6

Transformation Matrices

The machinery of linear algebra can be used to express many of the operations required to arrange objects in a 3D scene, view them with cameras, and get them onto the screen. *Geometric transformations* like rotation, translation, scaling, and projection can be accomplished with matrix multiplication, and the *transformation matrices* used to do this are the subject of this chapter.

We will show how a set of points transforms if the points are represented as offset vectors from the origin, and we will use the clock shown in Figure 6.1 as an example of a point set. So think of the clock as a bunch of points that are the ends of vectors whose tails are at the origin. We also discuss how these transforms operate differently on locations (points), displacement vectors, and surface normal vectors.

6.1 2D Linear Transformations

We can use a 2×2 matrix to change, or transform, a 2D vector:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}.$$

This kind of operation, which takes in a 2-vector and produces another 2-vector by a simple matrix multiplication, is a *linear transformation*.

By this simple formula we can achieve a variety of useful transformations, depending on what we put in the entries of the matrix, as will be discussed in the following sections. For our purposes, consider moving along the x-axis a horizontal move and along the y-axis, a vertical move.

6.1.1 Scaling

The most basic transform is a *scale* along the coordinate axes. This transform can change length and possibly direction:

$$\text{scale}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}.$$

Note what this matrix does to a vector with Cartesian components (x, y):

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}.$$

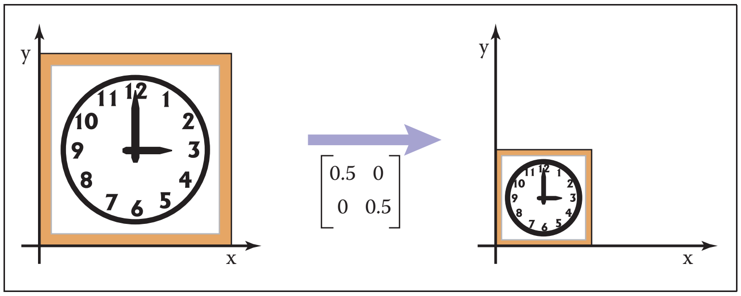
So, just by looking at the matrix of an axis-aligned scale, we can read off the two scale factors.

Example. The matrix that shrinks x and y uniformly by a factor of two is (Figure 6.1)

$$\text{scale}(0.5, 0.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

A matrix which halves in the horizontal and increases by three-halves in the vertical is (see Figure 6.2)

$$\text{scale}(0.5, 1.5) = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}.$$



$$\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

Figure 6.1. Scaling uniformly by half for each axis: The axis-aligned scale matrix has the proportion of change in each of the diagonal elements and zeroes in the off-diagonal elements.

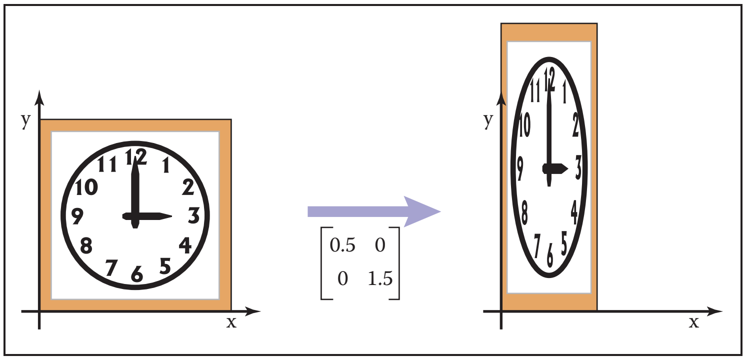


Figure 6.2. Scaling nonuniformly in *x* and *y*: The scaling matrix is diagonal with non-equal elements. Note that the square outline of the clock becomes a rectangle and the circular face becomes an ellipse.

6.1.2 Shearing

A shear is something that pushes things sideways, producing something like a deck of cards across which you push your hand; the bottom card stays put and cards move more the closer they are to the top of the deck. The horizontal and vertical shear matrices are

$$\text{shear-x}(s) = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}, \quad \text{shear-y}(s) = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}.$$

Example. The transform that shears horizontally so that vertical lines become $45°$ lines leaning toward the right is (see Figure 6.3)

$$\text{shear-x}(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

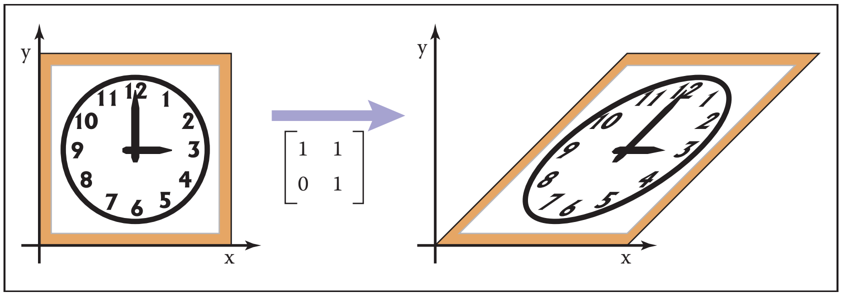


Figure 6.3. An *x*-shear matrix moves points to the right in proportion to their *y*-coordinate. Now the square outline of the clock becomes a parallelogram and, as with scaling, the circular face of the clock becomes an ellipse.

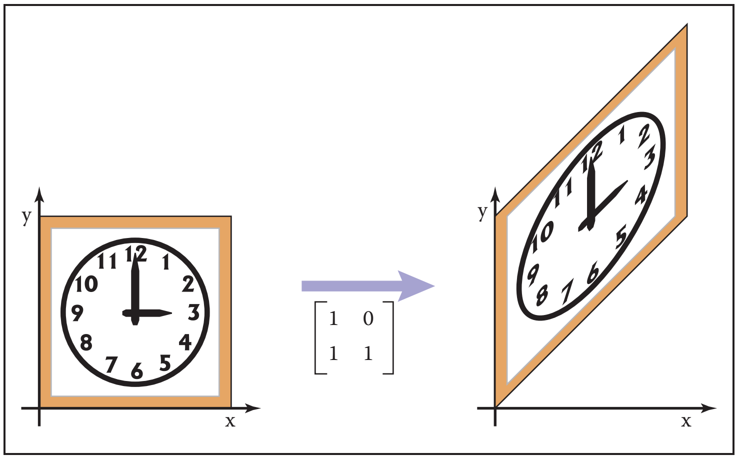


Figure 6.4. A y-shear matrix moves points up in proportion to their x-coordinate.

An analogous transform vertically is (see Figure 6.4)

$$\text{shear-y}(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

In both cases, the square outline of the sheared clock becomes a parallelogram, and the circular face of the sheared clock becomes an ellipse.

In fact, the image of a circle under any matrix transformation is an ellipse.

Another way to think of a shear is in terms of rotation of only the vertical (or horizontal) axis. The shear transform that takes a vertical axis and tilts it clockwise by an angle ϕ is

$$\begin{bmatrix} 1 & \tan\phi \\ 0 & 1 \end{bmatrix}.$$

Similarly, the shear matrix which rotates the horizontal axis counterclockwise by angle ϕ is

$$\begin{bmatrix} 1 & 0 \\ \tan\phi & 1 \end{bmatrix}.$$

6.1.3 Rotation

Suppose we want to rotate a vector $\mathbf{a}$ by an angle ϕ counterclockwise to get vector $\mathbf{b}$ (Figure 6.5). If $\mathbf{a}$ makes an angle α with the x-axis, and its length is $r = \sqrt{x_a^2 + y_a^2}$, then we know that

$$x_a = r\cos\alpha,$$
$$y_a = r\sin\alpha.$$

Because **b** is a rotation of **a**, it also has length r. Because it is rotated an angle ϕ from **a**, **b** makes an angle $(\alpha + \phi)$ with the x-axis. Using the trigonometric addition identities (Section 2.3.3):

$$\begin{aligned} x_b &= r\cos(\alpha+\phi) = r\cos\alpha\cos\phi - r\sin\alpha\sin\phi, \\ y_b &= r\sin(\alpha+\phi) = r\sin\alpha\cos\phi + r\cos\alpha\sin\phi. \end{aligned} \tag{6.1}$$

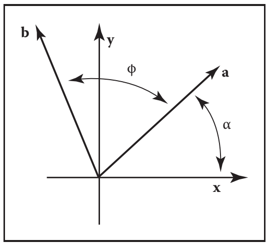


Figure 6.5. The geometry for Equation (6.1).

Substituting $x_a = r\cos\alpha$ and $y_a = r\sin\alpha$ gives

$$\begin{aligned} x_b &= x_a\cos\phi - y_a\sin\phi, \\ y_b &= y_a\cos\phi + x_a\sin\phi. \end{aligned}$$

In matrix form, the transformation that takes **a** to **b** is then

$$\text{rotate}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}.$$

Example. A matrix that rotates vectors by $\pi/4$ radians (45 degrees) is (see Figure 6.6)

$$\begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix}.$$

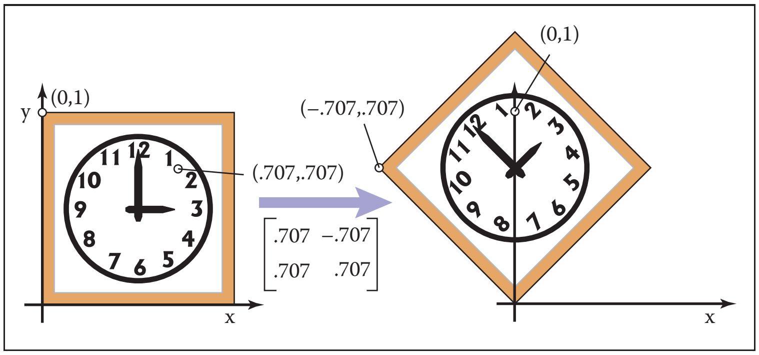


Figure 6.6. A rotation by 45°. Note that the rotation is counterclockwise and that cos(45°) = sin(45°) ≈ .707.

A matrix that rotates by $\pi/6$ radians (30 degrees) in the *clockwise* direction is a rotation by $-\pi/6$ radians in our framework (see Figure 6.7):

$$\begin{bmatrix} \cos\frac{-\pi}{6} & -\sin\frac{-\pi}{6} \\ \sin\frac{-\pi}{6} & \cos\frac{-\pi}{6} \end{bmatrix} = \begin{bmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{bmatrix}.$$

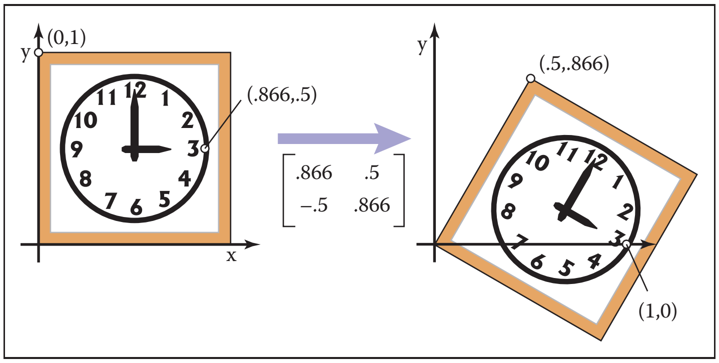


Figure 6.7. A rotation by –30 degrees. Note that the rotation is clockwise and that cos(–30°) ≈ .866 and sin(–30°) = –.5.

Because the norm of each row of a rotation matrix is one ($\sin^2\phi + \cos^2\phi = 1$), and the rows are orthogonal ($\cos\phi(-\sin\phi) + \sin\phi\cos\phi = 0$), we see that rotation matrices are orthogonal matrices (Section 5.2.4). By looking at the matrix we can read off two pairs of orthonormal vectors: the two columns, which are the vectors to which the transformation sends the canonical basis vectors $(1, 0)$ and $(0, 1)$; and the rows, which are the vectors that the transformations sends *to* the canonical basis vectors.

Said briefly, $\mathbf{Re}_i = \mathbf{u}_i$ and $\mathbf{Rv}_i = \mathbf{u}_i$, for a rotation with columns $\mathbf{u}_i$ and rows $\mathbf{v}_i$.

6.1.4 Reflection

We can reflect a vector across either of the coordinate axes by using a scale with one negative scale factor (see Figures 6.8 and 6.9):

$$\text{reflect-y} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{reflect-x} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

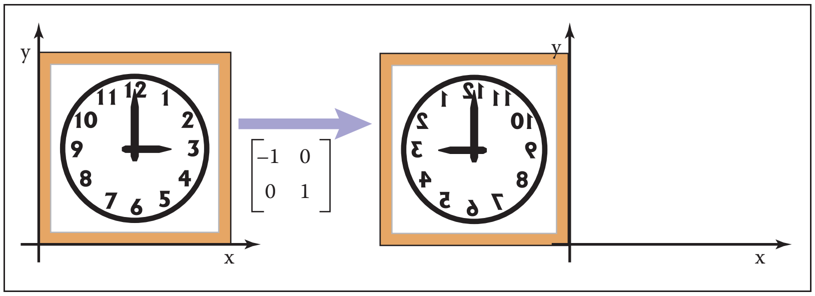


Figure 6.8. A reflection about the *y*-axis is achieved by multiplying all *x*-coordinates by –1.

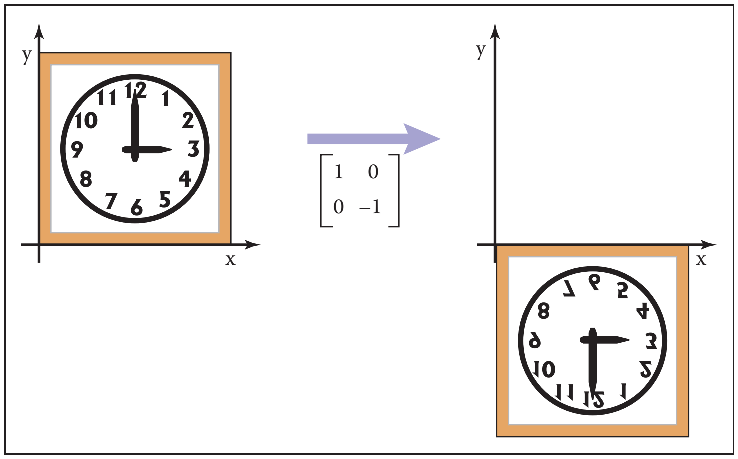


Figure 6.9. A reflection about the x-axis is achieved by multiplying all y-coordinates by –1.

While one might expect that the matrix with -1 in both elements of the diagonal is also a reflection, in fact it is just a rotation by π radians.

This rotation can also be called a "reflection through the origin."

6.1.5 Composition and Decomposition of Transformations

It is common for graphics programs to apply more than one transformation to an object. For example, we might want to first apply a scale $\mathbf{S}$, and then a rotation $\mathbf{R}$. This would be done in two steps on a 2D vector $\mathbf{v}_1$:

$$\text{first,}\mathbf{v}_2 = \mathbf{S}\mathbf{v}_1, \;\; \text{then,}\mathbf{v}_3 = \mathbf{R}\mathbf{v}_2.$$

Another way to write this is

$$\mathbf{v}_3 = \mathbf{R}\left(\mathbf{S}\mathbf{v}_1\right).$$

Because matrix multiplication is associative, we can also write

$$\mathbf{v}_3 = \left(\mathbf{R}\mathbf{S}\right)\mathbf{v}_1.$$

In other words, we can represent the effects of transforming a vector by two matrices in sequence using a single matrix of the same size, which we can compute by multiplying the two matrices: $\mathbf{M} = \mathbf{R}\mathbf{S}$ (Figure 6.10).

It is *very important* to remember that these transforms are applied from the *right side first*. So the matrix $\mathbf{M} = \mathbf{R}\mathbf{S}$ first applies $\mathbf{S}$ and then $\mathbf{R}$.

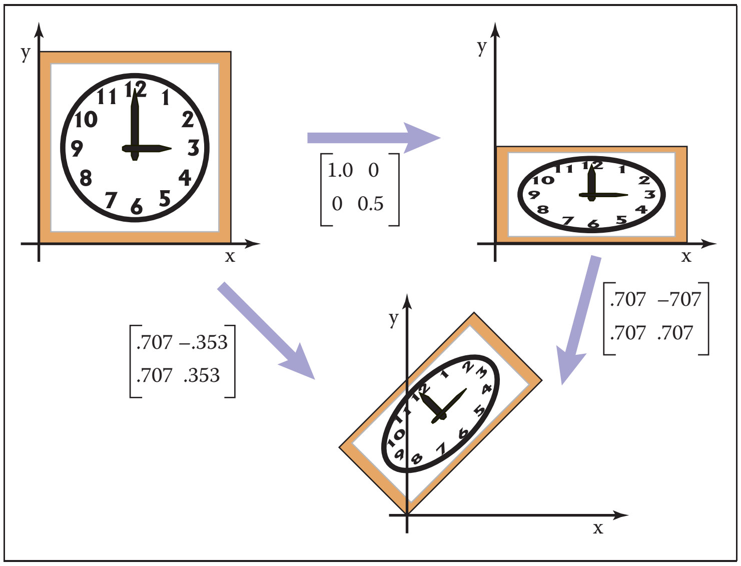


Figure 6.10. Applying the two transform matrices in sequence is the same as applying the product of those matrices once. This is a key concept that underlies most graphics hardware and software.

Example. Suppose we want to scale by one-half in the vertical direction and then rotate by $\pi/4$ radians (45 degrees). The resulting matrix is

$$\begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.353 \\ 0.707 & 0.353 \end{bmatrix}.$$

It is important to always remember that matrix multiplication is not commutative. So the order of transforms *does* matter. In this example, rotating first, and then scaling, results in a different matrix (see Figure 6.11):

$$\begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{bmatrix} = \begin{bmatrix} 0.707 & -0.707 \\ 0.353 & 0.353 \end{bmatrix}.$$

Example. Using the scale matrices we have presented, nonuniform scaling can only be done along the coordinate axes. If we wanted to stretch our clock by 50% along one of its diagonals, so that 8:00 through 1:00 move to the northwest and 2:00 through 7:00 move to the southeast, we can use rotation matrices in combination with an axis-aligned scaling matrix to get the result we want. The idea is to use a rotation to align the scaling axis with a coordinate axis, then scale along that axis, then rotate back. In our example, the scaling axis is the "backslash" diagonal of the square, and we can make it parallel to the x-axis with

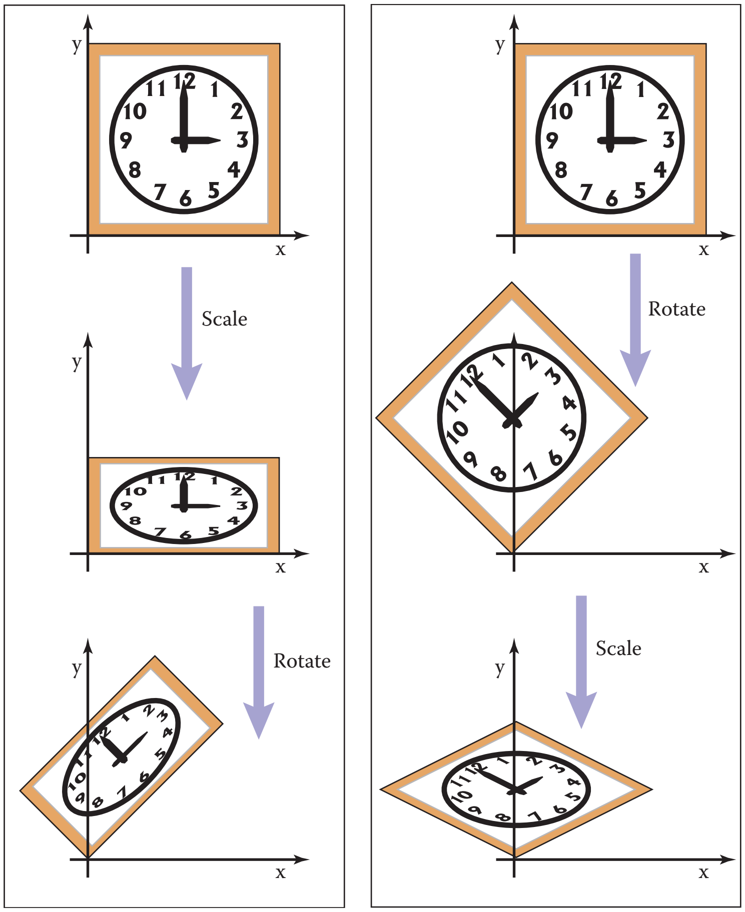


Figure 6.11. The order in which two transforms are applied is usually important. In this example, we do a scale by one-half in *y* and then rotate by 45°. Reversing the order in which these two transforms are applied yields a different result.

a rotation by $+45°$. Putting these operations together, the full transformation is

$$\text{rotate}(-45°)\,\text{scale}(1.5, 1)\,\text{rotate}(45°).$$

Remember to read the transformations from right to left.

In mathematical notation, this can be written $\mathbf{RSR}^{\mathrm{T}}$. The result of multiplying the three matrices together is

$$\begin{bmatrix} 1.25 & -0.25 \\ -0.25 & 1.25 \end{bmatrix}$$

It is no coincidence that this matrix is symmetric—try applying the transpose-of-product rule to the formula $\mathbf{RSR}^{\mathrm{T}}$.

Building up a transformation from rotation and scaling transformations actually works for any linear transformation, and this fact leads to a powerful way of thinking about these transformations, as explored in the next section.

6.1.6 Decomposition of Transformations

Sometimes it's necessary to "undo" a composition of transformations, taking a transformation apart into simpler pieces. For instance, it's often useful to present a transformation to the user for manipulation in terms of separate rotations and scale factors, but a transformation might be represented internally simply as a

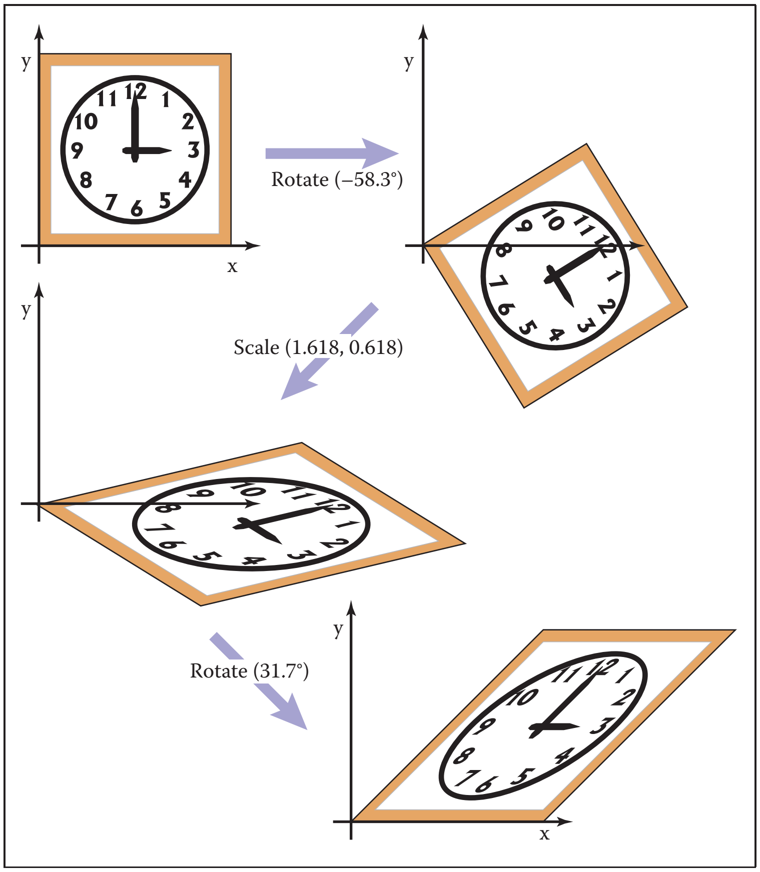


Figure 6.12. Singular Value Decomposition (SVD) for a shear matrix. Any 2D matrix can be decomposed into a product of rotation, scale, rotation. Note that the circular face of the clock must become an ellipse because it is just a rotated and scaled circle.

matrix, with the rotations and scales already mixed together. This kind of manipulation can be achieved if the matrix can be computationally disassembled into the desired pieces, the pieces adjusted, and the matrix reassembled by multiplying the pieces together again.

It turns out that this decomposition, or factorization, is possible, regardless of the entries in the matrix—and this fact provides a fruitful way of thinking about transformations and what they do to geometry that is transformed by them.

Symmetric Eigenvalue Decomposition

Let's start with symmetric matrices. Recall from Section 5.4 that a symmetric matrix can always be taken apart using the eigenvalue decomposition into a product of the form

$$\mathbf{A} = \mathbf{R}\mathbf{S}\mathbf{R}^{\mathrm{T}}$$

where $\mathbf{R}$ is an orthogonal matrix and $\mathbf{S}$ is a diagonal matrix; we will call the columns of $\mathbf{R}$ (the eigenvectors) by the names $\mathbf{v}_1$ and $\mathbf{v}_2$, and we'll call the diagonal entries of $\mathbf{S}$ (the eigenvalues) by the names λ_1 and λ_2.

In geometric terms we can now recognize $\mathbf{R}$ as a rotation and $\mathbf{S}$ as a scale, so this is just a multi-step geometric transformation (Figure 6.13):

1. Rotate $\mathbf{v}_1$ and $\mathbf{v}_2$ to the x- and y-axes (the transform by $\mathbf{R}^{\mathrm{T}}$).
2. Scale in x and y by (λ_1, λ_2) (the transform by $\mathbf{S}$).
3. Rotate the x- and y-axes back to $\mathbf{v}_1$ and $\mathbf{v}_2$ (the transform by $\mathbf{R}$).

If you like to count dimensions: a symmetric 2 $\times$ 2 matrix has 3 degrees of freedom, and the eigenvalue decomposition rewrites them as a rotation angle and two scale factors.

Looking at the effect of these three transforms together, we can see that they have the effect of a nonuniform scale along a pair of axes. As with an axis-aligned scale, the axes are perpendicular, but they aren't the coordinate axes; instead they

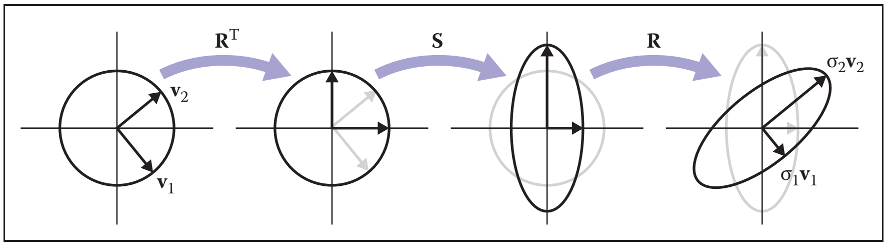


Figure 6.13. What happens when the unit circle is transformed by an arbitrary symmetric matrix $\mathbf{A}$, also known as a non–axis-aligned, nonuniform scale. The two perpendicular vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, which are the eigenvectors of $\mathbf{A}$, remain fixed in direction but get scaled. In terms of elementary transformations, this can be seen as first rotating the eigenvectors to the canonical basis, doing an axis-aligned scale, and then rotating the canonical basis back to the eigenvectors.

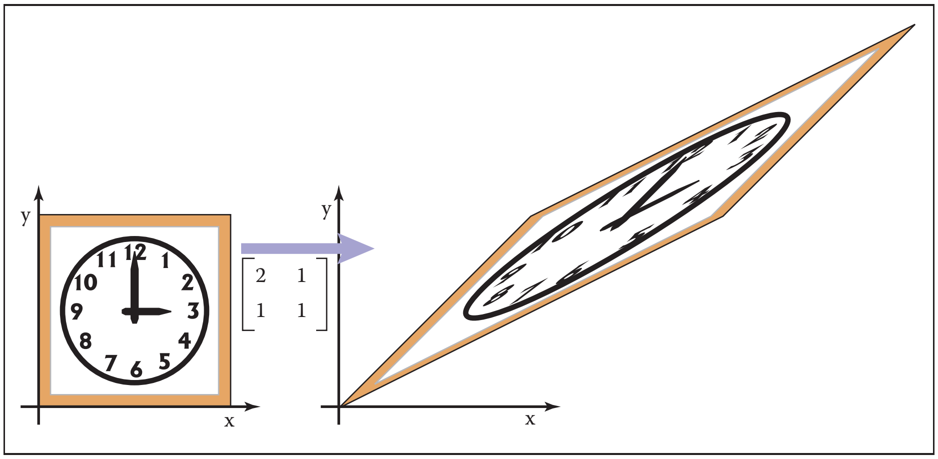


Figure 6.14. A symmetric matrix is always a scale along some axis. In this case it is along the $\phi = 31.7°$ direction which means the real eigenvector for this matrix is in that direction.

are the eigenvectors of $\mathbf{A}$. This tells us something about what it means to be a symmetric matrix: symmetric matrices are just scaling operations—albeit potentially nonuniform and non–axis-aligned ones.

Example. Recall the example from Section 5.4:

$$\begin{aligned}\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} &= \mathbf{R} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \mathbf{R}^{\mathrm{T}} \\ &= \begin{bmatrix} 0.8507 & -0.5257 \\ 0.5257 & 0.8507 \end{bmatrix} \begin{bmatrix} 2.618 & 0 \\ 0 & 0.382 \end{bmatrix} \begin{bmatrix} 0.8507 & 0.5257 \\ -0.5257 & 0.8507 \end{bmatrix} \\ &= \text{rotate}\,(31.7°)\ \text{scale}\,(2.618, 0.382)\ \text{rotate}\,(-31.7°).\end{aligned}$$

The matrix above, then, according to its eigenvalue decomposition, scales in a direction 31.7° counterclockwise from three o'clock (the x-axis). This is a touch before 2 p.m. on the clockface as is confirmed by Figure 6.14.

We can also reverse the diagonalization process; to scale by (λ_1, λ_2) with the first scaling direction an angle ϕ clockwise from the x-axis, we have

$$\begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} \lambda_1 \cos^2\phi + \lambda_2 \sin^2\phi & (\lambda_2 - \lambda_1)\cos\phi\sin\phi \\ (\lambda_2 - \lambda_1)\cos\phi\sin\phi & \lambda_2 \cos^2\phi + \lambda_1 \sin^2\phi \end{bmatrix}.$$

We should take heart that this is a symmetric matrix as we know must be true since we constructed it from a symmetric eigenvalue decomposition.

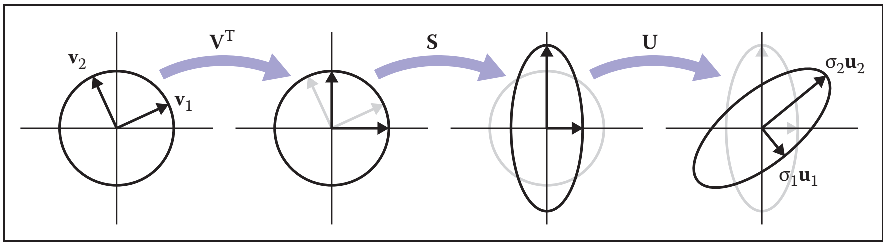


Figure 6.15. What happens when the unit circle is transformed by an arbitrary matrix $\mathbf{A}$. The two perpendicular vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, which are the right singular vectors of $\mathbf{A}$, get scaled and changed in direction to match the left singular vectors, $\mathbf{u}_1$ and $\mathbf{u}_2$. In terms of elementary transformations, this can be seen as first rotating the right singular vectors to the canonical basis, doing an axis-aligned scale, and then rotating the canonical basis to the left singular vectors.

Singular Value Decomposition

A very similar kind of decomposition can be done with nonsymmetric matrices as well: it's the Singular Value Decomposition (SVD), also discussed in Section 5.4.1. The difference is that the matrices on either side of the diagonal matrix are no longer the same:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

The two orthogonal matrices that replace the single rotation $\mathbf{R}$ are called $\mathbf{U}$ and $\mathbf{V}$, and their columns are called $\mathbf{u}_i$ (the *left singular vectors*) and $\mathbf{v}_i$ (the *right singular vectors*), respectively. In this context, the diagonal entries of $\mathbf{S}$ are called *singular values* rather than eigenvalues. The geometric interpretation is very similar to that of the symmetric eigenvalue decomposition (Figure 6.15):

1. Rotate $\mathbf{v}_1$ and $\mathbf{v}_2$ to the x- and y-axes (the transform by $\mathbf{V}^T$).

2. Scale in x and y by (σ_1, σ_2) (the transform by $\mathbf{S}$).

3. Rotate the x- and y-axes to $\mathbf{u}_1$ and $\mathbf{u}_2$ (the transform by $\mathbf{U}$).

For dimension counters: a general 2×2 matrix has 4 degrees of freedom, and the SVD rewrites them as two rotation angles and two scale factors. One more bit is needed to keep track of reflections, but that doesn't add a dimension.

The principal difference is between a single rotation and two different orthogonal matrices. This difference causes another, less important, difference. Because the SVD has different singular vectors on the two sides, there is no need for negative singular values: we can always flip the sign of a singular value, reverse the direction of one of the associated singular vectors, and end up with the same transformation again. For this reason, the SVD always produces a diagonal matrix with all positive entries, but the matrices $\mathbf{U}$ and $\mathbf{V}$ are not guaranteed to be rotations—they could include reflection as well. In geometric applications like graphics this is an inconvenience, but a minor one: it is easy to differentiate rotations from reflections by checking the determinant, which is $+1$ for rotations and -1 for reflections, and if rotations are desired, one of the singular values can be negated, resulting in a rotation–scale–rotation sequence where the reflection is rolled in with the scale, rather than with one of the rotations.

Example. The example used in Section 5.4.1 is in fact a shear matrix (Figure 6.12):

$$\begin{aligned}\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} &= \mathbf{R}_2\begin{bmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{bmatrix}\mathbf{R}_1\\ &= \begin{bmatrix}0.8507 & -0.5257\\ 0.5257 & 0.8507\end{bmatrix}\begin{bmatrix}1.618 & 0\\ 0 & 0.618\end{bmatrix}\begin{bmatrix}0.5257 & 0.8507\\ -0.8507 & 0.5257\end{bmatrix}\\ &= \text{rotate}\,(31.7^\circ)\ \text{scale}\,(1.618, 0.618)\ \text{rotate}\,(-58.3^\circ).\end{aligned}$$

An immediate consequence of the existence of SVD is that all the 2D transformation matrices we have seen can be made from rotation matrices and scale matrices. Shear matrices are a convenience, but they are not required for expressing transformations.

In summary, every matrix can be decomposed via SVD into a rotation times a scale times another rotation. Only symmetric matrices can be decomposed via eigenvalue diagonalization into a rotation times a scale times the inverse-rotation, and such matrices are a simple scale in an arbitrary direction. The SVD of a symmetric matrix will yield the same triple product as eigenvalue decomposition via a slightly more complex algebraic manipulation.

Paeth Decomposition of Rotations

Another decomposition uses shears to represent nonzero rotations (Paeth, 1990). The following identity allows this:

$$\begin{bmatrix}\cos\phi & -\sin\phi\\ \sin\phi & \cos\phi\end{bmatrix} = \begin{bmatrix}1 & \frac{\cos\phi-1}{\sin\phi}\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ \sin\phi & 1\end{bmatrix}\begin{bmatrix}1 & \frac{\cos\phi-1}{\sin\phi}\\ 0 & 1\end{bmatrix}.$$

For example, a rotation by $\pi/4$ (45 degrees) is (see Figure 6.16)

$$\text{rotate}(\frac{\pi}{4}) = \begin{bmatrix}1 & 1-\sqrt{2}\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ \frac{\sqrt{2}}{2} & 1\end{bmatrix}\begin{bmatrix}1 & 1-\sqrt{2}\\ 0 & 1\end{bmatrix}. \qquad (6.2)$$

This particular transform is useful for raster rotation because shearing is a very efficient raster operation for images; it introduces some jagginess, but will

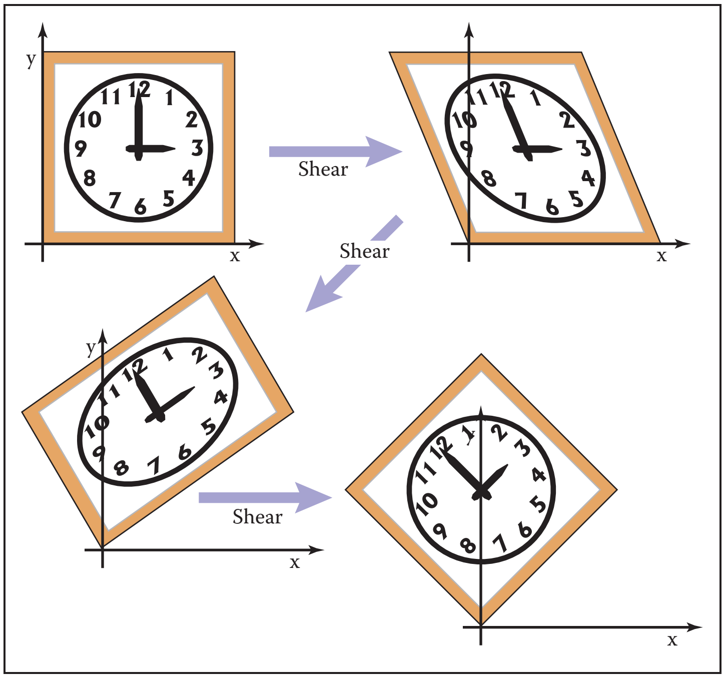


Figure 6.16. Any 2D rotation can be accomplished by three shears in sequence. In this case a rotation by 45° is decomposed as shown in Equation 6.2.

leave no holes. The key observation is that if we take a raster position (i, j) and apply a horizontal shear to it, we get

$$\begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i + sj \\ j \end{bmatrix}.$$

If we round sj to the nearest integer, this amounts to taking each row in the image and moving it sideways by some amount—a different amount for each row. Because it is the same displacement within a row, this allows us to rotate with no gaps in the resulting image. A similar action works for a vertical shear. Thus, we can implement a simple raster rotation easily.

6.2 3D Linear Transformations

The linear 3D transforms are an extension of the 2D transforms. For example, a scale along Cartesian axes is

$$\text{scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}. \tag{6.3}$$

Rotation is considerably more complicated in 3D than in 2D, because there are more possible axes of rotation. However, if we simply want to rotate about the z-axis, which will only change x- and y-coordinates, we can use the 2D rotation matrix with no operation on z:

$$\text{rotate-z}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Similarly we can construct matrices to rotate about the x-axis and the y-axis:

To understand why the minus sign is in the lower left for the y-axis rotation, think of the three axes in a circular sequence: y after x; z after y; x after z.

$$\text{rotate-x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix},$$

$$\text{rotate-y}(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}.$$

We will discuss rotations about arbitrary axes in the next section.

As in two dimensions, we can shear along a particular axis, for example,

$$\text{shear-x}(d_y, d_z) = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

As with 2D transforms, any 3D transformation matrix can be decomposed using SVD into a rotation, scale, and another rotation. Any symmetric 3D matrix has an eigenvalue decomposition into rotation, scale, and inverse-rotation. Finally, a 3D rotation can be decomposed into a product of 3D shear matrices.

6.2.1 Arbitrary 3D Rotations

As in 2D, 3D rotations are *orthogonal* matrices. Geometrically, this means that the three rows of the matrix are the Cartesian coordinates of three mutually orthogonal unit vectors as discussed in Section 2.4.5. The columns are three, potentially different, mutually orthogonal unit vectors. There are an infinite number of such rotation matrices. Let's write down such a matrix:

$$\mathbf{R}_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Here, $\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}$ and so on for $\mathbf{v}$ and $\mathbf{w}$. Since the three vectors are orthonormal we know that

$$\begin{aligned}\mathbf{u}\cdot\mathbf{u} = \mathbf{v}\cdot\mathbf{v} = \mathbf{w}\cdot\mathbf{w} = 1,\\ \mathbf{u}\cdot\mathbf{v} = \mathbf{v}\cdot\mathbf{w} = \mathbf{w}\cdot\mathbf{u} = 0.\end{aligned}$$

We can infer some of the behavior of the rotation matrix by applying it to the vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. For example,

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} \begin{bmatrix} x_u \\ y_u \\ z_u \end{bmatrix} = \begin{bmatrix} x_u x_u + y_u y_u + z_u z_u \\ x_v x_u + y_v y_u + z_v z_u \\ x_w x_u + y_w y_u + z_w z_u \end{bmatrix}.$$

Note that those three rows of $\mathbf{R}_{uvw}\mathbf{u}$ are all dot products:

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} \\ \mathbf{v}\cdot\mathbf{u} \\ \mathbf{w}\cdot\mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}.$$

Similarly, $\mathbf{R}_{uvw}\mathbf{v} = \mathbf{y}$, and $\mathbf{R}_{uvw}\mathbf{w} = \mathbf{z}$. So $\mathbf{R}_{uvw}$ takes the basis $\mathbf{uvw}$ to the corresponding Cartesian axes via rotation.

If $\mathbf{R}_{uvw}$ is a rotation matrix with orthonormal rows, then $\mathbf{R}_{uvw}^{\mathrm{T}}$ is also a rotation matrix with orthonormal columns, and in fact is the inverse of $\mathbf{R}_{uvw}$ (the inverse of an orthogonal matrix is always its transpose). An important point is that for transformation matrices, the algebraic inverse is also the geometric inverse. So if $\mathbf{R}_{uvw}$ takes $\mathbf{u}$ to $\mathbf{x}$, then $\mathbf{R}_{uvw}^{\mathrm{T}}$ takes $\mathbf{x}$ to $\mathbf{u}$. The same should be true of $\mathbf{v}$ and $\mathbf{y}$ as we can confirm:

$$\mathbf{R}_{uvw}^{\mathrm{T}}\mathbf{y} = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \mathbf{v}.$$

So we can always create rotation matrices from orthonormal bases.

If we wish to rotate about an arbitrary vector $\mathbf{a}$, we can form an orthonormal basis with $\mathbf{w} = \mathbf{a}$, rotate that basis to the canonical basis $\mathbf{xyz}$, rotate about the z-axis, and then rotate the canonical basis back to the $\mathbf{uvw}$ basis. In matrix form, to rotate about the w-axis by an angle ϕ:

$$\begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Here we have $\mathbf{w}$ a unit vector in the direction of $\mathbf{a}$ (i.e., $\mathbf{a}$ divided by its own length). But what are $\mathbf{u}$ and $\mathbf{v}$? A method to find reasonable $\mathbf{u}$ and $\mathbf{v}$ is given in Section 2.4.6.

If we have a rotation matrix and we wish to have the rotation in axis-angle form, we can compute the one real eigenvalue (which will be $\lambda = 1$), and the corresponding eigenvector is the axis of rotation. This is the one axis that is not changed by the rotation.

See Chapter 16 for a comparison of the few most-used ways to represent rotations, besides rotation matrices.

6.2.2 Transforming Normal Vectors

While most 3D vectors we use represent positions (offset vectors from the origin) or directions, such as where light comes from, some vectors represent *surface normals*. Surface normal vectors are perpendicular to the tangent plane of a surface. These normals do not transform the way we would like when the underlying surface is transformed. For example, if the points of a surface are transformed by a matrix $\mathbf{M}$, a vector $\mathbf{t}$ that is tangent to the surface and is multiplied by $\mathbf{M}$ will be tangent to the transformed surface. However, a surface normal vector $\mathbf{n}$ that is transformed by $\mathbf{M}$ may not be normal to the transformed surface (Figure 6.17).

We can derive a transform matrix $\mathbf{N}$ which does take $\mathbf{n}$ to a vector perpendicular to the transformed surface. One way to attack this issue is to note that a surface normal vector and a tangent vector are perpendicular, so their dot product is zero, which is expressed in matrix form as

$$\mathbf{n}^{\mathrm{T}}\mathbf{t} = \mathbf{0}. \tag{6.4}$$

If we denote the desired transformed vectors as $\mathbf{t}_M = \mathbf{M}\mathbf{t}$ and $\mathbf{n}_N = \mathbf{N}\mathbf{n}$, our goal is to find $\mathbf{N}$ such that $\mathbf{n}_N^{\mathrm{T}}\mathbf{t}_M = \mathbf{0}$. We can find $\mathbf{N}$ by some algebraic

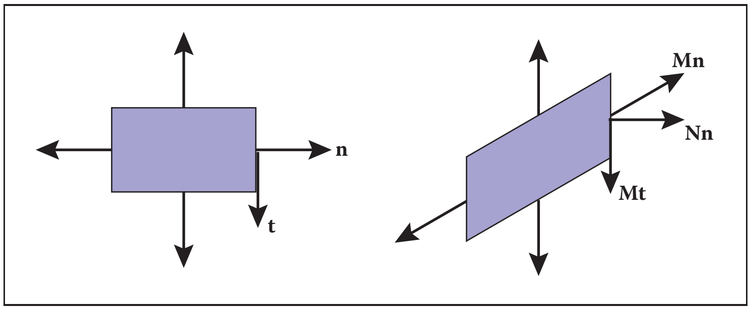


Figure 6.17. When a normal vector is transformed using the same matrix that transforms the points on an object, the resulting vector may not be perpendicular to the surface as is shown here for the sheared rectangle. The tangent vector, however, does transform to a vector tangent to the transformed surface.

tricks. First, we can sneak an identity matrix into the dot product, and then take advantage of $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$:

$$\mathbf{n}^{\mathrm{T}}\mathbf{t} = \mathbf{n}^{\mathrm{T}}\mathbf{I}\mathbf{t} = \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{M}\mathbf{t} = \mathbf{0}.$$

Although the manipulations above don't obviously get us anywhere, note that we can add parentheses that make the above expression more obviously a dot product:

$$\left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)(\mathbf{M}\mathbf{t}) = \left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)\mathbf{t}_M = \mathbf{0}.$$

This means that the row vector that is perpendicular to $\mathbf{t}_M$ is the left part of the expression above. This expression holds for any of the tangent vectors in the tangent plane. Since there is only one direction in 3D (and its opposite) that is perpendicular to all such tangent vectors, we know that the left part of the expression above must be the row vector expression for $\mathbf{n}_N$, i.e., it is $\mathbf{n}_N^{\mathrm{T}}$, so this allows us to infer $\mathbf{N}$:

$$\mathbf{n}_N^{\mathrm{T}} = \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1},$$

so we can take the transpose of that to get

$$\mathbf{n}_N = \left(\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\right)^{\mathrm{T}} = \left(\mathbf{M}^{-1}\right)^{\mathrm{T}}\mathbf{n}. \tag{6.5}$$

Therefore, we can see that the matrix which correctly transforms normal vectors so they remain normal is $\mathbf{N} = (\mathbf{M}^{-1})^{\mathrm{T}}$, i.e., the transpose of the inverse matrix. Since this matrix may change the length of $\mathbf{n}$, we can multiply it by an arbitrary scalar and it will still produce $\mathbf{n}_N$ with the right direction. Recall from Section 5.3 that the inverse of a matrix is the transpose of the cofactor matrix divided by the determinant. Because we don't care about the length of a normal vector, we can skip the division and find that for a 3×3 matrix,

$$\mathbf{N} = \begin{bmatrix} m^c_{11} & m^c_{12} & m^c_{13} \\ m^c_{21} & m^c_{22} & m^c_{23} \\ m^c_{31} & m^c_{32} & m^c_{33} \end{bmatrix}.$$

This assumes the element of $\mathbf{M}$ in row i and column j is m_{ij}. So the full expression for $\mathbf{N}$ is

$$\mathbf{N} = \begin{bmatrix} m_{22}m_{33} - m_{23}m_{32} & m_{23}m_{31} - m_{21}m_{33} & m_{21}m_{32} - m_{22}m_{31} \\ m_{13}m_{32} - m_{12}m_{33} & m_{11}m_{33} - m_{13}m_{31} & m_{12}m_{31} - m_{11}m_{32} \\ m_{12}m_{23} - m_{13}m_{22} & m_{13}m_{21} - m_{11}m_{23} & m_{11}m_{22} - m_{12}m_{21} \end{bmatrix}.$$

6.3 Translation and Affine Transformations

We have been looking at methods to change vectors using a matrix $\mathbf{M}$. In two dimensions, these transforms have the form,

$$\begin{array}{ccccc} x' & = & m_{11}x & + & m_{12}y, \\ y' & = & m_{21}x & + & m_{22}y. \end{array}$$

We cannot use such transforms to *move* objects, only to scale and rotate them. In particular, the origin $(0, 0)$ always remains fixed under a linear transformation. To move, or *translate*, an object by shifting all its points the same amount, we need a transform of the form,

$$\begin{array}{ccccc} x' & = & x & + & x_t, \\ y' & = & y & + & y_t. \end{array}$$

There is just no way to do that by multiplying (x, y) by a 2×2 matrix. One possibility for adding translation to our system of linear transformations is to simply associate a separate translation vector with each transformation matrix, letting the matrix take care of scaling and rotation and the vector take care of translation. This is perfectly feasible, but the bookkeeping is awkward and the rule for composing two transformations is not as simple and clean as with linear transformations.

Instead, we can use a clever trick to get a single matrix multiplication to do both operations together. The idea is simple: represent the point (x, y) by a 3D vector $[x\ y\ 1]^{\mathrm{T}}$, and use 3×3 matrices of the form

$$\begin{bmatrix} m_{11} & m_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix}.$$

The fixed third row serves to copy the 1 into the transformed vector, so that all vectors have a 1 in the last place, and the first two rows compute x' and y' as linear combinations of x, y, and 1:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}x + m_{12}y + x_t \\ m_{21}x + m_{22}y + y_t \\ 1 \end{bmatrix}.$$

The single matrix implements a linear transformation followed by a translation! This kind of transformation is called an *affine transformation*, and this way of implementing affine transformations by adding an extra dimension is called *homogeneous coordinates* (Roberts, 1965; Riesenfeld, 1981; Penna & Patterson, 1986). Homogeneous coordinates not only clean up the code for transformations, but this scheme also makes it obvious how to compose two affine transformations: simply multiply the matrices.

A problem with this new formalism arises when we need to transform vectors that are not supposed to be positions—they represent directions, or offsets between positions. Vectors that represent directions or offsets should not change when we translate an object. Fortunately, we can arrange for this by setting the third coordinate to zero:

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}.$$

If there is a scaling/rotation transformation in the upper-left 2×2 entries of the matrix, it will apply to the vector, but the translation still multiplies with the zero and is ignored. Furthermore, the zero is copied into the transformed vector, so direction vectors remain direction vectors after they are transformed.

This is exactly the behavior we want for vectors, so they fit smoothly into the system: the extra (third) coordinate will be either 1 or 0 depending on whether we are encoding a position or a direction. We actually do need to store the homogeneous coordinate so we can distinguish between locations and other vectors. For example,

This gives an explanation for the name "homogeneous:" translation, rotation, and scaling of positions and directions all fit into a single system.

$$\begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \text{ is a location} \quad \text{and} \quad \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix} \text{ is a displacement or direction.}$$

Later, when we do perspective viewing, we will see that it is useful to allow the homogeneous coordinate to take on values other than one or zero.

Homogeneous coordinates are used nearly universally to represent transformations in graphics systems. In particular, homogeneous coordinates underlie the design and operation of renderers implemented in graphics hardware. We will see in Chapter 7 that homogeneous coordinates also make it easy to draw scenes in perspective, another reason for their popularity.

Homogeneous coordinates are also ubiquitous in computer vision.

Homogeneous coordinates can be considered just a clever way to handle the bookkeeping for translation, but there is also a different, geometric interpretation. The key observation is that when we do a 3D shear based on the z-coordinate we get this transform:

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + x_t z \\ y + y_t z \\ z \end{bmatrix}.$$

Note that this almost has the form we want in x and y for a 2D translation, but has a z hanging around that doesn't have a meaning in 2D. Now comes the key decision: we will add a coordinate $z = 1$ to all 2D locations. This gives us

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ 1 \end{bmatrix}.$$

By associating a $(z = 1)$-coordinate with all 2D points, we now can encode translations into matrix form. For example, to first translate in 2D by (x_t, y_t) and then rotate by angle ϕ we would use the matrix

$$\mathbf{M} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix}.$$

Note that the 2D rotation matrix is now 3×3 with zeros in the "translation slots." With this type of formalism, which uses shears along $z = 1$ to encode translations, we can represent any number of 2D shears, 2D rotations, and 2D translations as one composite 3D matrix. The bottom row of that matrix will always be $(0, 0, 1)$, so we don't really have to store it. We just need to remember it is there when we multiply two matrices together.

In 3D, the same technique works: we can add a fourth coordinate, a homogeneous coordinate, and then we have translations:

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{bmatrix}.$$

Again, for a direction vector, the fourth coordinate is zero and the vector is thus unaffected by translations.

Example (Windowing transformations). Often in graphics we need to create a transform matrix that takes points in the rectangle $[x_l, x_h] \times [y_l, y_h]$ to the rectangle $[x'_l, x'_h] \times [y'_l, y'_h]$. This can be accomplished with a single scale and translate in sequence. However, it is more intuitive to create the transform from a sequence of three operations (Figure 6.18):

1. Move the point (x_l, y_l) to the origin.

2. Scale the rectangle to be the same size as the target rectangle.

3. Move the origin to point (x'_l, y'_l).

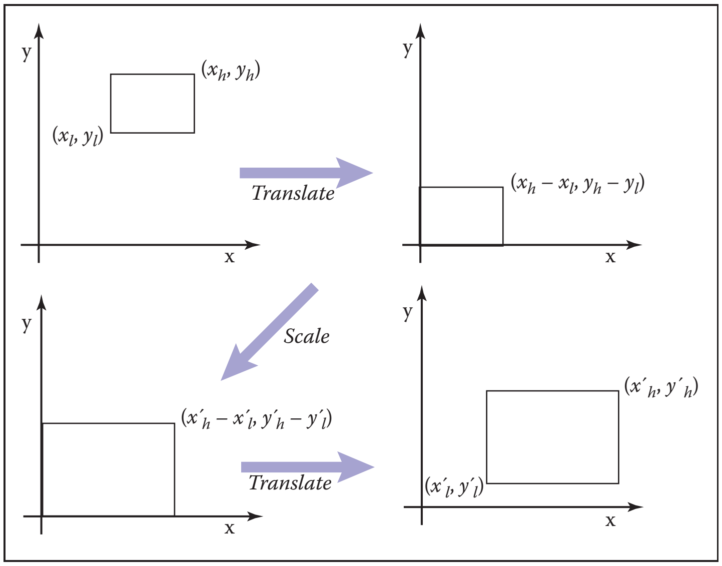


Figure 6.18. To take one rectangle (window) to the other, we first shift the lower-left corner to the origin, then scale it to the new size, and then move the origin to the lower-left corner of the target rectangle.

Remembering that the right-hand matrix is applied first, we can write

$$\begin{aligned}
\text{window} &= \text{translate}\,(x'_l, y'_l)\ \text{scale}\left(\frac{x'_h - x'_l}{x_h - x_l}, \frac{y'_h - y'_l}{y_h - y_l}\right)\ \text{translate}\,(-x_l, -y_l) \\
&= \begin{bmatrix} 1 & 0 & x'_l \\ 0 & 1 & y'_l \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_l \\ 0 & 1 & -y_l \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & 1 \end{bmatrix}. \qquad (6.6)
\end{aligned}$$

It is perhaps not surprising to some readers that the resulting matrix has the form it does, but the constructive process with the three matrices leaves no doubt as to the correctness of the result.

An exactly analogous construction can be used to define a 3D windowing transformation, which maps the box $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ to the box

$[x'_l, x'_h] \times [y'_l, y'_h] \times [z'_l, z'_h]$:

$$\begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & \frac{z'_h - z'_l}{z_h - z_l} & \frac{z'_l z_h - z'_h z_l}{z_h - z_l} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.7}$$

It is interesting to note that if we multiply an arbitrary matrix composed of scales, shears, and rotations with a simple translation (translation comes second), we get

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Thus, we can look at any matrix and think of it as a scaling/rotation part and a translation part because the components are nicely separated from each other.

An important class of transforms are *rigid-body* transforms. These are composed only of translations and rotations, so they have no stretching or shrinking of the objects. Such transforms will have a pure rotation for the a_{ij} above.

6.4 Inverses of Transformation Matrices

While we can always invert a matrix algebraically, we can use geometry if we know what the transform does. For example, the inverse of scale(s_x, s_y, s_z) is scale$(1/s_x, 1/s_y, 1/s_z)$. The inverse of a rotation is the same rotation with the opposite sign on the angle. The inverse of a translation is a translation in the opposite direction. If we have a series of matrices $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2 \cdots \mathbf{M}_n$ then $\mathbf{M}^{-1} = \mathbf{M}_n^{-1} \cdots \mathbf{M}_2^{-1}\mathbf{M}_1^{-1}$.

Also, certain types of transformation matrices are easy to invert. We've already mentioned scales, which are diagonal matrices; the second important example is rotations, which are orthogonal matrices. Recall (Section 5.2.4) that the inverse of an orthogonal matrix is its transpose. This makes it easy to invert rotations and rigid body transformations (see Exercise 6). Also, it's useful to know that a matrix with $[0\ 0\ 0\ 1]$ in the bottom row has an inverse that also has $[0\ 0\ 0\ 1]$ in the bottom row (see Exercise 7).

Interestingly, we can use SVD to invert a matrix as well. Since we know that any matrix can be decomposed into a rotation times a scale times a rotation, inversion is straightforward. For example, in 3D we have

$$\mathbf{M} = \mathbf{R}_1 \text{scale}(\sigma_1, \sigma_2, \sigma_3)\mathbf{R}_2,$$

and from the rules above it follows easily that

$$\mathbf{M}^{-1} = \mathbf{R}_2^{\mathrm{T}} \text{scale}(1/\sigma_1, 1/\sigma_2, 1/\sigma_3)\mathbf{R}_1^{\mathrm{T}}.$$

6.5 Coordinate Transformations

All of the previous discussion has been in terms of using transformation matrices to move points around. We can also think of them as simply changing the coordinate system in which the point is represented. For example, in Figure 6.19, we see two ways to visualize a movement. In different contexts, either interpretation may be more suitable.

For example, a driving game may have a model of a city and a model of a car. If the player is presented with a view out the windshield, objects inside the car are always drawn in the same place on the screen, while the streets and buildings appear to move backward as the player drives. On each frame, we apply a transformation to these objects that moves them farther back than on the previous frame. One way to think of this operation is simply that it moves the buildings backward; another way to think of it is that the buildings are staying put but the coordinate system in which we want to draw them—which is attached to the car—is moving. In the second interpretation, the transformation is changing

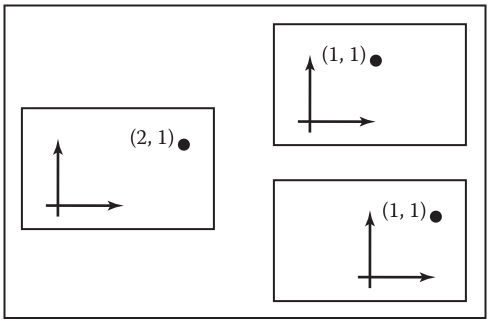


Figure 6.19. The point (2,1) has a transform "translate by (-1,0)" applied to it. On the top right is our mental image if we view this transformation as a physical movement, and on the bottom right is our mental image if we view it as a change of coordinates (a movement of the origin in this case). The artificial boundary is just an artifice, and the relative position of the axes and the point are the same in either case.

the coordinates of the city geometry, expressing them as coordinates in the car's coordinate system. Both ways will lead to exactly the same matrix that is applied to the geometry outside the car.

If the game also supports an overhead view to show where the car is in the city, the buildings and streets need to be drawn in fixed positions while the car needs to move from frame to frame. The same two interpretations apply: we can think of the changing transformation as moving the car from its canonical position to its current location in the world; or we can think of the transformation as simply changing the coordinates of the car's geometry, which is originally expressed in terms of a coordinate system attached to the car, to express them instead in a coordinate system fixed relative to the city. The change-of-coordinates interpretation makes it clear that the matrices used in these two modes (city-to-car coordinate change vs. car-to-city coordinate change) are inverses of one another.

The idea of changing coordinate systems is much like the idea of type conversions in programming. Before we can add a floating-point number to an integer, we need to convert the integer to floating point or the floating-point number to an integer, depending on our needs, so that the types match. And before we can draw the city and the car together, we need to convert the city to car coordinates or the car to city coordinates, depending on our needs, so that the coordinates match.

When managing multiple coordinate systems, it's easy to get confused and wind up with objects in the wrong coordinates, causing them to show up in unexpected places. But with systematic thinking about transformations between coordinate systems, you can reliably get the transformations right.

Geometrically, a coordinate system, or coordinate *frame*, consists of an origin and a basis—a set of three vectors. Orthonormal bases are so convenient that we'll normally assume frames are orthonormal unless otherwise specified. In a frame with origin $\mathbf{p}$ and basis $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$, the coordinates (u, v, w) describe the point

In 2D, of course, there are two basis vectors.

$$\mathbf{p} + u\mathbf{u} + v\mathbf{v} + w\mathbf{w}.$$

When we store these vectors in the computer, they need to be represented in terms of some coordinate system. To get things started, we have to designate some canonical coordinate system, often called "global" or "world" coordinates, which is used to describe all other systems. In the city example, we might adopt the street grid and use the convention that the x-axis points along Main Street, the y-axis points up, and the z-axis points along Central Avenue. Then when we write the origin and basis of the car frame in terms of these coordinates it is clear what we mean.

In 2D our convention is is to use the point $\mathbf{o}$ for the origin, and $\mathbf{x}$ and $\mathbf{y}$ for the right-handed orthonormal basis vectors $\mathbf{x}$ and $\mathbf{y}$ (Figure 6.20).

In 2D, right-handed means $\mathbf{y}$ is counterclockwise from $\mathbf{x}$.

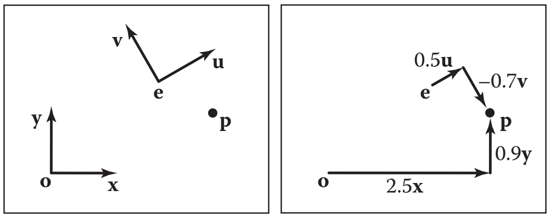


Figure 6.20. The point $\mathbf{p}$ can be represented in terms of either coordinate system.

Another coordinate system might have an origin $\mathbf{e}$ and right-handed orthonormal basis vectors $\mathbf{u}$ and $\mathbf{v}$. Note that typically the canonical data $\mathbf{o}$, $\mathbf{x}$, and $\mathbf{y}$ are never stored explicitly. They are the frame-of-reference for all other coordinate systems. In that coordinate system, we often write down the location of $\mathbf{p}$ as an ordered pair, which is shorthand for a full vector expression:

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y}.$$

For example, in Figure 6.20, $(x_p, y_p) = (2.5, 0.9)$. Note that the pair (x_p, y_p) implicitly assumes the origin $\mathbf{o}$. Similarly, we can express $\mathbf{p}$ in terms of another equation:

$$\mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p\mathbf{u} + v_p\mathbf{v}.$$

In Figure 6.20, this has $(u_p, v_p) = (0.5, -0.7)$. Again, the origin $\mathbf{e}$ is left as an implicit part of the coordinate system associated with $\mathbf{u}$ and $\mathbf{v}$.

We can express this same relationship using matrix machinery, like this:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}.$$

Note that this assumes we have the point $\mathbf{e}$ and vectors $\mathbf{u}$ and $\mathbf{v}$ stored in canonical coordinates; the (x, y)-coordinate system is the first among equals. In terms of the basic types of transformations we've discussed in this chapter, this is a rotation (involving $\mathbf{u}$ and $\mathbf{v}$) followed by a translation (involving $\mathbf{e}$). Looking at the matrix for the rotation and translation together, you can see it's very easy to write down: we just put $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{e}$ into the columns of a matrix, with the usual $[0\ 0\ 1]$ in the third row. To make this even clearer we can write the matrix like this:

$$\mathbf{p}_{xy} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uv}.$$

We call this matrix the *frame-to-canonical* matrix for the (u, v) frame. It takes points expressed in the (u, v) frame and converts them to the same points expressed in the canonical frame.

The name "frame-to-canonical" is based on thinking about changing the coordinates of a vector from one system to another. Thinking in terms of moving vectors around, the frame-to-canonical matrix maps the canonical frame to the (u,v) frame.

To go in the other direction we have

$$\begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_e \\ 0 & 1 & -y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}.$$

This is a translation followed by a rotation; they are the inverses of the rotation and translation we used to build the frame-to-canonical matrix, and when multiplied together they produce the inverse of the frame-to-canonical matrix, which is (not surprisingly) called the canonical-to-frame matrix:

$$\mathbf{p}_{uv} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xy}.$$

The canonical-to-frame matrix takes points expressed in the canonical frame and converts them to the same points expressed in the (u,v) frame. We have written this matrix as the inverse of the frame-to-canonical matrix because it can't immediately be written down using the canonical coordinates of $\mathbf{e}$, $\mathbf{u}$, and $\mathbf{v}$. But remember that all coordinate systems are equivalent; it's only our convention of storing vectors in terms of x- and y-coordinates that creates this seeming asymmetry. The canonical-to-frame matrix *can* be expressed simply in terms of the (u, v) coordinates of $\mathbf{o}$, $\mathbf{x}$, and $\mathbf{y}$:

$$\mathbf{p}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}.$$

All these ideas work strictly analogously in 3D, where we have

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix}$$
$$\mathbf{p}_{xyz} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uvw}, \tag{6.8}$$

and

$$\begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$
$$\mathbf{p}_{uvw} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xyz}. \tag{6.9}$$

Frequently Asked Questions

- **Can't I just hardcode transforms rather than use the matrix formalisms?**

Yes, but in practice it is harder to derive, harder to debug, and not any more efficient. Also, all current graphics APIs use this matrix formalism so it must be understood even to use graphics libraries.

- **The bottom row of the matrix is always (0,0,0,1). Do I have to store it?**

You do not have to store it unless you include perspective transforms (Chapter 7).

Notes

The derivation of the transformation properties of normals is based on *Properties of Surface Normal Transformations* (Turkowski, 1990). In many treatments through the mid-1990s, vectors were represented as row vectors and premultiplied, e.g., $\mathbf{b} = \mathbf{a}\mathbf{M}$. In our notation this would be $\mathbf{b}^{\mathrm{T}} = \mathbf{a}^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}$. If you want to find a rotation matrix $\mathbf{R}$ that takes one vector $\mathbf{a}$ to a vector $\mathbf{b}$ of the same length: $\mathbf{b} = \mathbf{R}\mathbf{a}$ you could use two rotations constructed from orthonormal bases. A more efficient method is given in *Efficiently Building a Matrix to Rotate One Vector to Another* (Akenine-Möller, Haines, & Hoffman, 2008).

Exercises

1. Write down the 4×4 3D matrix to move by (x_m, y_m, z_m).
2. Write down the 4×4 3D matrix to rotate by an angle θ about the y-axis.
3. Write down the 4×4 3D matrix to scale an object by 50% in all directions.
4. Write the 2D rotation matrix that rotates by 90 degrees clockwise.
5. Write the matrix from Exercise 4 as a product of three shear matrices.
6. Find the inverse of the rigid body transformation:

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

where $\mathbf{R}$ is a 3×3 rotation matrix and $\mathbf{t}$ is a 3-vector.

7. Show that the inverse of the matrix for an affine transformation (one that has all zeros in the bottom row except for a one in the lower right entry) also has the same form.

8. Describe in words what this 2D transform matrix does:

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

9. Write down the 3×3 matrix that rotates a 2D point by angle θ about a point $\mathbf{p} = (x_p, y_p)$.

10. Write down the 4×4 rotation matrix that takes the orthonormal 3D vectors $\mathbf{u} = (x_u, y_u, z_u)$, $\mathbf{v} = (x_v, y_v, z_v)$, and $\mathbf{w} = (x_w, y_w, z_w)$, to orthonormal 3D vectors $\mathbf{a} = (x_a, y_a, z_a)$, $\mathbf{b} = (x_b, y_b, z_b)$, and $\mathbf{c} = (x_c, y_c, z_c)$, So $M\mathbf{u} = \mathbf{a}$, $M\mathbf{v} = \mathbf{b}$, and $M\mathbf{w} = \mathbf{c}$.

11. What is the inverse matrix for the answer to the previous problem?

7

Viewing

In the previous chapter, we saw how to use matrix transformations as a tool for arranging geometric objects in 2D or 3D space. A second important use of geometric transformations is in moving objects between their 3D locations and their positions in a 2D view of the 3D world. This 3D to 2D mapping is called a *viewing transformation*, and it plays an important role in object-order rendering, in which we need to rapidly find the image-space location of each object in the scene.

When we studied ray tracing in Chapter 4, we covered the different types of perspective and orthographic views and how to generate viewing rays according to any given view. This chapter is about the inverse of that process. Here we explain how to use matrix transformations to express any parallel or perspective view. The transformations in this chapter project 3D points in the scene (world space) to 2D points in the image (image space), and they will project any point on a given pixel's viewing ray back to that pixel's position in image space.

If you have not looked at it recently, it is advisable to review the discussion of perspective and ray generation in Chapter 4 before reading this chapter.

By itself, the ability to project points from the world to the image is only good for producing *wireframe* renderings—renderings in which only the edges of objects are drawn, and closer surfaces do not occlude more distant surfaces (Figure 7.1). Just as a ray tracer needs to find the closest surface intersection along each viewing ray, an object-order renderer displaying solid-looking objects has to work out which of the (possibly many) surfaces drawn at any given point on the screen is closest and display only that one. In this chapter, we assume we

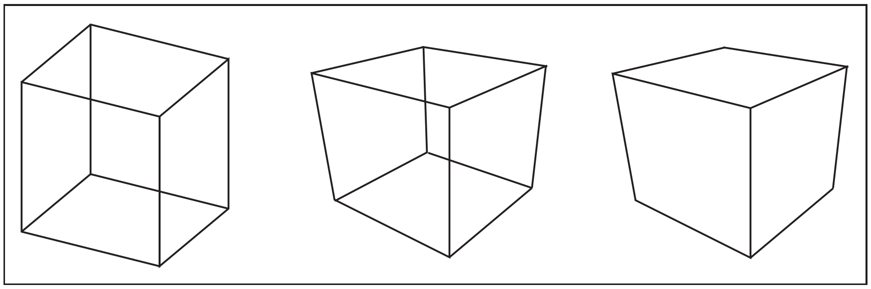


Figure 7.1. Left: wireframe cube in orthographic projection. Middle: wireframe cube in perspective projection. Right: perspective projection with hidden lines removed.

are drawing a model consisting only of 3D line segments that are specified by the (x, y, z) coordinates of their two endpoints. Later chapters will discuss the machinery needed to produce renderings of solid surfaces.

7.1 Viewing Transformations

The viewing transformation has the job of mapping 3D locations, represented as (x, y, z) coordinates in the canonical coordinate system, to coordinates in the image, expressed in units of pixels. It is a complicated beast that depends on many different things, including the camera position and orientation, the type of projection, the field of view, and the resolution of the image. As with all complicated transformations it is best approached by breaking it up into a product of several simpler transformations. Most graphics systems do this by using a sequence of three transformations:

Some APIs use "viewing transformation" for just the piece of our viewing transformation that we call the camera transformation.

- A *camera transformation* or *eye transformation*, which is a rigid body transformation that places the camera at the origin in a convenient orientation. It depends only on the position and orientation, or *pose*, of the camera.
- A *projection transformation*, which projects points from camera space so that all visible points fall in the range -1 to 1 in x and y. It depends only on the type of projection desired.
- A *viewport transformation* or *windowing transformation*, which maps this unit image rectangle to the desired rectangle in pixel coordinates. It depends only on the size and position of the output image.

To make it easy to describe the stages of the process (Figure 7.2), we give names to the coordinate systems that are the inputs and output of these transformations.

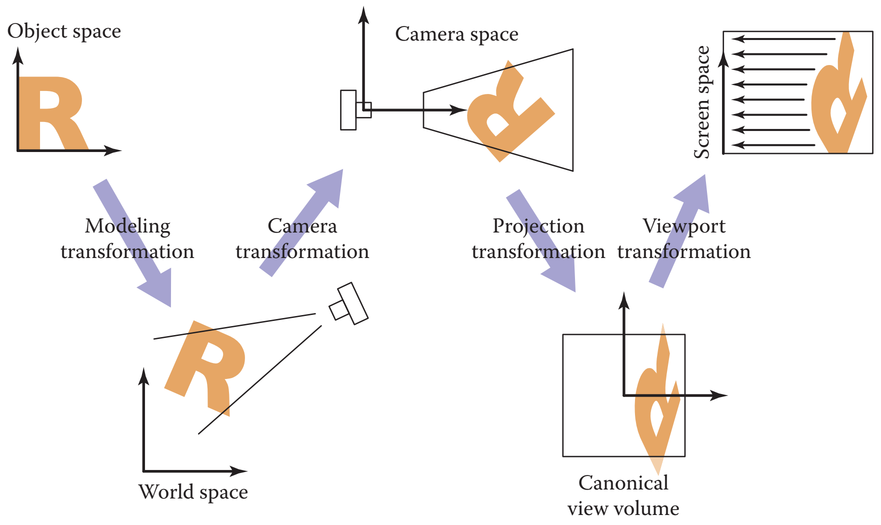


Figure 7.2. The sequence of spaces and transformations that gets objects from their original coordinates into screen space.

The camera transformation converts points in canonical coordinates (or world space) to *camera coordinates* or places them in *camera space*. The projection transformation moves points from camera space to the *canonical view volume*. Finally, the viewport transformation maps the canonical view volume to *screen space*.

Other names: camera space is also "eye space" and the camera transformation is sometimes the "viewing transformation;" the canonical view volume is also "clip space" or "normalized device coordinates;" screen space is also "pixel coordinates."

Each of these transformations is individually quite simple. We'll discuss them in detail for the orthographic case beginning with the viewport transformation, then cover the changes required to support perspective projection.

7.1.1 The Viewport Transformation

We begin with a problem whose solution will be reused for any viewing condition. We assume that the geometry we want to view is in the *canonical view volume*, and we wish to view it with an orthographic camera looking in the $-z$ direction. The canonical view volume is the cube containing all 3D points whose Cartesian coordinates are between -1 and $+1$—that is, $(x, y, z) \in [-1, 1]^3$ (Figure 7.3). We project $x = -1$ to the left side of the screen, $x = +1$ to the right side of the screen, $y = -1$ to the bottom of the screen, and $y = +1$ to the top of the screen.

The word "canonical" crops up again—it means something arbitrarily chosen for convenience. For instance, the unit circle could be called the "canonical circle."

Recall the conventions for pixel coordinates from Chapter 3: each pixel "owns" a unit square centered at integer coordinates; the image boundaries have a half-unit overshoot from the pixel centers; and the smallest pixel center coordinates are $(0, 0)$. If we are drawing into an image (or window on the screen) that has n_x by n_y pixels, we need to map the square $[-1, 1]^2$ to the rectangle $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

Mapping a square to a potentially non-square rectangle is not a problem; x and y just end up with different scale factors going from canonical to pixel coordinates.

For now, we will assume that all line segments to be drawn are completely inside the canonical view volume. Later we will relax that assumption when we discuss *clipping*.

Since the viewport transformation maps one axis-aligned rectangle to another, it is a case of the windowing transform given by Equation (6.6):

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}. \tag{7.1}$$

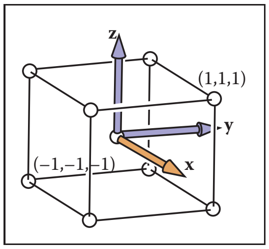


Figure 7.3. The canonical view volume is a cube with side of length two centered at the origin.

Note that this matrix ignores the z-coordinate of the points in the canonical view volume, because a point's distance along the projection direction doesn't affect where that point projects in the image. But before we officially call this the *viewport matrix*, we add a row and column to carry along the z-coordinate without changing it. We don't need it in this chapter, but eventually we will need the z values because they can be used to make closer surfaces hide more distant surfaces (see Section 8.2.3).

$$M_{\text{vp}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y - 1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.2}$$

7.1.2 The Orthographic Projection Transformation

Of course, we usually want to render geometry in some region of space other than the canonical view volume. Our first step in generalizing the view will keep the view direction and orientation fixed looking along $-z$ with $+y$ up, but will allow arbitrary rectangles to be viewed. Rather than replacing the viewport matrix, we'll augment it by multiplying it with another matrix on the right.

Under these constraints, the view volume is an axis-aligned box, and we'll name the coordinates of its sides so that the view volume is $[l, r] \times [b, t] \times [f, n]$ shown in Figure 7.4. We call this box the *orthographic view volume* and refer to

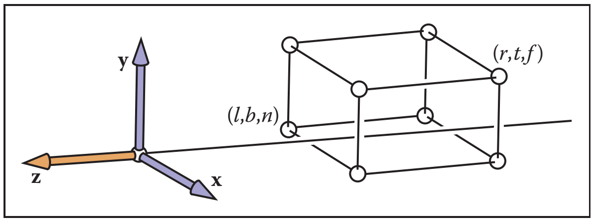


Figure 7.5. The orthographic view volume is along the negative z-axis, so f is a more negative number than n, thus $n > f$.

the bounding planes as follows:

$$\begin{aligned} x = l &\equiv \text{left plane},\\ x = r &\equiv \text{right plane},\\ y = b &\equiv \text{bottom plane},\\ y = t &\equiv \text{top plane},\\ z = n &\equiv \text{near plane},\\ z = f &\equiv \text{far plane}. \end{aligned}$$

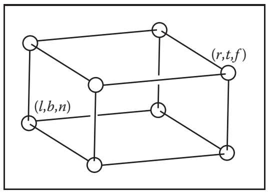


Figure 7.4. The orthographic view volume.

That vocabulary assumes a viewer who is looking along the *minus* z-axis with his head pointing in the y-direction.[1] This implies that $n > f$, which may be unintuitive, but if you assume the entire orthographic view volume has negative z values then the $z = n$ "near" plane is closer to the viewer if and only if $n > f$; here f is a smaller number than n, i.e., a negative number of larger absolute value than n.

This concept is shown in Figure 7.5. The transform from orthographic view volume to the canonical view volume is another windowing transform, so we can simply substitute the bounds of the orthographic and canonical view volumes into Equation (6.7) to obtain the matrix for this transformation:

This matrix is very close to the one used traditionally in OpenGL, except that n, f, and $z_{\text{canonical}}$ all have the opposite sign.

$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (7.3)$$

[1] Most programmers find it intuitive to have the x-axis pointing right and the y-axis pointing up. In a right-handed coordinate system, this implies that we are looking in the $-z$ direction. Some systems use a left-handed coordinate system for viewing so that the gaze direction is along $+z$. Which is best is a matter of taste, and this text assumes a right-handed coordinate system. A reference that argues for the left-handed system instead is given in the notes at the end of the chapter.

To draw 3D line segments in the orthographic view volume, we project them into screen x- and y-coordinates and ignore z-coordinates. We do this by combining Equations (7.2) and (7.3). Note that in a program we multiply the matrices together to form one matrix and then manipulate points as follows:

$$\begin{bmatrix} x_{\text{pixel}} \\ y_{\text{pixel}} \\ z_{\text{canonical}} \\ 1 \end{bmatrix} = (\mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The z-coordinate will now be in $[-1, 1]$. We don't take advantage of this now, but it will be useful when we examine z-buffer algorithms.

The code to draw many 3D lines with endpoints $\mathbf{a}_i$ and $\mathbf{b}_i$ thus becomes both simple and efficient:

This is a first example of how matrix transformation machinery makes graphics programs clean and efficient.

```
construct $\mathbf{M}_{\text{vp}}$
construct $\mathbf{M}_{\text{orth}}$
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
    drawline$(x_p, y_p, x_q, y_q)$
```

7.1.3 The Camera Transformation

We'd like to be able to change the viewpoint in 3D and look in any direction. There are a multitude of conventions for specifying viewer position and orientation. We will use the following one (see Figure 7.6):

- the eye position $\mathbf{e}$,
- the gaze direction $\mathbf{g}$,
- the view-up vector $\mathbf{t}$.

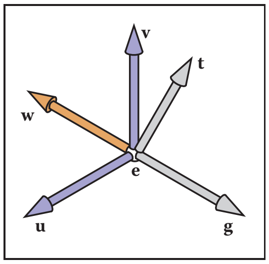


Figure 7.6. The user specifies viewing as an eye position **e**, a gaze direction **g**, and an up vector **t**. We construct a right-handed basis with **w** pointing opposite to the gaze and **v** being in the same plane as **g** and **t**.

The eye position is a location that the eye "sees from." If you think of graphics as a photographic process, it is the center of the lens. The gaze direction is any vector in the direction that the viewer is looking. The view-up vector is any vector in the plane that both bisects the viewer's head into right and left halves and points "to the sky" for a person standing on the ground. These vectors provide us with enough information to set up a coordinate system with origin $\mathbf{e}$ and a $\mathbf{uvw}$ basis,

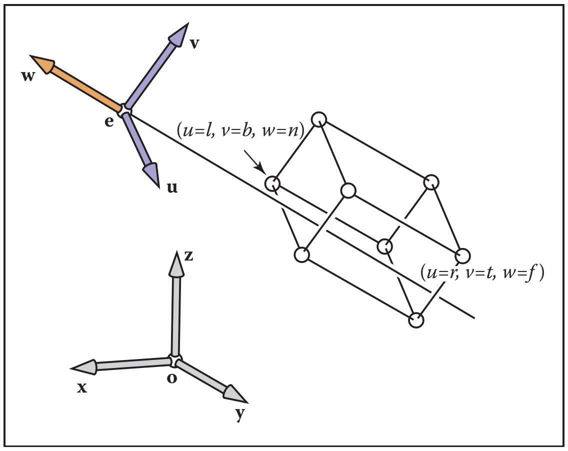


Figure 7.7. For arbitrary viewing, we need to change the points to be stored in the "appropriate" coordinate system. In this case it has origin **e** and offset coordinates in terms of **uvw**.

using the construction of Section 2.4.7:

$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|},$$

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

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

Our job would be done if all points we wished to transform were stored in coordinates with origin $\mathbf{e}$ and basis vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$. But as shown in Figure 7.7, the coordinates of the model are stored in terms of the canonical (or world) origin $\mathbf{o}$ and the $\mathbf{x}$-, $\mathbf{y}$-, and $\mathbf{z}$-axes. To use the machinery we have already developed, we just need to convert the coordinates of the line segment endpoints we wish to draw from xyz-coordinates into uvw-coordinates. This kind of transformation was discussed in Section 6.5, and the matrix that enacts this transformation is the canonical-to-basis matrix of the camera's coordinate frame:

$$\mathbf{M}_{\text{cam}} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (7.4)$$

Alternatively, we can think of this same transformation as first moving $\mathbf{e}$ to the origin, then aligning $\mathbf{u}, \mathbf{v}, \mathbf{w}$ to $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

To make our previously z-axis-only viewing algorithm work for cameras with any location and orientation, we just need to add this camera transformation to the product of the viewport and projection transformations, so that it converts the incoming points from world to camera coordinates before they are projected:

```
construct $\mathbf{M}_{\text{vp}}$
construct $\mathbf{M}_{\text{orth}}$
construct $\mathbf{M}_{\text{cam}}$
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{M}_{\text{cam}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
    drawline$(x_p, y_p, x_q, y_q)$
```

Again, almost no code is needed once the matrix infrastructure is in place.

7.2 Projective Transformations

We have left perspective for last because it takes a little bit of cleverness to make it fit into the system of vectors and matrix transformations that has served us so well up to now. To see what we need to do, let's look at what the perspective projection transformation needs to do with points in camera space. Recall that the viewpoint is positioned at the origin and the camera is looking along the z-axis.

For the moment we will ignore the sign of z to keep the equations simpler, but it will return on page 150.

The key property of perspective is that the size of an object on the screen is proportional to $1/z$ for an eye at the origin looking up the negative z-axis. This can be expressed more precisely in an equation for the geometry in Figure 7.8:

$$y_s = \frac{d}{z} y, \tag{7.5}$$

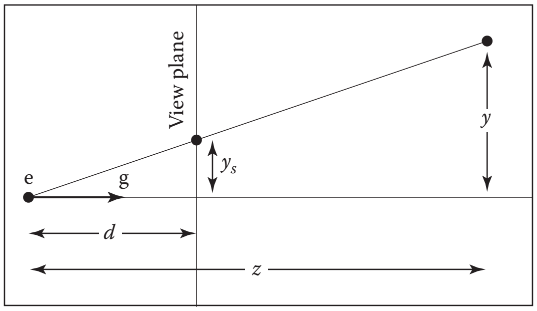


Figure 7.8. The geometry for Equation (7.5). The viewer's eye is at **e** and the gaze direction is **g** (the minus z-axis). The view plane is a distance d from the eye. A point is projected toward **e** and where it intersects the view plane is where it is drawn.

where y is the distance of the point along the y-axis, and y_s is where the point should be drawn on the screen.

We would really like to use the matrix machinery we developed for orthographic projection to draw perspective images; we could then just multiply another matrix into our composite matrix and use the algorithm we already have. However, this type of transformation, in which one of the coordinates of the input vector appears in the denominator, can't be achieved using affine transformations.

We can allow for division with a simple generalization of the mechanism of homogeneous coordinates that we have been using for affine transformations. We have agreed to represent the point (x, y, z) using the homogeneous vector $[x\ y\ z\ 1]^{\mathrm{T}}$; the extra coordinate, w, is always equal to 1, and this is ensured by always using $[0\ 0\ 0\ 1]^{\mathrm{T}}$ as the fourth row of an affine transformation matrix.

Rather than just thinking of the 1 as an extra piece bolted on to coerce matrix multiplication to implement translation, we now define it to be the denominator of the x-, y-, and z-coordinates: the homogeneous vector $[x\ y\ z\ w]^{\mathrm{T}}$ represents the point $(x/w, y/w, z/w)$. This makes no difference when $w = 1$, but it allows a broader range of transformations to be implemented if we allow any values in the bottom row of a transformation matrix, causing w to take on values other than 1.

Concretely, linear transformations allow us to compute expressions like

$$x' = ax + by + cz$$

and affine transformations extend this to

$$x' = ax + by + cz + d.$$

Treating w as the denominator further expands the possibilities, allowing us to compute functions like

$$x' = \frac{ax + by + cz + d}{ex + fy + gz + h};$$

this could be called a “linear rational function” of x, y, and z. But there is an extra constraint—the denominators are the same for all coordinates of the transformed point:

$$\begin{aligned} x' &= \frac{a_1x + b_1y + c_1z + d_1}{ex + fy + gz + h}, \\ y' &= \frac{a_2x + b_2y + c_2z + d_2}{ex + fy + gz + h}, \\ z' &= \frac{a_3x + b_3y + c_3z + d_3}{ex + fy + gz + h}. \end{aligned}$$

Expressed as a matrix transformation,

$$\begin{bmatrix}\tilde{x}\\ \tilde{y}\\ \tilde{z}\\ \tilde{w}\end{bmatrix} = \begin{bmatrix}a_1 & b_1 & c_1 & d_1\\ a_2 & b_2 & c_2 & d_2\\ a_3 & b_3 & c_3 & d_3\\ e & f & g & h\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

and

$$(x', y', z') = (\tilde{x}/\tilde{w}, \tilde{y}/\tilde{w}, \tilde{z}/\tilde{w}).$$

A transformation like this is known as a *projective transformation* or a *homography*.

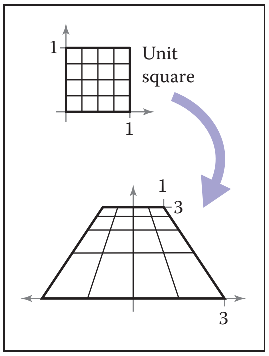


Figure 7.9. A projective transformation maps a square to a quadrilateral, preserving straight lines but not parallel lines.

Example. The matrix

$$\mathbf{M} = \begin{bmatrix}2 & 0 & -1\\ 0 & 3 & 0\\ 0 & \frac{2}{3} & \frac{1}{3}\end{bmatrix}$$

represents a 2D projective transformation that transforms the unit square ($[0,1] \times [0,1]$) to the quadrilateral shown in Figure 7.9.

For instance, the lower-right corner of the square at $(1,0)$ is represented by the homogeneous vector $[1\ 0\ 1]^{\mathrm{T}}$ and transforms as follows:

$$\begin{bmatrix}2 & 0 & -1\\ 0 & 3 & 0\\ 0 & \frac{2}{3} & \frac{1}{3}\end{bmatrix}\begin{bmatrix}1\\ 0\\ 1\end{bmatrix} = \begin{bmatrix}1\\ 0\\ \frac{1}{3}\end{bmatrix},$$

which represents the point $(1/\frac{1}{3}, 0/\frac{1}{3})$, or $(3,0)$. Note that if we use the matrix

$$3\mathbf{M} = \begin{bmatrix}6 & 0 & -3\\ 0 & 9 & 0\\ 0 & 2 & 1\end{bmatrix}$$

instead, the result is $[3\ 0\ 1]^{\mathrm{T}}$, which also represents $(3,0)$. In fact, any scalar multiple $c\mathbf{M}$ is equivalent: the numerator and denominator are both scaled by c, which does not change the result.

There is a more elegant way of expressing the same idea, which avoids treating the w-coordinate specially. In this view a 3D projective transformation is simply a 4D linear transformation, with the extra stipulation that all scalar multiples of a vector refer to the same point:

$$\mathbf{x} \sim \alpha\mathbf{x} \quad \text{for all } \alpha \neq 0.$$

The symbol $\sim$ is read as "is equivalent to" and means that the two homogeneous vectors both describe the same point in space.

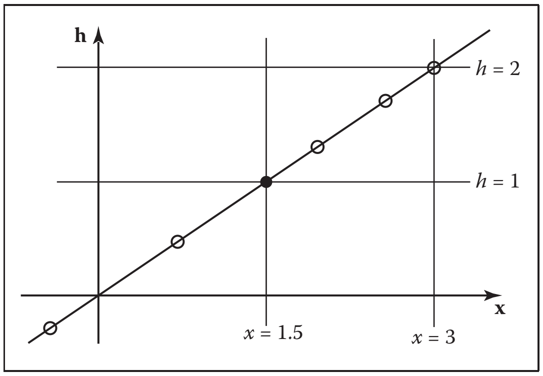


Figure 7.10. The point $x = 1.5$ is represented by any point on the line $x = 1.5h$, such as points at the hollow circles. However, before we interpret x as a conventional Cartesian coordinate, we first divide by h to get $(x,h) = (1.5,1)$ as shown by the black point.

Example. In 1D homogeneous coordinates, in which we use 2-vectors to represent points on the real line, we could represent the point (1.5) using the homogeneous vector $[1.5\ \ 1]^{\mathrm{T}}$, or any other point on the line $x = 1.5h$ in homogeneous space. (See Figure 7.10.)

In 2D homogeneous coordinates, in which we use 3-vectors to represent points in the plane, we could represent the point $(-1, -0.5)$ using the homogeneous vector $[-2; -1; 2]^{\mathrm{T}}$, or any other point on the line $\mathbf{x} = \alpha[-1\ \ -0.5\ \ 1]^{\mathrm{T}}$. Any homogeneous vector on the line can be mapped to the line's intersection with the plane $w = 1$ to obtain its Cartesian coordinates. (See Figure 7.11.)

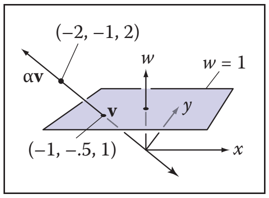


Figure 7.11. A point in homogeneous coordinates is equivalent to any other point on the line through it and the origin, and normalizing the point amounts to intersecting this line with the plane $w = 1$.

It's fine to transform homogeneous vectors as many times as needed, without worrying about the value of the w-coordinate—in fact, it is fine if the w-coordinate is zero at some intermediate phase. It is only when we want the ordinary Cartesian coordinates of a point that we need to normalize to an equivalent point that has $w = 1$, which amounts to dividing all the coordinates by w. Once we've done this we are allowed to read off the (x, y, z)-coordinates from the first three components of the homogeneous vector.

7.3 Perspective Projection

The mechanism of projective transformations makes it simple to implement the division by z required to implement perspective. In the 2D example shown in Figure 7.8, we can implement the perspective projection with a matrix transformation as follows:

$$\begin{bmatrix} y_s \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}.$$

This transforms the 2D homogeneous vector $[y; z; 1]^{\mathrm{T}}$ to the 1D homogeneous vector $[dy\ z]^{\mathrm{T}}$, which represents the 1D point (dy/z) (because it is equivalent to the 1D homogeneous vector $[dy/z\ 1]^{\mathrm{T}}$. This matches Equation (7.5).

For the "official" perspective projection matrix in 3D, we'll adopt our usual convention of a camera at the origin facing in the $-z$ direction, so the distance of the point (x, y, z) is $-z$. As with orthographic projection, we also adopt the notion of near and far planes that limit the range of distances to be seen. In this context, we will use the near plane as the projection plane, so the image plane distance is $-n$.

Remember, $n < 0$.

The desired mapping is then $y_s = (n/z)y$, and similarly for x. This transformation can be implemented by the *perspective matrix*:

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The first, second, and fourth rows simply implement the perspective equation. The third row, as in the orthographic and viewport matrices, is designed to bring the z-coordinate "along for the ride" so that we can use it later for hidden surface removal. In the perspective projection, though, the addition of a non-constant denominator prevents us from actually preserving the value of z—it's actually impossible to keep z from changing while getting x and y to do what we need them to do. Instead we've opted to keep z unchanged for points on the near or far planes.

More on this later.

There are many matrices that could function as perspective matrices, and all of them nonlinearly distort the z-coordinate. This specific matrix has the nice properties shown in Figures 7.12 and 7.13; it leaves points on the $(z = n)$-plane entirely alone, and it leaves points on the $(z = f)$-plane while "squishing" them in x and y by the appropriate amount. The effect of the matrix on a point (x, y, z) is

$$\mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \sim \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}.$$

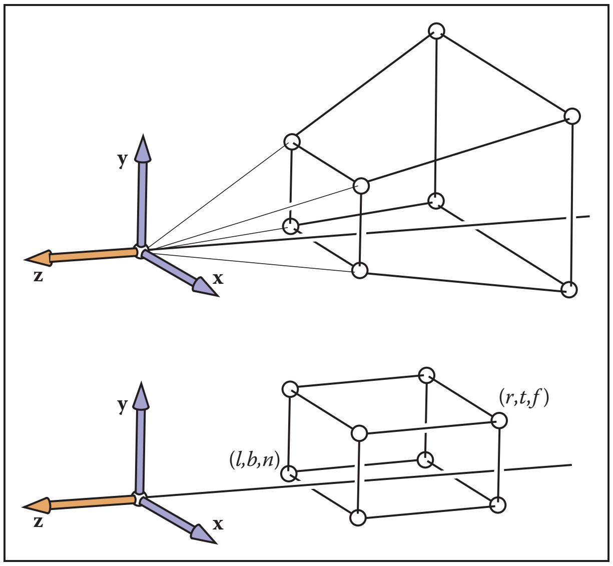


Figure 7.12. The perspective projection leaves points on the $z = n$ plane unchanged and maps the large $z = f$ rectangle at the back of the perspective volume to the small $z = f$ rectangle at the back of the orthographic volume.

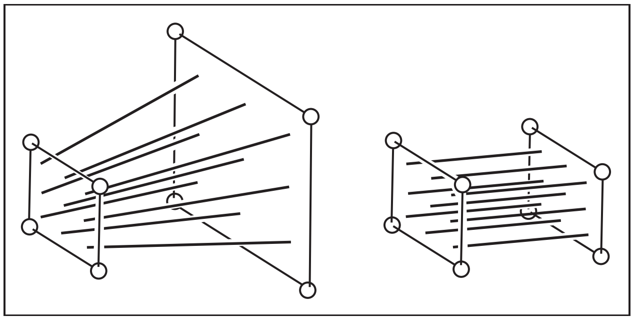


Figure 7.13. The perspective projection maps any line through the origin/eye to a line parallel to the z-axis and without moving the point on the line at $z = n$.

As you can see, x and y are scaled and, more importantly, divided by z. Because both n and z (inside the view volume) are negative, there are no "flips" in x and y. Although it is not obvious (see the exercise at the end of the chapter), the transform also preserves the relative order of z values between $z = n$ and $z = f$, allowing us to do depth ordering after this matrix is applied. This will be important later when we do hidden surface elimination.

Sometimes we will want to take the inverse of $\mathbf{P}$, for example, to bring a screen coordinate plus z back to the original space, as we might want to do for picking. The inverse is

$$\mathbf{P}^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}.$$

Since multiplying a homogeneous vector by a scalar does not change its meaning, the same is true of matrices that operate on homogeneous vectors. So we can write the inverse matrix in a prettier form by multiplying through by nf:

This matrix is not literally the inverse of the matrix **P**, but the transformation it describes *is* the inverse of the transformation described by **P**.

$$\mathbf{P}^{-1} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & fn \\ 0 & 0 & -1 & n+f \end{bmatrix}.$$

Taken in the context of the orthographic projection matrix $\mathbf{M}_{\text{orth}}$ in Equation (7.3), the perspective matrix simply maps the perspective view volume (which is shaped like a slice, or *frustum*, of a pyramid) to the orthographic view volume (which is an axis-aligned box). The beauty of the perspective matrix is that once we apply it, we can use an orthographic transform to get to the canonical view volume. Thus, all of the orthographic machinery applies, and all that we have added is one matrix and the division by w. It is also heartening that we are not "wasting" the bottom row of our four by four matrices!

Concatenating $\mathbf{P}$ with $\mathbf{M}_{\text{orth}}$ results in the *perspective projection matrix*,

$$\mathbf{M}_{\text{per}} = \mathbf{M}_{\text{orth}}\mathbf{P}.$$

One issue, however, is: How are *l,r,b,t* determined for perspective? They identify the "window" through which we look. Since the perspective matrix does not change the values of x and y on the $(z = n)$-plane, we can specify (l, r, b, t) on that plane.

To integrate the perspective matrix into our orthographic infrastructure, we simply replace $\mathbf{M}_{\text{orth}}$ with $\mathbf{M}_{\text{per}}$, which inserts the perspective matrix $\mathbf{P}$ after the camera matrix $\mathbf{M}_{\text{cam}}$ has been applied but before the orthographic projection. So the full set of matrices for perspective viewing is

$$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}.$$

The resulting algorithm is:

```
compute $\mathbf{M}_{\text{vp}}$
compute $\mathbf{M}_{\text{per}}$
compute $\mathbf{M}_{\text{cam}}$
```

```
$\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{per}}\mathbf{M}_{\text{cam}}$
for each line segment $(\mathbf{a}_i, \mathbf{b}_i)$ do
    $\mathbf{p} = \mathbf{M}\mathbf{a}_i$
    $\mathbf{q} = \mathbf{M}\mathbf{b}_i$
    drawline$(x_p/w_p, y_p/w_p, x_q/w_q, y_q/w_q)$
```

Note that the only change other than the additional matrix is the divide by the homogeneous coordinate w.

Multiplied out, the matrix $\mathbf{M}_{\text{per}}$ looks like this:

$$\mathbf{M}_{\text{per}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

This or similar matrices often appear in documentation, and they are less mysterious when one realizes that they are usually the product of a few simple matrices.

Example. Many APIs such as *OpenGL* (Shreiner, Neider, Woo, & Davis, 2004) use the same canonical view volume as presented here. They also usually have the user specify the absolute values of n and f. The projection matrix for *OpenGL* is

$$\mathbf{M}_{\text{OpenGL}} = \begin{bmatrix} \frac{2|n|}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2|n|}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{|n|+|f|}{|n|-|f|} & \frac{2|f||n|}{|n|-|f|} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Other APIs send n and f to 0 and 1, respectively. Blinn (J. Blinn, 1996) recommends making the canonical view volume $[0, 1]^3$ for efficiency. All such decisions will change the the projection matrix slightly.

7.4 Some Properties of the Perspective Transform

An important property of the perspective transform is that it takes lines to lines and planes to planes. In addition, it takes line segments in the view volume to line segments in the canonical volume. To see this, consider the line segment

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}).$$

When transformed by a 4×4 matrix $\mathbf{M}$, it is a point with possibly varying homogeneous coordinate:

$$\mathbf{Mq} + t(\mathbf{MQ} - \mathbf{Mq}) \equiv \mathbf{r} + t(\mathbf{R} - \mathbf{r}).$$

The homogenized 3D line segment is

$$\frac{\mathbf{r} + t(\mathbf{R} - \mathbf{r})}{w_r + t(w_R - w_r)}. \tag{7.6}$$

If Equation (7.6) can be rewritten in a form

$$\frac{\mathbf{r}}{w_r} + f(t)\left(\frac{\mathbf{R}}{w_R} - \frac{\mathbf{r}}{w_r}\right), \tag{7.7}$$

then all the homogenized points lie on a 3D line. Brute force manipulation of Equation (7.6) yields such a form with

$$f(t) = \frac{w_R t}{w_r + t(w_R - w_r)}. \tag{7.8}$$

It also turns out that the line segments do map to line segments preserving the ordering of the points (Exercise 8), i.e., they do not get reordered or "torn."

A byproduct of the transform taking line segments to line segments is that it takes the edges and vertices of a triangle to the edges and vertices of another triangle. Thus, it takes triangles to triangles and planes to planes.

7.5 Field-of-View

While we can specify any window using the (l, r, b, t) and n values, sometimes we would like to have a simpler system where we look through the center of the window. This implies the constraint that

$$\begin{aligned} l &= -r, \\ b &= -t. \end{aligned}$$

If we also add the constraint that the pixels are square, i.e., there is no distortion of shape in the image, then the ratio of r to t must be the same as the ratio of the number of horizontal pixels to the number of vertical pixels:

$$\frac{n_x}{n_y} = \frac{r}{t}.$$

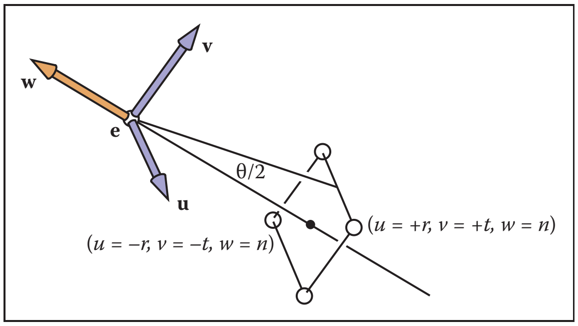


Figure 7.14. The field-of-view θ is the angle from the bottom of the screen to the top of the screen as measured from the eye.

Once n_x and n_y are specified, this leaves only one degree of freedom. That is often set using the *field-of-view* shown as θ in Figure 7.14. This is sometimes called the vertical field-of-view to distinguish it from the angle between left and right sides or from the angle between diagonal corners. From the figure we can see that

$$\tan\frac{\theta}{2} = \frac{t}{|n|}.$$

If n and θ are specified, then we can derive t and use code for the more general viewing system. In some systems, the value of n is hard-coded to some reasonable value, and thus we have one fewer degree of freedom.

Frequently Asked Questions

• Is orthographic projection ever useful in practice?

It is useful in applications where relative length judgments are important. It can also yield simplifications where perspective would be too expensive as occurs in some medical visualization applications.

• The tessellated spheres I draw in perspective look like ovals. Is this a bug?

No. It is correct behavior. If you place your eye in the same relative position to the screen as the virtual viewer has with respect to the viewport, then these ovals will look like circles because they themselves are viewed at an angle.