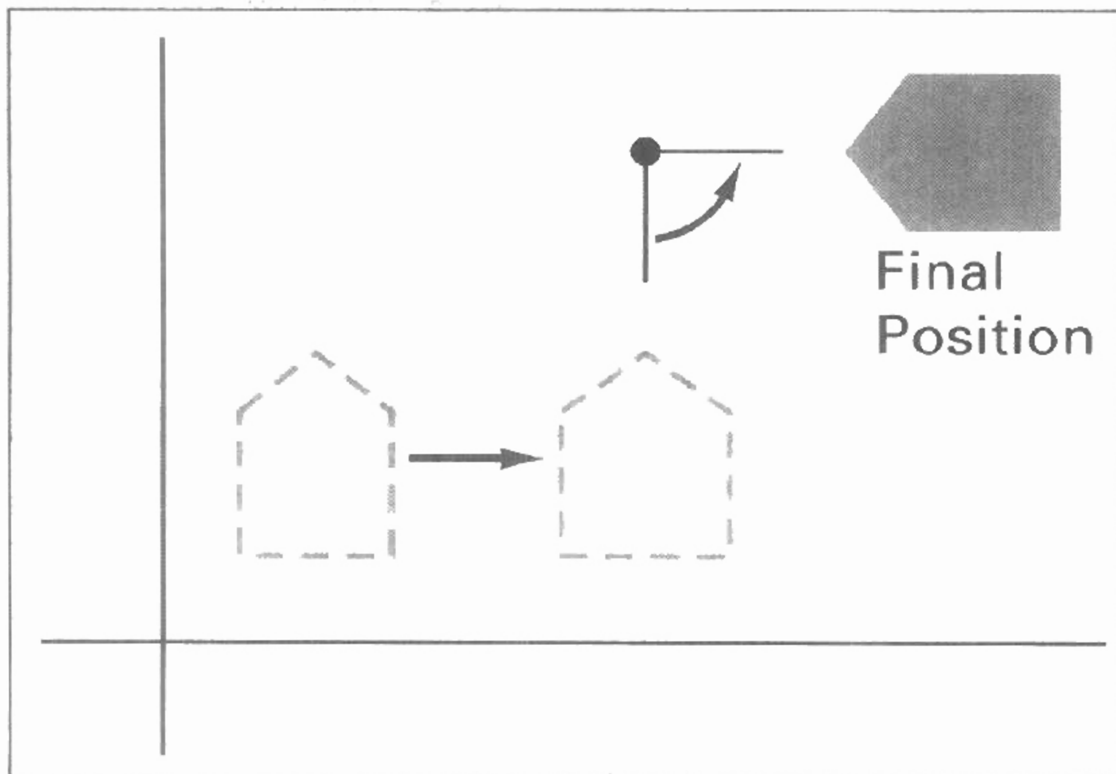
CHAPTER

5 Two-Dimensional Geometric Transformations



With the procedures for displaying output primitives and their attributes, we can create a variety of pictures and graphs. In many applications, there is also a need for altering or manipulating displays. Design applications and facility layouts are created by arranging the orientations and sizes of the component parts of the scene. And animations are produced by moving the "camera" or the objects in a scene along animation paths. Changes in orientation, size, and shape are accomplished with **geometric transformations** that alter the coordinate descriptions of objects. The basic geometric transformations are translation, rotation, and scaling. Other transformations that are often applied to objects include reflection and shear. We first discuss methods for performing geometric transformations and then consider how transformation functions can be incorporated into graphics packages.

5-1 BASIC TRANSFORMATIONS

Here, we first discuss general procedures for applying translation, rotation, and scaling parameters to reposition and resize two-dimensional objects. Then, in Section 5-2, we consider how transformation equations can be expressed in a more convenient matrix formulation that allows efficient combination of object transformations.

Translation

A **translation** is applied to an object by repositioning it along a straight-line path from one coordinate location to another. We translate a two-dimensional point by adding **translation distances**, t_x and t_y, to the original coordinate position (x, y) to move the point to a new position (x', y') (Fig. 5-1).

$$x' = x + t_x, \qquad y' = y + t_y \tag{5-1}$$

The translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector.**

We can express the translation equations 5-1 as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

$$\mathbf{P} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{5-2}$$

This allows us to write the two-dimensional translation equations in the matrix form:

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \tag{5-3}$$

Sometimes matrix-transformation equations are expressed in terms of coordinate row vectors instead of column vectors. In this case, we would write the matrix representations as $\mathbf{P} = [x \ y]$ and $\mathbf{T} = [t_x \ t_y]$. Since the column-vector representation for a point is standard mathematical notation, and since many graphics packages, for example, GKS and PHIGS, also use the column-vector representation, we will follow this convention.

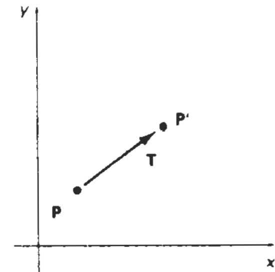
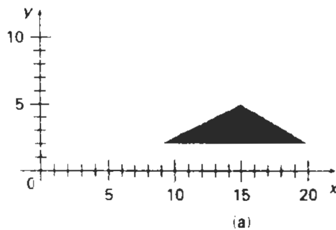


Figure 5-1
Translating a point from position **P** to position **P'** with translation vector **T**.

Translation is a *rigid-body transformation* that moves objects without deformation. That is, every point on the object is translated by the same amount. A straight line segment is translated by applying the transformation equation 5-3 to each of the line endpoints and redrawing the line between the new endpoint positions. Polygons are translated by adding the translation vector to the coordinate position of each vertex and regenerating the polygon using the new set of vertex coordinates and the current attribute settings. Figure 5-2 illustrates the application of a specified translation vector to move an object from one position to another.

Similar methods are used to translate curved objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. We translate other curves (for example, splines) by displacing the coordinate positions defining the objects, then we reconstruct the curve paths using the translated coordinate points.



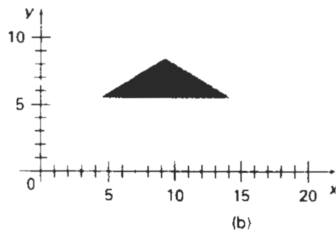


Figure 5-2
Moving a polygon from position (a) to position (b) with the translation vector (-5.50, 3.75).

Rotation

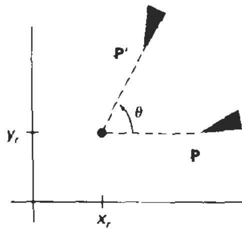


Figure 5-3
Rotation of an object through angle θ about the pivot point (x_r, y_r).

A two-dimensional **rotation** is applied to an object by repositioning it along a circular path in the xy plane. To generate a rotation, we specify a **rotation angle** θ and the position (x_r, y_r) of the **rotation point** (or **pivot point**) about which the object is to be rotated (Fig. 5-3). Positive values for the rotation angle define counterclockwise rotations about the pivot point, as in Fig. 5-3, and negative values rotate objects in the clockwise direction. This transformation can also be described as a rotation about a **rotation axis** that is perpendicular to the xy plane and passes through the pivot point.

We first determine the transformation equations for rotation of a point position **P** when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in Fig. 5-4. In this figure, r is the constant distance of the point from the origin, angle ϕ is the original angular position of the point from the horizontal, and θ is the rotation angle. Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ and ϕ as

$$\begin{aligned} x' &= r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' &= r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{aligned} \tag{5-4}$$

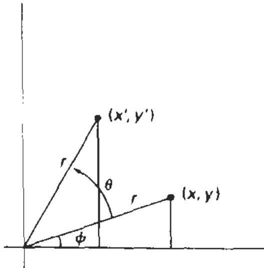


Figure 5-4
Rotation of a point from position (x, y) to position (x', y') through an angle θ relative to the coordinate origin. The original angular displacement of the point from the x axis is ϕ.

The original coordinates of the point in polar coordinates are

$$x = r\cos\phi, \qquad y = r\sin\phi \tag{5-5}$$

Substituting expressions 5-5 into 5-4, we obtain the transformation equations for rotating a point at position (x, y) through an angle θ about the origin:

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned} \tag{5-6}$$

With the column-vector representations 5-2 for coordinate positions, we can write the rotation equations in the matrix form:

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \tag{5-7}$$

where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{5-8}$$

When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation 5-7 is transposed so that the transformed row coordinate vector $[x' \ y']$ is calculated as

$$\begin{aligned} \mathbf{P}'^T &= (\mathbf{R} \cdot \mathbf{P})^T \\ &= \mathbf{P}^T \cdot \mathbf{R}^T \end{aligned}$$

where $\mathbf{P}^T = [x \ y]$, and the transpose $\mathbf{R}^T$ of matrix **R** is obtained by interchanging rows and columns. For a rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

Rotation of a point about an arbitrary pivot position is illustrated in Fig. 5-5. Using the trigonometric relationships in this figure, we can generalize Eqs. 5-6 to obtain the transformation equations for rotation of a point about any specified rotation position (x_r, y_r):

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \tag{5-9}$$

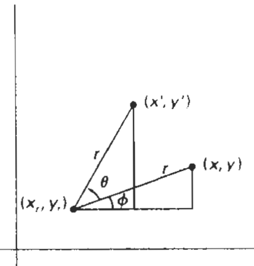


Figure 5-5
Rotating a point from position (x, y) to position (x', y') through an angle θ about rotation point (x_r, y_r).

These general rotation equations differ from Eqs. 5-6 by the inclusion of additive terms, as well as the multiplicative factors on the coordinate values. Thus, the matrix expression 5-7 could be modified to include pivot coordinates by matrix addition of a column vector whose elements contain the additive (translational) terms in Eqs. 5-9. There are better ways, however, to formulate such matrix equations, and we discuss in Section 5-2 a more consistent scheme for representing the transformation equations.

As with translations, rotations are rigid-body transformations that move objects without deformation. Every point on an object is rotated through the same angle. A straight line segment is rotated by applying the rotation equations 5-9 to each of the line endpoints and redrawing the line between the new endpoint positions. Polygons are rotated by displacing each vertex through the specified rotation angle and regenerating the polygon using the new vertices. Curved lines are rotated by repositioning the defining points and redrawing the curves. A circle or an ellipse, for instance, can be rotated about a noncentral axis by moving the center position through the arc that subtends the specified rotation angle. An ellipse can be rotated about its center coordinates by rotating the major and minor axes.

Scaling

A **scaling** transformation alters the size of an object. This operation can be carried out for polygons by multiplying the coordinate values (x, y) of each vertex by **scaling factors** s_x and s_y to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y \tag{5-10}$$

Scaling factor s_x scales objects in the x direction, while s_y scales in the y direction. The transformation equations 5-10 can also be written in the matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{5-11}$$

or

$$\mathbf{P'} = \mathbf{S} \cdot \mathbf{P} \tag{5-12}$$

where S is the 2 by 2 scaling matrix in Eq. 5-11.

Any positive numeric values can be assigned to the scaling factors s_x and s_y. Values less than 1 reduce the size of objects; values greater than 1 produce an enlargement. Specifying a value of 1 for both s_x and s_y leaves the size of objects unchanged. When s_x and s_y are assigned the same value, a **uniform scaling** is pro-

duced that maintains relative object proportions. Unequal values for s_x and s_y result in a **differential scaling** that is often used in design applications, where pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Fig. 5-6).

Figure 5-6
Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

Objects transformed with Eq. 5-11 are both scaled and repositioned. Scaling factors with values less than 1 move objects closer to the coordinate origin, while values greater than 1 move coordinate positions farther from the origin. Figure 5-7 illustrates scaling a line by assigning the value 0.5 to both s_x and s_y in Eq. 5-11. Both the line length and the distance from the origin are reduced by a factor of 1/2.

We can control the location of a scaled object by choosing a position, called the **fixed point**, that is to remain unchanged after the scaling transformation. Coordinates for the fixed point (x_f, y_f) can be chosen as one of the vertices, the object centroid, or any other position (Fig. 5-8). A polygon is then scaled relative to the fixed point by scaling the distance from each vertex to the fixed point. For a vertex with coordinates (x, y), the scaled coordinates (x', y') are calculated as

$$x' = x_f + (x - x_f)s_x, \qquad y' = y_f + (y - y_f)s_y \tag{5-13}$$

We can rewrite these scaling transformations to separate the multiplicative and additive terms:

$$\begin{aligned} x' &= x \cdot s_x + x_f(1 - s_x) \\ y' &= y \cdot s_y + y_f(1 - s_y) \end{aligned} \tag{5-14}$$

where the additive terms $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constant for all points in the object.

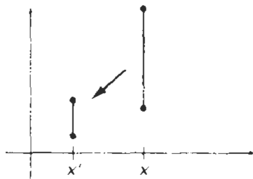


Figure 5-7
A line scaled with Eq 5-12 using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.

Including coordinates for a fixed point in the scaling equations is similar to including coordinates for a pivot point in the rotation equations. We can set up a column vector whose elements are the constant terms in Eqs. 5-14, then we add this column vector to the product **S** · **P** in Eq. 5-12. In the next section, we discuss a matrix formulation for the transformation equations that involves only matrix multiplication.

Polygons are scaled by applying transformations 5-14 to each vertex and then regenerating the polygon using the transformed vertices. Other objects are scaled by applying the scaling transformation equations to the parameters defining the objects. An ellipse in standard position is resized by scaling the semimajor and semiminor axes and redrawing the ellipse about the designated center coordinates. Uniform scaling of a circle is done by simply adjusting the radius. Then we redisplay the circle about the center coordinates using the transformed radius.

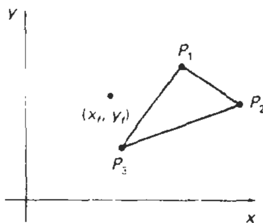


Figure 5-8
Scaling relative to a chosen fixed point (x_f, y_f). Distances from each polygon vertex to the fixed point are scaled by transformation equations 5-13.

5-2 MATRIX REPRESENTATIONS AND HOMOGENEOUS COORDINATES

Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an object to be translated and rotated at each increment of the motion. In design and picture construction applications,

we perform translations, rotations, and scalings to fit the picture components into their proper positions. Here we consider how the matrix representations discussed in the previous sections can be reformulated so that such transformation sequences can be efficiently processed.

We have seen in Section 5-1 that each of the basic transformations can be expressed in the general matrix form

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \tag{5-15}$$

with coordinate positions **P** and **P'** represented as column vectors. Matrix $\mathbf{M}_1$ is a 2 by 2 array containing multiplicative factors, and $\mathbf{M}_2$ is a two-element column matrix containing translational terms. For translation, $\mathbf{M}_1$ is the identity matrix. For rotation or scaling, $\mathbf{M}_2$ contains the translational terms associated with the pivot point or scaling fixed point. To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinates one step at a time. First, coordinate positions are scaled, then these scaled coordinates are rotated, and finally the rotated coordinates are translated. A more efficient approach would be to combine the transformations so that the final coordinate positions are obtained directly from the initial coordinates, thereby eliminating the calculation of intermediate coordinate values. To be able to do this, we need to reformulate Eq. 5-15 to eliminate the matrix addition associated with the translation terms in $\mathbf{M}_2$.

We can combine the multiplicative and translational terms for two-dimensional geometric transformations into a single matrix representation by expanding the 2 by 2 matrix representations to 3 by 3 matrices. This allows us to express all transformation equations as matrix multiplications, providing that we also expand the matrix representations for coordinate positions. To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position (x, y) with the **homogeneous coordinate** triple (x_h, y_h, h), where

$$x = \frac{x_h}{h}, \qquad y = \frac{y_h}{h} \tag{5-16}$$

Thus, a general homogeneous coordinate representation can also be written as $(h \cdot x, h \cdot y, h)$. For two-dimensional geometric transformations, we can choose the homogeneous parameter h to be any nonzero value. Thus, there is an infinite number of equivalent homogeneous representations for each coordinate point (x, y). A convenient choice is simply to set $h = 1$. Each two-dimensional position is then represented with homogeneous coordinates $(x, y, 1)$. Other values for parameter h are needed, for example, in matrix formulations of three-dimensional viewing transformations.

The term *homogeneous coordinates* is used in mathematics to refer to the effect of this representation on Cartesian equations. When a Cartesian point (x, y) is converted to a homogeneous representation (x_h, y_h, h), equations containing x and y, such as $f(x, y) = 0$, become homogeneous equations in the three parameters x_h, y_h, and h. This just means that if each of the three parameters is replaced by any value v times that parameter, the value v can be factored out of the equations.

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications. Coordinates are

represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices. For translation, we have

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5-17}$$

which we can write in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \tag{5-18}$$

with $\mathbf{T}(t_x, t_y)$ as the 3 by 3 translation matrix in Eq. 5-17. The inverse of the translation matrix is obtained by replacing the translation parameters t_x and t_y with their negatives: $-t_x$ and $-t_y$.

Similarly, rotation transformation equations about the coordinate origin are now written as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5-19}$$

or as

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \tag{5-20}$$

The rotation transformation operator $\mathbf{R}(\theta)$ is the 3 by 3 matrix in Eq. 5-19 with rotation parameter θ. We get the inverse rotation matrix when θ is replaced with $-\theta$.

Finally, a scaling transformation relative to the coordinate origin is now expressed as the matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5-21}$$

or

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \tag{5-22}$$

where $\mathbf{S}(s_x, s_y)$ is the 3 by 3 matrix in Eq. 5-21 with parameters s_x and s_y. Replacing these parameters with their multiplicative inverses ($1/s_x$ and $1/s_y$) yields the inverse scaling matrix.

Matrix representations are standard methods for implementing transformations in graphics systems. In many systems, rotation and scaling functions produce transformations with respect to the coordinate origin, as in Eqs. 5-19 and 5-21. Rotations and scalings relative to other reference positions are then handled as a succession of transformation operations. An alternate approach in a graphics package is to provide parameters in the transformation functions for the scaling fixed-point coordinates and the pivot-point coordinates General rotation and scaling matrices that include the pivot or fixed point are then set up directly without the need to invoke a succession of transformation functions.

5-3 COMPOSITE TRANSFORMATIONS

With the matrix representations of the previous section, we can set up a matrix for any sequence of transformations as a **composite transformation matrix** by calculating the matrix product of the individual transformations. Forming products of transformation matrices is often referred to as a **concatenation**, or **composition**, of matrices. For column-matrix representation of coordinate positions, we form composite transformations by multiplying matrices in order from right to left. That is, each successive transformation matrix premultiplies the product of the preceding transformation matrices.

Translations

If two successive translation vectors (t_{x1}, t_{y1}) and (t_{x2}, t_{y2}) are applied to a coordinate position **P**, the final transformed location **P'** is calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{T}(t_{x2}, t_{y2}) \cdot \{\mathbf{T}(t_{x1}, t_{y1}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1})\} \cdot \mathbf{P} \end{aligned} \qquad (5\text{-}23)$$

where **P** and **P'** are represented as homogeneous-coordinate column vectors. We can verify this result by calculating the matrix product for the two associative groupings. Also, the composite transformation matrix for this sequence of translations is

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \qquad (5\text{-}24)$$

or

$$\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1}) = \mathbf{T}(t_{x1} + t_{x2}, t_{y1} + t_{y2}) \qquad (5\text{-}25)$$

which demonstrates that two successive translations are additive.

Rotations

Two successive rotations applied to point **P** produce the transformed position

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned} \qquad (5\text{-}26)$$

By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \qquad (5\text{-}27)$$

so that the final rotated coordinates can be calculated with the composite rotation matrix as

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P} \qquad (5\text{-}28)$$

Scalings

Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix:

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-29}$$

or

$$\mathbf{S}(s_{x2}, s_{y2}) \cdot \mathbf{S}(s_{x1}, s_{y1}) = \mathbf{S}(s_{x1} \cdot s_{x2}, s_{y1} \cdot s_{y2}) \tag{5-30}$$

The resulting matrix in this case indicates that successive scaling operations are multiplicative. That is, if we were to triple the size of an object twice in succession, the final size would be nine times that of the original.

General Pivot-Point Rotation

With a graphics package that only provides a rotate function for revolving objects about the coordinate origin, we can generate rotations about any selected pivot point (x_r, y_r) by performing the following sequence of translate–rotate–translate operations:

1. Translate the object so that the pivot-point position is moved to the coordinate origin.
2. Rotate the object about the coordinate origin.
3. Translate the object so that the pivot point is returned to its original position.

This transformation sequence is illustrated in Fig. 5-9. The composite transforma-

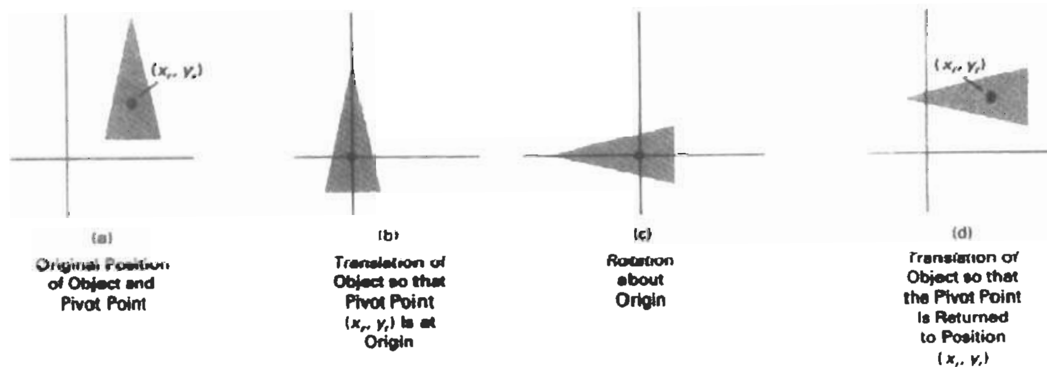


Figure 5-9
A transformation sequence for rotating an object about a specified pivot point using the rotation matrix $\mathbf{R}(\theta)$ of transformation 5-19.

tion matrix for this sequence is obtained with the concatenation

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \tag{5-31}$$

which can be expressed in the form

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta) \tag{5-32}$$

where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$. In general, a rotate function can be set up to accept parameters for pivot-point coordinates, as well as the rotation angle, and to generate automatically the rotation matrix of Eq. 5-31.

General Fixed-Point Scaling

Figure 5-10 illustrates a transformation sequence to produce scaling with respect to a selected fixed position (x_f, y_f) using a scaling function that can only scale relative to the coordinate origin.

1. Translate object so that the fixed point coincides with the coordinate origin.
2. Scale the object with respect to the coordinate origin.
3. Use the inverse translation of step 1 to return the object to its original position.

Concatenating the matrices for these three operations produces the required scaling matrix

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \tag{5-33}$$

or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y) \tag{5-34}$$

This transformation is automatically generated on systems that provide a scale function that accepts coordinates for the fixed point.

General Scaling Directions

Parameters s_x and s_y scale objects along the x and y directions. We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.

Suppose we want to apply scaling factors with values specified by parameters s_1 and s_2 in the directions shown in Fig. 5-11. To accomplish the scaling with-

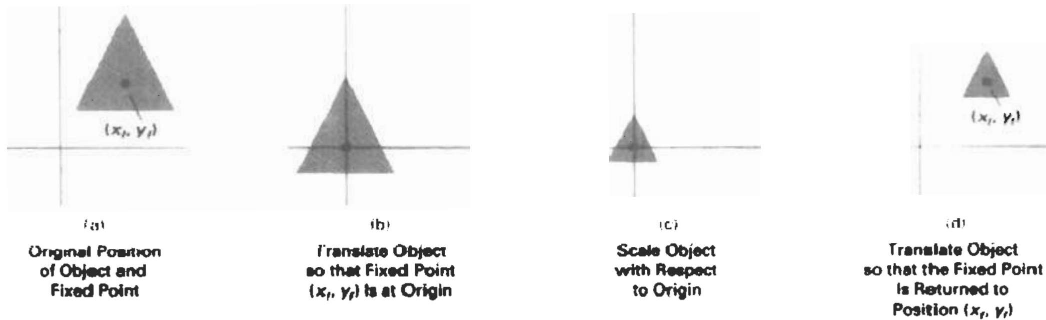


Figure 5-10
A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix $\mathbf{S}(s_x, s_y)$ of transformation 5-21.

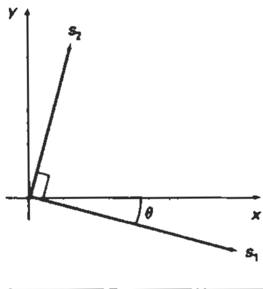


Figure 5-11
Scaling parameters s_1 and s_2 are to be applied in orthogonal directions defined by the angular displacement θ.

out changing the orientation of the object, we first perform a rotation so that the directions for s_1 and s_2 coincide with the x and y axes, respectively. Then the scaling transformation is applied, followed by an opposite rotation to return points to their original orientations. The composite matrix resulting from the product of these three transformations is

$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2\theta + s_2 \sin^2\theta & (s_2 - s_1)\cos\theta\sin\theta & 0 \\ (s_2 - s_1)\cos\theta\sin\theta & s_1\sin^2\theta + s_2\cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-35}$$

As an example of this scaling transformation, we turn a unit square into a parallelogram (Fig. 5-12) by stretching it along the diagonal from (0, 0) to (1, 1). We rotate the diagonal onto the y axis and double its length with the transformation parameters $\theta = 45°$, $s_1 = 1$, and $s_2 = 2$.

In Eq. 5-35, we assumed that scaling was to be performed relative to the origin. We could take this scaling operation one step further and concatenate the matrix with translation operators, so that the composite matrix would include parameters for the specification of a scaling fixed position.

Concatenation Properties

Matrix multiplication is associative. For any three matrices, **A**, **B**, and **C**, the matrix product **A** · **B** · **C** can be performed by first multiplying **A** and **B** or by first multiplying **B** and **C**:

$$\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C} = \mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}) \tag{5-36}$$

Therefore, we can evaluate matrix products using either a left-to-right or a right-to-left associative grouping.

On the other hand, transformation products may not be commutative: The matrix product **A** · **B** is not equal to **B** · **A**, in general. This means that if we want

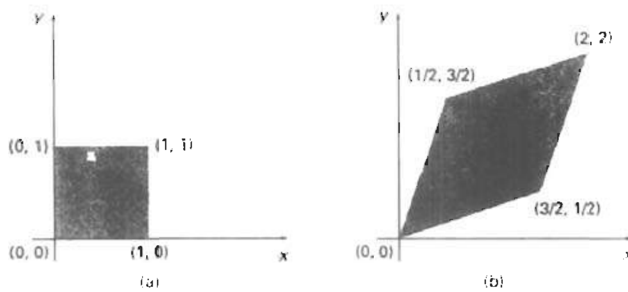


Figure 5-12
A square (a) is converted to a parallelogram (b) using the composite transformation matrix 5-35, with $s_1 = 1$, $s_2 = 2$, and $\theta = 45°$.

to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated (Fig. 5-13). For some special cases, such as a sequence of transformations all of the same kind, the multiplication of transformation matrices is commutative. As an example, two successive rotations could be performed in either order and the final position would be the same. This commutative property holds also for two successive translations or two successive scalings. Another commutative pair of operations is rotation and uniform scaling ($s_x = s_y$).

General Composite Transformations and Computational Efficiency

A general two-dimensional transformation, representing a combination of translations, rotations, and scalings, can be expressed as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5-37}$$

The four elements rs_{ij} are the multiplicative rotation-scaling terms in the transformation that involve only rotation angles and scaling factors. Elements trs_x and trs_y are the translational terms containing combinations of translation distances, pivot-point and fixed-point coordinates, and rotation angles and scaling parameters. For example, if an object is to be scaled and rotated about its centroid coordinates (x_c, y_c) and then translated, the values for the elements of the composite transformation matrix are

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y)$$
$$= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & x_c(1 - s_x \cos\theta) + y_c s_y \sin\theta + t_x \\ s_x \sin\theta & s_y \cos\theta & y_c(1 - s_y \cos\theta) - x_c s_x \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5-38}$$

Although matrix equation 5-37 requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

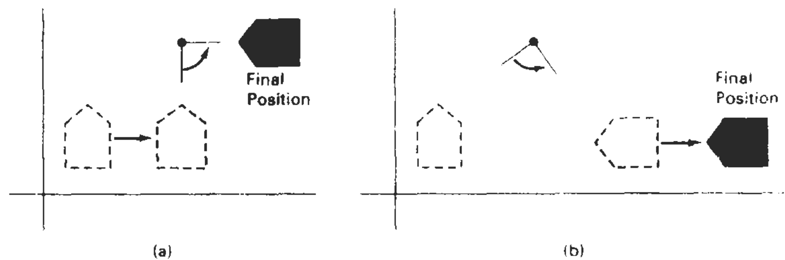


Figure 5-13
Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated, then rotated In (b), the object is rotated first, then translated.

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x, \qquad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y \tag{5-39}$$

Thus, we actually only need to perform four multiplications and four additions to transform coordinate positions. This is the maximum number of computations required for any transformation sequence, once the individual matrices have been concatenated and the elements of the composite matrix evaluated. Without concatenation, the individual transformations would be applied one at a time and the number of calculations could be significantly increased. An efficient implementation for the transformation operations, therefore, is to formulate transformation matrices, concatenate any transformation sequence, and calculate transformed coordinates using Eq. 5-39. On parallel systems, direct matrix multiplications with the composite transformation matrix of Eq. 5-37 can be equally efficient.

A general **rigid-body transformation matrix**, involving only translations and rotations, can be expressed in the form

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5-40}$$

where the four elements r_{ij} are the multiplicative rotation terms, and elements tr_x and tr_y are the translational terms. A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion** transformation. All angles and distances between coordinate positions are unchanged by the transformation. In addition, matrix 5-40 has the property that its upper-left 2-by-2 submatrix is an orthogonal matrix. This means that if we consider each row of the submatrix as a vector, then the two vectors (r_{xx}, r_{xy}) and (r_{yx}, r_{yy}) form an orthogonal set of unit vectors: Each vector has unit length

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \tag{5-41}$$

and the vectors are perpendicular (their dot product is 0):

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0 \tag{5-42}$$

Therefore, if these unit vectors are transformed by the rotation submatrix, (r_{xx}, r_{xy}) is converted to a unit vector along the x axis and (r_{yx}, r_{yy}) is transformed into a unit vector along the y axis of the coordinate system:

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{5-43}$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \tag{5-44}$$

As an example, the following rigid-body transformation first rotates an object through an angle θ about a pivot point (x_r, y_r) and then translates:

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5-45}$$

Here, orthogonal unit vectors in the upper-left 2-by-2 submatrix are $(\cos\theta, -\sin\theta)$ and $(\sin\theta, \cos\theta)$, and

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta \\ -\sin\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{5-46}$$

Similarly, unit vector $(\sin\theta, \cos\theta)$ is converted by the transformation matrix in Eq. 5-46 to the unit vector (0, 1) in the y direction.

The orthogonal property of rotation matrices is useful for constructing a rotation matrix when we know the final orientation of an object rather than the amount of angular rotation necessary to put the object into that position. Directions for the desired orientation of an object could be determined by the alignment of certain objects in a scene or by selected positions in the scene. Figure 5-14 shows an object that is to be aligned with the unit direction vectors **u'** and **v'**. Assuming that the original object orientation, as shown in Fig. 5-14(a), is aligned with the coordinate axes, we construct the desired transformation by assigning the elements of **u'** to the first row of the rotation matrix and the elements of **v'** to the second row. This can be a convenient method for obtaining the transformation matrix for rotation within a local (or "object") coordinate system when we know the final orientation vectors. A similar transformation is the conversion of object descriptions from one coordinate system to another, and in Section 5-5, we consider how to set up transformations to accomplish this coordinate conversion.

Since rotation calculations require trignometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations. In animations and other applications that involve many repeated transformations and small rotation angles, we can use approximations and iterative calculations to reduce computa-

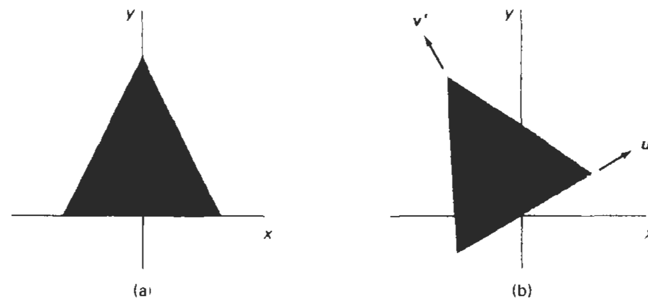


Figure 5-14
The rotation matrix for revolving an object from position (a) to position (b) can be constructed with the values of the unit orientation vectors **u'** and **v'** relative to the original orientation.

tions in the composite transformation equations. When the rotation angle is small, the trigonometric functions can be replaced with approximation values based on the first few terms of their power-series expansions. For small enough angles (less than 10°), cos θ is approximately 1 and sin θ has a value very close to the value of θ in radians. If we are rotating in small angular steps about the origin, for instance, we can set cos θ to 1 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated:

$$x' = x - y \sin \theta, \qquad y' = x \sin \theta + y \tag{5-47}$$

where sin θ is evaluated once for all steps, assuming the rotation angle does not change. The error introduced by this approximation at each step decreases as the rotation angle decreases. But even with small rotation angles, the accumulated error over many steps can become quite large. We can control the accumulated error by estimating the error in x' and y' at each step and resetting object positions when the error accumulation becomes too great.

Composite transformations often involve inverse matrix calculations. Transformation sequences for general scaling directions and for reflections and shears (Section 5-4), for example, can be described with inverse rotation components. As we have noted, the inverse matrix representations for the basic geometric transformations can be generated with simple procedures. An inverse translation matrix is obtained by changing the signs of the translation distances, and an inverse rotation matrix is obtained by performing a matrix transpose (or changing the sign of the sine terms). These operations are much simpler than direct inverse matrix calculations.

An implementation of composite transformations is given in the following procedure. Matrix **M** is initialized to the identity matrix. As each individual transformation is specified, it is concatenated with the total transformation matrix **M**. When all transformations have been specified, this composite transformation is applied to a given object. For this example, a polygon is scaled and rotated about a given reference point. Then the object is translated. Figure 5-15 shows the original and final positions of the polygon transformed by this sequence.

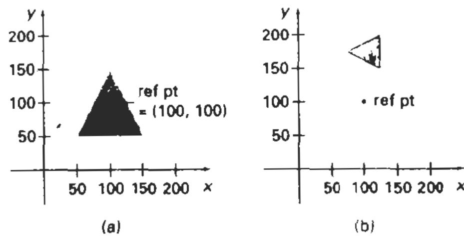


Figure 5-15
A polygon (a) is transformed into (b) by the composite operations in the following procedure.

```
#include <math.h>
#include "graphics.h"

typedef float Matrix3x3[3][3];
Matrix3x3 theMatrix;

void matrix3x3SetIdentity (Matrix3x3 m)
{
  int i,j;

  for (i=0; i<3; i++) for (j=0; j<3; j++) m[i][j] = (i == j);
}

/* Multiplies matrix a times b, putting result in b */
void matrix3x3PreMultiply (Matrix3x3 a, Matrix3x3 b)
{
  int r,c;
  Matrix3x3 tmp;

  for (r = 0; r < 3; r++)
    for (c = 0; c < 3; c++)
      tmp[r][c] =
        a[r][0]*b[0][c] + a[r][1]*b[1][c] + a[r][2]*b[2][c];

  for (r = 0; r < 3; r++)
    for (c = 0; c < 3; c++)
      b[r][c] - tmp[r][c];
}

void translate2 (int tx, int ty)
{
  Matrix3x3 m;

  matrix3x3SetIdentity (m);
  m[0][2] = tx;
  m[1][2] = ty;
  matrix3x3PreMultiply (m, theMatrix);
```

```
}

void scale2 (float sx, float sy, wcPt2 refpt)
{
  Matrix3x3 m;

  matrix3x3SetIdentity (m);
  m[0][0] = sx;
  m[0][2] = (1 - sx) * refpt.x;
  m[1][1] = sy;
  m[1][2] = (1 - sy) * refpt.y;
  matrix3x3PreMultiply (m, theMatrix);
}

void rotate2 (float a, wcPt2 refPt)
{
  Matrix3x3 m;

  matrix3x3SetIdentity (m);
  a = pToRadians (a);
  m[0][0] = cosf (a);
  m[0][1] = -sinf (a);
  m[0][2] = refPt.x * (1 - cosf (a)) + refPt.y * sinf (a);
  m[1][0] = sinf (a);
  m[1][1] = cosf (a);
  m[1][2] = refPt.y * (1 - cosf (a)) - refPt.x * sinf (a);
  matrix3x3PreMultiply (m, theMatrix);
}

void transformPoints2 (int npts, wcPt2 *pts)
{
  int k;
  float tmp;

  for (k = 0; k < npts; k++) {
    tmp = theMatrix[0][0] * pts[k].x + theMatrix[0][1] *
      pts[k].y + theMatrix[0][2];
    pts[k].y = theMatrix[1][0] * pts[k].x + theMatrix[1][1] *
      pts[k].y + theMatrix[1][2];
    pts[k].x = tmp;
  }
}

void main (int argc, char ** argv)
{
  wcPt2 pts[3] = { 50.0, 50.0, 150.0, 50.0, 100.0, 150.0};
  wcPt2 refPt = {100.0, 100.0};
  long windowID = openGraphics (*argv, 200, 350);

  setBackground (WHITE);
  setColor (BLUE);
  pFillArea (3, pts);
  matrix3x3SetIdentity (theMatrix);
  scale2 (0.5, 0.5, refPt);
  rotate2 (90.0, refPt);
  translate2 (0, 150);
  transformPoints2 (3, pts),
  pFillArea (3,pts);
  sleep (10);
  closeGraphics (windowID);
}
```

5-4 OTHER TRANSFORMATIONS

Basic transformations such as translation, rotation, and scaling are included in most graphics packages. Some packages provide a few additional transformations that are useful in certain applications. Two such transformations are reflection and shear.

Reflection

A **reflection** is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an **axis of reflection** by rotating the object 180° about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane. When the reflection axis is a line in the xy plane, the rotation path about this axis is in a plane perpendicular to the xy plane. For reflection axes that are perpendicular to the xy plane, the rotation path is in the xy plane. Following are examples of some common reflections.

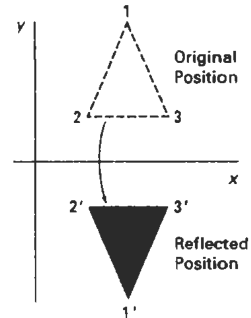


Figure 5-16
Reflection of an object about the x axis.

Reflection about the line $y = 0$, the x axis, is accomplished with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-48}$$

This transformation keeps x values the same, but "flips" the y values of coordinate positions. The resulting orientation of an object after it has been reflected about the x axis is shown in Fig. 5-16. To envision the rotation transformation path for this reflection, we can think of the flat object moving out of the xy plane and rotating 180° through three-dimensional space about the x axis and back into the xy plane on the other side of the x axis.

A reflection about the y axis flips x coordinates while keeping y coordinates the same. The matrix for this transformation is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-49}$$

Figure 5-17 illustrates the change in position of an object that has been reflected about the line $x = 0$. The equivalent rotation in this case is 180° through three-dimensional space about the y axis.

We flip both the x and y coordinates of a point by reflecting relative to an axis that is perpendicular to the xy plane and that passes through the coordinate origin. This transformation, referred to as a reflection relative to the coordinate origin, has the matrix representation:

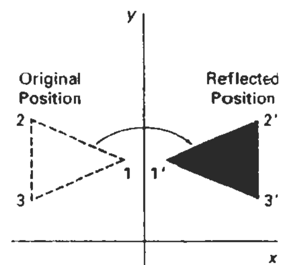


Figure 5-17
Reflection of an object about the y axis.

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-50}$$

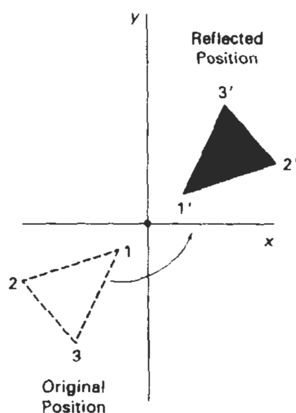


Figure 5-18
Reflection of an object relative to an axis perpendicular to the xy plane and passing through the coordinate origin.

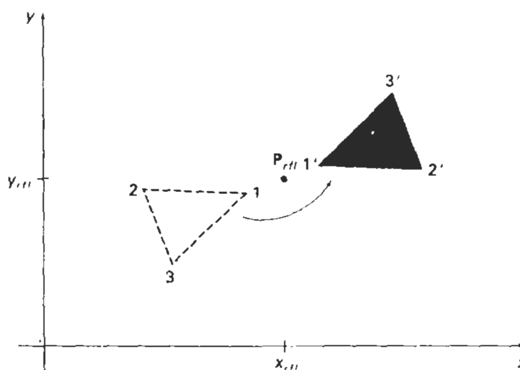


Figure 5-19
Reflection of an object relative to an axis perpendicular to the xy plane and passing through point $\mathbf{P}_{rfl}$.

An example of reflection about the origin is shown in Fig. 5-18. The reflection matrix 5-50 is the rotation matrix $\mathbf{R}(\theta)$ with $\theta = 180°$. We are simply rotating the object in the xy plane half a revolution about the origin.

Reflection 5-50 can be generalized to any reflection point in the xy plane (Fig. 5-19). This reflection is the same as a 180° rotation in the xy plane using the reflection point as the pivot point.

If we chose the reflection axis as the diagonal line $y = x$ (Fig. 5-20), the reflection matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-51}$$

We can derive this matrix by concatenating a sequence of rotation and coordinate-axis reflection matrices. One possible sequence is shown in Fig. 5-21. Here, we first perform a clockwise rotation through a 45° angle, which rotates the line $y = x$ onto the x axis. Next, we perform a reflection with respect to the x axis. The final step is to rotate the line $y = x$ back to its original position with a counterclockwise rotation through 45°. An equivalent sequence of transformations is first to reflect the object about the x axis, and then to rotate counterclockwise 90°.

To obtain a transformation matrix for reflection about the diagonal $y = -x$, we could concatenate matrices for the transformation sequence: (1) clockwise rotation by 45°, (2) reflection about the y axis, and (3) counterclockwise rotation by 45°. The resulting transformation matrix is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-52}$$

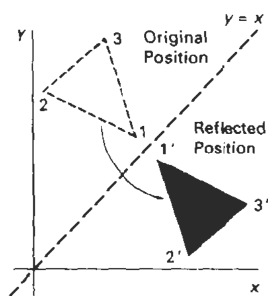


Figure 5-20
Reflection of an object with respect to the line $y = x$.

Figure 5-22 shows the original and final positions for an object transformed with this reflection matrix.

Reflections about any line $y = mx + b$ in the xy plane can be accomplished with a combination of translate-rotate-reflect transformations. In general, we first translate the line so that it passes through the origin. Then we can rotate the line onto one of the coordinate axes and reflect about that axis. Finally, we restore the line to its original position with the inverse rotation and translation transformations.

We can implement reflections with respect to the coordinate axes or coordinate origin as scaling transformations with negative scaling factors. Also, elements of the reflection matrix can be set to values other than ± 1. Values whose magnitudes are greater than 1 shift the mirror image farther from the reflection axis, and values with magnitudes less than 1 bring the mirror image closer to the reflection axis.

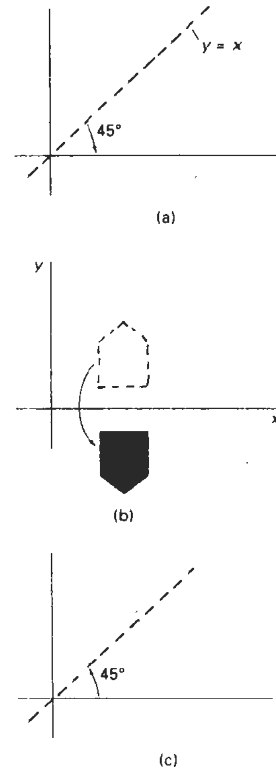


Figure 5-21
Sequence of transformations to produce reflection about the line $y = x$: (a) clockwise rotation of 45°, (b) reflection about the x axis, and (c) counterclockwise rotation by 45°.

Shear

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear**. Two common shearing transformations are those that shift coordinate x values and those that shift y values.

An x-direction shear relative to the x axis is produced with the transformation matrix

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-53}$$

which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \qquad y' = y \tag{5-54}$$

Any real number can be assigned to the shear parameter sh_x. A coordinate position (x, y) is then shifted horizontally by an amount proportional to its distance (y value) from the x axis ($y = 0$). Setting sh_x to 2, for example, changes the square in Fig. 5-23 into a parallelogram. Negative values for sh_x shift coordinate positions to the left.

We can generate x-direction shears relative to other reference lines with

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-55}$$

with coordinate positions transformed as

$$x' = x + sh_x(y - y_{ref}), \qquad y' = y \tag{5-56}$$

An example of this shearing transformation is given in Fig. 5-24 for a shear parameter value of 1/2 relative to the line $y_{ref} = -1$.

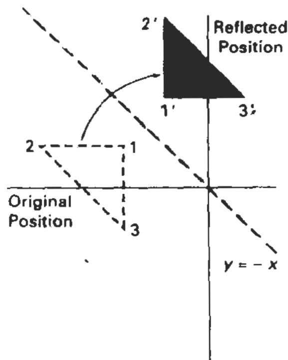


Figure 5-22
Reflection with respect to the line $y = -x$.

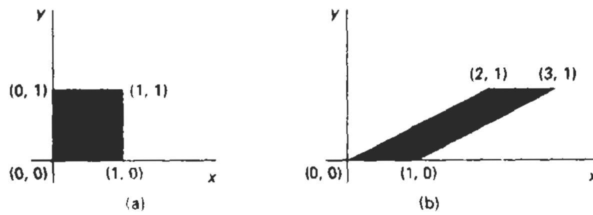


Figure 5-23
A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 5-53 with $sh_x = 2$.

A y-direction shear relative to the line $x = x_{ref}$ is generated with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix} \tag{5-57}$$

which generates transformed coordinate positions

$$x' = x, \qquad y' = sh_y(x - x_{ref}) + y \tag{5-58}$$

This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{ref}$. Figure 5-25 illustrates the conversion of a square into a parallelogram with $sh_y = 1/2$ and $x_{ref} = -1$.

Shearing operations can be expressed as sequences of basic transformations. The x-direction shear matrix 5-53, for example, can be written as a composite transformation involving a series of rotation and scaling matrices that would scale the unit square of Fig. 5-23 along its diagonal, while maintaining the original lengths and orientations of edges parallel to the x axis. Shifts in the positions of objects relative to shearing reference lines are equivalent to translations.

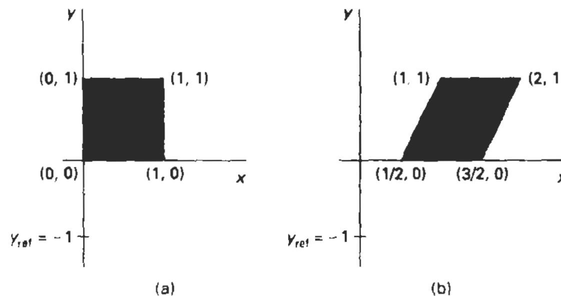


Figure 5-24
A unit square (a) is transformed to a shifted parallelogram (b) with $sh_x = 1/2$ and $y_{ref} = -1$ in the shear matrix 5-55.

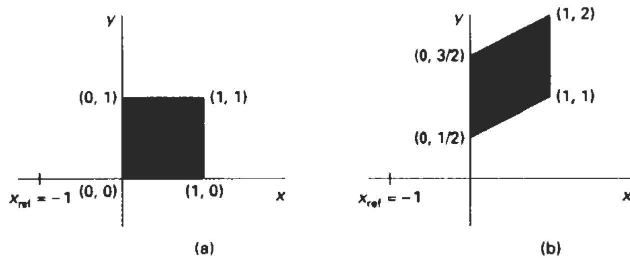


Figure 5-25
A unit square (a) is turned into a shifted parallelogram (b) with parameter values $sh_y = 1/2$ and $x_{ref} = -1$ in the y-direction using shearing transformation 5-57.

5-5 TRANSFORMATIONS BETWEEN COORDINATE SYSTEMS

Graphics applications often require the transformation of object descriptions from one coordinate system to another. Sometimes objects are described in non-Cartesian reference frames that take advantage of object symmetries. Coordinate descriptions in these systems must then be converted to Cartesian device coordinates for display. Some examples of two-dimensional non-Cartesian systems are polar coordinates, elliptical coordinates, and parabolic coordinates. In other cases, we need to transform between two Cartesian systems. For modeling and design applications, individual objects may be defined in their own local Cartesian references, and the local coordinates must then be transformed to position the objects within the overall scene coordinate system. A facility management program for office layouts, for instance, has individual coordinate reference descriptions for chairs and tables and other furniture that can be placed into a floor plan, with multiple copies of the chairs and other items in different positions. In other applications, we may simply want to reorient the coordinate reference for displaying a scene. Relationships between Cartesian reference systems and some common non-Cartesian systems are given in Appendix A. Here, we consider transformations between two Cartesian frames of reference.

Figure 5-26 shows two Cartesian systems, with the coordinate origins at (0, 0) and (x_0, y_0) and with an orientation angle θ between the x and x' axes. To transform object descriptions from xy coordinates to $x'y'$ coordinates, we need to set up a transformation that superimposes the $x'y'$ axes onto the xy axes. This is done in two steps:

1. Translate so that the origin (x_0, y_0) of the $x'y'$ system is moved to the origin of the xy system.
2. Rotate the x' axis onto the x axis.

Translation of the coordinate origin is expressed with the matrix operation

$$T(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-59}$$

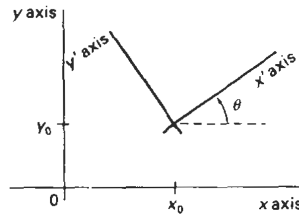


Figure 5-26
A Cartesian $x'y'$ system positioned at (x_0, y_0) with orientation θ in an xy Cartesian system.

and the orientation of the two systems after the translation operation would appear as in Fig. 5-27. To get the axes of the two systems into coincidence, we then perform the clockwise rotation

$$R(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-60}$$

Concatinating these two transformations matrices gives us the complete composite matrix for transforming object descriptions from the xy system to the $x'y'$ system:

$$\mathbf{M}_{xy,x'y'} = \mathbf{R}(-\theta) \cdot \mathbf{T}(-x_0, -y_0) \tag{5-61}$$

An alternate method for giving the orientation of the second coordinate system is to specify a vector **V** that indicates the direction for the positive y' axis, as shown in Fig. 5-28. Vector **V** is specified as a point in the xy reference frame relative to the origin of the xy system. A unit vector in the y' direction can then be obtained as

$$\mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|} = (v_x, v_y) \tag{5-62}$$

And we obtain the unit vector **u** along the x' axis by rotating **v** 90° clockwise:

$$\begin{aligned} \mathbf{u} &= (v_y, -v_x) \\ &= (u_x, u_y) \end{aligned} \tag{5-63}$$

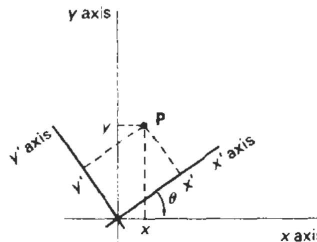


Figure 5-27
Position of the reference frames shown in Fig. 5-26 after translating the origin of the $x'y'$ system to the coordinate origin of the xy system.

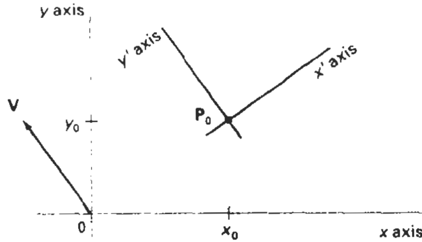


Figure 5-28
Cartesian system $x'y'$ with origin at $\mathbf{P}_0 = (x_0, y_0)$ and y' axis parallel to vector **V**.

In Section 5-3, we noted that the elements of any rotation matrix could be expressed as elements of a set of orthogonal unit vectors. Therefore, the matrix to rotate the $x'y'$ system into coincidence with the xy system can be written as

$$R = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5-64}$$

As an example, suppose we choose the orientation for the y' axis as $\mathbf{V} = (-1, 0)$, then the x' axis is in the positive y direction and the rotation transformation matrix is

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

Equivalently, we can obtain this rotation matrix from 5-60 by setting the orientation angle as $\theta = 90°$.

In an interactive application, it may be more convenient to choose the direction for **V** relative to position $\mathbf{P}_0$ than it is to specify it relative to the xy-coordinate origin. Unit vectors **u** and **v** would then be oriented as shown in Fig. 5-29. The components of **v** are now calculated as

$$\mathbf{v} = \frac{\mathbf{P}_1 - \mathbf{P}_0}{|\mathbf{P}_1 - \mathbf{P}_0|} \tag{5-65}$$

and **u** is obtained as the perpendicular to **v** that forms a right-handed Cartesian system.

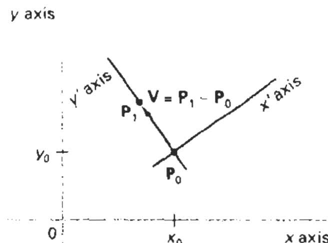


Figure 5-29
A Cartesian $x'y'$ system defined with two coordinate positions, $\mathbf{P}_0$ and $\mathbf{P}_1$, within an xy reference frame.

5-6 AFFINE TRANSFORMATIONS

A coordinate transformation of the form

$$x' = a_{xx}x + a_{xy}y + b_x, \qquad y' = a_{yx}x + a_{yy}y + b_y \tag{5-66}$$

is called a two-dimensional **affine transformation**. Each of the transformed coordinates x' and y' is a linear function of the original coordinates x and y, and parameters a_{ij} and b_k are constants determined by the transformation type. Affine transformations have the general properties that parallel lines are transformed into parallel lines and finite points map to finite points.

Translation, rotation, scaling, reflection, and shear are examples of two-dimensional affine transformations. Any general two-dimensional affine transformation can always be expressed as a composition of these five transformations. Another affine transformation is the conversion of coordinate descriptions from one reference system to another, which can be described as a combination of translation and rotation. An affine transformation involving only rotation, translation, and reflection preserves angles and lengths, as well as parallel lines. For these three transformations, the lengths and angle between two lines remains the same after the transformation.

5-7 TRANSFORMATION FUNCTIONS

Graphics packages can be structured so that separate commands are provided to a user for each of the basic transformation operations, as in `procedure transformObject`. A composite transformation is then set up by referencing individual functions in the order required for the transformation sequence. An alternate formulation is to provide users with a single transformation function that includes parameters for each of the basic transformations. The output of this function is the composite transformation matrix for the specified parameter values. Both options are useful. Separate functions are convenient for simple transformation operations, and a composite function can provide an expedient method for specifying complex transformation sequences.

The PHIGS library provides users with both options. Individual commands for generating the basic transformation matrices are

```
translate (translateVector, matrixTranslate)
rotate (theta, matrixRotate)
scale (scaleVector, matrixScale)
```

Each of these functions produces a 3 by 3 transformation matrix that can then be used to transform coordinate positions expressed as homogeneous column vectors. Parameter `translateVector` is a pointer to the pair of translation distances t_x and t_y. Similarly, parameter `scaleVector` specifies the pair of scaling values s_x and s_y. Rotate and scale matrices (`matrixTranslate` and `matrixScale`) transform with respect to the coordinate origin.

We concatenate transformation matrices that have been previously set up with the function

```
composeMatrix (matrix2, matrix1, matrixOut)
```

where elements of the composite output matrix are calculated by postmultiplying `matrix2` by `matrix1`. A composite transformation matrix to perform a combination scaling, rotation, and translation is produced with the function

```
buildTransformationMatrix (referencePoint, translateVector,
                           theta, scaleVector, matrix)
```

Rotation and scaling are carried out with respect to the coordinate position specified by parameter `referencePoint`. The order for the transformation sequence is assumed to be (1) scale, (2) rotate, and (3) translate, with the elements for the composite transformation stored in parameter `matrix`. We can use this function to generate a single transformation matrix or a composite matrix for two or three transformations (in the order stated). We could generate a translation matrix by setting `scaleVector` = (1, 1), `theta` = 0, and assigning x and y shift values to parameter `translateVector`. Any coordinate values could be assigned to parameter `referencePoint`, since the transformation calculations are unaffected by this parameter when no scaling or rotation takes place. But if we only want to set up a translation matrix, we can use function `translate` and simply specify the translation vector. A rotation or scaling transformation matrix is specified by setting `translateVector` = (0, 0) and assigning appropriate values to parameters `referencePoint`, `theta`, and `scaleVector`. To obtain a rotation matrix, we set `scaleVector` = (1, 1); and for scaling only, we set `theta` = 0. If we want to rotate or scale with respect to the coordinate origin, it is simpler to set up the matrix using either the `rotate` or `scale` function.

Since the function `buildTransformationMatrix` always generates the transformation sequence in the order (1) scale, (2) rotate, and (3) translate, the following function is provided to allow specification of other sequences:

```
composeTransformationMatrix (matrixIn, referencePoint,
                   translateVector, theta, scaleVector, matrixOut)
```

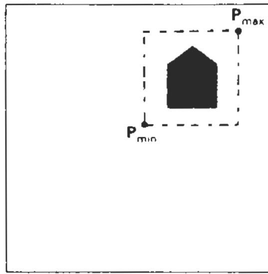
We can use this function in combination with the `buildTransformationMatrix` function or with any of the other matrix-construction functions to compose any transformation sequence. For example, we could set up a scale matrix about a fixed point with the `buildTransformationMatrix` function, then we could use the `composeTransformationMatrix` function to concatenate this scale matrix with a rotation about a specified pivot point. The composite rotate-scale sequence is then stored in `matrixOut`.

After we have set up a transformation matrix, we can apply the matrix to individual coordinate positions of an object with the function

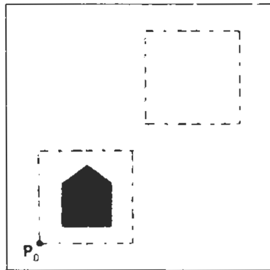
```
transformPoint (inPoint, matrix, outPoint)
```

where parameter `inPoint` gives the initial xy-coordinate position of an object point, and parameter `outPoint` contains the corresponding transformed coordinates. Additional functions, discussed in Chapter 7, are available for performing two-dimensional modeling transformations.

5-8
RASTER METHODS FOR TRANSFORMATIONS



(a)



(b)

Figure 5-30
Translating an object from screen position (a) to position (b) by moving a rectangular block of pixel values. Coordinate positions $\mathbf{P}_{min}$ and $\mathbf{P}_{max}$ specify the limits of the rectangular block to be moved, and $\mathbf{P}_0$ is the destination reference position.

The particular capabilities of raster systems suggest an alternate method for transforming objects. Raster systems store picture information as pixel patterns in the frame buffer. Therefore, some simple transformations can be carried out rapidly by simply moving rectangular arrays of stored pixel values from one location to another within the frame buffer. Few arithmetic operations are needed, so the pixel transformations are particularly efficient.

Raster functions that manipulate rectangular pixel arrays are generally referred to as **raster ops**. Moving a block of pixels from one location to another is also called a **block transfer** of pixel values. On a bilevel system, this operation is called a **bitBlt (bit-block transfer),** particularly when the function is hardware implemented. The term **pixBlt** is sometimes used for block transfers on multilevel systems (multiple bits per pixel).

Figure 5-30 illustrates translation performed as a block transfer of a raster area. All bit settings in the rectangular area shown are copied as a block into another part of the raster. We accomplish this translation by first reading pixel intensities from a specified rectangular area of a raster into an array, then we copy the array back into the raster at the new location. The original object could be erased by filling its rectangular area with the background intensity (assuming the object does not overlap other objects in the scene).

Typical raster functions often provided in graphics packages are:

- *copy* - move a pixel block from one raster area to another.
- *read* - save a pixel block in a designated array.
- *write* - transfer a pixel array to a position in the frame buffer.

Some implementations provide options for combining pixel values. In *replace* mode, pixel values are simply transfered to the destination positions. Other options for combining pixel values include Boolean operations (*and*, *or*, and *exclusive or*) and binary arithmetic operations. With the *exclusive or* mode, two successive copies of a block to the same raster area restores the values that were originally present in that area. This technique can be used to move an object across a scene without destroying the background. Another option for adjusting pixel values is to combine the source pixels with a specified mask. This allows only selected positions within a block to be transferred or shaded by the patterns defined in the mask.

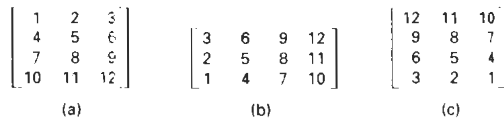


Figure 5-31
Rotating an array of pixel values. The original array orientation is shown in (a), the array orientation after a 90° counterclockwise rotation is shown in (b), and the array orientation after a 180° rotation is shown in (c).

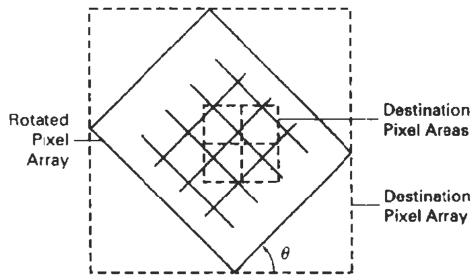


Figure 5-32
A raster rotation for a rectangular block of pixels is accomplished by mapping the destination pixel areas onto the rotated block.

Rotations in 90-degree increments are easily accomplished with block transfers. We can rotate an object 90° counterclockwise by first reversing the pixel values in each row of the array, then we interchange rows and columns. A 180° rotation is obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows. Figure 5-31 demonstrates the array manipulations necessary to rotate a pixel block by 90° and by 180°.

For array rotations that are not multiples of 90°, we must perform more computations. The general procedure is illustrated in Fig. 5-32. Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated. An intensity for the destination pixel is then computed by averaging the intensities of the overlapped source pixels, weighted by their percentage of area overlap.

Raster scaling of a block of pixels is analogous to the cell-array mapping discussed in Section 3-13. We scale the pixel areas in the original block using specified values for s_x and s_y and map the scaled rectangle onto a set of destination pixels. The intensity of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas (Fig. 5-33).

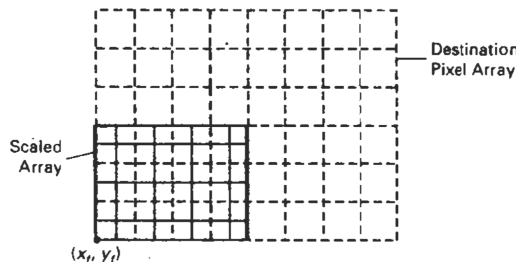


Figure 5-33
Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point (x_f, y_f).

SUMMARY

The basic geometric transformations are translation, rotation, and scaling. Translation moves an object in a straight-line path from one position to another. Rotation moves an object from one position to another in a circular path around a specified pivot point (rotation point). Scaling changes the dimensions of an object relative to a specified fixed point.

We can express two-dimensional geometric transformations as 3 by 3 matrix operators, so that sequences of transformations can be concatenated into a single composite matrix. This is an efficient formulation, since it allows us to reduce computations by applying the composite matrix to the initial coordinate positions of an object to obtain the final transformed positions. To do this, we also need to express two-dimensional coordinate positions as three-element column or row matrices. We choose a column-matrix representation for coordinate points because this is the standard mathematical convention and because many graphics packages also follow this convention. For two-dimensional transformations, coordinate positions are then represented with three-element homogeneous coordinates with the third (homogeneous) coordinate assigned the value 1.

Composite transformations are formed as multiplications of any combination of translation, rotation, and scaling matrices. We can use combinations of translation and rotation for animation applications, and we can use combinations of rotation and scaling to scale objects in any specified direction. In general, matrix multiplications are not commutative. We obtain different results, for example, if we change the order of a translate-rotate sequence. A transformation sequence involving only translations and rotations is a rigid-body transformation, since angles and distances are unchanged. Also, the upper-left submatrix of a rigid-body transformation is an orthogonal matrix. Thus, rotation matrices can be formed by setting the upper-left 2-by-2 submatrix equal to the elements of two orthogonal unit vectors. Computations in rotational transformations can be reduced by using approximations for the sine and cosine functions when the rotation angle is small. Over many rotational steps, however, the approximation error can accumulate to a significant value.

Other transformations include reflections and shears. Reflections are transformations that rotate an object 180° about a reflection axis. This produces a mirror image of the object with respect to that axis. When the reflection axis is perpendicular to the xy plane, the reflection is obtained as a rotation in the xy plane. When the reflection axis is in the xy plane, the reflection is obtained as a rotation in a plane that is perpendicular to the xy plane. Shear transformations distort the shape of an object by shifting x or y coordinate values by an amount proportional to the coordinate distance from a shear reference line.

Transformations between Cartesian coordinate systems are accomplished with a sequence of translate-rotate transformations. One way to specify a new coordinate reference frame is to give the position of the new coordinate origin and the direction of the new y axis. The direction of the new x axis is then obtained by rotating the y direction vector 90° clockwise. Coordinate descriptions of objects in the old reference frame are transferred to the new reference with the transformation matrix that superimposes the new coordinate axes onto the old coordinate axes. This transformation matrix can be calculated as the concatentation of a translation that moves the new origin to the old coordinate origin and a rotation to align the two sets of axes. The rotation matrix is obtained from unit vectors in the x and y directions for the new system.

Two-dimensional geometric transformations are affine transformations. That is, they can be expressed as a linear function of coordinates x and y. Affine transformations transform parallel lines to parallel lines and transform finite points to finite points. Geometric transformations that do not involve scaling or shear also preserve angles and lengths.

Transformation functions in graphics packages are usually provided only for translation, rotation, and scaling. These functions include individual procedures for creating a translate, rotate, or scale matrix, and functions for generating a composite matrix given the parameters for a transformation sequence.

Fast raster transformations can be performed by moving blocks of pixels. This avoids calculating transformed coordinates for an object and applying scan-conversion routines to display the object at the new position. Three common raster operations (bitBlts or pixBlts) are copy, read, and write. When a block of pixels is moved to a new position in the frame buffer, we can simply replace the old pixel values or we can combine the pixel values using Boolean or arithmetic operations. Raster translations are carried out by copying a pixel block to a new location in the frame buffer. Raster rotations in multiples of 90° are obtained by manipulating row and column positions of the pixel values in a block. Other rotations are performed by first mapping rotated pixel areas onto destination positions in the frame buffer, then calculating overlap areas. Scaling in raster transformations is also accomplished by mapping transformed pixel areas to the frame-buffer destination positions.

REFERENCES

For additional information on homogeneous coordinates in computer graphics, see Blinn (1977 and 1978).

Transformation functions in PHIGS are discussed in Hopgood and Duce (1991), Howard et al. (1991), Gaskins (1992), and Blake (1993). For information on GKS transformation functions, see Hopgood et al. (1983) and Enderle, Kansy, and Pfaff (1984).

EXERCISES

5-1 Write a program to continuously rotate an object about a pivot point. Small angles are to be used for each successive rotation, and approximations to the sine and cosine functions are to be used to speed up the calculations. The rotation angle for each step is to be chosen so that the object makes one complete revolution in less than 30 seconds. To avoid accumulation of coordinate errors, reset the original coordinate values for the object at the start of each new revolution.

5-2 Show that the composition of two rotations is additive by concatinating the matrix representations for $\mathbf{R}(\theta_1)$ and $\mathbf{R}(\theta_2)$ to obtain

$$\mathbf{R}(\theta_1) \cdot \mathbf{R}(\theta_2) = \mathbf{R}(\theta_1 + \theta_2)$$

5-3 Write a set of procedures to implement the `buildTransformationMatrix` and the `composeTransformationMatrix` functions to produce a composite transformation matrix for any set of input transformation parameters.

5-4 Write a program that applies any specified sequence of transformations to a displayed object. The program is to be designed so that a user selects the transformation sequence and associated parameters from displayed menus, and the composite transfor-

mation is then calculated and used to transform the object. Display the original object and the transformed object in different colors or different fill patterns.

5-5 Modify the transformation matrix (5-35), for scaling in an arbitrary direction, to include coordinates for any specified scaling fixed point (x_f, y_f).

5-6 Prove that the multiplication of transformation matrices for each of the following sequence of operations is commutative:
(a) Two successive rotations.
(b) Two successive translations.
(c) Two successive scalings.

5-7 Prove that a uniform scaling ($s_x = s_y$) and a rotation form a commutative pair of operations but that, in general, scaling and rotation are not commutative operations.

5-8 Multiply the individual scale, rotate, and translate matrices in Eq. 5-38 to verify the elements in the composite transformation matrix.

5-9 Show that transformation matrix (5-51), for a reflection about the line $y = x$, is equivalent to a reflection relative to the x axis followed by a counterclockwise rotation of 90°.

5-10 Show that transformation matrix (5-52), for a reflection about the line $y = -x$, is equivalent to a reflection relative to the y axis followed by a counterclockwise rotation of 90°.

5-11 Show that two successive reflections about either of the coordinate axes is equivalent to a single rotation about the coordinate origin.

5-12 Determine the form of the transformation matrix for a reflection about an arbitrary line with equation $y = mx + b$.

5-13 Show that two successive reflections about any line passing through the coordinate origin is equivalent to a single rotation about the origin

5-14 Determine a sequence of basic transformations that are equivalent to the x-direction shearing matrix (5-53).

5-15 Determine a sequence of basic transformations that are equivalent to the y-direction shearing matrix (5-57)

5-10 Set up a shearing procedure to display italic characters, given a vector font definition. That is, all character shapes in this font are defined with straight-line segments, and italic characters are formed with shearing transformations. Determine an appropriate value for the shear parameter by comparing italics and plain text in some available font. Define a simple vector font for input to your routine.

5-17 Derive the following equations for transforming a coordinate point $\mathbf{P} = (x, y)$ in one Cartesian system to the coordinate values (x', y') in another Cartesian system that is rotated by an angle θ, as in Fig. 5-27. Project point $\mathbf{P}$ onto each of the four axes and analyse the resulting right triangles.

$$x' = x\cos\theta + y\sin\theta, \qquad y' = -x\sin\theta + y\cos\theta$$

5-18 Write a procedure to compute the elements of the matrix for transforming object descriptions from one Cartesian coordinate system to another. The second coordinate system is to be defined with an origin point $\mathbf{P}_0$ and a vector $\mathbf{V}$ that gives the direction for the positive y' axis of this system.

5-19 Set up procedures for implementing a block transfer of a rectangular area of a frame buffer, using one function to read the area into an array and another function to copy the array into the designated transfer area.

5-20 Determine the results of performing two successive block transfers into the same area of a frame buffer using the various Boolean operations.

5-21 What are the results of performing two successive block transfers into the same area of a frame buffer using the binary arithmetic operations?

5-22 Implement a routine to perform block transfers in a frame buffer using any specified Boolean operation or a replacement (copy) operation

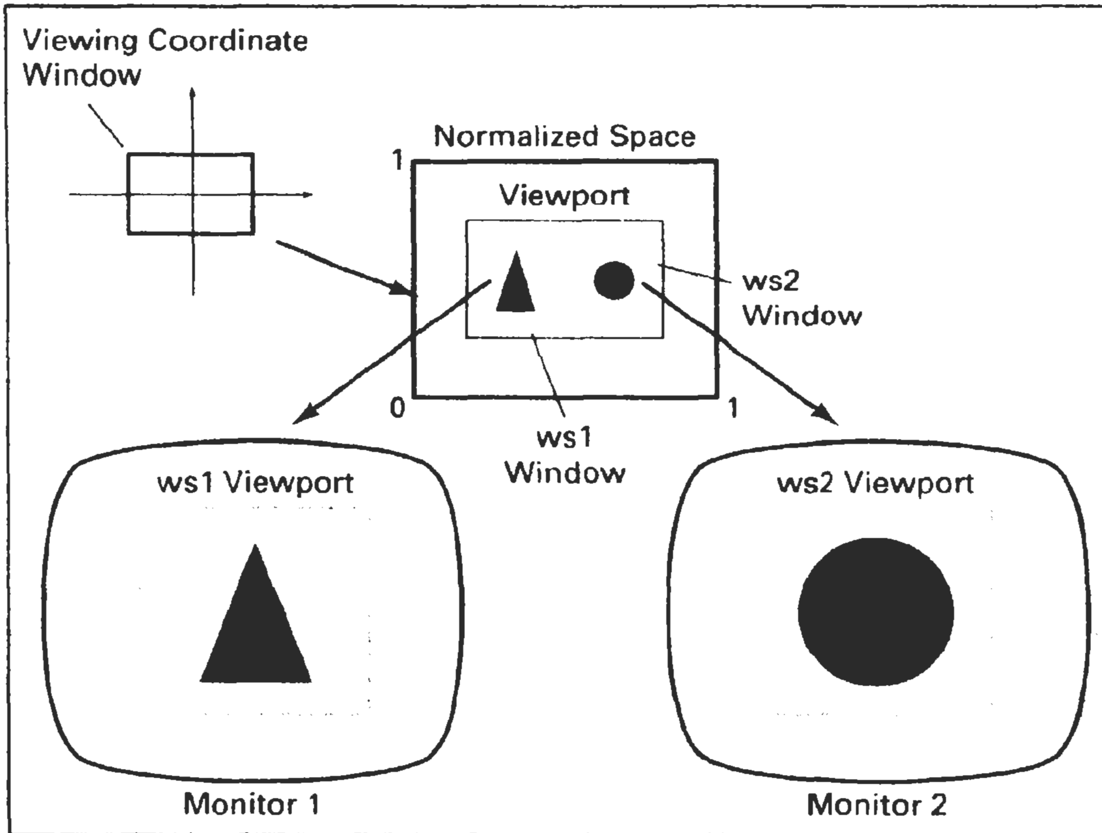
5-23 Write a routine to implement rotations in increments of 90° in frame-buffer block transfers.

5-24 Write a routine to implement rotations by any specified angle in a frame-buffer block transfer.

5-25 Write a routine to implement scaling as a raster transformation of a pixel block.

CHAPTER

6 Two-Dimensional Viewing



We now consider the formal mechanism for displaying views of a picture on an output device. Typically, a graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture. For a two-dimensional picture, a view is selected by specifying a subarea of the total picture area. A user can select a single area for display, or several areas could be selected for simultaneous display or for an animated panning sequence across a scene. The picture parts within the selected areas are then mapped onto specified areas of the device coordinates. When multiple view areas are selected, these areas can be placed in separate display locations, or some areas could be inserted into other, larger display areas. Transformations from world to device coordinates involve translation, rotation, and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected display area.

6-1 THE VIEWING PIPELINE

A world-coordinate area selected for display is called a **window**. An area on a display device to which a window is mapped is called a **viewport.** The window defines *what* is to be viewed; the viewport defines *where* it is to be displayed. Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes. Other window or viewport geometries, such as general polygon shapes and circles, are used in some applications, but these shapes take longer to process. In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a **viewing transformation**. Sometimes the two-dimensional viewing transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation.* But, in general, viewing involves more than just the transformation from the window to the viewport. Figure 6-1 illustrates the mapping of a picture section that falls within a rectangular window onto a designated rectangular viewport.

In computer graphics terminology, the term *window* originally referred to an area of a picture that is selected for viewing, as defined at the beginning of this section. Unfortunately, the same term is now used in window-manager systems to refer to any rectangular screen area that can be moved about, resized, and made active or inactive. In this chapter, we will only use the term window to

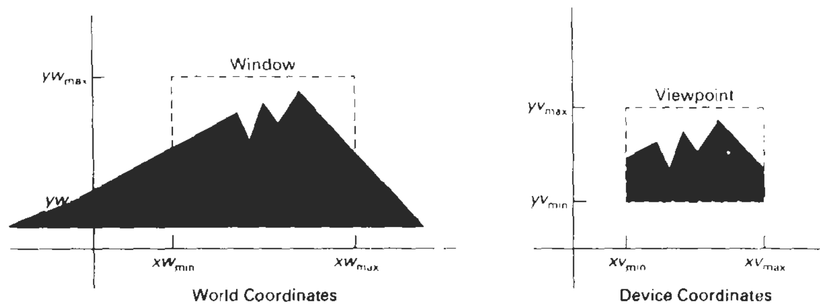


Figure 6-1
A viewing transformation using standard rectangles for the window and viewport.

refer to an area of a world-coordinate scene that has been selected for display. When we consider graphical user interfaces in Chapter 8, we will discuss screen windows and window-manager systems.

Some graphics packages that provide window and viewport operations allow only standard rectangles, but a more general approach is to allow the rectangular window to have any orientation. In this case, we carry out the viewing transformation in several steps, as indicated in Fig. 6-2. First, we construct the scene in world coordinates using the output primitives and attributes discussed in Chapters 3 and 4. Next, to obtain a particular orientation for the window, we can set up a two-dimensional **viewing-coordinate system** in the world-coordinate plane, and define a window in the viewing-coordinate system. The viewing-coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates. We then define a viewport in normalized coordinates (in the range from 0 to 1) and map the viewing-coordinate description of the scene to normalized coordinates. At the final step, all parts of the picture that lie outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. Figure 6-3 illustrates a rotated viewing-coordinate reference frame and the mapping to normalized coordinates.

By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized windows on a

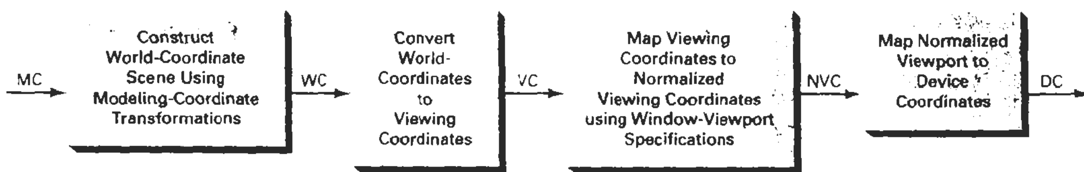


Figure 6-2
The two-dimensional viewing-transformation pipeline.

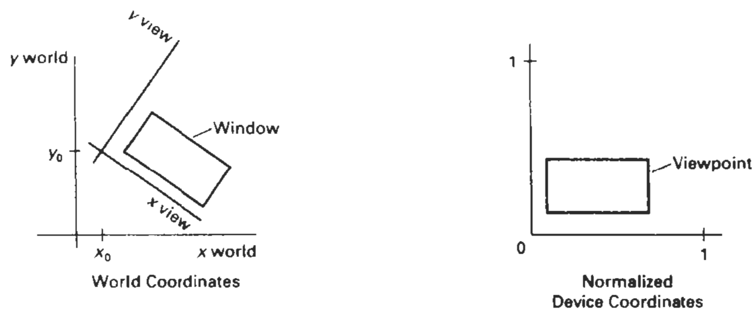


Figure 6-3
Setting up a rotated world window in viewing coordinates and the corresponding normalized-coordinate viewport.

fixed-size viewport. As the windows are made smaller, we zoom in on some part of a scene to view details that are not shown with larger windows. Similarly, more overview is obtained by zooming out from a section of a scene with successively larger windows. Panning effects are produced by moving a fixed-size window across the various objects in a scene.

Viewports are typically defined within the unit square (normalized coordinates). This provides a means for separating the viewing and other transformations from specific output-device requirements, so that the graphics package is largely device-independent. Once the scene has been transferred to normalized coordinates, the unit square is simply mapped to the display area for the particular output device in use at that time. Different output devices can be used by providing the appropriate device drivers.

When all coordinate transformations are completed, viewport clipping can be performed in normalized coordinates or in device coordinates. This allows us to reduce computations by concatenating the various transformation matrices. Clipping procedures are of fundamental importance in computer graphics. They are used not only in viewing transformations, but also in window-manager systems, in painting and drawing packages to eliminate parts of a picture inside or outside of a designated screen area, and in many other applications.

6-2 VIEWING COORDINATE REFERENCE FRAME

This coordinate system provides the reference frame for specifying the world-coordinate window. We set up the viewing coordinate system using the procedures discussed in Section 5-5. First, a viewing-coordinate origin is selected at some world position: $\mathbf{P}_0 = (x_0, y_0)$. Then we need to establish the orientation, or rotation, of this reference frame. One way to do this is to specify a world vector $\mathbf{V}$ that defines the viewing y_v direction. Vector $\mathbf{V}$ is called the **view up vector**.

Given $\mathbf{V}$, we can calculate the components of unit vectors $\mathbf{v} = (v_x, v_y)$ and $\mathbf{u} = (u_x, u_y)$ for the viewing y_v and x_v axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix $\mathbf{R}$ that aligns the viewing $x_v y_v$ axes with the world $x_w y_w$ axes.

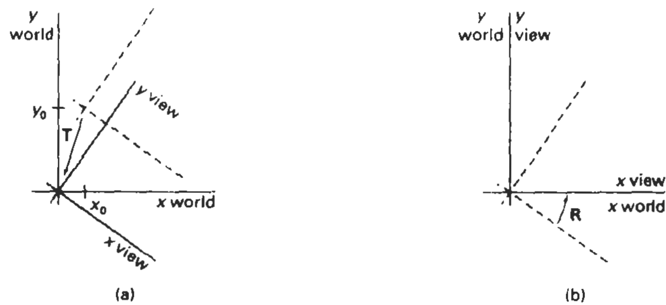


Figure 6-4
A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.

We obtain the matrix for converting world-coordinate positions to viewing coordinates as a two-step composite transformation: First, we translate the viewing origin to the world origin, then we rotate to align the two coordinate reference frames. The composite two-dimensional transformation to convert world coordinates to viewing coordinates is

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T} \tag{6-1}$$

where **T** is the translation matrix that takes the viewing origin point $\mathbf{P}_0$ to the world origin, and **R** is the rotation matrix that aligns the axes of the two reference frames. Figure 6-4 illustrates the steps in this coordinate transformation.

6-3 WINDOW-TO-VIEWPORT COORDINATE TRANSFORMATION

Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates (Fig. 6-3). Object descriptions are then transferred to normalized device coordinates. We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates. If a coordinate position is at the center of the viewing window, for instance, it will be displayed at the center of the viewport.

Figure 6-5 illustrates the window-to-viewport mapping. A point at position (xw, yw) in the window is mapped into position (xv, yv) in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that

$$\begin{aligned} \frac{xv - xv_{min}}{xv_{max} - xv_{min}} &= \frac{xw - xw_{min}}{xw_{max} - xw_{min}} \\ \frac{yv - yv_{min}}{yv_{max} - yv_{min}} &= \frac{yw - yw_{min}}{yw_{max} - yw_{min}} \end{aligned} \tag{6-2}$$

Figure 6-5
A point at position (xw, yw) in a designated window is mapped to viewport coordinates (xv, yv) so that relative positions in the two areas are the same.

Solving these expressions for the viewport position (xv, yv), we have

$$\begin{aligned} xv &= xv_{min} + (xw - xw_{min})sx \\ yv &= yv_{min} + (yw - yw_{min})sy \end{aligned} \tag{6-3}$$

where the scaling factors are

$$\begin{aligned} sx &= \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}} \\ sy &= \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}} \end{aligned} \tag{6-4}$$

Equations 6-3 can also be derived with a set of transformtions that converts the window area into the viewport area. This conversion is performed with the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of (xw_{min}, yw_{min}) that scales the window area to the size of the viewport.
2. Translate the scaled window area to the position of the viewport.

Relative proportions of objects are maintained if the scaling factors are the same ($sx = sy$). Otherwise, world objects will be stretched or contracted in either the x or y direction when displayed on the output device.

Character strings can be handled in two ways when they are mapped to a viewport. The simplest mapping maintains a constant character size, even though the viewport area may be enlarged or reduced relative to the window. This method would be employed when text is formed with standard character fonts that cannot be changed. In systems that allow for changes in character size, string definitions can be windowed the same as other primitives. For characters formed with line segments, the mapping to the viewport can be carried out as a sequence of line transformations.

From normalized coordinates, object descriptions are mapped to the various display devices. Any number of output devices can be open in a particular application, and another window-to-viewport transformation can be performed for each open output device. This mapping, called the **workstation transforma-**

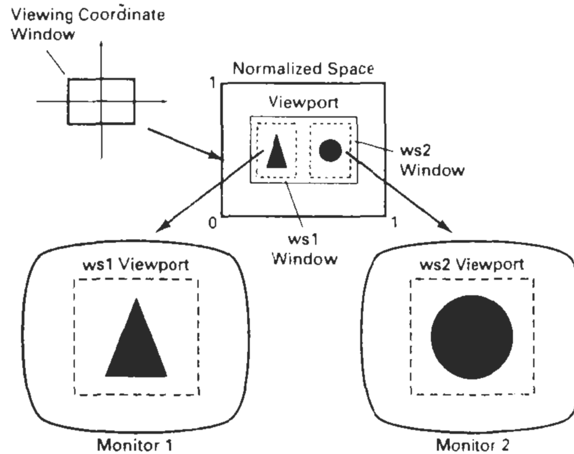


Figure 6-6
Mapping selected parts of a scene in normalized coordinates to different video monitors with workstation transformations.

tion, is accomplished by selecting a window area in normalized space and a viewport area in the coordinates of the display device. With the workstation transformation, we gain some additional control over the positioning of parts of a scene on individual output devices. As illustrated in Fig. 6-6, we can use workstation transformations to partition a view so that different parts of normalized space can be displayed on different output devices.

6-4 TWO-DIMENSIONAL VIEWING FUNCTIONS

We define a viewing reference system in a PHIGS application program with the following function:

```
evaluateViewOrientationMatrix (x0, y0, xV, yV,
                                error, viewMatrix)
```

where parameters x0 and y0 are the coordinates of the viewing origin, and parameters xV and yV are the world-coordinate positions for the view up vector. An integer error code is generated if the input parameters are in error; otherwise, the viewMatrix for the world-to-viewing transformation is calculated. Any number of viewing transformation matrices can be defined in an application.

To set up the elements of a window-to-viewport mapping matrix, we invoke the function

```
evaluateViewMappingMatrix (xwmin, xwmax, ywmin, ywmax,
    xvmin, xvmax, yvmin, yvmax, error, viewMappingMatrix)
```

Here, the window limits in viewing coordinates are chosen with parameters xwmin, xwmax, ywmin, and ywmax; and the viewport limits are set with the nor-

malized coordinate positions xvmin, xvmax, yvmin, yvmax. As with the viewing-transformation matrix, we can construct several window-viewport pairs and use them for projecting various parts of the scene to different areas of the unit square.

Next, we can store combinations of viewing and window–viewport mappings for various workstations in a *viewing table* with

```
setViewRepresentation (ws, viewIndex, viewMatrix,
   viewMappingMatrix, xclipmin, xclipmax, yclipmin,
         yclipmax, clipxy)
```

where parameter ws designates the output device (workstation), and parameter viewIndex sets an integer identifier for this particular window–viewport pair. The matrices viewMatrix and viewMappingMatrix can be concatenated and referenced by the viewIndex. Additional clipping limits can also be specified here, but they are usually set to coincide with the viewport boundaries. And parameter clipxy is assigned either the value *noclip* or the value *clip*. This allows us to turn off clipping if we want to view the parts of the scene outside the viewport. We can also select *noclip* to speed up processing when we know that all of the scene is included within the viewport limits.

The function

```
setViewIndex (viewIndex)
```

selects a particular set of options from the viewing table. This view-index selection is then applied to subsequently specified output primitives and associated attributes and generates a display on each of the active workstations.

At the final stage, we apply a workstation transformation by selecting a workstation window–viewport pair:

```
setWorkstationWindow (ws, xwsWindmin, xwsWindmax,
                          ywsWindmin, ywsWindmax)
setWorkstationViewport (ws, xwsVPortmin, xwsVPortmax,
                          ywsVPortmin, ywsVPortmax)
```

where parameter ws gives the workstation number. Window-coordinate extents are specified in the range from 0 to 1 (normalized space), and viewport limits are in integer device coordinates.

If a workstation viewport is not specified, the unit square of the normalized reference frame is mapped onto the largest square area possible on an output device. The coordinate origin of normalized space is mapped to the origin of device coordinates, and the aspect ratio is retained by transforming the unit square onto a square area on the output device.

Example 6-1 Two-Dimensional Viewing Example

As an example of the use of viewing functions, the following sequence of statements sets up a rotated window in world coordinates and maps its contents to the upper right corner of workstation 2. We keep the viewing coordinate origin at the world origin, and we choose the view up direction for the window as (1, 1). This gives us a viewing-coordinate system that is rotated 45° clockwise in the world-coordinate reference frame. The view index is set to the value 5.

```
evaluateViewOrientationMatrix (0, 0, 1, 1,
                               viewError, viewMat);
evaluateViewMappingMatrix (-60.5, 41.24, -20.75, 82.5, 0.5,
           0.8, 0.7, 1.0, viewMapError, viewMapMat);
setViewRepresentation (2, 5, viewMat, viewMapMat, 0.5, 0.8,
                               0.7, 1.0, clip);
setViewIndex (5);
```

Similarly, we could set up an additional transformation with view index 6 that would map a specified window into a viewport at the lower left of the screen. Two graphs, for example, could then be displayed at opposite screen corners with the following statements.

```
setViewIndex (5);
polyline (3, axes);
polyline (15, data1);
setViewIndex (6);
polyline (3, axes);
polyline (25, data2);
```

View index 5 selects a viewport in the upper right of the screen display, and view index 6 selects a viewport in the lower left corner. The function `polyline (3, axes)` produces the horizontal and vertical coordinate reference for the data plot in each graph.

6-5 CLIPPING OPERATIONS

Generally, any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a **clipping algorithm**, or simply **clipping**. The region against which an object is to clipped is called a **clip window**.

Applications of clipping include extracting part of a defined scene for viewing; identifying visible surfaces in three-dimensional views; antialiasing line segments or object boundaries; creating objects using solid-modeling procedures; displaying a multiwindow environment; and drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating. Depending on the application, the clip window can be a general polygon or it can even have curved boundaries. We first consider clipping methods using rectangular clip regions, then we discuss methods for other clip-region shapes.

For the viewing transformation, we want to display only those picture parts that are within the window area (assuming that the clipping flags have not been set to *noclip*). Everything outside the window is discarded. Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates. Alternatively, the complete world-coordinate picture can be mapped first to device coordinates, or normalized device coordinates, then clipped against the viewport boundaries. World-coordinate clipping removes those primitives outside the window from further consideration, thus eliminating the processing necessary to transform those primitives to device space. Viewport clipping, on the other hand, can reduce calculations by allowing concatenation of viewing and geometric transformation matrices. But

viewport clipping does require that the transformation to device coordinates be performed for all objects, including those outside the window area. On raster systems, clipping algorithms are often combined with scan conversion.

In the following sections, we consider algorithms for clipping the following primitive types

- Point Clipping
- Line Clipping (straight-line segments)
- Area Clipping (polygons)
- Curve Clipping
- Text Clipping

Line and polygon clipping routines are standard components of graphics packages, but many packages accommodate curved objects, particularly spline curves and conics, such as circles and ellipses. Another way to handle curved objects is to approximate them with straight-line segments and apply the line- or polygon-clipping procedure.

6-6 POINT CLIPPING

Assuming that the clip window is a rectangle in standard position, we save a point $\mathbf{P} = (x, y)$ for display if the following inequalities are satisfied:

$$\begin{aligned} xw_{min} \le x \le xw_{max} \\ yw_{min} \le y \le yw_{max} \end{aligned} \tag{6-5}$$

where the edges of the clip window (xw_{min}, xw_{max}, yw_{min}, yw_{max}) can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

Although point clipping is applied less often than line or polygon clipping, some applications may require a point-clipping procedure. For example, point clipping can be applied to scenes involving explosions or sea foam that are modeled with particles (points) distributed in some region of the scene.

6-7 LINE CLIPPING

Figure 6-7 illustrates possible relationships between line positions and a standard rectangular clipping region. A line-clipping procedure involves several parts. First, we can test a given line segment to determine whether it lies completely inside the clipping window. If it does not, we try to determine whether it lies completely outside the window. Finally, if we cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries. We process lines through the "inside-outside" tests by checking the line endpoints. A line with both endpoints inside all clipping boundaries, such as the line from $\mathbf{P}_1$ to $\mathbf{P}_2$, is saved. A line with both endpoints outside any one of the clip boundaries (line $\overline{\mathbf{P}_3\mathbf{P}_4}$ in Fig. 6-7) is outside the win-

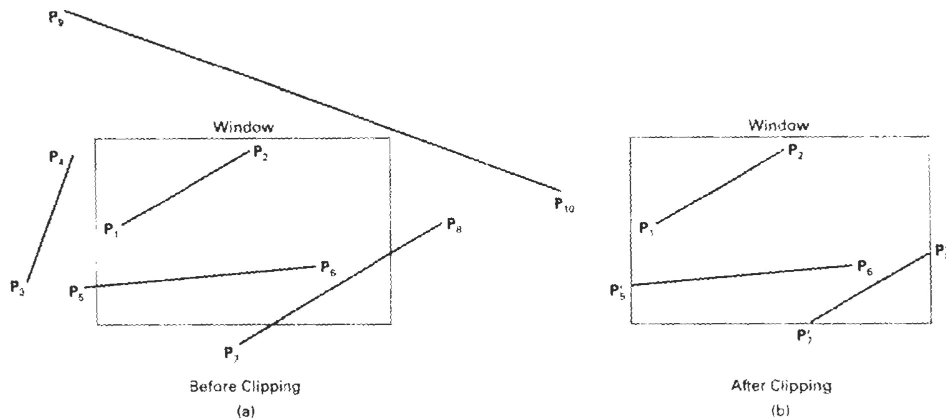


Figure 6-7
Line clipping against a rectangular clip window.

dow. All other lines cross one or more clipping boundaries, and may require calculation of multiple intersection points. To minimize calculations, we try to devise clipping algorithms that can efficiently identify outside lines and reduce intersection calculations.

For a line segment with endpoints (x_1, y_1) and (x_2, y_2) and one or both endpoints outside the clipping rectangle, the parametric representation

$$
\begin{aligned}
x &= x_1 + u(x_2 - x_1) \\
y &= y_1 + u(y_2 - y_1), \qquad 0 \le u \le 1
\end{aligned}
\tag{6-6}
$$

could be used to determine values of parameter u for intersections with the clipping boundary coordinates. If the value of u for an intersection with a rectangle boundary edge is outside the range 0 to 1, the line does not enter the interior of the window at that boundary. If the value of u is within the range from 0 to 1, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in turn to determine whether any part of the line segment is to be displayed. Line segments that are parallel to window edges can be handled as special cases.

Clipping line segments with these parametric tests requires a good deal of computation, and faster approaches to clipping are possible. A number of efficient line clippers have been developed, and we survey the major algorithms in the next sections. Some algorithms are designed explicitly for two-dimensional pictures and some are easily adapted to three-dimensional applications.

Cohen-Sutherland Line Clipping

This is one of the oldest and most popular line-clipping procedures. Generally, the method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated. Every line end-

point in a picture is assigned a four-digit binary code, called a **region code**, that identifies the location of the point relative to the boundaries of the clipping rectangle. Regions are set up in reference to the boundaries as shown in Fig. 6-8. Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the clip window: to the left, right, top, or bottom. By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions can be correlated with the bit positions as

bit 1: left
bit 2: right
bit 3: below
bit 4: above

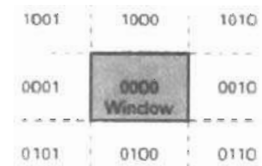


Figure 6-8
Binary region codes assigned to line endpoints according to relative position with respect to the clipping rectangle.

A value of 1 in any bit position indicates that the point is in that relative position; otherwise, the bit position is set to 0. If a point is within the clipping rectangle, the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

Bit values in the region code are determined by comparing endpoint coordinate values (x, y) to the clip boundaries. Bit 1 is set to 1 if $x < xw_{min}$. The other three bit values can be determined using similar comparisons. For languages in which bit manipulation is possible, region-code bit values can be determined with the following two steps: (1) Calculate differences between endpoint coordinates and clipping boundaries. (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code. Bit 1 is the sign bit of $x - xw_{min}$; bit 2 is the sign bit of $xw_{max} - x$; bit 3 is the sign bit of $y - yw_{min}$; and bit 4 is the sign bit of $yw_{max} - y$.

Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside. Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially accept these lines. Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we trivially reject these lines. We would discard the line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code. A method that can be used to test lines for total clipping is to perform the logical *and* operation with both region codes. If the result is not 0000, the line is completely outside the clipping region.

Lines that cannot be identified as completely inside or completely outside a clip window by these tests are checked for intersection with the window boundaries. As shown in Fig. 6-9, such lines may or may not cross into the window interior. We begin the clipping process for a line by comparing an outside endpoint to a clipping boundary to determine how much of the line can be discarded. Then the remaining part of the line is checked against the other boundaries, and we continue until either the line is totally discarded or a section is found inside the window. We set up our algorithm to check line endpoints against clipping boundaries in the order left, right, bottom, top.

To illustrate the specific steps in clipping lines against rectangular boundaries using the Cohen–Sutherland algorithm, we show how the lines in Fig. 6-9 could be processed. Starting with the bottom endpoint of the line from $\mathbf{P}_1$ to $\mathbf{P}_2$,

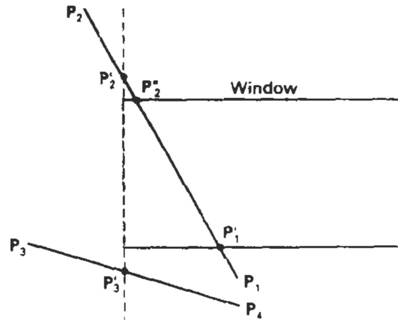


Figure 6-9
Lines extending from one coordinate region to another may pass through the clip window, or they may intersect clipping boundaries without entering the window.

we check $\mathbf{P}_1$ against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle. We then find the intersection point $\mathbf{P}'_1$ with the bottom boundary and discard the line section from $\mathbf{P}_1$ to $\mathbf{P}'_1$. The line now has been reduced to the section from $\mathbf{P}'_1$ to $\mathbf{P}_2$. Since $\mathbf{P}_2$ is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point $\mathbf{P}'_2$ is calculated, but this point is above the window. So the final intersection calculation yields $\mathbf{P}''_2$, and the line from $\mathbf{P}'_1$ to $\mathbf{P}''_2$ is saved. This completes processing for this line, so we save this part and go on to the next line. Point $\mathbf{P}_3$ in the next line is to the left of the clipping rectangle, so we determine the intersection $\mathbf{P}'_3$ and eliminate the line section from $\mathbf{P}_3$ to $\mathbf{P}'_3$. By checking region codes for the line section from $\mathbf{P}'_3$ to $\mathbf{P}_4$, we find that the remainder of the line is below the clip window and can be discarded also.

Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates (x_1, y_1) and (x_2, y_2), the y coordinate of the intersection point with a vertical boundary can be obtained with the calculation

$$y = y_1 + m(x - x_1) \tag{6-7}$$

where the x value is set either to $xw_{\min}$ or to $xw_{\max}$, and the slope of the line is calculated as $m = (y_2 - y_1)/(x_2 - x_1)$. Similarly, if we are looking for the intersection with a horizontal boundary, the x coordinate can be calculated as

$$x = x_1 + \frac{y - y_1}{m} \tag{6-8}$$

with y set either to $yw_{\min}$ or to $yw_{\max}$.

The following procedure demonstrates the Cohen-Sutherland line-clipping algorithm. Codes for each endpoint are stored as bytes and processed using bit manipulations.

```
#define ROUND(a)    ((int)(a+0.5))

/* Bit masks encode a point's position relative to the clip edges.  A
   point's status is encoded by OR'ing together appropriate bit masks.
*/
#define LEFT_EDGE   0x1
```

```
#define RIGHT_EDGE  0x2
#define BOTTOM_EDGE 0x4
#define TOP_EDGE    0x8

/* Points encoded as 0000 are completely Inside the clip rectangle;
   all others are outside at least one edge.  If OR'ing two codes is
   FALSE (no bits are set in either code), the line can be Accepted.  If
   the AND operation between two codes is TRUE, the line defined by those
   endpoints is completely outside the clip region and can be Rejected.
*/
#define INSIDE(a)    (!a)
#define REJECT(a,b)  (a&b)
#define ACCEPT(a,b)  (!(a|b))

unsigned char encode (wcPt2 pt, dcPt winMin, dcPt winMax)
{
  unsigned char code=0x00;

  if (pt.x < winMin.x)
    code = code | LEFT_EDGE;
  if (pt.x > winMax.x)
    code = code | RIGHT_EDGE;
  if (pt.y < winMin.y)
    code = code | BOTTOM_EDGE;
  if (pt.y > winMax.y)
    code = code | TOP_EDGE;
  return (code);
}

void swapPts (wcPt2 * p1, wcPt2 * p2)
{
  wcPt2 tmp;

  tmp = *p1; *p1 = *p2; *p2 = tmp;
}

void swapCodes (unsigned char * c1, unsigned char * c2)
{
  unsigned char tmp;

  tmp = *c1; *c1 = *c2; *c2 = tmp;
}

void clipLine (dcPt winMin, dcPt winMax, wcPt2 p1, wcPt2 p2)
{
  unsigned char code1, code2;
  int done = FALSE, draw = FALSE;
  float m;

  while (!done) {
    code1 = encode (p1, winMin, winMax);
    code2 = encode (p2, winMin, winMax);
    if (ACCEPT (code1, code2)) {
      done = TRUE;
      draw = TRUE;
    }
    else
      if (REJECT (code1, code2))
        done = TRUE;
      else {
        /* Ensure that p1 is outside window */
        if (INSIDE (code1)) {
```

```
        swapPts (&p1, &p2);
        swapCodes (&code1, &code2);
      }
      /* Use slope (m) to find line-clipEdge intersections */
      if (p2.x != p1.x)
        m = (p2.y - p1.y) / (p2.x - p1.x);
      if (code1 & LEFT_EDGE) {
        p1.y += (winMin.x - p1.x) * m;
        p1.x = winMin.x;
      }
      else
        if (code1 & RIGHT_EDGE) {
          p1.y += (winMax.x - p1.x) * m;
          p1.x = winMax.x;
        }
        else
          if (code1 & BOTTOM_EDGE) {
            /* Need to update p1.x for non-vertical lines only */
            if (p2.x != p1.x)
              p1.x += (winMin.y - p1.y) / m;
            p1.y = winMin.y;
          }
          else
            if (code1 & TOP_EDGE) {
              if (p2.x != p1.x)
                p1.x += (winMax.y - p1.y) / m;
              p1.y = winMax.y;
            }
    }
  }
  if (draw)
    lineDDA (ROUND(p1.x), ROUND(p1.y), ROUND(p2.x), ROUND(p2.y));
}
```

Liang–Barsky Line Clipping

Faster line clippers have been developed that are based on analysis of the parametric equation of a line segment, which we can write in the form

$$x = x_1 + u\Delta x \qquad\qquad (6\text{-}9)$$
$$y = y_1 + u\Delta y, \qquad 0 \le u \le 1$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Using these parametric equations, Cyrus and Beck developed an algorithm that is generally more efficient than the Cohen–Sutherland algorithm. Later, Liang and Barsky independently devised an even faster parametric line-clipping algorithm. Following the Liang–Barsky approach, we first write the point-clipping conditions 6-5 in the parametric form:

$$xw_{min} \le x_1 + u\Delta x \le xw_{max} \qquad\qquad (6\text{-}10)$$
$$yw_{min} \le y_1 + u\Delta y \le yw_{max}$$

Each of these four inequalities can be expressed as

$$up_k \le q_k, \qquad k = 1, 2, 3, 4 \qquad\qquad (6\text{-}11)$$

where parameters p and q are defined as

$$\begin{aligned} p_1 &= -\Delta x, & q_1 &= x_1 - xw_{min} \\ p_2 &= \Delta x, & q_2 &= xw_{max} - x_1 \\ p_3 &= -\Delta y, & q_3 &= y_1 - yw_{min} \\ p_4 &= \Delta y, & q_4 &= yw_{max} - y_1 \end{aligned} \qquad (6\text{-}12)$$

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of k corresponding to that boundary (k = 1, 2, 3, and 4 correspond to the left, right, bottom, and top boundaries, respectively). If, for that value of k, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k \geq 0$, the line is inside the parallel clipping boundary.

When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping boundary. If $p_k > 0$, the line proceeds from the inside to the outside. For a nonzero value of p_k, we can calculate the value of u that corresponds to the point where the infinitely extended line intersects the extension of boundary k as

$$u = \frac{q_k}{p_k} \qquad (6\text{-}13)$$

For each line, we can calculate values for parameters u_1 and u_2 that define that part of the line that lies within the clip rectangle. The value of u_1 is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate $r_k = q_k/p_k$. The value of u_1 is taken as the largest of the set consisting of 0 and the various values of r. Conversely, the value of u_2 is determined by examining the boundaries for which the line proceeds from inside to outside ($p > 0$). A value of r_k is calculated for each of these boundaries, and the value of u_2 is the minimum of the set consisting of 1 and the calculated r values. If $u_1 > u_2$, the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter u.

This algorithm is presented in the following procedure. Line intersection parameters are initialized to the values $u_1 = 0$ and $u_2 = 1$. For each clipping boundary, the appropriate values for p and q are calculated and used by the function *clipTest* to determine whether the line can be rejected or whether the intersection parameters are to be adjusted. When $p < 0$, the parameter r is used to update u_1; when $p > 0$, parameter r is used to update u_2. If updating u_1 or u_2 results in $u_1 > u_2$, we reject the line. Otherwise, we update the appropriate u parameter only if the new value results in a shortening of the line. When $p = 0$ and $q < 0$, we can discard the line since it is parallel to and outside of this boundary. If the line has not been rejected after all four values of p and q have been tested, the endpoints of the clipped line are determined from values of u_1 and u_2.

```
#include "graphics.h"

#define ROUND(a) ((int)(a+0.5))

int clipTest (float p, float q, float * u1, float * u2)
```

```
{
  float r;
  int retVal = TRUE;

  if (p < 0.0) {
    r = q / p;
    if (r > *u2)
      retVal = FALSE;
    else
      if (r > *u1)
        *u1 = r;
  }
  else
    if (p > 0.0) {
      r = q / p;
      if (r < *u1)
        retVal = FALSE;
      else if (r < *u2)
        *u2 = r;
    }
    else
      /* p = 0, so line is parallel to this clipping edge */
      if (q < 0.0)
        /* Line is outside clipping edge */
        retVal = FALSE;

  return (retVal);
}

void clipLine (dcPt winMin, dcPt winMax, wcPt2 p1, wcPt2 p2)
{
  float u1 = 0.0, u2 = 1.0, dx = p2.x - p1.x, dy;

  if (clipTest (-dx, p1.x - winMin.x, &u1, &u2))
    if (clipTest (dx, winMax.x - p1.x, &u1, &u2)) {
      dy = p2.y - p1.y;
      if (clipTest (-dy, p1.y - winMin.y, &u1, &u2))
        if (clipTest (dy, winMax.y - p1.y, &u1, &u2)) {
          if (u2 < 1.0) {
            p2.x = p1.x + u2 * dx;
            p2.y = p1.y + u2 * dy;
          }
          if (u1 > 0.0) {
            p1.x += u1 * dx;
            p1.y += u1 * dy;
          }
          lineDDA (ROUND(p1.x), ROUND(p1.y), ROUND(p2.x), ROUND(p2.y));
        }
    }
}
```

In general, the Liang–Barsky algorithm is more efficient than the Cohen–Sutherland algorithm, since intersection calculations are reduced. Each update of parameters u_1 and u_2 requires only one division; and window intersections of the line are computed only once, when the final values of u_1 and u_2 have been computed. In contrast, the Cohen–Sutherland algorithm can repeatedly calculate intersections along a line path, even though the line may be completely outside the clip window. And, each intersection calculation requires both a division and a multiplication. Both the Cohen–Sutherland and the Liang–Barsky algorithms can be extended to three-dimensional clipping (Chapter 12).

Nicholl–Lee–Nicholl Line Clipping

By creating more regions around the clip window, the Nicholl–Lee–Nicholl (or NLN) algorithm avoids multiple clipping of an individual line segment. In the Cohen–Sutherland method, for example, multiple intersections may be calculated along the path of a single line before an intersection on the clipping rectangle is located or the line is completely rejected. These extra intersection calculations are eliminated in the NLN algorithm by carrying out more region testing before intersection positions are calculated. Compared to both the Cohen–Sutherland and the Liang–Barsky algorithms, the Nicholl–Lee–Nicholl algorithm performs fewer comparisons and divisions. The trade-off is that the NLN algorithm can only be applied to two-dimensional clipping, whereas both the Liang–Barsky and the Cohen–Sutherland methods are easily extended to three-dimensional scenes.

For a line with endpoints $\mathbf{P}_1$ and $\mathbf{P}_2$, we first determine the position of point $\mathbf{P}_1$ for the nine possible regions relative to the clipping rectangle. Only the three regions shown in Fig. 6-10 need be considered. If $\mathbf{P}_1$ lies in any one of the other six regions, we can move it to one of the three regions in Fig. 6-10 using a symmetry transformation. For example, the region directly above the clip window can be transformed to the region left of the clip window using a reflection about the line $y = -x$, or we could use a 90° counterclockwise rotation.

Next, we determine the position of $\mathbf{P}_2$ relative to $\mathbf{P}_1$. To do this, we create some new regions in the plane, depending on the location of $\mathbf{P}_1$. Boundaries of the new regions are half-infinite line segments that start at the position of $\mathbf{P}_1$ and pass through the window corners. If $\mathbf{P}_1$ is inside the clip window and $\mathbf{P}_2$ is outside, we set up the four regions shown in Fig. 6-11. The intersection with the appropriate window boundary is then carried out, depending on which one of the four regions (L, T, R, or B) contains $\mathbf{P}_2$. Of course, if both $\mathbf{P}_1$ and $\mathbf{P}_2$ are inside the clipping rectangle, we simply save the entire line.

If $\mathbf{P}_1$ is in the region to the left of the window, we set up the four regions, L, LT, LR, and LB, shown in Fig. 6-12. These four regions determine a unique boundary for the line segment. For instance, if $\mathbf{P}_2$ is in region L, we clip the line at the left boundary and save the line segment from this intersection point to $\mathbf{P}_2$. But if $\mathbf{P}_2$ is in region LT, we save the line segment from the left window boundary to the top boundary. If $\mathbf{P}_2$ is not in any of the four regions, L, LT, LR, or LB, the entire line is clipped.

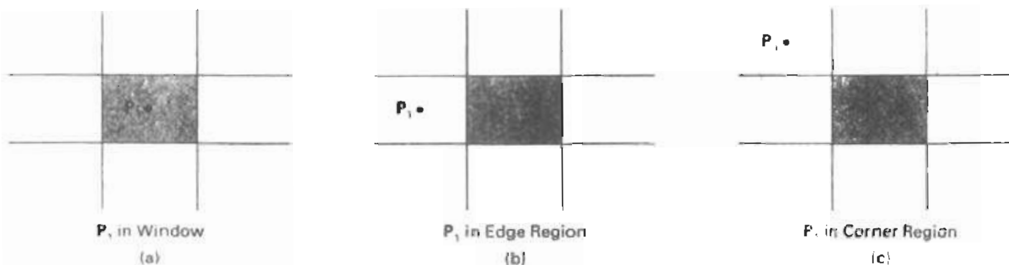


Figure 6-10
Three possible positions for a line endpoint $\mathbf{P}_1$ in the NLN line-clipping algorithm.

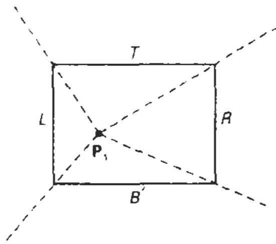


Figure 6-11
The four clipping regions used in the NLN algorithm when $\mathbf{P}_1$ is inside the clip window and $\mathbf{P}_2$ is outside.

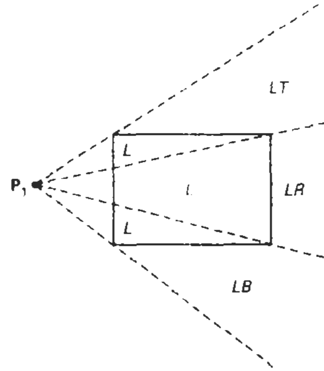


Figure 6-12
The four clipping regions used in the NLN algorithm when $\mathbf{P}_1$ is directly left of the clip window.

For the third case, when $\mathbf{P}_1$ is to the left and above the clip window, we use the clipping regions in Fig. 6-13. In this case, we have the two possibilites shown, depending on the position of $\mathbf{P}_1$ relative to the top left corner of the window. If $\mathbf{P}_2$ is in one of the regions *T, L, TR, TB, LR*, or *LB*, this determines a unique clip-window edge for the intersection calculations. Otherwise, the entire line is rejected.

To determine the region in which $\mathbf{P}_2$ is located, we compare the slope of the line to the slopes of the boundaries of the clip regions. For example, if $\mathbf{P}_1$ is left of the clipping rectangle (Fig. 6-12), then $\mathbf{P}_2$ is in region *LT* if

$$\text{slope } \overline{\mathbf{P}_1\mathbf{P}_{TR}} < \text{slope } \overline{\mathbf{P}_1\mathbf{P}_2} < \text{slope } \overline{\mathbf{P}_1\mathbf{P}_{TL}} \tag{6-14}$$

or

$$\frac{y_T - y_1}{x_R - x_1} < \frac{y_2 - y_1}{x_2 - x_1} < \frac{y_T - y_1}{x_L - x_1} \tag{6-15}$$

And we clip the entire line if

$$(y_T - y_1)(x_2 - x_1) < (x_L - x_1)(y_2 - y_1) \tag{6-16}$$

The coordinate difference and product calculations used in the slope tests are saved and also used in the intersection calculations. From the parametric equations

$$x = x_1 + (x_2 - x_1)u$$
$$y = y_1 + (y_2 - y_1)u$$

an x-intersection position on the left window boundary is $x = x_L$, with $u = (x_L - x_1)/(x_2 - x_1)$, so that the y-intersection position is

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_L - x_1) \tag{6-17}$$

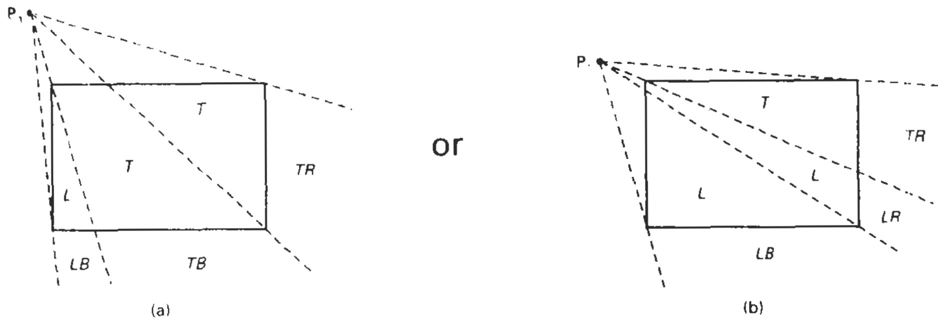


Figure 6-13
The two possible sets of clipping regions used in the NLN algorithm when P_1 is above and to the left of the clip window.

And an intersection position on the top boundary has $y = y_T$ and $u = (y_T - y_1)/(y_2 - y_1)$, with

$$x = x_1 + \frac{x_2 - x_1}{y_2 - y_1}(y_T - y_1) \qquad (6\text{-}18)$$

Line Clipping Using Nonrectangular Clip Windows

In some applications, it is often necessary to clip lines against arbitrarily shaped polygons. Algorithms based on parametric line equations, such as the Liang–Barsky method and the earlier Cyrus–Beck approach, can be extended easily to convex polygon windows. We do this by modifying the algorithm to include the parametric equations for the boundaries of the clip region. Preliminary screening of line segments can be accomplished by processing lines against the coordinate extents of the clipping polygon. For concave polygon-clipping regions, we can still apply these parametric clipping procedures if we first split the concave polygon into a set of convex polygons.

Circles or other curved-boundary clipping regions are also possible, but less commonly used. Clipping algorithms for these areas are slower because intersection calculations involve nonlinear curve equations. At the first step, lines can be clipped against the bounding rectangle (coordinate extents) of the curved clipping region. Lines that can be identified as completely outside the bounding rectangle are discarded. To identify inside lines, we can calculate the distance of line endpoints from the circle center. If the square of this distance for both endpoints of a line is less than or equal to the radius squared, we can save the entire line. The remaining lines are then processed through the intersection calculations, which must solve simultaneous circle-line equations.

Splitting Concave Polygons

We can identify a concave polygon by calculating the cross products of successive edge vectors in order around the polygon perimeter. If the z component of

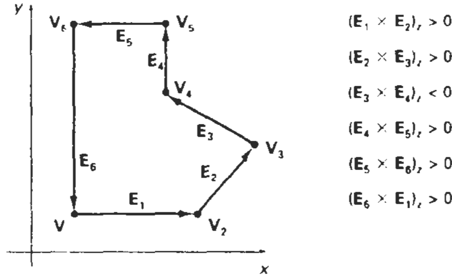


Figure 6-14
Identifying a concave polygon by calculating cross products of successive pairs of edge vectors.

some cross products is positive while others have a negative z component, we have a concave polygon. Otherwise, the polygon is convex. This is assuming that no series of three successive vertices are collinear, in which case the cross product of the two edge vectors for these vertices is zero. If all vertices are collinear, we have a degenerate polygon (a straight line). Figure 6-14 illustrates the edge-vector cross-product method for identifying concave polygons.

A *vector method* for splitting a concave polygon in the xy plane is to calculate the edge-vector cross products in a counterclockwise order and to note the sign of the z component of the cross products. If any z component turns out to be negative (as in Fig. 6-14), the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair. The following example illustrates this method for splitting a concave polygon.

Example 6-2: Vector Method for Splitting Concave Polygons

Figure 6-15 shows a concave polygon with six edges. Edge vectors for this polygon can be expressed as

$$\mathbf{E}_1 = (1, 0, 0), \qquad \mathbf{E}_2 = (1, 1, 0)$$
$$\mathbf{E}_3 = (1, -1, 0), \qquad \mathbf{E}_4 = (0, 2, 0)$$
$$\mathbf{E}_5 = (-3, 0, 0), \qquad \mathbf{E}_6 = (0, -2, 0)$$

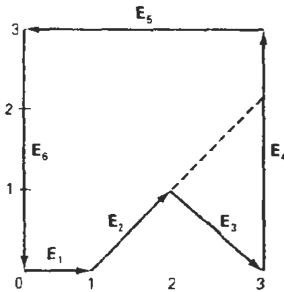


Figure 6-15
Splitting a concave polygon using the vector method.

where the z component is 0, since all edges are in the xy plane. The cross product $\mathbf{E}_i \times \mathbf{E}_j$ for two successive edge vectors is a vector perpendicular to the xy plane with z component equal to $E_{ix}E_{jy} - E_{jx}E_{iy}$.

$$\mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1), \qquad \mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, -2)$$
$$\mathbf{E}_3 \times \mathbf{E}_4 = (0, 0, 2), \qquad \mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6)$$
$$\mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6), \qquad \mathbf{E}_6 \times \mathbf{E}_1 = (0, 0, 2)$$

Since the cross product $\mathbf{E}_2 \times \mathbf{E}_3$ has a negative z component, we split the polygon along the line of vector $\mathbf{E}_2$. The line equation for this edge has a slope of 1 and a y intercept of -1. We then determine the intersection of this line and the other

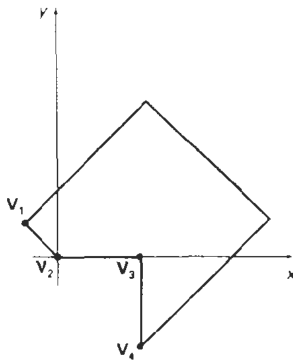


Figure 6-16
Splitting a concave polygon using the rotational method. After rotating $\mathbf{V}_3$ onto the x axis, we find that $\mathbf{V}_4$ is below the x axis. So we split the polygon along the line of $\overline{\mathbf{V}_2\mathbf{V}_3}$.

polygon edges to split the polygon into two pieces. No other edge cross products are negative, so the two new polygons are both convex.

We can also split a concave polygon using a *rotational method*. Proceeding counterclockwise around the polygon edges, we translate each polygon vertex V_k in turn to the coordinate origin. We then rotate in a clockwise direction so that the next vertex V_{k+1} is on the x axis. If the next vertex, V_{k+2}, is below the x axis, the polygon is concave. We then split the polygon into two new polygons along the x axis and repeat the concave test for each of the two new polygons. Otherwise, we continue to rotate vertices on the x axis and to test for negative y vertex values. Figure 6-16 illustrates the rotational method for splitting a concave polygon.

6-8 POLYGON CLIPPING

To clip polygons, we need to modify the line-clipping procedures discussed in the previous section. A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments (Fig. 6-17), depending on the orientation of the polygon to the clipping window. What we really want to display is a bounded area after clipping, as in Fig. 6-18. For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.

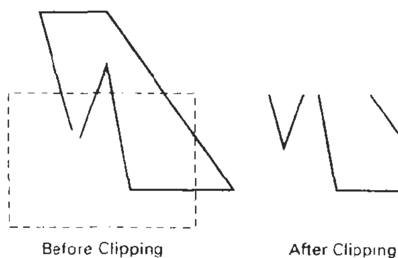


Figure 6-17
Display of a polygon processed by a line-clipping algorithm

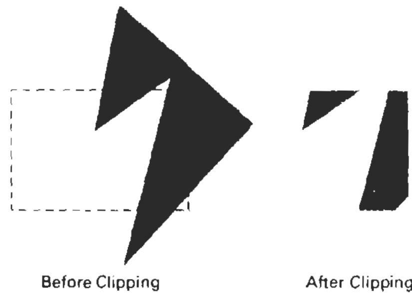


Figure 6-18
Display of a correctly clipped polygon.

Sutherland-Hodgeman Polygon Clipping

We can correctly clip a polygon by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary in turn. Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then be successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper, as in Fig. 6-19. At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.

There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests: (1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list. (2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list. (3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list. (4) If both input vertices are outside the window boundary, nothing is added to the output list. These four cases are illustrated in Fig. 6-20 for successive pairs of polygon vertices. Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

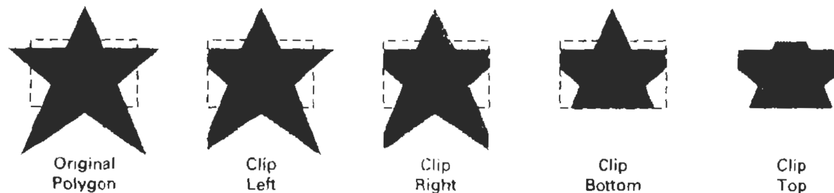


Figure 6-19
Clipping a polygon against successive window boundaries.

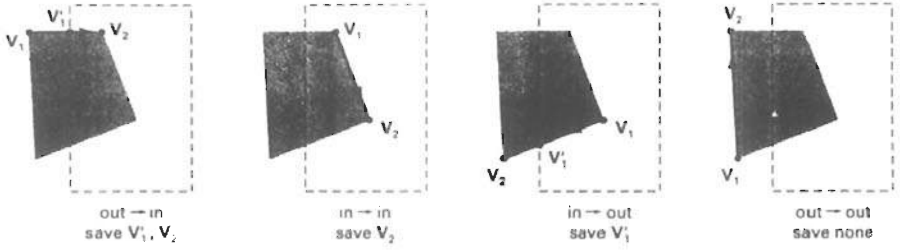


Figure 6-20
Successive processing of pairs of polygon vertices against the left window boundary.

We illustrate this method by processing the area in Fig. 6-21 against the left window boundary. Vertices 1 and 2 are found to be on the outside of the boundary. Moving along to vertex 3, which is inside, we calculate the intersection and save both the intersection point and vertex 3. Vertices 4 and 5 are determined to be inside, and they also are saved. The sixth and final vertex is outside, so we find and save the intersection point. Using the five saved points, we would repeat the process for the next window boundary.

Implementing the algorithm as we have just described requires setting up storage for an output list of vertices as a polygon is clipped against each window boundary. We can eliminate the intermediate output vertex lists by simply clipping individual vertices at each step and passing the clipped vertices on to the next boundary clipper. This can be done with parallel processors or a single processor and a pipeline of clipping routines. A point (either an input vertex or a calculated intersection point) is added to the output vertex list only after it has been been determined to be inside or on a window boundary by all four boundary clippers. Otherwise, the point does not continue in the pipeline. Figure 6-22 shows a polygon and its intersection points with a clip window. In Fig. 6-23, we illustrate the progression of the polygon vertices in Fig. 6-22 through a pipeline of boundary clippers.

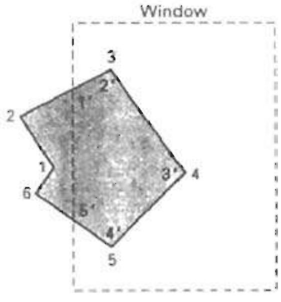


Figure 6-21
Clipping a polygon against the left boundary of a window, starting with vertex 1. Primed numbers are used to label the points in the output vertex list for this window boundary.

The following procedure demonstrates the pipeline clipping approach. An array, s, records the most recent point that was clipped for each clip-window boundary. The main routine passes each vertex p to the `clipPoint` routine for clipping against the first window boundary. If the line defined by endpoints p and `s[boundary]` crosses this window boundary, the intersection is calculated and passed to the next clipping stage. If p is inside the window, it is passed to the next clipping stage. Any point that survives clipping against all window boundaries is then entered into the output array of points. The array `firstPoint` stores for each window boundary the first point clipped against that boundary. After all polygon vertices have been processed, a closing routine clips lines defined by the first and last points clipped against each boundary.

```
typedef enum { Left, Right, Bottom, Top } Edge;
#define N_EDGE 4

int inside (wcPt2 p, Edge b, dcPt wMin, dcPt wMax)
{
  switch (b) {
```

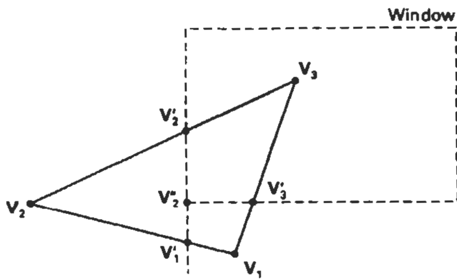


Figure 6-22
A polygon overlapping a rectangular clip window.

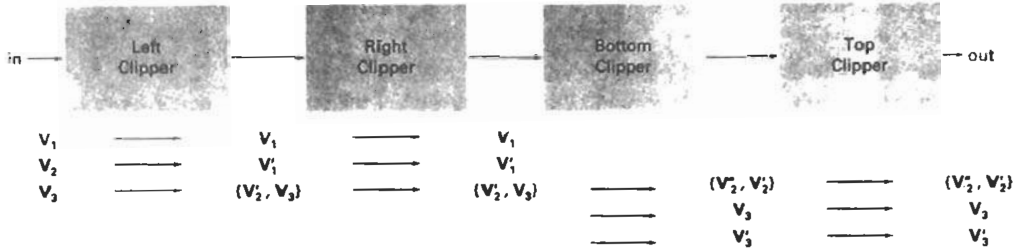


Figure 6-23
Processing the vertices of the polygon in Fig. 6-22 through a boundary-clipping pipeline. After all vertices are processed through the pipeline, the vertex list for the clipped polygon is $\{V_2'', V_2', V_3, V_3'\}$.

```
  case Left:   if (p.x < wMin.x) return (FALSE); break;
  case Right:  if (p.x > wMax.x) return (FALSE); break;
  case Bottom: if (p.y < wMin.y) return (FALSE); break;
  case Top:    if (p.y > wMax.y) return (FALSE); break;
  }
  return (TRUE);
}

int cross (wcPt2 p1, wcPt2 p2, Edge b, dcPt wMin, dcPt wMax)
{
  if (inside (p1, b, wMin, wMax) == inside (p2, b, wMin, wMax))
    return (FALSE);
  else return (TRUE);
}

wcPt2 intersect (wcPt2 p1, wcPt2 p2, Edge b, dcPt wMin, dcPt wMax)
{
  wcPt2 iPt;
  float m;

  if (p1.x != p2.x) m = (p1.y - p2.y) / (p1.x - p2.x);
  switch (b) {
  case Left:
    iPt.x = wMin.x;
    iPt.y = p2.y + (wMin.x - p2.x) * m;
    break;
  case Right:
    iPt.x = wMax.x;
```

```
    iPt.y = p2.y + (wMax.x - p2.x) * m;
    break;
  case Bottom:
    iPt.y = wMin.y;
    if (p1.x != p2.x) iPt.x = p2.x + (wMin.y - p2.y) / m;
    else iPt.x = p2.x;
    break;
  case Top:
    iPt.y = wMax.y;
    if (p1.x != p2.x) iPt.x = p2.x + (wMax.y - p2.y) / m;
    else iPt.x = p2.x;
    break;
  }
  return (iPt);
}

void clipPoint (wcPt2 p, Edge b, dcPt wMin, dcPt wMax,
                wcPt2 * pOut, int * cnt, wcPt2 * first[], wcPt2 * s)
{
  wcPt2 iPt;

  /* If no previous point exists for this edge, save this point. */
  if (!first[b])
    first[b] = &p;
  else
    /* Previous point exists.  If 'p' and previous point cross edge,
       find intersection.  Clip against next boundary, if any.  If
       no more edges, add intersection to output list. */
    if (cross (p, s[b], b, wMin, wMax)) {
      iPt = intersect (p, s[b], b, wMin, wMax);
      if (b < Top)
        clipPoint (iPt, b+1, wMin, wMax, pOut, cnt, first, s);
      else {
        pOut[*cnt] = iPt;  (*cnt)++;
      }
    }

  s[b] = p;           /* Save 'p' as most recent point for this edge */

  /* For all, if point is 'inside' proceed to next clip edge, if any */
  if (inside (p, b, wMin, wMax))
    if (b < Top)
      clipPoint (p, b+1, wMin, wMax, pOut, cnt, first, s);
    else {
      pOut[*cnt] = p;  (*cnt)++;
    }
}

void closeClip (dcPt wMin, dcPt wMax, wcPt2 * pOut,
                int * cnt, wcPt2 * first[], wcPt2 * s)
{
  wcPt2 i;
  Edge b;

  for (b = Left; b <= Top; b++) {
    if (cross (s[b], *first[b], b, wMin, wMax)) {
      i = intersect (s[b], *first[b], b, wMin, wMax);
      if (b < Top)
        clipPoint (i, b+1, wMin, wMax, pOut, cnt, first, s);
      else {
        pOut[*cnt] = i;  (*cnt)++;
      }
    }
```

```
  }
}

int clipPolygon (dcPt wMin, dcPt wMax, int n, wcPt2 * pIn, wcPt2 * pOut)
{
  /* 'first' holds pointer to first point processed against a clip
     edge.  's' holds most recent point processed against an edge */
  wcPt2 * first[N_EDGE] = { 0, 0, 0, 0 }  s[N_EDGE];
  int i, cnt = 0;

  for (i=0; i<n; i++)
    clipPoint (pIn[i], Left, wMin, wMax, pOut, &cnt, first, s);
  closeClip (wMin, wMax, pOut, &cnt, first, s);
  return (cnt);
}
```

Convex polygons are correctly clipped by the Sutherland-Hodgeman algorithm, but concave polygons may be displayed with extraneous lines, as demonstrated in Fig. 6-24. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex. There are several things we could do to correctly display concave polygons. For one, we could split the concave polygon into two or more convex polygons and process each convex polygon separately. Another possibility is to modify the Sutherland-Hodgeman approach to check the final vertex list for multiple vertex points along any clip window boundary and correctly join pairs of vertices. Finally, we could use a more general polygon clipper, such as either the Weiler-Atherton algorithm or the Weiler algorithm described in the next section.

Weiler-Atherton Polygon Clipping

Here, the vertex-processing procedures for window boundaries are modified so that concave polygons are displayed correctly. This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

The basic idea in this algorithm is that instead of always proceeding around the polygon edges as vertices are processed, we sometimes want to follow the window boundaries. Which path we follow depends on the polygon-processing direction (clockwise or counterclockwise) and whether the pair of polygon vertices currently being processed represents an outside-to-inside pair or an inside-

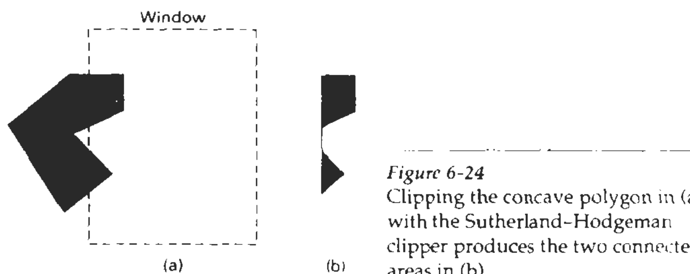


Figure 6-24
Clipping the concave polygon in (a) with the Sutherland-Hodgeman clipper produces the two connected areas in (b).

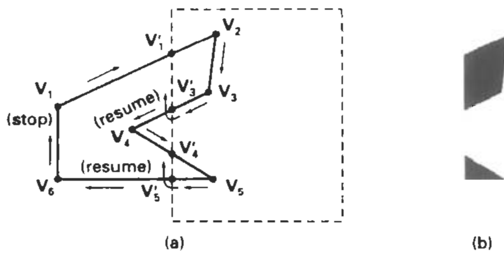


Figure 6-25
Clipping a concave polygon (a) with the Weiler–Atherton algorithm generates the two separate polygon areas in (b).

to-outside pair. For clockwise processing of polygon vertices, we use the following rules:

- For an outside-to-inside pair of vertices, follow the polygon boundary.
- For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction.

In Fig. 6-25, the processing direction in the Weiler–Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.

An improvement on the Weiler–Atherton algorithm is the Weiler algorithm, which applies constructive solid geometry ideas to clip an arbitrary polygon against any polygon-clipping region. Figure 6-26 illustrates the general idea in this approach. For the two polygons in this figure, the correctly clipped polygon is calculated as the intersection of the clipping polygon and the polygon object.

Other Polygon-Clipping Algorithms

Various parametric line-clipping methods have also been adapted to polygon clipping. And they are particularly well suited for clipping against convex polygon-clipping windows. The Liang–Barsky Line Clipper, for example, can be extended to polygon clipping with a general approach similar to that of the Sutherland–Hodgeman method. Parametric line representations are used to process polygon edges in order around the polygon perimeter using region-testing procedures similar to those used in line clipping.

Figure 6-26
Clipping a polygon by determining the intersection of two polygon areas.

6-9 CURVE CLIPPING

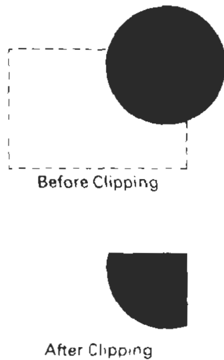


Figure 6-27
Clipping a filled circle.

Areas with curved boundaries can be clipped with methods similar to those discussed in the previous sections. Curve-clipping procedures will involve nonlinear equations, however, and this requires more processing than for objects with linear boundaries.

The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window. If the bounding rectangle for the object is completely inside the window, we save the object. If the rectangle is determined to be completely outside the window, we discard the object. In either case, there is no further computation necessary. But if the bounding rectangle test fails, we can look for other computation-saving approaches. For a circle, we can use the coordinate extents of individual quadrants and then octants for preliminary testing before calculating curve-window intersections. For an ellipse, we can test the coordinate extents of individual quadrants. Figure 6-27 illustrates circle clipping against a rectangular window.

Similar procedures can be applied when clipping a curved object against a general polygon clip region. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region. If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.

6-10 TEXT CLIPPING

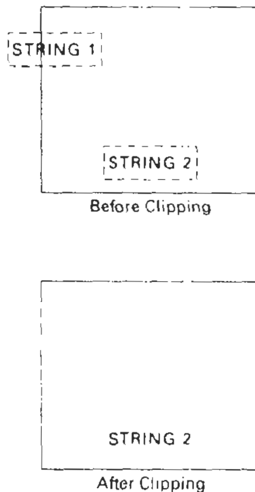


Figure 6-28
Text clipping using a bounding rectangle about the entire string.

There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.

The simplest method for processing character strings relative to a window boundary is to use the *all-or-none string-clipping* strategy shown in Fig. 6-28. If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern. The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping.

An alternative to rejecting an entire character string that overlaps a window boundary is to use the *all-or-none character-clipping* strategy. Here we discard only those characters that are not completely inside the window (Fig. 6-29). In this case, the boundary limits of individual characters are compared to the window. Any character that either overlaps or is outside a window boundary is clipped.

A final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines. If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window (Fig. 6-30). Outline character fonts formed with line segments can be processed in this way using a line-clipping algorithm. Characters defined with bit maps would be clipped by comparing the relative position of the individual pixels in the character grid patterns to the clipping boundaries.

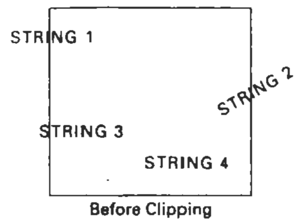
6-11 EXTERIOR CLIPPING

So far, we have considered only procedures for clipping a picture to the interior of a region by eliminating everything outside the clipping region. What is saved by these procedures is *inside* the region. In some cases, we want to do the reverse, that is, we want to clip a picture to the exterior of a specified region. The picture parts to be saved are those that are *outside* the region. This is referred to as **exterior clipping**.

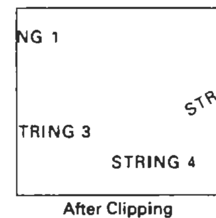
A typical example of the application of exterior clipping is in multiple-window systems. To correctly display the screen windows, we often need to apply both internal and external clipping. Figure 6-31 illustrates a multiple-window display. Objects within a window are clipped to the interior of that window. When other higher-priority windows overlap these objects, the objects are also clipped to the exterior of the overlapping windows.

Exterior clipping is used also in other applications that require overlapping pictures. Examples here include the design of page layouts in advertising or publishing applications or for adding labels or design patterns to a picture. The technique can also be used for combining graphs, maps, or schematics. For these applications, we can use exterior clipping to provide a space for an insert into a larger picture.

Procedures for clipping objects to the interior of concave polygon windows can also make use of external clipping. Figure 6-32 shows a line $\overline{P_1P_2}$ that is to be clipped to the interior of a concave window with vertices $V_1V_2V_3V_4V_5$. Line $\overline{P_1P_2}$ can be clipped in two passes: (1) First, $\overline{P_1P_2}$ is clipped to the interior of the convex polygon $V_1V_2V_3V_4$ to yield the clipped segment $\overline{P'_1P'_2}$ (Fig. 6-32(b)). (2) Then an external clip of $\overline{P'_1P'_2}$ is performed against the convex polygon $V_1V_5V_4$ to yield the final clipped line segment $\overline{P''_1P'_2}$.

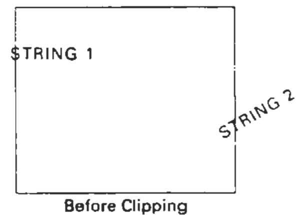


Before Clipping

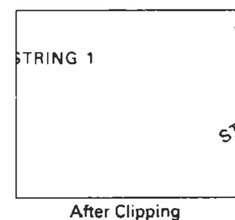


After Clipping

Figure 6-29
Text clipping using a bounding rectangle about individual characters.



Before Clipping



After Clipping

Figure 6-30
Text clipping performed on the components of individual characters.

SUMMARY

In this chapter, we have seen how we can map a two-dimensional world-coordinate scene to a display device. The viewing-transformation pipeline in-

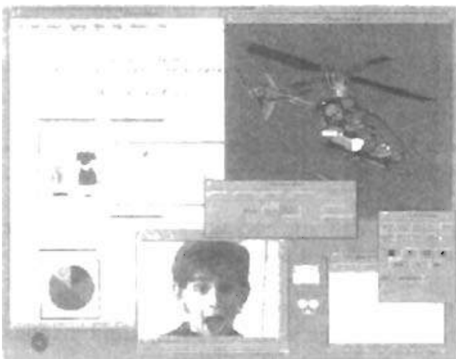


Figure 6-31
A multiple-window screen display showing examples of both interior and exterior clipping. (*Courtesy of Sun Microsystems*).

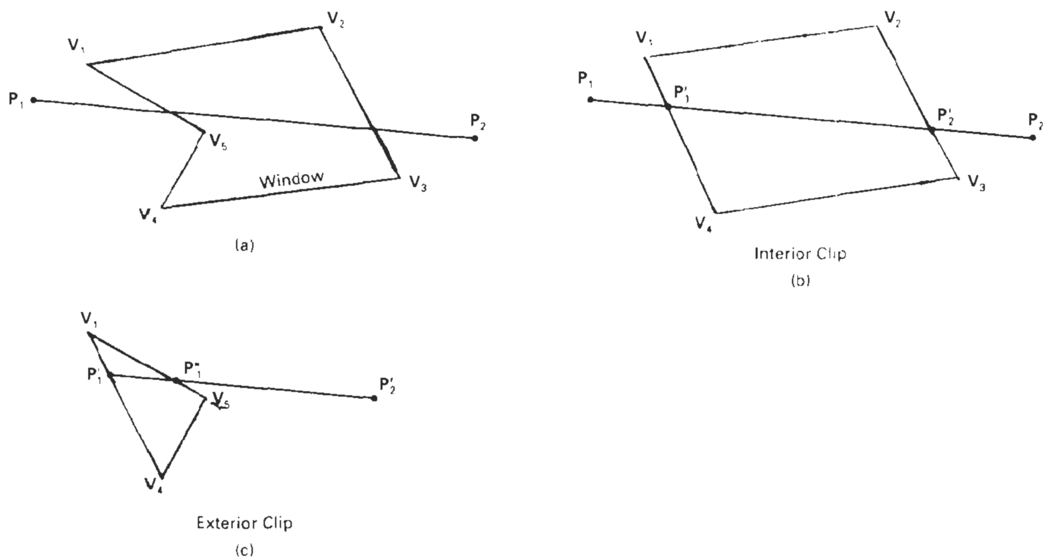


Figure 6-32
Clipping line $\overline{P_1P_2}$ to the interior of a concave polygon with vertices $V_1V_2V_3V_4V_5$ (a), using convex polygons $V_1V_2V_3V_4$ (b) and $V_1V_5V_4$ (c), to produce the clipped line $\overline{P''_1P'_2}$.

cludes constructing the world-coordinate scene using modeling transformations transferring world-coordinates to viewing coordinates, mapping the viewing-coordinate descriptions of objects to normalized device coordinates, and finally mapping to device coordinates. Normalized coordinates are specified in the range from 0 to 1, and they are used to make viewing packages independent of particular output devices.

Viewing coordinates are specified by giving the world-coordinate position of the viewing origin and the view up vector that defines the direction of the viewing y axis. These parameters are used to construct the viewing transformation matrix that maps world-coordinate object descriptions to viewing coordinates.

A window is then set up in viewing coordinates, and a viewport is specified in normalized device coordinates. Typically, the window and viewport are rectangles in standard position (rectangle boundaries are parallel to the coordinate axes). The mapping from viewing coordinates to normalized device coordinates is then carried out so that relative positions in the window are maintained in the viewport.

Viewing functions in a graphics programming package are used to create one or more sets of viewing parameters. One function is typically provided to calculate the elements of the matrix for transforming world coordinates to viewing coordinates. Another function is used to set up the window-to-viewport transformation matrix, and a third function can be used to specify combinations of viewing transformations and window mapping in a viewing table. We can

then select different viewing combinations by specifying particular view indices listed in the viewing table.

When objects are displayed on the output device, all parts of a scene outside the window (and the viewport) are clipped off unless we set clip parameters to turn off clipping. In many packages, clipping is done in normalized device coordinates so that all transformations can be concatenated into a single transformation operation before applying the clipping algorithms. The clipping region is commonly referred to as the clipping window, or as the clipping rectangle when the window and viewport are standard rectangles. Several algorithms have been developed for clipping objects against the clip-window boundaries.

Line-clipping algorithms include the Cohen-Sutherland method, the Liang-Barsky method, and the Nicholl-Lee-Nicholl method. The Cohen-Sutherland method is widely used, since it was one of the first line-clipping algorithms to be developed. Region codes are used to identify the position of line endpoints relative to the rectangular, clipping window boundaries. Lines that cannot be immediately identified as completely inside the window or completely outside are then clipped against window boundaries. Liang and Barsky use a parametric line representation, similar to that of the earlier Cyrus-Beck algorithm, to set up a more efficient line-clipping procedure that reduces intersection calculations. The Nicholl-Lee-Nicholl algorithm uses more region testing in the xy plane to reduce intersection calculations even further. Parametric line clipping is easily extended to convex clipping windows and to three-dimensional clipping windows.

Line clipping can also be carried out for concave, polygon clipping windows and for clipping windows with curved boundaries. With concave polygons, we can use either the vector method or the rotational method to split a concave polygon into a number of convex polygons. With curved clipping windows, we calculate line intersections using the curve equations.

Polygon-clipping algorithms include the Sutherland-Hodgeman method, the Liang-Barsky method, and the Weiler-Atherton method. In the Sutherland-Hodgeman clipper, vertices of a convex polygon are processed in order against the four rectangular window boundaries to produce an output vertex list for the clipped polygon. Liang and Barsky use parametric line equations to represent the convex polygon edges, and they use similar testing to that performed in line clipping to produce an output vertex list for the clipped polygon. Both the Weiler-Atherland method and the Weiler method correctly clip both convex and concave polygons, and these polygon clippers also allow the clipping window to be a general polygon. The Weiler-Atherland algorithm processes polygon vertices in order to produce one or more lists of output polygon vertices. The Weiler method performs clipping by finding the intersection region of the two polygons.

Objects with curved boundaries are processed against rectangular clipping windows by calculating intersections using the curve equations. These clipping procedures are slower than line clippers or polygon clippers, because the curve equations are nonlinear.

The fastest text-clipping method is to completely clip a string if any part of the string is outside any window boundary. Another method for text clipping is to use the all-or-none approach with the individual characters in a string. A third method is to apply either point, line, polygon, or curve clipping to the individual characters in a string, depending on whether characters are defined as point grids or as outline fonts.

In some applications, such as creating picture insets and managing multiple-screen windows, exterior clipping is performed. In this case, all parts of a scene that are inside a window are clipped and the exterior parts are saved.

REFERENCES

Line-clipping algorithms are discussed in Sproull and Sutherland (1968), Cyrus and Beck (1978), and Liang and Barsky (1984). Methods for improving the speed of the Cohen–Sutherland line-clipping algorithm are given in Duvanenko (1990).

Polygon-clipping methods are presented in Sutherland and Hodgeman (1974) and in Liang and Barsky (1983). General techniques for clipping arbitrarily shaped polygons against each other are given in Weiler and Atherton (1977) and in Weiler (1980).

Two-dimensional viewing operations in PHIGS are discussed in Howard et al. (1991), Gaskins (1992), Hopgood and Duce (1991), and Blake (1993). For information on GKS viewing operations, see Hopgood et al. (1983) and Enderle et al. (1984).

EXERCISES

6-1. Write a procedure to to implement the `evaluateViewOrientationMatrix` function that calculates the elements of the matrix for transforming world coordinates to viewing coordinates, given the viewing coordinate origin $\mathbf{P}_0$ and the view up vector $\mathbf{V}$.

6-2. Derive the window-to-viewport transformation equations 6-3 by first scaling the window to the size of the viewport and then translating the scaled window to the viewport position.

6-3. Write a procedure to implement the `evaluateViewMappingMatrix` function that calculates the elements of a matrix for performing the window-to-viewport transformation.

6-4. Write a procedure to implement the `setViewRepresentation` function to concatenate `viewMatrix` and `viewMappingMatrix` and to store the result, referenced by a specified view index, in a viewing table.

6-5. Write a set of procedures to implement the viewing pipeline without clipping and without the workstation transformation. Your program should allow a scene to be constructed with modeling-coordinate transformations, a specified viewing system, and a specified window–viewport pair. As an option, a viewing table can be implemented to store different sets of viewing transformation parameters.

6-6. Derive the matrix representation for a workstation transformation.

6-7. Write a set of procedures to implement the viewing pipeline without clipping, but including the workstation transformation. Your program should allow a scene to be constructed with modeling-coordinate transformations, a specified viewing system, a specified window–viewport pair, and workstation transformation parameters. For a given world-coordinate scene, the composite viewing transformation matrix should transform the scene to an output device for display.

6-8. Implement the Cohen–Sutherland line-clipping algorithm.

6-9. Carefully discuss the rationale behind the various tests and methods for calculating the intersection parameters u_1 and u_2 in the Liang–Barsky line-clipping algorithm.

6-10. Compare the number of arithmetic operations performed in the Cohen–Sutherland and the Liang–Barsky line-clipping algorithms for several different line orientations relative to a clipping window.

6-11. Write a procedure to implement the Liang–Barsky line-clipping algorithm.

6-12. Devise symmetry transformations for mapping the intersection calculations for the three regions in Fig. 6-10 to the other six regions of the xy plane.

6-13. Set up a detailed algorithm for the Nicholl–Lee–Nicholl approach to line clipping for any input pair of line endpoints.

6-14. Compare the number of arithmetic operations performed in NLN algorithm to both the Cohen–Sutherland and the Liang–Barsky line-clipping algorithms for several different line orientations relative to a clipping window.

6-15. Write a routine to identify concave polygons by calculating cross products of pairs of edge vectors.

6-16. Write a routine to split a concave polygon using the vector method.

6-17. Write a routine to split a concave polygon using the rotational method.

6-18. Adapt the Liang–Barsky line-clipping algorithm to polygon clipping.

6-19. Set up a detailed algorithm for Weiler–Atherton polygon clipping assuming that the clipping window is a rectangle in standard position.

6-20. Devise an algorithm for Weiler–Atherton polygon clipping, where the clipping window can be any specified polygon.

6-21. Write a routine to clip an ellipse against a rectangular window.

6-22. Assuming that all characters in a text string have the same width, develop a text-clipping algorithm that clips a string according to the "all-or-none character-clipping" strategy.

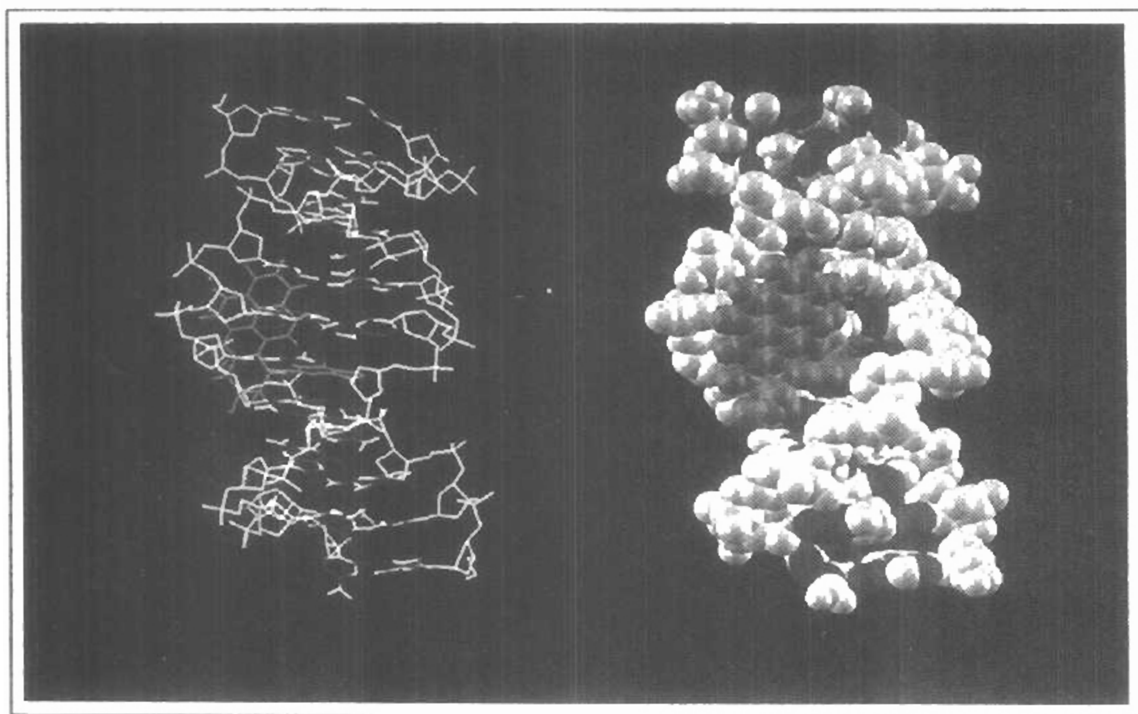
6-23. Develop a text-clipping algorithm that clips individual characters assuming that the characters are defined in a pixel grid of a specified size.

6-24. Write a routine to implement exterior clipping on any part of a defined picture using any specified window.

6-25. Write a routine to perform both interior and exterior clipping, given a particular window-system display. Input to the routine is a set of window positions on the screen, the objects to be displayed in each window, and the window priorities. The individual objects are to be clipped to fit into their respective windows, then clipped to remove parts with overlapping windows of higher display priority.

CHAPTER

7 Structures and Hierarchical Modeling



For a great many applications, it is convenient to be able to create and manipulate individual parts of a picture without affecting other picture parts. Most graphics packages provide this capability in one form or another. With the ability to define each object in a picture as a separate module, we can make modifications to the picture more easily. In design applications, we can try out different positions and orientations for a component of a picture without disturbing other parts of the picture. Or we can take out parts of the picture, then we can easily put the parts back into the display at a later time. Similarly, in modeling applications, we can separately create and position the subparts of a complex object or system into the overall hierarchy. And in animations, we can apply transformations to individual parts of the scene so that one object can be animated with one type of motion, while other objects in the scene move differently or remain stationary.

7-1 STRUCTURE CONCEPTS

A labeled set of output primitives (and associated attributes) in PHIGS is called a **structure**. Other commonly used names for a labeled collection of primitives are *segments* (GKS) and *objects* (Graphics Library on Silicon Graphics systems). In this section, we consider the basic structure-managing functions in PHIGS. Similar operations are available in other packages for handling labeled groups of primitives in a picture.

Basic Structure Functions

When we create a structure, the coordinate positions and attribute values specified for the structure are stored as a labeled group in a system structure list called the **central structure store**. We create a structure with the function

```
openStructure (id)
```

The label for the segment is the positive integer assigned to parameter `id`. In PHIGS+, we can use character strings to label the structures instead of using integer names. This makes it easier to remember the structure identifiers. After all primitives and attributes have been listed, the end of the structure is signaled with the `closeStructure` statement. For example, the following program

statements define structure 6 as the line sequence specified in polyline with the designated line type and color:

```
openStructure (6);
     setLinetype (lt);
     setPolylineColourIndex (lc);
     polyline (n, pts);
closeStructure;
```

Any number of structures can be created for a picture, but only one structure can be open (in the creation process) at a time. Any open structure must be closed before a new structure can be created. This requirement eliminates the need for a structure identification number in the `closeStructure` statement.

Once a structure has been created, it can be displayed on a selected output device with the function

```
postStructure (ws, id, priority)
```

where parameter ws is the workstation identifier, `id` is the structure name, and `priority` is assigned a real value in the range from 0 to 1. Parameter `priority` sets the display priority relative to other structures. When two structures overlap on an output display device, the structure with the higher priority will be visible. For example, if structures 6 and 9 are posted to workstation 2 with the following priorities

```
postStructure (2, 6, 0.8)
postStructure (2, 9, 0.1)
```

then any parts of structure 9 that overlap structure 6 will be hidden, since structure 6 has higher priority. If two structures are assigned the same priority value, the last structure to be posted is given display precedence

When a structure is posted to an active workstation, the primitives in the structure are scanned and interpreted for display on the selected output device (video monitor, laser printer, etc.). Scanning a structure list and sending the graphical output to a workstation is called **traversal**. A list of current attribute values for primitives is stored in a data structure called a **traversal state list**. As changes are made to posted structures, both the system structure list and the traversal state list are updated. This automatically modifies the display of the posted structures on the workstation.

To remove the display of a structure from a particular output device, we invoke the function

```
unpostStructure (ws, id)
```

This deletes the structure from the active list of structures for the designated output device, but the system structure list is not affected. On a raster system, a structure is removed from the display by redrawing the primitives in the background color. This process, however, may also affect the display of primitives from other structures that overlap the structure we want to erase. To remedy this, we can use the coordinate extents of the various structures in a scene to deter-

mine which ones overlap the structure we are erasing. Then we can simply redraw these overlapping structures after we have erased the structure that is to be unposted. All structures can be removed from a selected output device with

```
unpostAllStructures (ws)
```

If we want to remove a particular structure from the system structure list, we accomplish that with the function

```
deleteStructure (id)
```

Of course, this also removes the display of the structure from all posted output devices. Once a structure has been deleted, its name can be reused for another set of primitives. The entire system structure list can be cleared with

```
deleteAllStructures
```

It is sometimes useful to be able to relabel a structure. This is accomplished with

```
changeStructureIdentifier (oldID, newID)
```

One reason for changing a structure label is to consolidate the numbering of the structures after several structures have been deleted. Another is to cycle through a set of structure labels while displaying a structure in multiple locations to test the structure positioning.

Setting Structure Attributes

We can set certain display characteristics for structures with **workstation filters**. The three properties we can set with filters are visibility, highlighting, and the capability of a structure to be selected with an interactive input device.

Visibility and invisibility settings for structures on a particular workstation for a selected device are specified with the function

```
setInvisibilityFilter (ws, devCode, invisSet, visSet)
```

where parameter `invisSet` contains the names of structures that will be invisible, and parameter `visSet` contains the names of those that will be visible. With the invisibility filter, we can turn the display of structures on and off at selected workstations without actually deleting them from the workstation lists. This allows us, for example, to view the outline of a building without all the interior details; and then to reset the visibility so that we can view the building with all internal features included. Additional parameters that we can specify are the number of structures for each of the two sets. Structures are made invisible on a raster monitor using the same procedures that we discussed for unposting and for deleting a structure. The difference, however, is that we do not remove the structure from the active structure list for a device when we are simply making it invisible.

Highlighting is another convenient structure characteristic. In a map display, we could highlight all cities with populations below a certain value; or for a

landscape layout, we could highlight certain varieties of shrubbery; or in a circuit diagram, we could highlight all components within a specific voltage range. This is done with the function

```
setHighlightingFilter (ws, devCode, highlightSet,
                       nohighlightSet)
```

Parameter highlightSet contains the names of the structures that are to be highlighted, and parameter nohighlightSet contains the names of those that are not to be highlighted. The kind of highlighting used to accent structures depends on the type and capabilities of the graphics system. For a color video monitor, highlighted structures could be displayed in a brighter intensity or in a color reserved for highlighting. Another common highlighting implementation is to turn the visibility on and off rapidly so that blinking structures are displayed. Blinking can also be accomplished by rapidly alternating the intensity of the highlighted structures between a low value and a high value.

The third display characteristic we can set for structures is *pickability*. This refers to the capability of the structure to be selected by pointing at it or positioning the screen cursor over it. If we want to be sure that certain structures in a display can never be selected, we can declare them to be nonpickable with the pickability filter. In the next chapter, we take up the topic of input methods in more detail.

7-2 EDITING STRUCTURES

Often, we would like to modify a structure after it has been created and closed. Structure modification is needed in design applications to try out different graphical arrangements, or to change the design configuration in response to new test data.

If additional primitives are to be added to a structure, this can be done by simply reopening the structure with the openStructure function and appending the required statements. As an example of simple appending, the following program segment first creates a structure with a single fill area and then adds a second fill area to the structure:

```
openStructure (shape);
     setInteriorStyle (solid);
     setInteriorColourIndex (4);
     fillArea (n1, verts1);
closeStructure;
          .
          .
          .
openStructure (shape);
     setInteriorStyle (hollow);
     fillArea (n2, verts2);
closeStructure;
```

This sequence of operations is equivalent to initially creating the structure with both fill areas:

```
openStructure (shape);
     setInteriorStyle (solid);
     setInteriorColourIndex (4);
     fillArea (n1, verts1);
     setInteriorStyle (hollow);
     fillArea (n2, verts2);
closeStructure;
```

In addition to appending, we may also want sometimes to delete certain items in a structure, to change primitives or attribute settings, or to insert items at selected positions within the structure. General editing operations are carried out by accessing the sequence numbers for the individual components of a structure and setting the edit mode.

Structure Lists and the Element Pointer

Individual items in a structure, such as output primitives and attribute values, are referred to as **structure elements**, or simply **elements**. Each element is assigned a reference position value as it is entered into the structure. Figure 7-1 shows the storage of structure elements and associated position numbers created by the following program segment.

```
openStructure (gizmo);
     setLinetype (lt1);
     setPolylineColourIndex (lc1);
     polyline (n1, pts1);
     setLinetype (lt2);
     setPolylineColourIndex (lc2);
     polyline (n2, pts2);
closeStructure;
```

Structure elements are numbered consecutively with integer values starting at 1, and the value 0 indicates the position just before the first element. When a structure is opened, an **element pointer** is set up and assigned a position value that can be used to edit the structure. If the opened structure is new (not already existing in the system structure list), the element pointer is set to 0. If the opened structure does already exist in the system list, the element pointer is set to the position value of the last element in the structure. As elements are added to a structure, the element pointer is incremented by 1.

We can set the value of the element pointer to any position within a structure with the function

```
setElementPointer (k)
```

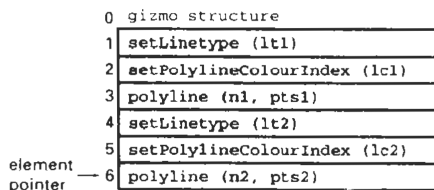


Figure 7-1
Element position values for structure gizmo.

where parameter k can be assigned any integer value from 0 to the maximum number of elements in the structure. It is also possible to position the element pointer using the following offset function that moves the pointer relative to the current position:

```
offsetElementPointer (dk)
```

with dk assigned a positive or negative integer offset from the present position of the pointer. Once we have positioned the element pointer, we can edit the structure at that position.

Setting the Edit Mode

Structures can be modified in one of two possible modes. This is referred to as the **edit mode** of the structure. We set the value of the edit mode with

```
setEditMode (mode)
```

where parameter mode is assigned either the value *insert*, or the value *replace*.

Inserting Structure Elements

When the edit mode is set to *insert*, the next item entered into a structure will be placed in the position immediately following the element pointer. Elements in the structure list following the inserted item are then automatically renumbered.

To illustrate the insertion operation, let's change the standard line width currently in structure gizmo (Fig. 7-2) to some other value. We can do this by inserting a line width statement anywhere before the first polyline command:

```
openStructure (gizmo);
     setEditMode (insert);
     setElementPointer (0);
     setLinewidth (lw);
               .
               .
               .
closeStructure;
```

Figure 7-2 shows the modified element list of gizmo, created by the previous insert operation. After this insert, the element pointer is assigned the value 1 (the position of the new line-width attribute). Also, all elements after the line-width statement have been renumbered, starting at the value 2.

		gizmo structure
	0	
element pointer →	1	setLinewidth (lw)
	2	setLinetype (lt1)
	3	setPolylineColourIndex (lc1)
	4	polyline (n1, pts1)
	5	setLinetype (lt2)
	6	setPolylineColourIndex (lc2)
	7	polyline (n2, pts2)

Figure 7-2
Modified element list and position of the element pointer after inserting a line-width attribute into structure gizmo.

When a new structure is created, the edit mode is automatically set to the value *insert*. Assuming the edit mode has not been changed from this default value before we reopen this structure, we can append items at the end of the element list without setting values for either the edit mode or element pointer, as demonstrated at the beginning of Section 7-2. This is because the edit mode remains at the value *insert* and the element pointer for the reopened structure points to the last element in the list.

Replacing Structure Elements

When the edit mode is set to the value *replace*, the next item entered into a structure is placed at the position of the element pointer. The element originally at that position is deleted, and the value of the element pointer remains unchanged.

As an example of the replace operation, suppose we want to change the color of the second polyline in structure gizmo (Fig. 7-1). We can do this with the sequence:

```
openStructure (gizmo);
     setEditMode (replace);
     setElementPointer (5);
     setPolylineColourIndex (lc2New);
           .
           .
           .
closeStructure;
```

Figure 7-3 shows the element list of gizmo with the new color for the second polyline. After the replace operation, the element pointer remains at position 5 (the position of the new line color attribute).

Deleting Structure Elements

We can delete the element at the current position of the element pointer with the function

```
deleteElement
```

This removes the element from the structure and sets the value of the element pointer to the immediately preceding element.

As an example of element deletion, suppose we decide to have both polylines in structure gizmo (Fig. 7-1) displayed in the same color. We can accomplish this by deleting the second color attribute:

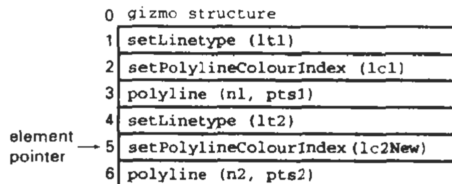


Figure 7-3
Modified element list and position of the element pointer after changing the color of the second polyline in structure gizmo.

```
openStructure (gizmo);
     setElementPointer (5);
     deleteElement;
          .
          .
          .
closeStructure;
```

The element pointer is then reset to the value 4 and all following elements are renumbered, as shown in Fig. 7-4.

A contiguous group of structure elements can be deleted with the function

```
deleteElementRange (k1, k2)
```

where integer parameter k1 gives the beginning position number, and k2 specifies the ending position number. For example, we can delete the second polyline and associated attributes in structure gizmo with

```
deleteElementRange (4, 6)
```

And all elements in a structure can be deleted with the function

```
emptyStructure (id)
```

Labeling Structure Elements

Once we have made a number of modifications to a structure, we could easily lose track of the element positions. Deleting and inserting elements shift the element position numbers. To avoid having to keep track of new position numbers as modifications are made, we can simply label the different elements in a structure with the function

```
label (k)
```

where parameter k is an integer position identifier. Labels can be inserted anywhere within the structure list as an aid to locating structure elements without referring to position number. The label function creates structure elements that have no effect on the structure traversal process. We simply use the labels stored in the structure as editing references rather than using the individual element positions. Also, the labeling of structure elements need not be unique. Sometimes it is convenient to give two or more elements the same label value, particularly if the same editing operations are likely to be applied to several positions in the structure.

```
                  0  gizmo structure
                  1  setLinetype (lt1)
                  2  setPolylineColourIndex (lc1)
                  3  polyline (n1, pts1)
element pointer → 4  setLinetype (lt2)
                  5  polyline (n2, pts2)
```

Figure 7-4
Modified element list and position of the element pointer after deleting the color-attribute statement for the second polyline in structure gizmo.

To illustrate the use of labeling, we create structure `labeledGizmo` in the following routine that has the elements and position numbers as shown in Fig. 7-5.

```
openStructure (labeledGizmo);
     label (object1Linetype);
     setLinetype (lt1);
     label (object1Color);
     setPolylineColourIndex (lc1);
     label (object1);
     polyline (n1, pts1);
     label (object2Linetype);
     setLinetype (lt2);
     label (object2Color);
     setPolylineColourIndex (lc2);
     label (object2);
     polyline (n2, pts2);
closeStructure;
```

Now if we want to change any of the primitives or attributes in this structure, we can do it by referencing the labels. Although we have labeled every item in this structure, other labeling schemes could be used depending on what type and how much editing is anticipated. For example, all attributes could be lumped under one label, or all color attributes could be given the same label identifier.

A label is referenced with the function

```
setElementPointerAtLabel (k)
```

which sets the element pointer to the value of parameter `k`. The search for the label begins at the current element-pointer position and proceeds forward through the element list. This means that we may have to reset the pointer when reopening a structure, since the pointer is always positioned at the last element in a reopened structure, and label searching is not done backward through the element list. If, for instance, we want to change the color of the second object in structure `labeledGizmo`, we could reposition the pointer at the start of the element list after reopening the structure to search for the appropriate color attribute statement label:

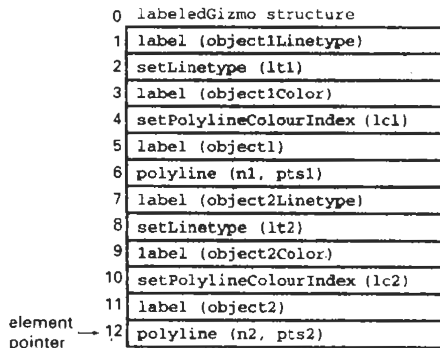


Figure 7-5
A set of labeled objects and associated position numbers stored in structure `labeledGizmo`.

```
openStructure (labeledGizmo);
     setElementPointer (0);
     setEditMode (replace);
     setElementPointerAtLabel (object2Color);
     offsetElementPointer (1);
     setPolylineColourIndex (lc2New);
          .
          .
          .
closeStructure;
```

Deleting an item referenced with a label is similar to the replacement operation illustrated in the last openStructure routine. We first locate the appropriate label and then offset the pointer. For example, the color attribute for the second polyline in structure labeledGizmo can be deleted with the sequence

```
openStructure (labeledGizmo);
     setElementPointer (0);
     setEditMode (replace);
     setElementPointerAtLabel (object2Color);
     offsetElementPointer (1);
     deleteElement;
          .
          .
          .
closeStructure;
```

We can also delete a group of structure elements between specified labels with the function

```
deleteElementsBetweenLabels (k1, k2)
```

After the set of elements is deleted, the element pointer is set to position k1.

Copying Elements from One Structure to Another

We can copy all the entries from a specified structure into an open structure with

```
copyAllElementsFromStructure (id)
```

The elements from structure id are inserted into the open structure starting at the position immediately following the element pointer, regardless of the setting of the edit mode. When the copy operation is complete, the element pointer is set to the position of the last item inserted into the open structure.

7-3 BASIC MODELING CONCEPTS

An important use of structures is in the design and representation of different types of systems. Architectural and engineering systems, such as building layouts and electronic circuit schematics, are commonly put together using computer-aided design methods. Graphical methods are used also for representing economic, financial, organizational, scientific, social, and environmental systems. Representations for these systems are often constructed to simulate the behavior

of a system under various conditions. The outcome of the simulation can serve as an instructional tool or as a basis for making decisions about the system. To be effective in these various applications, a graphics package must possess efficient methods for constructing and manipulating the graphical system representations.

The creation and manipulation of a system representation is termed **modeling**. Any single representation is called a **model** of the system. Models for a system can be defined graphically, or they can be purely descriptive, such as a set of equations that defines the relationships between system parameters. Graphical models are often referred to as **geometric models**, because the component parts of a system are represented with geometric entities such as lines, polygons, or circles. We are concerned here only with graphics applications, so we will use the term model to mean a computer-generated geometric representation of a system.

Model Representations

Figure 7-6 shows a representation for a logic circuit, illustrating the features common to many system models. Component parts of the system are displayed as geometric structures, called **symbols**, and relationships between the symbols are represented in this example with a network of connecting lines. Three standard symbols are used to represent logic gates for the Boolean operations: *and*, *or*, and *not*. The connecting lines define relationships in terms of input and output flow (from left to right) through the system parts. One symbol, the *and* gate, is displayed at two different positions within the logic circuit. Repeated positioning of a few basic symbols is a common method for building complex models. Each such occurrence of a symbol within a model is called an **instance** of that symbol. We have one instance for the *or* and *not* symbols in Fig. 7-6 and two instances of the *and* symbol.

In many cases, the particular graphical symbols chosen to represent the parts of a system are dictated by the system description. For circuit models, standard electrical or logic symbols are used. With models representing abstract concepts, such as political, financial, or economic systems, symbols may be any convenient geometric pattern.

Information describing a model is usually provided as a combination of geometric and nongeometric data. Geometric information includes coordinate positions for locating the component parts, output primitives and attribute functions to define the structure of the parts, and data for constructing connections between the parts. Nongeometric information includes text labels, algorithms describing the operating characteristics of the model, and rules for determining the relationships or connections between component parts, if these are not specified as geometric data.

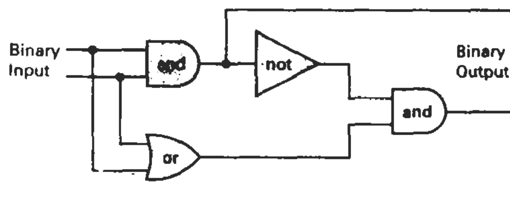


Figure 7-6
Model of a logic circuit.

There are two methods for specifying the information needed to construct and manipulate a model. One method is to store the infomation in a data structure, such as a table or linked list. The other method is to specify the information in procedures. In general, a model specification will contain both data structures and procedures, although some models are defined completely with data structures and others use only procedural specifications. An application to perform solid modeling of objects might use mostly information taken from some data structure to define coordinate positions, with very few procedures. A weather model, on the other hand, may need mostly procedures to calculate plots of temperature and pressure variations.

As an example of how combinations of data structures and procedures can be used, we consider some alternative model specifications for the logic circuit of Fig. 7-6. One method is to define the logic components in a data table (Table 7-1), with processing procedures used to specify how the network connections are to be made and how the circuit operates. Geometric data in this table include coordinates and parameters necessary for drawing and positioning the gates. These symbols could all be drawn as polygon shapes, or they could be formed as combinations of straight-line segments and elliptical arcs. Labels for each of the component parts also have been included in the table, although the labels could be omitted if the symbols are displayed as commonly recognized shapes. Procedures would then be used to display the gates and construct the connecting lines, based on the coordinate positions of the gates and a specified order for connecting them. An additional procedure is used to produce the circuit output (binary values) for any given input. This procedure could be set up to display only the final output, or it could be designed to display intermediate output values to illustrate the internal functioning of the circuit.

Alternatively, we might specify graphical information for the circuit model in data structures. The connecting lines, as well as the gates, could then be defined in a data table that explicitly lists endpoints for each of the lines in the circuit. A single procedure might then display the circuit and calculate the output. At the other extreme, we could completely define the model in procedures, using no external data structures.

Symbol Hierarchies

Many models can be organized as a hierarchy of symbols. The basic "building blocks" for the model are defined as simple geometric shapes appropriate to the type of model under consideration. These basic symbols can be used to form composite objects, called **modules**, which themselves can be grouped to form higher-level modules, and so on, for the various components of the model. In the

TABLE 7-1
A DATA TABLE DEFINING THE STRUCTURE AND POSITION OF EACH GATE IN THE CIRCUIT OF FIG. 7-6

Symbol Code	*Geometric Description*	*Identifying Label*
Gate 1	(Coordinates and other parameters)	and
Gate 2	⋮	or
Gate 3	⋮	not
Gate 4	⋮	and

simplest case, we can describe a model by a one-level hierarchy of component parts, as in Fig. 7-7. For this circuit example, we assume that the gates are positioned and connected to each other with straight lines according to connection rules that are specified with each gate description. The basic symbols in this hierarchical description are the logic gates. Although the gates themselves could be described as hierarchies—formed from straight lines, elliptical arcs, and text—that sort of description would not be a convenient one for constructing logic circuits, in which the simplest building blocks are gates. For an application in which we were interested in designing different geometric shapes, the basic symbols could be defined as straight-line segments and arcs.

An example of a two-level symbol hierarchy appears in Fig. 7-8. Here a facility layout is planned as an arrangement of work areas. Each work area is outfitted with a collection of furniture. The basic symbols are the furniture items: worktable, chair, shelves, file cabinet, and so forth. Higher-order objects are the work areas, which are put together with different furniture organizations. An instance of a basic symbol is defined by specifying its size, position, and orientation within each work area. For a facility-layout package with fixed sizes for objects, only position and orientation need be specified by a user. Positions are given as coordinate locations in the work areas, and orientations are specified as rotations that determine which way the symbols are facing. At the second level up the hierarchy, each work area is defined by specifying its size, position, and orientation within the facility layout. The boundary for each work area might be fitted with a divider that encloses the work area and provides aisles within the facility.

More complex symbol hierarchies are formed by repeated grouping of symbol clusters at each higher level. The facility layout of Fig. 7-8 could be extended to include symbol clusters that form different rooms, different floors of a building, different buildings within a complex, and different complexes at widely separated physical locations.

Modeling Packages

Some general-purpose graphics systems, GKS, for example, are not designed to accommodate extensive modeling applications. Routines necessary to handle modeling procedures and data structures are often set up as separate modeling packages, and graphics packages then can be adapted to interface with the modeling package. The purpose of graphics routines is to provide methods for gener-

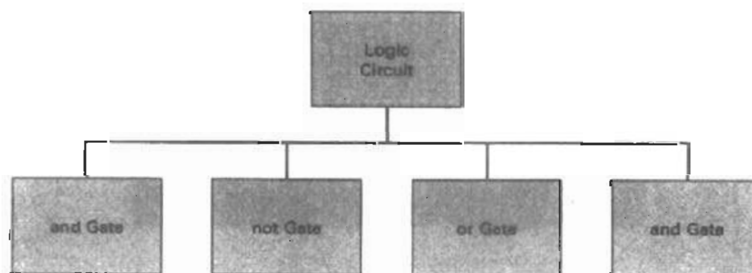


Figure 7-7
A one-level hierarchical description of a circuit formed with logic gates.

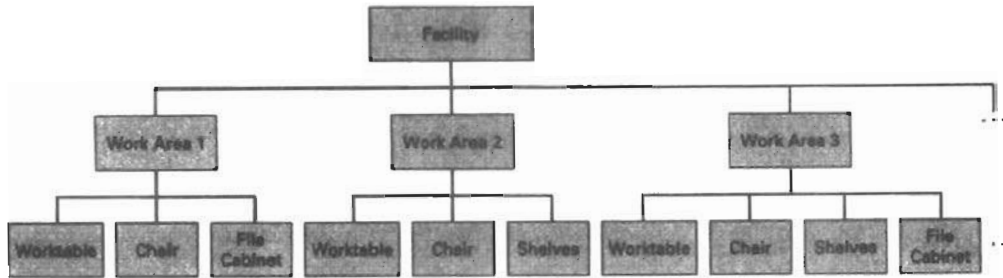


Figure 7-8
A two-level hierarchical description of a facility layout.

ating and manipulating final output displays. Modeling routines, by contrast, provide a means for defining and rearranging model representations in terms of symbol hierarchies, which are then processed by the graphics routines for display. Systems, such as PHIGS and Graphics Library (GL) on Silicon Graphics equipment, are designed so that modeling and graphics functions are integrated into one package.

Symbols available in an application modeling package are defined and structured according to the type of application the package has been designed to handle. Modeling packages can be designed for either two-dimensional or three-dimensional displays. Figure 7-9 illustrates a two-dimensional layout used in circuit design. An example of three-dimensional molecular modeling is shown in Fig. 7-10, and a three-dimensional facility layout is given in Fig. 7-11. Such three-dimensional displays give a designer a better appreciation of the appearance of a layout. In the following sections, we explore the characteristic features of modeling packages and the methods for interfacing or integrating modeling functions with graphics routines.

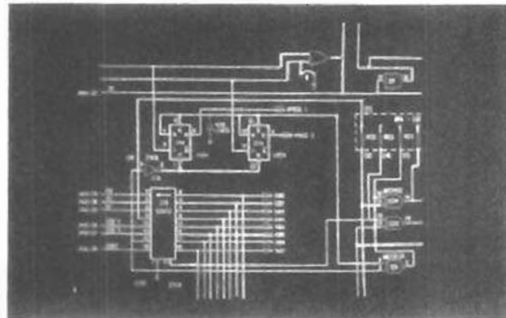


Figure 7-9
Two-dimensional modeling layout used in circuit design. *(Courtesy of Summagraphics)*

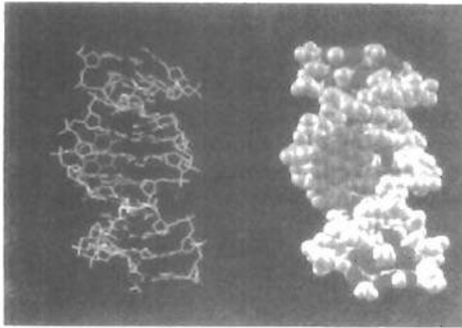


Figure 7-10
One-half of a stereoscopic image pair showing a three-dimensional molecular model of DNA. Data supplied by Tamar Schlick, NYU, and Wilma K. Olson, Rutgers University; visualization by Jerry Greenberg, SDSC. *(Courtesy of Stephanie Sides, San Diego Supercomputer Center.)*

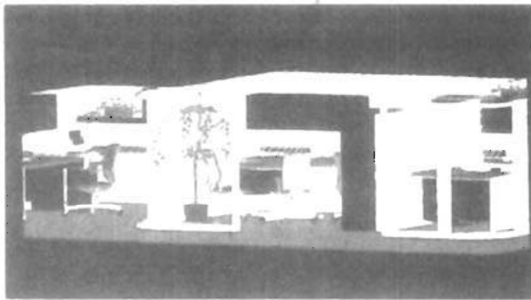


Figure 7-11
A three-dimensional view of an office layout. *Courtesy of Intergraph Corporation.*

7-4 HIERARCHICAL MODELING WITH STRUCTURES

A hierarchical model of a system can be created with structures by nesting the structures into one another to form a tree organization. As each structure is placed into the hierarchy, it is assigned an appropriate transformation so that it will fit properly into the overall model. One can think of setting up an office facility in which furniture is placed into the various offices and work areas, which in turn are placed into departments, and so forth on up the hierarchy.

Local Coordinates and Modeling Transformations

In many design applications, models are constructed with instances (transformed copies) of the geometric shapes that are defined in a basic symbol set. Instances are created by positioning the basic symbols within the world-coordinate reference of the model. The various graphical symbols to be used in an application are each defined in an independent coordinate reference called the **modeling-coordinate system**. Modeling coordinates are also referred to as **local coordinates**, or sometimes **master coordinates**. Figure 7-12 illustrates local coordinate definitions

for two symbols that could be used in a two-dimensional facility-layout application.

To construct the component parts of a graphical model, we apply transformations to the local-coordinate definitions of symbols to produce instances of the symbols in world coordinates. Transformations applied to the modeling-coordinate definitions of symbols are referred to as **modeling transformations**. Typically, modeling transformations involve translation, rotation, and scaling to position a symbol in world coordinates, but other transformations might also be used in some applications.

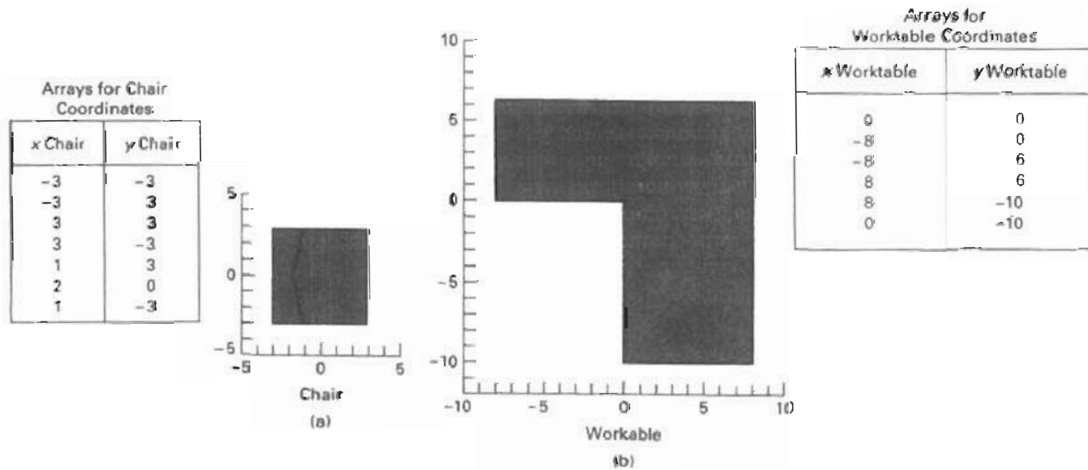
Modeling Transformations

We obtain a particular modeling-transformation matrix using the geometric-transformation functions discussed in Chapter 5. That is, we can set up the individual transformation matrices to accomplish the modeling transformation, or we can input the transformation parameters and allow the system to build the matrices. In either case, the modeling package concatenates the individual transformations to construct a homogeneous-coordinate modeling transformation matrix, **MT**. An instance of a symbol in world coordinates is then produced by applying **MT** to modeling-coordinate positions ($\mathbf{P}_{mc}$) to generate corresponding world-coordinate positions ($\mathbf{P}_{wc}$):

$$\mathbf{P}_{wc} = \mathbf{MT} \cdot \mathbf{P}_{mc} \tag{7-1}$$

Structure Hierarchies

As we have seen, modeling applications typically require the composition of basic symbols into groups, called modules; these modules may be combined into



Arrays for Chair Coordinates

x Chair	y Chair
-3	-3
-3	3
3	3
3	-3
1	3
2	0
1	-3

Arrays for Worktable Coordinates

x Worktable	y Worktable
0	0
-8	0
-8	6
8	6
8	-10
0	-10

Figure 7-12
Objects defined in local coordinates.

higher-level modules; and so on. Such symbol hierarchies can be created by embedding structures within structures at each successive level in the tree. We can first define a module (structure) as a list of symbol instances and their transformation parameters. At the next level, we define each higher-level module as a list of the lower-module instances and their transformation parameters. This process is continued up to the root of the tree, which represents the total picture in world coordinates.

A structure is placed within another structure with the function

```
executeStructure (id)
```

To properly orient the structure, we first assign the appropriate local transformation to structure `id`. This is done with

```
setLocalTransformation (mlt, type)
```

where parameter `mlt` specifies the transformation matrix. Parameter `type` is assigned one of the following three values: *pre*, *post*, or *replace*, to indicate the type of matrix composition to be performed with the current modeling-transformation matrix. If we simply want to replace the current transformation matrix with `lmt`, we set parameter `type` to the value *replace*. If we want the current matrix to be premultipled with the local matrix we are specifying in this function, we choose *pre*; and similarly for the value *post*. The following code section illustrates a sequence of modeling statements to set the first instance of an object into the hierarchy below the root node.

```
createStructure (id0);
     setLocalTransformation (lmt, type);
     executeStructure (id1);


closeStructure;
```

The same procedure is used to instance other objects within structure `id0` to set the other nodes into this level of the hierarchy. Then we can create the next level down the tree by instancing objects within structure `id1` and the other structures that are in `id0`. We repeat this process until the tree is complete. The entire tree is then displayed by posting the root node: structure `id0` in the previous example. In the following procedure, we illustrate how a hierarchical structure can be used to model an object.

```
void main ()
{
  enum { Frame, Wheel, Bicycle };
  int nPts;
  wcPt2 pts[256];
  pMatrix3 m;
  /* Routines to generate geometry */
  extern void getWheelVertices (int * nPts, wcPt2 pts);
  extern void getFrameVertices (int * nPts, wcPt2 pts);

  /* Make the wheel structure */
```

```
    getWheelVertices (nPts, pts);
    openStructure (Wheel);
    setLineWidth (2.0);
    polyline (nPts, pts);
    closeStructure;
    /* Make the frame structure */
    getFrameVertices (nPts, pts);
    openStructure (Frame);
    setLineWidth (2.0);
    polyline (nPts, pts);
    closeStructure;

    /* Make the bicycle */
    openStructure (Bicycle);
    /* Include the frame */
    executeStructure (Frame);
    /* Position and include rear wheel */
    matrixSetIdentity (m);
    m[0,2] := -1.0; m[1,2] := -0.5;
    setLocalTransformationMatrix (m, REPLACE);
    executeStructure (Wheel);
    /* Position and include front wheel */
    m[0,2] := 1.0; m[1,2] := -0.5;
    setLocalTransformationmatrix (m, REPLACE);
    executeStructure (Wheel);
    closeStructure;
}
```

We delete a hierarchy with the function

```
deleteStructureNetwork (id)
```

where parameter id references the root structure of the tree. This deletes the root node of the hierarchy and all structures that have been placed below the root using the executeStructure function, assuming that the hierarchy is organized as a tree.

SUMMARY

A structure (also called a segment or an object in some systems) is a labeled group of output statements and associated attributes. By designing pictures as sets of structures, we can easily add, delete, or manipulate picture components independently of each another. As structures are created, they are entered into a central structure store. Structures are then displayed by posting them to various output devices with assigned priorities. When two structures overlap, the structure with the higher priority is displayed over the structure with the lower priority.

We can use workstation filters to set attributes, such as visibility and highlighting, for structures. With the visibility filter, we can turn off the display of a structure while retaining it in the structure list. The highlighting filter is used to emphasize a displayed structure with blinking, color, or high-intensity patterns.

Various editing operations can be applied to structures. We can reopen structures to carry out append, insert, or delete operations. Locations in a structure are referenced with the element pointer. In addition, we individually label the primitives or attributes in a structure.

The term model, in graphics applications, refers to a graphical representation for some system. Components of the system are represented as symbols, defined in local (modeling) coordinate reference frames. Many models, such as electrical circuits, are constructed by placing instances of the symbols at selected locations.

Many models are constructed as symbol hierarchies. A bicycle, for instance, can be constructed with a bicycle frame and the wheels. The frame can include such parts as the handlebars and the pedals. And the wheels contain spokes, rims, and tires. We can construct a hierarchial model by nesting structures. For example, we can set up a bike structure that contains a frame structure and a wheel structure. Both the frame and wheel structures can then contain primitives and additional structures. We continue this nesting down to structures that contain only output primitives (and attributes).

As each structure is nested within another structure, an associated modeling transformation can be set for the nested structure. This transformation describes the operations necessary to properly orient and scale the structure to fit into the hierarchy.

REFERENCES

Structure operations and hierarchical modeling in PHIGS are discussed in Hopgood and Duce (1991), Howard et al. (1991), Gaskins (1992), and Blake (1993).

For information on GKS segment operations see Hopgood (1983) and Enderle et al. (1984)

EXERCISES

7-1. Write a procedure for creating and manipulating the information in a central structure store. This procedure is to be invoked by functions such as `openStructure`, `deleteStructure`, and `changeStructureIdentifier`.

7-2. Write a routine for storing information in a traversal state list.

7-3. Write a routine for erasing a specified structure on a raster system, given the coordinate extents for all displayed structures in a scene

7-4. Write a procedure to implement the `unpostStructure` function on a raster system.

7-5. Write a procedure to implement the `deleteStructure` function on a raster system.

7-6. Write a procedure to implement highlighting as a blinking operation.

7-7. Write a set of routines for editing structures. Your routines should provide for the following types of editing: appending, inserting, replacing, and deleting structure elements.

7-8. Discuss model representations that would be appropriate for several distinctly different kinds of systems. Also discuss how graphical representations might be implemented for each system.

7-9 For a logic-circuit modeling application, such as that in Fig. 7-6, give a detailed graphical description of the standard logic symbols to be used in constructing a display of a circuit.

7-10. Develop a modeling package for electrical design that will allow a user to position electrical symbols within a circuit network. Only translations need be applied to place an instance of one of the electrical menu shapes into the network. Once a component has been placed in the network, it is to be connected to other specified components with straight line segments.

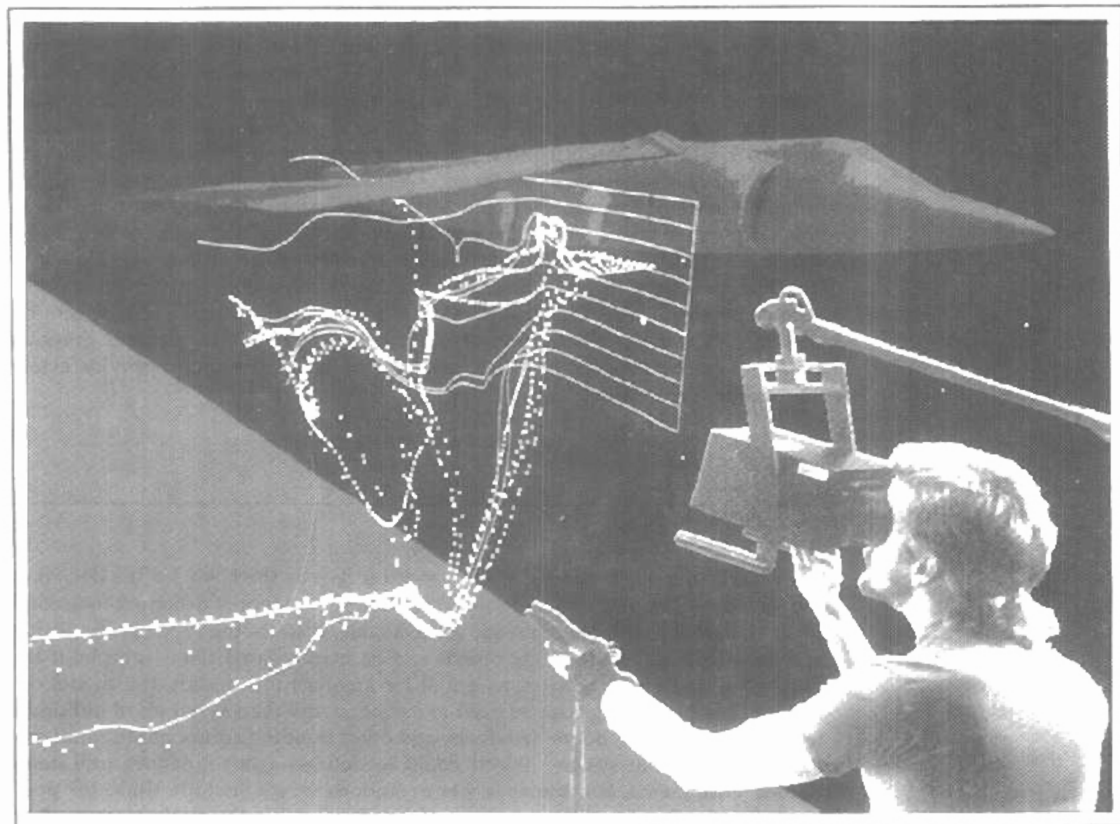
7-11 Devise a two-dimensional facility-layout package. A menu of furniture shapes is to be

provided to a designer, who can place the objects in any location within a single room (one-level hierarchy). Instance transformations can be limited to translations and rotations.

7-12. Devise a two-dimensional facility-layout package that presents a menu of furniture shapes. A two-level hierarchy is to be used so that furniture items can be placed into various work areas, and the work areas can be arranged within a larger area. Instance transformations may be limited to translations and rotations, but scaling could be used if furniture items of different sizes are to be available.

CHAPTER

8 Graphical User Interfaces and Interactive Input Methods



The human–computer interface for most systems involves extensive graphics, regardless of the application. Typically, general systems now consist of windows, pull-down and pop-up menus, icons, and pointing devices, such as a mouse or spaceball, for positioning the screen cursor. Popular graphical user interfaces include X Windows, Windows, Macintosh, OpenLook, and Motif. These interfaces are used in a variety of applications, including word processing, spreadsheets, databases and file-management systems, presentation systems, and page-layout systems. In graphics packages, specialized interactive dialogues are designed for individual applications, such as engineering design, architectural design, data visualization, drafting, business graphs, and artist's paintbrush programs. For general graphics packages, interfaces are usually provided through a standard system. An example is the X Window System interface with PHIGS. In this chapter, we take a look at the basic elements of graphical user interfaces and the techniques for interactive dialogues. We also consider how dialogues in graphics packages, in particular, can allow us to construct and manipulate picture components, select menu options, assign parameter values, and select and position text strings. A variety of input devices exists, and general graphics packages can be designed to interface with various devices and to provide extensive dialogue capabilities.

8-1 THE USER DIALOGUE

For a particular application, the *user's model* serves as the basis for the design of the dialogue. The user's model describes what the system is designed to accomplish and what graphics operations are available. It states the type of objects that can be displayed and how the objects can be manipulated. For example, if the graphics system is to be used as a tool for architectural design, the model describes how the package can be used to construct and display views of buildings by positioning walls, doors, windows, and other building components. Similarly, for a facility-layout system, objects could be defined as a set of furniture items (tables, chairs, etc.), and the available operations would include those for positioning and removing different pieces of furniture within the facility layout. And a circuit-design program might use electrical or logic elements for objects, with positioning operations available for adding or deleting elements within the overall circuit design

All information in the user dialogue is then presented in the language of the application. In an architectural design package, this means that all interactions are described only in architectural terms, without reference to particular data structures or other concepts that may be unfamiliar to an architect. In the following sections, we discuss some of the general considerations in structuring a user dialogue.

Windows and Icons

Figure 8-1 shows examples of common window and icon graphical interfaces. Visual representations are used both for objects to be manipulated in an application and for the actions to be performed on the application objects.

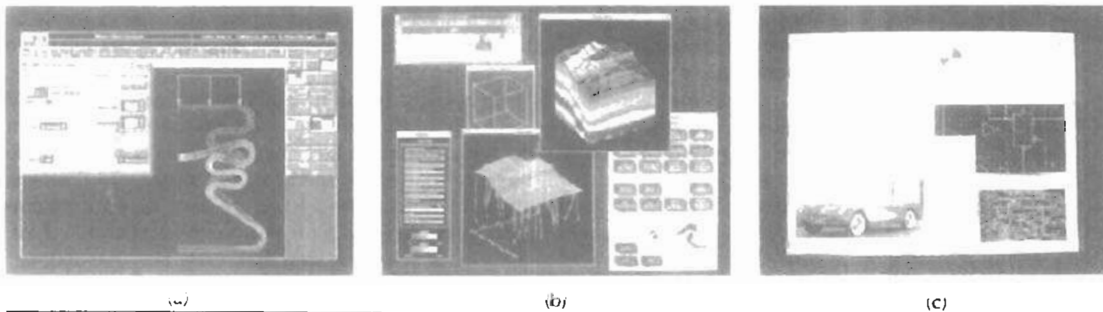
A window system provides a window-manager interface for the user and functions for handling the display and manipulation of the windows. Common functions for the window system are opening and closing windows, repositioning windows, resizing windows, and display routines that provide interior and exterior clipping and other graphics functions. Typically, windows are displayed with sliders, buttons, and menu icons for selecting various window options. Some general systems, such as X Windows and NeWS, are capable of supporting multiple window managers so that different window styles can be accommodated, each with its own window manager. The window managers can then be designed for particular applications. In other cases, a window system is designed for one specific application and window style.

Icons representing objects such as furniture items and circuit elements are often referred to as **application icons**. The icons representing actions, such as rotate, magnify, scale, clip, and paste, are called **control icons, or command icons.**

Accommodating Multiple Skill Levels

Usually, interactive graphical interfaces provide several methods for selecting actions. For example, options could be selected by pointing at an icon and clicking different mouse buttons, or by accessing pull-down or pop-up menus, or by typing keyboard commands. This allows a package to accommodate users that have different skill levels.

For a less experienced user, an interface with a few easily understood operations and detailed prompting is more effective than one with a large, compre-



(a) (b) (c)

Figure 8-1
Examples of screen layouts using window systems and icons. *(Courtesy of (a) Intergraph Corporation, (b) Visual Numerics, Inc., and (c) Sun Microsystems.)*

hensive operation set. A simplified set of menus and options is easy to learn and remember, and the user can concentrate on the application instead of on the details of the interface. Simple point-and-click operations are often easiest for an inexperienced user of an applications package. Therefore, interfaces typically provide a means for masking the complexity of a package, so that beginners can use the system without being overwhelmed with too much detail.

Experienced users, on the other hand, typically want speed. This means fewer prompts and more input from the keyboard or with multiple mouse-button clicks. Actions are selected with function keys or with simultaneous combinations of keyboard keys, since experienced users will remember these shortcuts for commonly used actions.

Similarly, help facilities can be designed on several levels so that beginners can carry on a detailed dialogue, while more experienced users can reduce or eliminate prompts and messages. Help facilities can also include one or more tutorial applications, which provide users with an introduction to the capabilities and use of the system.

Consistency

An important design consideration in an interface is consistency. For example, a particular icon shape should always have a single meaning, rather than serving to represent different actions or objects depending on the context. Some other examples of consistency are always placing menus in the same relative positions so that a user does not have to hunt for a particular option, always using a particular combination of keyboard keys for the same action, and always color coding so that the same color does not have different meanings in different situations.

Generally, a complicated, inconsistent model is difficult for a user to understand and to work with in an effective way. The objects and operations provided should be designed to form a minimum and consistent set so that the system is easy to learn, but not oversimplified to the point where it is difficult to apply.

Minimizing Memorization

Operations in an interface should also be structured so that they are easy to understand and to remember. Obscure, complicated, inconsistent, and abbreviated command formats lead to confusion and reduction in the effectiveness of the use of the package. One key or button used for all delete operations, for example, is easier to remember than a number of different keys for different types of delete operations.

Icons and window systems also aid in minimizing memorization. Different kinds of information can be separated into different windows, so that we do not have to rely on memorization when different information displays overlap. We can simply retain the multiple information on the screen in different windows, and switch back and forth between window areas. Icons are used to reduce memorizing by displaying easily recognizable shapes for various objects and actions. To select a particular action, we simply select the icon that resembles that action.

Backup and Error Handling

A mechanism for backing up, or aborting, during a sequence of operations is another common feature of an interface. Often an operation can be canceled before

execution is completed, with the system restored to the state it was in before the operation was started. With the ability to back up at any point, we can confidently explore the capabilities of the system, knowing that the effects of a mistake can be erased.

Backup can be provided in many forms. A standard *undo* key or command is used to cancel a single operation. Sometimes a system can be backed up through several operations, allowing us to reset the system to some specified point. In a system with extensive backup capabilities, all inputs could be saved so that we can back up and "replay" any part of a session.

Sometimes operations cannot be undone. Once we have deleted the trash in the desktop wastebasket, for instance, we cannot recover the deleted files. In this case, the interface would ask us to verify the delete operation before proceeding.

Good diagnostics and error messages are designed to help determine the cause of an error. Additionally, interfaces attempt to minimize error possibilities by anticipating certain actions that could lead to an error. Examples of this are not allowing us to transform an object position or to delete an object when no object has been selected, not allowing us to select a line attribute if the selected object is not a line, and not allowing us to select the paste operation if nothing is in the clipboard.

Feedback

Interfaces are designed to carry on a continual interactive dialogue so that we are informed of actions in progress at each step. This is particularly important when the response time is high. Without feedback, we might begin to wonder what the system is doing and whether the input should be given again.

As each input is received, the system normally provides some type of response. An object is highlighted, an icon appears, or a message is displayed. This not only informs us that the input has been received, but it also tells us what the system is doing. If processing cannot be completed within a few seconds, several feedback messages might be displayed to keep us informed of the progress of the system. In some cases, this could be a flashing message indicating that the system is still working on the input request. It may also be possible for the system to display partial results as they are completed, so that the final display is built up a piece at a time. The system might also allow us to input other commands or data while one instruction is being processed.

Feedback messages are normally given clearly enough so that they have little chance of being overlooked, but not so overpowering that our concentration is interrupted. With function keys, feedback can be given as an audible click or by lighting up the key that has been pressed. Audio feedback has the advantage that it does not use up screen space, and we do not need to take attention from the work area to receive the message. When messages are displayed on the screen, a fixed message area can be used so that we always know where to look for messages. In some cases, it may be advantageous to place feedback messages in the work area near the cursor. Feedback can also be displayed in different colors to distinguish it from other displayed objects.

To speed system response, feedback techniques can be chosen to take advantage of the operating characteristics of the type of devices in use. A typical raster feedback technique is to invert pixel intensities, particularly when making menu selections. Other feedback methods include highlighting, blinking, and color changes.

Special symbols are designed for different types of feedback. For example, a cross, a frowning face, or a thumbs-down symbol is often used to indicate an error; and a blinking "at work" sign is used to indicate that processing is in progress. This type of feedback can be very effective with a more experienced user, but the beginner may need more detailed feedback that not only clearly indicates what the system is doing but also what the user should input next.

With some types of input, *echo* feedback is desirable. Typed characters can be displayed on the screen as they are input so that we can detect and correct errors immediately. Button and dial input can be echoed in the same way. Scalar values that are selected with dials or from displayed scales are usually echoed on the screen to let us check input values for accuracy. Selection of coordinate points can be echoed with a cursor or other symbol that appears at the selected position. For more precise echoing of selected positions, the coordinate values can be displayed on the screen.

8-2 INPUT OF GRAPHICAL DATA

Graphics programs use several kinds of input data. Picture specifications need values for coordinate positions, values for the character-string parameters, scalar values for the transformation parameters, values specifying menu options, and values for identification of picture parts. Any of the input devices discussed in Chapter 2 can be used to input the various graphical data types, but some devices are better suited for certain data types than others. To make graphics packages independent of the particular hardware devices used, input functions can be structured according to the data description to be handled by each function. This approach provides a **logical input-device classification** in terms of the kind of data to be input by the device.

Logical Classification of Input Devices

The various kinds of input data are summarized in the following six logical device classifications used by PHIGS and GKS:

LOCATOR—a device for specifying a coordinate position (x, y)
STROKE—a device for specifying a series of coordinate positions
STRING—a device for specifying text input
VALUATOR—a device for specifying scalar values
CHOICE—a device for selecting menu options
PICK—a device for selecting picture components

In some packages, a single logical device is used for both locator and stroke operations. Some other mechanism, such as a switch, can then be used to indicate whether one coordinate position or a "stream" of positions is to be input.

Each of the six logical input device classifications can be implemented with any of the hardware devices, but some hardware devices are more convenient for certain kinds of data than others. A device that can be pointed at a screen position is more convenient for entering coordinate data than a keyboard, for example. In the following sections, we discuss how the various physical devices are used to provide input within each of the logical classifications.

Locator Devices

A standard method for interactive selection of a coordinate point is by positioning the screen cursor. We can do this with a mouse, joystick, trackball, spaceball, thumbwheels, dials, a digitizer stylus or hand cursor, or some other cursor-positioning device. When the screen cursor is at the desired location, a button is activated to store the coordinates of that screen point.

Keyboards can be used for locator input in several ways. A general-purpose keyboard usually has four cursor-control keys that move the screen cursor up, down, left, and right. With an additional four keys, we can move the cursor diagonally as well. Rapid cursor movement is accomplished by holding down the selected cursor key. Alternatively, a joystick, joydisk, trackball, or thumbwheels can be mounted on the keyboard for relative cursor movement. As a last resort, we could actually type in coordinate values, but this is a slower process that also requires us to know exact coordinate values.

Light pens have also been used to input coordinate positions, but some special implementation considerations are necessary. Since light pens operate by detecting light emitted from the screen phosphors, some nonzero intensity level must be present at the coordinate position to be selected. With a raster system, we can paint a color background onto the screen. As long as no black areas are present, a light pen can be used to select any screen position. When it is not possible to eliminate all black areas in a display (such as on a vector system, for example), a light pen can be used as a locator by creating a small light pattern for the pen to detect. The pattern is moved around the screen until it finds the light pen.

Stroke Devices

This class of logical devices is used to input a sequence of coordinate positions. Stroke-device input is equivalent to multiple calls to a locator device. The set of input points is often used to display line sections.

Many of the physical devices used for generating locator input can be used as stroke devices. Continuous movement of a mouse, trackball, joystick, or tablet hand cursor is translated into a series of input coordinate values. The graphics tablet is one of the more common stroke devices. Button activation can be used to place the tablet into "continuous" mode. As the cursor is moved across the tablet surface, a stream of coordinate values is generated. This process is used in paintbrush systems that allow artists to draw scenes on the screen and in engineering systems where layouts can be traced and digitized for storage.

String Devices

The primary physical device used for string input is the keyboard. Input character strings are typically used for picture or graph labels.

Other physical devices can be used for generating character patterns in a "text-writing" mode. For this input, individual characters are drawn on the screen with a stroke or locator-type device. A pattern-recognition program then interprets the characters using a stored dictionary of predefined patterns.

Valuator Devices

This logical class of devices is employed in graphics systems to input scalar values. Valuators are used for setting various graphics parameters, such as rotation

angle and scale factors, and for setting physical parameters associated with a particular application (temperature settings, voltage levels, stress factors, etc.).

A typical physical device used to provide valuator input is a set of control dials. Floating-point numbers within any predefined range are input by rotating the dials. Dial rotations in one direction increase the numeric input value, and opposite rotations decrease the numeric value. Rotary potentiometers convert dial rotation into a corresponding voltage. This voltage is then translated into a real number within a defined scalar range, such as −10.5 to 25.5. Instead of dials, slide potentiometers are sometimes used to convert linear movements into scalar values.

Any keyboard with a set of numeric keys can be used as a valuator device. A user simply types the numbers directly in floating-point format, although this is a slower method than using dials or slide potentiometers.

Joysticks, trackballs, tablets, and other interactive devices can be adapted for valuator input by interpreting pressure or movement of the device relative to a scalar range. For one direction of movement, say, left to right, increasing scalar values can be input. Movement in the opposite direction decreases the scalar input value.

Another technique for providing valuator input is to display sliders, buttons, rotating scales, and menus on the video monitor. Figure 8-2 illustrates some possibilities for scale representations. Locator input from a mouse, joystick, spaceball, or other device is used to select a coordinate position on the display, and the screen coordinate position is then converted to a numeric input value. As a feedback mechanism for the user, the selected position on a scale can be marked with some symbol. Numeric values may also be echoed somewhere on the screen to confirm the selections.

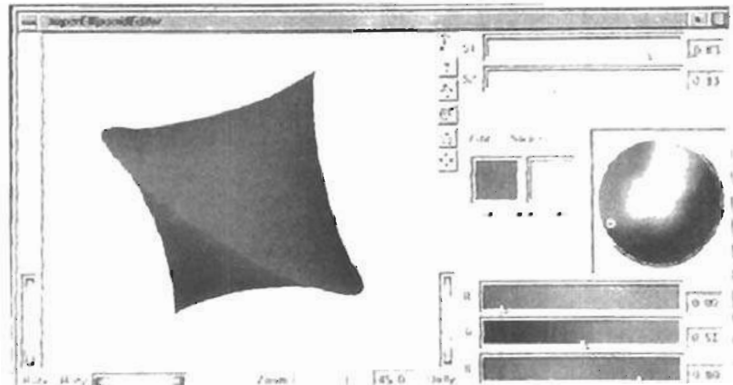


Figure 8-2
Scales displayed on a video monitor for interactive selection of parameter values. In this display, sliders are provided for selecting scalar values for superellipse parameters, s1 and s2, and for individual R, G, and B color values. In addition, a small circle can be positioned on the color wheel for selection of a combined RGB color, and buttons can be activated to make small changes in color values.

Choice Devices

Graphics packages use menus to select programming options, parameter values, and object shapes to be used in constructing a picture (Fig. 8-1). A choice device is defined as one that enters a selection from a list (menu) of alternatives. Commonly used choice devices are a set of buttons; a cursor positioning device, such as a mouse, trackball, or keyboard cursor keys; and a touch panel.

A function keyboard, or "button box", designed as a stand-alone unit, is often used to enter menu selections. Usually, each button is programmable, so that its function can be altered to suit different applications. Single-purpose buttons have fixed, predefined functions. Programmable function keys and fixed-function buttons are often included with other standard keys on a keyboard.

For screen selection of listed menu options, we can use cursor-control devices. When a coordinate position (x, y) is selected, it is compared to the coordinate extents of each listed menu item. A menu item with vertical and horizontal boundaries at the coordinate values x_{min}, x_{max}, y_{min}, and y_{max} is selected if the input coordinates (x, y) satisfy the inequalities

$$x_{min} \le x \le x_{max}, \qquad y_{min} \le y \le y_{max} \tag{8-1}$$

For larger menus with a few options displayed at a time, a touch panel is commonly used. As with a cursor-control device, such as a mouse, a selected screen position is compared to the area occupied by each menu choice.

Alternate methods for choice input include keyboard and voice entry. A standard keyboard can be used to type in commands or menu options. For this method of choice input, some abbreviated format is useful. Menu listings can be numbered or given short identifying names. Similar codings can be used with voice-input systems. Voice input is particularly useful when the number of options is small (20 or less).

Pick Devices

Graphical object selection is the function of this logical class of devices. Pick devices are used to select parts of a scene that are to be transformed or edited in some way.

Typical devices used for object selection are the same as those for menu selection: the cursor-positioning devices. With a mouse or joystick, we can position the cursor over the primitives in a displayed structure and press the selection button. The position of the cursor is then recorded, and several levels of search may be necessary to locate the particular object (if any) that is to be selected. First, the cursor position is compared to the coordinate extents of the various structures in the scene. If the bounding rectangle of a structure contains the cursor coordinates, the picked structure has been identified. But if two or more structure areas contain the cursor coordinates, further checks are necessary. The coordinate extents of individual lines in each structure can be checked next. If the cursor coordinates are determined to be inside the coordinate extents of only one line, for example, we have identified the picked object. Otherwise, we need additional checks to determine the closest line to the cursor position.

One way to find the closest line to the cursor position is to calculate the distance squared from the cursor coordinates (x, y) to each line segment whose bounding rectangle contains the cursor position (Fig. 8-3). For a line with endpoints (x_1, y_1) and (x_2, y_2), distance squared from (x, y) to the line is calculated as

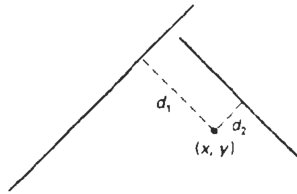


Figure 8-3
Distances to line segments from the pick position.

$$d^2 = \frac{[\Delta x(y - y_1) - \Delta y(x - x_1)]^2}{\Delta x^2 + \Delta y^2} \tag{8-2}$$

where $\Delta x = x_2 - x_1$, and $\Delta y = y_2 - y_1$. Various approximations can be used to speed up this distance calculation, or other identification schemes can be used.

Another method for finding the closest line to the cursor position is to specify the size of a **pick window**. The cursor coordinates are centered on this window and the candidate lines are clipped to the window, as shown in Fig. 8-4. By making the pick window small enough, we can ensure that a single line will cross the window. The method for selecting the size of a pick window is described in Section 8-4, where we consider the parameters associated with various input functions.

A method for avoiding the calculation of pick distances or window clipping intersections is to highlight the candidate structures and allow the user to resolve the pick ambiguity. One way to do this is to highlight the structures that overlap the cursor position one by one. The user then signals when the desired structure is highlighted.

An alternative to cursor positioning is to use button input to highlight successive structures. A second button is used to stop the process when the desired structure is highlighted. If very many structures are to be searched in this way, the process can be speeded up and an additional button is used to help identify the structure. The first button can initiate a rapid successive highlighting of structures. A second button can again be used to stop the process, and a third button can be used to back up more slowly if the desired structure passed before the operator pressed the stop button.

Finally, we could use a keyboard to type in structure names. This is a straightforward, but less interactive, pick-selection method. Descriptive names can be used to help the user in the pick process, but the method has several drawbacks. It is generally slower than interactive picking on the screen, and a user will probably need prompts to remember the various structure names. In addition, picking structure subparts from the keyboard can be more difficult than picking the subparts on the screen.

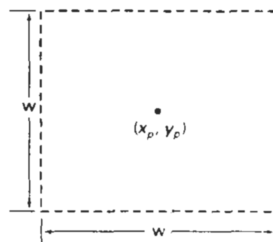


Figure 8-4
A pick window, centered on pick coordinates (x_p, y_p), used to resolve pick object overlaps.

8-3 INPUT FUNCTIONS

Graphical input functions can be set up to allow users to specify the following options:

- Which physical devices are to provide input within a particular logical classification (for example, a tablet used as a stroke device).
- How the graphics program and devices are to interact (input mode). Either the program or the devices can initiate data entry, or both can operate simultaneously.
- When the data are to be input and which device is to be used at that time to deliver a particular input type to the specified data variables.

Input Modes

Functions to provide input can be structured to operate in various **input modes**, which specify how the program and input devices interact. Input could be initiated by the program, or the program and input devices both could be operating simultaneously, or data input could be initiated by the devices. These three input modes are referred to as request mode, sample mode, and event mode.

In **request mode**, the application program initiates data entry. Input values are requested and processing is suspended until the required values are received. This input mode corresponds to typical input operation in a general programming language. The program and the input devices operate alternately. Devices are put into a wait state until an input request is made; then the program waits until the data are delivered.

In **sample mode**, the application program and input devices operate independently. Input devices may be operating at the same time that the program is processing other data. New input values from the input devices are stored, replacing previously input data values. When the program requires new data, it samples the *current* values from the input devices.

In **event mode**, the input devices initiate data input to the application program. The program and the input devices again operate concurrently, but now the input devices deliver data to an input queue. All input data are saved. When the program requires new data, it goes to the data queue.

Any number of devices can be operating at the same time in sample and event modes. Some can be operating in sample mode, while others are operating in event mode. But only one device at a time can be providing input in request mode.

An input mode within a logical class for a particular physical device operating on a specified workstation is declared with one of six input-class functions of the form

```
set ... Mode (ws, deviceCode, inputMode, echoFlag)
```

where `deviceCode` is a positive integer; `inputMode` is assigned one of the values: *request*, *sample*, or *event*; and parameter `echoFlag` is assigned either the value *echo* or the value *noecho*. How input data will be echoed on the display device is determined by parameters set in other input functions to be described later in this section.

TABLE 8-1
ASSIGNMENT OF INPUT-DEVICE CODES

Device Code	*Physical Device Type*
1	Keyboard
2	Graphics Tablet
3	Mouse
4	Joystick
5	Trackball
6	Button

Device code assignment is installation-dependent. One possible assignment of device codes is shown in Table 8-1. Using the assignments in this table, we could make the following declarations:

```
setLocatorMode (1, 2, sample, noecho)
setTextMode (2, 1, request, echo)
setPickMode (4, 3, event, echo)
```

Thus, the graphics tablet is declared to be a locator device in sample mode on workstation 1 with no input data feedback echo; the keyboard is a text device in request mode on workstation 2 with input echo; and the mouse is declared to be a pick device in event mode on workstation 4 with input echo.

Request Mode

Input commands used in this mode correspond to standard input functions in a high-level programming language. When we ask for an input in request mode, other processing is suspended until the input is received. After a device has been assigned to request mode, as discussed in the preceding section, input requests can be made to that device using one of the six logical-class functions represented by the following:

```
request ... (ws, deviceCode, status, ... )
```

Values input with this function are the workstation code and the device code. Returned values are assigned to parameter `status` and to the data parameters corresponding to the requested logical class.

A value of *ok* or *none* is returned in parameter `status`, according to the validity of the input data. A value of *none* indicates that the input device was activated so as to produce invalid data. For locator input, this could mean that the coordinates were out of range. For pick input, the device could have been activated while not pointing at a structure. Or a "break" button on the input device could have been pressed. A returned value of *none* can be used as an end-of-data signal to terminate a programming sequence.

Locator and Stroke Input in Request Mode

The request functions for these two logical input classes are

```
requestLocator (ws, devCode, status, viewIndex, pt)
requestStroke (ws, devCode, nMax, status, viewIndex, n, pts)
```

For locator input, pt is the world-coordinate position selected. For stroke input, pts is a list of n coordinate positions, where parameter nMax gives the maximum number of points that can go in the input list. Parameter viewIndex is assigned the two-dimensional view index number.

Determination of a world-coordinate position is a two-step process: (1) The physical device selects a point in device coordinates (usually from the video-display screen) and the inverse of the workstation transformation is performed to obtain the corresponding point in normalized device coordinates. (2) Then, the inverse of the window-to-viewport mapping is carried out to get to viewing coordinates, then to world coordinates.

Since two or more views may overlap on a device, the correct viewing transformation is identified according to the **view-transformation input priority** number. By default, this is the same as the view index number, and the lower the number, the higher the priority. View index 0 has the highest priority. We can change the view priority relative to another (reference) viewing transformation with

```
setViewTransformationInputPriority (ws, viewIndex,
                    refViewIndex, priority)
```

where viewIndex identifies the viewing transformation whose priority is to be changed, refViewIndex identifies the reference viewing transformation, and parameter priority is assigned either the value *lower* or the value *higher*. For example, we can alter the priority of the first four viewing transformations on workstation 1, as shown in Fig. 8-5, with the sequence of functions:

```
setViewTransformationInputPriority (1, 3, 1, higher)
setViewTransformationInputPriority (1, 0, 2, lower)
```

String Input in Request Mode

Here, the request input function is

```
requestString (ws, devCode, status, nChars, str)
```

Parameter str in this function is assigned an input string. The number of characters in the string is given in parameter nChars.

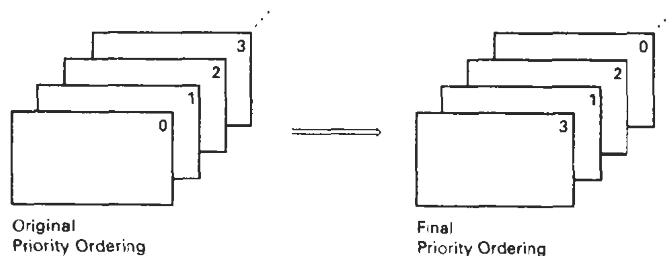


Figure 8-5
Rearranging viewing priorities.

Valuator Input in Request Mode

A numerical value is input in request mode with

```
requestValuator (ws, devCode, status, value)
```

Parameter `value` cai be assigned any real-number value.

Choice Input in Request Mode

We make a menu selection with the following request function:

```
requestChoice (ws, devCode, status, itemNum)
```

Parameter `itemNum` is assigned a positive integer value corresponding to the menu item selected.

Pick Input in Request Mode

For this mode, we obtain a structure identifier number with the function

```
requestPick (ws, devCode, maxPathDepth, status, pathDepth,
                    pickPath)
```

Parameter `pickPath` is a list of information identifying the primitive selected. This list contains the structure name, pick identifier for the primitive, and the element sequence number. Parameter `pickDepth` is the number of levels returned in `pickPath`, and `maxPathDepth` is the specified maximum path depth that can be included in `pickPath`.

Subparts of a structure can be labeled for pick input with the following function:

```
setPickIdentifier (pickID)
```

An example of sublabeling during structure creation is given in the following programming sequence:

```
openStructure (id);
     for (k = 0; k < n; k++){
                setPickIdentifier (k);
                          .
                          .
                          .
     }
closeStructure;
```

Picking of structures and subparts of structures is also controlled by some workstation filters (Section 7-1) Objects cannot be picked if they are invisible. Also, we can set the ability to pick objects independently of their visibility. This is accomplished with the pick filter:

```
setPickFilter (ws, devCode, pickables, nonpickables)
```

where the set `pickables` contains the names of objects (structures and primitives) that we may want to select with the specified pick device. Similarly, the set `nonpickables` contains the names of objects that we do not want to be available for picking with this input device.

Sample Mode

Once sample mode has been set for one or more physical devices, data input begins without waiting for program direction. If a joystick has been designated as a locator device in sample mode, coordinate values for the current position of the activated joystick are immediately stored. As the activated stick position changes, the stored values are continually replaced with the coordinates of the current stick position.

Sampling of the current values from a physical device in this mode begins when a sample command is encountered in the application program. A locator device is sampled with one of the six logical-class functions represented by the following:

```
sample ... (ws, deviceCode, ... )
```

Some device classes have a status parameter in sample mode, and some do not. Other input parameters are the same as in request mode.

As an example of sample input, suppose we want to translate and rotate a selected object. A final translation position for the object can be obtained with a locator, and the rotation angle can be supplied by a valuator device, as demonstrated in the following statements.

```
sampleLocator (ws1, dev1, viewIndex, pt)
sampleValuator (ws2, dev2, angle)
```

Event Mode

When an input device is placed in event mode, the program and device operate simultaneously. Data input from the device is accumulated in an event queue, or input queue. All input devices active in event mode can enter data (referred to as "events") into this single-event queue, with each device entering data values as they are generated. At any one time, the event queue can contain a mixture of data types, in the order they were input. Data entered into the queue are identified according to logical class, workstation number, and physical-device code.

An application program can be directed to check the event queue for any input with the function

```
awaitEvent (time, ws, deviceClass, deviceCode)
```

Parameter `time` is used to set a maximum waiting time for the application program. If the queue happens to be empty, processing is suspended until either the number of seconds specified in time has elapsed or an input arrives. Should the waiting time run out before data values are input, the parameter `deviceClass` is assigned the value *none*. When `time` is given the value 0, the program checks the queue and immediately returns to other processing if the queue is empty.

If processing is directed to the event queue with the `awaitEvent` function and the queue is not empty, the first event in the queue is transferred to a *current event record.* The particular logical device class, such as locator or stroke, that made this input is stored in parameter `deviceClass`. Codes, identifying the particular workstation and physical device that made the input, are stored in parameters ws and `deviceCode`, respectively.

To retrieve a data input from the current event record, an event-mode input function is used. The functions in event mode are similar to those in request and sample modes. However, no workstation and device-code parameters are necessary in the commands, since the values for these parameters are stored in the data record. A user retrieves data with

```
get ... ( ... )
```

For example, to ask for locator input, we invoke the function

```
getLocator (viewIndex, pt)
```

In the following program section, we give an example of the use of the `awaitEvent` and `get` functions. A set of points from a tablet (device code 2) on workstation 1 is input to plot a series of straight-line segments connecting the input coordinates:

```
setStrokeMode (1, 2, event, noecho);

do {
  awaitEvent (0, ws, deviceClass, deviceCode)
} while (deviceClass != stroke);
getStroke (nMax, viewIndex, n, pts);
polyline (n, pts);
```

The `repeat-until` loop bypasses any data from other devices that might be in the queue. If the tablet is the only active input device in event mode, this loop is not necessary.

A number of devices can be used at the same time in event mode for rapid interactive processing of displays. The following statements plot input lines from a tablet with attributes specified by a button box:

```
setPolylineIndex (1);
/* set tablet to stroke device, event mode */
setStrokeMode (1, 2, event, noecho);

/* set buttons to choice device, event mode */
setChoiceMode (1, 6, event, noecho);

do {
  awaitEvent (60, ws, deviceClass, deviceCode);
  if (deviceClass == choice) {
    getChoice (status, option);
    setPolylineIndex (option);
  }
  else
    if (deviceClass == stroke) {
      getStroke (nMax, viewIndex, n, pts);
      polyline (n, pts);
    }
} while (deviceClass != none);
```

Some additional housekeeping functions can be used in event mode. Functions for clearing the event queue are useful when a process is terminated and a new application is to begin. These functions can be set to clear the entire queue or to clear only data associated with specified input devices and workstations.

Concurrent Use of Input Modes

An example of the simultaneous use of input devices in different modes is given in the following procedure. An object is dragged around the screen with a mouse. When a final position has been selected, a button is pressed to terminate any further movement of the object. The mouse positions are obtained in sample mode, and the button input is sent to the event queue

```
/* drags object in response to mouse input */
/* terminate processing by button press */
setLocatorMode (1, 3, sample, echo);
setChoiceMode (1, 6, event, noecho);
do {
  sampleLocator (1, 3, viewIndex, pt);

  /* translate object centroid to position pt and draw */

  awaitEvent (0, ws, class, code);
} while (class != choice);
```

8-4 INITIAL VALUES FOR INPUT-DEVICE PARAMETERS

Quite a number of parameters can be set for input devices using the `initialize` function for each logical class:

```
initialize ... (ws, deviceCode, ... , pe, coordExt, dataRec)
```

Parameter pe is the prompt and echo type, parameter `coordExt` is assigned a set of four coordinate values, and parameter `dataRec` is a record of various control parameters.

For locator input, some values that can be assigned to the prompt and echo parameter are

pe = 1: installation defined
pe = 2: crosshair cursor centered at current position
pe = 3: line from initial position to current position
pe = 4: rectangle defined by current and initial points

Several other options are also available.

For structure picking, we have the following options:

pe = 1: highlight picked primitives
pe = 2: highlight all primitives with value of pick id
pe = 3: highlight entire structure

as well as several others.

When an echo of the input data is requested, it is displayed within the bounding rectangle specified by the four coordinates in parameter `coordExt`. Additional options can also be set in parameter `dataRec`. For example, we can set any of the following:

- size of the pick window
- minimum pick distance
- type and size of cursor display
- type of structure highlighting during pick operations
- range (min and max) for valuator input
- resolution (scale) for valuator input

plus a number of other options.

8-5 INTERACTIVE PICTURE-CONSTRUCTION TECHNIQUES

There are several techniques that are incorporated into graphics packages to aid the interactive construction of pictures. Various input options can be provided, so that coordinate information entered with locator and stroke devices can be adjusted or interpreted according to a selected option. For example, we can restrict all lines to be either horizontal or vertical. Input coordinates can establish the position or boundaries for objects to be drawn, or they can be used to rearrange previously displayed objects.

Basic Positioning Methods

Coordinate values supplied by locator input are often used with positioning methods to specify a location for displaying an object or a character string. We interactively select coordinate positions with a pointing device, usually by positioning the screen cursor. Just how the object or text-string positioning is peformed depends on the selected options. With a text string, for example, the screen point could be taken as the center string position, or the start or end position of the string, or any of the other string-positioning options discussed in Chapter 4. For lines, straight line segments can be displayed between two selected screen positions.

As an aid in positioning objects, numeric values for selected positions can be echoed on the screen. Using the echoed coordinate values as a guide, we can make adjustments in the selected location to obtain accurate positioning.

Constraints

With some applications, certain types of prescribed orientations or object alignments are useful. A constraint is a rule for altering input-coordinate values to produce a specified orientation or alignment of the displayed coordinates. There are many kinds of constraint functions that can be specified, but the most common constraint is a horizontal or vertical alignment of straight lines. This type of constraint, shown in Figs. 8-6 and 8-7, is useful in forming network layouts. With this constraint, we can create horizontal and vertical lines without worrying about precise specification of endpoint coordinates.

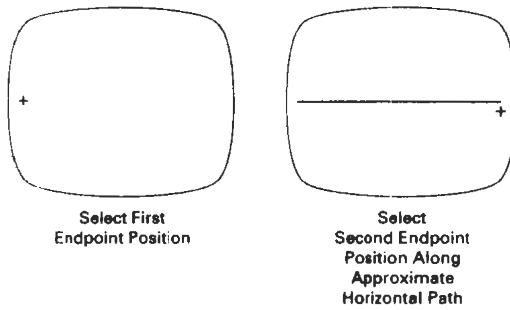


Figure 8-6
Horizontal line constraint.

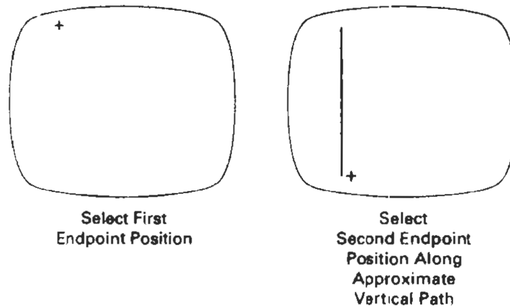


Figure 8-7
Vertical line constraint.

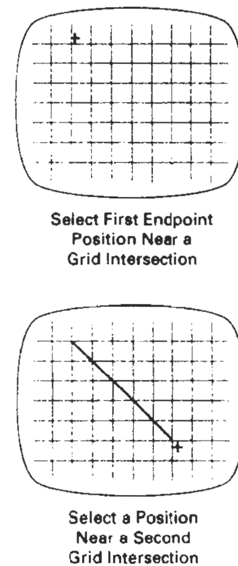


Figure 8-8
Line drawing using a grid.

A horizontal or vertical constraint is implemented by determining whether any two input coordinate endpoints are more nearly horizontal or more nearly vertical. If the difference in the y values of the two endpoints is smaller than the difference in x values, a horizontal line is displayed. Otherwise, a vertical line is drawn. Other kinds of constraints can be applied to input coordinates to produce a variety of alignments. Lines could be constrained to have a particular slant, such as 45°, and input coordinates could be constrained to lie along predefined paths, such as circular arcs.

Grids

Another kind of constraint is a grid of rectangular lines displayed in some part of the screen area. When a grid is used, any input coordinate position is rounded to the nearest intersecton of two grid lines. Figure 8-8 illustrates line drawing with a grid. Each of the two cursor positions is shifted to the nearest grid intersection point, and the line is drawn between these grid points. Grids facilitate object constructions, because a new line can be joined easily to a previously drawn line by selecting any position near the endpoint grid intersection of one end of the displayed line.

Spacing between grid lines is often an option that can be set by the user. Similarly, grids can be turned on and off, and it is sometimes possible to use partial grids and grids of different sizes in different screen areas.

Gravity Field

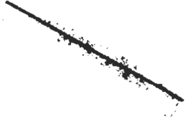


Figure 8-9
Gravity field around a line. Any selected point in the shaded area is shifted to a position on the line.

In the construction of figures, we sometimes need to connect lines at positions between endpoints. Since exact positioning of the screen cursor at the connecting point can be difficult, graphics packages can be designed to convert any input position near a line to a position on the line.

This conversion of input position is accomplished by creating a *gravity field* area around the line. Any selected position within the gravity field of a line is moved ("gravitated") to the nearest position on the line. A gravity field area around a line is illustrated with the shaded boundary shown in Fig. 8-9. Areas around the endpoints are enlarged to make it easier for us to connect lines at their endpoints. Selected positions in one of the circular areas of the gravity field are attracted to the endpoint in that area. The size of gravity fields is chosen large enough to aid positioning, but small enough to reduce chances of overlap with other lines. If many lines are displayed, gravity areas can overlap, and it may be difficult to specify points correctly. Normally, the boundary for the gravity field is not displayed.

Rubber-Band Methods

Straight lines can be constructed and positioned using *rubber-band* methods, which stretch out a line from a starting position as the screen cursor is moved. Figure 8-10 demonstrates the rubber-band method. We first select a screen position for one endpoint of the line. Then, as the cursor moves around, the line is displayed from the start position to the current position of the cursor. When we finally select a second screen position, the other line endpoint is set.

Rubber-band methods are used to construct and position other objects besides straight lines. Figure 8-11 demonstrates rubber-band construction of a rectangle, and Fig. 8-12 shows a rubber-band circle construction.

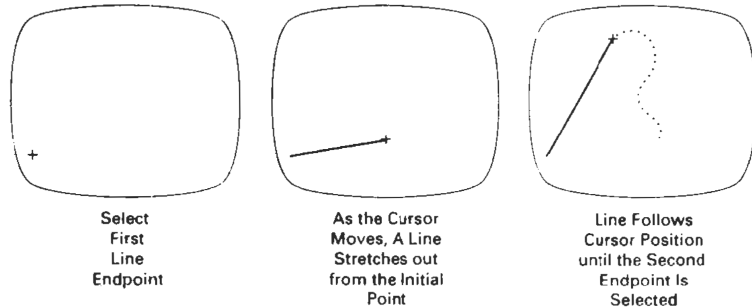


Figure 8-10
Rubber-band method for drawing and positioning a straight line segment.

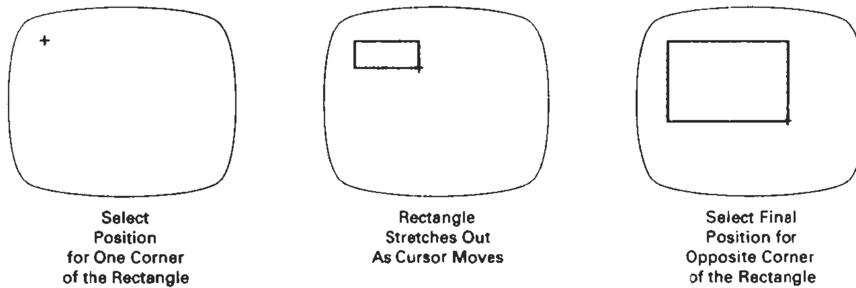


Figure 8-11
Rubber-band method for constructing a rectangle.

Dragging

A technique that is often used in interactive picture construction is to move objects into position by dragging them with the screen cursor. We first select an object, then move the cursor in the direction we want the object to move, and the selected object follows the cursor path. Dragging objects to various positions in a scene is useful in applications where we might want to explore different possibilities before selecting a final location.

Painting and Drawing

Options for sketching, drawing, and painting come in a variety of forms. Straight lines, polygons, and circles can be generated with methods discussed in the previous sections. Curve-drawing options can be provided using standard curve shapes, such as circular arcs and splines, or with freehand sketching procedures. Splines are interactively constructed by specifying a set of discrete screen points that give the general shape of the curve. Then the system fits the set of points with a polynomial curve. In freehand drawing, curves are generated by following the path of a stylus on a graphics tablet or the path of the screen cursor on a video monitor. Once a curve is displayed, the designer can alter the curve shape by adjusting the positions of selected points along the curve path.

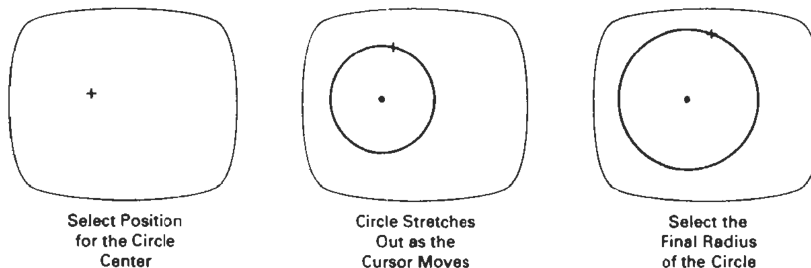


Figure 8-12
Constructing a circle using a rubber-band method.

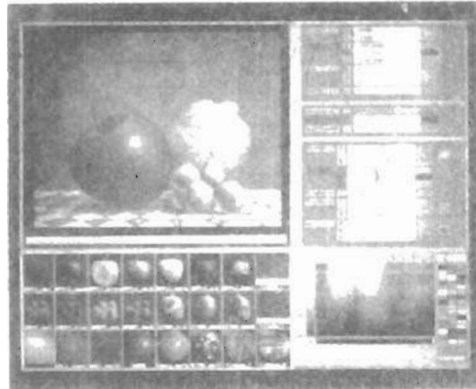


Figure 8-13
A screen layout showing one type of interface to an artist's painting package. (*Courtesy of Thomson Digital Image.*)

Line widths, line styles, and other attribute options are also commonly found in painting and drawing packages. These options are implemented with the methods discussed in Chapter 4. Various brush styles, brush patterns, color combinations, object shapes, and surface-texture patterns are also available on many systems, particularly those designed as artist's workstations. Some paint systems vary the line width and brush strokes according to the pressure of the artist's hand on the stylus. Figure 8-13 shows a window and menu system used with a painting package that allows an artist to select variations of a specified object shape, different surface textures, and a variety of lighting conditions for a scene.

8-6 VIRTUAL-REALITY ENVIRONMENTS

A typical virtual-reality environment is illustrated in Fig. 8-14. Interactive input is accomplished in this environment with a data glove (Section 2-5), which is capable of grasping and moving objects displayed in a virtual scene. The computer-generated scene is displayed through a head-mounted viewing system (Section 2-1) as a stereoscopic projection. Tracking devices compute the position and orientation of the headset and data glove relative to the object positions in the scene. With this system, a user can move through the scene and rearrange object positions with the data glove.

Another method for generating virtual scenes is to display stereoscopic projections on a raster monitor, with the two stereoscopic views displayed on alternate refresh cycles. The scene is then viewed through stereoscopic glasses. Interactive object manipulations can again be accomplished with a data glove and a tracking device to monitor the glove position and orientation relative to the position of objects in the scene.

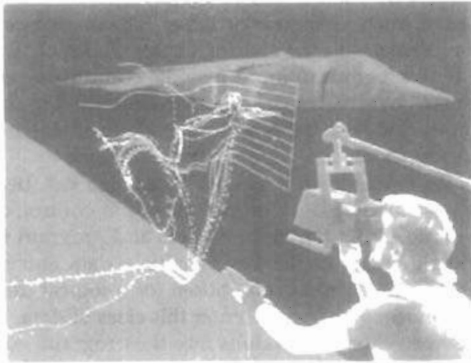


Figure 8-14
Using a head-tracking stereo display, called the BOOM (Fake Space Labs, Inc.), and a Dataglove (VPL, Inc.), a researcher interactively manipulates exploratory probes in the unsteady flow around a Harrier jet airplane. Software developed by Steve Bryson; data from Harrier. *(Courtesy of Sam Uselton, NASA Ames Research Center.)*

SUMMARY

A dialogue for an applications package can be designed from the user's model, which describes the functions of the applications package. All elements of the dialogue are presented in the language of the applications. Examples are electrical and architectural design packages.

Graphical interfaces are typically designed using windows and icons. A window system provides a window-manager interface with menus and icons that allows users to open, close, reposition, and resize windows. The window system then contains routines to carry out these operations, as well as the various graphics operations. General window systems are designed to support multiple window managers. Icons are graphical symbols that are designed for quick identification of application processes or control processes.

Considerations in user-dialogue design are ease of use, clarity, and flexibility. Specifically, graphical interfaces are designed to maintain consistency in user interaction and to provide for different user skill levels. In addition, interfaces are designed to minimize user memorization, to provide sufficient feedback, and to provide adequate backup and error-handling capabilities.

Input to graphics programs can come from many different hardware devices, with more than one device providing the same general class of input data. Graphics input functions can be designed to be independent of the particular input hardware in use, by adopting a logical classification for input devices. That is, devices are classified according to the type of graphics input, rather than a

hardware designation, such as mouse or tablet. The six logical devices in common use are locator, stroke, string, valuator, choice, and pick. Locator devices are any devices used by a program to input a single coordinate position. Stroke devices input a stream of coordinates. String devices are used to input text. Valuator devices are any input devices used to enter a scalar value. Choice devices enter menu selections. And pick devices input a structure name.

Input functions available in a graphics package can be defined in three input modes. Request mode places input under the control of the application program. Sample mode allows the input devices and program to operate concurrently. Event mode allows input devices to initiate data entry and control processing of data. Once a mode has been chosen for a logical device class and the particular physical device to be used to enter this class of data, input functions in the program are used to enter data values into the program. An application program can make simultaneous use of several physical input devices operating in different modes.

Interactive picture-construction methods are commonly used in a variety of applications, including design and painting packages. These methods provide users with the capability to position objects, to constrain figures to predefined orientations or alignments, to sketch figures, and to drag objects around the screen. Grids, gravity fields, and rubber-band methods are used to aid in positioning and other picture-construction operations.

REFERENCES

Guidelines for user-interface design are presented in Apple (1987), Bleser (1988), Digital (1989), and OSF/MOTIF (1989). For information on the X Window System, see Young (1990) and Cutler, Gilly, and Reilly (1992). Additional discussions of interface design can be found in Phillips (1977), Goodman and Spence (1978), Lodding (1983), Swezey and Davis (1983), Carroll and Carrithers (1984), Foley, Wallace, and Chan (1984), and Good et al. (1984).

The evolution of the concept of logical (or virtual) input devices is discussed in Wallace (1976) and in Rosenthal et al. (1982). An early discussion of input-device classifications is to be found in Newman (1968).

Input operations in PHIGS can be found in Hopgood and Duce (1991), Howard et al. (1991), Gaskins (1992), and Blake (1993). For information on GKS input functions, see Hopgood et al. (1983) and Enderle, Kansy, and Pfaff (1984).

EXERCISES

8-1. Select some graphics application with which you are familiar and set up a user model that will serve as the basis for the design of a user interface for graphics applications in that area.

8-2. List possible help facilities that can be provided in a user interface and discuss which types of help would be appropriate for different levels of users.

8-3. Summarize the possible ways of handling backup and errors. State which approaches are more suitable for the beginner and which are better suited to the experienced user.

8-4. List the possible formats for presenting menus to a user and explain under what circumstances each might be appropriate.

8-5. Discuss alternatives for feedback in terms of the various levels of users.

8-6. List the functions that must be performed by a window manager in handling screen layouts with multiple overlapping windows.

8-7. Set up a design for a window-manager package.

8-8. Design a user interface for a painting program.

8-9. Design a user interface for a two-level hierarchical modeling package.

8-10. For any area with which you are familiar, design a complete user interface to a graphics package providing capabilities to any users in that area.

8-11. Develop a program that allows objects to be positioned on the screen using a locator device. An object menu of geometric shapes is to be presented to a user who is to select an object and a placement position. The program should allow any number of objects to be positioned until a "terminate" signal is given.

8-12. Extend the program of the previous exercise so that selected objects can be scaled and rotated before positioning. The transformation choices and transformation parameters are to be presented to the user as menu options.

8-13. Write a program that allows a user to interactively sketch pictures using a stroke device.

8-14. Discuss the methods that could be employed in a pattern-recognition procedure to match input characters against a stored library of shapes.

8-15. Write a routine that displays a linear scale and a slider on the screen and allows numeric values to be selected by positioning the slider along the scale line. The number value selected is to be echoed in a box displayed near the linear scale.

8-16. Write a routine that displays a circular scale and a pointer or a slider that can be moved around the circle to select angles (in degrees). The angular value selected is to be echoed in a box displayed near the circular scale.

8-17. Write a drawing program that allows users to create a picture as a set of line segments drawn between specified endpoints. The coordinates of the individual line segments are to be selected with a locator device.

8-18. Write a drawing package that allows pictures to be created with straight line segments drawn between specified endpoints. Set up a gravity field around each line in a picture, as an aid in connecting new lines to existing lines.

8-19. Modify the drawing package in the previous exercise that allows lines to be constrained horizontally or vertically.

8-20. Develop a drawing package that can display an optional grid pattern so that selected screen positions are rounded to grid intersections. The package is to provide line-drawing capabilities, with line endpoints selected with a locator device.

8-21. Write a routine that allows a designer to create a picture by sketching straight lines with a rubber-band method.

8-22. Write a drawing package that allows straight lines, rectangles, and circles to be constructed with rubber-band methods.

8-23. Write a program that allows a user to design a picture from a menu of basic shapes by dragging each selected shape into position with a pick device.

8-24. Design an implementation of the input functions for request mode.

8-25. Design an implementation of the sample-mode input functions.

8-26. Design an implementation of the input functions for event mode.

8-27. Set up a general implementation of the input functions for request, sample, and event modes.