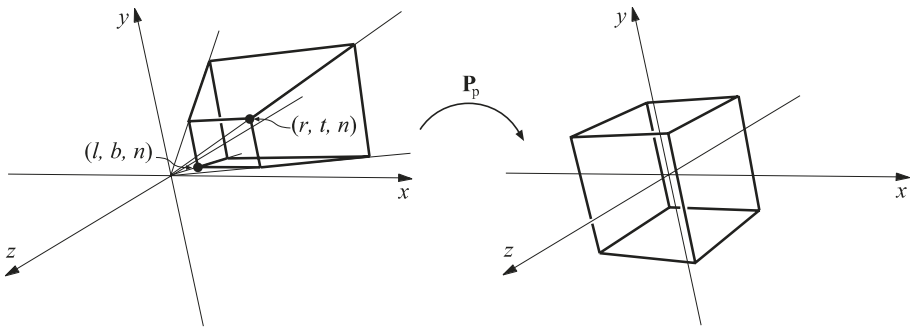

Figure 4.20. The matrix $\mathbf{P}_p$ transforms the view frustum into the unit cube, which is called the canonical view volume.

That this matrix yields the correct perspective projection is confirmed by

$$\mathbf{q} = \mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}. \tag{4.69}$$

The last step comes from the fact that the whole vector is divided by the w-component (in this case $-p_z/d$), to get a 1 in the last position. The resulting z value is always $-d$ since we are projecting onto this plane.

Intuitively, it is easy to understand why homogeneous coordinates allow for projection. One geometrical interpretation of the homogenization process is that it projects the point (p_x, p_y, p_z) onto the plane $w = 1$.

As with the orthographic transformation, there is also a perspective transform that, rather than actually projecting onto a plane (which is noninvertible), transforms the view frustum into the canonical view volume described previously. Here the view frustum is assumed to start at $z = n$ and end at $z = f$, with $0 > n > f$. The rectangle at $z = n$ has the minimum corner at (l, b, n) and the maximum corner at (r, t, n). This is shown in Figure 4.20.

The parameters (l, r, b, t, n, f) determine the view frustum of the camera. The horizontal field of view is determined by the angle between the left and the right planes (determined by l and r) of the frustum. In the same manner, the vertical field of view is determined by the angle between the top and the bottom planes (determined by t and b). The greater the field of view, the more the camera "sees." Asymmetric frusta can be created by $r \neq -l$ or $t \neq -b$. Asymmetric frusta are, for example, used for stereo viewing and for virtual reality (Section 21.2.3).

The field of view is an important factor in providing a sense of the scene. The eye itself has a physical field of view compared to the computer screen. This relationship is

$$\phi = 2\arctan(w/(2d)), \tag{4.70}$$

where ϕ is the field of view, w is the width of the object perpendicular to the line of sight, and d is the distance to the object. For example, a 25-inch monitor is about 22 inches wide. At 12 inches away, the horizontal field of view is 85 degrees; at 20 inches, it is 58 degrees; at 30 inches, 40 degrees. This same formula can be used to convert from camera lens size to field of view, e.g., a standard 50mm lens for a 35mm camera (which has a 36mm wide frame size) gives $\phi = 2\arctan(36/(2 \cdot 50)) = 39.6$ degrees.

Using a narrower field of view compared to the physical setup will lessen the perspective effect, as the viewer will be zoomed in on the scene. Setting a wider field of view will make objects appear distorted (like using a wide angle camera lens), especially near the screen's edges, and will exaggerate the scale of nearby objects. However, a wider field of view gives the viewer a sense that objects are larger and more impressive, and has the advantage of giving the user more information about the surroundings.

The perspective transform matrix that transforms the frustum into a unit cube is given by Equation 4.71:

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.71}$$

After applying this transform to a point, we will get another point $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$. The w-component, q_w, of this point will (most often) be nonzero and not equal to one. To get the projected point, $\mathbf{p}$, we need to divide by q_w, i.e.,

$$\mathbf{p} = (q_x/q_w, q_y/q_w, q_z/q_w, 1). \tag{4.72}$$

The matrix $\mathbf{P}_p$ always sees to it that $z = f$ maps to $+1$ and $z = n$ maps to -1.

Objects beyond the far plane will be clipped and so will not appear in the scene. The perspective projection can handle a far plane taken to infinity, which makes Equation 4.71 become

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & 1 & -2n \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.73}$$

To sum up, the perspective transform (in any form), $\mathbf{P}_p$, is applied, followed by clipping and homogenization (division by w), which results in normalized device coordinates.

To get the perspective transform used in OpenGL, first multiply with $\mathbf{S}(1, 1, -1, 1)$, for the same reasons as for the orthographic transform. This simply negates the values in the third column of Equation 4.71. After this mirroring transform has been applied, the near and far values are entered as positive values, with $0 < n' < f'$, as they would traditionally be presented to the user. However, they still represent distances along the world's negative z-axis, which is the direction of view. For reference purposes, here is the OpenGL equation:

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}. \tag{4.74}$$

A simpler setup is to provide just the vertical field of view, ϕ, the aspect ratio $a = w/h$ (where $w \times h$ is the screen resolution), n', and f'. This results in

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \tag{4.75}$$

where $c = 1.0/\tan(\phi/2)$. This matrix does exactly what the old `gluPerspective()` did, which is part of the OpenGL Utility Library (GLU).

Some APIs (e.g., DirectX) map the near plane to $z = 0$ (instead of $z = -1$) and the far plane to $z = 1$. In addition, DirectX uses a left-handed coordinate system to define its projection matrix. This means DirectX looks along the positive z-axis and presents the near and far values as positive numbers. Here is the DirectX equation:

$$\mathbf{P}_{p[0,1]} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f'}{f'-n'} & -\frac{f'n'}{f'-n'} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.76}$$

DirectX uses row-major form in its documentation, so this matrix is normally presented in transposed form.

One effect of using a perspective transformation is that the computed depth value does not vary linearly with the input p_z value. Using any of Equations 4.74–4.76 to

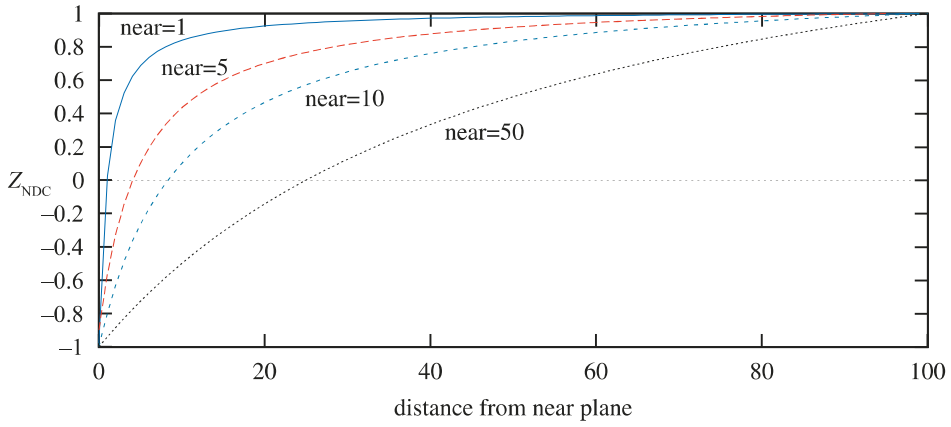

Figure 4.21. The effect of varying the distance of the near plane from the origin. The distance $f' - n'$ is kept constant at 100. As the near plane becomes closer to the origin, points nearer the far plane use a smaller range of the normalized device coordinate (NDC) depth space. This has the effect of making the z-buffer less accurate at greater distances.

multiply with a point $\mathbf{p}$, we can see that

$$\mathbf{v} = \mathbf{P}\mathbf{p} = \begin{pmatrix} \cdots \\ \cdots \\ dp_z + e \\ \pm p_z \end{pmatrix}, \tag{4.77}$$

where the details of v_x and v_y have been omitted, and the constants d and f depend on the chosen matrix. If we use Equation 4.74, for example, then $d = -(f'+n')/(f'-n')$, $e = -2f'n'/(f' - n')$, and $v_x = -p_z$. To obtain the depth in normalized device coordinates (NDC), we need to divide by the w-component, which results in

$$z_{\text{NDC}} = \frac{dp_z + e}{-p_z} = d - \frac{e}{p_z}, \tag{4.78}$$

where $z_{\text{NDC}} \in [-1, +1]$ for the OpenGL projection. As can be seen, the output depth z_{NDC} is inversely proportional to the input depth, p_z.

For example, if $n' = 10$ and $f' = 110$ (using the OpenGL terminology), when p_z is 60 units down the negative z-axis (i.e., the halfway point) the normalized device coordinate depth value is 0.833, not 0. Figure 4.21 shows the effect of varying the distance of the near plane from the origin. Placement of the near and far planes affects the precision of the z-buffer. This effect is discussed further in Section 23.7.

There are several ways to increase the depth precision. A common method, which we call *reversed z*, is to store $1.0 - z_{\text{NDC}}$ [978] either with floating point depth or with integers. A comparison is shown in Figure 4.22. Reed [1472] shows with simulations

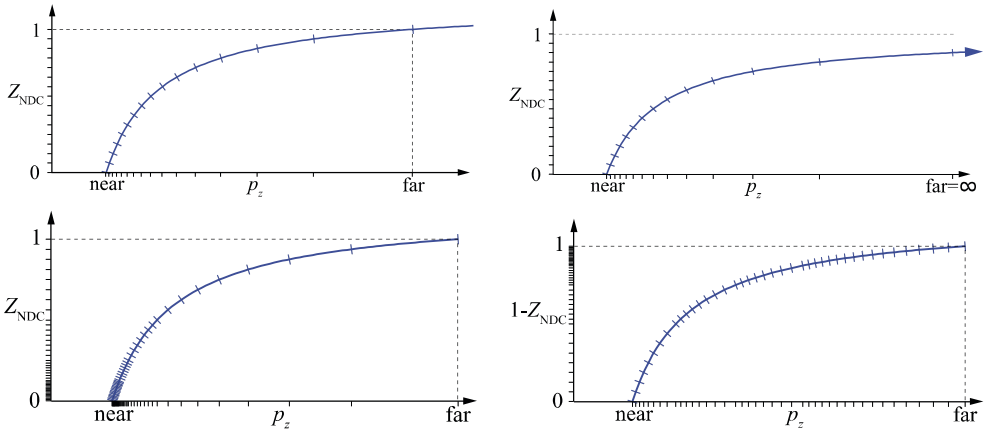

Figure 4.22. Different ways to set up the depth buffer with the DirectX transform, i.e., $z_{\text{NDC}} \in [0, +1]$. Top left: standard integer depth buffer, shown here with 4 bits of precision (hence the 16 marks on the y-axis). Top right: the far plane set to ∞, the small shifts on both axes showing that one does not lose much precision by doing so. Bottom left: with 3 exponent bits and 3 mantissa bits for floating point depth. Notice how the distribution is nonlinear on the y-axis, which makes it even worse on the x-axis. Bottom right: reversed floating point depth, i.e., $1 - z_{\text{NDC}}$, with a much better distribution as a result. *(Illustrations courtesy of Nathan Reed.)*

that using a floating point buffer with reversed z provides the best accuracy, and this is also the preferred method for integer depth buffers (which usually have 24 bits per depth). For the standard mapping (i.e., non-reversed z), separating the projection matrix in the transform decreases the error rate, as suggested by Upchurch and Desbrun [1803]. For example, it can be better to use $\mathbf{P}(\mathbf{Mp})$ than $\mathbf{Tp}$, where $\mathbf{T} = \mathbf{PM}$. Also, in the range of $[0.5, 1.0]$, fp32 and int24 are quite similar in accuracy, since fp32 has a 23-bit mantissa. The reason for having z_{NDC} proportional to $1/p_z$ is that it makes hardware simpler and compression of depth more successful, which is discussed more in Section 23.7.

Lloyd [1063] proposed to use a logarithm of the depth values to improve precision for shadow maps. Lauritzen et al. [991] use the previous frame's z-buffer to determine a maximum near plane and minimum far plane. For screen-space depth, Kemen [881] proposes to use the following remapping per vertex:

$$\begin{aligned} z &= w\left(\log_2\left(\max(10^{-6}, 1+w)\right)f_c - 1\right), && \text{[OpenGL]} \\ z &= w\log_2\left(\max(10^{-6}, 1+w)\right)f_c/2, && \text{[DirectX]} \end{aligned} \tag{4.79}$$

where w is the w-value of the vertex after the projection matrix, and z is the output z from the vertex shader. The constant f_c is $f_c = 2/\log_2(f+1)$, where f is the far plane. When this transform is applied in the vertex shader only, depth will still be interpolated linearly over the triangle by the GPU in between the nonlinearly transformed depths at vertices (Equation 4.79). Since the logarithm is a monotonic

function, occlusion culling hardware and depth compression techniques will still work as long as the difference between the piecewise linear interpolation and the accurate nonlinearly transformed depth value is small. That's true for most cases with sufficient geometry tessellation. However, it is also possible to apply the transform per fragment. This is done by outputting a per-vertex value of $e = 1 + w$, which is then interpolated by the GPU over the triangle. The pixel shader then modifies the fragment depth as $\log_2(e_i)f_c/2$, where e_i is the interpolated value of e. This method is a good alternative when there is no floating point depth in the GPU and when rendering using large distances in depth.

Cozzi [1605] proposes to use multiple frusta, which can improve accuracy to effectively any desired rate. The view frustum is divided in the depth direction into several non-overlapping smaller sub-frusta whose union is exactly the frustum. The sub-frusta are rendered to in back-to-front order. First, both the color and depth buffers are cleared, and all objects to be rendered are sorted into each sub-frusta that they overlap. For each sub-frusta, its projection matrix is set up, the depth buffer is cleared, and then the objects that overlap the sub-frusta are rendered.

Further Reading and Resources

The *immersive linear algebra* site [1718] provides an interactive book about the basics of this subject, helping build intuition by encouraging you to manipulate the figures. Other interactive learning tools and transform code libraries are linked from realtimerendering.com.

One of the best books for building up one's intuition about matrices in a painless fashion is Farin and Hansford's *The Geometry Toolbox* [461]. Another useful work is Lengyel's *Mathematics for 3D Game Programming and Computer Graphics* [1025]. For a different perspective, many computer graphics texts, such as Hearn and Baker [689], Marschner and Shirley [1129], and Hughes et al. [785] also cover matrix basics. The course by Ochiai et al. [1310] introduces the matrix foundations as well as the exponential and logarithm of matrices, with uses for computer graphics. The *Graphics Gems* series [72, 540, 695, 902, 1344] presents various transform-related algorithms and has code available online for many of these. Golub and Van Loan's *Matrix Computations* [556] is the place to start for a serious study of matrix techniques in general. More on skeleton-subspace deformation/vertex blending and shape interpolation can be read in Lewis et al.'s SIGGRAPH paper [1037].

Hart et al. [674] and Hanson [663] provide visualizations of quaternions. Pletinckx [1421] and Schlag [1566] present different ways of interpolating smoothly between a set of quaternions. Vlachos and Isidoro [1820] derive formulae for C^2 interpolation of quaternions. Related to quaternion interpolation is the problem of computing a consistent coordinate system along a curve. This is treated by Dougan [374].

Alexa [28] and Lazarus and Verroust [1000] present surveys on many different morphing techniques. Parent's book [1354] is an excellent source for techniques about computer animation.

Chapter 5
Shading Basics

"A good picture is equivalent to a good deed."
—Vincent Van Gogh

When you render images of three-dimensional objects, the models should not only have the proper geometrical shape, they should also have the desired visual appearance. Depending on the application, this can range from photorealism—an appearance nearly identical to photographs of real objects—to various types of stylized appearance chosen for creative reasons. See Figure 5.1 for examples of both.

This chapter will discuss those aspects of shading that are equally applicable to photorealistic and stylized rendering. Chapter 15 is dedicated specifically to stylized rendering, and a significant part of the book, Chapters 9 through 14, focuses on physically based approaches commonly used for photorealistic rendering.

5.1 Shading Models

The first step in determining the appearance of a rendered object is to choose a *shading model* to describe how the object's color should vary based on factors such as surface orientation, view direction, and lighting.

As an example, we will use a variation on the *Gooch shading model* [561]. This is a form of non-photorealistic rendering, the subject of Chapter 15. The Gooch shading model was designed to increase legibility of details in technical illustrations.

The basic idea behind Gooch shading is to compare the surface normal to the light's location. If the normal points toward the light, a warmer tone is used to color the surface; if it points away, a cooler tone is used. Angles in between interpolate between these tones, which are based on a user-supplied surface color. In this example, we add a stylized "highlight" effect to the model to give the surface a shiny appearance. Figure 5.2 shows the shading model in action.

Shading models often have properties used to control appearance variation. Setting the values of these properties is the next step in determining object appearance. Our example model has just one property, surface color, as shown in the bottom image of Figure 5.2.