Chapter 5. Transformations of Objects

"Minus times minus is plus, the reason for this we need not discuss"
W.H.Auden

"If I eat one of these cakes", she thought,
"it's sure to make some change in my size."
So she swallowed one …
and was delighted to find that she began shrinking directly.
Lewis Carroll, Alice in Wonderland

Many of the brightly colored tile-covered walls and floors of the
Alhambra in Spain show us that the Moors were masters
in the art of filling a plane with similar interlocking figures,
bordering each other without gaps.
What a pity that their religion forbade them to make images!
M. C. Escher

Goals of the Chapter

- Develop tools for transforming one picture into another.
- Introduce the fundamental concepts of affine transformations, which perform combinations of rotations, scalings, and translations.
- Develop functions that apply affine transformations to objects in computer programs
- Develop tools for transforming coordinate frames.
- See how to set up a camera to render a 3D scene using OpenGL
- Learn to design scenes in the Scene Design language SDL, and to write programs that read SDL files and draw the scenes they describe.

Preview.

Section 5.1 motivates the use of 2D and 3D transformations in computer graphics, and sets up some basic definitions. Section 5.2 defines 2D affine transformations and establishes terminology for them in terms of a matrix. The notation of coordinate frames is used to keep clear what objects are being altered and how. The section shows how elementary affine transformations can perform scaling, rotation, translation, and shearing. Section 5.2.5 shows that you can combine as many affine transformations as you wish, and the result is another affine transformation, also characterized by a matrix. Section 5.2.7 discusses key properties of all affine transformations — most notably that they preserve straight lines, planes, and parallelism — and shows why they are so prevalent in computer graphics.

Section 5.3 extends these ideas to 3D affine transformations, and shows that all of the basic properties hold here as well. 3D transformations are more complex than 2D ones, however, and more difficult to visualize, particularly when it comes to 3D rotations. So special attention is paid to describing and combining various rotations.

Section 5.4 discusses the relationship between transforming points and transforming coordinate systems. Section 5.5 shows how transformations are managed within a program when OpenGL is available, and how transformations can greatly simplify many operations commonly needed in a graphics program. Modeling transformations and the use of the "current transformation" are motivated through a number of examples. Section 5.6 discusses modeling 3D scenes and drawing them using OpenGL. A "camera" is defined that is positioned and oriented so that it takes the desired snapshot of the scene. The section discusses how transformations are used to size and position objects as desired in a scene. Some example 3D scenes are modeled and rendered, and the code required to do it is examined. This section also introduces a Scene Description language, SDL, and shows how to write an application that can draw any scene described in the language. This

requires the development of a number of classes to support reading and parsing SDL files, and creating lists of objects that can be rendered. These classes are available from the book's web site.

The chapter ends with a number of Case Studies that elaborate on the main ideas and provide opportunities to work with affine transformations in graphics programs. One case study asks you to develop routines that perform transformations when OpenGL is not available. Also described there are ways to decompose an affine transformation into its elementary operations, and the development of a fast routine to draw arcs of circles that capitalizes on the equivalence between a rotation and three successive shears.

5.1. Introduction.

The main goal in this chapter is to develop techniques for working with a particularly powerful family of transformations called *affine transformations*, both with pencil and paper and in a computer program, with and without OpenGL. These transformations are a fundamental cornerstone of computer graphics, and are central to OpenGL as well as most other graphics systems. They are also a source of difficulty for many programmers because it is often difficult to get them right.

One particularly delicate area is the confusion of points and vectors. Points and vectors seem very similar, and are often expressed in a program using the same data type, perhaps a list of three numbers like (3.0, 2.5, -1.145) to express them in the current coordinate system. But this practice can lead to disaster in the form of serious bugs that are very difficult to ferret out, principally because points and vectors do *not* transform the same way. We need a way to keep them straight, which is offered by using *coordinate frames* and appropriate homogeneous coordinates as introduced in Chapter 4.

5.2. Introduction to Transformations.

The universe is full of magical things, patiently waiting for our wits to grow sharper.
E. Phillpotts

We have already seen some examples of transformations, at least in 2D. In Chapter 3, for instance, the window to viewport transformation was used to scale and translate objects situated in the world window to their final size and position in the viewport.

We want to build on those ideas, and gain more flexible control over the size, orientation, and position of objects of interest. In the following sections we develop the required tools, based on the powerful **affine transformation**, which is a staple in computer graphics. We operate in both two and three dimensions.

Figure 5.1a shows two versions of a simple house, drawn before and after each of its points has been transformed. In this case the house has been scaled down in size, rotated a small amount, and then moved up and to the right. The overall transformation here is a combination of three more elementary ones: scaling, rotation, and translation. Figure 5.1b shows a 3D house before and after it is similarly transformed: each 3D point in the house is subjected by the transformation to a scaling, a rotation, and a translation.

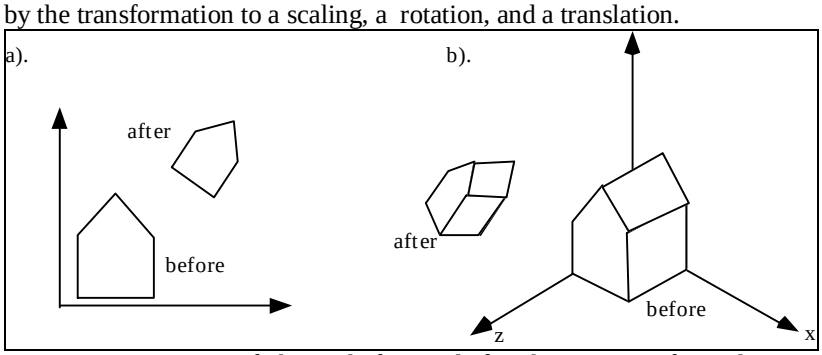

Figure 5.1. Drawings of objects before and after they are transformed.

Transformations are very useful in a number of situations:

a). We can compose a "scene" out of a number of objects, as in Figure 5.2. Each object such as the arch is most easily designed (once for all) in its own "master" coordinate system. The scene is then fashioned by placing a number of "instances" of the arch at different places and with different sizes, using the proper transformation for each.

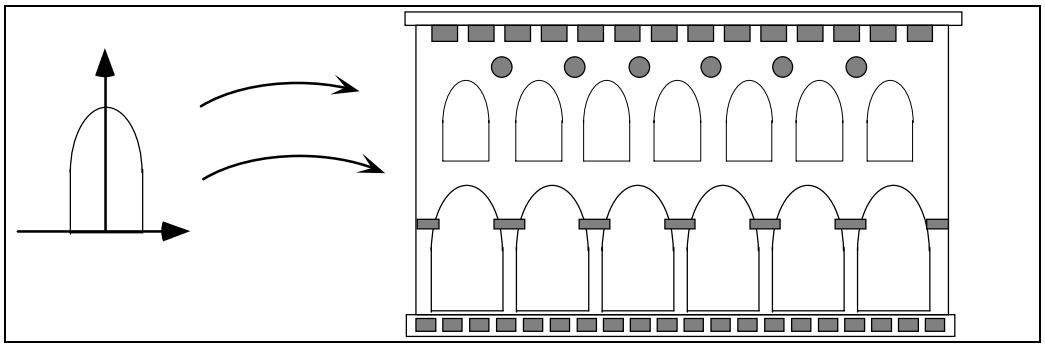

Figure 5.2. Composing a picture from many instances of a simple form.

Figure 5.3 shows a 3D example, where the scene is composed of many instances of cubes that have been scaled and positioned in a "city".

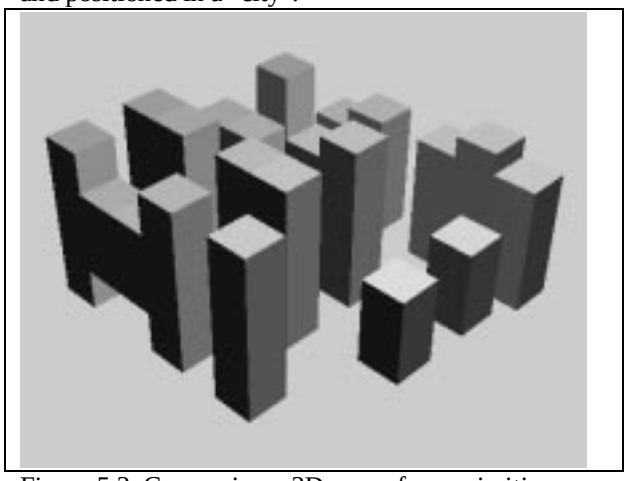

Figure 5.3. Composing a 3D scene from primitives.

b). Some objects, such as the snowflake shown in Figure 5.4, exhibit certain symmetries. We can design a single "motif" and then fashion the whole shape by appropriate reflections, rotations, and translations of the motif.

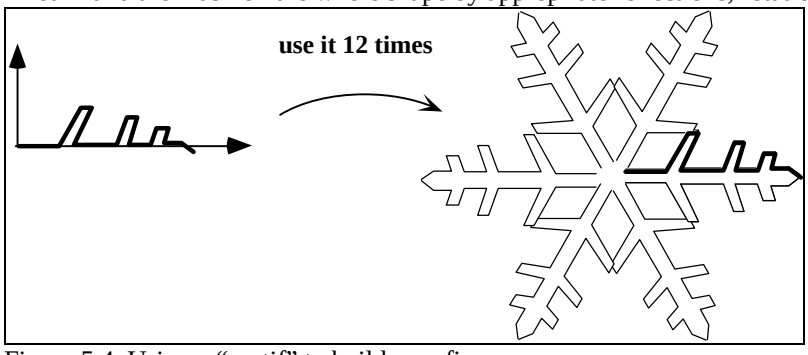

Figure 5.4. Using a "motif" to build up a figure.

c). A designer may want to view an object from different vantage points, and make a picture from each one. The scene can be rotated and viewed with the same camera. But as suggested in Figure 5.5 it's more natural to leave

the scene alone and move the camera to different orientations and positions for each snapshot. Positioning and reorienting a camera can be carried out through the use of 3D affine transformations.

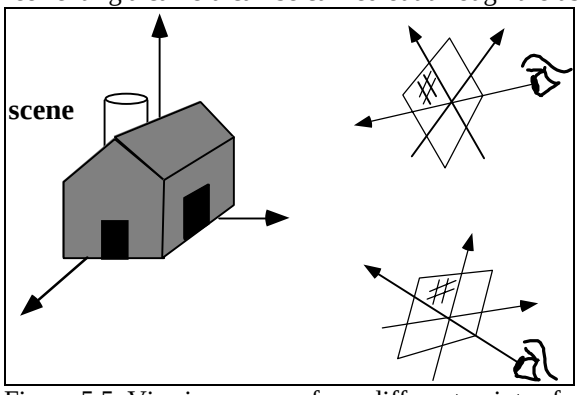

Figure 5.5. Viewing a scene from different points of view.

d). In a computer animation several objects must move relative to one another from frame to frame This can be achieved by shifting and rotating their local coordinate systems as the animation proceeds. Figure 5.6 shows an example.

author supplied

Figure 5.6. Animating by transforming shapes.

Where are we headed? Using Transformations with OpenGL.

The first few sections of this chapter present the basic concepts of affine transformations, and show how they produce certain geometric effects such as scaling, rotations, and translations, both in 2D and 3D space. Ultimately, of course, the goal is to produce graphical drawings of objects that have been transformed to the proper size, orientation, and position so they produce the desired scene. A number of graphics platforms, including OpenGL, provide a "graphics pipeline", or sequence of operations that are applied to all points that are "sent through it". A drawing is produced by processing each point.

Figure 5.7 shows a simplified view of the OpenGL graphics pipeline. An application "sends it" a sequence of points P_1, P_2, P_3, ... using the commands like the now familiar:

Figure 5.7. The OpenGL pipeline.

```
glBegin(GL_LINES);
   glVertex3f(...); // send P1 through the pipeline
   glVertex3f(...); // send P2 through the pipeline
   glVertex3f(...); // send P3 through the pipeline
   ...
glEnd();
```

As shown in the figure these points first encounter a transformation called the "current transformation" ("CT"), which alters their values into a different set of points, say Q_1, Q_2, Q_3, ... Just as the original points P_i describe some geometric object, the points Q_i describe the transformed version of the same object. These points are then sent through additional steps, and ultimately are used to draw the final image on the display.

The current transformation therefore provides a crucial tool in the manipulation of graphical objects, and it is essential for the application programmer to know how to adjust the CT so that the desired transformations are produced. After developing the underlying theory of affine transformations, we turn in Section 5.5 to showing how this is done.

Object transformations versus coordinate transformations.

There are two ways to view a transformation: as an **object transformation** or as a **coordinate transformation**. An object transformation alters the coordinates of each point on the object according to some rule, leaving the underlying coordinate system fixed. A coordinate transformation defines a new coordinate system in terms of the old one, then represents all of the object's points in this new system. The two views are closely connected, and each has its advantages, but they are implemented somewhat differently. We shall first develop the central ideas in terms of object transformations, and then relate them to coordinate transformations.

5.2.1. Transforming Points and Objects.

We look here at the general idea of a transformation, and then specialize to affine transformations.

A transformation alters each point, *P*, in space (2D or 3D) into a new point, *Q*, using a specific formula or algorithm. Figure 5.8 shows 2D and 3D examples.

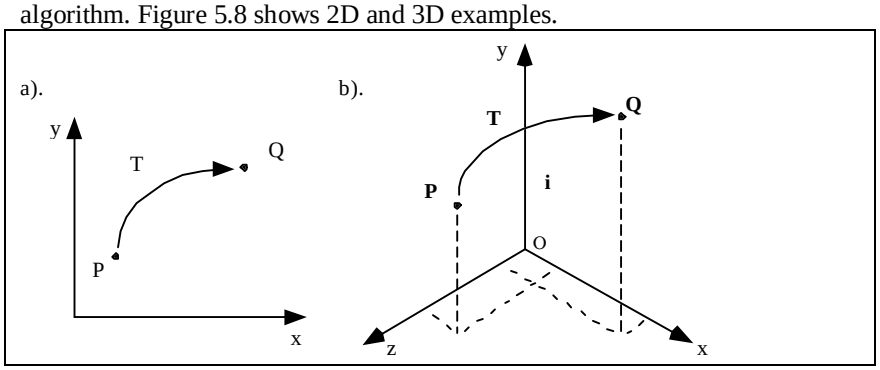

Figure 5.8. Mapping points into new points.

As Figure 5.8 illustrates, an arbitrary point *P* in the plane is **mapped** to *Q*. We say *Q* is the **image** of *P* under the mapping *T*. Part a) shows a 2D point *P* being mapped to a new point *Q*; part b) shows a 3D point *P* being mapped to a new *Q*. We transform an object by transforming each of its points, using, of course, the same function *T*() for each point. We can map whole collections of points at once. The collection might be all the points on a line or circle. The **image** of line *L* under *T* , for instance, consists of the images of all the individual points of L.[1]

Most mappings of interest are continuous, so the image of a straight line is still a connected curve of some shape, although it's not necessarily a straight line. Affine transformations, however, do preserve lines as we shall see: The image under *T* of a straight line is also a straight line. Most of this chapter will focus on affine transformations, but other kinds can be used to create special effects. Figure 5.9, for instance, shows a complex warping of a figure that cannot be achieved with an affine transformation. This transformation might be used for visual effect, or to emphasize important features of an object.

old Fig 11.2 Da Vinci warped peculiarly

Figure 5.9. A complex warping of a figure.

To keep things straight we use an explicit coordinate frame when performing transformations. Recall from Chapter 4 that a coordinate frame consists of a particular point, ϑ, called the origin, and some mutually perpendicular vectors (called **i** and **j** in the 2D case; **i, j,** and **k** in the 3D case) that serve as the axes of the coordinate frame.

[1]More formally, if *S* is a set of points, its **image** *T*(*S*) is the set of all points *T*(*P*) where *P* is some point in *S*.

Take the 2D case first, as it is easier to visualize. In whichever coordinate frame we are using, point *P* and *Q* have the representations $\tilde{P}$ and $\tilde{Q}$ given by:

$$\tilde{P} = \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}, \tilde{Q} = \begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix}$$

Recall that this means the point *P* "is at" location $P = P_x \mathbf{i} + P_y \mathbf{j} + \vartheta$, and similarly for *Q*. P_x and P_y are familiarly called the "coordinates" of P. The transformation operates on the representation $\tilde{P}$ and produces the representation $\tilde{Q}$ according to some function, *T*(),

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = T \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \tag{5.1}$$

or more succinctly,

$$\tilde{Q} = T(\tilde{P}). \tag{5.2}$$

The function *T*() could be complicated, as in

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(P_x)e^{-P_y} \\ \dfrac{\ln(P_y)}{1+P_x^{\,2}} \\ 1 \end{pmatrix}$$

and such transformations might have interesting geometric effects, but we restrict ourselves to much simpler families of functions, those that are *linear* in P_x and P_y This property characterizes the affine transformations.

5.2.2. The Affine Transformations.

What is algebra, exactly? Is it those three-cornered things?
J. M. Barrie

Affine transformations are the most common transformations used in computer graphics. Among other things they make it easy to scale, rotate, and reposition figures. A succession of affine transformations can easily be combined into a simple overall affine transformation, and affine transformations permit a compact matrix representation.

Affine transformations have a simple form: the coordinates of *Q* are linear combinations of those of *P*:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{pmatrix} \tag{5.3}$$

for some six given constants m_{11}, m_{12}, etc. Q_x consists of portions of both of P_x and P_y, and so does Q_y. This "cross fertilization" between the *x*- and *y*-components gives rise to rotations and shears.

The affine transformation of Equation 5.3 has a useful matrix representation that helps to organize your thinking:[2]

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \qquad (5.4)$$

(Just multiply this out to see that it's the same as Equation 5.3. In particular, note how the third row of the matrix forces the third component of *Q* to be 1.) For an affine transformation the third row of the matrix is always (0, 0, 1).

Vectors can be transformed as well as points. Recall that if vector *V* has coordinates *V*x and *V*y then its coordinate frame representation is a column vector with a third component of 0. When transformed by the same affine transformation as above the result is

$$\begin{pmatrix} W_x \\ W_y \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 0 \end{pmatrix} \qquad (5.5)$$

which is clearly another vector: its third component is always 0.

Practice Exercise 5.2.1. Apply the transformation. An affine transformation is specified by the matrix:

$$\begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Find the image *Q* of point *P* = (1, 2).

Solution: $\begin{pmatrix} 8 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$

5.2.3. Geometric Effects of Elementary 2D Affine Transformations.

What geometric effects are produced by affine transformations? They produce combinations of four elementary transformations: (1) a translation, (2) a scaling, (3) a rotation, and (4) a shear.

Figure 5.10 shows an example of the effect of each kind of transformation applied individually.

1st Ed. Figure 11.5

Figure 5.10. Transformations of a map: a) translation, b) scaling, c)rotation, d) shear.

Translation.

You often want to translate a picture into a different position on a graphics display. The translation part of the affine transformation arises from the third column of the matrix

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & m_{13} \\ 0 & 1 & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \qquad (5.6)$$

[2]See Appendix 2 for a review of matrices.

or simply

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} P_x + m_{13} \\ P_y + m_{23} \\ 1 \end{pmatrix}$$

so in ordinary coordinates $Q = P + \mathbf{d}$, where "offset vector" **d** has components (m_{13}, m_{23}).

For example, if the offset vector is (2, 3), every point will be altered into a new point that is two units farther to the right and three units above the original point. The point (1, -5), for instance, is transformed into (3, - 2), and the point (0, 0) is transformed into (2, 3).

Scaling.

A scaling changes the size of a picture and involves two scale factors, S_X and S_Y, for the *x*- and *y*-coordinates, respectively:

$(Q_X, Q_Y) = (S_X P_X, S_Y P_Y)$

Thus the matrix for a scaling by itself is simply

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (5.7)$$

Scaling in this fashion is more accurately called **scaling about the origin**, because each point *P* is moved S_X times farther from the origin in the *x*-direction, and S_Y times farther from the origin in the *y*-direction. If a scale factor is negative, then there is also a **reflection** about a coordinate axis. Figure 5.11 shows an example in which the scaling $(S_X, S_Y) = (-1, 2)$ is applied to a collection of points. Each point is both reflected about the *y*-axis and scaled by 2 in the *y*-direction.

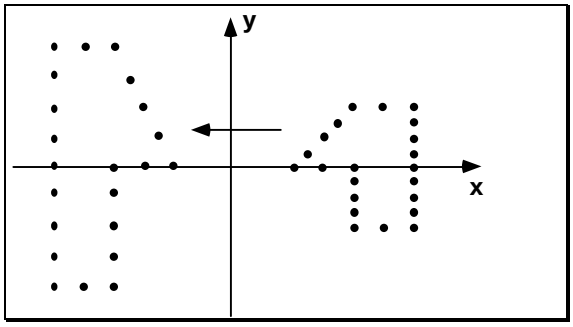

Figure 5.11. A scaling and a reflection.

There are also "pure" reflections, for which each of the scale factors is +1 or -1. An example is

$T(P_X, P_Y) = (-P_X, P_Y)$ (5.8)

which produces a mirror image of a picture by "flipping" it horizontally about the *y*-axis, replacing each occurrence of *x* with *-x*. (What is the matrix of this transformation?)

If the two scale factors are the same, $S_X = S_Y = S$, the transformation is a **uniform scaling**, or a magnification about the origin, with magnification factor $|S|$. If *S* is negative, there are reflections about both axes. A point is

moved outward from the origin to a position $|S|$ times farther away from the origin. If $|S| < 1$, the points will be moved closer to the origin, producing a reduction (or "demagnification"). If, on the other hand, the scale factors are not the same, the scaling is called a **differential scaling**.

Practice Exercise 5.2.2. Sketch the effect. A pure scaling affine transformation uses scale factors $S_x = 3$ and $S_y = -2$. Find the image of each of the three objects in Figure 5.12 under this transformation, and sketch them. (Make use of the facts - to be proved later - that an affine transformations maps straight lines to straight lines, and ellipses to ellipses.)

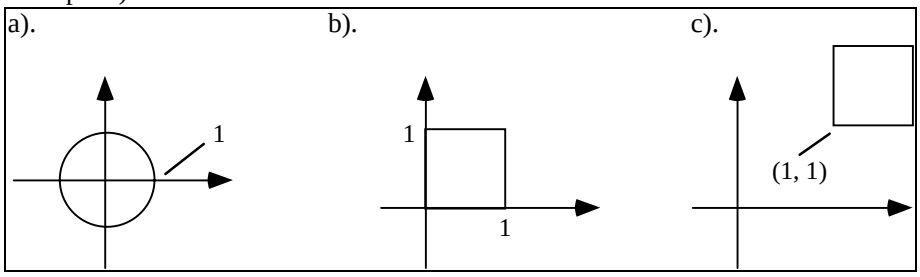

Figure 5.12. Objects to be scaled.

Rotation.

A fundamental graphics operation is the rotation of a figure about a given point through some angle. Figure 5.13 shows a set of points rotated about the origin through an angle of $\theta = 60^{o}$.

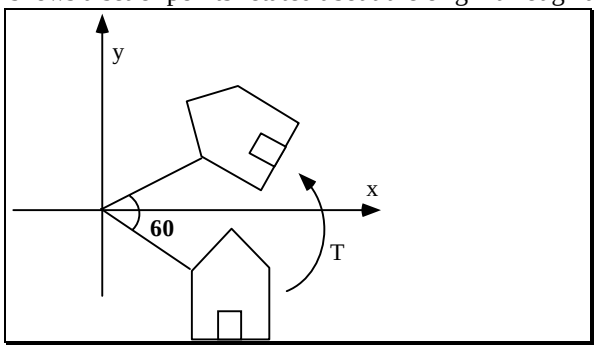

Figure 5.13. Rotation of points through an angle of 60°.

When $T(\ )$ is a rotation about the origin, the offset vector **d** is zero and $Q = T(P)$ has the form

$$Q_x = P_x \cos(\theta) - P_y \sin(\theta)$$
$$Q_y = P_x \sin(\theta) + P_y \cos(\theta) \qquad (5.9)$$

As we derive next, this form causes positive values of θ to perform a counterclockwise (CCW) rotation. In terms of its matrix form, a pure rotation about the origin is given by

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (5.10)$$

Example 5.2.1. Find the transformed point, Q, caused by rotating $P = (3, 5)$ about the origin through an angle of 60^{0}. **Solution**: For an angle of 60^{0}, $\cos(\theta) = .5$ and $\sin(\theta) = .866$, and Equation 5.9 yields $Q_X = (3)(0.5) - (5)(0.866) = -2.83$ and $Q_Y = (3)(0.866) + (5)(0.5) = 5.098$. Check this on graph paper by swinging an arc of 60^{0} from (3, 5) and reading off the position of the mapped point. Also check numerically that Q and P are at the same distance from the origin. (What is this distance?)

Derivation of the Rotation Mapping.
We wish to demonstrate that Equation 5.9 is correct. Figure 5.14 shows how to find the coordinates of a point *Q* that results from rotating point *P* about the origin through an angle θ . If *P* is at a distance *R* from the origin, at some angle ϕ, then $P = (R\cos(\phi), R\sin(\phi))$. Now *Q* must be at the same distance as *P* , and at angle θ + ϕ. Using trigonometry, the coordinates of *Q* are

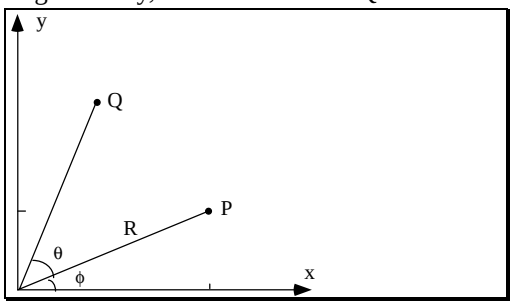

Figure 5.14. Derivation of the rotation mapping.

$$Q_x = R\cos(\theta + \phi)$$
$$Q_y = R\sin(\theta + \phi)$$

Substitute into this equation the two familiar trigonometric relations:

$\cos(\theta + \phi) = \cos(\theta)\cos(\phi) - \sin(\theta)\sin(\phi)$
$\sin(\theta + \phi) = \sin(\theta)\cos(\phi) + \cos(\theta)\sin(\phi)$

and use $P_X = R\cos(\phi)$ and $P_Y = R\sin(\phi)$ to obtain Equation 5.9.

Practice Exercise 5.2.3. Rotate a Point. Use Equation 5.9 to find the image of each of the following points after rotation about the origin:
a). (2, 3) through an angle of -45^o
b). (1, 1) through an angle of -180^o.
c). (60, 61) through an angle of 4^o.
In each case check the result on graph paper, and compare numerically the distances of the original point and its image from the origin.
Solution: a). (3.5355, .7071), b). (-1, -1), c). (55.5987, 65.0368).

Shearing.

An example of shearing is illustrated in Figure 5.15 is a shear "in the *x*-direction" (or "along *x*"). In this case the *y*-coordinate of each point is unaffected, whereas each *x*-coordinate is translated by an amount that increases linearly with *y*. A shear in the *x*-direction is given by

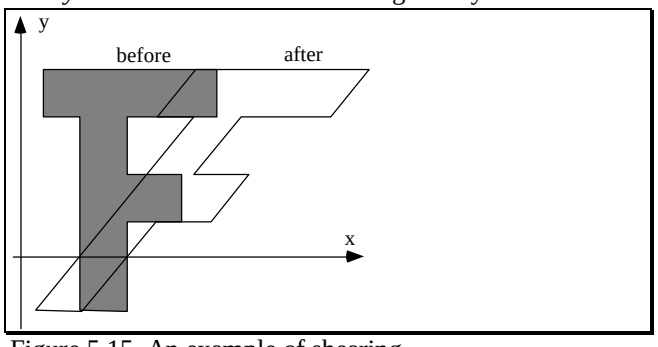

Figure 5.15. An example of shearing.

$$Q_x = P_x + h\,P_y$$

$Q_y = P_y$

where the coefficient h specifies what fraction of the y-coordinate of P is to be added to the x-coordinate. The quantity h can be positive or negative. Shearing is sometimes used to make italic letters out of regular letters. The matrix associated with this shear is:

$$\begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (5.11)$$

One can also have a shear "along y", for which $Q_x = P_x$ and $Q_y = g\,P_x + P_y$ for some value g, so that the matrix is given by

$$\begin{pmatrix} 1 & 0 & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (5.12)$$

Example 5.2.2: Into which point does (3, 4) shear when h = .3 in Equation 5.11? **Solution**: Q = (3 + (.3)4 , 4) = (4.2, 4).
Example 5.2.3: Let g = 0.2 in Equation 5.12. To what point does (6, - 2) map? **Solution**: Q = (6, 0.2 · 6 - 2) = (6, - 0.8).

A more general shear "along" an arbitrary line is discussed in a Case Study at the end of the chapter. A notable feature of a shear is that its matrix has a determinant of 1. As we see later this implies that the area of a figure is unchanged when it is sheared.

Practice Exercise 5.2.4. Shearing Lines. Consider the shear for which g = .4 and h = 0. Experiment with various sets of three collinear points to build some assurance that the sheared points are still collinear. Then, assuming that lines do shear into lines, determine into what objects the following line segments shear:
a. the horizontal segment between (- 3, 4) and (2, 4);
b. the horizontal segment between (- 3, - 4) and (2, - 4);
c. the vertical segment between (- 2, 5) and (- 2, - 1);
d. the vertical segment between (2, 5) and (2, - 1);
e. the segment between (- 1, - 2) and (3, 2);
Into what shapes do each of the objects in Figure 5.2.12 shear?

5.2.4. The Inverse of an Affine Transformation

Most affine transformations of interest are **nonsingular**, which means that the determinant of M in Equation 5.4, which evaluates to[3]

$$\det M = m_{11}m_{22} - m_{12}m_{21} \qquad (5.13)$$

is nonzero. Notice that the third column of M, which represents the amount of translation, does not affect the determinant. This is a direct consequence of the two zeroes appearing in the third row of M. We shall make special note on those rare occasions that we use singular transformations.

[3] See Appendix 2 for a review of determinants.

It is reassuring to be able to undo the effect of a transformation. This is particularly easy to do with nonsingular affine transformations. If point *P* is mapped into point *Q* according to *Q* = *MP*, simply premultiply both sides by the **inverse** of *M*, denoted M^{-1}, and write

$$P = M^{-1} Q \tag{5.14}$$

The inverse of *M* is given by[4]

$$M^{-1} = \frac{1}{det\ M} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \tag{5.15}$$

We therefore obtain the following matrices for the elementary inverse transformations:

• *Scaling* (use *M* as found in Equation 5.7):

$$M^{-1} = \begin{pmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

• *Rotation* (use *M* as found in Equation 5.10):

$$M^{-1} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

• *Shearing* (using the version of *M* in Equation 5.11):

$$M^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -h & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

• *Translations*: The inverse transformation simply subtracts the offset rather than adds it.

$$M^{-1} = \begin{pmatrix} 1 & 0 & -m_{13} \\ 0 & 1 & -m_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

Practice Exercises.

5.2.5. What Is the Inverse of a Rotation? Show that the inverse of a rotation through θ is a rotation through -θ. Is this reasonable geometrically? Why?

5.2.6. Inverting a Shear. Is the inverse of a shear also a shear? Show why or why not.

5.2.7. An Inverse Matrix. Compute the inverse of the matrix

$$M = \begin{pmatrix} 3 & 2 & 1 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

[4] See Appendix 2 for a review of inverse matrices.

5.2.5. Composing Affine Transformations.

Progress might have been all right once , but it has gone on too long.
Ogden Nash

It's rare that we want to perform just one elementary transformation; usually an application requires that we build a compound transformation out of several elementary ones. For example, we may want to

- translate by (3, - 4)
- then rotate through 30^0
- then scale by (2, - 1)
- then translate by (0, 1.5)
- and finally rotate through -30^0.

How do these individual transformations combine into one overall transformation? The process of applying several transformations in succession to form one overall transformation is called **composing** (or **concatenating**) the transformations. As we shall see, when two affine transformations are composed, the resulting transformation is (happily) also affine.

Consider what happens when two 2D transformations, $T_1(\ )$ and $T_2(\ )$, are composed. As suggested in Figure 5.16, $T_1(\ )$ maps P into Q, and $T_2(\ )$ maps Q into point W. What is the transformation, $T(\ )$, that maps P directly into W? That is, what is the nature of $W = T_2(Q) = T_2(T_1(P))$?

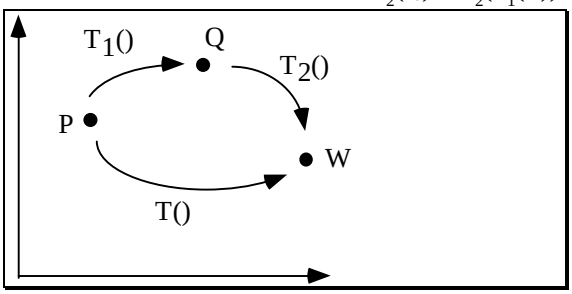

Figure 5.16. The composition of two transformations.

Suppose the two transformations are represented by the matrices $\tilde{M}_1$ and $\tilde{M}_2$. Thus $\tilde{P}$ is first transformed to the point $\tilde{M}_1\,\tilde{P}$ which is then transformed to $\tilde{M}_2\ (\tilde{M}_1\,\tilde{P})$. By associativity this is just $(\tilde{M}_2\,\tilde{M}_1)\,\tilde{P}$, and so we have

$$\tilde{W} = \tilde{M}\tilde{P} \qquad (5.16)$$

where the overall transformation is represented by the single matrix

$$\tilde{M} = \tilde{M}_2\,\tilde{M}_1. \qquad (5.17)$$

When homogeneous coordinates are used, composing affine transformations is accomplished by simple matrix multiplication. Notice that the matrices appear in *reverse* order to that in which the transformations are applied: if we first apply T_1 with matrix $\tilde{M}_1$, and then apply T_2 with matrix $\tilde{M}_2$ to the result, the overall transformation has matrix $\tilde{M}_2\tilde{M}_1$, with the "second" matrix appearing first in the product as you read from left to right. (Just the opposite order will be seen when we transform coordinate systems.)

By applying the same reasoning, any number of affine transformations can be composed simply by multiplying their associated matrices. In this way, transformations based on an arbitrary succession of rotations, scalings, shears, and translations can be formed and captured in a single matrix.

Example 5.2.4. Build one. Build a transformation that
a). rotates through 45 degrees;
b). then scales in x by 1.5 and in y by - 2;
c). then translates through (3, 5).
Find the image under this transformation of the point (1,2).

Solution: Construct the three matrices and multiply them in the proper order (first one last, etc.) to form:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1.5 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} .707 & -.707 & 0 \\ .707 & .707 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1.06 & -1.06 & 3 \\ -1.414 & -1.414 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

Now to transform point (1, 2), enlarge it to the triple (1, 2, 1), multiply it by the composite matrix to obtain (1.94, 0.758, 1), and drop the one to form the image point (1.94, 0.758). It is instructive to use graph paper, and to perform each of this transformations in turn to see how (1, 2) is mapped.

5.2.6. Examples of Composing 2D Transformations.

Art is the imposing of a pattern on experience,
and our aesthetic enjoyment is recognition of the pattern.
Alfred North Whitehead

We examine some important examples of composing 2D transformations, and see how they behave.

Example 5.2.5. Rotating About an Arbitrary Point.
So far all rotations have been about the origin. But suppose we wish instead to rotate points about some other point in the plane. As suggested in Figure 5.17, the desired "pivot" point is $V = (V_X, V_Y)$, and we wish to rotate points such as P through angle θ to position Q. To do this we must relate the rotation about V to an elementary rotation about the origin.

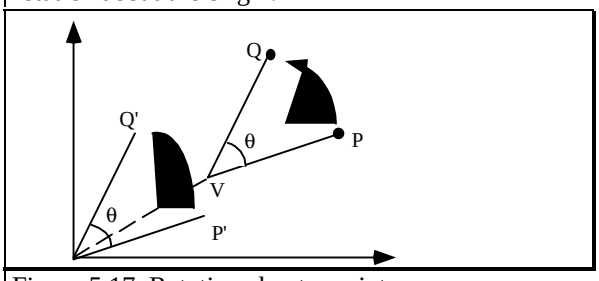

Figure 5.17. Rotation about a point.

Figure 5.2.17 shows that if we first translate all points so that V coincides with the origin, then a rotation about the origin (which maps P' to Q') will be appropriate. Once done, the whole plane is shifted back to restore V to its original location. The rotation therefore consists of the following three elementary transformations:

1. Translate point P through vector $\mathbf{v} = (-V_X, -V_Y)$:
2. Rotate about the origin through angle θ;
3. Translate P back through $\mathbf{v}$.

Creating a matrix for each elementary transformation, and multiplying them out produces:

$$\begin{pmatrix} 1 & 0 & V_x \\ 0 & 1 & V_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & -V_x \\ 0 & 1 & -V_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & d_x \\ \sin(\theta) & \cos(\theta) & d_y \\ 0 & 0 & 1 \end{pmatrix}$$

where the overall translation components are

$$d_x = -\cos(\theta)V_x + \sin(\theta)V_y + V_x$$
$$d_y = -\sin(\theta)V_x - \cos(\theta)V_y + V_y$$

Because the same *cos*(θ) and *sin*(θ) terms appear in this result as in a rotation about the origin, we see that a rotation about an arbitrary point is equivalent to a rotation about the origin followed by a complicated translation through (d_x, d_y).

As a specific example, we find the transformation that rotates points through 30° about (-2, 3), and determine to which point the point (1, 2) maps. A 30° rotation uses *cos*(θ) = 0.866 and *sin*(θ) = 0.5. The offset vector is then (1.232, 1.402), and so the transformation applied to any point (P_X, P_Y) is

$Q_X = 0.866\ P_X - 0.5\ P_Y + 1.232$
$Q_Y = 0.5\ P_X + 0.866\ P_Y + 1.402$

Applying this to (1, 2) yields (1.098, 3.634). This is the correct result, as can be checked by sketching it on graph paper. (Do it!)

Example 5.2.6. Scaling and Shearing about arbitrary "pivot" points.
In a similar manner we often want to scale all points about some pivot point other than the origin. Because the elementary scaling operation of Equation 5.13 scales points about the origin, we do the same "shift-transform-unshift" sequence as for rotations. This and generalizing the shearing operation are explored in the exercises.

Example 5.2.7. Reflections about a tilted line.
Consider the line through the origin that makes an angle of β with the *x*-axis, as shown in Figure 5.18. Point *A* reflects into point *B*, and each house shown reflects into the other. We want to develop the transformation that reflects any point *P* about this axis, to produce point *Q*. Is this an affine transformation?

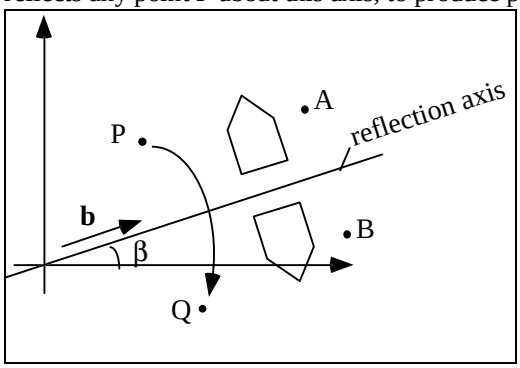

Figure 5.18. Reflecting about a tilted axis.

To show that it is affine, we build it out of three parts:
- A rotation through angle -β (so the axis coincides with the *x*-axis);
- A reflection about the *x*-axis;
- A rotation back through β that "restores" the axis.

Each of these is represented by a matrix, so the overall transformation is given by the product of the three matrices, so it *is* affine. Check that each of the steps is properly represented in the following three matrices, and that the product is also correct:

$$\begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} c^2 - s^2 & -2cs & 0 \\ -2cs & s^2 - c^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where *c* stands for *cos*(β) and *s* for *sin*(β). Using trigonometric identities, the final matrix can be written (check this out!)

$$\begin{pmatrix} \cos(2\beta) & \sin(2\beta) & 0 \\ \sin(2\beta) & -\cos(2\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$ {a reflection about the axis at angle β} (5.18)

This has the general look of a rotation matrix, except the angle has been doubled and minus signs have crept into the second column. But in fact it is the matrix for a reflection about the axis at angle β.

Practice Exercises.

Exercise 5.2.8. The classic: the Window to Viewport Transformation.
We developed this transformation in Chapter 3. Rewriting Equation 3.2 in the current notation we have:

$$\tilde{M} = \begin{pmatrix} A & 0 & C \\ 0 & B & D \\ 0 & 0 & 1 \end{pmatrix}$$

where the ingredients *A*, *B*, *C*, and *D* depend on the window and viewport and are given in Equation 3.3. Show that this transformation is composed of:

- A translation through (-*W.l*, -*W.b*) to place the lower left corner of the window at the origin;
- A scaling by (*A*, *B*) to size things.
- A translation through (*V.l*, *V.b*) to move the corner of the viewport to the desired position.

5.2.9. Alternative Form for a Rotation About a Point. Show that the transformation of Figure 5.17 can be written out as

$Q_X = cos(\theta)(P_X - V_X) - sin(\theta)(P_Y - V_Y) + V_X$

$Q_Y = \sin(\theta)(P_X - V_X) + cos(\theta)(P_Y - V_Y) + V_Y$

This form clearly reveals that the point is first translated by ($-V_X$, $-V_Y$), rotated, and then translated by (V_X, V_Y).

5.2.10. Where does it end up? Where is the point (8, 9) after it is rotated through 50^0 about the point (3, 1)? Find the *M* matrix.

5.2.11. Seeing it two ways. On graph paper place point *P* = (4, 7) and the result *Q* of rotating *P* about *V* = (5, 4) through 45^0. Now rotate *P* about the origin through 45^0 to produce *Q*', which is clearly different from *Q*. The difference between them is *V* - *VM*. Show the point *V* - *VM* in the graph, and check that *Q* - *Q*' equals *V* - *VM*.

5.2.12. What if the axis doesn't go through the origin? Find the affine transformation that produces a reflection about the line given parametrically by $L(t) = A + \mathbf{b}t$. Show that it reduces to the result in Equation 5.20 when $A + \mathbf{b}t$ does pass through the origin.

5.2.13. Reflection in *x* = *y*. Show that a reflection about the line *x* = *y* is equivalent to a reflection in x followed by a 90^0 rotation.

5.2.14. Scaling About an Arbitrary Point. Fashion the affine transformation that scales points about a pivot point, (V_X, V_Y). Test the overall transformation on some sample points, to confirm that the scaling operation is correct. Compare this with the transformation for rotation about a pivot point.

5.2.15. Shearing Along a Tilted Axis. Fashion the transformation that shears a point along the axis described by vector **u** tilted at angle θ, as shown in Figure 5.19. Point *P* is shifted along **u** an amount that is fraction *f* of the displacement *d* of *P* from the axis.

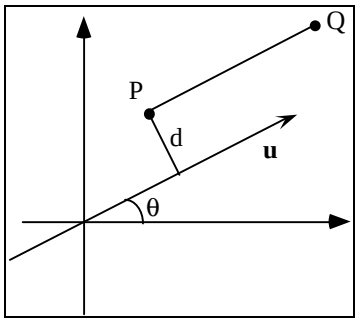

Figure 5.19. Shearing along a tilted axis.

5.2.16. Transforming Three Points. An affine transformation is completely determined by specifying what it does to three points. To illustrate this, find the affine transformation that converts triangle *C* with vertices (- 3, 3), (0, 3), and (0, 5) into equilateral triangle *D* with vertices (0, 0), (2, 0), and (1, $\sqrt{3}$), as shown in Figure 5.20.

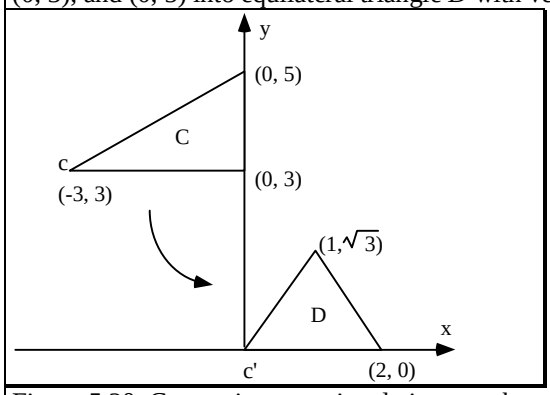

Figure 5.20. Converting one triangle into another.

Do this by a sequence of three elementary transformations:

1. Translate *C* down by 3 and right by 3 to place vertex *c* at *c*'.
2. Scale in *x* by 2/3 and in *y* by $\sqrt{3}$ /2 so *C* matches *D* in width and height.
3. Shear by $1/\sqrt{3}$ in the *x*-direction to align the top vertex of *C* with that of *D*.

Check that this transformation does in fact transform triangle *C* into triangle *D*. Also find the inverse of this transformation and show that it converts triangle *D* back into triangle *C*.

5.2.17. Fixed Points of an Affine Transformation. The point *F* is a *fixed point* of the affine transformation $T(p) = Mp$ if $T(F) = F$. That is, if *F* satisfies $F\,M = F$.

a). Show that when the third column of *M* is (0,0,1) - such that there is no translation, the origin is always a fixed point of *T* .

b). Show that if *F* is a fixed point of *T* then for *any P* we have $T(P) = M(P - F) + F$.

c). Show that *F* must always satisfy: $F = \mathbf{d}(I - M)^{-1}$. Does every affine transformation have a fixed point?

d). What is a fixed point for a rotation about point *V*? Show that it satisfies the relationship in part b above.

e). What is the fixed point for a scaling with scale factors S_x and S_y, about point *V*?

f). Consider the "5-th iterate" of *T*() applied to *P*, given by $R = T(T(T(T(T(P)))))$. (Recall IFS's described at the end of Chapter 1). Use the result in b) to show a simple form for output *R* in terms of fixed point *F* of *T*(): $R = (P - F)M^5 + F$.

5.2.18. Finding Matrices. Give the explicit form of the 3-by-3 matrix representing each of the following transformations:

a. Scaling by a factor of 2 in the *x*-direction and then rotating about (2, 1).

b. Scaling about (2, 3) and following by translation through (1, 1).

c. Shearing in *x* by 30%, scaling by 2 in *x*, and then rotating about (1, 1) through 30°.

5.2.19. Normalizing a Box. Find the affine transformation that maps the box with corners (0, 0), (2, 1), (0, 5), and (- 2, 4) into the square with corners (0, 0), (1, 0), (1, 1), and (0, 1). Sketch the boxes.

5.2.20. Some Transformations Commute. Show that uniform scaling **commutes** with rotation, in that the resulting transformation does not depend on the order in which the individual transformations are applied. Show that two translations commute, as do two scalings. Show that differential scaling does not commute with rotation.
5.2.21. Reflection plus a rotation. Show that a reflection in *x* followed by a reflection in *y* is the same as a rotation by 180^0.
5.2.22. Two Successive Rotations. Suppose that $R(\theta)$ denotes the transformation that produces a rotation through angle θ. Show that applying $R(\theta_1)$ followed by $R(\theta_2)$ is equivalent to applying the single rotation $R(\theta_1 + \theta_2)$. Thus successive rotations are additive.
5.2.23. A Succession of Shears. Find the composition of a pure shear along the *x*-axis followed by a pure shear along the *y*-axis. Is this still a shear? Sketch by hand an example of what happens to a square centered at the origin when subjected to a simultaneous shear versus a succession of shears along the two axes.

5.2.8. Some Useful Properties of Affine Transformations.

We have seen how to represent 2D affine transformations with matrices, how to compose complex transformations from a sequence of elementary ones, and the geometric effect of different 2D affine transformations. Before moving on to 3D transformations it is useful to summarize some general properties of affine transformations. These properties are easy to establish, and because no reference is made of the dimensionality of the objects being transformed, they apply equally well to 3D affine transformations. The only fact about 3D transformations we need at this point is that, like their 2D counterparts, they can be represented in homogeneous coordinates by a matrix.

1). Affine transformations *preserve* affine combinations of points.

We know that an affine combination of two points P_1 and P_2 is the point

$W = a_1 P_1 + a_2 P_2$, where $a_1 + a_2 = 1$

What happens when we apply an affine transformation $T()$ to this point W? We claim $T(W)$ is the *same* affine combination of the transformed points, that is:

Claim: $T(a_1 P_1 + a_2 P_2) = a_1 T(P_1) + a_2 T(P_2)$, (5.19)

For instance, $T(0.7\ (2, 9) + 0.3\ (1, 6)) = 0.7\ T((2, 9)) + 0.3\ T((1, 6))$.

The truth of this is simply a matter of linearity. Using homogeneous coordinates, the point $T(W)$ is $\tilde{M}\tilde{W}$, and we can do the following steps using linearity of matrix multiplication :

$$\tilde{M}\tilde{W} = \tilde{M}(a_1\tilde{P}_1 + a_2\tilde{P}_2) = a_1\tilde{M}\tilde{P}_1 + a_2\tilde{M}\tilde{P}_2$$

which in ordinary coordinates is just $a_1T(P_1) + a_2T(P_2)$ as claimed. The property that affine combinations of points are preserved under affine transformations seems fairly elementary and abstract, but it turns out to be pivotal. It is sometimes taken as the *definition* of what an affine transformation is.

2). Affine transformations preserve lines and planes.

Affine transformations preserve collinearity and "flatness", and so the image of a straight line is another straight line. To see this, recall that the parametric representation $L(t)$ of a line through A and B is itself an affine combination of A and B:

$L(t) = (1 - t)\ A + t\ B$

This is an affine combination of points, so by the previous result the image of $L(t)$ is the same affine combination of the images of A and B:

$Q(t) = (1 - t)\ T(A) + t\ T(B)$, (5.20)

This is another straight line passing through $T(A)$ and $T(B)$. In computer graphics this vastly simplifies drawing transformed line segments: We need only compute the two transformed endpoints $T(A)$ and $T(B)$ and then draw a straight line between them! This saves having to transform *each* of the points along the line, which is obviously impossible.

The argument is the same to show that a plane is transformed into another plane. Recall from Equation 4.45 that the parametric representation for a plane can be written as an affine combination of points:

$P(s, t) = sA + tB + (1 - s - t)C$

When each point is transformed this becomes:

$T(P(s, t)) = sT(A) + t\,T(B) + (1 - s - t)T(C)$

which is clearly also the parametric representation of some plane.

Preservation of collinearity and "flatness" guarantees that polygons will transform into polygons, and planar polygons (those whose vertices all lie in a plane) will transform into planar polygons. In particular, triangles will transform into triangles.

3). Parallelism of lines and planes is preserved.

If two lines or planes are parallel, their images under an affine transformation are also parallel. This is easy to show. We first do it for lines. Take an arbitrary line $A + \mathbf{b}t$ having direction $\mathbf{b}$. It transforms to the line given in homogeneous coordinates by $\tilde{M}(\tilde{A} + \tilde{\mathbf{b}}t) = \tilde{M}\tilde{A} + (\tilde{M}\tilde{\mathbf{b}})t$ which has direction vector $\tilde{M}\tilde{\mathbf{b}}$. This new direction does *not* depend on point A. Thus two different lines $A_1 + \mathbf{b}t$ and $A_2 + \mathbf{b}t$ that have the same direction will transform into two lines both having the direction $\tilde{M}\tilde{\mathbf{b}}$, so they are parallel. An important consequence of this property is that *parallelograms map into other parallelograms.*

The same argument applies to planes: its direction vectors (see Equation 4.43) transform into new direction vectors whose values do not depend on the location of the plane. A consequence of this is that parallelepipeds[5] map into other parallelepipeds.

Example 5.2.8. How is a grid transformed?
Because affine transformations map parallelograms into parallelograms they are rather limited in how much they can alter the shape of geometrical objects. To illustrate this apply any 2D affine transformation T to a unit square grid, as in Figure 5.21. Because a grid consists of two sets of parallel lines, T maps the square grid to another grid consisting of two sets of parallel lines. Think of the grid "carrying along" whatever objects are defined in the grid, to get an idea of how the objects are warped by the transformation. This is all that an affine transformation can do: warp figures in the same way that one grid is mapped into another. The new lines can be tilted at any angle; they can be any (fixed) distance apart; and the two new axes need not be perpendicular. And of course the whole grid can be positioned anywhere in the plane.

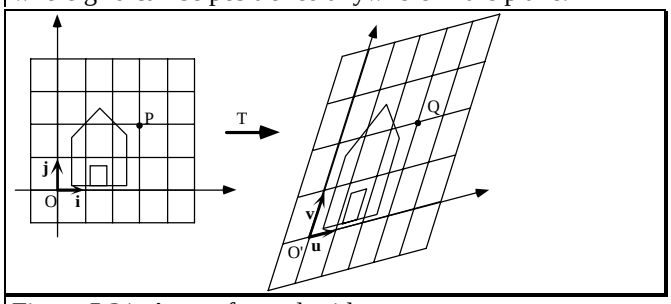

Figure 5.21. A transformed grid.

[5] As we see later, a parallelipiped is the 3D analog of a parallelogram: it has six sides that occur in pairs of parallel faces.

The same result applies in 3D: all a 3D affine transformation can do is map a cubical grid into a grid of parallelipipeds.

4). The Columns of the Matrix reveal the Transformed Coordinate Frame.

It is useful to examine the columns of the matrix M of an affine transformation, for they prescribe how the coordinate frame is transformed. Suppose the matrix M is given by

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = \left(\mathbf{m}_1 \vdots \mathbf{m}_2 \vdots m_3\right) \qquad (5.21)$$

so its columns are $\mathbf{m}_1$, $\mathbf{m}_2$, and m_3. The first two columns are vectors (their third component is 0) and the last column is a point (its third component is a 1). As always the coordinate frame of interest is defined by the origin ϑ, and the basis vectors **i** and **j**, which have representations:

$$\vartheta = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \mathbf{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \text{and } \mathbf{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Notice that vector **i** transforms into the vector $\mathbf{m}_1$ (check this out):

$$\mathbf{m}_1 = M\mathbf{i}$$

and similarly **j** maps into $\mathbf{m}_2$ and ϑ maps into the point m_3. This is illustrated in Figure 5.22a. The coordinate frame (**i**, **j**, ϑ) transforms into the coordinate frame ($\mathbf{m}_1$, $\mathbf{m}_2$, m_3), and these new objects are precisely the columns of the matrix.

a). b).

Figure 5.22. The transformation forms a new coordinate frame.

The axes of the new coordinate frame are not necessarily perpendicular, nor must they be unit length. (They are still perpendicular if the transformation involves only rotations and uniform scalings.) Any point $P = P_x\mathbf{i} + P_y\mathbf{j} + \vartheta$ transforms into $Q = P_x\,\mathbf{m}_1 + P_y\mathbf{m}_2 + m_3$. It is sometimes very revealing to look at the matrix of an affine transformation in this way.

Example 5.2.9. Rotation about a point. The transformation explored in Example 5.2.5 is a rotation of 30° about the point (-2, 3). This yielded the matrix:

$$\begin{pmatrix} .866 & -.5 & 1.232 \\ .5 & .866 & 1.402 \\ 0 & 0 & 1 \end{pmatrix}$$

As shown in Figure 5.22b the coordinate frame therefore maps into the new coordinate frame with origin at (1.232, 1.402, 1) and coordinate axes given by the vectors (0.866, 0.5, 0) and (-0.5, 0.866, 0). Note that these axes are still perpendicular, since only a rotation is involved.

5). Relative Ratios Are Preserved.

Affine transformations have yet another useful property. Consider a point P that lies at the fraction t of the way between two given points, A and B, as shown in Figure 5.23. Apply affine transformation $T(\ )$ to A , B, and P. We claim the transformed point, $T(P)$, also lies the *same* fraction t of the way between the images $T(A)$ and $T(B)$. This is not hard to show (see the exercises).

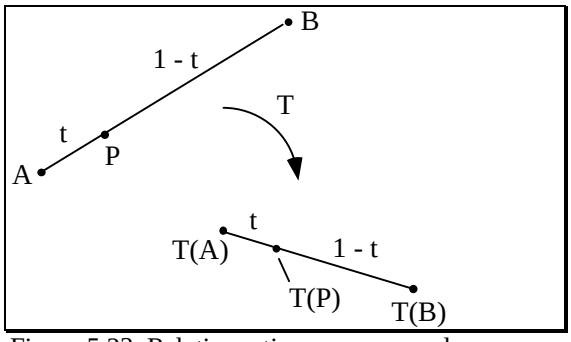

Figure 5.23. Relative ratios are preserved.

As a special case, midpoints of lines map into midpoints. This result pops out a nice geometric result: the diagonals of any parallelogram bisect each other. (Proof: any parallelogram is an affine-transformed square (why?), and the diagonals of a square bisect each other, so the diagonals of a parallelogram also bisect each other.) The same applies in 3D space: the diagonals of any parallelepiped bisect each other.

Interesting Aside. In addition to preserving lines, parallelism, and relative ratios, affine transformations also preserve ellipses and ellipsoids, as we see in Chapter 8!

6). Effect of Transformations on the Areas of Figures.

In CAD applications it is often important to compute the area or volume of an object. For instance, how is the area of a polygon affected when all of its vertices are subjected to an affine transformation? It is clear geometrically that neither translations nor rotations have any effect on the area of a figure, but scalings certainly do, and shearing might.

The result is simple, and is developed in the exercises: When the 2D transformation with matrix M is applied to an object, its area is multiplied by the *magnitude of the determinant* of M:

$$\frac{area\ after\ transformation}{area\ before\ transformation} = |\det M| \qquad (5.22)$$

In 2D the determinant of M in Equation 5.6 is $m_{11}m_{22} - m_{12}m_{21}$.[6] Thus for a pure scaling as in Equation 5.10, the new area is $S_X S_Y$ times the original area, whereas for a shear along one axis the new area is the same as the original area! Equation 5.21 also confirms that a rotation does not alter the area of a figure, since $\cos^2(\theta) + \sin^2(\theta) = 1$.

In 3D similar arguments apply, and we can conclude that the volume of a 3D object is scaled by $|\det M|$ when the object is transformed by the 3D transformation based on matrix M.

Example 5.2.9: The Area of an Ellipse. What is the area of the ellipse that fits inside a rectangle with width W and height H? **Solution**: This ellipse can be formed by scaling the unit circle $x^2 + y^2 = 1$ by the scale factors $S_X = W$ and $S_Y = H$, a transformation for which the matrix M has determinant WH. The unit circle is known to have area π, and so the ellipse has area πWH.

7). Every Affine Transformation is Composed of Elementary Operations.

We can construct complex affine transformations by composing a number of elementary ones. It is interesting to turn the question around and ask, what elementary operations "reside in" a given affine transformation?

[6] The determinant of the homogeneous coordinate version is the same (why?)

Basically a matrix $\tilde{M}$ may be factored into a product of elementary matrices in various ways. One particular way of factoring the matrix $\tilde{M}$ associated with a 2D affine transformation, elaborated upon in Case Study 5.3, yields the result:

$\tilde{M}$ = (shear)(scaling)(rotation)(translation)

That is, any 3 by 3 matrix $\tilde{M}$ that represents a 2D affine transformation can be written as the product of (reading right to left) a translation matrix, a rotation matrix, a scaling matrix, and a shear matrix. The specific ingredients of each matrix are given in the Case Study.

In 3D things are somewhat more complicated. The 4 by 4 matrix $\tilde{M}$ that represents a 3D affine transformation can be written as:

$\tilde{M}$ = (scaling)(rotation)(shear$_1$)(shear$_2$)(translation),

the product of (reading right to left) a translation matrix, a shear matrix, another shear matrix, a rotation matrix, and a scaling matrix. This result is developed in Case study 5.???.

Practice Exercises.
5.2.294 Generalizing the argument. Show that if W is an affine combination of the N points P_i, i = 1,..,N, and $T()$ is an affine transformation, then $T(W)$ is the same affine combination of the N points $T(P_i)$, i = 1,..,N.
5.2.25. Show that relative ratios are preserved. Consider P given by $A + \mathbf{b}t$ where $\mathbf{b} = B - A$. Find the distances $|P - A|$ and $|P - B|$ from P to A and B respectively, showing that they lie in the ratio t to $1 - t$. Is this true if t lies outside of the range 0 to 1? Do the same for the distances $|T(P) - T(A)|$ and $|T(P) - T(B)|$.
5.2.26. Effect on Area. Show that a 2D affine transformation causes the area of a figure to be multiplied by the factor given in Equation 5.27. Hint: View a geometric figure as made up of many very small squares, each of which is mapped into a parallelogram, and then find the area of this parallelogram.

5.3. 3D Affine Transformations.

The same ideas apply to 3D affine transformations as apply to 2D affine transformations, but of course the expressions are more complicated, and it is considerably harder to visualize the effect of a 3D transformation.

Again we use coordinate frames, and suppose that we have an origin ϑ and three mutually perpendicular axes in the directions **i**, **j**, and **k** (see Figure 5.8). Point P in this frame is given by $P = \vartheta + P_x\mathbf{i} + P_y\mathbf{j} + P_z\mathbf{k}$, and so has as the representation

$$\tilde{P} = \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Suppose $T()$ is an affine transformation that transforms point P to point Q. Then just as in the 2D case $T()$ is represented by a matrix $\tilde{M}$ which is now 4 by 4:

$$\tilde{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.23)$$

and we can say that the representation of point Q is found by multiplying P by matrix $\tilde{M}$:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \tilde{M} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} \tag{5.24}$$

Notice that once again for an affine transformation the final row of the matrix is a string of zeroes followed a lone one. (This will cease to be the case when we examine projective matrices in Chapter 7.)

5.3.1. The Elementary 3D Transformations.

We consider the nature of elementary 3D transformations individually, and then compose them into general 3D affine transformations.

Translation.

For a pure translation, the matrix $\tilde{M}$ has the simple form.

$$\begin{pmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Check that $Q = MP$ is simply a shift in Q by the vector $\mathbf{m} = (m_{14}, m_{24}, m_{34})$.

Scaling.

Scaling in three dimensions is a direct extension of the 2D case, having a matrix given by:

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.25}$$

where the three constants S_X, S_Y, and S_Z cause scaling of the corresponding coordinates. Scaling is about the origin, just as in the 2D case. Figure 5.24 shows the effect of scaling in the z-direction by 0.5 and in the x-direction by a factor of two.

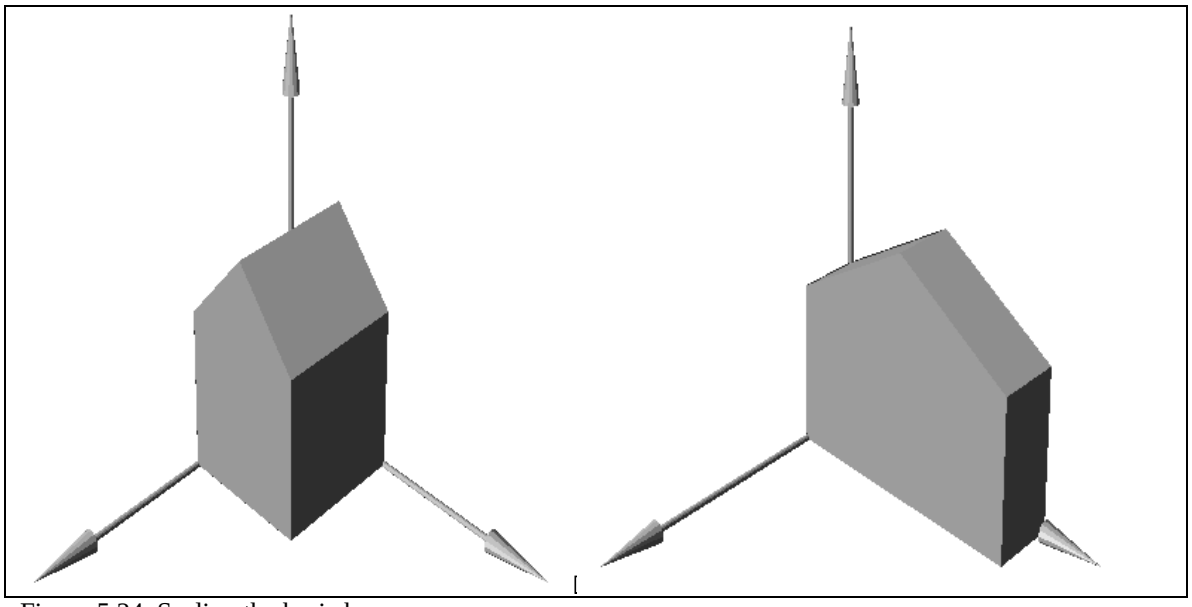

Figure 5.24. Scaling the basic barn.

Notice that this figure shows various lines before and after being transformed. It capitalizes on the important fact that straight lines transform to straight lines.

Shearing.

Three-dimensional shears appear in greater variety than do their two-dimensional counterparts. The matrix for the simplest elementary shear is the identity matrix with one zero term replaced by some value, as in

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.26)$$

which produces $Q = (P_x, f P_x + P_y, P_z)$; that is, P_y is offset by some amount proportional to P_x, and the other components are unchanged. This causes an effect similar to that in 2D shown in Figure 5.15. Goldman [goldman???] has developed a much more general form for a 3D shear, which is described in Case Study 5.???.

Rotations.

Rotations in three dimensions are common in graphics, for we often want to rotate an object or a camera in order to obtain different views. There is a much greater variety of rotations in three than in two dimensions, since we must specify an axis about which the rotation occurs, rather than just a single point. One helpful approach is to decompose a rotation into a combination of simpler ones.

Elementary rotations about a coordinate axis.

The simplest rotation is a rotation about one of the coordinate axes. We call a rotation about the *x*-axis an "*x*-roll", a rotation about the *y*-axis a "*y*-roll", and one about the z-axis a "z-roll". We present individually the matrices that produce an *x*-roll, a *y*-roll, and a *z*-roll. In each case the rotation is through an angle, β, about the given axis. We define positive angles using a "looking inward" convention:

Positive values of β cause a counterclockwise (CCW) rotation about an axis as one looks inward from a point on the positive axis toward the origin.

The three basic positive rotations are illustrated in Figure 5.25.[7]

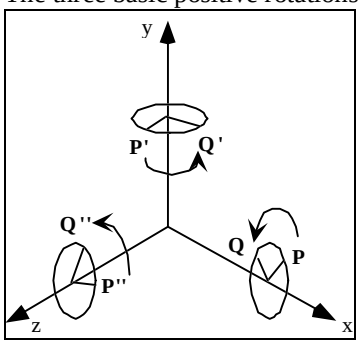

Figure 5.25. Positive rotations about the three axes.

This formulation is also consistent with our notion of 2D rotations: a positive rotation in two dimensions is equivalent to a *z*-roll as we look at the *xy*-plane from a point on the positive *z*-axis.

Notice what happens with this convention for the particular case of a 90° rotation:

- For a *z*-roll, the *x*-axis rotates to the *y*-axis.
- For an *x*-roll, the *y*-axis rotates to the *z*-axis.
- For a *y*-roll, the *z*-axis rotates to the *x*-axis.

The following three matrices represent transformations that rotate points through angle β about a coordinate axis. We use the suggestive notation $R_X(\ )$, $R_Y(\ )$, and $R_Z(\ )$ to denote *x*-, *y*-, and *z*-rolls, respectively. The parameter is the angle through which points are rotated, given in radians, and c stands for $\cos(\beta)$ and s for $\sin(\beta)$.

1. An *x*-roll:

$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.27}$$

2. A *y*-roll:

$$R_y(\beta) = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.28}$$

3. A *z*-roll:

[7] In a left-handed system the sense of a rotation through a positive β would be CCW looking **outward** along the positive axis from the origin. This formulation is used by some authors.

$$R_z(\beta) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.29}$$

Note that 12 of the terms in each matrix are the zeros and ones of the identity matrix. They occur in the row and column that correspond to the axis about which the rotation is being made (e.g., the first row and column for a *x*-roll). They guarantee that the corresponding coordinate of the point being transformed will not be altered. The *c* and *s* terms always appear in a rectangular pattern in the other rows and columns.

Aside: Why is the *y*-roll different? The *-s* term appears in the lower row for the *x*- and *z*-rolls, but in the upper row for the *y*- roll. Is a *y*-roll inherently different in some way? This question is explored in the exercises.

Example 5.3.1. Rotating the barn. Figure 5.26 shows a "barn" in its original orientation (part a), and after a -70° *x*-roll (part b), a 30° *y*-roll (part c), and a -90° *z*-roll (part d).

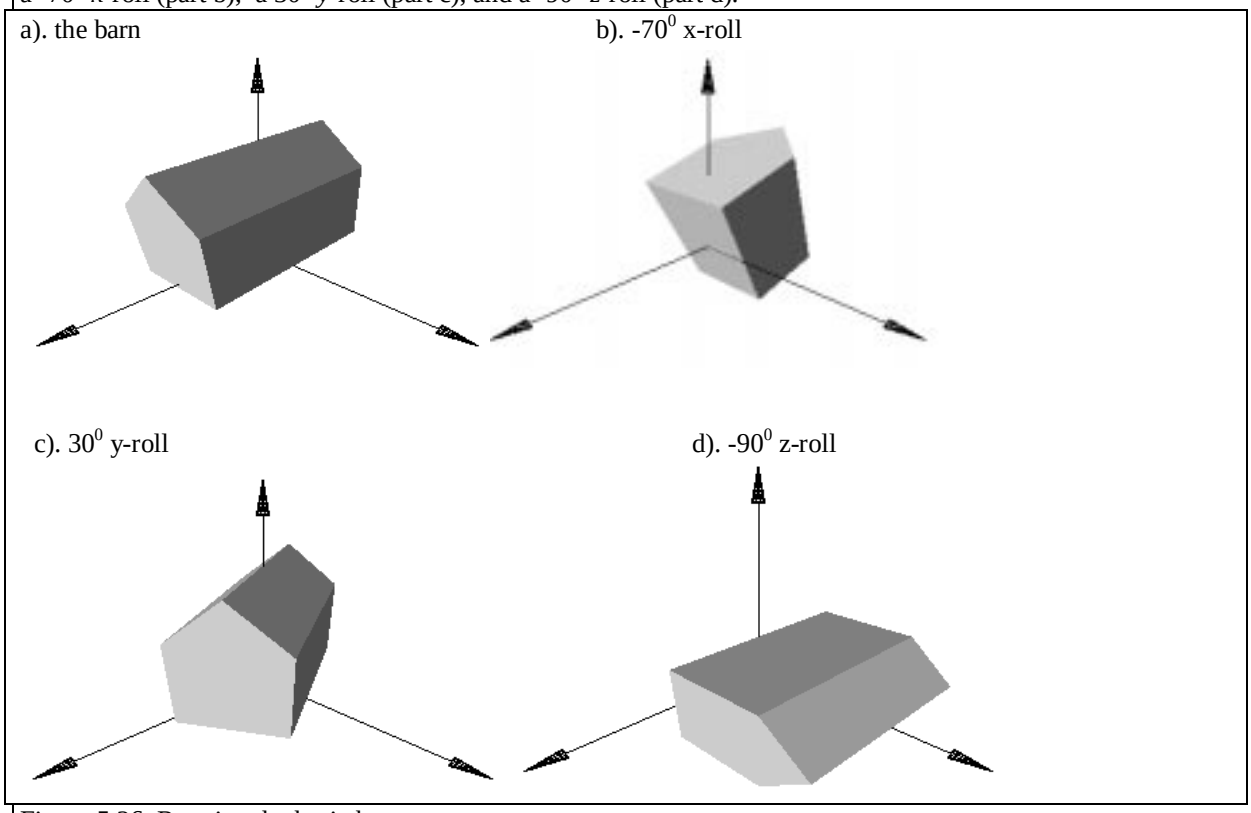

Figure 5.26. Rotating the basic barn.

Example 5.3.2. Rotate the point P = (3, 1, 4) through 30° about the *y*-axis. **Solution**: Using Equation 9.9.10 with c = .866 and s = .5, P is transformed into

$$Q = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 4.6 \\ 1 \\ 1.964 \\ 1 \end{pmatrix}$$

As expected, the *y*-coordinate of the point is not altered.

Practice Exercises.

5.3.1. Visualizing the 90° Rotations. Draw a right-handed 3D system and convince yourself that a 90° rotation (CCW looking toward the origin) about each axis rotates the other axes into one another, as specified in the preceding list. What is the effect of rotating a point on the *x*-axis about the *x*-axis?

5.3.2. Rotating the Basic Barn. Sketch the basic barn after each vertex has experienced a 45° *x*-roll. Repeat for *y*- and *z*-rolls.

5.3.3. Do a Rotation. Find the image *Q* of the point *P* = (1, 2, -1) after a 45° *y*-roll. Sketch *P* and *Q* in a 3D coordinate system and show that your result is reasonable.

5.3.4. Testing 90° Rotations of the Axes. This exercise provides a useful trick for remembering the form of the rotation matrices. Apply each of the three rotation matrices to each of the standard unit position vectors, **i**, **j**, and **k**, using a 90° rotation. In each case discuss the effect of the transformation on the unit vector.

5.3.5. Is a y-roll indeed different? The minus sign in Equation 5.28 seems to be in the wrong place: on the lower *s* rather than the upper one. Here you show that Equations 5.27-29 are in fact consistent. It's just a matter of how things are ordered. Think of the three axes *x*, *y*, and *z* as occurring cyclically: *x* -> *y* -> *z* -> *x* -> *y* …, etc. If we are discussing a rotation about some "current" axis (*x*-, *y*-, or *z*-) then we can identify the "previous" axis and the "next" axis. For instance, if *x*- is the current axis, then the previous one is *z*- and the next is *y*-. Show that with this naming all three types of rotations use the same equations: $Q_{curr} = P_{curr}$, $Q_{next} = c\,P_{next} - s\,P_{prev}$, and $Q_{prev} = s\,P_{next} + c\,P_{prev}$. Write these equations out for each of the three possible "current" axes.

5.3.2. Composing 3D Affine Transformations.

Not surprisingly, 3D affine transformations can be composed, and the result is another 3D affine transformation. The thinking is exactly parallel to that which led to Equation 5.17 in the 2D case. The matrix that represents the overall transformation is the product of the individual matrices M_1 and M_2 that perform the two transformations, with M_2 *premultiplying* M_1:

$$\tilde{M} = \tilde{M}_2\,\tilde{M}_1. \qquad (5.30)$$

Any number of affine transformations can be composed in this way, and a single matrix results that represents the overall transformation.

Figure 5.27 shows an example, where a barn is first transformed using some M_1, then that transformed barn is again transformed using M_2. The result is the same as the barn transformed once using M_2M_1.

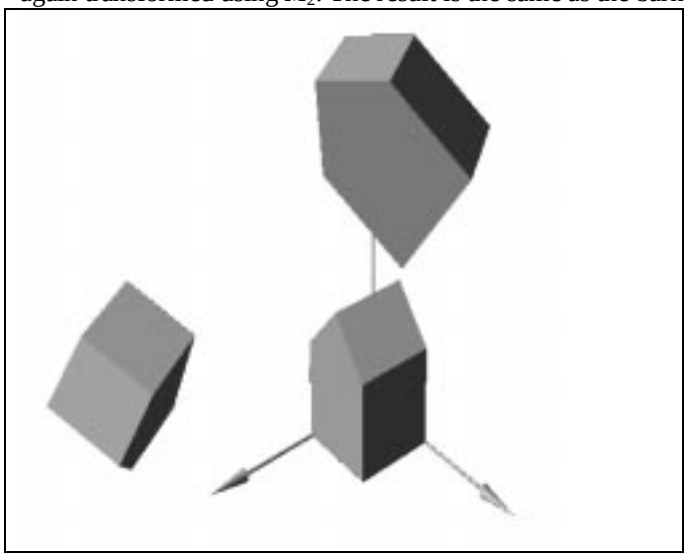

Figure 5.27. Composing 3D affine transformations. (file: fig5.27.bmp)

5.3.3. Combining Rotations.

Results! Why, man, I have gotten a lot of results.

I know several thousand things that won't work.
Thomas A. Edison

One of the most important distinctions between 2D and 3D transformations is the manner in which rotations combine. In 2D two rotations, say $R(\beta_1)$ and $R(\beta_2)$, combine to produce $R(\beta_1+\beta_2)$, and the order in which they are combined makes no difference. In 3D the situation is much more complicated, because rotations can be about different axes. The order in which two rotations about different axes are performed *does* matter: 3D rotation matrices do **not** commute. We explore some properties of 3D rotations here, investigating different ways that a rotation can be represented, and see how to create rotations that do a certain job.

It's very common to build a rotation in 3D by composing three elementary rotations: an *x*-roll followed by a *y*-roll, and then a *z*-roll. Using the notation of Equations 5.27-29 for each individual roll, the overall rotation is given by

$$M = R_x(\beta_x)R_y(\beta_y)R_z(\beta_z) \tag{5.31}$$

In this context the angles β_1, β_2, and β_3 are often called **Euler**[8] **angles**. One form of **Euler's Theorem** asserts that *any* 3D rotation can be obtained by three rolls about the *x*-, *y*-, and *z*-axes, so any rotation can be written as in Equation 5.32 for the appropriate choice of Euler angles. This implies that it takes three values to completely specify a rotation.

Example 5.3.3. What is the matrix associated with an *x*-roll of 45^0 followed by a *y*-roll of 30^0 followed by a *z*-roll of 60^0 ? Direct multiplication of the three component matrices (in the proper "reverse" order) yields:

$$\begin{pmatrix} .5 & -.866 & 0 & 0 \\ .866 & .5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} .866 & 0 & .5 & .0 \\ 0 & 1 & 0 & 0 \\ -.5 & 0 & .866 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & .707 & -.707 & 0 \\ 0 & .707 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} .433 & -.436 & .789 & 0 \\ .75 & .66 & -.047 & 0 \\ -.5 & .612 & .612 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Some people use a different ordering of "rolls" to create a complicated rotation. For instance, they might express a rotation as $R_y(\beta_1)R_z(\beta_2)R_x(\beta_3)$: first a *y*-roll then a *z*-roll then an *x*-roll. Because rotations in 3D do not commute this requires the use of different Euler angles β_1, β_2, and β_3 to create the same rotation as in Equation 5.32. There are 12 possible orderings of the three individual rolls, and each uses different values for β_1, β_2, and β_3.

Rotations About an Arbitrary Axis.
When using Euler angles we perform a sequence of *x*-, *y*-, and *z*-rolls, that is, rotations about a coordinate axis. But it can be much easier to work with rotations if we have a way to rotate about an axis that points in an arbitrary direction. Visualize the earth, or a toy top, spinning about a tilted axis. In fact, Euler's theorem states that every rotation can be represented as one of this type:

Euler's Theorem: *Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point.*[9]

What is the matrix for such a rotation, and can we work with it conveniently?

Figure 5.28 shows an axis represented by vector **u**, and an arbitrary point *P* that is to be rotated through angle β about **u** to produce point *Q*.

---

[8] Leonhard Euler, 1707-1783, a Swiss mathematician of extraordinary ability who made important contributions to all branches of mathematics.

[9] This is sometimes stated: Given two rectangular coordinate systems with the same origin and arbitrary directions of axes, one can always specify a line through the origin such that one coordinate system goes into the other by a rotation about this line. [gellert75]

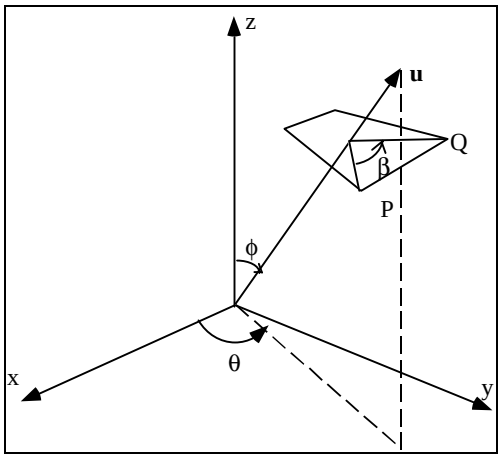

Figure 5.28. Rotation about an axis through the origin.

Because **u** can have any direction, it would seem at first glance to be very difficult to find a single matrix that represents such a rotation. But in fact it can be found in two rather different ways, a classic way and a constructive way.

1). The classic way. Decompose the required rotation into a sequence of known steps:
1. Perform two rotations so that **u** becomes aligned with the *z*-axis.
2. Do a *z*-roll through angle β.
3. Undo the two alignment rotations to restore **u** to its original direction.

This is reminiscent of rotating about a point in two dimensions: The first step prepares the situation for a simpler known operation; the simple operation is done; and finally the preparation step is undone. The result (discussed in the exercises) is that the transformation requires the multiplication of five matrices:

$$R_{\mathbf{u}}(\beta) = R_Z(-\theta)\, R_y(-\phi)\, R_Z(\beta)\, R_y(\phi)\, R_Z(\theta) \qquad (5.32)$$

each being a rotation about one of the coordinate axes. This is tedious to do by hand but is straightforward to carry out in a program. However, expanding out the product gives little insight into how the ingredients go together.

2). The constructive way. Using some vector tools we can obtain a more revealing expression for the matrix $R_{\mathbf{u}}(\beta)$. This approach has become popular recently, and versions of it are described by several authors in GEMS I [glass90]. We adapt the derivation of Maillot [mail90].

Figure 5.29 shows the axis of rotation **u**, and we wish to express the operation of rotating point *P* through angle β into point *Q*. The method, spelled out in Case Study 5.5, effectively establishes a 2D coordinate system in the plane of rotation as shown. This defines two orthogonal vectors **a** and **b** lying in the plane, and as shown in Figure 5.29b point *Q* is expressed as a linear combination of them. The expression for *Q* involves dot products and cross products of various ingredients in the problem. But because each of the terms is linear in the coordinates of *P*, it can be rewritten as *P* times a matrix.

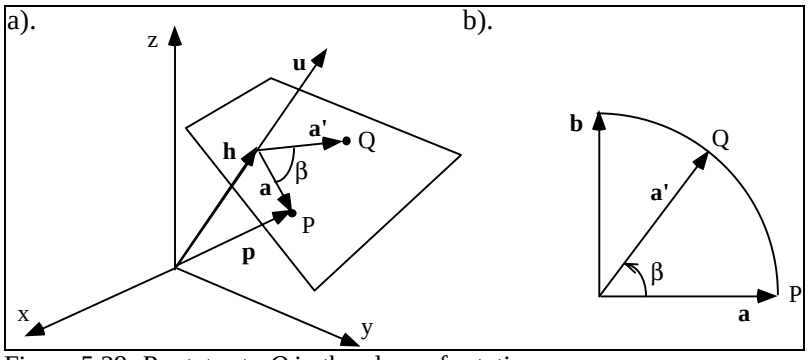

Figure 5.29. *P* rotates to *Q* in the plane of rotation.

The final result is the matrix:

$$R_{\mathbf{u}}(\beta) = \begin{pmatrix} c+(1-c)u_x^2 & (1-c)u_yu_x - su_z & (1-c)u_zu_x + su_y & 0 \\ (1-c)u_xu_y + su_z & c+(1-c)u_y^2 & (1-c)u_zu_y - su_x & 0 \\ (1-c)u_xu_z - su_y & (1-c)u_yu_z + su_x & c+(1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.33)$$

where $c = cos(\beta)$, and $s = sin(\beta)$, and (u_x, u_y, u_z) are the components of the unit vector **u**. This looks more complicated than it is. In fact, as we see later, there is so much structure in the terms that, given an arbitrary rotation matrix, we can find the specific axis and angle that produces the rotation (which proves Euler's theorem).

As we see later, OpenGL provides a function to create a rotation about an arbitrary axis:

```
glRotated(angle, ux, uy, uz);
```

Example 5.3.4. Rotating about an axis. Find the matrix that produces a rotation through 45^0 about the axis $\mathbf{u} = (1,1,1)/\sqrt{3} = (0.577,0.577,0.577)$. **Solution:** For a 45^0 rotation, $c = s = 0.707$, and filling in the terms in Equation 5.33 we obtain:

$$R_{\mathbf{u}}(45^0) = \begin{pmatrix} .8047 & -.31 & .5058 & 0 \\ .5058 & .8047 & -.31 & 0 \\ -.31 & .5058 & .8047 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This has a determinant of 1 as expected. Figure 5.30 shows the basic barn, shifted away from the origin, before it is rotated (dark), after a rotation through 22.5^o (medium), and after a rotation of 45^o (light).

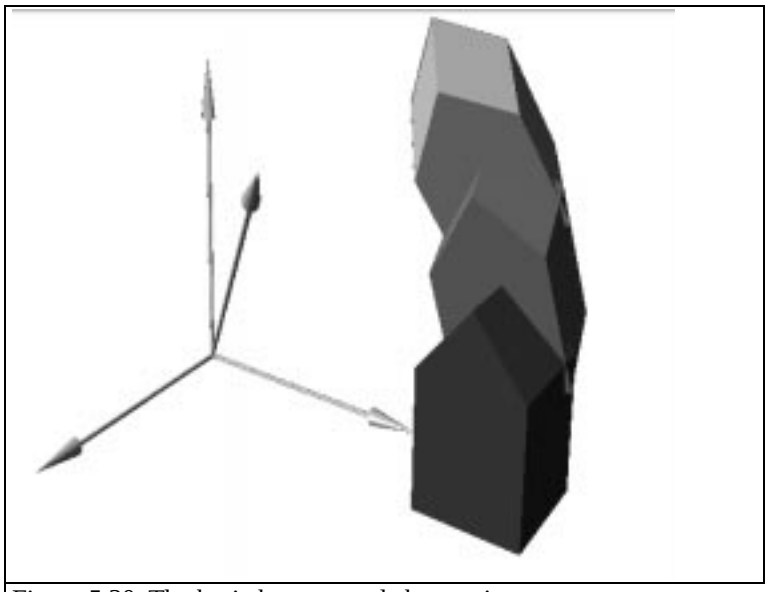

Figure 5.30. The basic barn rotated about axis **u.**

Finding the Axis and Angle of Rotation.
Euler's theorem guarantees that any rotation is equivalent to a rotation about some axis. It is useful, when presented with some rotation matrix, to determine the specific axis and angle. That is, given values m_{ij} for the matrix:

$$R_{\mathbf{u}}(\beta) = \begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

extract the angle β and the unit vector **u**.

This is surprisingly easy to do by examining Equation 5.33[watt92]. First note that the trace of $R_u(\beta)$, that is the sum of the three diagonal elements, is $3c + (1-c)(u_x^2 + u_y^2 + u_z^2) = 1 + 2\cos(\beta)$. So we can solve for *cos*(β) directly:

$$\cos(\beta) = \tfrac{1}{2}(m_{11} + m_{22} + m_{33} - 1).$$

Take the arc cosine of this value to obtain β, and use it to find $s = sin(\beta)$ as well. Now see that pairs of elements of the matrix combine to reveal the individual components of **u**:

$$u_x = \frac{m_{32} - m_{23}}{2\sin(\beta)}$$
$$u_y = \frac{m_{13} - m_{31}}{2\sin(\beta)} \qquad (5.34)$$
$$u_z = \frac{m_{21} - m_{12}}{2\sin(\beta)}$$

Example 5.3.5. Find the axis and angle. Pretend you don't know the underlying axis and angle for the rotation matrix in Example 5.3.3, and solve for it. The trace is 2.414, so $\cos(\beta) = 0.707$, β must be 45^0, and $\sin(\beta) = 0.707$. Now calculate each of the terms in Equation 5.35: they all yield the value 0.577, so $\mathbf{u} = (1,1,1)/\sqrt{3}$, just as we expected.

Practice Exercises.

5.3.6. Which ones commute? Consider two affine transformations T_1 and T_2. Is T_1T_2 the same as T_2T_1 when:
a). They are both pure translations? b). They are both scalings? c). They are both shears?
d). One is a rotation and one a translation? e). One is a rotation and one is a scaling?
f). One is a scaling and one is a shear?

5.3.7. Special cases of rotation about a general axis u. It always helps to see that a complicated result collapses to a familiar one in special cases. Check that this happens in Equation 5.34 when **u** is itself
a). the *x*-axis, **i**; b). the *y*-axis, **j**; c). the *z*-axis, **k**.

5.3.8. Classic Approach to Rotation about an Axis. Here we suggest how to find the rotations that cause **u** to become aligned with the *z*-axis. (See Appendix 2 for a review of spherical coordinates.) Suppose the direction of **u** is given by the spherical coordinate angles ϕ and θ as indicated in Figure 5.27. Align **u** with the *z*-axis by a *z*-roll through $-\theta$: this swings **u** into the *xz*-plane to form the new axis, **u'** (sketch this). Use Equation 5.29 to obtain $R_Z(-\theta)$. Second, a *y*-roll through $-\phi$ completes the alignment process. With **u** aligned along the *z*-axis, do the desired *z*-roll through angle β, using Equation 5.29. Finally, the alignment rotations must be undone to restore the axis to its original direction. Use the inverse matrices to $R_Y(-\phi)$ and $R_Z(-\theta)$, which are $R_Y(\phi)$ and $R_Z(\theta)$, respectively. First undo the *y*-roll and then the *z*-roll. Finally, multiply these five elementary rotations to obtain Equation 5.34. Work out the details, and apply them to find the matrix *M* that performs a rotation through angle 35° about the axis situated at θ=30° and ϕ = 45°. Show that the final result is:

$$\tilde{M} = \begin{pmatrix} .877 & -.366 & .281 & 0 \\ .445 & .842 & -.306 & 0 \\ -.124 & .396 & .910 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5.3.9. Orthogonal Matrices. A matrix is **orthogonal** if its columns are mutually orthogonal unit-length vectors. Show that each of the three rotation matrices given in Equations 5.27-29 is orthogonal. What is the determinant of an orthogonal matrix? An orthogonal matrix has a splendid property: *Its inverse is identical to its transpose* (also see Appendix 2). Show why the orthogonality of the columns guarantees this. Find the inverse of each of the three rotation matrices above, and show that the inverse of a rotation is simply a rotation in the opposite direction.

5.3.10. The Matrix Is Orthogonal. Show that the complicated rotation matrix in Equation 5.34 is orthogonal .

5.3.11. Structure of a rotation matrix. Show that for a 3x3 rotation *M* the three rows are pair-wise orthogonal, and the third is the cross product of first two.

5.3.12. What if the axis of rotation does not pass through the origin? If the axis does not pass through the origin but instead is given by $S + \mathbf{u}t$ for some point *S*, then we must first translate to the origin through *-S*, apply the appropriate rotation, and then translate back through *S*. Derive the overall matrix that results.

5.3.4. Summary of Properties of 3D Affine Transformations.

The properties noted for affine transformations in Section 5.2.8 apply, of course, to 3D affine transformations. Stated in terms of any 3D affine transformation $T(.)$ having matrix *M*, they are:

- **Affine transformations *preserve* affine combinations of points**. If $a + b = 1$ then $aP + bQ$ is a meaningful 3D point, and $T(aP + bQ) = aT(P) + bT(Q)$.
- **Affine transformations preserve lines and planes.** Straightness is preserved: The image $T(L)$ of a line *L* in 3D space is another straight line; the image $T(W)$ of a plane *W* in 3D space is another plane.
- **Parallelism of lines and planes is preserved.** If *W* and *Z* are parallel lines (or planes), then $T(W)$ and $T(Z)$ are also parallel.

- **The Columns of the Matrix reveal the Transformed Coordinate Frame.** If the columns of M are the vectors $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, and the point m_4, the transformation maps the frame ($\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$, ϑ) to the frame ($\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, m_4).
- **Relative Ratios Are Preserved.** If P is fraction f of the way from point A to point B, then $T(P)$ is also fraction f of the way from point $T(A)$ to $T(B)$.
- **Effect of Transformations on the Areas of Figures.** If 3D object D has volume V, then its image $T(D)$ has volume $|det\ M\ |\ V$, where $|det\ M|$ is the absolute value of the determinant of M.
- **Every Affine Transformation is Composed of Elementary Operations.** A 3D affine transformation may be decomposed into a composition of elementary transformations. This can be done in several ways.

5.4. Changing Coordinate Systems.

There's another way to think about affine transformations. In many respect it is a more natural approach when modeling a scene. Instead of viewing an affine transformation as producing a different point in a fixed coordinate system, you think of it as producing a new coordinate system in which to represent points.

Aside: a word on notation: To make things fit better on the printed page, we shall sometimes use the notation $(P_x, P_y, 1)^T$ in place of $\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$. (Also see Appendix 2.) The superscript T denotes the **transpose**, so we are simply writing the column vector as a transposed row vector.

Suppose we have a 2D coordinate frame #1 as shown in Figure 5.31, with origin ϑ and axes $\mathbf{i}$ and $\mathbf{j}$. Further suppose we have an affine transformation $T(.)$ represented by matrix M. So $T(.)$ transforms coordinate frame #1 into coordinate frame #2, with new origin $\vartheta' = T(\vartheta)$, and new axes $\mathbf{i}' = T(\mathbf{i})$ and $\mathbf{j}' = T(\mathbf{j})$.

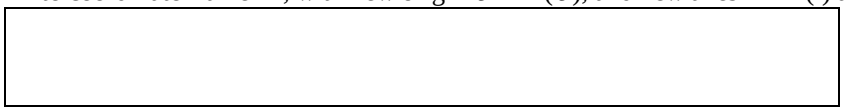

Figure 5.31. Transforming a coordinate frame.

Now let P be a point with representation $(c, d, 1)^T$ in the new system #2. What are the values of a and b in its representation $(a, b, 1)^T$ in the original system #1? The answer: just multiply $(c, d, 1)^T$ by M:

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} \qquad (5.35)$$

Summarizing: Suppose coordinate system #2 is formed from coordinate system #1 by the affine transformation M. Further suppose that point $P = (P_x, P_y, P_z, 1)$ are the coordinates of a point P expressed in system #2. Then the coordinates of P expressed in system #1 are MP.

This may seem obvious to some readers, but in case it doesn't, a derivation is developed in the exercises. This result also holds for 3D systems, of course, and we use it extensively when calculating how 3D points are transformed as they are passed down the graphics pipeline.

Example 5.4.1. Rotating a coordinate system. Consider again the transformation of Example 5.2.5 that rotates points through 30^o about the point (-2, 3). (See Figure 5.22.) This transformation maps the origin ϑ and axes $\mathbf{i}$ and $\mathbf{j}$ into the system #2 as shown in that figure. Now consider the point P with coordinates $(P_x, P_y, 1)^T$ in the *new* coordinate system. What are the coordinates of this point expressed in the *original* system #1? The answer is simply MP. For instance, $(1, 2, 1)^T$ in the new system lies at $M(1, 2, 1)^T = (1.098, 3.634, 1)^T$ in the original system. (Sketch this in the figure.) Notice that the point $(-2, 3, 1)^T$, the center of rotation of the transformation, is

a *fixed point* of the transformation: $M(2, 3, 1)^T = (2, 3, 1)^T$. Thus if we take $P = (-2, 3, 1)^T$ in the new system, it maps to $(-2, 3, 1)^T$ in the original system (check this visually).

Successive Changes in a Coordinate frame.
Now consider forming a transformation by making two successive changes of the coordinate system. What is the overall effect? As suggested in Figure 5.32, system #1 is converted to system #2 by transformation $T_1(.)$, and system #2 is then transformed to system #3 by transformation $T_2(.)$. Note that system #3 is transformed *relative* to #2.

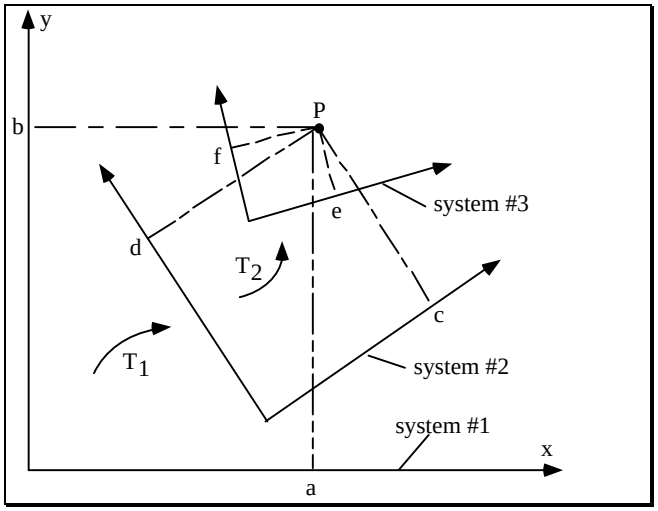

Figure 5.32. Transforming a coordinate system twice.

Again the question is: if point *P* has representation $(e, f, 1)^T$ with respect to system #3, what are its coordinates $(a, b, 1)^T$ with respect to the original system #1?

To answer this just work backwards and collect the effects of each transformation. In terms of system #2 the point *P* has coordinates $(c, d, 1)^T = M_2(e, f, 1)^T$. And in terms of system #1 the point $(c, d, 1)^T$ has coordinates $(a, b, 1)^T = M_1(c, d, 1)^T$. Putting these together:

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M_1 \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} = M_1 M_2 \begin{pmatrix} e \\ f \\ 1 \end{pmatrix} \qquad (5.36)$$

The essential point is that when determining the desired coordinates $(a, b, 1)^T$ from $(e, f, 1)^T$ we *first* apply M_2 and *then* M_1, just the *opposite* order as when applying transformations to points.

We summarize this fact for the case of three successive transformations. The result generalizes immediately to any number of transformations.

Transforming points. To apply a sequence of transformations $T_1()$, $T_2()$, $T_3()$ (in that order) to a point *P*, form the matrix:

$$M = M_3 \times M_2 \times M_1$$

Then *P* is transformed to *MP*. To compose each successive transformation M_i you must *premultiply* by M_i.

Transforming the coordinate system. To apply a sequence of transformations $T_1()$, $T_2()$, $T_3()$ (in that order) to the coordinate system, form the matrix:

$$M = M_1 \times M_2 \times M_3$$

Then a point P expressed in the transformed system has coordinates MP in the original system. To compose each additional transformation M_i you must *postmultiply* by M_i.

How OpenGL operates.
We shall see in the next section that OpenGL provides tools for successively applying transformations in order to build up an overall "current transformation". In fact OpenGL is organized to *postmultiply* each new transformation matrix to combine it with the current transformation. Thus it will often seem more natural to the modeler to think in terms of successively transforming the coordinate system involved, as the order in which these transformations is carried out is the *same* as the order in which OpenGL computes them.

Practice Exercises.

5.4.1. How transforming a coordinate system relates to transforming a point.
We wish to show the result in Equation 5.35. To do this, show each of the following steps.
a). Show why the point P with representation $(c, d, 1)^T$ used in system #2 lies at $c\mathbf{i}' + d\mathbf{j}' + \mathcal{O}'$.
b). We want to find where this point lies in system #1. Show that the representation (in system #1) of $\mathbf{i}'$ is $M(1, 0, 0)^T$, that of $\mathbf{j}'$ is $M(0, 1, 0)^T$, and that of $\mathcal{O}'$ is $M(0, 0, 1)$.
c). Show that therefore the representation of the point $c\mathbf{i}' + d\mathbf{j}' + \mathcal{O}'$ is $cM(1, 0, 0)^T + dM(0, 1, 0)^T + M(0, 0, 1)^T$.
d). Show that this is the same as $M(c, 0, 0)^T + M(0, d, 0)^T + M(0, 0, 1)$ and that this is $M(c, d, 1)^T$, as claimed.

5.4.2. Using elementary examples. Figure 5.33 shows the effect of four elementary transformations of a coordinate system. In each case the original system with axes x and y is transformed into the new system with axes x' and y'.

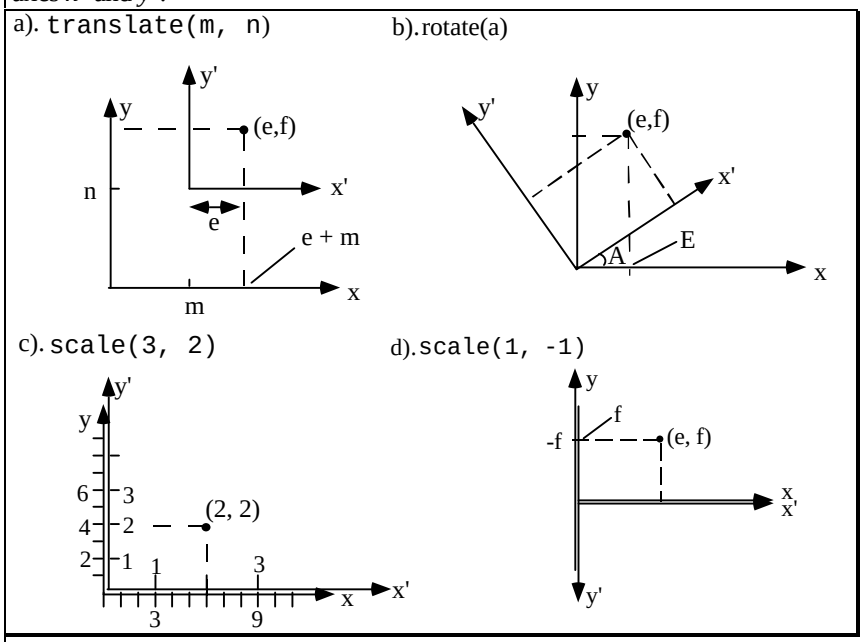

Figure 5.33. Elementary changes between coordinate systems.

a). Part a) shows the effect of a translation through (m, n). Show that point (e, f) in the new system lies at $(e + m, f + n)$ in the original system.
b). Part b) shows the effect of a rotation about the origin through A degrees. Show that the point (e, f) in the new system lies at $(e\ cos(a) - f\ sin(a), e\ sin(a) + f\ cos(a))$, where $a = \pi A / 180$ radians.
c). Part c) shows the effect of a scaling of the axes by (3, 2). To make the figure clearer the new and old axes are shown slightly displaced. Show that a point (e, f) in the new system lies at $(3e, 2f)$ in the original system.
d). part d) shows a special case of scaling, a reflection about the x-axis. Show that the point (e, f) lies in the original system at $(e, -f)$.

5.5. Using Affine Transformations in a Program.

We want to see how to apply the theory of affine transformations in a program to carry out scaling, rotating, and translating of graphical objects. We also investigate how it is done when OpenGL is used. We look at 2D examples first as they are easier to visualize, then move on to 3D examples.

To set the stage, suppose you have a routine `house()` that draws the house #1 in Figure 5.34. But you wish to draw the version #2 shown that has been rotated through -30^0 and then translated through (32, 25). This is a frequently encountered situation: an object is defined at a convenient size and position, but we want to draw it (perhaps many times) at different sizes, orientations, and locations.

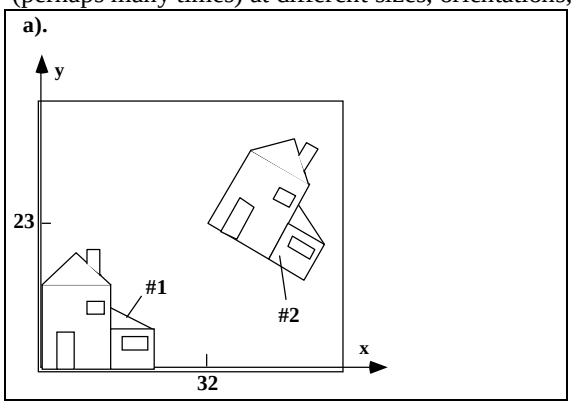

Figure 5.34. Drawing a rotated and translated house.

As we discussed in Chapter 3, `house()` would draw the various polylines of the figure. If it were written in "raw" OpenGL it might consist of a large number of chunks like:

```
glBegin(GL_LINES);
      glVertex2d(V[0].x, V[0].y);
      glVertex2d(V[1].x, V[1].y);
      glVertex2d(V[2].x, V[2].y);
      .... // the remaining points
glEnd();
```

based on some array `V[]` of points. Or if we use the Canvas class developed in Chapter 3 there would be a number of calls to `moveTo()` and `lineTo()` as in (using the global canvas object `cvs`):

```
cvs.moveTo(V[0]);
cvs.lineTo(V[1]);
cvs.lineTo(V[2]);
...  // the remaining points
```

In either case we would set up a world window and a viewport with calls like:

```
cvs.setWindow(...);
cvs.setViewport(...);
```

and we would be assured that all vertex positions `V[i]` are "quietly" converted from world coordinates to screen window coordinates by the underlying window to viewport transformation.

But how do we arrange matters so that house #2 is drawn instead? There is the hard way and the easy way.

The hard way.
With this approach we construct the matrix for the desired transformation, say `M`, and build a routine, say `transform2D()`, that transforms one point into another, such that:

```
Q =  transform2D(M, P);
```

The routine produces $\tilde{Q} = \tilde{M}\tilde{P}$. To apply the transformation to each point `V[i]` in `house`() we must adjust the source code above as in

```
cvs.moveTo(transform2D(M, V[0])); // move to the transformed point
cvs.lineTo(transform2D(M, V[1]));
cvs.lineTo(transform2D(M, V[2]));
...
```

so that the *transformed* points are sent to `moveTo`() and `lineTo`(). This is workable if the source code for `house`() is at hand. But it is cumbersome at best, and *not possible* at all if the source code for `house`() is not available. It also requires tools to create the matrix `M` in the first place.

The easy way.
We cause the desired transformation to be applied automatically to each vertex. Just as we know the window to viewport mapping is "quietly" applied to each vertex as part of `moveTo`() and `lineTo`(), we can have an additional transformation be quietly applied as well. It is often called the **current transformation,** *CT*. We enhance `moveTo`() and `lineTo`() in the Canvas class so that they first quietly apply this transformation to the argument vertex, and then apply the window to viewport mapping. (Clipping is performed at the world window boundary as well.)

Figure 5.35 provides a slight elaboration of the graphics pipeline we introduced in Figure 5.7. When `glVertex2d()`is called with argument `V`, the vertex *V* is first transformed by the *CT* to form point *Q*. *Q* is then passed through the window to viewport mapping to form point *S* in the screen window. (As we see later, clipping is also performed, "inside" this last mapping process.)

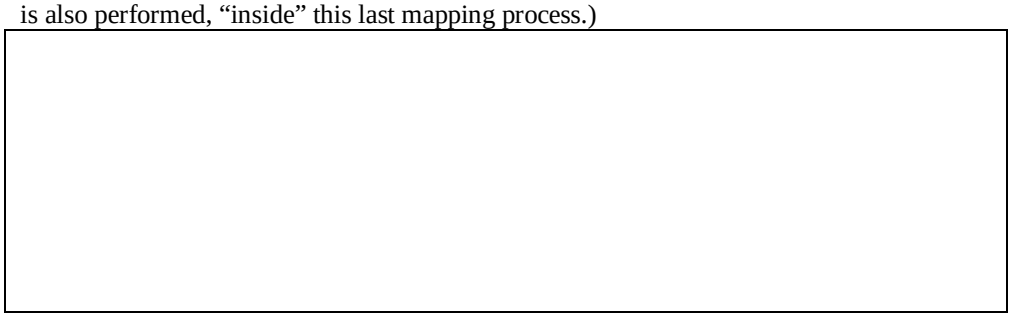

Figure 5.35. The current transformation is applied to vertices.

How do we extend `moveTo`() and `lineTo`() so they quietly carry out this additional mapping? (i.e. how do we rewrite these functions in the Canvas class?) If you are not using OpenGL you must write code that actually performs the transformation; this is described in Case Study 5.1. If you are using OpenGL it is done automatically! OpenGL maintains a so-called **modelview matrix,** and every vertex that is passed down the graphics pipeline is multiplied by it. We need only set up the modelview matrix to embody the desired transformation.

OpenGL works entirely in 3D, so its modelview matrix produces 3D transformations. Here we work with the modelview matrix in a restricted way to perform 2D transformations. Later we use its full power. Figure 5.36 shows how we restrict the 3D transformations to carry out the desired 2D transformations. The main idea is that 2D drawing is done in the *xy*-plane: the *z*-coordinate is understood to be zero. Therefore when we transform 2D points we set the part of the underlying 3D transformation that affects the *z*-coordinate so that it has no effect at all. For example, rotating about the origin in 2D is equivalent to rotating about the *z*-axis in 3D, as shown in the figure. Further, although a scaling in 3D takes three scale factors, *Sx*, *Sy*, and *Sz* to scale in the *x*, *y*, and *z* dimensions, respectively, we set the scale factor *Sz* = 1.

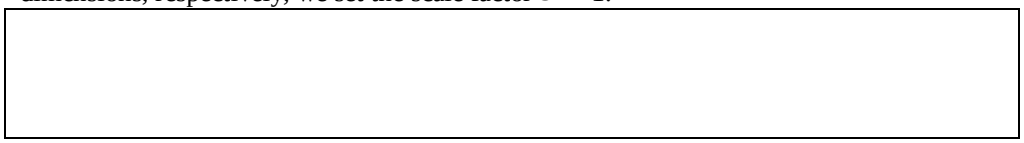

Figure 5.36. 2D drawing takes place in the *xy*-plane.

The principal routines for altering the modelview matrix are `glRotated`()[10], `glScaled`(), and `glTranslated`(). These don't set the *CT* directly; instead each *postmultiplies* the *CT* (the modelview matrix) by a particular matrix, say *M*, and puts the result back into the *CT*. That is, each of these routines creates a matrix *M* as required for the new transformation, and performs:

$$CT = CT * M \tag{5.37}$$

The order is important. As we saw earlier, applying C*T* * *M* to a point is equivalent to first performing the transformation embodied in *M*, followed by performing the transformation dictated by the previous value of *CT*. Or if we are thinking in terms of transforming the coordinate system, it is equivalent to performing one additional transformation to the existing current coordinate system.

OpenGL routines for applying transformations in the 2D case are:

- `glScaled(sx, sy, 1.0);` Postmultiply *CT* by a matrix that performs a scaling by *sx* in *x* and by *sy* in *y*; Put the result back in *CT*. No scaling in *z* is done.

- `glTranslated(dx, dy, 0);` Postmultiply *CT* by a matrix that performs a translation by *dx* in *x* and by *dy* in *y*; Put the result back in *CT*. No translation in *z* is done.

- `glRotated(angle, 0, 0, 1);` Postmultiply *CT* by a matrix that performs a rotation through *angle* degrees about the *z*-axis (indicated by (0, 0, 1))[11]. Put the result back in *CT*.

Since these routines only compose a transformation with the *CT*, we need some way to get started: to initialize the *CT* to the identity transformation. OpenGL provides `glLoadIdentity`(). And because these functions can be set to work on any of the matrices that OpenGL supports, we must inform OpenGL which matrix we are altering. This is accomplished using `glMatrixMode(GL_MODELVIEW)`.

Figure 5.37 shows suitable definitions of four new methods of the *Canvas* class that manage the *CT* and allow us to build up arbitrarily complex 2D transformations. Their pleasing simplicity is possible because OpenGL is doing the hard work.

```
//<<<<<<<<<<<<<< initCT >>>>>>>>>>>>>>>>>
void Canvas:: initCT(void)
{
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();        // set CT to the identity matrix
}
//<<<<<<<<<<<<<< scale2D >>>>>>>>>>>>>>>>>>
void Canvas:: scale2D(double sx, double sy)
{
      glMatrixMode(GL_MODELVIEW);
      glScaled(sx, sy, 1.0); // set CT to CT * (2D scaling)

}
//<<<<<<<<<<<<<< translate2D >>>>>>>>>>>>>>>>>
void Canvas:: translate2D(double dx, double dy)
{
```

[10] The suffix 'd' indicates that its arguments are `doubles`. There is also the version `glRotatef`() that takes `float` arguments.

[11] Here, as always, positive angles produce CCW rotations.

```
      glMatrixMode(GL_MODELVIEW);
      glTranslated(dx, dy, 1.0); // set CT to CT * (2D translation)
}
//<<<<<<<<<<<<<<< rotate2D >>>>>>>>>>>>>>>>>>>
void Canvas:: rotate2D(double angle)
{
      glMatrixMode(GL_MODELVIEW);
      glRotated(angle, 0.0, 0.0, 1.0); // set CT to CT * (2D rotation)
}
```

Figure 5.37. Routines to manage the *CT* for 2D transformations.

We are now in a position to use 2D transformations. Returning to drawing version #2 of the house in Figure 5.34, we next show the code that first rotates the house through -30^0 and then translates it through (32, 25). Notice that, to get the ordering straight, it calls the operations in opposite order to the way they are applied: *first* the translation operation, and *then* the rotation operation.

```
cvs.setWindow(...);
cvs.setViewport(..);     // set the window to viewport mapping
cvs.initCT();            // get started with the identity transformation
house();                 // draw the untransformed house first
cvs.translate2D(32, 25); // CT now includes translation
cvs.rotate2D(-30.0);     // CT now includes translation and rotation
house();                 // draw the transformed house
```

Notice that we can scale, rotate, and position the house in any manner we choose, and never need to "go inside" the routine `house`() or alter it. (In particular, the source code for `house`() need not be available.)

Some people find it more natural to think in terms of transforming the coordinate system. As shown in Figure 5.38 they would think of first translating the coordinate system through (32, 25) to form system #2, and then rotating *that* system through -30^o to obtain coordinate system #3. Because OpenGL applies transformations in the order that coordinate systems are altered, the code for doing it this way first calls `cvs.translate2D(32, 25)` and then calls `cvs.rotate2D(-30.0)`. This is, of course, *identical* to the code obtained doing it the other way, but it has been arrived at through a different thinking process.

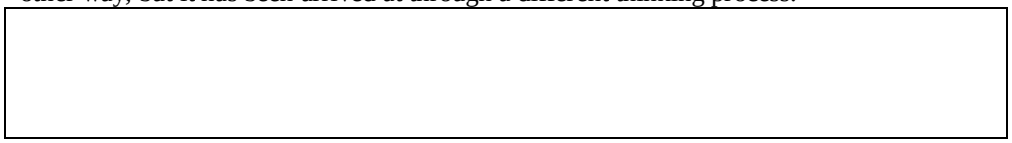

Figure 5.38. The same transformation viewed as a sequence of coordinate system changes.

We give some further examples to show how easily the *CT* is manipulated to produce various effects.

Example 5.5.1. Capitalizing on rotational symmetry.
Figure 5.39a shows a star made of stripes that seem to interlock with one another. This is easy to draw using `rotate2D`(). Suppose that routine `starMotif()` draws a part of the star, the polygon shown in Figure 5.39b. (Determining the positions of this polygon's vertices is challenging, and is addressed in Case Study 5.2.) To draw the whole star we just draw the motif five times, each time rotating the motif through an additional 72°:

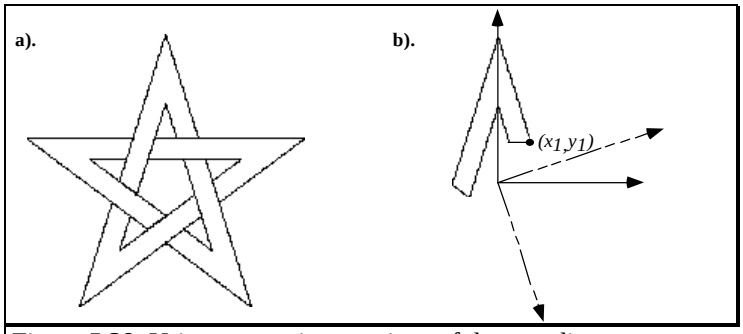

Figure 5.39. Using successive rotations of the coordinate system.

```
for(int count = 0; count < 5; count++)
{
   starMotif();
   cvs.rotate2D(72.0);   // concatenate another rotation
}
```

Visualize what is happening during each of these steps.

Example 5.5.2. Drawing snowflakes.
The beauty of a snowflake arises in good measure from its high degree of symmetry. A snowflake has six identical spokes oriented 60^o apart, and each spoke is symmetrical about its own axis. It is easy to produce a complex snowflake by designing one half of a spoke, and drawing it 12 times. Figure 5.40a shows a snowflake, based on the motif shown in Figure 5.40b. The motif is a polyline that meanders around above the positive *x*-axis. (To avoid any overlap with other parts of the snowflake, the polyline is kept below the 30^0 line shown in the figure.) Suppose the routine `flakeMotif()` draws this polyline.

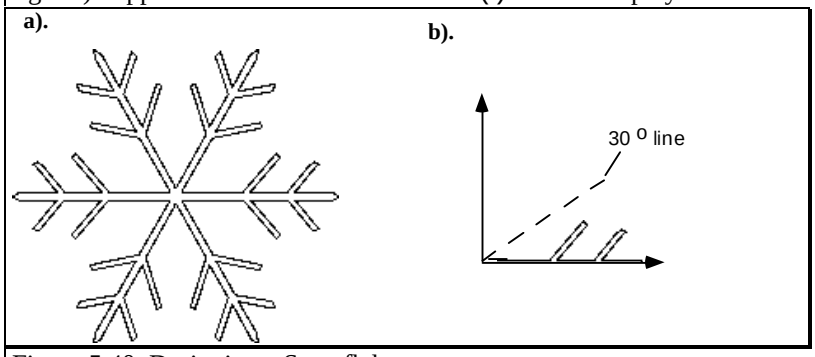

Figure 5.40. Designing a Snowflake.

Each spoke of the snowflake is a combination of the motif and a reflected version. A reflection about the *x*-axis is achieved by the use of `scale2D(1,-1)(why?)`, so the motif plus its reflection can be drawn using

```
flakeMotif();                  // draw the top half
cvs.scale2D(1.0,-1.0);         // flip it vertically
flakeMotif();                  // draw the bottom half
cvs.scale2D(1.0,-1.0);         // restore the original axis
```

To draw the entire snowflake just do this six times, with an intervening rotation of 60^o:

```
void drawFlake()
{
   for(int count = 0; count < 6; count++) // draw a snowflake
   {
      flakeMotif();
      cvs.scale2D(1.0,-1.0);
      flakeMotif();
```

```
      cvs.scale2D(1.0,-1.0);
      cvs.rotate2D(60.0);                  // concatenate a 60 degree rotation
   }
}
```

Example 5.5.3. A Flurry of Snowflakes.
A flurry of snowflakes like that shown in Figure 5.41 can be drawn by drawing the flake repeatedly at random positions, as in:

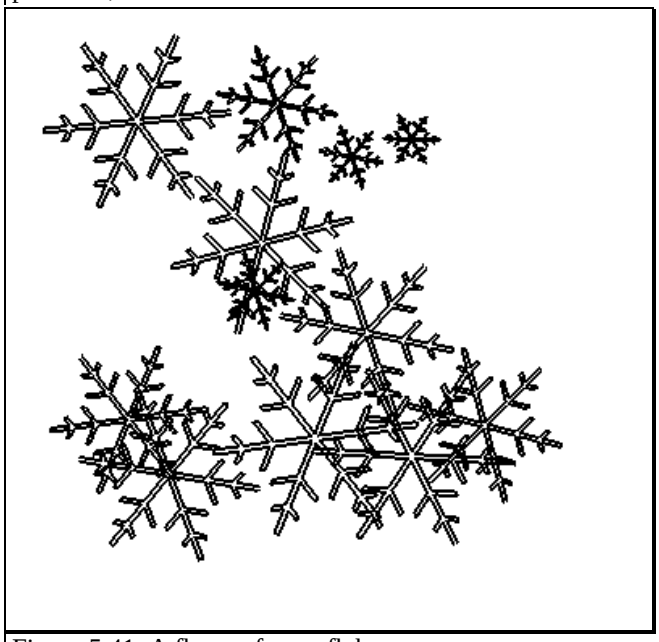

Figure 5.41. A flurry of snowflakes.

```
while(!bored)
{
      cvs.initCT();
      cvs.translate2D(random amount, random amount);
      drawFlake();
}
```

Notice that the *CT* has to be initialized each time, to prevent the translations from accumulating.

Example 5.5.4. Making patterns from a motif.
Figure 5.42 shows two configurations of the dinosaur motif. The dinosaurs are distributed around a circle in both versions, but in one case each dinosaur is rotated so that its feet point toward the origin, and in the other all the dinosaurs are upright. In both cases a combination of rotations and translations is used to create the pattern. It is interesting to see how the ordering of the operations affects the picture.

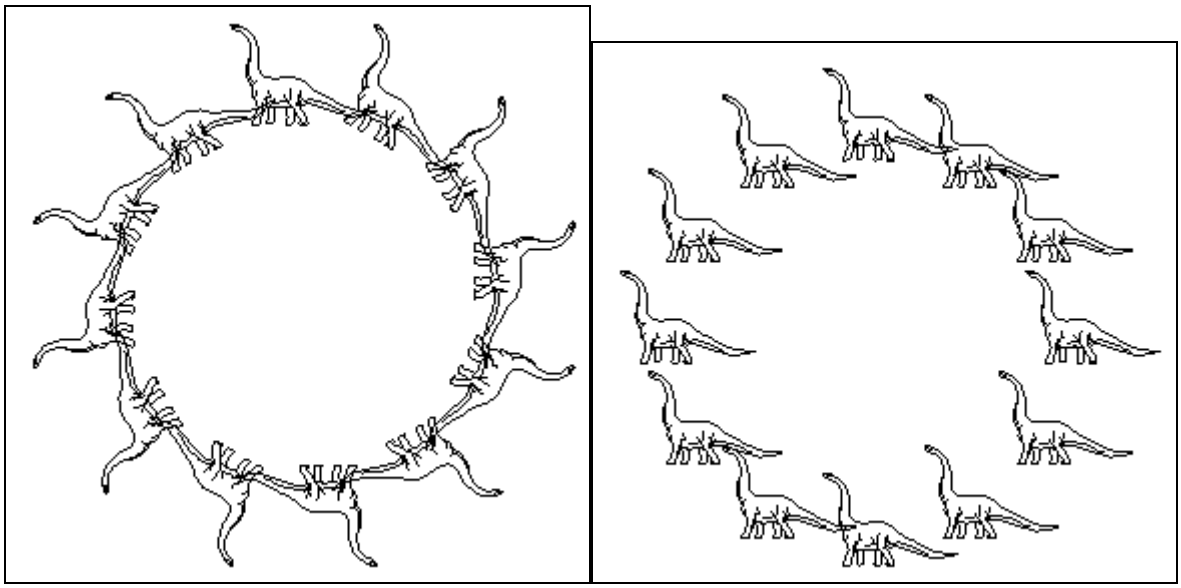
Figure 5.42. Two patterns based on a motif. a). each motif is rotated separately. b). all motifs are upright.

Suppose that `drawDino`() draws an upright dinosaur centered at the origin. In part a) the coordinate system for each motif is first rotated about the origin through a suitable angle, and then this coordinate system is translated along its *y*-axis by *H* units as shown in the following code. Note that the *CT* is reinitialized each time through the loop so that the transformations don't accumulate. (Think through the transformations you would use if instead you took the point of view of transforming points of the motif.)

```
const int numMotifs = 12;
for(int i = 0; i < numMotifs; i++)
{
      cvs.initCT(); // init CT at each iteration
      cvs.rotate2D(i * 360 / numMotifs); // rotate
      cvs.translate2D(0.0, H); // shift along y-axis
      drawDino();
}
```

An easy way to keep the motifs upright as in part b) is to "pre-rotate" each motif before translating it. If a particular motif is to appear finally at 120^o, it is first rotated (while still at the origin) through -120^o, then translated up by *H* units, and then rotated through 120^o. What ajustments to the preceding code will achieve this?

5.5.1. Saving the *CT* for later use.

A program can involve rather lengthy sequences of calls to `rotate2D`(), `scale2D`(), and `translate2D`(). These functions make "additional" or "relative" changes to the *CT*, but sometimes we may need to "back up" to some prior *CT* in order to follow a different path of transformations for the next instance of a picture. In order to "remember" the desired *CT* we make a copy of it and store it in a convenient location. Then at a later point we can restore this matrix as the *CT*, effectively returning to the transformation that was in effect at that point. We may even want to keep a collection of "prior" *CT*'s, and return to selected ones at key moments.

To do this you can work with a **stack of transformations**, as suggested by Figure 5.43. The top matrix on the stack is the actual *CT*, and operations like `rotate2D`() compose their transformation with it in the manner described earlier. To save this *CT* for later use a copy of it is made and "pushed" onto the stack using a routine `pushCT`(). This makes the top two items on the stack identical. The top item can now be altered further with additional calls to `scale2D`() and the like. To return to the previous *CT* the top item is simply "popped" off the stack using `popCT`(), and discarded. This way we can return to the most recent *CT*, the next most recent *CT*, and so forth, in a last-in, first-out order.

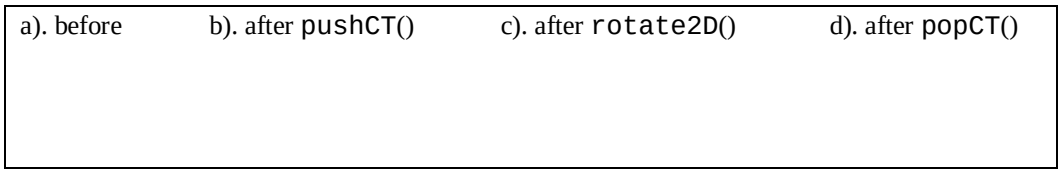

Figure 5.43. Manipulating a stack of *CT*'s.

The implementation of `pushCT()` and `popCT()` is simple when OpenGL is being used, since OpenGL has routines `glPushMatrix`() and `glPopMatrix`() to manage several different stacks of matrices. Figure 5.44 shows the required functions. Note that each of them must inform OpenGL which matrix stack is being affected.

```
void Canvas:: pushCT(void)
{
      glMatrixMode(GL_MODELVIEW);
      glPushMatrix();           // push a copy of the top matrix
}
void Canvas:: popCT(void)
{
      glMatrixMode(GL_MODELVIEW);
      glPopMatrix();            // pop the top matrix from the stack
}
```

Figure 5.44. Routines to save and restore CT's.

Example 5.5.5: Tilings made easy.
Many beautiful designs called **tilings** appear on walls, pottery, and fabric. They are based on the repetition of a basic motif both horizontally and vertically. Consider tiling the window with some motif, as suggested in Figure 5.45. The motif is drawn centered in its own coordinate system as shown in part a) using some routine `drawMotif`(). Copies of the motif are drawn *L* units apart in the x-direction, and *D* units apart in the *y*-direction, as shown in part b).

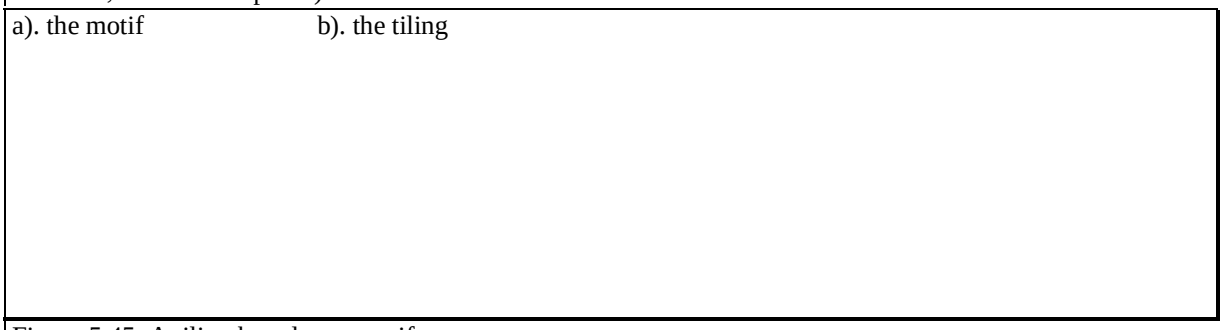

Figure 5.45. A tiling based on a motif.

Figure 5.46 shows how easily the coordinate system can be manipulated in a double loop to draw the tiling. The *CT* is restored after drawing each row, so it returns to the start of that row, ready to move up to start the next row. In addition, the whole block of code is surrounded with a `pushCT`() and a `popCT`(), so that after the tiling has been drawn the *CT* is returned to its initial value, in case more drawing needs to be done.

```
cvs.pushCT();                 // so we can return here
cvs.translate2D(W, H);        // position for the first motif
for(row = 0; row < 3; row++)  // draw each row
{
  cvs.pushCT();
  for(col = 0; col < 4; col++)// draw the next row of motifs
  {
      motif();
      cvs.translate2D(L, 0);  // move to the right
  }
  cvs.popCT();                // back to the start of this row
  cvs.translate2D(0, D);      // move up to the next row
```

```
}
cvs.popCT();                    // back to where we started
```
Figure 5.46. Drawing a hexagonal tiling.

Example 5.5.6. Using modeling transformations in a CAD program. Some programs must draw many instances of a small collection of shapes. Figure 5.47 shows the example of a CAD program that analyzes the behavior of an interconnection of digital logic gates. The user can construct a circuit by "picking and placing" different gates at different places in the work area, possibly with different sizes and orientations. Each picture of the object in the scene is called an **instance** of the object. A single definition of the object is given in a coordinate system that is convenient for that object shape, called its **master coordinate system.** The transformation that carries the object from its own master coordinate system to world coordinates to produce an instance is often called a **modeling transformation.**

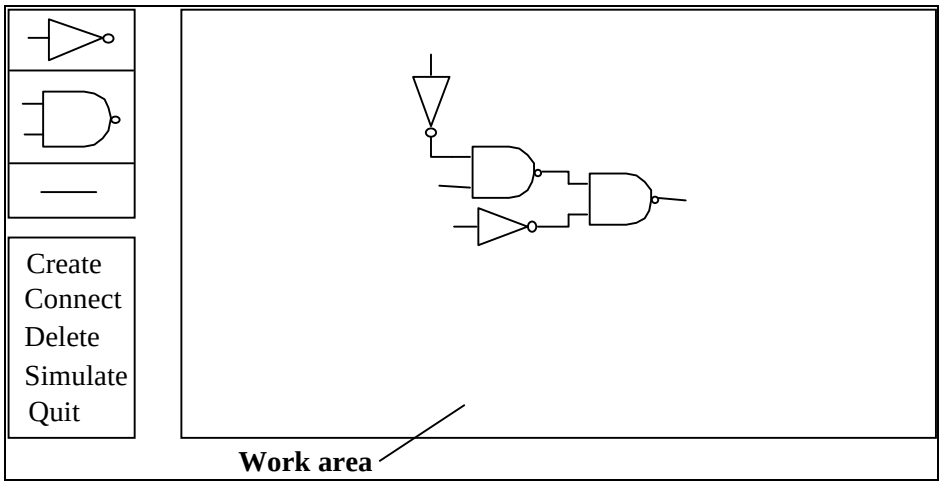

Figure 5.47. Creating instances in a pick-and-place application.

Figure 5.48 shows two logic gates, each defined once in its own master coordinate system. As the user creates each instance of one of these gates, the appropriate modeling transformation is generated that orients and positions the instance. The transformation might be stored simply as a set of parameters, say, *S*, *A*, *dx*, *dy*, with the understanding that the modeling transformation would always consist of:

1. A scaling by factor *S*;
2. A rotation through angle *A*
3. A translation through (*dx, dy*)

performed in that order. A list is kept of the gates in the circuit, along with the transformation parameters of each gate.

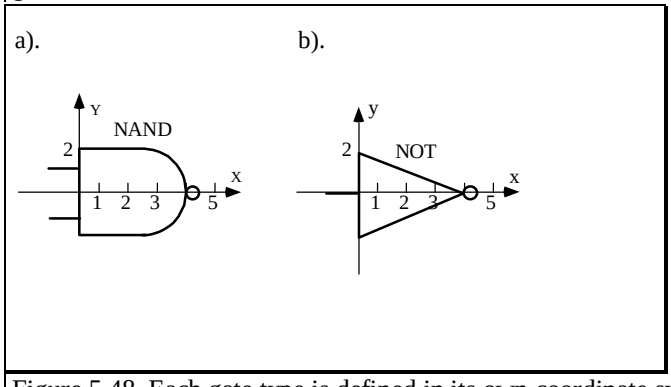

Figure 5.48. Each gate type is defined in its own coordinate system.

Whenever the drawing must be refreshed, each instance is drawn in turn, with the proper modeling transformation applied. Code to do this might look like:

```
clear the screen
for(i = 0; i < numberOfGates; i++) // for each gate
{
      pushCT(); // remember the CT
      translate2D(dx[i], dy[i]); // apply the transformation
      rotate2D(A[i]);
      scale2D( S[i], S[i]);
      drawGate(type[i]); // draw one of the two types
      popCT(); // restore the CT
}
```

The CT is pushed before drawing each instance, so that it can be restored after the instance has been drawn. The modeling transformation for the instance is determined by its parameters, and then one of the two gate shapes is drawn. The necessary code has a simple organization, because the burden of sizing, orienting, and positioning each instance has been passed to the underlying tools that maintain the *CT* and its stack.

Practice Exercises.

5.5.1. Developing the Transformations. Supposing OpenGL were not available, detail how you would write the routines that perform elementary coordinate system changes:

```
void scale2D(double sx, double  sy);
void translate2D(double dx, double  dy);
```

5.5.2. Implementing the transformation stack. Define, in the absence of OpenGL, appropriate data types for a stack of transformations, and write the routines `pushCT()` and `popCT()`.

5.5.3. A hexagonal tiling. A hexagonal pattern provides a rich setting for tilings, since regular hexagons fit together neatly as in a beehive. Figure 5.49 shows 9 columns of stacked 6-gons. Here the hexagons are shown empty, but we could draw interesting figures inside them.

a). Show that the length of a hexagon with radius *R is also R.*

b). Show that the centers of adjacent hexagons in a column are separated vertically by $\sqrt{3}\ R$ and adjacent columns are separated horizontally by 3 *R* / 2.

c). Develop code that draws this hexagonal tiling, using `pushCT`() and `popCT`() and suitable transformations to keep track of where each row and column start.

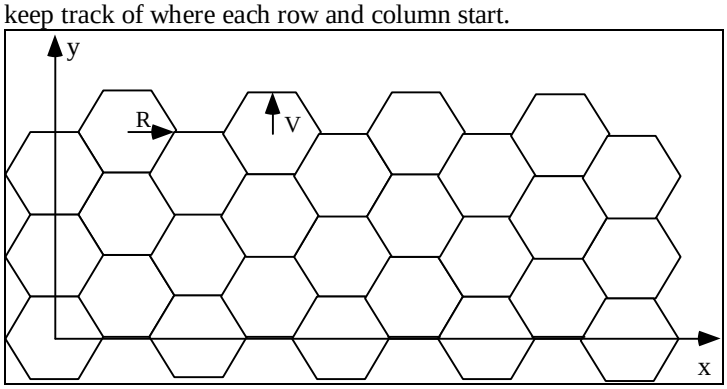

Figure 5.49. A simple hexagonal tiling.

5.6. Drawing 3D Scenes with OpenGL

We introduced the drawing of 2D objects in Chapter 3, developing a simple class *Canvas*. This class provides functions that establish a window and viewport, and that do line drawing through `moveTo`() and `lineTo`(). So far in this chapter we have added the notion of the *CT*, and provided functions that perform 2D rotations, scalings, and translations. These 2D transformations are really just special cases of 3D transformations: they basically ignore the third dimension.

In this section we examine how 3D transformations are used in an OpenGL-based program. The main emphasis is on transforming objects in order to orient and position them as desired in a 3D scene. Not surprisingly it is all done with matrices, and OpenGL provides the necessary functions to build the required matrices. Further, the matrix stack maintained by OpenGL makes it easy to set up a transformation for one object, and then "back up" to a previous transformation, in preparation for transforming another object.

It is very satisfying to build a program that draws different scenes using a collection of 3D transformations. Experimenting with such a program also improves your ability to visualize what the various 3D transformations do. OpenGL makes it easy to set up a "camera" that takes a "snapshot" of the scene from a particular point of view. The camera is created with a matrix as well, and we study in detail in Chapter 7 how this is done. Here we just use an OpenGL tool to set up a reasonable camera, so that attention can be focussed on transforming objects. Granted we are using a tool before seeing exactly how it operates, but the payoff is high: you can make impressive pictures of 3D scenes with a few simple calls to OpenGL functions.

5.6.1. An overview of the viewing process and the graphics pipeline.

All of our 2D drawing so far has actually used a special case of 3D viewing, based on a simple "parallel projection". We have been using the "camera" suggested in Figure 5.50. The "eye" that is viewing the scene looks along the z-axis at the "window", a rectangle lying in the *xy*-plane. The **view volume** of the camera is a rectangular parallelepiped, whose four side walls are determined by the border of the window, and whose other two walls are determined by a **near plane** and a **far plane**. Points lying inside the view volume are projected onto the window along lines parallel to the z-axis. This is equivalent to simply ignoring their z-component, so that the 3D point $(x_1\ y_1,\ z_1)$ projects to $(x_1,\ y_1,\ 0)$. Points lying outside the view volume are clipped off. A separate **viewport transformation** maps the projected points from the window to the viewport on the display device.

Figure 5.50. Simple viewing used in OpenGL for 2D drawing.

Now we move into 3D graphics and place 3D objects in a scene. For the examples here we continue to use a parallel projection. (The more realistic perspective projection, for which more remote objects appear smaller than nearby objects, is described in Chapter 7.) Therefore we use the same camera as in Figure 5.50, but allow it to have a more general position and orientation in the 3D scene, in order to produce better views of the scene.

Figure 5.51 shows such a camera immersed in a scene, The scene consists of block, part of which lies outside the view volume. The image produced by this camera is also shown.

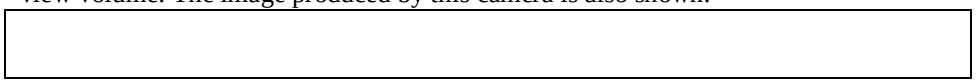

Figure 5.51. A camera to produce parallel views of a scene.

We saw in the previous section that OpenGL provides the three functions `glScaled(..), glRotated(..), and glTranslated(..)` for applying modeling transformations to a shape. The block in Figure 5.51 is in fact a cube that has been stretched, rotated, and shifted as shown. OpenGL also provides functions for defining the view volume and its position in the scene.

The graphics pipeline implemented by OpenGL does its major work through matrix transformations, so we first give insight into what each of the matrices in the pipeline does. At this point it is important only to grasp the basic idea of how each matrix operates: in Chapter 7 we give a detailed discussion. Figure 5.52 shows the pipeline (slightly simplified). Each vertex of an object is passed through this pipeline with a call such as `glVertex3d(x, y, z)`. The vertex is multiplied by the various matrices shown, it is clipped if necessary, and if it survives clipping it is ultimately mapped onto the viewport. Each vertex encounters three matrices:

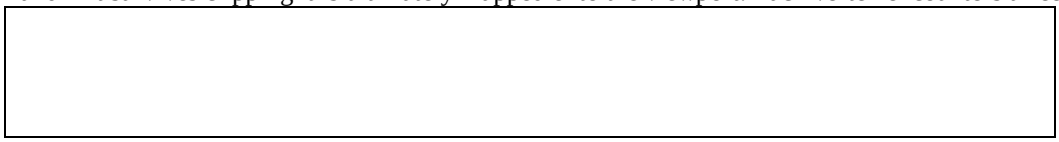

Figure 5.52. The OpenGL pipeline (slightly simplified).

- The **modelview matrix;**
- The **projection matrix;**
- The **viewport matrix;**

The **modelview matrix** basically provides what we have been calling the *CT*. It combines two effects: the sequence of modeling transformations applied to objects, and the transformation that orients and positions the camera in space (hence its peculiar name *modelview*). Although it is a single matrix in the actual pipeline, it is easier to think of it as the product of two matrices, a modeling matrix *M*, and a viewing matrix *V*. The modeling matrix is applied first, and then the viewing matrix, so the modelview matrix is in fact the product *VM* (why?).

Figure 5.53 suggests what the *M* and *V* matrices do, for the situation introduced in Figure 5.51, where a camera "looks down" on a scene consisting of a block. Part a shows a unit cube centered at the origin. A modeling transformation based on *M* scales, rotates, and translates the cube into block shown in part b. Part b also shows the relative position of the camera's view volume.

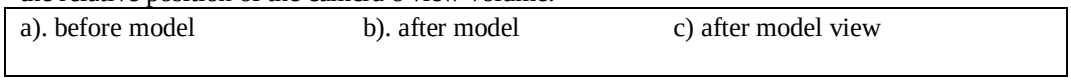

Figure 5.53. Effect of the modelview matrix in the graphics pipeline. a). Before the transformations. b). After the modeling transformation. c). After the modelview transformation.

The *V* matrix is now used to rotate and translate the block into a new position. The specific transformation used is that which would carry the camera from its position in the scene to its "generic" position, with the eye at the origin and the view volume aligned with the *z*-axis, as shown in part c. The vertices of the block are now positioned (that is, their coordinates have the proper values) so that projecting them onto a plane such as the near plane yields the proper values for displaying the projected image. So the matrix *V* in fact effects a change of coordinates of the scene vertices into the camera's coordinate system. (Camera coordinates are sometimes also called **eye coordinates**.)

In the camera coordinate system the edges of the view volume are parallel to the *x*-, *y*-, and *z*-axes. The view volume extends from *left* to *right* in *x*, from *bottom* to *top* in *y*, and from *-near* to *-far* in *z*, as shown. When the vertices of the original cube have passed through the entire modelview matrix, they are located as shown in part c.

The **projection matrix** scales and shifts each vertex in a particular way, so that all those that lie inside the view volume will lie inside a *standard cube* that extends from -1 to 1 in each dimension[12]. (When perspective projections are being used this matrix does quite a bit more, as we see in Chapter 7.) This matrix effectively squashes the view volume into the cube centered at the origin. This cube is a particularly efficient boundary against which to clip objects, as we see in Chapter 7. Scaling the block in this fashion might badly distort it, of course, but this distortion will be compensated for in the viewport transformation. The projection matrix also reverses the sense of the z-axis, so that increasing values of *z* now represent increasing values of depth of a point from the eye. Figure 5.54 shows how the block is transformed into a different block by this transformation.

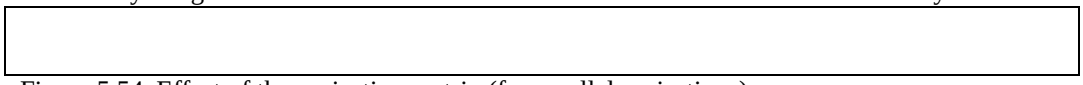

Figure 5.54. Effect of the projection matrix (for parallel projections).

(Notice that the view volume of the camera need never be created as an object itself. It is defined only as that particular shape that the projection matrix converts into the standard cube!)

Clipping is now performed, which eliminates the portion of the block that lies outside the standard cube.

Finally, the **viewport matrix** maps the surviving portion of the block into a "3D viewport". This matrix maps the standard cube into a block shape whose *x* and *y* values extend across the viewport (in screen coordinates), and whose *z*-component extends from 0 to 1 and retains a measure of the depth of point, as shown in Figure 5.55.

[12] Coordinates in this system are sometimes called *Normalized Device Coordinates*.

Figure 5.55. Effect of the viewport transformation.

This overview has outlined how the whole OpenGL graphics pipeline operates, showing that transformations are a central ingredient. Each vertex of an object is subjected to a sequence of transformations that carry it from world coordinates into eye coordinates, into a neutral system specially designed for clipping, and finally into the right coordinate system for proper display. Each transformation is effected by a matrix multiplication.

Aside: Some important details of the pipeline have been suppressed in this first overview. When perspective projections are used we need to include a "perspective division" that capitalizes on a property of homogeneous coordinates. And of course many interesting details lurk in the last step of actually rendering the image, such as computing the color of pixels "in between" vertices, and checking for proper hidden surface removal. These are all addressed in later chapters.

5.6.2. Some OpenGL Tools for Modeling and Viewing.

We now see what functions OpenGL provides for modeling and setting the camera, and how to use them.

1). Three functions are used to set modeling transformations.

The following functions are normally used to modify the modelview matrix, so the modelview matrix is first made "current" by executing: `glMatrixMode(GL_MODELVIEW);`

- `glScaled(sx, sy, sz);` Postmultiply the current matrix by a matrix that performs a scaling by *sx* in *x*, by *sy* in *y*, and by *sz* in *z*; Put the result back in the current matrix.

- `glTranslated(dx, dy, dz);` Postmultiply the current matrix by a matrix that performs a translation by *dx* in *x*, by *dy* in *y*, and by *dz* in *z*; Put the result back in the current matrix.

- `glRotated(angle, ux, uy, uz);` Postmultiply the current matrix by a matrix that performs a rotation through *angle* degrees about the axis that passes through the origin and the point (*ux*, *uy*, *uz*).[13] Put the result back in the current matrix. Equation 5.33 shows the matrix used to perform the rotation.

2). Setting the camera in OpenGL (for a parallel projection).

`glOrtho(left,right,bott,top,near,far)`; Establishes as a view volume a parallelipiped that extends from[14] `left` to `right` in x, from `bott` to `top` in y, and from `-near` to `-far` in *z*. (Since this definition operates in eye coordinates, the camera's eye is at the origin, looking down the negative *z*-axis.) This function creates a matrix and postmultiplies the current matrix by it. (We show in Chapter 7 precisely what values this matrix contains.)

Thus to set the projection matrix use:

```
glMatrixMode(GL_PROJECTION);        // make the projection matrix current
glLoadIdentity();                        // set it to the identity matrix
glOrtho(left,right,bottom,top,near,far); // multiply it by the new matrix
```

Notice the minus signs above: because the default camera is located at the origin looking down the negative *z*-axis, using a value of 2 for `near` means to place the near plane at *z* = -2, that is, 2 units in front of the eye. Similarly, using 20 for `far` places the far plane 20 units in front of the eye.

3). Positioning and aiming the camera.

---

[13] Positive values of *angle* produce rotations that are CCW as one looks along the axis from the point (*ux*, *uy*, *uz*) towards the origin.

[14] All parameters of `glOrtho`() are of type `GLdouble`.

OpenGL offers a function that makes it easy to set up a basic camera:

`gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z);`
Creates the view matrix and postmultiplies the current matrix by it.

It takes as parameters the eye position, `eye`, of the camera and the look-at point, `look`. It also takes an approximate "up" direction, `up`. Since the programmer knows where an interesting part of the scene is situated, it is usually straightforward to choose reasonable values for `eye` and `lookAt` for a good first view. And `up` is most often set to (0, 1, 0) to suggest an "up" direction parallel to the *y*-axis. Later we develop more powerful tools for setting up a camera and for interactively "flying it" in an animation.

We want this function to set the *V* part of the modelview matrix *VM*. So it is invoked before any modeling transformations are added, since subsequent modeling transformations will postmultiply the modelview matrix. So to use `gluLookAt`() follow the sequence:

```
glMatrixMode(GL_MODELVIEW);          // make the modelview matrix current
glLoadIdentity();                    // start with a unit matrix
gluLookAt(eye.x, eye.y, eye.z,           // the eye position
          look.x, look.y, look.z,    // the "look at" point
          up.x, up.y, up.z)              // the up direction
```

We discuss how this function operates in Chapter 7, and also develop more flexible tools for establishing the camera. For those curious about what values `gluLookAt`() actually places in the modelview matrix, see the exercises.

Example 5.6.1: Set up a typical camera. Cameras are often set to "look down" on the scene from some nearby position. Figure 5.56 shows the camera with its eye situated at `eye` = (4,4,4) looking at the origin with `lookAt` = (0,1,0). The `up` direction is set to `up` = (0, 1, 0). Suppose we also want the view volume to have a width of 6.4, a height of 4.8 (so its aspect ratio is 640/480), and to set `near` to 1 and `far` to 50. This camera would be established using:

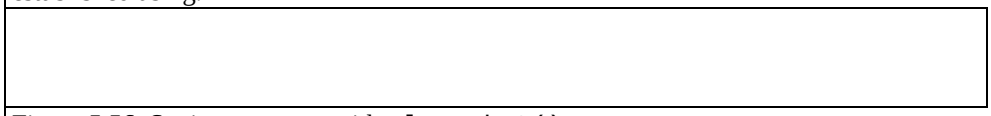

Figure 5.56. Setting a camera with `gluLookAt()`.

```
glMatixMode(GL_PROJECTION); // set the view volume
glLoadIdentity();
glOrtho(-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode(GL_MODELVIEW); // place and aim the camera
glLoadIdentity();
gluLookAt(2, 4, 5, 0, 0, 0, 0, 1, 0);
```
The exercises show the specific values that get placed in the modelview matrix.

Practice Exercises.
5.6.1. What does `gluLookAt()` do? We know that `gluLookAt`() builds a matrix that converts world coordinates into eye coordinates. Figure 5.57 shows the camera as a coordinate system suspended in the world, with its origin at *eye*, and oriented according to its three mutually perpendicular unit vectors **u**, **v**, and **n**. The eye is "looking" in the direction **-n**. `gluLookAt`() uses the parameters *eye*, *look*, and **up** to create **u**, **v**, and **n** according to

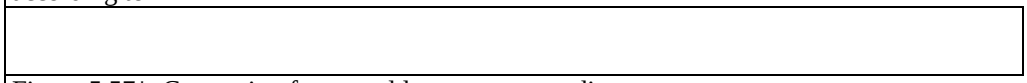

Figure 5.57*. Converting from world to camera coordinates.

$$\mathbf{n} = eye - look$$
$$\mathbf{u} = \mathbf{up} \times \mathbf{n} \qquad (5.38)$$
$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$

and then normalizes all three of these to unit length. It builds the matrix

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the point *d* has components $(d_x, d_y, d_z) = (-eye \cdot \mathbf{u}, -eye \cdot \mathbf{v}, -eye \cdot \mathbf{n})$.

a). Show that **u**, **v**, and **n** are mutually perpendicular.
b). Show that matrix *V* properly converts world coordinates to eye coordinate by showing that it maps *eye* into the origin $(0,0,0,1)^T$, **u** into **i** = (1,0,0,0), **v** into **j** = (0,1,0), and **n** into **k** = (0,0,1).
c). Show for the case *eye* = (4, 4, 4), *lookAt* = (0,1,0), and **up** = (0,1,0) that the resulting matrix is:

$$V = \begin{pmatrix} .70711 & 0 & -.70711 & 0 \\ -.3313 & .88345 & -.3313 & -.88345 \\ .6247 & .4685 & .6247 & -6.872 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5.6.2. Inquiring of the values in a matrix in OpenGL. Print out the values in the modelview matrix to test some of the assertions made about how it is formed. To see what is stored in the modelview matrix in OpenGL define an array `GLfloat mat[16]` and use `glGetFloatv(GL_MODELVIEW_MATRIX,mat)` which copies into `mat[]` the 16 values in the modelview matrix. `M[i][j]` is copied into the element `mat[4j+i]`, for *i, j* = 0, 1,..,3.

5.6.3. Drawing Elementary Shapes Provided by OpenGL

We see in the next chapter how to create our own 3D objects, but at this point we need some 3D objects to draw, in order to experiment with setting and using cameras. The GLUT provides several ready-made 3D objects. These include a sphere, a cone, a torus, the five Platonic solids (discussed in Chapter 6), and the famous teapot. Each is available as a "wireframe" model and as a "solid" model with faces that can be shaded.

- **cube:** `glutWireCube(GLdouble size);` Each side is of length `size`
- **sphere:** `glutWireSphere(GLdouble radius, GLint nSlices, GLint nStacks)`
- **torus:** `glutWireTorus(GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks)`
- **teapot:** `glutWireTeapot(GLdouble size)`

There is also a `glutSolidCube`(), `glutSolidSphere`(), etc., that we use later. The shape of the torus is determined by i inner radius `inRad` and outer radius `outRad`. The sphere and torus are approximated by polygonal faces, and you can adjus parameters `nSlices` and `nStacks` to specify how many faces to use in the approximation. `nSlices` is the number of subdivisions around the *z*-axis, and `nStacks` is the number of 'bands" along the *z*-axis, as if the shape were a stack of `nSta` disks.

Four of the Platonic solids (the fifth is the cube, already presented):

- **tetrahedron:** `glutWireTetrahedron()`
- **octahedron:** `glutWireOctahedron()`
- **dodecahedron:** `glutWireDodecahedron()`
- **icosahedron:** `glutWireIcosahedron()`

All of the shapes above are centered at the origin.

- **cone:** `glutWireCone(GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStac`

- **tapered cylinder:** `gluCylinder(GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, GLint nSlices, GLint nStacks)`

The axes of the cone and tapered cylinder coincide with the *z*-axis. Their bases rest on the $z = 0$ plane, and they extend to $z =$ `height` along the *z*-axis. The radius of the cone and tapered cylinder at $z = 0$ is given by `baseRad`. The radius of the tapere cylinder at $z =$ `height` is `topRad`.

The **tapered cylinder** is actually a *family* of shapes, distinguished by the value of `topRad`. When `topRad` is 1 there is no ta this is the classic **cylinder**. When `topRad` is 0 the tapered cylinder is identical to the **cone**.

Note that drawing the tapered cylinder in OpenGL requires some extra work, because it is a special case of a quadric surface, we shall see in Chapter 6. To draw it you must 1) define a new quadric object, 2). set the drawing style (`GLU_LINE` for a wireframe, `GLU_FILL` for a solid rendering), and 3). draw the object:

```
GLUquadricObj * qobj = gluNewQuadric();               // make a quadric object
gluQuadricDrawStyle(qobj,GLU_LINE);                   // set style to wireframe
gluCylinder(qobj, baseRad, topRad, nSlices, nStacks); // draw the cylinder
```

Figure 5.58. Shapes available in the GLUT.

We next employ some of these shapes in two substantial examples that focus on using affine transformations to model and view a 3D scene. The complete program to draw each scene is given. A great deal of insight can be obtained if you enter these programs and produce the figures, and then see the effect of varying the various parameters.

Example 5.6.2: A scene composed of wireframe objects.
Figure 5.59 shows a scene with several objects disposed at the corners of a unit cube. The cube has one corner at the origin. Seven objects appear at various corners of the cube, all drawn as wireframes.

The camera is given a view volume that extends from -2 to 2 in *y*, with an aspect ratio of aspect = 640/480. Its near plane is at $N = 0.1$, and its far plane is at $F = 100$. This is accomplished using:

```
glOrtho(-2.0* aspect, 2.0* aspect, -2.0, 2.0, 0.1, 100);
```

The camera is positioned with *eye* = (2, 2.2, 3), *lookAt* = (0, 0, 0), and *up* = (0, 1, 0) (parallel to the *y*-axis), using:

```
gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

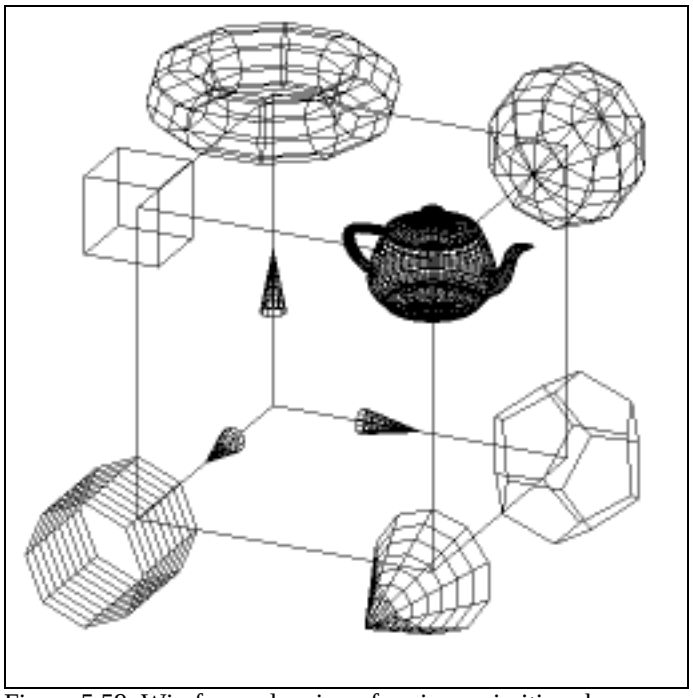

Figure 5.59. Wireframe drawing of various primitive shapes.

Figure 5.60 shows the complete program to produce the drawing. The `main`() routine initializes a 640 by 480 pixel screen window, sets the viewport and background color, and specifies `displayWire`() as the display function to be called to perform the drawing. In `displayWire`() the camera shape and position are established first. Then each object is drawn in turn. Most objects need their own modeling matrix, in order to rotate and position them as desired. Before each modeling transformation is established, a `glPushMatrix`() is used to member the current transformation, and after the object has bee drawn this prior current transformation is restored with a `glPopMatrix`(). Thus the code to draw each object is imbedded i a `glPushMatrix`(), `glPopMatrix`() pair. Check each transformation carefully to see that it places its object at the proper place.

Also shown in the figure are the *x*-, *y*-, and *z*- axes drawn with conical arrow heads. Displaying the underlying coordinate system can help to orient the viewer. To draw the *x*-axis, the *z*-axis is rotated 90° about the *y*-axis to form a rotated system, an the axis is redrawn in its new orientation. Note that this axis is drawn without immersing it in a `glPushMatrix`(), `glPopMatrix`() pair, so the next rotation to produce the *y*-axis takes place in the already rotated coordinate system. Check that it's the proper rotation.

```
#include <windows.h>  //suitable when using Windows 95/98/NT
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//<<<<<<<<<<<<<<<<<< axis >>>>>>>>>>>>>>
void axis(double length)
{ // draw a z-axis, with cone at end
      glPushMatrix();
      glBegin(GL_LINES);
         glVertex3d(0, 0, 0); glVertex3d(0,0,length); // along the z-axis
      glEnd();
      glTranslated(0, 0,length -0.2);
      glutWireCone(0.04, 0.2, 12, 9);
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<< displayWire >>>>>>>>>>>>>>>>>>>>>>
void displayWire(void)
{
      glMatrixMode(GL_PROJECTION); // set the view volume shape
      glLoadIdentity();
```

```
      glOrtho(-2.0*64/48.0, 2.0*64/48.0, -2.0, 2.0, 0.1, 100);
      glMatrixMode(GL_MODELVIEW); // position and aim the camera
      glLoadIdentity();
      gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

      glClear(GL_COLOR_BUFFER_BIT); // clear the screen
      glColor3d(0,0,0: // draw black lines
      axis(0.5);                          // z-axis
      glPushMatrix();
      glRotated(90, 0,1.0, 0);
      axis(0.5);                         // y-axis
      glRotated(-90.0, 1, 0, 0);
      axis(0.5);                         // z-axis
      glPopMatrix();

      glPushMatrix();
      glTranslated(0.5, 0.5, 0.5); // big cube at (0.5, 0.5, 0.5)
      glutWireCube(1.0);
      glPopMatrix();

      glPushMatrix();
      glTranslated(1.0,1.0,0);      // sphere at (1,1,0)
      glutWireSphere(0.25, 10, 8);
      glPopMatrix();

      glPushMatrix();
      glTranslated(1.0,0,1.0);      // cone at (1,0,1)
      glutWireCone(0.2, 0.5, 10, 8);
      glPopMatrix();

      glPushMatrix();
      glTranslated(1,1,1);
      glutWireTeapot(0.2); // teapot at (1,1,1)
      glPopMatrix();

      glPushMatrix();
      glTranslated(0, 1.0 ,0); // torus at (0,1,0)
      glRotated(90.0, 1,0,0);
      glutWireTorus(0.1, 0.3, 10,10);
      glPopMatrix();

      glPushMatrix();
      glTranslated(1.0, 0 ,0); // dodecahedron at (1,0,0)
      glScaled(0.15, 0.15, 0.15);
      glutWireDodecahedron();
      glPopMatrix();

      glPushMatrix();
      glTranslated(0, 1.0 ,1.0); // small cube at (0,1,1)
      glutWireCube(0.25);
      glPopMatrix();

      glPushMatrix();
      glTranslated(0, 0 ,1.0); // cylinder at (0,0,1)
      GLUquadricObj * qobj;
      qobj = gluNewQuadric();
      gluQuadricDrawStyle(qobj,GLU_LINE);
      gluCylinder(qobj, 0.2, 0.2, 0.4, 8,8);
      glPopMatrix();
      glFlush();
}
//<<<<<<<<<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
```

```
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
      glutInitWindowSize(640,480);
      glutInitWindowPosition(100, 100);
      glutCreateWindow("Transformation testbed - wireframes");
      glutDisplayFunc(displayWire);
      glClearColor(1.0f, 1.0f, 1.0f,0.0f);  // background is white
      glViewport(0, 0, 640, 480);
      glutMainLoop();
}
```

Figure 5.60. Complete program to draw Figure 5.65 using OpenGL.

Notice that the sides of the large cube that are parallel in 3D are also displayed as parallel. This is a result of using a parallel projection. The cube looks slightly unnatural because we are used to seeing the world with a perspective projection. As we see in Chapter 7, if a perspective projection were used instead, these parallel edges would not be drawn parallel.

Example 5.6.3. A 3D scene rendered with shading.
We develop a somewhat more complex scene to illustrate further the use of modeling transformations. We also show how easy OpenGL makes it to draw much more realistic drawings of solid objects by incorporating shading, along with proper hidden surface removal.

Two views of a scene are shown in Figure 5.61. Both views use a camera set by `gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0,1.0,0.0)`. Part a uses a large view volume that encompasses the whole scene; part b uses a small view volume that encompasses only a small portion of the scene, thereby providing a close-up view.

The scene contains three objects resting on a table in the corner of a "room". Each of the three walls is made by flattening a cube into a thin sheet, and moving it into position. (Again, they look somewhat unnatural due to the use of a parallel projection.) The "jack" is composed of three stretched spheres oriented at right angles plus six small spheres at their ends.

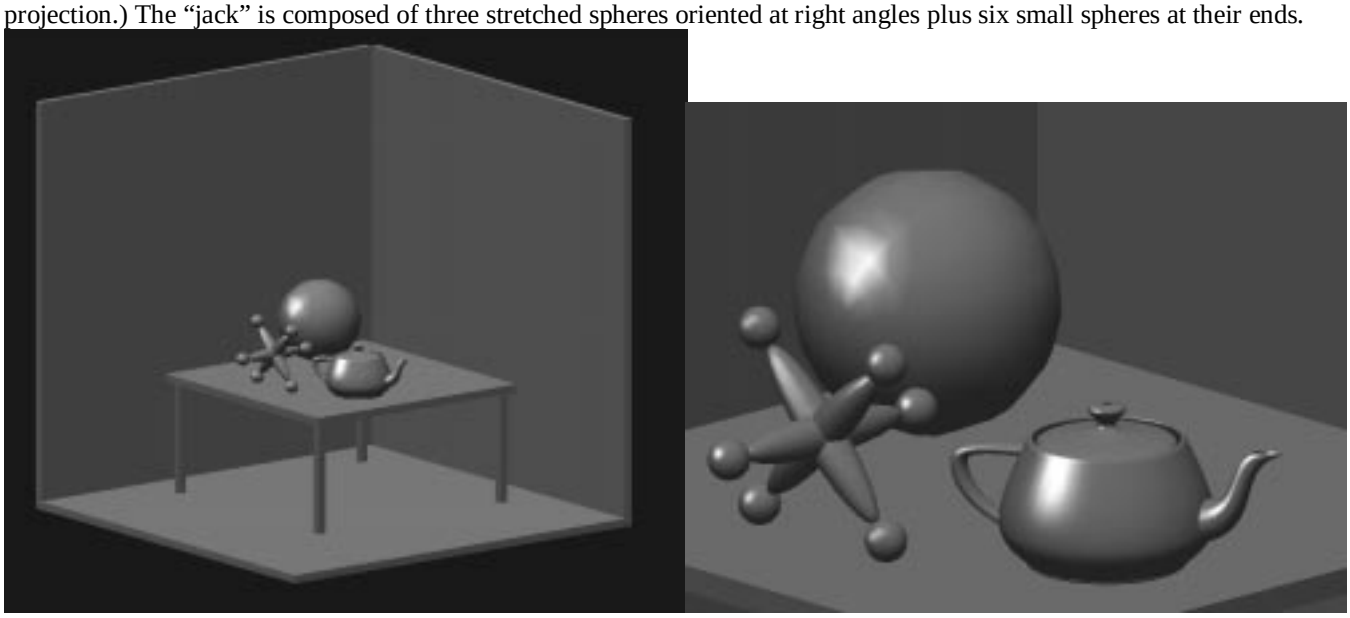

Figure 5.61. A simple 3D scene - a). Using a large view volume. b). Using a small view volume.

The table consists of a table top and four legs. Each of the table's five pieces is a cube that has been scaled to the desired size and shape. The layout for the table is shown in Figure 5.62. It is based on four parameters that characterize the size of its parts: `topWidth`, `topThick`, `legLen`, and `legThick`. A routine `tableLeg`() draws each leg, and is called four times within the routine `table`() to draw the legs in the four different locations. The different parameters used produce different modeling transformations within `tableLeg`(). As always, a `glPushMatrix`(), `glPopMatrix`() pair surrounds the modeling functions to isolate their effect.

a). b).

Figure 5.62. Designing the table.

The complete code for this program is shown in Figure 5.63. Note that the "solid" version of each shape, such as `glutSolidSphere`(), is used here, rather than the "wire" version. Check the assorted transformations used to orient and position each object in the scene. Check the jack model particularly. This example is designed to whet your appetite for trying out scenes on your own, practicing with transformations.

The code also shows the various things that must be done to create shaded images. The position and properties of a light source must be specified, along with certain properties of the objects' surfaces, in order to describe how they reflect light. Since we discuss the shading process in Chapter 8 we just present the various function calls here; using them as shown will generate shading.

```
#include <windows.h>
#include <iostream.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//<<<<<<<<<<<<<< wall >>>>>>>>>>>>>>>
void wall(double thickness)
{ // draw thin wall with top = xz-plane, corner at origin
      glPushMatrix();
      glTranslated(0.5, 0.5 * thickness,  0.5);
      glScaled(1.0, thickness, 1.0);
      glutSolidCube(1.0);
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<< tableLeg >>>>>>>>>>>>>>>>>>
void tableLeg(double thick, double len)
{
      glPushMatrix();
      glTranslated(0, len/2, 0);
      glScaled(thick, len, thick);
      glutSolidCube(1.0);
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<< jack part >>>>>>>>>>>>>
void jackPart()
{ // draw one axis of the unit jack - a stretched sphere
      glPushMatrix();
      glScaled(0.2,0.2,1.0);
      glutSolidSphere(1,15,15);
      glPopMatrix();
      glPushMatrix();
      glTranslated(0,0,1.2); // ball on one end
      glutSolidSphere(0.2,15,15);
      glTranslated(0,0, -2.4);
      glutSolidSphere(0.2,15,15); // ball on the other end
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<<< jack >>>>>>>>>>>>>>>>>>>
void jack()
{ // draw a unit jack out of spheroids
      glPushMatrix();
      jackPart();
      glRotated(90.0, 0, 1, 0);
      jackPart();
      glRotated(90.0, 1,0,0);
      jackPart();
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<< table >>>>>>>>>>>>>>>>>>>>
```

```
void table(double topWid, double topThick, double legThick, double legLen)
{ // draw the table - a top and four legs
      glPushMatrix(); // draw the table top
      glTranslated(0, legLen, 0);
      glScaled(topWid, topThick, topWid);
      glutSolidCube(1.0);
      glPopMatrix();
      double dist = 0.95 * topWid/2.0 - legThick / 2.0;
      glPushMatrix();
      glTranslated(dist, 0, dist);
      tableLeg(legThick, legLen);
      glTranslated(0, 0, -2 * dist);
      tableLeg(legThick, legLen);
      glTranslated(-2 * dist, 0, 2*dist);
      tableLeg(legThick, legLen);
      glTranslated(0, 0, -2*dist);
      tableLeg(legThick, legLen);
      glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<< displaySolid >>>>>>>>>>>>>>>>>>>>
void displaySolid(void)
{
      //set properties of the surface material
      GLfloat mat_ambient[] = { 0.7f, 0.7f, 0.7f, 1.0f}; // gray
      GLfloat mat_diffuse[] = {0.6f, 0.6f, 0.6f, 1.0f};
      GLfloat mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
      GLfloat mat_shininess[] = {50.0f};
      glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
      glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
      // set the light source properties
      GLfloat lightIntensity[] = {0.7f, 0.7f, 0.7f, 1.0f};
      GLfloat light_position[] = {2.0f, 6.0f, 3.0f, 0.0f};
      glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, lightIntensity);
      // set the camera
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      double winHt = 1.0; // half-height of the window
      glOrtho(-winHt*64/48.0, winHt*64/48.0, -winHt, winHt, 0.1, 100.0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0,1.0,0.0);

      // start drawing
      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // clear the screen
      glPushMatrix();
      glTranslated(0.4, 0.4, 0.6);
      glRotated(45,0,0,1);
      glScaled(0.08, 0.08, 0.08);
      jack();                        // draw the jack
      glPopMatrix();
      glPushMatrix();
      glTranslated(0.6, 0.38, 0.5);
      glRotated(30,0,1,0);
      glutSolidTeapot(0.08);         // draw the teapot
      glPopMatrix();
      glPushMatrix();
      glTranslated(0.25, 0.42, 0.35);// draw the sphere
      glutSolidSphere(0.1, 15, 15);
      glPopMatrix();
      glPushMatrix();
      glTranslated(0.4, 0, 0.4);
```

```
      table(0.6, 0.02, 0.02, 0.3); // draw the table
      glPopMatrix();
      wall(0.02);                  // wall #1: in xz-plane
      glPushMatrix();
      glRotated(90.0, 0.0, 0.0, 1.0);
      wall(0.02);                  // wall #2: in yz-plane
      glPopMatrix();
      glPushMatrix();
      glRotated(-90.0,1.0, 0.0, 0.0);
      wall(0.02);                  // wall #3: in xy-plane
      glPopMatrix();
      glFlush();
}
//<<<<<<<<<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB| GLUT_DEPTH);
      glutInitWindowSize(640,480);
      glutInitWindowPosition(100, 100);
      glutCreateWindow("shaded example - 3D scene");
      glutDisplayFunc(displaySolid);
      glEnable(GL_LIGHTING); // enable the light source
      glEnable(GL_LIGHT0);
      glShadeModel(GL_SMOOTH);
      glEnable(GL_DEPTH_TEST); // for hidden surface removal
      glEnable(GL_NORMALIZE); // normalize vectors for proper shading
      glClearColor(0.1f,0.1f,0.1f,0.0f);  // background is light gray
      glViewport(0, 0, 640, 480);
      glutMainLoop();
}
```

Figure 5.63. Complete program to draw the shaded scene.

Practice Exercises.

5.6.3. Inquiring of the values in a matrix in OpenGL. Test some of the assertions above that put specific values into the modelview matrix. You can see what is stored in the modelview matrix in OpenGL by defining an array `GLfloat mat[16]` and using `glGetFloatv(GL_MODELVIEW_MATRIX,mat)` which copies into `mat[]` the 16 values in the modelview matrix. `M[i][j]` is copied into the element `mat[4j+i]`, for i, j = 0, 1,..,3.

5.6.4. Reading a Scene Description from a File.

In the previous examples the scene was described through specific OpenGL calls that transform and draw each object, as in

```
glTranslated(0.25, 0.42, 0.35);
glutSolidSphere(0.1, 15, 15); // draw a sphere
```

The objects in the scene were therefore "hard-wired" into the program. This way of specifying a scene is cumbersome and error-prone. It is a boon when the designer can specify the objects in a scene through a simple language, and place the description in a file. The drawing program becomes a (much simpler) general-purpose program: it reads a scene file at run-time and draws whatever objects are encountered in the file.

The Scene Description Language (SDL) described in Appendix 4 provides such a tool. We define a `Scene` class – also described in Appendix 4, and available on the book's web site - that supports the reading of an SDL file and the drawing of the objects described in the file. It is very simple to use the `Scene` class in an application: a global `Scene` object is created:

```
Scene scn; // create a scene object
```

and the `read`() method of the class is called to read in a scene file:

```
scn.read("simple.dat");  // read the scene file & build an object list
```

Figure 5.64 shows the data structure for the `scn` object, after the following simple SDL file has been read.

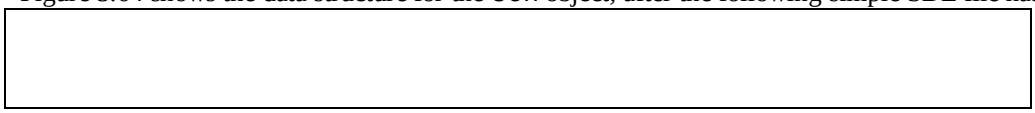

Figure 5.64. An object of the `Scene` class.

```
! simple.dat: a simple scene having one light and four shapes
background 0 0 1        ! give the scene a blue background
light 2 9 8 1 1 1       ! put a white light at (9, 9, 9)
diffuse .9 .1 .1        ! make the following objects reddish
translate 3 5 –2  sphere      ! put a sphere at 3 5 –2
translate –4 –6 8  cone ! put a cone in the scene
translate  1 1 1 cube   ! add a cube
diffuse 0 1 0           ! make the following objects green
translate 40 5 2 scale .2 .2 .2 sphere !add a tiny sphere
```

The first line is a comment; comments extend to the end of the line. This scene has a bright blue background color (*red*, *green*, *blue*) = (0, 0, 1), a bright white (1, 1, 1) light situated at (2, 9, 8), and four objects: two spheres, a cone and a cube. The `light` field points to the list of light sources, and the `obj` field points to the object list. Each shape object has its own affine transformation *M* that describes how it is scaled, rotated, and positioned in the scene. It also contains various data fields that specify its material properties, that are important when we want to render it faithfully as discussed in Chapter 8. Only the diffuse field is shown in the figure.

Once the light list and object list have been built, the application can render the scene:

```
scn.makeLightsOpenGL(),
scn.drawSceneOpenGL();  // render the scene using OpenGL
```

The first instruction passes a description of the light sources to OpenGL. The second uses the method `drawSceneOpenGL`() to draw each object in the object list. The code for this method is very simple:

```
void Scene :: drawSceneOpenGL()
{
      for(GeomObj* p = obj; p ; p = p->next)
            p->drawOpenGL(); // draw it
}
```

It moves a pointer through the object list, calling `drawOpenGL`() for each object in turn. It's a nice example of using polymorphism which is a foundation-stone of object-oriented programming: Each different shape "knows" how to draw itself: it has a method `drawOpenGL`() that calls the appropriate routine for that shape. So when `p` points to a sphere the `drawOpenGL`() routine for a sphere is called automatically; when `p` points to a cone the `drawOpenGL`() routines for a cone is called, etc. Figure 5.65 shows the methods for the `Sphere` and `Cone` classes; they differ only in the final OpenGL drawing routine that is called. Each first passes the object's material properties to OpenGL, then updates the modelview matrix with the object's specific affine transformation. The original modelview matrix is pushed and later restored, to protect it from being affected after this object has been drawn.

```
void Sphere :: drawOpenGL()
{
  tellMaterialsGL();  //pass material data to OpenGL
  glPushMatrix();
  glMultMatrixf(transf.m); // load this object's matrix
  glutSolidSphere(1.0,10,12); // draw a sphere
  glPopMatrix();
}
void Cone :: drawOpenGL()
```

```
{
  tellMaterialsGL();//pass material data to OpenGL
  glPushMatrix();
  glMultMatrixf(transf.m); // load this object's matrix
  glutSolidCone(1.0,1.0, 10,12); // draw a cone
  glPopMatrix();
}
```

Figure 5.65. the `drawOpenGL()` methods for two shapes.

Figure 5.66 shows the program that reads an SDL file and draws the scene. It is very short, (but of course the code for the classes `Scene`, `Shape`, etc. must be loaded as well). It reads the particular SDL file `myScene1.dat`, which recreates the same scene as in Figure 5.63. Note that by simply changing the SDL file that is read this program can draw *any* scene described in SDL, without any changes in its code.

```
#include "basicStuff.h"
#include "Scene.h"
//############################ GLOBALS ########################
Scene scn;  // construct the scene object
//<<<<<<<<<<<<<<<<<<<<<<< displaySDL >>>>>>>>>>>>>>>>>>>>>>>
void displaySDL(void)
{
      glMatrixMode(GL_PROJECTION); //set the camera
      glLoadIdentity();
      double winHt = 1.0; // half-height of the window
      glOrtho(-winHt*64/48.0, winHt*64/48.0, -winHt, winHt, 0.1, 100.0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0,1.0,0.0);
      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);     // clear screen
      scn.drawSceneOpenGL();
} // end of display
//<<<<<<<<<<<<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGB |GLUT_DEPTH);
      glutInitWindowSize(640, 480);
      glutInitWindowPosition(100, 100);
      glutCreateWindow("read and draw an SDL scene");
      glutDisplayFunc(displaySDL);
      glShadeModel(GL_SMOOTH);
      glEnable(GL_DEPTH_TEST);
      glEnable(GL_NORMALIZE);
      glViewport(0, 0, 640, 480);
      scn.read("myScene1.dat"); //read the SDL file and build the objects
      glEnable(GL_LIGHTING);
      scn.makeLightsOpenGL(); // scan the light list and make OpenGL lights
      glutMainLoop();
}
```

Figure 5.66. Drawing a scene read in from an SDL file.

The SDL file that describes the scene of Figure 5.63 is shown in Figure 5.67. It defines the jack shape of nine spheres by first defining a `jackPart` and then using it three times, as explained in Appendix 4. Similarly a leg of the table is first defined as a unit, and then used four times.

```
! – myScene1.dat
light 20 60 30 .7 .7 .7 !put a light at (20,60,30),color:(.7, .7, .7)
ambient .7 .7 .7 ! set material properties for all of the objects
diffuse .6 .6 .6
specular 1 1 1
exponent 50

def jackPart{ push scale .2 .2 1 sphere pop
```

```
push translate  0 0 1.2 scale .2 .2 .2 sphere pop
push translate 0 0 -1.2 scale .2 .2 .2 sphere pop
}

def jack{ push use jackPart
rotate 90 0 1 0 use jackPart
rotate 90 1 0 0 use jackPart pop
}

def wall{push translate 1 .01 1 scale 1 .02 1 cube pop}
def leg {push translate 0 .15 0 scale .01 .15 .01 cube pop}

def table{
push translate  0  .3  0 scale .3 .01 .3 cube pop !table top
push
translate .275  0 .275 use leg
translate  0 0 -.55 use leg
translate -.55 0 .55 use leg
translate 0  0 -.55 use leg pop
}
!now add the objects themselves
push translate .4 .4 .6 rotate 45 0 0 1 scale .08 .08 .08 use jack pop
push translate .25 .42 .35 scale .1 .1 .1 sphere pop
push translate .6 .38 .5 rotate 30 0 1 0 scale .08 .08 .08 teapot pop
push translate 0.4 0 0.4 use table pop

use wall
push rotate 90 0 0 1 use wall pop
push rotate -90 1 0 0 use wall pop
```

Figure 5.67. The SDL file to create the scene of Figure 5.63.

With a scene description language like SDL available, along with tools to read and parse it, the scene designer can focus on creating complex scenes without having to work at the application code level. The scene being developed can be edited and tested again and again until it is right. The code developer puts the initial effort into constructing an application that can render any scene describable in SDL.

5.7. Summary of the Chapter.

Affine transformations are a staple in computer graphics, for they offer a unified tool for manipulating graphical objects in important ways. A designer always needs to scale, orient, and position objects in order to compose a scene as well as an appropriate view of the scene, and affine transformations make this simple to manage in a program.

Affine transformations convert one coordinate frame into another, and when homogeneous coordinates are used, an affine transformation is captured in a single matrix form. A sequence of such transformations can be combined into a single transformation whose matrix is simply the product of the individual transformation matrices. Significantly, affine transformations preserve straightness, so the image of a line is another line, and the image of a plane is a plane. This vastly simplifies working with lines and planes in a program, where one benefits from the simple representations of a line (its two endpoints) and a plane (by three points or four coefficients). In addition, parallelism is preserved, so that parallelograms map to parallelograms, and in 3D parallelepipeds map to parallelepipeds. This makes it simpler to visualize the geometric effects of affine transformations.

Three dimensional affine transformations are much more complex than their 2D counterparts, particularly when it comes to visualize a combination of rotations. A given rotation can be viewed as three elementary rotations through Euler angles, or a rotation about some axis, or simply as a matrix that has special properties. (Its columns are orthogonal unit vectors). It is often important to move between these different forms.

OpenGL and other graphics packages offer powerful tools for manipulating and applying transformations. In OpenGL all points are passed through several transformations, and the programmer can exploit this to define and

manipulate a "camera", as well as to size and position different objects into a scene. Two of the matrices used by OpenGL (the modelview and viewport transformations) define affine transformations, whereas the projection matrix normally defines a perspective transformation, to be examined thoroughly in Chapter 7. OpenGL also maintains a stack of transformations, which make it easy for the scene designer to control the dependency of one object's position on that of another, and to create objects that are composed of several related parts.

The SDL language, along with the Scene and Shape classes, make it much simpler to separate programming issues from scene design issues. An application is developed once that can draw any scene described by a list of light sources and a list of geometric objects. This application is then used over and over again with different scene files. A key task in the scene design process is applying the proper geometric transformations to each object. Since a certain amount of trial and error is usually required, it is convenient to be able to express these transformations in a concise and readable way.

The next section presents a number of Case Studies that elaborate on the main ideas of the chapter and suggest ways to practice with affine transformations in a graphics program. These range from plunging deeper into the theory of transformations to actual modeling and rendering of objects such as electronic CAD circuits and robots.

5.8 Case Studies.

Case study 5.1. Doing your own transforming by the CT in Canvas.

(Level of Effort: II). It's easy to envision situations where you must implement the transformation mechanism yourself, rather than rely on OpenGL to do it. In this Case Study you add the support of a current transformation to the Canvas class for 2D drawing. This involves writing several functions, those to initialize and alter the *CT* itself:

```
void Canvas:: initCT(void);   // init CT to unit transformation
void Canvas:: scale2D(double sx, double sy);
void Canvas:: translate2D(double dx, double dy);
void Canvas:: rotate2D(double angle);
```

as well as others that are incorporated into moveTo() and `lineTo`() so that all points sent to them are "silently" transformed before being used. For extra benefit add the stack mechanism for the CT as well, along with functions `pushCT`() and `popCT`(). Exercise your new tools on some interesting 2D modeling and drawing examples.

Case Study 5.2. Draw the star of Fig 5.39. using multiple rotations.

(Level of Effort: I). Develop a function that draws the polygon in figure 5.39b that is one fifth of the star. Use it with rotation transformations to draw the whole star.

Case Study 5.3. Decomposing a 2D Affine Transformation.

(Level of Effort: II). We have seen that some affine transformations can be expressed as combinations of others. Here we "decompose" any 2D affine transformation into a combination of scalings, rotations, and translations. We also show that a rotation can be performed by three successive shears, which gives rise to a very fast arc drawing routine. Some results on what lurks in any 3D affine transformation are also discussed. We just summarize results here.

Because an affine transformation is a linear transformation followed by an offset we omit the translation part of the affine transformation, and focus on what a linear transformation is. So we use 2 by 2 matrices here for simplicity.

a). Two 2D linear transformations.

Consider the 2 by 2 matrix *M* that represents a 2D linear transformation. Matrix *M* can always be factored into a rotation, a scaling , and a shear. Call the four scalars in *M* by the names *a*, *b*, *c*, and *d* for brevity. Verify by direct multiplication that *M* is the product of the three matrices [martin82]:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{ac+bd}{R^2} & 1 \end{pmatrix} \begin{pmatrix} R & 0 \\ 0 & \frac{ad-bc}{R} \end{pmatrix} \begin{pmatrix} \frac{a}{R} & \frac{b}{R} \\ -\frac{b}{R} & \frac{a}{R} \end{pmatrix} \qquad (5.39)$$

where $R = \sqrt{a^2 + b^2}$. The leftmost matrix on the right hand side is recognized as a shear, the middle one as a scaling, and the rightmost one as a rotation (why?).

Thus *any* 2D affine transformation is a rotation followed by a scaling followed by a shear followed by a translation.

An alternative decomposition, based on the so-called "Gram-Schmidt process", is discussed in Case Study 5.5.

Example 5.8.1. Decompose the matrix $M = \begin{pmatrix} 4 & -3 \\ 2 & 7 \end{pmatrix}$ into the product of shear, scaling, and rotation matrices.

Solution: Check the following, obtained by direct substitution in Equation 5.39:

$$M = \begin{pmatrix} 4 & -3 \\ 2 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -13/25 & 1 \end{pmatrix} \begin{pmatrix} 5 & 0 \\ 0 & 34/5 \end{pmatrix} \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix}$$

b). A 2D Rotation is three shears.

Matrices can be factored in different ways. In fact a rotation matrix can be written as the product of three shear matrices! [Paeth90] This leads to a particularly fast method for performing a series of rotations, as we will see.

Equation 5.40 shows a rotation represented as three successive shears. It can be verified by direct multiplication. It demonstrates that a rotation is a shear in y followed by a shear in x, followed by a repetition of the first shear.

$$\begin{pmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{pmatrix} = \begin{pmatrix} 1 & \tan(a/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\sin(a) & 1 \end{pmatrix} \begin{pmatrix} 1 & \tan(a/2) \\ 0 & 1 \end{pmatrix} \qquad (5.40)$$

Calling $T = tan(a/2)$, and $S = sin(a)$ show that we can write the sequence of operations that rotate point (x, y) as[15]:

```
x' = T * y + x;                 {first shear}
y' = y;

x'' = x';                       {second shear}
y'' = y' - S *x';

x''' = T * y'' + x';            {third shear}
y''' = y'';
```

using primes to distinguish new values from old. But operations like $x'' = x'$ do nothing, so the primes are unnecessary and this sequence reduces to:

```
x  = x + T * y;
y  = y - S * x;                 {actual operations for the three shears}
x  = x + T *y;
```

[15]Note that $sin(a)$ may be found quickly from tan(a/2) by one multiplication and one division (see Appendix 2): $S = (2T)/(1 + T^2)$

If we have only one rotation to perform there is no advantage to going about it this way. However, it becomes very efficient if we need to do a succession of rotations through the same angle. Two places where we need to do this are 1). calculating the vertices of an *n*-gon — which are equispaced points around a circle, and 2). computing the points along the arc of a circle.

Figure 5.68 shows a code fragment for calculating the positions of *n* points around a circle. It loads the successive values ($cos(i * 2\pi/n + b)$, $sin(i * 2\pi/n + b)$) into the array of points `p[]`.

```
T = tan(PI/n);            // tangent of half angle[16]
S = 2 * T/(1 + T * T);    // sine of angle
p[0].x = sin(b);          // initial angle, vertex 0
p[0].y = cos(b);
for (int i = 1; i < n;  i++)
{
   p[i].y = p[i-1].x * T + p[i-1].y; //1st shear
   p[i].x = p[i-1].x  -S * p[i-1].y; //2nd shear
   p[i].y = p[i-1].x * T + p[i-1].y; //3rd shear
}
```

Figure 5.68. Building the vertices of an n-gon efficiently.

Figure 5.69 shows how to use shears to build a fast arc drawer. It does the same job as `drawArc()` in Chapter 3, but much more efficiently, since it avoids the repetitive computation of *sin*() and *cos*().

```
void drawArc2(RealPoint c, double R,
            double startangle, double sweep) // in degrees
{
    #define n 30
    #define RadPerDeg .01745329
    double delang =  RadPerDeg * sweep / n;
    double T =  tan(delang/2);          // tan. of half angle
    double S =  2 * T/(1 + T * T);      // sine of half angle
    double snR = R * sin(RadPerDeg * startangle);
    double csR = R * cos(RadPerDeg * startangle);
    moveTo(c.x + csR, c.y + snR);
    for(int i = 1; i < n; i++)
    {
       snR += T * csR;     // build next snR, csR pair
       csR -= S * snR;
       snR += T * csR;
       lineTo(c.x + csR, c.y + snR);
    }
}
```

Figure 5.69. A fast arc drawer.

Develop a test program that uses this routine to draw arcs. Compare its efficiency with arc drawers that compute each vertex using trigonometry.

c). Is a shear "fundamental"?

One sometimes reads in the graphics literature that the fundamental elementary transformations are the rotation, scaling, and translation; shears seem to be second class citizens. This attitude may stem from the fact that any shear can be decomposed into a combination of rotations and scalings. The following equation may be verified by multiplying out the three matrices on the right [Greene90]:

$$\begin{pmatrix} 1 & 0 \\ a - \frac{1}{a} & 1 \end{pmatrix} = \frac{1}{\sqrt{1+a^2}} \begin{pmatrix} 1 & -a \\ a & 1 \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & 1/a \end{pmatrix} \begin{pmatrix} a & 1 \\ -1 & a \end{pmatrix} \frac{1}{\sqrt{1+a^2}} \tag{5.41}$$

[16] Note: Some C environments do not support *tan*() directly. Use *sin*()/*cos*() - of course first checking that this denominator is not zero.

The middle matrix is a scaling, while the outer two matrices (when combined with the scale factors shown) are rotations. For the left-hand rotation associate $1/\sqrt{1 + a^2}$ with $cos(\alpha)$ and $-a/\sqrt{1 + a^2}$ with $sin(\alpha)$ for some angle α. Thus $tan(\alpha) = -a$. Similarly for the right-hand rotation associate $cos(\beta)$ and $sin\beta)$ for angle β, where $tan(\beta) = 1/a$. Note that α and β are related: $\beta = \alpha + \pi / 2$ (why?).

Using this decomposition, write the shear in Equation 5.41 as a "rotation then a scaling then a rotation" to conclude that every 2D affine transformation is the following sequence of elementary transformations:

Any affine transformation = Scale * Rotation * Scale * Rotation * Translation (5.42)

Practice Exercises

5.8.1. A "golden" decomposition. Consider the special case of a "unit shear" where the term $a - 1/a$ in Equation 5.3.5 is 1. What value must a have? Determine the two angles α and β associated with the rotations.

Solution: Since a must satisfy $a = 1 + 1/a$, a is the golden ratio ϕ!. Thus

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix} \begin{pmatrix} \phi & 0 \\ 0 & \frac{1}{\varphi} \end{pmatrix} \begin{pmatrix} cos(\beta) & sin(\beta) \\ -sin(\beta) & cos(\beta) \end{pmatrix} \quad (5.43)$$

where $\alpha = tan^{-1}(\phi) = 58.28°$ and $\beta = tan - 1(1/\phi) = 31.72°$.

5.8.2. Unit Shears. Show that any shear in x contains within it a unit shear. Decompose the shear given by

$$\begin{pmatrix} 1 & 0 \\ h & 1 \end{pmatrix}$$

into the product of a scaling, a unit shear, and another scaling.

5.8.3. Seeing It Graphically. Drawing upon the ideas in Exercise 5.8.1, draw a rectangle on graph paper; rotate it through $-58.28°$; scale it by $(\phi, 1/\phi)$; and finally rotate it through $31.72°$. Sketch each intermediate result, and show that the final result is the same parallelogram obtained when the original rectangle undergoes a unit shear.

5.8.4. Decompose a Transformation. Decompose the transformation

$Q_X = 3 P_X - 2 P_Y + 5$

$Q_Y = 4 P_X + P_Y - 6$

into a product of rotations, scalings, and translations.

5.8.5. One reflection can always be in the x-axis. Show that a rotation through angle A about the origin may always be effected by a reflection in the x-axis followed by a reflection in a line at angle $A/2$.

5.8.6. Isometries. A particularly important family of affine transformations in the study of symmetry are the **isometries** ("same measure"), since they don't alter the distance between two points and their images. For any two points P and Q the distance $|T(P) - T(Q)|$ is the same as the distance $|P - Q|$ when $T()$ is an isometry. Show that if $T()$ is affine with matrix M , then T is an isometry if and only if each row of M considered as a vector is of unit length, and the two rows are orthogonal.

Solution: Call $\mathbf{r} = P - Q$. Then $|T(P) - T(Q)| = |PM + \mathbf{d} - QM - \mathbf{d}| = |\mathbf{r}M| = |(r_X m_{11} + r_Y m_{21}, r_X m_{12} + r_Y m_{22})|$.

Equating $|\mathbf{r}M|^2$ to $|\mathbf{r}|^2$ requires $m_{11}^2 + m_{12}^2 = 1$, $m_{21}^2 + m_{22}^2 = 1$, and $m_{11}m_{21} + m_{22}m_{12} = 0$, as claimed.

5.8.7. Ellipses Are Invariant. Show that ellipses are invariant under an affine transformation. That is, if E is an ellipse and if T is an affine transformation, then the image $T(E)$ of the points in E also makes an ellipse. **Hint to Solution**: Any affine is a combination of rotations and scalings. When an ellipse is rotated it is still an ellipse, so only the non-uniform scalings could possibly destroy the ellipse property. So it is necessary only to show that an ellipse, when subjected to a non-uniform scaling, is still an ellipse.

5.8.8. What else is invariant? Consider what class of shapes (perhaps a broader class than ellipses) is invariant to affine transformations. If the equation $f(x,y)=0$ describes a shape, show that after transforming it with transformation T, the new shape is described by all points that satisfy $g(x,y) = f(T^{-1}(x,y)) = 0$. Then show the details of this form when T is affine. Finally, try to describe the largest class of shapes which is preserved under affine transformations.

Case Study 5. 4. Generalized 3D Shears.

(Level of Effort:II) A shear can be more general than those discussed in Section 5.???. As suggested by Goldman [goldman 91] the ingredients of a shear are:

- A plane through the origin having unit normal vector **m**;
- A unit vector **v** lying in the plane (thus perpendicular to **m**);
- An angle ϕ.

Then as shown in Figure 5.70 point *P* is sheared to point *Q* by shifting it in direction **v** a certain amount. The amount is proportional to both the distance at which *P* lies from the plane, and to tan ϕ. Goldman shows that this shear has the matrix representation:

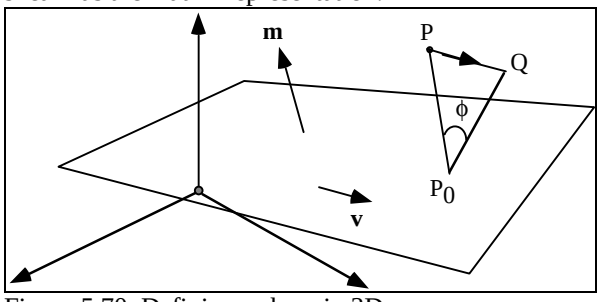

Figure 5.70. Defining a shear in 3D.

$$M = I + \tan(\phi)\begin{pmatrix} m_x v_x & m_x v_y & m_x v_z \\ m_y v_x & m_y v_y & m_y v_z \\ m_z v_x & m_z v_y & m_z v_z \end{pmatrix} \qquad (5.44)$$

where *I* is the 3 by 3 identity matrix, and an offset vector of **0**. Some details of the derivation are given in the exercises.

Example 5.8.2: Find the shear associated with the plane having unit normal vector **m** = $(1,1,1)/\sqrt{3}$ = (0.577, 0.577, 0.577), unit vector **v** = (0, 0.707 -0.707), and angle ϕ = 30°. **Solution:** Note that **v** lies in the plane as required (why?). Applying Equation 9.9.5 we obtain:

$$M = I + 0.577\begin{pmatrix} 0 & 0.408 & -0.408 \\ 0 & 0.408 & -0.408 \\ 0 & 0.408 & -0.408 \end{pmatrix} = \begin{pmatrix} 1 & 0.235 & -0.235 \\ 0 & 1.235 & -0.235 \\ 0 & 0.235 & 0.764 \end{pmatrix}$$

Derive the shear matrix. We want to express the point *Q* in Figure 5.73 in terms of *P* and the ingredients of the shear. The (signed) distance of *P* from the plane is given by $P \cdot \mathbf{m}$ (considering *P* as a position vector pinned to the origin). a). Put the ingredients together to obtain:

$$Q = P + (P \cdot \mathbf{m}) \tan(\phi)\, \mathbf{v} \qquad (5.45)$$

Note that points "on the other side" of the plane are sheared in the opposite direction, as we would expect.

Now we need to the second term as *P* times some matrix. Use Appendix 2 to show that $P \cdot \mathbf{m}$ is $P\mathbf{m}^{\mathbf{T}}$, and therefore that Q = P (I + tan(ϕ) $\mathbf{m}^{\mathbf{T}}$ **v**), and that the shear matrix is (I + tan(ϕ) $\mathbf{m}^{\mathbf{T}}$ **v**). Show that the matrix $\mathbf{m}^{\mathbf{T}}$ **v** has the form:

$$\mathbf{m}^T\mathbf{v} = \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix}(v_x, v_y, v_z) = \begin{pmatrix} m_x v_x & m_x v_y & m_x v_z \\ m_y v_x & m_y v_y & m_y v_z \\ m_z v_x & m_z v_y & m_z v_z \end{pmatrix} \qquad (5.46)$$

This is called the **outer product** (or **tensor product**) of **m** with **v**.

Practice Exercises.

5.8.9. Consistency with a simple shear. Express the matrix for the shear associated with direction **v** = (1,0,0) and the plane having normal vector (0, 1, 0). Show that this is a form of the elementary shear matrix given in Equation 5.45, for any angle ϕ.

5.8.10. Find the shear. Compute the shear matrix for the shear that uses a normal **m** = 0.577(1, -1, 1), direction vector **v** = 0.707(1,1,0), and angle $\phi = 45°$. Sketch the vectors and plane involved, and sketch how a cube centered at the origin with edges aligned with the coordinate axes is affected by the shear.

5.8.11. How Is a Three-Dimensional Shear Like a Two-Dimensional Translation in Homogeneous Coordinates?

Consider the specific shear:

$$Q = P\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t & s & 1 \end{pmatrix} \qquad (5.47)$$

which alters the 3D point *P* by shearing it along *x* with factor *t* owing to *z* and along *y* with factor *s* owing to *z*. If *P* lies in the *z* = 1 plane with coordinates $P = (p_x, p_y, 1)$, show that *P* is transformed into $(p_x + t, p_y + s, 1)$. Hence it is simply shifted in *x* by amount *t* and in *y* by amount *s*. Show that for any point in the *z* = 1 plane, this shear is equivalent to a translation. Show that the matrix in Equation 5.48 is identical to the homogeneous coordinate form for a pure translation in two dimensions. This correspondence helps reinforce how homogeneous coordinates operate.

Case Study 5.5. Rotation about an Axis: The Constructive Approach.

(Level of Effort: II) You are asked to fill in the details in the following derivation of the rotation matrix $R_{\mathbf{u}}(\beta)$ of Equation 9.9.15. For simplicity we assume **u** is a unit vector: $|\mathbf{u}| = 1$. Denote the position vector based on *P* by the name **p**, so **p** = *P* - *O* where *O* is the origin of the coordinate system. Now project **p** onto **u** to obtain vector **h**, as shown in Figure 9.9.8a.

a). Show it has the form $\mathbf{h} = (\mathbf{p} \cdot \mathbf{u})\,\mathbf{u}$. Define two perpendicular vectors **a** and **b** that lie in the plane of rotation: **a** = **p** - **h**, and $\mathbf{b} = \mathbf{u} \times \mathbf{a}$

b). Show that they are perpendicular, they have the same length, they both lie in the plane, and that $\mathbf{b} = \mathbf{u} \times (\mathbf{p} - \mathbf{h})$ simplifies to $\mathbf{u} \times \mathbf{p}$. This effectively establishes a 2D coordinate system in the plane of rotation. Now look onto the plane of rotation, as in Figure 9.9.8b. The rotation rotates **a** to **a'** = cos β **a** + sin β **b**, so the rotated point *Q* is given by:

Q = **h** + cos β **a** + sin β **b**, or using the expressions for **a** and **b,**

$$Q = \mathbf{p}\cos\beta + (1 - \cos\beta)(\mathbf{p} \cdot \mathbf{u})\,\mathbf{u} + \sin\beta\,(\mathbf{u} \times \mathbf{p}) \qquad (5.48)$$

This is a rather general result, showing how the rotated point *Q* can be decomposed into portions along **h** and along two orthogonal axes lying in the plane of rotation.

This form for *Q* hardly looks like the multiplication of *P* by some matrix, but it is because each of the three terms is linear in **p**. Convert each of the terms to the proper form as follows:

c). Replace **p** with *P* to obtain immediately $\mathbf{p}(\cos\beta) = P(\cos\beta)\ I$ where *I* is the 3 by 3 identity matrix.

d). Use the result (Appendix 2) that a dot product $\mathbf{p} \cdot \mathbf{c}$ can be rewritten as *P* times a matrix: $P\,\mathbf{u}^{\mathrm{T}}$, to show that $(\mathbf{p} \cdot \mathbf{u})\mathbf{u} = P\,\mathbf{u}^{\mathrm{T}}\mathbf{u}$ where $\mathbf{u}^{\mathrm{T}}\mathbf{u}$ is an outer product similar to Equation 9.9.7.

e). Use the fact developed in Appendix 2 that a cross product $\mathbf{u} \times \mathbf{p}$ can also be written as *P* times a matrix to show that $\mathbf{u} \times \mathbf{p} = P\ \mathit{Cross}(\mathbf{u})$ where matrix *Cross*(**c**) is:

$$Cross(\mathbf{u}) = \begin{pmatrix} 0 & u_z & -u_y \\ -u_z & 0 & u_x \\ u_y & -u_x & 0 \end{pmatrix} \quad (5.49)$$

f). Put these together to obtain the matrix

$$R_u(\beta) = cos\ \beta\, I + (1 - \cos\beta)\, \mathbf{u}^T\mathbf{u} + \sin\beta\, Cross(\mathbf{u})) \quad (5.50)$$[17]

M is therefore the sum of three weighted matrices. It surely is easier to build than the product of five matrices as in the classic route.
g). Write this out to obtain Equation 5.33.

Case Study 5.6. Decomposing 3D Affine Transformations.

(Level of Effort:III) This Case Study looks at several broad families of affine tansformations.

What is in a 3D Affine Transformation?
Once again we ignore the translation portion of an affine transformation and focus on the linear transformation part represented by 3 x 3 matrix *M*. What kind of transformations are "imbedded" in it? It is somewhat more complicated. Goldman[GEMSIII, p.108] shows that every such *M* is the product of a scaling *S*, a rotation *R* , and two shears H_1 and H_2.

$$M = S\, R\, H_1\, H_2 \quad (5.51)$$

Every 3D affine transformation, then, can be viewed as this sequence of elementary operations, followed by a translation. In Case Study 5.??? we explore the mathematics behind this form, and see how an actual decomposition might be carried out.

Useful Classes of Transformations.
It's useful to categorize affine transformations, according to what they "do" or "don't do" to certain properties of an object when it is transformed. We know they always preserve parallelism of edges of an object, but which transformations also preserve the length of each edge, and which ones preserve the angles between each pair of edges?

1). Rigid Body Motions.
It is intuitively clear that translating an object, or rotating it, will not change its shape or size. In addition, reflecting it about a plane has no effect on its shape or size. Since the shape is not affected by any one of these transformations alone, it is also not affected by an arbitrary composition of them. We denote by

T_{rigid} = {rotations, reflections, translations}

the collection of all affine transformations that consist of any sequence of rotations, reflections, and translations. These are known classically as the **rigid body motions**, since an object is moved rigidly from one position and orientation to another. Such transformations have *orthogonal* matrices in homogeneous coordinates. These are matrices for which the inverse is the same as the transpose:

$$\tilde{M}^{-1} = \tilde{M}^{T}$$

2). Angle-Preserving Transformations.
A uniform scaling (having equal scale factors $S_x = S_y = S_z$) expands or shrinks an object, but does so uniformly, so there is no change in the object's shape. Thus the angle between any two edges is unaffected. We can denote such a class as

[17] Goldman [gold90] reports the same form for *M*, and gives compact results for several other complex transformations.

T_{angle} = {rotations, reflections, translations, uniform scalings}

This class is larger than the rigid body motions, because it includes uniform scaling. It is an important class because, as we see in Chapter ???, lighting and shading calculations depend on the dot products between various vectors. If a certain transformation does not alter angles then it does not alter dot products, and lighting calculations can take place in either the transformed or the untransformed space.

(Estimate of time required: three hours.) Given a 3D affine transformation {*M*, **d**} we wish to see how it is composed of a sequence of elementary transformations. Following Goldman [GEMS III, p. 108] we develop the steps required to decompose the 3 by 3 matrix *M* into the product of a scaling *S*, a rotation *R*, and two shears H_1 and H_2:

$$M = S\,R\,H_1\,H_2 \qquad (5.52)$$

You are asked to verify each step along the way, and to develop a routine that will produce the individual matrices *S*,*R* , H_1 and H_2.

Suppose the matrix *M* has rows **u**, **v**, and **w**, each a 3D vector:

$$M = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{pmatrix}$$

Goldman's approach is based on the classical Gram-Schmidt orthogonalization procedure, whereby the rows of *M* are combined in such a way that they become mutually orthogonal and of unit length. The matrix composed of these rows is therefore orthogonal (see Practice Exercise 9.9.14) and so represents a rotation (or a rotation with a reflection). Goldman shows that the orthogonalization process is in fact two shears. The rest is detail.

The steps are as follows. Carefully follow each step, and do each of the tasks given in brackets.
1. Normalize **u** to $\mathbf{u}^* = \mathbf{u}/S_1$, where $S_1 = |\mathbf{u}|$.
2. Subtract a piece of $\mathbf{u}^*$ from **v** so that what is left is orthogonal to $\mathbf{u}^*$: Call $\mathbf{b} = \mathbf{v} - d\mathbf{u}^*$ where $d = \mathbf{v}\cdot\mathbf{u}^*$. [Show that $\mathbf{b}\cdot\mathbf{u}^* = 0$.]
3. Normalize **b**: set $\mathbf{v}^* = \mathbf{b}/S_2$, where $S_2 = |\mathbf{b}|$.
4. Set up some intermediate values: $m = \mathbf{w}\cdot\mathbf{u}^*$ and $n = \mathbf{w}\cdot\mathbf{v}^*$, $e = \sqrt{m^2 + n^2}$, and $\mathbf{r} = (m\mathbf{u}^* + n\mathbf{v}^*)/e$.
5. Subtract a piece of **r** from **w** so that what is left is orthogonal to both $\mathbf{u}^*$ and $\mathbf{v}^*$: Call $\mathbf{c} = \mathbf{w} - e\mathbf{r}$. [Show that $\mathbf{c}\cdot\mathbf{u}^* = \mathbf{c}\cdot\mathbf{v}^* = 0$.]
5. Normalize **c**: set $\mathbf{w}^* = \mathbf{c}/S_3$ where $S_3 = |\mathbf{c}|$.
7. The matrix

$$R = \begin{pmatrix} \mathbf{u}^* \\ \mathbf{v}^* \\ \mathbf{w}^* \end{pmatrix}$$

is therefore orthogonal, and so represents a rotation. (Compute its determinant: if it is -1 then simply replace $\mathbf{w}^*$ with $-\mathbf{w}^*$.)

8. Define the shear matrix $H_1 = I + \frac{d}{S_2}(\mathbf{v}^* \otimes \mathbf{u}^*)$ (see Practice exercise 9.9.1), where

$(\mathbf{v}^* \otimes \mathbf{u}^*) = (\mathbf{v}^*)^T \mathbf{u}^*$ is the outer product of v* and u*.

9. Define the shear matrix $H_2 = I + \frac{e}{S_3}(\mathbf{w}^* \otimes \mathbf{r})$, where $(\mathbf{w}^* \otimes \mathbf{r}) = (\mathbf{w}^*)^T \mathbf{r}$.

10. [Show that $\mathbf{u}^* H_1 = \mathbf{u}^*$, $\mathbf{v}^* H_1 = \mathbf{v}^* + d\mathbf{u}^*/S_2 = \mathbf{v}/S_2$, and $\mathbf{w}^* H_1 = \mathbf{w}^*$. First show the property of the outer product that for any vectors **a**, **b**, and **c**: $\mathbf{a}(\mathbf{b} \otimes \mathbf{c}) = (\mathbf{a}\cdot\mathbf{b})\mathbf{c}$. Then use this property, along with the orthogonality of $\mathbf{u}^*$, $\mathbf{v}^*$ and $\mathbf{w}^*$, to show the three relations.]
11. [Show that $\mathbf{u}^* H_2 = \mathbf{u}^*$, $\mathbf{v}^* H_2 = \mathbf{v}^*$, $\mathbf{w}^* H_2 = \mathbf{w}^* + e\mathbf{r}/S_3 = \mathbf{w}/S_3$.

12. [Put these together to show that $M = SRH_1H_2$.] S is defined to be

$$S = \begin{pmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{pmatrix}$$

Note that the decomposition is not unique, since the vectors **u**, **v**, and **w** could be orthogonalized in a different order. For instance, we could first form **w*** as **w**/|**w**|, then subtract a piece of **v** from **w*** to make a vector orthogonal to **w***, then subtract a vector from **u** to make a vector orthogonal to the other two.

Write the routine:

```
void decompose(DBL m[3][3],DBL S[3][3],DBL R[3][3],DBL H1[3][3],DBL
H2[3][3])
```

where `DBL` is defined as `double`, that takes matrix m and computes the matrices S, R, H1 and H2 as described above. Test your routine on several matrices.

Other ways to decompose a 3D transformation have been found as well. See for instance [thomas, GEMS II, p. 320] and [Shoemake, GEMS IV p. 207].

Case Study 5.7. Drawing 3D scenes described by SDL.

(Level of Effort:II) Develop a complete application that uses the `Scene`, `Shape`, `Affine4`, etc. classes and supports reading in and drawing the scene described in an SDL file. To assist you, use any classes provided on the book's web site. Flesh out any `drawOpenGL`() methods that are needed. Develop a scene file that contains descriptions of the jack, a wooden chair, and several walls.

5.9. For Further Reading

There are some excellent books on transformations that supplement the treatment here. George Martin's TRANSFORMATION GEOMETRY [martin82] is superb, as is Yaglom's GEOMETRIC TRANSFORMATIONS [yaglom62]. Hogar also provides a fine chapter on vectors and transformations [hoggar92]. Several of Blinn's engaging articles in JIM BLINN'S CORNER - A TRIP DOWN THE GRAPHICS PIPELINE [blinn96] provide excellent discussions of homogeneous coordinates and transformations in computer graphics.