## Fill – Area Primitives:

A picture component which as an area that is filled with some solid color or pattern is referred to as **fill area** or a **filled area.**

Fill areas are used to describe surfaces of solid objects, but, they are also useful in a variety of other applications. Also, fill regions are usually planar surfaces, mainly polygons. But, in general, there are many possible shapes for a region in a picture that we might wish to fill with some color option.

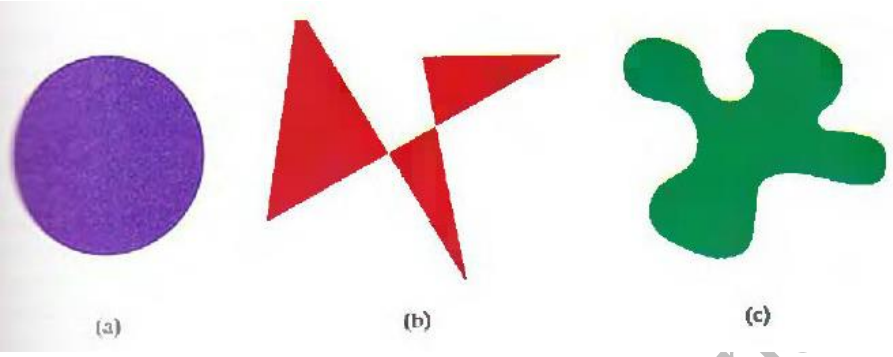



FIGURE 3-40 Solid-color fill areas specified with various boundaries. (a) A circular fill region. (b) A fill area bounded by a closed polyline. (c) A filled area specified with an irregular curved boundary.

Most library routines require that a fill area be specified as a polygon. Graphics routines can more efficiently process polygons than other kinds of fill shapes because polygon boundaries are described with linear equations.

Most curved surfaces can be approximated reasonably well with a set of polygon patches, just as a curved line can be approximated with a set of straight-line segments.

Objects described with a set of polygon surface patches are usually referred to as **standard graphics objects** , or just **graphics objects.**

FIGURE 3-41 Wire-frame representation for a cylinder, showing only the front (visible) faces of the polygon mesh used to approximate the surfaces.

# Polygon Fill Areas:

A **polygon** is a plane figure specified by a set of three or more coordinate positions, called **vertices**, that are connected in sequence by straight line segments, called the **edges** or **sides** of the polygon.

By definition, a polygon must have all its vertices within a single plane and there can be no edge crossings. Examples of polygons include triangles, rectangles, octagons and decagons.

A polygon with no crossing edges is referred to as a **standard polygon** or a **simple polygon.**

## Polygon Classifications:

An **interior angle** of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges. If all interior angles of a polygon are less than or equal to 180°, then the polygon is **convex.**

An equivalent definition of a convex polygon is that its interior lies completely on one side of the infinite extension line of any one of its edges. Also, if we select any two points in the interior of a convex polygon, the line segment joining the two points is also in the interior.

A polygon that is not convex is called a **concave** polygon.

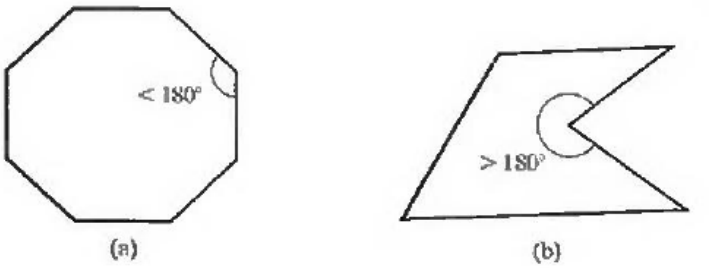



FIGURE 3-42 A convex polygon (a), and a concave polygon (b).

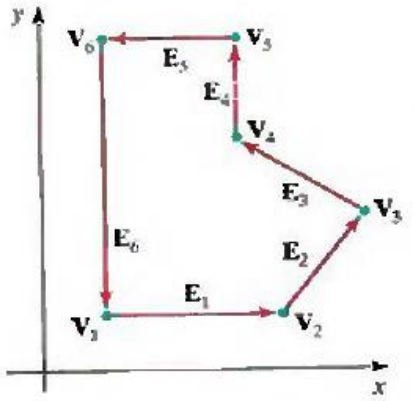
The term **degenerate polygon** is often used to describe a set of vertices that are collinear or that have repeated coordinate positions. Collinear vertices generate a line segment.

Implementations of fill algorithms and other graphics routines are more complicated for concave polygons, so it is generally more efficient to split a concave polygon into a set of convex polygons before processing.

## Identifying Concave Polygons:

A concave polygon has at least one interior angle greater than 180°. Also, the extension of some edges of a concave polygon will intersect other edges, and some pair of interior points will produce a line segment that intersects the polygon boundary. Therefore, we can use any one of these characteristics of a concave polygon as a basis for constructing an identification algorithm.

If we set up a vector for each polygon edge, then we can use the cross product of adjacent edges to test for concavity. All such vector products will be of the same sign (positive or negative) for a convex polygon. Therefore, if some cross product yield a positive value and some a negative value, we have a concave polygon.




$(E_1 \times E_2)_z > 0$

$(E_2 \times E_3)_z > 0$

$(E_3 \times E_4)_z < 0$

$(E_4 \times E_5)_z > 0$

$(E_5 \times E_6)_z > 0$

$(E_6 \times E_1)_z > 0$

FIGURE 3-43 Identifying a concave polygon by calculating cross products of successive pairs of edge vectors.

Another way to identify a concave polygon is to take a look at the polygon vertex positions relative to the extension line of any edge. If some vertices are on one side of the extension line and some vertices are on the other side, the polygon is concave.

## Splitting Concave Polygons:

Once we have identified a concave polygon, we can split it into a set of convex polygons. This can be accomplished using edge vectors and edge cross products. Or, we can use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other.

### Vector Method:

Let us assume that all polygons are in the ***xy*** plane.

In vector method for splitting a concave polygon, we first need to form the edge vectors. Given two consecutive vertex positions, $\mathbf{V_k}$ and $\mathbf{V_{k+1}}$, we define the edge vector between them as

$$\mathbf{E_k = V_{k+1} - V_k}$$

Next we calculate the cross products of successive edge vectors in order around the polygon perimeter.

***If the z component of some cross products is positive while other cross products have negative z component, the polygon is concave.***

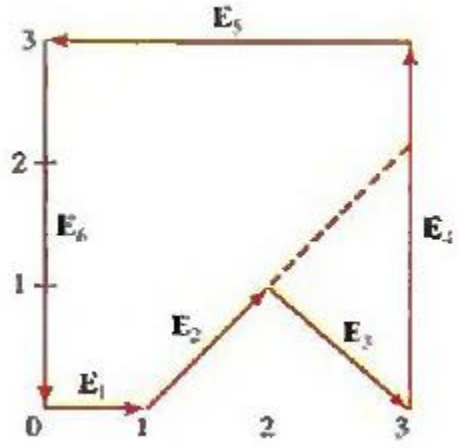
We assume the no series of three successive vertices are collinear, in which case the cross product of the two edge vectors for these vertices would be zero. If all vertices are collinear, we have a degenerate polygon (a straight line).

We can apply the vector method by processing edge vectors in a **counterclockwise order.**

If the polygon is found be concave , then we can split it along the line of the first edge vector in the cross-product pair.

**Example:**

Consider the following figure that shows a concave polygon with six edges.




Edge vectors for this polygon can be expressed as

**E1** = (1, 0, 0) **E2** = (1, 1, 0) **E3** = (1, -1, 0)
**E4** = (0, 3, 0) **E5** = (-3, 0, 0) **E6** = (0, -3, 0)

Where the z component is 0, since all edges are in the ***xy*** plane.

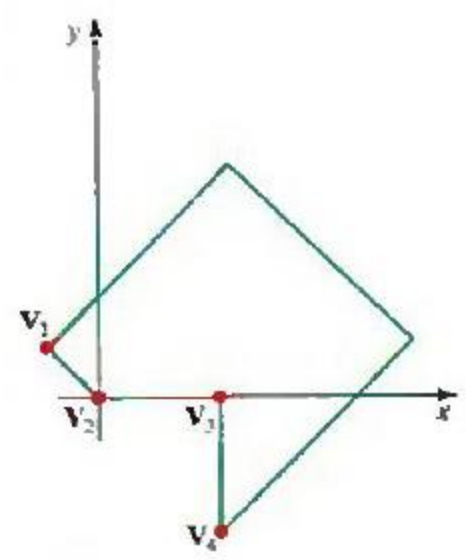
The cross product $\mathbf{E_j \times E_k}$ for two successive edge vectors is a vector perpendicular to the ***xy*** plane with z component equal to $\mathbf{E_{jx}E_{ky} - E_{kx}E_{jy}}$ :

$\mathbf{E_1 \times E_2} = (0,0,1)$ $\mathbf{E_2 \times E_3} = (0,0,-2)$ $\mathbf{E_3 \times E_4} = (0,0,3)$
$\mathbf{E_4 \times E_5} = (0,0,6)$ $\mathbf{E_5 \times E_6} = (0,0,6)$ $\mathbf{E_6 \times E_1} = (0,0,2)$

Since the cross product $\mathbf{E_2 \times E_3}$ has a negative z component, we split the polygon along the line of vector $\mathbf{E_2}$. We then determine the intersection of this line with the other polygon edges to split the polygon into two pieces. No other edge cross products are negative, so the two polygons are both convex.

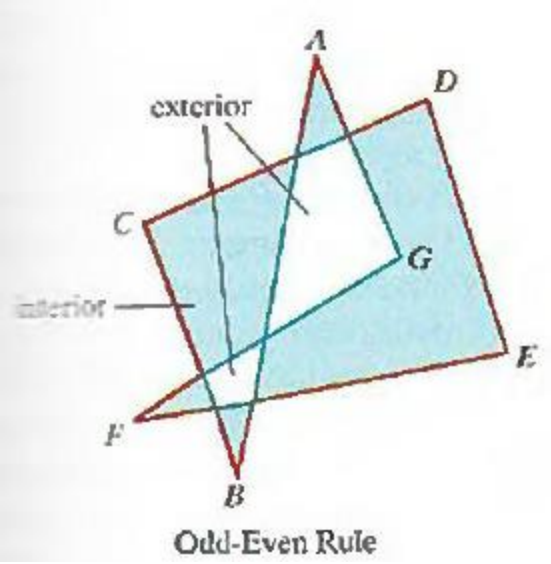
**Rotational Method:**

- Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex Vk in turn is at the coordinate origin.
- Then, we rotate the polygon about the origin in a clockwise direction so that the next vertex Vk+1 is on the x axis.
- If the following vertex, Vk+2, is below the x axis, the polygon is concave. We then split the polygon along the x axis to form two new polygons, and we repeat the concave test for each of the two new polygons.
- The above mentioned steps are repeated until we have tested all vertices in the polygon list.




**Splitting a Convex Polygon into a Set of Triangles:**

- Once we have a vertex list for a convex polygon, we could transform it into a set of triangles. This can be accomplished by first defining any sequence of three consecutive vertices to be a new polygon (a triangle).
- The middle triangle vertex is then deleted from the original vertex list.
- Then the same procedure is applied to this modified vertex list to strip off another triangle.
- We continue forming triangles in this manner until the original polygon is reduced to just three vertices, which define the last triangle in the set.
- A concave polygon can also be divided into a set of triangles using this approach, as long as the three selected vertices at each step form an interior angle that is less than 180°.(a convex angle)



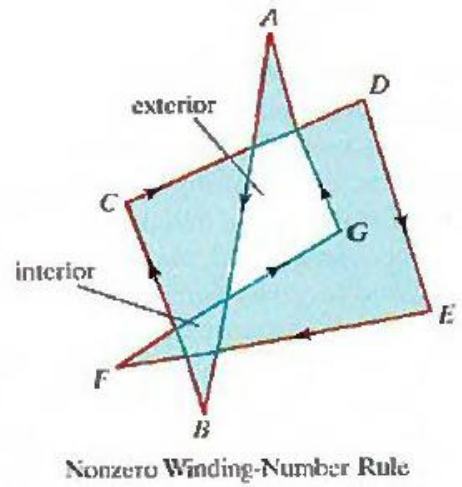
Odd-Even Rule

## Inside-Outside Tests:

### Odd-even rule:

- We apply the odd-even rule, also called the odd parity rule or the evenodd rule, by first conceptually drawing a line from any position **P** to a distant point outside the coordinate extents of the closed polyline.
- Then we count the number of edge crossings along the line.
- If the number of polygon edges crossed by this line is odd, **then P is an interior point**. **Otherwise, P is an exterior point**.
- To obtain an accurate edge count, we must be sure that the line path we choose does not intersect any polygon vertices.
- The above shows the interior and exterior regions obtained from the odd-even rule for a self-intersecting set of edges.
- We can use this procedure to fill the interior between two concentric circles or two concentric polygons with a specified color.

### Nonzero Winding Number Rule:



Nonzero Winding-Number Rule

- This is another method, which counts the number of times the polygon edges wind around a particular point in the counterclockwise direction. This count is called the **winding number.**
- We apply the nonzero winding number rule to polygons by initializing the winding number to 0 and again imagining a line drawn from any position P to a distant point beyond the coordinate extents of the object.
- The line we choose must not pass through any vertices. As we move along the line from position P to the distant point, we count the number of edges that cross the line in each direction.
- We add 1 to the winding number every time we intersect a polygon edge that crosses the line from right to left, and we subtract 1 every time we intersect an edge that crosses from left to right.
- The final value of the winding number, after all edge crossings have been counted, determines the relative position of **P**.
- If the winding number is nonzero, P is defined to be an interior point. Otherwise, P is taken to be an exterior point.

- The above figure shows the interior and exterior regions defined by the nonzero winding number rule for a self-intersecting set of edges.
- For standard polygons and other simple shapes, the nonzero winding number rule and the odd-even rule give the same results. But for more complicated shapes, the two methods may yield different interior and exterior regions.

**Variations of non zero winding number rule:**

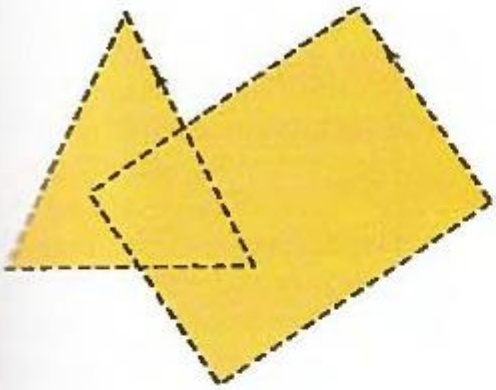


FIGURE 3–47 A fill area defined as a region that has a positive value for the winding number. This fill area is the union of two regions, each with a counterclockwise border direction.

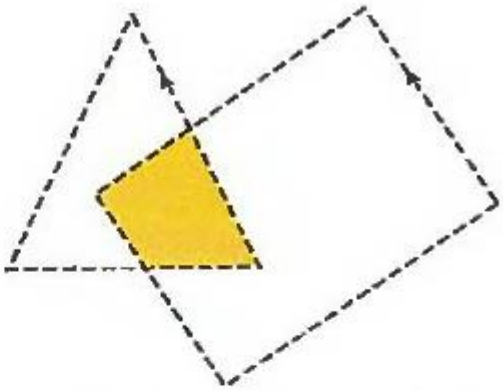


FIGURE 3–48 A fill area defined as a region with a winding number greater than 1. This fill area is the intersection of two regions, each with a counterclockwise border direction.

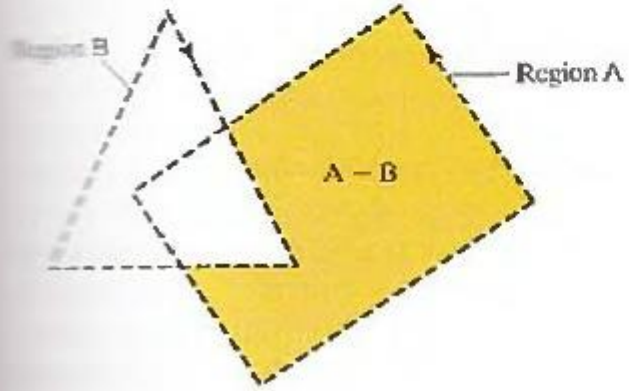



FIGURE 3–49 A fill area defined as a region with a positive value for the winding number. This fill area is the difference, A – B, of two regions, where region A has a positive border direction (counterclockwise) and region B has a negative border direction (clockwise).

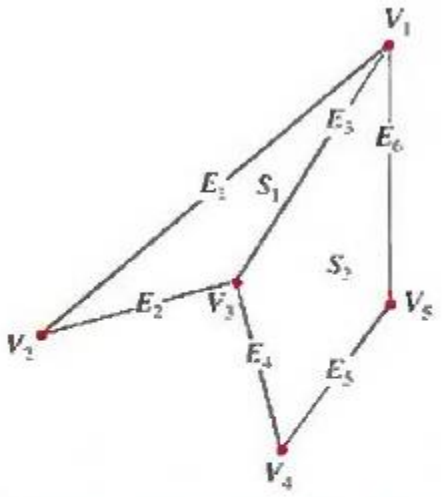
## Polygon Tables:

The objects in a scene are described as sets of polygon surface facets. The graphics packages often provide functions for defining a surface shape as a mesh of polygon patches. The description for each object includes coordinate information specifying the geometry for the polygon facets and other surface parameters such as color, transparency and light-reflection properties.

As information for each polygon is input, the data are placed into tables that are to be used in the subsequent processing, display and manipulation of the objects in the scene. These polygon data tables can be organized into two groups: **geometric tables** and **attribute tables.**

**Geometric data tables** contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces.

**Attribute** information for an object includes parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics.




| VERTEX TABLE | |
|---|---|
| $V_1$: | $x_1, y_1, z_1$ |
| $V_2$: | $x_2, y_2, z_2$ |
| $V_3$: | $x_3, y_3, z_3$ |
| $V_4$: | $x_4, y_4, z_4$ |
| $V_5$: | $x_5, y_5, z_5$ |

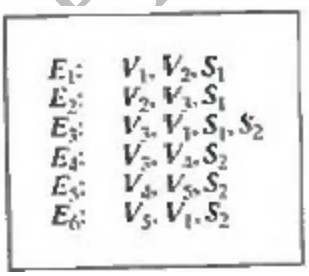
| EDGE TABLE | |
|---|---|
| $E_1$: | $V_1, V_2$ |
| $E_2$: | $V_2, V_3$ |
| $E_3$: | $V_3, V_1$ |
| $E_4$: | $V_3, V_4$ |
| $E_5$: | $V_4, V_5$ |
| $E_6$: | $V_5, V_1$ |

| SURFACE-FACET TABLE | |
|---|---|
| $S_1$: | $E_1, E_2, E_3$ |
| $S_2$: | $E_3, E_4, E_5, E_6$ |

**Geometric data** for the objects in a scene are arranged conveniently in three lists: a **vertex table**, an **edge table** and a **surface-facet table**. Coordinate values for each vertex in the object are stored in the vertex table. The edge table contains pointers back into the vertex table to

identify the vertices for each polygon edge. And the surface-facet table contains pointers back into the edge table to identify the edges for each polygon as shown in the above figure. In addition, individual objects and their component polygon faces can be assigned object and facet identifiers for easy reference.

Listing the geometric data in three tables provides convenient reference to the individual components (vertices, edges and surface facets) for each object. Also, the object can be displayed efficiently by using data from the edge table to identify polygon boundaries.




We can add extra information to the data tables for faster information extraction. For instance, we could expand the edge table to include forward pointers into the surface-facet table so that a common edge between polygons could be identified more rapidly. This is particularly useful for rendering procedures that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table could be expanded to reference corresponding edges, for faster information retrieval.

Additional geometric information that is usually stored in the data tables includes the slope for each edge and the coordinate extents for polygon edges, polygon facets, and each object in a scene.

Since the geometric data tables may contain extensive listings of vertices and edges for complex objects and scenes, it is important that the data be checked for consistency and completeness. When vertex, edge and polygon definitions are specified, it is possible, particularly in interactive applications, that certain input errors could be made that would distort the display of the objects.

The more information included in the data tables, the easier it is to check for errors. Therefore, error checking is easier when three data tables are used. Some of the tests that could be performed by a graphics package are:

1. That every vertex is listed as an endpoint for at least two edges.
2. That every edge is part of at least one polygon
3. That every polygon is closed
4. That each polygon has at least one shared edge
5. That if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

## Plane Equations:

Each polygon in a scene is contained within a plane of infinite extent. The general equation of a plane is

$$\mathbf{Ax + By + Cz + D = 0}$$

where (x, y, z ) is any point on the plane, and the coefficients A, B, C, and D are constants describing the, spatial properties of the plane. We can obtain the values of A , B, C, and D by solving a set of three plane equations using the coordinate values for three noncollinear points in the plane. For this purpose, we can select three successive convex-polygon vertices, ( $x_1$, $y_1$, $z_1$), ( $x_2$, $y_2$, $z_2$) and ( $x_3$, $y_3$, $z_3$), in a counterclockwise order and solve the following set of simultaneous linear plane equations for the ratios A/D, B/D, and C/D:

$$(A/D)x_k + (B/D)y_k + (C/D)z_k = -1, \qquad k = 1, 2, 3$$

The solution to this set of equations can be obtained in determinant form, using Cramer's rule, as

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$

$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

Expanding the determinants, we can write the calculations for the plane coefficients in the form

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$
$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$
$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$
$$D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1)$$

These calculations are valid for any three coordinate positions, including those for which D=0. When vertex coordinates and other information are entered into the polygon data structure, values for A, B, C and D can be computed for each polygon facet and stored with the other polygon data.

### Front and Back Polygon Faces:

Since we are usually dealing with polygon surfaces that enclose an object interior, we need to distinguish between the two sides of the surface. The side of the plane that faces the object interior is called the **back face**, and the visible or outward side is the **front face**.

Every polygon is contained within an infinite plane that partitions space into two regions. Any point that is not on the plane and that is visible to the front face of a polygon surface section is said to be in front of (or outside) the plane, and, thus, outside the object. And any point that is visible to the back face of the polygon is behind (or inside) the plane. A point that is behind (inside) all polygon surface planes is inside the object.

Plane equations are used also to identify the position of spatial points relative to the plane surfaces of an object. For any point (x, y, z) not on a plane with parameters A, B, C, D, we have

$$Ax + By + Cz + D \neq 0$$

We can identify the point as either inside or outside the plane surface according to the sign (negative or positive) of Ax + By + Cz + D:

if $Ax + By + Cz + D < 0$, the point $(x, y, z)$ is inside the surface

if $Ax + By + Cz + D > 0$, the point $(x, y, z)$ is outside the surface

These inequality tests are valid in a right-handed Cartesian system, provided the plane parameters A, B, C, and D were calculated using vertices selected in a counterclockwise order when viewing the surface in a front-to-back direction.

For example, in Fig. 3.52, any point outside (in front of) the shaded plane satisfies the inequality x - 1 > 0, while any point inside (in back of) the plane has an x coordinate value less than 1.

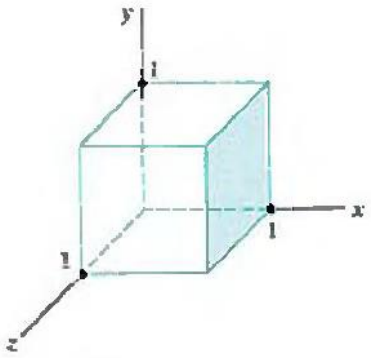



FIGURE 3-52 The shaded polygon surface of the unit cube has plane equation $x - 1 = 0$.

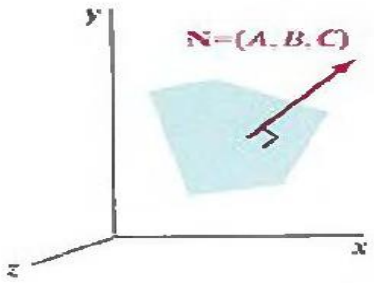



FIGURE 3-53 The normal vector N for a plane described with the equation $Ax + By + Cz + D = 0$ is perpendicular to the plane and has Cartesian components $(A, B, C)$.

Orientation of a plane surface in space can be described with the normal vector to the plane, as shown in Fig. 3.53. This surface normal vector has Cartesian components (A, B, C), where parameters A, B, and C are the plane coefficients.

To determine the components of the normal vector for the shaded surface shown in Fig. 3-52, we select three of the four vertices along the boundary of the polygon. These points are selected in a

counterclockwise direction as we view from outside the cube toward the origin. Coordinates for these vertices, in the order selected, can be used to obtain the plane coefficients: A = 1, B = 0, C = 0, D = -1. Thus, the normal vector for this plane is in the direction of the positive x axis.

The elements of the plane normal can also be obtained using a vector cross-product calculation. We again select three vertex positions, $\mathbf{V_1}$, $\mathbf{V_2}$, and $\mathbf{V_3}$, taken in counterclockwise order when viewing the surface from outside to inside in a right-handed Cartesian system. Forming two vectors, one from $\mathbf{V_1}$ to $\mathbf{V_2}$ and the other from $\mathbf{V_1}$ to $\mathbf{V_3}$, we calculate $\mathbf{N}$ as the vector cross product:

$$\mathbf{N} = (\mathbf{V_2} - \mathbf{V_1}) \times (\mathbf{V_3} - \mathbf{V_1})$$

This generates values for the plane parameters A, B, and C. We can then obtain the value for parameter D by substituting these values and the coordinates for one of the polygon vertices in plane equation 10-1 and solving for D. The plane equation can be expressed in vector form using the normal N and the position P of any point in the plane as

$$\mathbf{N} \cdot \mathbf{P} = -D$$

## OpenGL Polygon Fill Area Functions:

A **glVertex** function is used to input the coordinates for a single polygon vertex, and a complete polygon is described with a list of vertices placed between a **glBegin / glEnd** pair.

A polygon is displayed in a solid color, determined by the current color settings. As options we can fill a polygon with a pattern and we can display polygon edges as line borders around the interior fill. There are six different symbolic constants that we can use as the argument in the **glBegin** function to describe polygon fill areas. These six primitive constants allow us to display a single fill polygon, a set of unconnected fill polygons, or set of connected fill polygons.

In OpenGL, a fill area must be specified as a convex polygon. Thus, a vertex list for a fill polygon must contain at least three vertices, there can be no crossing edges, and all interior angles for the polygon must be less than 180°. And a single polygon fill area can be defined with only one vertex list, which precludes any specifications that contain holes in the polygon interior , as shown in the following figure. However, we could describe such a figure using two overlapping convex polygons

.

Each polygon that we specify has two faces: a back face and a front face. In OpenGL, fill color and other attributes can be set for each face separately and back/front identification is needed in both two-dimensional and three-dimensional viewing routines. Therefore polygon vertices should be specified in a counterclockwise order as we view the polygon from "outside". This identifies the front face for the polygon.

OpenGL provides a special rectangle function that directly accepts the vertex specification in the xy plane.

**glRect*(x1, y1, x2, y2);**

One corner of this rectangle is at coordinate position (x1,y1), and the opposite corner of the rectangle is at the position (x2,y2).

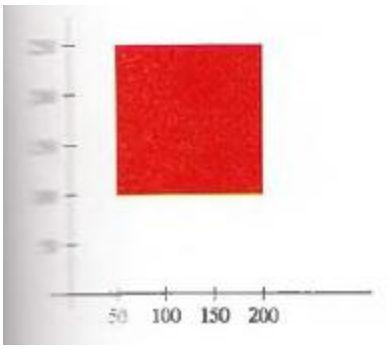
Suffix codes for **glRect** specify the coordinate data type and whether coordinates are to be expressed as array elements. These codes are i (for integer), s (for short), f (for float), d (for double), and v (for vector). The rectangle is displayed with edges parallel to the xy coordinate axes.

**Ex:**

**glRecti(200, 100, 50, 250);**

If we put the coordinate values for this rectangle into arrays, we can generate the same square with the following code

```
int vertex1[ ] = {200,100};
int vertex2[ ] = {50,250};
glRectiv(vertex1, vertex2);
```



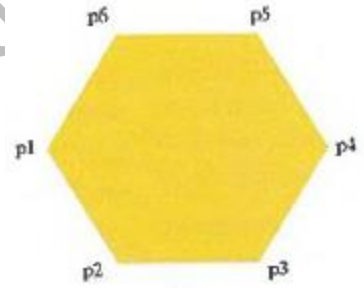

When a rectangle is generated with function glRect, the polygon edges are formed between the vertices in the order (x1,y1), (x1,y1), (x1,y1), (x1,y1) and then back to the first vertex.

Each of the other six polygon fill primitives is specified with a symbolic constant in the glBegin function, along with a list of glVertex commands.

With the OpenGL primitive constant **GL_POLYGON**, we can display a single polygon fill area such as shown in the following figure.

For this example, we assume that we have a list of six points, labeled p1 through p6, specifying two-dimensional polygon vertex positions in a counterclockwise ordering.

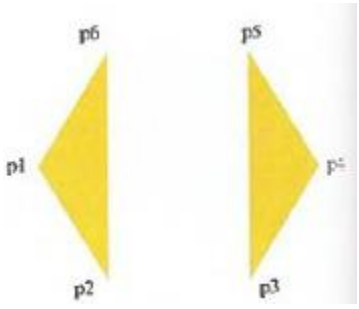
```
glBegin (GL_POLYGON);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
        glVertex2iv(p6);
glEnd();
```




Each of the points is represented as an array of (x,y) coordinate values.

If we reorder the vertex list and change the primitive constant in the previous code example to **GL_TRIANGLES** , we obtain the two separated triangle fill areas as shown in the following figure

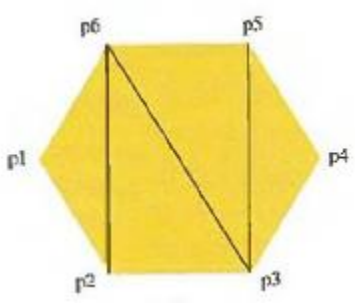
```
glBegin (GL_TRIANGLES);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p6);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
glEnd();
```



In the above case, the first three coordinate points define the vertices for one triangle, the next three points define the next triangle and so forth. For each triangle fill area, we specify the vertex positions in a counterclockwise order. A set of connected triangles is displayed with this primitive constant unless some vertex coordinates are repeated. Nothing is displayed if we do not list at least three vertices. And the number of vertices specified is not a multiple of three, the final one or two vertex positions are not used.

By reordering the vertex list once more and changing the primitive constant to **GL_TRIANGLE_STRIP**, we can display the set of connected triangles shown in the following figure.

```
glBegin (GL_TRIANGLE_STRIP);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p6);
        glVertex2iv(p3);
        glVertex2iv(p5);
        glVertex2iv(p4);
glEnd();
```

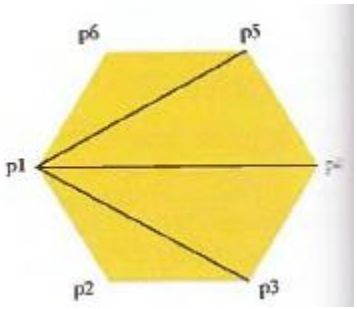



Assuming that no coordinate positions are repeated in a list of N vertices, we obtain N-2 triangles in the strip. Clearly, we must have N≥ 3 or nothing is displayed. In the above example, N=6 and we obtain four triangles. Each successive triangle shares an edge with the previously defined triangle, so the ordering of the vertex list must be set up to ensure a consistent display.

Each vertex position **n** in the vertex list is processes in the order n=1, n=2, . . , n = N-2 and arranging the order of the corresponding set of three vertices according to whether n is an odd number or an even number. If **n** is odd, the polygon table listing for the triangle vertices is in the order **n,n+1,n+2**. If n is even, the triangle vertices are listed in the order **n+1,n,n+2.**

Another way to generate a set of connected triangles is to use the "fan" approach as illustrated in the following figure, where all triangles share a common vertex. We obtain this arrangement of triangles using the primitive constant **GL_TRIANGLE_FAN** and the original ordering of our six vertices:

```
glBegin (GL_TRIANGLE_FAN);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
        glVertex2iv(p6);
glEnd();
```



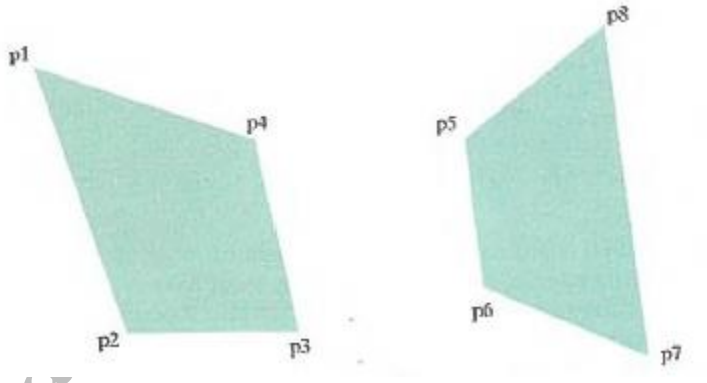
For N vertices, we obtain N – 2 triangles, providing no vertex positions are repeated, and we must list at least three vertices. In addition, the vertices must be specified in the proper order to correctly define front and back faces for each triangle. The first coordinate position listed is a vertex for each triangle in the fan. If the coordinate positions are listed as n =1, n=2,..n=N-2, then vertices for triangle n are listed in the polygon tables in the order 1, n+1, n+2.

OpenGL also provides two types of quadrilaterals.

**GL_QUADS:**

With the GL_QUADS primitive constant and the following list of eight vertices, specified as two-dimensional coordinate arrays, we can generate the display as shown below:

```
glBegin (GL_QUADS);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
        glVertex2iv(p6);
        glVertex2iv(p7);
        glVertex2iv(p8);
glEnd();
```

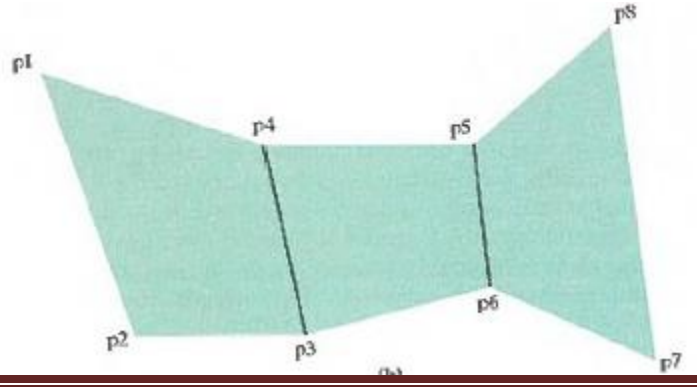



The first four coordinate points define the vertices for one quadrilateral, the next four points define the next quadrilateral, and so on. For each quadrilateral fill area, we specify the vertex positions in a counterclockwise order. If no vertex coordinates are repeated, we display a set of unconnected four-sided fill areas. We must list at least four vertices with this primitive. Otherwise the nothing is displayed. And if the number of vertices is not a multiple of four, the extra vertex positions are ignored.

**GL_QUAD_STRIP:**

With the GL_QUAD_STRIP primitive constant we can obtain a set of connected quadrilaterals as shown in the following figure.

```
glBegin (GL_QUADS);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p4);
        glVertex2iv(p3);
        glVertex2iv(p5);
        glVertex2iv(p6);
```



```
        glVertex2iv(p8);
        glVertex2iv(p7);
glEnd();
```

A quadrilateral is set up for each pair of vertices specified after the first two vertices in the list, and we need to list the vertices so that we generate a correct counterclockwise vertex ordering for each polygon. For a list of N vertices, we obtain (N/2)-1 quadrilaterals, providing that N≥4.

If N is not a multiple of 4, any extra coordinate positions in the list are not used. We can enumerate these fill polygons and the vertices listed as n=1,n=2,…,n=(N/2)-1. Then polygon tables will list the vertices for quadrilateral n in the vertex order number 2n-1,2n,2n+2,2n+1.

Most graphics packages display curved surfaces as a set of approximating plane facets. This is because the plane equations are linear, and processing the liner equations is much quicker than processing quadric or other types of curve equations.

Although the OpenGL core library allows only convex polygons, the OpenGL Utility (GLU) provides functions for dealing with concave polygons and other nonconvex objects with linear boundaries. A set of GLU polygon tessellation routines is available for converting such shapes into a set of triangles, triangle meshes, triangle fans and straight line segments.

## Fill-Area Attributes:

There are two basic procedures for filling an area on raster systems, once the definition of the fill region has been mapped to pixel coordinates.

One procedure first determines the overlap intervals for scan lines that cross the area. Then, pixel positions along these overlap intervals are set to the fill color.

Another method for area filling is to start from a given interior position and "paint" outward, pixel-by-pixel, from this point until we encounter specified boundary conditions.

### Fill Styles:

A basic fill-area attribute provided by a general graphics library is the display style of the interior. We can display a region with a single color, a specified pattern, or in a "hollow" style by showing only the boundary of the region, as illustrated in the following figures.

We can also fill selected regions of a scene using various brush styles, color-blending combinations, or textures. We could also show the edges in different colors, widths, and styles. We can also select different display attributes for the front and back faces of a region.

Fill patterns can be defined in rectangular color arrays that list different colors for different positions in the array. Or, a fill pattern could be specified as a bit array that indicates which relative positions are to be displayed in a single selected color.

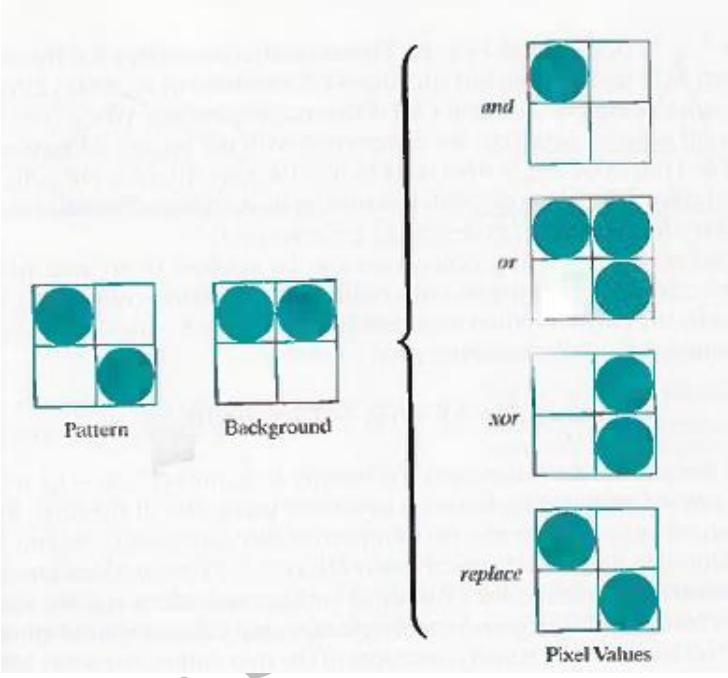
Some graphics systems provide an option for selecting an arbitrary initial position for overlying the mask. From this starting position, the mask is replicated in the horizontal and vertical directions until the display area is filed with nonoverlapping copies of the pattern. Where the pattern overlaps specified fill areas, the array pattern indicates which pixels should be displayed in a particular color. This process of filling an area with a rectangular pattern is called **tiling**, and a rectangular fill pattern is sometimes referred to as **tiling pattern**.

Sometime, predefined fill patterns are available in a system, such as the hatch fill patterns as shown in the figure below:


Hatch fill could be applied to regions by drawing sets of line segments to display either single hatching or crosshatching. Spacing and slope for the hatch lines could be set as parameters in a hatch table.

## Color-Blended Fill Regions:

It is possible to combine a fill pattern with background colors in various ways. A pattern could be combined with background colors using a transparency factor that determine show much of a background should be mixed with the object color. Or we could use simple logical or replace operations. The following figure demonstrates how logical and replace operations would combine a 2 by 2 fill pattern with a background pattern for a binary (black-and-white) system.




Some fill methods using blended colors have been referred to as **soft-skill** or **tint-fill** algorithms. One use of these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges. Another application of a soft-fill algorithm is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of the brush color and the background colors "behind" the area. In either case, we want the new fill color to have the same variations over the area as the current fill color.

The linear soft-fill algorithm repaints an area that was originally painted by merging a foreground color **F** with a single background color **B**, where **F** = **B**. Assuming we know the values for **F** and **B**, we can determine how these colors were originally combined by checking the current color contents of the frame buffer. The current RGB color **P** of each pixel within the area to be refilled is some linear combination of **F** and **B**:

$$\mathbf{P} = t\mathbf{F} + (1 - t)\mathbf{B} \qquad (1)$$

where the "transparency" factor **t** has a value between 0 and 1 for each pixel. For values of **t** less than 0.5, the background color contributes more to the interior color of the region than does the fill color. The vector equation (1) holds for each RGB component of the colors, with

$$\mathbf{P} = (P_R, P_G, P_B), \qquad \mathbf{F} = (F_R, F_G, F_B), \qquad \mathbf{B} = (B_R, B_G, B_B) \qquad (2)$$

We can thus calculate the value of parameter **t** using one of the RGB color components as

$$t = \frac{P_k - B_k}{F_k - B_k} \qquad (3)$$

where k = R, G, or B; and $F_k \neq B_k$.

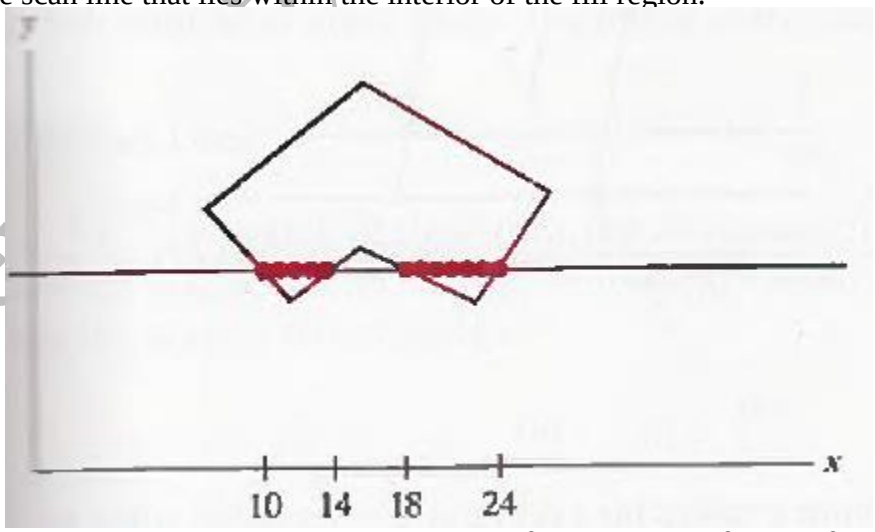
Similar soft-fill procedures can be applied to an area whose foreground color is to be merged with multiple background color areas, such as a checkerboard pattern. When two background colors $B_1$ and $B_2$ are mixed with foreground color **F**, the resulting pixel color **P** is

$$\mathbf{P} = t_0\mathbf{F} + t_1\mathbf{B}_1 + (1 - t_0 - t_1)\mathbf{B}_2$$

where the sum of the coefficients $t_0$, $t_1$, and $(1 - t_0 - t_1)$ on the color terms must equal 1.

## General Scan-Line Polygon-Fill Algorithm:

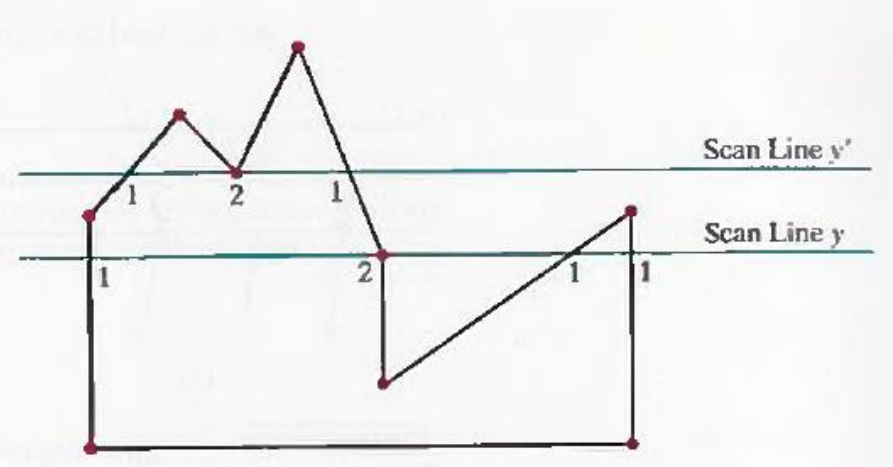
A scan-line fill of a region is performed by first determining the intersection positions of the boundaries of the fill region with the screen scan lines. Then the fill colors are applied to each section of the scan line that lies within the interior of the fill region.




The above figure illustrates the scan-line procedure for solid color filling of polygon areas. For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and

the corresponding frame-buffer positions between each intersection pair are set to the specified fill color. In the example of above figure, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from x = 10 to x = 14 and from x = 18 to x = 24.

Some scan-line intersections at polygon vertices require special handling. A scan line passing through a vertex intersects two polygon edges at that position, adding two points to the list of intersections for the scan line.

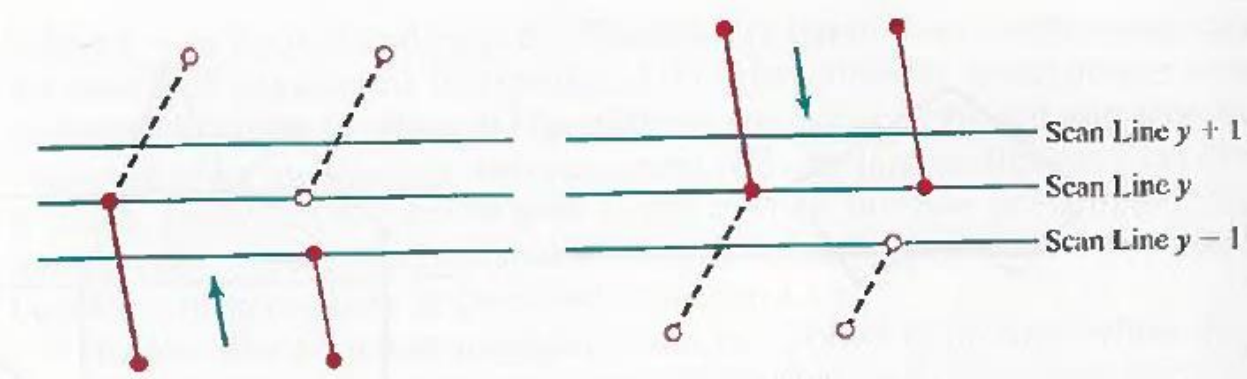



The above figure shows two scan lines at positions y and y' that intersect edge endpoints. Scan line y intersects five polygon edges. Scan line y', however, intersects an even number of edges although it also passes through a vertex. Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points.

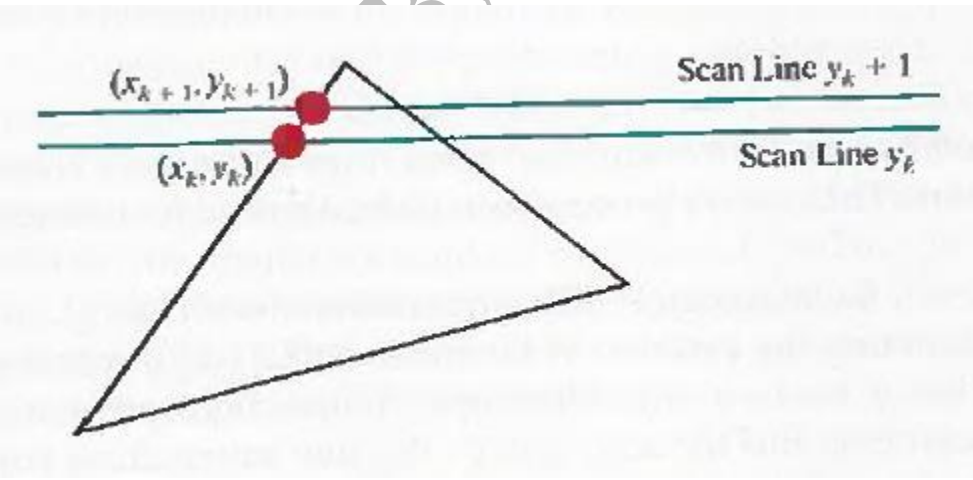
The topological difference between scan line y and scan line y' in the above figure is identified by noting the position of the intersecting edges relative to the scan line. For scan line y, the two intersecting edges sharing a vertex are on opposite sides of the scan line. But for scan line y', the two intersecting edges are both above the scan line. Thus, the vertices that require additional processing are those that have connecting edges on opposite sides of the scan line. We can identify these vertices by tracing around the polygon boundary either in clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next. If the endpoint y values of two consecutive edges monotonically increase or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex. Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polvgon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

One way to resolve the question as to whether we should count a vertex as one intersection or two is to shorten some polygon edges to split those vertices that should be counted as one intersection. We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise. As we process each edge, we can check to determine whether that edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint y values. If so, the lower edge can be shortened to ensure that

only one intersection point is generated for the scan line going through the common vertex joining the two edges. The following figure illustrates shortening of an edge. When the endpoint y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1, as in Figure(a). When the endpoint y values are monotonically decreasing, as in Figure(b), we decrease they coordinate of the upper endpoint of the edge following the current edge.




Calculations performed in scan-conversion and other graphics algorithms typically take advantage of various coherence properties of a scene that is to be displayed. What we mean by coherence is simply that the properties of one part of a scene are related in some way to other parts of the scene so that the relationship can be used to reduce processing. Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines. In determining edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next.




The following figure shows two successive scan lines crossing a left edge of a polygon. The slope of this polygon boundary line can be expressed in terms of the scan-line intersection coordinates:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k}$$

Since the change in $y$ coordinates between the two scan lines is simply

$$y_{k+1} - y_k = 1$$

the x-intersection value $x_{k+1}$ on the upper scan line can be determined from the x-intersection value $x_k$ on the preceding scan line as

$$x_{k+1} = x_k + \frac{1}{m}$$

Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

An obvious parallel implementation of the fill algorithm is to assign each scan line crossing the polygon area to a separate processor. Edge-intersection calculations are then performed independently. Along an edge with slope m, the intersection $x_k$ value for scan line *k* above the initial scan line can be calculated as

$$x_k = x_0 + \frac{k}{m}$$

In a sequential fill algorithm, the increment of x values by the amount l/m along an edge can be accomplished with integer operations by recalling that the slope m is the ratio of two integers:

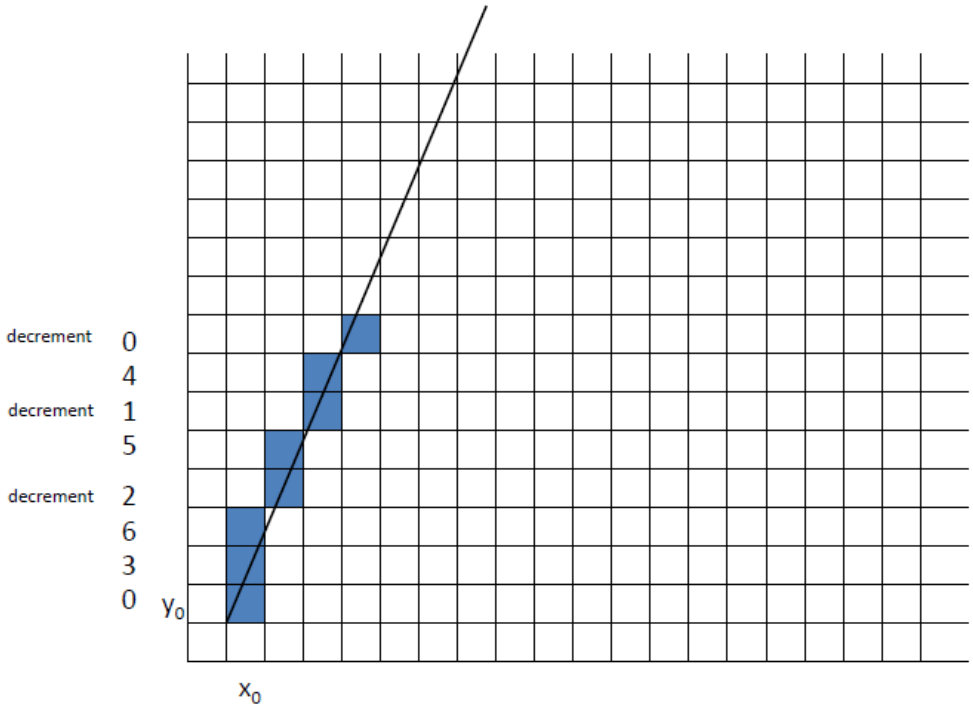
$$m = \frac{\Delta y}{\Delta x}$$

where Δx and Δy are the differences between the edge endpoint x and y coordinate values. Thus, incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$$

Using this equation, we can perform integer evaluation of the x intercepts by initializing a counter to 0, then incrementing the counter by the value of Δx each time we move up to a new scan line. Whenever the counter value becomes equal to or greater than Δy, we increment the current x intersection value by 1 and decrease the counter by the value Δy. This procedure is equivalent to maintaining integer and fractional parts for x intercepts and incrementing the fractional part until we reach the next integer value.

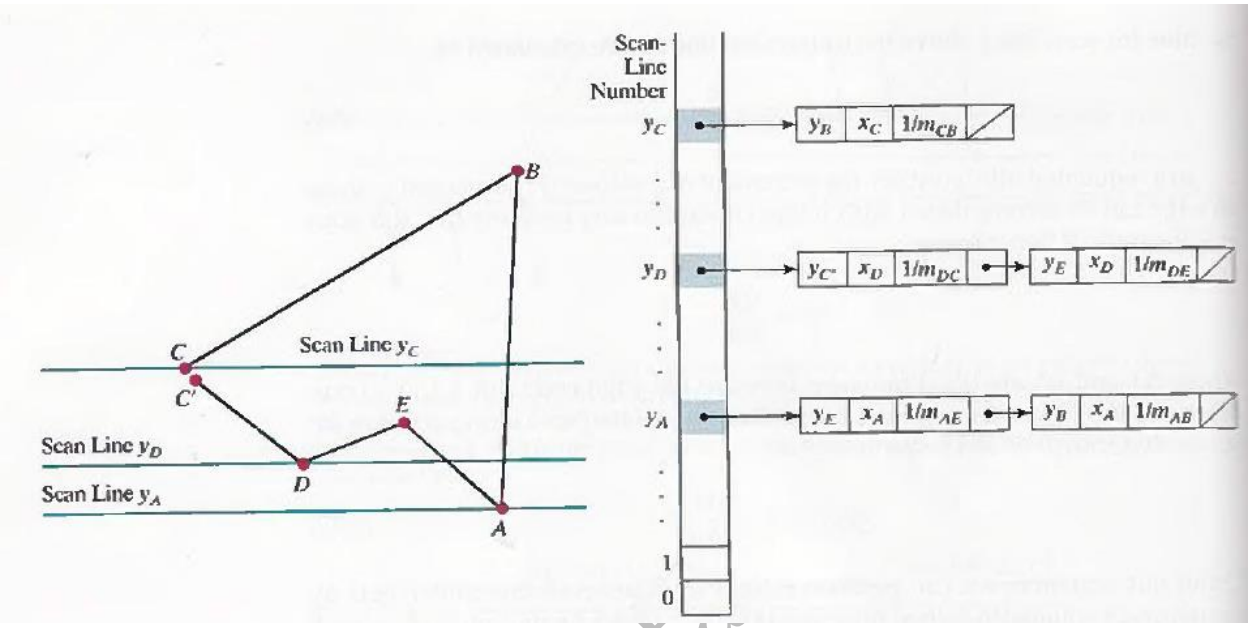
As an example of integer incrementing, suppose we have an edge with slope m = 7/3. At the initial scan line, we set the counter to 0 and the counter increment to 3. As we move up to the next three scan lines along this edge, the counter is successively assigned the values 3, 6, and 9. On the third scan line above the initial scan line, the counter now has a value greater than 7. So we increment the x-intersection coordinate by 1, and reset the counter to the value 9 - 7 = 2. We continue determining the scan-line intersections in this way until we reach the upper endpoint of the edge. Similar calculations are carried out to obtain intersections for edges with negative slopes.




We can round to the nearest pixel x-intersection value, instead of truncating to obtain integer positions, by modifying the edge-intersection algorithm so that the increment is compared to $\Delta y/2$. This can be done with integer arithmetic by incrementing the counter with the value $2\Delta x$ at each step and comparing the increment to $\Delta y$. When the increment is greater than or equal to $\Delta y$, we increase the x value by 1 and decrement the counter by the value of $2\Delta y$. In our previous example with m = 7/3, the counter values for the first few scan lines above the initial scan line on this edge would now be 6, 12 (reduced to -2), 4, 10 (reduced to -4), 2, 8 (reduced to -6), 0, 6, and 12 (reduced to - 2). Now x would be incremented on scan lines 2, 4, 6, 9, etc., above the initial scan line for this edge. The extra calculations required for each edge are $2\Delta x = \Delta x + \Delta x$ and $2\Delta y = \Delta x + \Delta x$.

To efficiently perform a polygon fill, we can first store the polygon boundary in a **sorted edge table** that contains all the information necessary to process the scan lines efficiently. Proceeding around the edges in either a clockwise or a counterclockwise order, we can use a bucket sort to store the edges, sorted on the smallest y value of each edge, in the correct scan-line positions. Only nonhorizontal edges are entered into the sorted edge table. As the edges are processed, we can also shorten certain edges to resolve the vertex-intersection question. Each entry in the table

for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge. For each scan line, the edges are in sorted order from left to right. The following Figure shows a polygon and the associated sorted edge table.




Next, we process the scan lines from the bottom of the polygon to its top, producing an active edge list for each scan line crossing the polygon boundaries. The active edge list for a scan line contains all edges crossed by that scan line, with iterative coherence calculations used to obtain the edge intersections.

## OpenGL Fill Area Attribute Functions:

We generate displays of filled convex polygons in four steps:

1. Define a fill pattern
2. Invoke the polygon-fill routine.
3. Activate the polygon-fill feature of OpenGL
4. Describe the polygons to be filled.

A polygon fill pattern is displayed up to and including the polygon edges. Thus, there are no boundary lines around the fill region unless we specifically add them to the display.

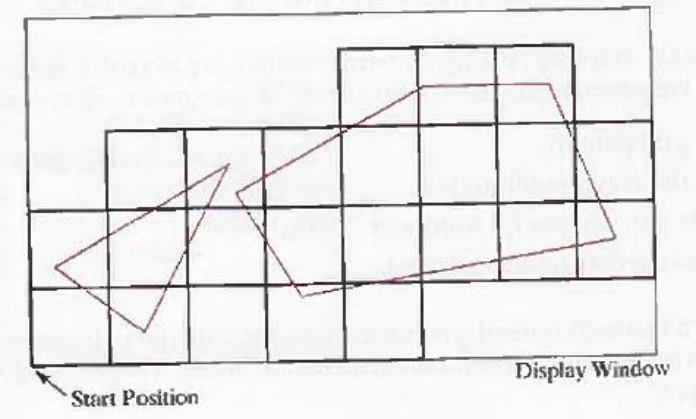
In addition to specifying a fill pattern for a polygon interior, there are a number of other options available. One option is to display a hollow polygon, where no interior color or pattern is applied and only the edges are generated. A hollow polygon is equivalent to the display of a closed polyline primitive. Another option is to show the polygon vertices, with no interior fill and no edges. Also, we designate different attributes for the front and back faces of a polygon fill areas.

## OpenGL Fill-Pattern Function:

By default, a convex polygon is displayed as a solid-color region, using the current color setting. To fill the polygon with a pattern in OpenGL, we use a 32-bit by 32-bit mask. A value of 1 in the mask indicates that the corresponding pixel is to be set to the current color, and a 0 leaves the value of that frame-buffer position unchanged. We define a bit pattern with hexadecimal values as, for example,

```
GLubyte fillPattern [ ] = {
        0xff, 0x00, 0xff, 0x00, ….}
```

The bits must be specified starting with the bottom row of the pattern, and continuing up to the topmost row (32) of the pattern. This pattern is replicated across the entire area of the display window, starting at the lower-left window corner, and specified polygons are filled where the pattern overlaps those polygons as shown in the figure below:




Once we have set a mask, we can establish it as the current fill pattern with the function

```
glPolygonStipple (fillPattern);
```

Next, we need to enable fill routines before we specify the vertices for the polygons that are to be filled with the current pattern. We do this with the statement

```
glEnable(GL_POLYGON_STIPPLE);
```

Similarly, we turn off pattern filling with

```
glDisable(GL_POLYGON_STIPPLE);
```

## OpenGL Texture and Interpolation Patterns:

We can produce fill patterns that simulate the surface appearance of wood, brick, brushed steel, or some other material. Also, we obtain an interpolation coloring of a polygon interior. We assign different colors to polygon vertices. Interpolation fill of a polygon interior is used to produce realistic displays of shaded surfaces under various lighting conditions.

**Ex:**

The following code segment assigns either a blue, red, or green color to each of the three vertices of a triangle. The polygon fill is then a linear interpolation of the colors at the vertices.

```
glShadeModel(GL_SMOOTH);
        glBegin (GL_TRIANGLES);
        glColor3f(0.0,0.0,1.0);
        glVertex2i(50,50);
        glColor3f(1.0,0.0,0.0);
        glVertex2i(50,50);
        glColor3f(0.0,1.0,0.0);
        glVertex2i(50,50);
glEnd();
```

## OpenGL Wire-Frame Methods:

We can also choose to show only polygon edges. This produces a wire-frame or hollow display of the polygon. Or we could display a polygon by only plotting a set of points at the vertex positions. These options are selected with the function

```
glPolygonMode(face, disaplayMode);
```

We use parameter face to designate which face of the polygon we want to show as edges only or vertices only. This parameter is then assigned either **GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK**. Then if we want only the polygon edges displayed for our selection, we assign the constant **GL_LINE** to parameter displayMode.

To plot only the polygon vertex points, we assign the constant **GL_POINT** to parameter displayMode. A third option is **GL_FILL**. But this is the default mode, so we usually only invoke **glPolygonMode** when we want to set attributes for the polygon edges or vertices.

Another option is to display a polygon with both an interior fill and a different color or pattern for its edges (or for its vertices). This is accomplished by specifying the polygon twice: once with parameter displayMode set to **GL_FILL** and then again with displayMode set to **GL_LINE (GL_POINT).**

**Ex:**

```
glColor3f(0.0,1.0,0.0);

/*Invoke polygon-generating routine*/

glColor3f(1.0,0.0,0.0);

glPloygonMode(GL_FRONT, GL_LINE);

/*Invoke polygon-generating routine again*/
```

For a three dimensional polygon (one that does not have all vertices in the xy plane), this method for displaying the edges of a filled polygon may produce gaps along the edges. This effect, sometimes reffered to as **stitching**, is caused by the differences between calculations in the scan-line fill algorithm and calculations in the edge line-drawing algorithm.

One way to eliminate the gaps along displayed edges of a three-dimensional polygon is to shift the depth values calculated by the fill routine so that they do not overlap with the edge depth values for that polygon. We do this with the following two OpenGL functions.

```
glEnable(GL_POLYGON_OFFSET_FILL);

glPolygonOffset(factor1,factor2);
```

The first function activates the offset routine for scan-line filling, and the second function is used to set a couple of floating –point values factor1 and factor2 that are used to calculate the amount of depth offset.

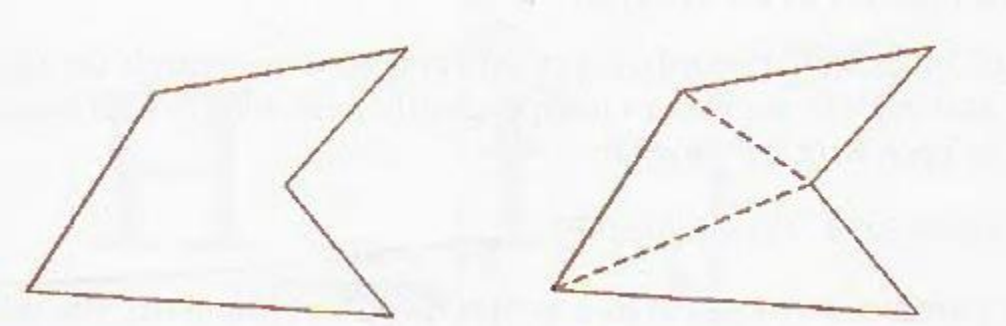
**depthOffset = factor1.maxSlope + factor2.const**

where **maxSlope** is the maximum slope of the polygon and **const** is an implementation constant.

**Ex:** Assigning values to offset factors:

```
glColor3f(0.0,1.0,0.0);

glEnable(GL_POLYGON_OFFSET_FILL);

/*Invoke polygon-generating routine*/

glDisable(GL_POLYGON_OFFSET_FILL);

glColor3f(1.0,0.0,0.0);

glPloygonMode(GL_FRONT, GL_LINE);

/*Invoke polygon-generating routine again*/
```

To display a concave polygon using OpenGL routines, we must first split it into a set of convex polygons. We typically divide a concave polygon into a set of triangles as shown in the figure below:



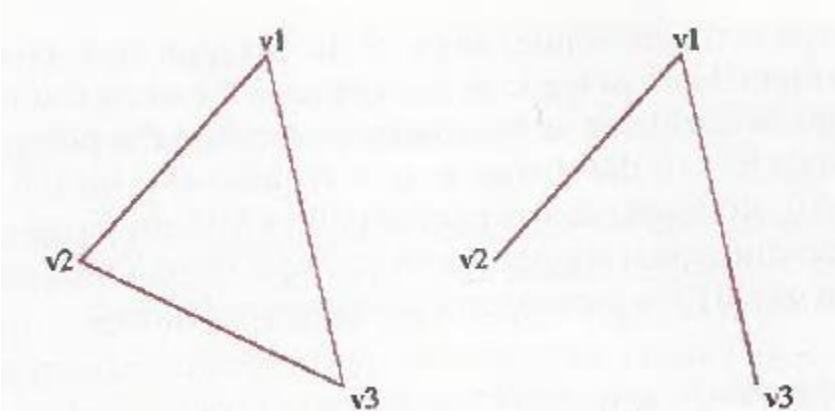
OpenGL provides a mechanism that allows us to eliminate selected edges from a wire-frame display. Each polygon vertex is stored with a one-bit flag that indicates whether or not that vertex is connected to the next vertex by a boundary edge. We need to set that bit flag to "off" and the edge following that vertex will not be displayed. We set this flag for an edge with the following function.

**glEdgeFlag(flag);**

To indicate that a vertex does not precede a boundary edge, we assign the OpenGL constant GL_FALSE to parameter **flag**. This applies to all subsequently specified vertices until the next call to **glEdgeFlag** is made. The OpenGL constant GL_TRUE turns the edge flag back on again, which is the default. The function **glEdgeFlag** can be placed between glBegin/glEnd pairs.

Ex:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin();
        glVertex3fv(v1);
        glEdgeFlag(GL_FALSE);
        glVertex3fv(v2);
        glEdgeFlag(GL_TRUE);
        glVertex3fv(v3);
glEnd();
```



**OpenGL Front-Face Function:**

Although, by default, the ordering of polygon vertices controls the identification of front and back faces, we can independently label selected surfaces in a scene as front or back with the function

**glFrontFace(vertexOrder);**

If we set parameter **vertexOrder** to the OpenGL constant **GL_CW,** then a subsequently defined polygon with a clockwise ordering for its vertices is considered to be front facing. This OpenGL feature can be used to swap faces of a polygon for which we have specified vertices in a clockwise order. The constant **GL_CCW** labels a counterclockwise ordering of polygon vertices as front facing, which is the default ordering.

## Geometric Transformations:

Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations.**

## Basic Two-Dimensional Transformation:

The geometric-transformation functions that are available in all graphics packages are those for translation, rotation and scaling. Other useful routines that area sometimes included in a package are reflection and shearing operations.

### Two-Dimensional Translation:

We perform a **translation** on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position. In effect, we are moving the original point position along a straight-line path to its new location.

A translation is applied to an object that is defined with multiple coordinates positions, such as a quadrilateral, by relocating all the coordinate positions by the same displacement along parallel paths. Then the complete object is displayed at the new location.

To translate a two-dimensional position, we add translation distances tx and ty to the original coordinates (x,y) to obtain the new coordinate position ($\mathbf{x}^{|}$,$\mathbf{y}^{|}$) as shown in the following figure.

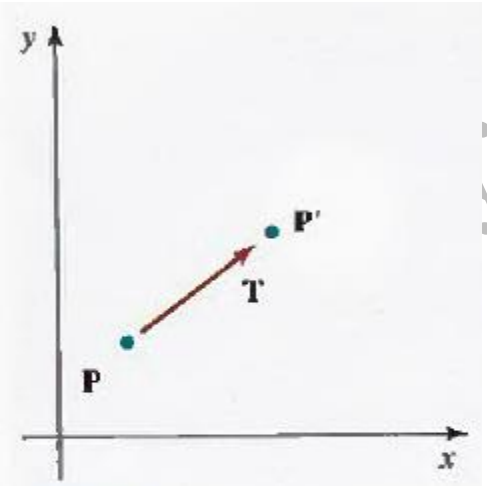



FIGURE 5–1 Translating a point from position P to position P′ using a translation vector T.

$$\mathbf{x}^{|} = \mathbf{x} + \mathbf{tx}, \qquad \mathbf{y}^{|} = \mathbf{x} + \mathbf{tx}$$

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

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

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

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

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

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 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. The following figure illustrates the application of a specified translation vector to move an object from one position to another.

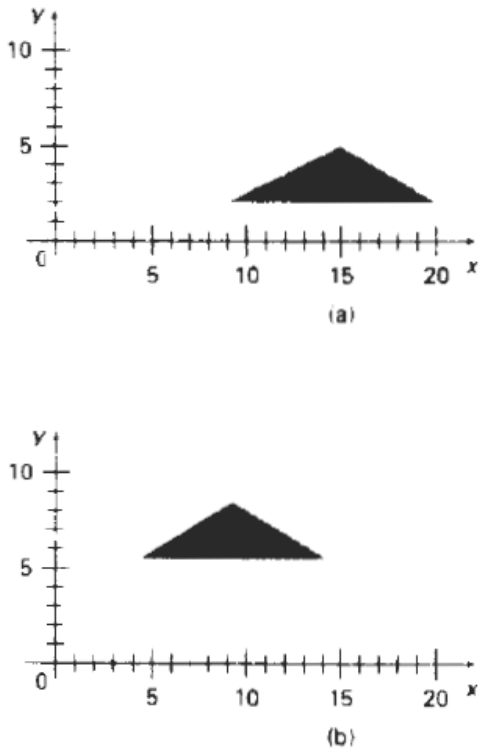



*Figure 5-2*
Moving a polygon from position (a) to position (b) with the translation vector (-5.50, 3.75).

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void translatePolygon (wcPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
   GLint k;

   for (k = 0; k < nVerts; k++) {
      verts [k].x = verts [k].x + tx;
      verts [k].y = verts [k].y + ty;
   }
   glBegin (GL_POLYGON);
      for (k = 0; k < nVerts; k++)
         glVertex2f (verts [k].x, verts [k].y);
   glEnd ( );
}
```

The above routine illustrates the translation operation. An input translation vector is used to move the n vertices of a polygon from one world-coordinate position to another, and OpenGL routines are used to regenerate the translated polygon.

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 translate the points that define the curve paths and then reconstruct the curve sections between the new coordinate positions.

**Two-Dimensional Rotation:**

We generate a **rotation** transformation of an object by specifying a **rotation axis** and a **rotation angle**. All points of the objects are then transformed to new positions by rotating the points through the specified angle about the rotation axis.

A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the xy plane. In this case, we are rotating the object about a rotation axis that is perpendicular to 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 as shown in the figure 5.3. Positive values for the rotation angle θ define counterclockwise rotations about the pivot point, as in Fig. 5-3, and negative values rotate objects in the clockwise direction.

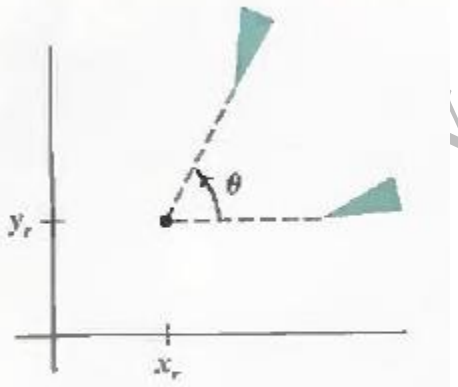



FIGURE 5-3 Rotation of an object through angle $\theta$ about the pivot point $(x_r, y_r)$.

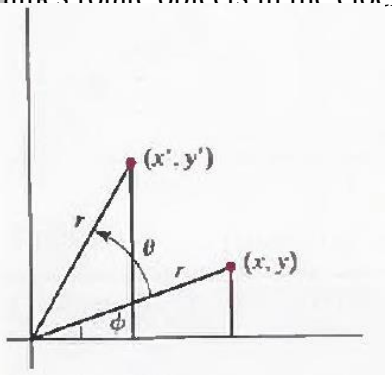



FIGURE 5-4 Rotation of a point from position $(x, y)$ to position $(x', y')$ through an angle $\theta$ relative to the coordinate origin. The original angular displacement of the point from the $x$ axis is $\phi$.

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 $\phi$ 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

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta \qquad (5\text{-}4)$$

The original coordinates of the point in polar coordinates are

$$x = r\cos\phi, \qquad y = r\sin\phi \qquad (5\text{-}5)$$

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:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta \qquad (5\text{-}6)$$

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

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \qquad (5\text{-}7)$$

where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad (5\text{-}8)$$

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

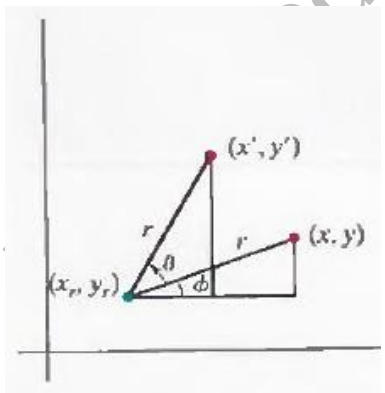



FIGURE 5-5 Rotating a point from position (x, y) to position (x', y') through an angle θ about rotation point ($x_r$, $y_r$).

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

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 includt: pivot coordinates by matrix addition of a column vector whose elements contain the additive (translational) terms in Eqs. 5-9.

Rotations are also rigid-body transformations that move objects with distortion.

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 two 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 pivot point by moving the center position through the arc that subtends the specified rotation angle.

In the following code example, a polygon is rotated about a specified world-coordinate pivot point.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt,
                    GLdouble theta)
{
   wcPt2D * vertsRot;
   GLint k;

   for (k = 0; k < nVerts; k++) {
      vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta)
                               - (verts [k].y - pivPt.y) * sin (theta);
      vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta)
                               + (verts [k].y - pivPt.y) * cos (theta);
   }
   glBegin (GL_POLYGON);
      for (k = 0; k < nVerts; k++)
         glVertex2f (vertsRot [k].x, vertsRot [k].y);
   glEnd ( );
}
```

Parameters input to the rotation procedure are the original vertices of the polygon, the pivot-point coordinates, and the rotation angle *theta* specified in radians. Following the transformation of the vertex positions, the polygon is regenerated using OpenGL routines.

## Two-dimensional Scaling:

To alter the size of an object, we apply a **scaling** transformation. A simple two dimensional scaling operation is performed by multiplying object positions (x,y) 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 basic two-dimensional scaling 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 produced 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, whew pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Fig. 5-6)

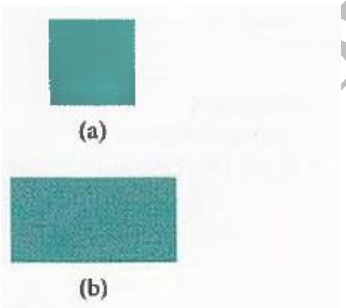



FIGURE 5–6 Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

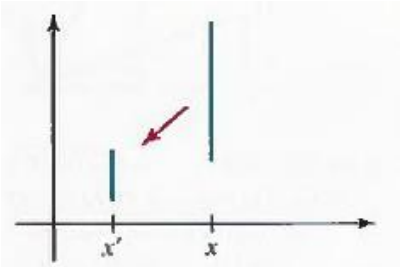



FIGURE 5–7 A line scaled with Eq. 5-12 using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.

Objects transformed with Eq. 5-11 are both scaled and repositioned. Scaling factors with absolute values less than 1 move objects closer to the coordinate origin, while absolute values greater than 1 move coordinate positions farther from the origin. Figure5-7 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).

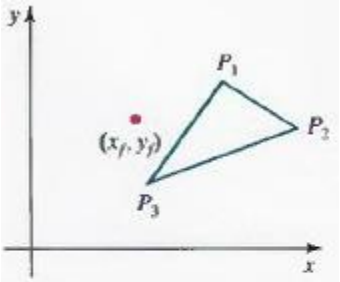



FIGURE 5-8 Scaling relative to a chosen fixed point $(x_f, y_f)$. The distance from each polygon vertex to the fixed point is scaled by transformation equations 5-13.

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 ($x_f$, $y_f$), 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 $\mathbf{x_f(1 - s_x)}$ and $\mathbf{y_f(1 - s_y)}$ are constant for all points in the object.

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 mu1tiplication.

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.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt,
                   GLfloat sx, GLfloat sy)
{
   wcPt2D vertsNew;
   GLint k;

   for (k = 0; k < nVerts; k++) {
     vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
     vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
   }
   glBegin (GL_POLYGON);
      for (k = 0; k < nVerts; k++)
         glVertex2f (vertsNew [k].x, vertsNew [k].y);
   glEnd ( );
}
```

The above procedure illustrates an application of the scaling calculations for a polygon. Coordinates for the polygon vertices and for the fixed point are input parameters, along with the scaling factors. After the coordinate transformations, OpenGL routines are used to generate the scale polygon.

## Matrix Representations and Homogeneous Coordinates

Each of the basic transformations can be expressed in the general matrix form

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \qquad (5\text{-}15)$$

with coordinate positions P and P' represented as column vectors.

Matrix M1 is a 2 by 2 array containing multiplicative factors, and $M_2$ is a two-element column matrix containing translational terms. For translation, M1 is the identity matrix. For rotation or scaling, M2 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 M2.

**Homogenous coordinates:**

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

$$x = \frac{x_h}{h} \quad , \quad y = \frac{y_h}{h}$$

Therefore, a general homogeneous coordinate representation can also be written as ***(h.x, h.y, h)***.

For two-dimensional geometric transformations, we can choose the homogenous 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 n times that parameter, the value n can he factored out of the equations.

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications, which is standard method used in graphics systems. Two dimensional coordinate positions are represented with three-element column vectors, and two –dimensional transformation operations are expressed as 3 by 3 matrices.

**Two-Dimensional Translation Matrix:**

Using a homogenous-coordinate approach, we can represent the equations for a two-dimensional translation of a coordinate position using the following matrix multiplication.

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

This translation operation can be written in the abbreviated form

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

with $\mathbf{T}(t_x, t_y)$ as the 3 by 3 translation matrix in Eq. 5-17. We can simply represent the translation matrix as **T**

**Two-Dimensional Rotation Matrix:**

Similarly, two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

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

or as

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

The rotation transformation operator R(θ) 1s the 3 by 3 matrix in Eq. 5-19 with rotation parameter θ. We can also write this rotation matrix simply as **R**

**Two-Dimensional Scaling Matrix:**

A scaling transformation relative to the coordinate origin is now expressed as the matrix multiplication

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

Or

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

where $\mathbf{S}(s_x, s_y)$ is the 3 by 3 matrix in Eq. 5-21 with parameters $s_x$ and $s_y$.

## Inverse Transformations:

For translation, we obtain the inverse matrix by negating the translation distances. Thus, if we have two-dimensional translation distances $t_x$ and $t_y$, the inverse translation matrix is

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (5\text{-}23)$$

This produces a translation in the opposite direction, and the product of a translation matrix and its inverse produces the identity matrix.

An inverse rotation is accomplished by replacing the rotation angle by its negative. For example, a two-dimensional rotation through an angle $\theta$ about the coordinate origin has the inverse transformation matrix

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5\text{-}24)$$

Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produced when any rotation matrix is multiplied by its inverse. Since only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. That is, we can calculate the inverse of any rotation matrix $\mathbf{R}$ by evaluating its transpose ($\mathbf{R}^{-1} = \mathbf{R}^T$).

We form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals. For two-dimensional scaling with parameters $s_x$ and $s_y$ applied relative to the coordinate origin, the inverse transformation matrix is

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5\text{-}25)$$

The inverse matrix generates an opposite scaling transformation, so the multiplication of any scaling matrix with its inverse produces the identity matrix.

## Two Dimensional Composite Transformations:

Using matrix representations, 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.

Since a coordinate position is represented with a homogenous column matrix, we must pre-multiply the column matrix by the matrices representing any transformation sequence. Since many positions in a scene are typically transformed by the same sequence, it is more efficient to first multiply the transformation matrices to form a single composite matrix.

Thus, if we want to apply two transformations to point position P, the transformed location would be calculated as

$$\mathbf{P}' = \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P}$$

$$= \mathbf{M} \cdot \mathbf{P}$$

The coordinate position is transformed using the composite matrix **M**, rather than applying the individual transformations $\mathbf{M}_1$ and then $\mathbf{M}_2$.

## Composite Two-Dimensional Translations:

If two successive translation vectors ($t_{1x}$, $t_{1y}$) and ($t_{2x}$, $t_{2y}$) are applied to a two-dimensional coordinate position **P**, the final transformed location **P'** is calculated as

$$\mathbf{P}' = \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\}$$
$$= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P}$$

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

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \quad (5\text{-}28)$$

Or

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}) \qquad (5\text{-}29)$$

which demonstrates that two successive translation are additive.

## Composite Two-Dimensional Rotations:

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

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned} \qquad (5\text{-}30)$$

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

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \qquad (5\text{-}31)$$

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

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

## Composite Two-Dimensional Scalings:

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

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

or

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, \; s_{1y} \cdot s_{2y})$$

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 Two-Dimensional Pivot-Point Rotation:

When a graphics package that provides only a rotate function for revolving objects about the coordinate origin, we can generate rotations about any selected pivot point (x, y,) 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.

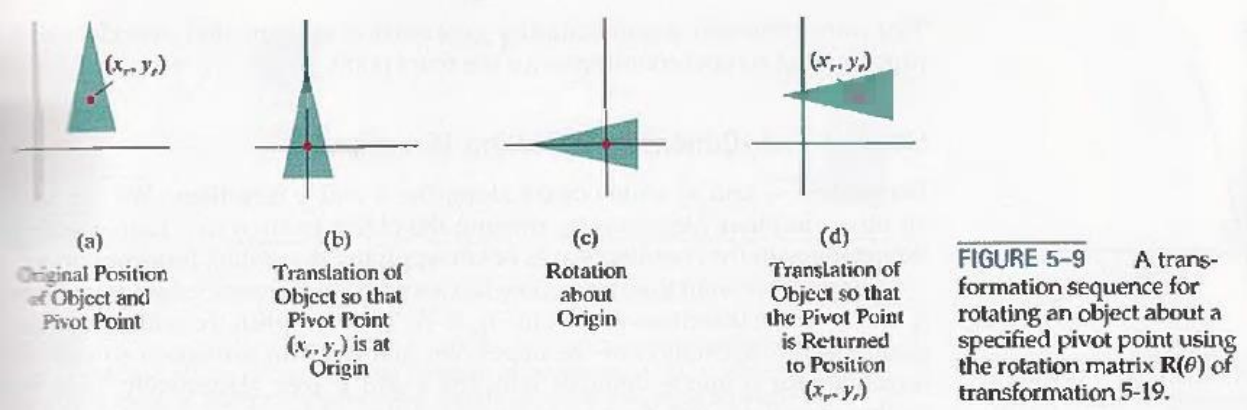



FIGURE 5–9 A transformation sequence for rotating an object about a specified pivot point using the rotation matrix $\mathbf{R}(\theta)$ of transformation 5-19.

The composite transformation matrix for this sequence is obtained with the concatenation

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

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

which can be expressed in the form

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

where $T(-x_r, -y_r) = T^{-1}(x_r, y_r)$ .

## General Two-Dimensional Fixed-Point Scaling:

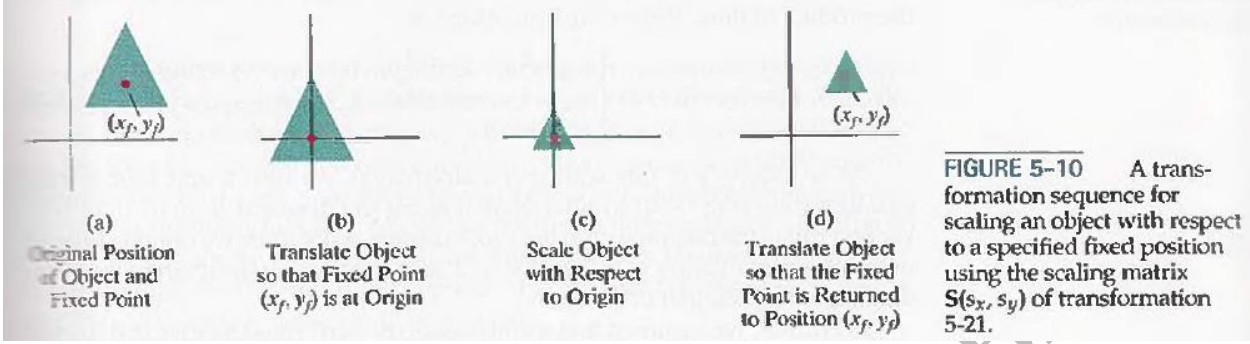



FIGURE 5-10 A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix $S(s_x, s_y)$ of transformation 5-21.

Figure 5-10 illustrates a transformation sequence to produce a two-dimensional scaling with respect to a selected fixed position ($x_f$, $y_f$), when we have a function that can scale relative to the coordinate origin only. This sequence is

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

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

Or

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

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

## General Two-Dimensional 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.

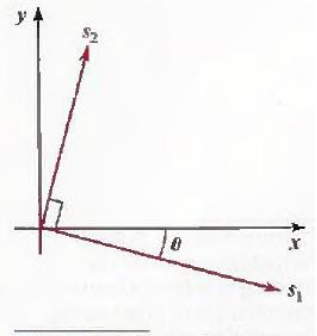



FIGURE 5-11 Scaling parameters $s_1$ and $s_2$ along orthogonal directions defined by the angular displacement $\theta$.

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 without 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 $S(s_1,s_2)$ 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

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

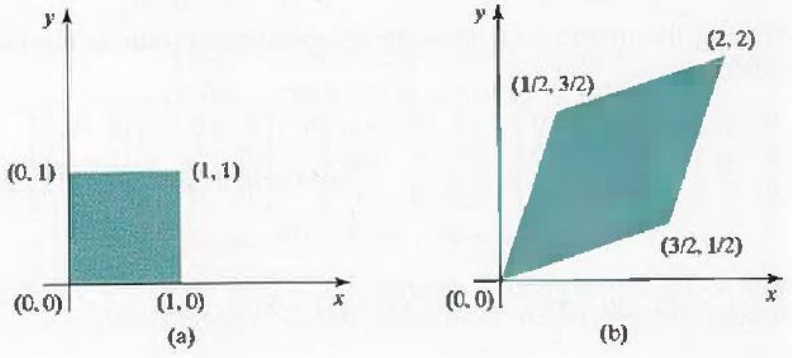



FIGURE 5-12 A square (a) is converted to a parallelogram (b) using the composite transformation matrix 5-39, with $s_1 = 1$, $s_2 = 2$, and $\theta = 45°$.

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 they axis and double its length with the transformation parameters $\theta = 45°$, $s_1 = 1$, and $s_2 = 2$, and then we rotate again to return the diagonal to its original orientation.

## Matrix Concatenation Properties:

Matrix multiplication is **associative**. For any three matrices, $\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{M_3}$, the matrix product $\mathbf{M_3}.\mathbf{M_2}.\mathbf{M_1}$ can be performed by first multiplying $\mathbf{M_3}$ and $\mathbf{M_2}$ or by first multiplying $\mathbf{M_2}$ and $\mathbf{M_1}$:

$$\mathbf{M_3}\cdot\mathbf{M_2}\cdot\mathbf{M_1}=(\mathbf{M_3}\cdot\mathbf{M_2})\cdot\mathbf{M_1}=\mathbf{M_3}\cdot(\mathbf{M_2}\cdot\mathbf{M_1})$$

Therefore, we can construct a composite matrix either by multiplying from left-to-right (premultiplying) or by multiplying from right-to-left (postmultiplying).

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 to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated (Fig. 5-13).

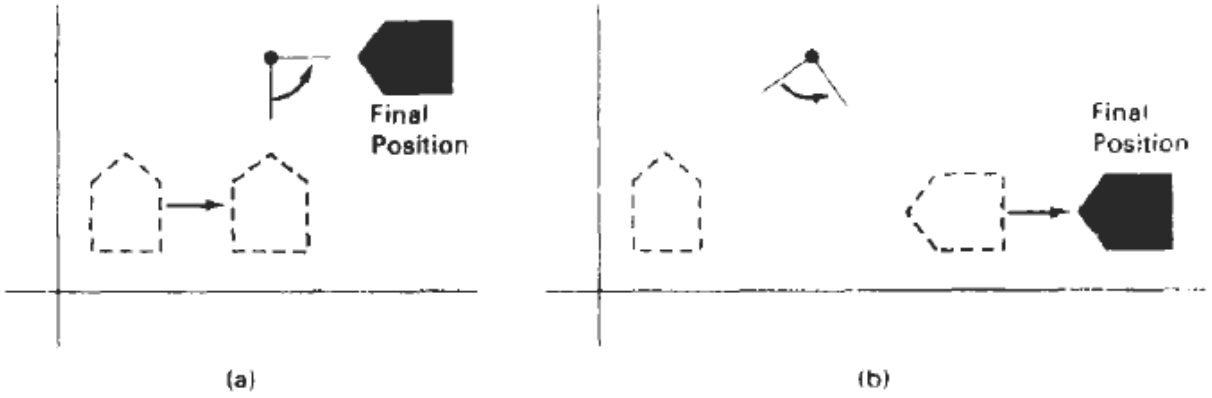



*Figure 5-13*
Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated, then rotated. In (b), the object is rotated first, then translated.

For some special cases, such as a sequence of transformations that are 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

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

Thus, we actually only need to perform four multiplications and four additions to transform coordinate positions. This is the maximum number of computation 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 the above equation.

Since rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become important consideration in rotation transformations.

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°), cosθ is approximately 1.0 and sinθ has a value very close t the value of θ in radians. If we are rotating in small angular steps about the origin, for instance, we can set cosθ to 1.0 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated. These calculations are

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

where sinθ is evaluated once for all steps, assuming the rotation angle does not change.

## Two-Dimensional Rigid-Body Transformation:

If a transformation matrix includes only translation and rotation parameters, it is a **rigid-body transformation matrix.**

The general form for a two-dimensional rigid-body transformation matrix is

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \qquad (5\text{-}45)$$

where the four elements $r_{jk}$ 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 5-45 has the property that its upper-left 2 by 2 submatrix is an **orthogonal matrix**. This means that if we consider each row (or column) of the submatrix as a

vector, then the two vectors ($r_{xx}$, $r_{xy}$) and ($r_{yx}$, $r_{yy}$) (or the two column vectors) form an orthogonal set of unit vectors. Such a set of vectors is also referred to as an orthogonal vector set.

Each vector has unit length:

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \qquad (5\text{-}46)$$

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

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0 \qquad (5\text{-}47)$$

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

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \qquad (5\text{-}48)$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \qquad (5\text{-}49)$$

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 the object.

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \qquad (5\text{-}50)$$

Here, orthogonal unit vectors in the upper-left 2 by 2 submatrix are (cos θ, -sinθ) and (sinθ, cosθ), and

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta \\ -\sin\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \qquad (5\text{-}51)$$

Similarly, unit vector (sinθ, cosθ) is converted by the preceding transformation matrix in to the unit vector (0,l) in the y direction.

## Constructing Two-Dimensional Rotation Matrices:

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 reference positions in the scene.

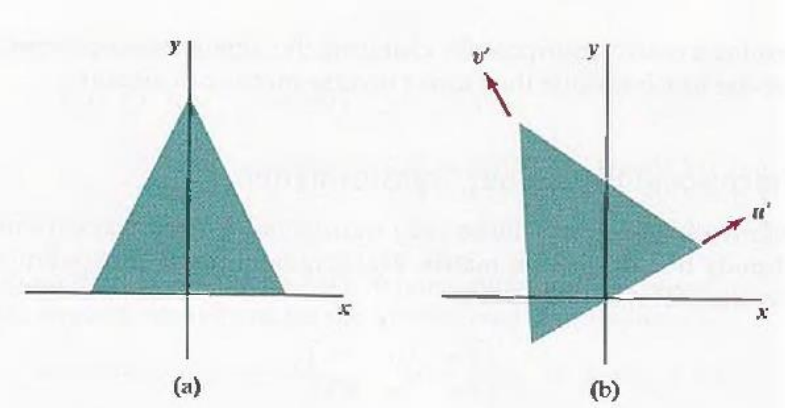



Figure 5-14 shows an object that is to be aligned with the unit direction vectors u' and v'. Assuming that the original object orientation, as shown in Fig. 5-14(a), is aligned with the coordinate axes, we construct the desired transformation by assigning the elements of u' to the first row of the rotation 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.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/*  Set initial display-window size.  */
GLsizei winWidth = 600, winHeight = 600;

/*  Set range for world coordinates.  */
GLfloat xwcMin = 0.0, xwcMax = 225.0;
GLfloat ywcMin = 0.0, ywcMax = 225.0;

class wcPt2D {
   public:
      GLfloat x, y;
};

typedef GLfloat Matrix3x3 [3][3];

Matrix3x3 matComposite;

const GLdouble pi = 3.14159;

void init (void)
{
   /*  Set color of display window to white.  */
   glClearColor (1.0, 1.0, 1.0, 0.0);
}

/* Construct the 3 by 3 identity matrix. */
void matrix3x3SetIdentity (Matrix3x3 matIdent3x3)
{
   GLint row, col;

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3; col++)
         matIdent3x3 [row][col] = (row == col);
}
```

```
/* Premultiply matrix m1 times matrix m2, store result in m2. */
void matrix3x3PreMultiply (Matrix3x3 m1, Matrix3x3 m2)
{
   GLint row, col;
   Matrix3x3 matTemp;

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3 ; col++)
         matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                              m2 [1][col] + m1 [row][2] * m2 [2][col];

   for (row = 0; row < 3; row++)
      for (col = 0; col < 3; col++)
         m2 [row][col] = matTemp [row][col];
}

void translate2D (GLfloat tx, GLfloat ty)
{
   Matrix3x3 matTransl;

   /*  Initialize translation matrix to identity.  */
   matrix3x3SetIdentity (matTransl);

   matTransl [0][2] = tx;
   matTransl [1][2] = ty;

   /*  Concatenate matTransl with the composite matrix.  */
   matrix3x3PreMultiply (matTransl, matComposite);
}
```

```
void rotate2D (wcPt2D pivotPt, GLfloat theta)
{
   Matrix3x3 matRot;

   /*  Initialize rotation matrix to identity.  */
   matrix3x3SetIdentity (matRot);

   matRot [0][0] = cos (theta);
   matRot [0][1] = -sin (theta);
   matRot [0][2] = pivotPt.x * (1 - cos (theta)) +
                   pivotPt.y * sin (theta);
   matRot [1][0] = sin (theta);
   matRot [1][1] = cos (theta);
   matRot [1][2] = pivotPt.y * (1 - cos (theta)) -
                   pivotPt.x * sin (theta);

   /*  Concatenate matRot with the composite matrix.  */
   matrix3x3PreMultiply (matRot, matComposite);
}

void scale2D (GLfloat sx, GLfloat sy, wcPt2D fixedPt)
{
   Matrix3x3 matScale;

   /*  Initialize scaling matrix to identity.  */
   matrix3x3SetIdentity (matScale);

   matScale [0][0] = sx;
   matScale [0][2] = (1 - sx) * fixedPt.x;
   matScale [1][1] = sy;
   matScale [1][2] = (1 - sy) * fixedPt.y;

   /*  Concatenate matScale with the composite matrix.  */
   matrix3x3PreMultiply (matScale, matComposite);
}

/* Using the composite matrix, calculate transformed coordinates. */
void transformVerts2D (GLint nVerts, wcPt2D * verts)
{
   GLint k;
   GLfloat temp;

   for (k = 0; k < nVerts; k++) {
      temp = matComposite [0][0] * verts [k].x + matComposite [0][1] *
             verts [k].y + matComposite [0][2];
      verts [k].y = matComposite [1][0] * verts [k].x + matComposite [1][1] *
                    verts [k].y + matComposite [1][2];
      verts [k].x = temp;
   }
}
void triangle (wcPt2D *verts)
{
   GLint k;

   glBegin (GL_TRIANGLES);
      for (k = 0; k < 3; k++)
         glVertex2f (verts [k].x, verts [k].y);
   glEnd ( );
}
```

```
void displayFcn (void)
{
   /*  Define initial position for triangle.  */
   GLint nVerts = 3;
   wcPt2D verts [3] = { {50.0, 25.0}, {150.0, 25.0}, {100.0, 100.0} };

   /*  Calculate position of triangle centroid.  */
   wcPt2D centroidPt;

   GLint k, xSum = 0, ySum = 0;
   for (k = 0; k < nVerts;  k++) {
      xSum += verts [k].x;
      ySum += verts [k].y;
   }
   centroidPt.x = GLfloat (xSum) / GLfloat (nVerts);
   centroidPt.y = GLfloat (ySum) / GLfloat (nVerts);

   /*  Set geometric transformation parameters.  */
   wcPt2D pivPt, fixedPt;
   pivPt = centroidPt;
   fixedPt = centroidPt;

   GLfloat tx = 0.0, ty = 100.0;
   GLfloat sx = 0.5, sy = 0.5;
   GLdouble theta = pi/2.0;

   glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.

   glColor3f (0.0, 0.0, 1.0);       //  Set initial fill color to blue.
   triangle (verts);                //  Display blue triangle.

   /*  Initialize composite matrix to identity.  */
   matrix3x3SetIdentity (matComposite);

   /*  Construct composite matrix for transformation sequence.  */
   scale2D (sx, sy, fixedPt);   //  First transformation: Scale.
   rotate2D (pivPt, theta);     //  Second transformation: Rotate
   translate2D (tx, ty);        //  Final transformation: Translate.

   /*  Apply composite matrix to triangle vertices.  */
   transformVerts2D (nVerts, verts);

   glColor3f (1.0, 0.0, 0.0);  // Set color for transformed triangle.
   triangle (verts);           // Display red transformed triangle.

   glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );
   gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);

   glClear (GL_COLOR_BUFFER_BIT);
}
```

```
void main (int argc, char ** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (50, 50);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Geometric Transformation Sequence");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (winReshapeFcn);

   glutMainLoop ( );
}
```

## OTHER TWO-DIMESIONAL TRANSFORMATIONS:

### Reflection:

A transformation that produces a mirror image of an object is called a **reflection**. 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.

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

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

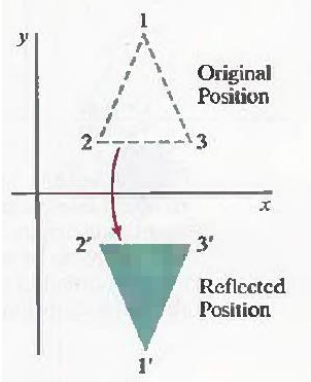



FIGURE 5-16 Reflection of an object about the *x* axis.

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

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

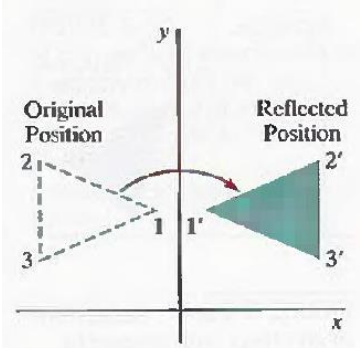



FIGURE 5-17 Reflection of an object about the *y* axis.

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:

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

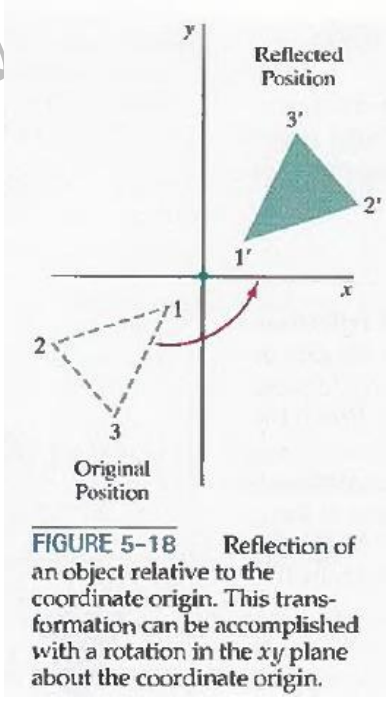



FIGURE 5-18 Reflection of an object relative to the coordinate origin. This transformation can be accomplished with a rotation in the *xy* plane about the coordinate origin.

An example of reflection about the origin is shown in Fig. 5-18. The reflection matrix is the rotation matrix R(θ) with θ = 180°. We are simply rotating the object in thc xy plane half a revolution about the origin.

The above reflection matrix can be generalized to any reflection point in the xy planeas sown in the figure below.

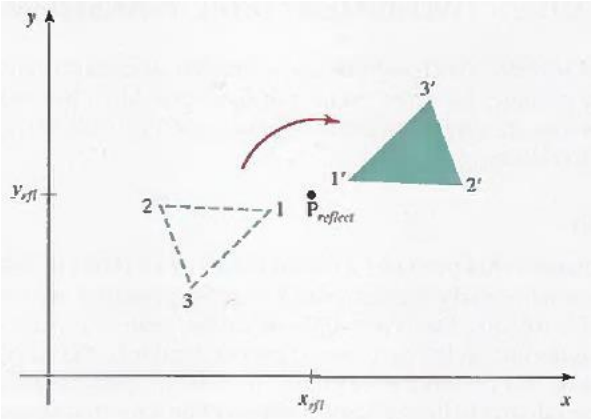



FIGURE 5-19 Reflection of an object relative to an axis perpendicular to the *xy* plane and passing through point $P_{reflect}$.

This reflection is same as a 180° rotation in the xy plane about the reflection point.

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

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

We can derive this matrix by concatenating a sequence of rotation and coordinate- axis reflection matrices.

One possible sequence is 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".

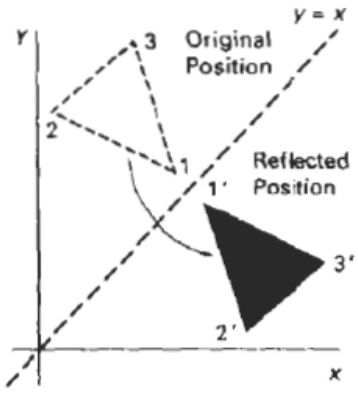



*Figure 5-20*
Reflection of an object with respect to the line $y = x$.

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

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

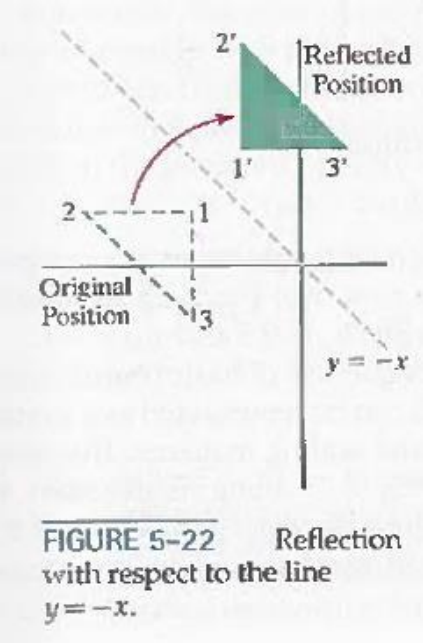



FIGURE 5-22 Reflection with respect to the line $y = -x$.

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 ±l.

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. Thus the reflected object can also be enlarged, reduced or distorted.

## 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

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

which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \qquad y' = y$$

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

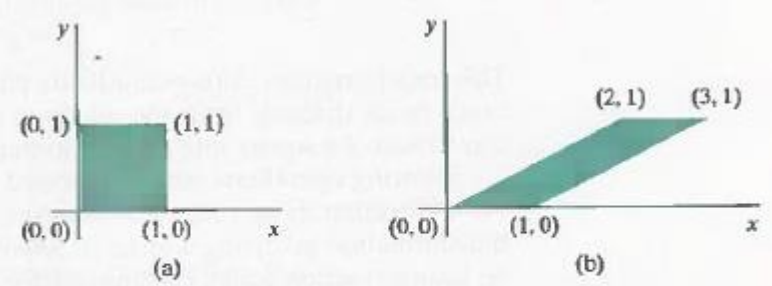



FIGURE 5-23 A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 5-57 with $sh_x = 2$.

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

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

with coordinate positions transformed as

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

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

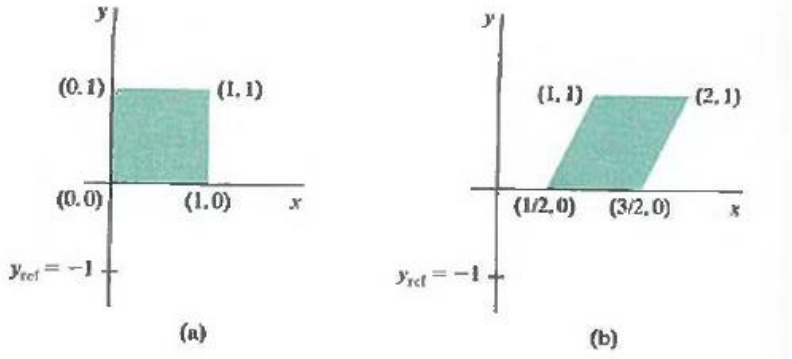



FIGURE 5-24 A unit square (a) is transformed to a shifted parallelogram (b) with $sh_x = 0.5$ and $y_{ref} = -1$ in the shear matrix 5-59.

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

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

which generates transformed coordinate positions

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

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 $\mathbf{sh_y}$ = 1/2 and $x_{ref}$ = -1.

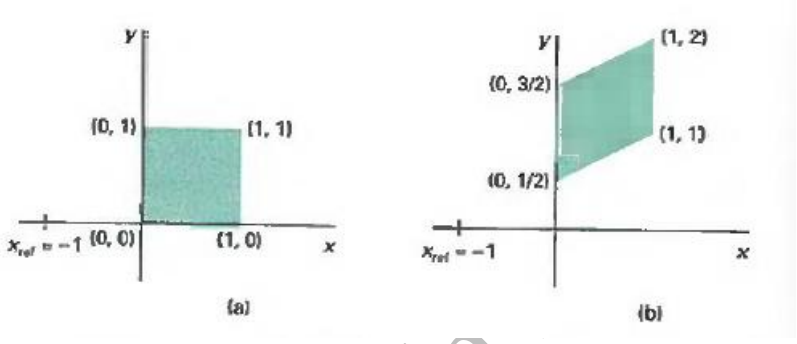
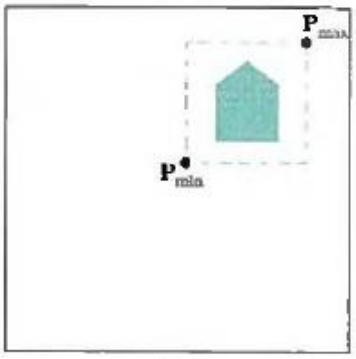



FIGURE 5-25 A unit square (a) is turned into a shifted parallelogram (b) with parameter values $sh_y = 0.5$ and $x_{ref} = -1$ in the y-direction shearing transformation 5-61.

## Raster Method for Geometric Transformation:

Raster systems store picture information as color patterns in the frame buffer. Therefore, some simple object transformations can be carried out rapidly by manipulating an array of pixel values. The pixel transformations are efficient.

The functions that manipulate the rectangular pixel arrays are called raster operations , and moving a block of pixel values from one position to another is termed as **block transfer**.

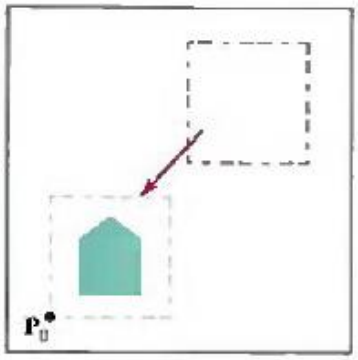



(a)

The figure on the left illustrates a two-dimensional translation implemented as a block transfer of a refresh buffer area. All bit settings in the rectangular area shown are copied as a block into another part of the frame buffer.

Rotations in 90 degree increments are easily accomplished by rearranging the elements of a pixel array. We can rotate a two-dimensional object in counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.

180 degree rotations obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows.




(b)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \quad \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \quad \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(a) (b) (c)

FIGURE 5-27 Rotating an array of pixel values. The original array is shown in (a), the positions of the array elements after a 90° counterclockwise rotation are shown in (b), and the positions of the array elements after a 180° rotation are shown in (c).

For array rotations that are not multiples of 90° , the general procedure is illustrated in the following figure:

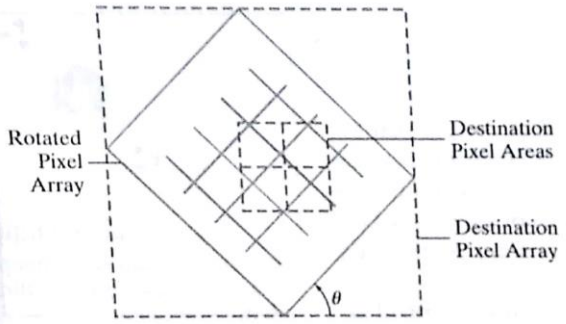



FIGURE 5-28 A raster rotation for a rectangular block of pixels can be accomplished by mapping the destination pixel areas onto the rotated block.

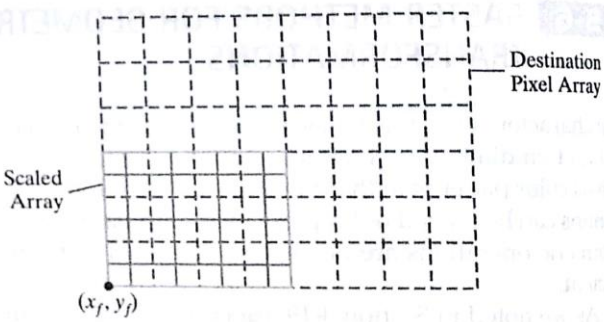



FIGURE 5-29 Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point $(x_f, y_f)$.

Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated. A color for a destination pixel can then be computed by averaging the colors of the overlapped source pixels, weighted by their percentage of area overlap.

We can use similar methods to scale a block of pixels. Pixel area in the original block are scaled, using specified values for $s_x$ and $s_y$, and then mapped onto a set of destination pixels. The color of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas.

An object can be reflected using raster transformations that reverse row or column values in a pixel block, combined with translations. Shears are produced with shifts in the positions of the array values along rows or columns.

## OpenGL Raster Transformations:

A translation of rectangular array of pixel-color values from one buffer area to another can be accomplished in OpenGL as a copy operation:

```
glReadPixels (xmin, ymin, width, height, GL_RGB,
                    GL_UNSIGNED_BYTE, colorArray);
```

The first four parameters in this function give the location and dimensions of the pixel block. The fifth parameter, GL_COLOR specifies that it is the color values that are to be copied.

This array of pixels is to be copied to a rectangular area of a refresh buffer whose lower-left corner is specified by the current raster position.
A source buffer for the **glCopyPixels** functions is chosen with the **glReadPixels** routine, and a destination buffer is selected with **glDrawPixels** routine.

```
glCopyPixels (xmin, ymin, width, height, GL_COLOR);
```

```
glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE,
                    colorArray);
```

A two –dimensional scaling transformation can be performed as a raster operation in OpenGL by specifying scaling factors and then invoking either **glCopyPixels or glDrawPixels**.

For the raster operations, we set the scaling factors with

```
glPixelZoom (sx, sy);
```

Where the parameters sx and sy can be assigned any nonzero floating-point values.

## OpenGL Geometric Transformation Functions:

### Basic OpenGL Geometric Transformations

A 4× 4 translation matrix is constructed with the following routine:

**glTranslate* (tx, ty, tz);**

Translation parameters **tx**, **ty**, and **tz** can be assigned any real-number values, and the single suffix code to be affixed to this function is either **f** (float) or **d** (double).
For two-dimensional applications, we set **tz** = 0.0; and a two-dimensional position is represented as a four-element column matrix with the *z* component equal to 0.0. The translation matrix generated by this function is used to transform positions of objects defined after this function is invoked.

For example, we translate subsequently defined coordinate positions 25 units in the *x* direction and −10

units in the *y* direction with the statement

**glTranslatef (25.0, -10.0, 0.0);**

Similarly, a 4 × 4 rotation matrix is generated with

**glRotate* (theta, vx, vy, vz);**

where the vector **v** = (**vx**, **vy**, **vz**) can have any floating-point values for its components.

This vector defines the orientation for a rotation axis that passes through the coordinate origin. If **v** is not specified as a unit vector, then it is normalized automatically before the elements of the rotation matrix are computed. The suffix code can be either **f** or **d**, and parameter **theta** is to be assigned a rotation angle in degrees, which the routine converts to radians for the trigonometric calculations.
The rotation specified here will be applied to positions defined after this function call. Rotation in two-dimensional systems is rotation about the *z* axis, specified as a unit vector with *x* and *y* components of zero, and a *z* component of 1.0. For example, the statement

**glRotatef (90.0, 0.0, 0.0, 1.0);**
sets up the matrix for a 90◦ rotation about the *z* axis. We should note here that internally, this function generates a rotation matrix using *quaternions*.

This method is more efficient when rotation is about an arbitrarily-specific axis.

We obtain a 4 × 4 scaling matrix with respect to the coordinate origin with the following routine:

**glScale* (sx, sy, sz);**

The suffix code is again either **f** or **d**, and the scaling parameters can be assigned any real-number values. Scaling in a two-dimensional system involves changes in the *x* and *y* dimensions, so a typical two-dimensional scaling operation has a *z* scaling factor of 1.0 (which causes no change in the *z* coordinate of positions).

Because the scaling parameters can be any real-number value, this function will also generate reflections when negative values are assigned to the scaling parameters.For example, the following statement produces a matrix that scales by a factor of 2 in the *x* direction, scales by a factor of 3 in the *y* direction, and reflects with respect to the *x* axis:

**glScalef (2.0, -3.0, 1.0);**

A zero value for any scaling parameter can cause a processing error because an inverse matrix cannot be calculated. The scale-reflect matrix is applied to subsequently defined objects.

It is important to note that internally OpenGL uses composite matrices to hold transformations. As a result, transformations are cumulative—that is, if we apply a translation and then apply a rotation, objects whose positions are specified after that will have both transformations applied to

them. If that is not the behavior we desired, we must be able to remove the effects of previous transformations. This requires additional functions for manipulating the composite matrices.

**OpenGL Matrix Operations:**

The **glMatrixMode** routine is used to set the *projection mode* , which designates the matrix that is to be used for the projection transformation.
This transformation determines how a scene is to be projected onto the screen. We use the same routine to set up a matrix for the geometric transformations.
In this case, however, the matrix is referred to as the *modelview matrix,* and it is used to store and combine the geometric transformations.
It is also used to combine the geometric transformations with the transformation to a viewing-coordinate system.

We specify the *modelview mode* with the statement

**glMatrixMode (GL_MODELVIEW);**

which designates the 4×4 modelview matrix as the **current matrix.**

The OpenGL transformation routines discussed in the previous section are all applied to whatever composite matrix is the current matrix, so it is important to use **glMatrixMode** to change to the modelview matrix before applying geometric transformations.

Following this call, OpenGL transformation routines are used to modify the modelview matrix, which is then applied to transform coordinate positions in a scene. Two other modes that we can set with the **glMatrixMode** function are the *texture mode* and the *color mode*.
The texture matrix is used for mapping texture patterns to surfaces, and the color matrix is used to convert from one color model to another.

The default argument for the **glMatrixMode** function is **GL MODELVIEW**.

Once we are in the modelview mode (or any other mode), a call to a transformation routine generates a matrix that is multiplied by the current matrix for that mode.
In addition, we can assign values to the elements of the current matrix, and there are two functions in the OpenGL library for this purpose.With the following function, we assign the identity matrix to the current matrix:

**glLoadIdentity ( );**

Alternatively, we can assign other values to the elements of the current matrix using

**glLoadMatrix* (elements16);**

A single-subscripted, 16-element array of floating-point values is specified with parameter **elements16**, and a suffix code of either **f** or **d** is used to designate the data type.
The elements in this array must be specified in *column-major* order. That is, we first list the four elements in the first column, and then we list the four elements in the second column, the third column, and finally the fourth column. To illustrate this ordering, we initialize the modelview matrix with the following code:

```
glMatrixMode (GL_MODELVIEW);

GLfloat elems [16];
GLint k;

for (k = 0; k < 16; k++)
    elems [k] = float (k);
glLoadMatrixf (elems);
```

which produces the matrix

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

We can also concatenate a specified matrix with the current matrix:

**glMultMatrix*(otherElements16);**

The suffix code is either f or d, and parameter otherElements16 is a 16-element , single subscripted array that lists the elements of some other matrix in column-major order.

The current matrix is postmultiplied by the matrix specified in glMultMatrix, and this product replaces the current matrix. Thus, assuming that the current matrix is the modelview matrix, which we designate as M, then the updated modelview matrix is compted as

$$\mathbf{M = M.M^1}$$

**OpenGL Matrix Stacks:**

- For each of the four modes (modelview, projection, texture and color) that we can select with the glMatrixMode function, OpenGL maintains a matrix stack.
- Initially, each stack contains only the identity matrix.
- At any time during the processing of a scene, the top matrix on each stack is called the "current matrix" for that mode.
- After we specify the viewing and geometric transformations, the top of the modelview matrix stack is the 4 by 4 composite matrix that combines the viewing transformations and the various geometric transformations that we want to apply to a scene.
- We can determine the number of positions available in the modelview stack for a particular implementation of OpenGL with

**glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);**

which returns a single integer value to array stackSize

Other OpenGL symbolic constants that can be used are:
**GL_MAX_PROJECTION_STACK_DEPTH,**

**GL_MAX_TEXTURE_STACK_DEPTH**

**GL_MAX_COLOR_STACK_DEPTH**

We can also find out how many matrices are currently in the stack with

**glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, numMats);**

We have two functions available in OpenGL for processing the matrices in a stack. These stack-processing functions are more efficient than manipulating the stack matrices individually.

With the following function, we copy the current matrix at the top of the active stack and store the copy in the second stack position.

**glPushMatrix();**

This gives us duplicate matrices at the top two positions of the stack. The other stack function is

**glPopMatrix();**

which destroys the matrix at the top of the stack, and the second matrix in the stack becomes the current matrix.

To "pop" the top of the stack, there must be at least two matrices in the stack. Otherwise, we generate an error.

**OpenGL Geometric-Transformation Programming Examples:**

In the following code segment, we apply each of the basic geometric transformations, one at a a time, to a rectangle.

```
glMatrixMode (GL_MODELVIEW);

glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);          // Display blue rectangle.

glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0); // Set translation parameters.
glRecti (50, 100, 200, 150);          // Display red, translated rectangle.

glLoadIdentity ( );                   // Reset current matrix to identity.
glRotatef (90.0, 0.0, 0.0, 1.0);   // Set 90-deg. rotation about z axis.
glRecti (50, 100, 200, 150);          // Display red, rotated rectangle.

glLoadIdentity ( );                   // Reset current matrix to identity.
glScalef (-0.5, 1.0, 1.0);            // Set scale-reflection parameters.
glRecti (50, 100, 200, 150);          // Display red, transformed rectangle.
```

- Initially, the modelview matrix is the identity matrix and we display a blue rectangle.
- Next, we reset the current color to red, specify two-dimensional translation parameters, and display the red translated rectangle.
- Since we do not want to combine the transformations, we next reset the current matrix to the identity.
- Then a rotation matrix is constructed and concatenated with the current matrix (the identity matrix).
- When the original rectangle is again referenced, it is rotated about the z-axis and displayed in a red color.
- We repeat this process once more to generate the scaled and reflected red rectangle.

Usually it is more efficient to use the stack-processing functions than to use the matrix-manipulation functions. This is particularly true when we want to make several changes in the viewing or geometric transformations. In the following code we repeat the rectangle transformations of the preceding example using stack processing instead of the glLoadIdentity function.

```
glMatrixMode (GL_MODELVIEW);

glColor3f (0.0, 0.0, 1.0);          // Set current color to blue.
glRecti (50, 100, 200, 150);        // Display blue rectangle.

glPushMatrix ( );                   // Make copy of identity (top) matrix.
glColor3f (1.0, 0.0, 0.0);          // Set current color to red.

glTranslatef (-200.0, -50.0, 0.0); // Set translation parameters.
glRecti (50, 100, 200, 150);        // Display red, translated rectangle.

glPopMatrix ( );                    // Throw away the translation matrix.
glPushMatrix ( );                   // Make copy of identity (top) matrix.

glRotatef (90.0, 0.0, 0.0, 1.0);    // Set 90-deg. rotation about z axis.
glRecti (50, 100, 200, 150);        // Display red, rotated rectangle.

glPopMatrix ( );                    // Throw away the rotation matrix.
glScalef (-0.5, 1.0, 1.0);          // Set scale-reflection parameters.
glRecti (50, 100, 200, 150);        // Display red, transformed rectangle.
```

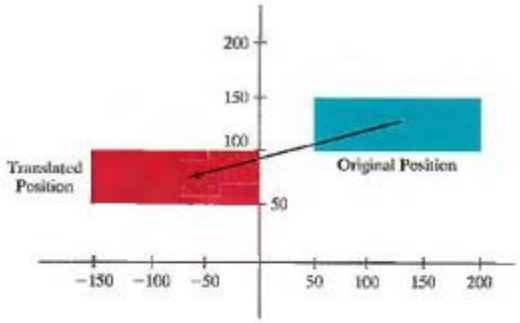



FIGURE 5-55 Translating a rectangle using the OpenGL function `glTranslatef (−200.0, −50.0, 0.0)`.

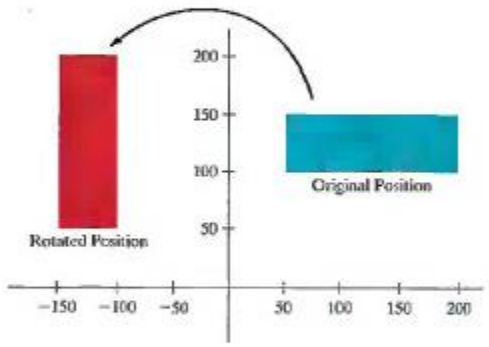



FIGURE 5-56 Rotating a rectangle about the z axis using the OpenGL function `glRotatef (90.0, 0.0, 0.0, 1.0)`.

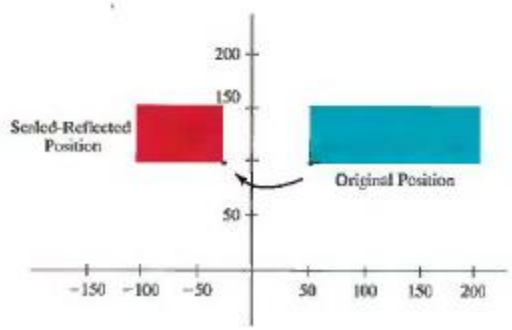
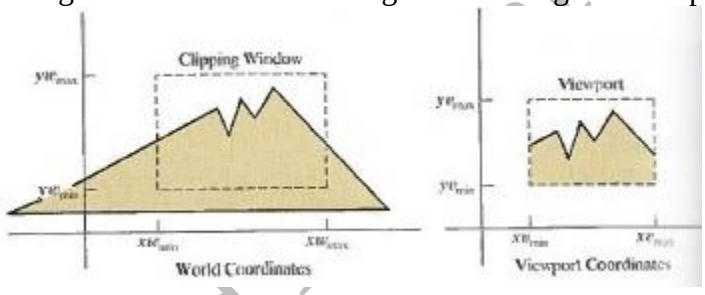



FIGURE 5-57 Scaling and reflecting a rectangle using the OpenGL function `glScalef (−0.5, 1.0, 1.0)`.

## The Two-Dimensional Viewing Pipeline:

- A section of a two-dimensional scene that is selected for display is called a **clipping window.**
- An area on a display device to which a window is mapped is called a **viewport**.
- The clipping window selects **what** we want to see; the viewport defines where it is to be viewed on the output device.
- By changing the position of a viewport , we can view objects at different positions on the display area of an output device.
- Multiple viewports can be used to display different sections of a scene at different screen positions. Also by varying the size of viewports, we can change the size and proportions of displayed objects.
- Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
- 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.
- The following Figure illustrates the mapping of a picture section that falls within a rectangular window onto a designated rectangular viewport.




- In general viewing involves more than just the transformation from clipping-wndow coordinates to viewport coordinates.

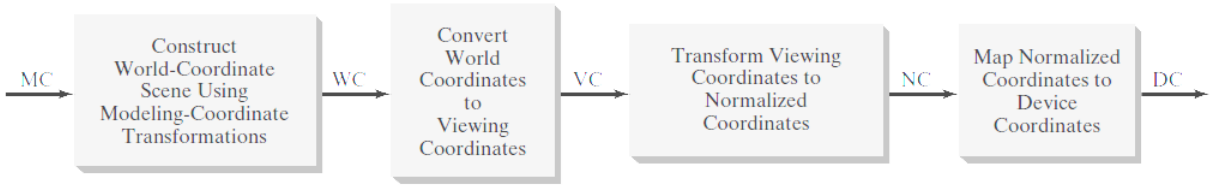



FIGURE 2
Two-dimensional viewing-transformation pipeline.

- Once a world coordinate scene has been constructed, we could set up a separate two-dimensional, viewing coordinate reference frame for specifying the clipping window.

- To make the viewing process independent of the requirements of any output device, graphics systems convert object descriptions to normalized coordinates and apply the clipping routines.
- Depending upon the graphics library in use, the viewport is defined either in normalized coordinates or in screen coordinates after the normalization process.
- At the final step of the viewing transformation, the contents of the viewport are transferred to positions within the display window.
- Clipping is usually performed in normalized coordinates. This allows us to reduce computations by first concatenating the various transformation matrices.

**Viewing Coordinate Clipping Window:**

- A general approach to the two-dimensional viewing transformation is to set up a viewing-coordinate system within the world-coordinate frame.
- This viewing frame provides a reference for specifying a rectangular clipping window with any selected orientation and position as shown in the following figure.

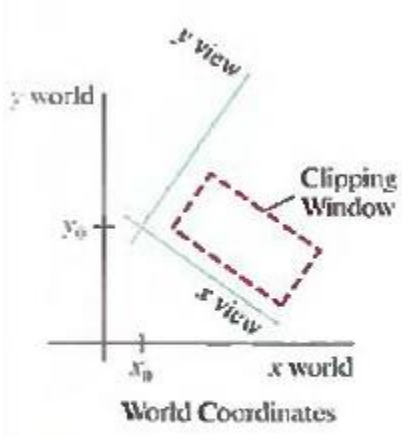



FIGURE 6–4 A rotated clipping window defined in viewing coordinates.

- First, a viewing-coordinate origin is selected at some world position: $P_o = (x_o, y_o)$.
- Then we need to establish the orientation, or rotation, of this reference frame. One way to do this is to specify a world vector V that defines the viewing $y_{view}$ direction. Vector V is called **the view up vector.**
- Given the orientation vector **V**, we can calculate the components of unit vectors $v = (v_x, v_y)$ and $u = (u_x, u_y)$ for the viewing $y_{view}$ and $x_{view}$ axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix R that aligns the viewing $x_{view}y_{view}$ axes with the world $x_wy_w$ axes.
- The composite two-dimensional transformation to convert world coordinates to viewing coordinate is

$$\mathbf{M_{WC,VC} = R.T}$$

where T is the translation matrix that takes the viewing origin point Po to the world origin, and R is the rotation matrix that rotates the viewing frame of reference into coincidence with the world-coordinate system.

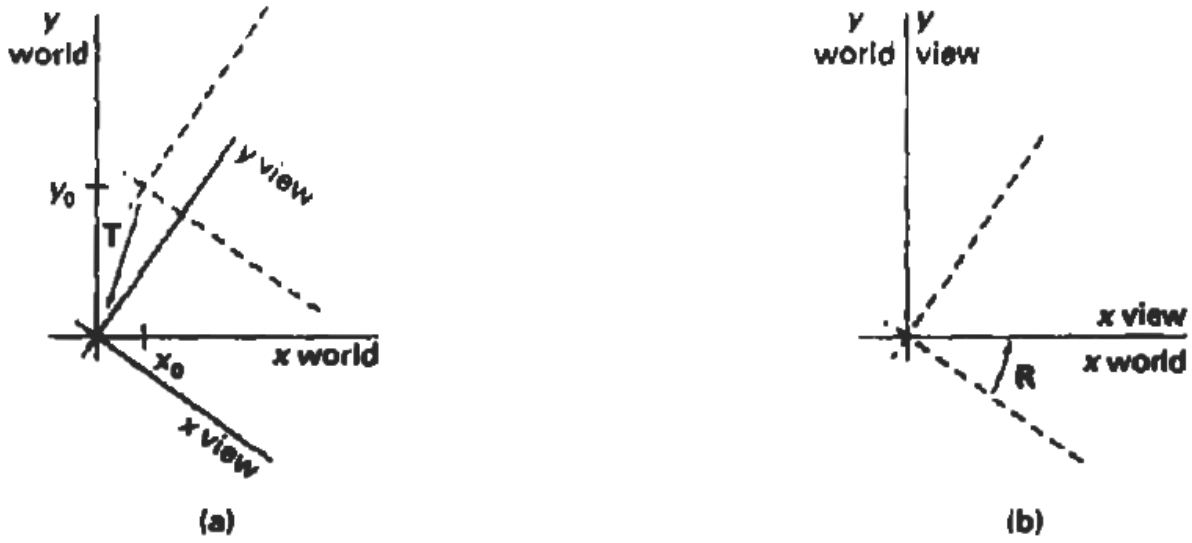



***Figure 6-4***
**A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.**

**World Coordinate Clipping Window:**

- A routine for defining a standard, rectangular clipping window in world coordinates is provided in a graphics-programming library.
- We simply specify two world-coordinate positions, which are then assigned to the two opposite corners of a standard rectangle.
- Once the clipping window has been established, the scene description is processed through the viewing routines to the output device.

## OpenGL Two-Dimensional Viewing Function:

Actually, the basic OpenGL library has no functions specifically for two-dimensional viewing because it is designed primarily for three-dimensional applications. But we can adapt the three-dimensional viewing routines to a two-dimensional scene, and the core library contains a viewport function.
In addition, the GLU library provides a function for specifying a two-dimensional clipping window, and we have GLUT library functions for handling display windows. Therefore, we can use these two-dimensional routines, along with the OpenGL viewport function, for all the viewing operations we need.

### OpenGL Projection Mode

Before we select a clipping window and a viewport in OpenGL, we need to establish the appropriate mode for constructing the matrix to transform from world coordinates to screen coordinates.
With OpenGL, we cannot set up a separate two-dimensional viewing-coordinate system and we must set the parameters for the clipping window as part of the projection transformation. Therefore, we must first select the projection mode. We do this with the same function we used to set the modelview mode for the geometric transformations.
Subsequent commands for defining a clipping window and viewport will then be applied to the projection matrix.

```
glMatrixMode (GL_PROJECTION);
```

This designates the projection matrix as the current matrix, which is originally set to the identity matrix. However, if we are going to loop back through this statement for other views of a scene, we can also set the initialization as

```
glLoadIdentity ( );
```

This ensures that each time we enter the projection mode, the matrix will be reset to the identity matrix so that the new viewing parameters are not combined with the previous ones.

### GLU Clipping-Window Function

To define a two-dimensional clipping window, we can use the GLU function:

```
gluOrtho2D (xwmin, xwmax, ywmin, ywmax);
```

Coordinate positions for the clipping-window boundaries are given as doubleprecision numbers.
This function specifies an orthogonal projection for mapping the scene to the screen. For a three-dimensional scene, this means that objects would be projected along parallel lines that are perpendicular to the twodimensional *xy* display screen.

But for a two-dimensional application, objects are already defined in the *xy* plane. Therefore, the orthogonal projection has no effect on our two-dimensional scene other than to convert object positions to normalized coordinates.

We must specify the orthogonal projection because our two-dimensional scene is processed through the full three dimensional OpenGL viewing pipeline. In fact, we could specify the clipping window using the three-dimensional OpenGL core-library version of the **gluOrtho2D** function.

Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping routines. And the **gluOrtho2D** function sets up a three-dimensional version of transformation matrix 9 for mapping objects within the clipping window to normalized coordinates. Objects outside the normalized square (and outside the clipping window) are eliminated from the scene to be displayed.

If we do not specify a clipping window in an application program, the default coordinates are

(*xw*min, *yw*min) = (−1.0, −1.0) and (*xw*max, *yw*max) = (1.0, 1.0). Thus the default clipping window is the normalized square centered on the coordinate origin with a side length of 2.0.

**OpenGL Viewport Function**

We specify the viewport parameters with the OpenGL function

**glViewport (xvmin, yvmin, vpWidth, vpHeight);**

where all parameter values are given in integer screen coordinates relative to the display window.
Parameters **xvmin** and **yvmin** specify the position of the lowerleft corner of the viewport relative to the lower-left corner of the display window, and the pixel width and height of the viewport are set with parameters **vpWidth**
and **vpHeight.**
If we do not invoke the **glViewport** function in a program, the default viewport size and position are the same as the size and position of the display window.
After the clipping routines have been applied, positions within the normalized square are transformed into the viewport rectangle.

Coordinates for the upper-right corner of the viewport are calculated for this transformation matrix in terms of the viewport width and height:

***xv*max = *xv*min + *vpWidth*, *yv*max = *yv*min + *vpHeight***

For the final transformation, pixel colors for the primitives within the viewport are loaded into the refresh buffer at the specified screen locations.
Multiple viewports can be created in OpenGL for a variety of applications.
We can obtain the parameters for the currently active viewport using the query function

**glGetIntegerv (GL_VIEWPORT, vpArray);**

where **vpArray** is a single-subscript, four-element array. This **Get** function returns the parameters for the current viewport to **vpArray** in the order **xvmin**, **yvmin**, **vpWidth**, and **vpHeight.**
In an interactive application, for example, we can use this function to obtain parameters for the viewport that contains the screen cursor.

## Creating a GLUT Display Window

Because the GLUT library interfaces with any window-management system, we use the GLUT routines for creating and manipulating display windows so that our example programs will be independent of any specific machine.
To access these routines, we first need to initialize GLUT with the following function:

**glutInit (&argc, argv);**

Parameters for this initialization function are the same as those for the **main** procedure, and we can use **glutInit** to process command-line arguments.
We have three functions inGLUTfor defining a display window and choosing its dimensions and position:

**glutInitWindowPosition (xTopLeft, yTopLeft);**
**glutInitWindowSize (dwWidth, dwHeight);**
**glutCreateWindow ("Title of Display Window");**

- The first of these functions gives the integer, screen-coordinate position for the top-left corner of the display window, relative to the top-left corner of the screen.
- If either coordinate is negative, the display-window position on the screen is determined by the window-management system.
- With the second function, we choose a width and height for the display window in positive integer pixel dimensions. If we do not use these two functions to specify a size and position, the default size is 300 by 300 and the default position is *(*−1, −1*)*, which leaves the positioning of the display window to the window-management system.
- In any case, the display-window size and position specified with GLUT routines might be ignored, depending on the state of the window-management system or the other requirements currently in effect for it. Thus, the window system might position and size the display window differently.
- The third function creates the display window, with the specified size and position, and assigns a title, although the use of the title also depends on the windowing system. At this point, the display window is defined but not shown on the screen until all the GLUT setup operations are complete.

## Setting the GLUT Display-Window Mode and Color

Various display-window parameters are selected with the GLUT function

**glutInitDisplayMode (mode);**

We use this function to choose a color mode (RGB or index) and different buffer combinations, and the selected parameters are combined with the logical **or** operation.
The default mode is single buffering and the RGB (or RGBA) color mode, which is the same as setting this mode with the statement

**glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);**

The color mode specification **GLUT RGB** is equivalent to **GLUT RGBA.** A background color for the display window is chosen in RGB mode with the OpenGL routine

**glClearColor (red, green, blue, alpha);**

In color-index mode, we set the display-window color with

**glClearIndex (index);**

where parameter **index** is assigned an integer value corresponding to a position within the color table.

**GLUT Display-Window Identifier**

Multiple display windows can be created for an application, and each is assigned a positive-integer **display-window identifier,** starting with the value 1 for thefirst window that is created. At the time that we initiate a display window, we can record its identifier with the statement

**windowID = glutCreateWindow ("A Display Window");**

Once we have saved the integer display-window identifier in variable name **windowID**, we can use the identifier number to change display parameters or to delete the display window.

**Deleting a GLUT Display Window**

The GLUT library also includes a function for deleting a display window that we have created. If we know the display window's identifier, we can eliminate it with the statement

**glutDestroyWindow (windowID);**

**Current GLUT Display Window**

When we specify any display-window operation, it is applied to the **current display window,** which is either the last display window that we created or the one we select with the following command:

**glutSetWindow (windowID);**

In addition, at any time, we can query the system to determine which window is the current display window:

**currentWindowID = glutGetWindow ( );**

A value of 0 is returned by this function if there are no display windows or if the current display window was destroyed.

**Relocating and Resizing a GLUT Display Window**

We can reset the screen location for the current display window with

**glutPositionWindow (xNewTopLeft, yNewTopLeft);**

where the coordinates specify the new position for the upper-left display-window corner, relative to the upper-left corner of the screen.
Similarly, the following function resets the size of the current display window:

**glutReshapeWindow (dwNewWidth, dwNewHeight);**

With the following command, we can expand the current display window to fill the screen:

**glutFullScreen ( );**

The exact size of the display window after execution of this routine depends on the window-management system. A subsequent call to either

**glutPositionWindow** or **glutReshapeWindow** will cancel the request for an expansion to full-screen size.

Whenever the size of a display window is changed, its aspect ratio may

**glutReshapeFunc (winReshapeFcn);**

This GLUT routine is activated when the size of a display window is changed, and the new width and height are passed to its argument: the function **winReshapeFcn**, in this example. Thus, **winReshapeFcn** is the "callback function" for the "reshape event."

We can then use this callback function to change the parameters for the viewport so that the original aspect ratio of the scene is maintained.

In addition, we could also reset the clipping-window boundaries, change the display-window color, adjust other viewing parameters, and perform any other tasks.

**Managing Multiple GLUT Display Windows**

The GLUT library also has a number of routines for manipulating a display window in various ways.

These routines are particularly useful when we have multiple display windows on the screen and we want to rearrange them or locate a particular display window.

We use the following routine to convert the current display window to an icon in the form of a small picture or symbol representing the window:

**glutIconifyWindow ( );**

The label on this icon will be the same name that we assigned to the window, but we can change this with the following command:

**glutSetIconTitle ("Icon Name");**

change and objects may be distorted from their original shapes. We can adjust for a change in display-window dimensions using the statement

We also can change the name of the display window with a similar command:

**glutSetWindowTitle ("New Window Name");**

With multiple display windows open on the screen, some windows may overlap or totally obscure other display windows.

We can choose any display window to be in front of all other windows by first designating it as the current window, and then issuing the "pop-window" command:

**glutSetWindow (windowID);**
**glutPopWindow ( );**

In a similar way, we can "push" the current display window to the back so that it is behind all other display windows. This sequence of operations is

**glutSetWindow (windowID);**
**glutPushWindow ( );**

We can also take the current window off the screen with

**glutHideWindow ( );**

In addition, we can return a "hidden" display window, or one that has been converted to an icon, by designating it as the current display window and then invoking the function
**glutShowWindow ( );**

**GLUT Subwindows**

Within a selected display window, we can set up any number of second-level display windows, which are called *subwindows.*
This provides a means for partitioning display windows into different display sections.We create a subwindow
with the following function:

**glutCreateSubWindow (windowID, xBottomLeft, yBottomLeft, width, height);**

Parameter **windowID** identifies the display window in which we want to set up the subwindow.With the remaining parameters, we specify the subwindow's size and the placement of its lower-left corner relative to the lower-left corner of the display window.
Subwindows are assigned a positive integer identifier in the same way that first-level display windows are numbered, and we can place a subwindow inside another subwindow. Also, each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows, just as we can with first-level display windows. But we cannot convert a GLUT subwindow to an icon.

**Selecting a Display-Window Screen-Cursor Shape**

We can use the following GLUT routine to request a shape for the screen cursor that is to be used with the current window:

**glutSetCursor (shape);**

The possible cursor shapes that we can select are an arrow pointing in a chosen direction, a bidirectional arrow, a rotating arrow, a crosshair, a wristwatch, a question mark, or even a skull and crossbones.
For example, we can assign the symbolic constant **GLUT CURSOR UP DOWN** to parameter **shape** to obtain an

up-down arrow. A rotating arrow is chosen with **GLUT CURSOR CYCLE**, a wristwatch shape is selected with **GLUT CURSOR WAIT**, and a skull and crossbones is obtained with the constant **GLUT CURSOR DESTROY.**

A cursor shape can be assigned to a display window to indicate a particular kind of application, such as an animation. However, the exact shapes that we can use are system dependent.

**Viewing Graphics Objects in a GLUT Display Window**

After we have created a display window and selected its position, size, color, and other characteristics, we indicate what is to be shown in that window.

If more than one display window has been created, we first designate the one we want as the current display window. Then we invoke the following function to assign something to that window:

**glutDisplayFunc (pictureDescrip);**

The argument is a routine that describes what is to be displayed in the current window. This routine, called **pictureDescrip** for this example, is referred to as a *callback function* because it is the routine that is to be executed whenever GLUT determines that the display-window contents should be renewed.

Routine **pictureDescrip** usually contains the OpenGL primitives and attributes that define a picture, although it could specify other constructs such as a menu display.

If we have set up multiple display windows, then we repeat this process for each of the display windows or subwindows. Also, we may need to call **glutDisplayFunc** after the **glutPopWindow** command if the display window has been damaged during the process of redisplaying the windows. In this case, the following function is used to indicate that the contents of the current display window should be renewed:

**glutPostRedisplay ( );**

This routine is also used when an additional object, such as a pop-up menu, is to be shown in a display window.

**Executing the Application Program**

When the program setup is complete and the display windows have been created and initialized, we need to issue the final GLUT command that signals execution of the program:

**glutMainLoop ( );**

At this time, display windows and their graphic contents are sent to the screen. The program also enters the **GLUT processing loop** that continually checks for new "events," such as interactive input from a mouse or a graphics tablet.

**Other GLUT Functions**

Sometimes it is convenient to designate a function that is to be executed when there are no other events for the system to process. We can do that with

**glutIdleFunc (function);**
The parameter for this GLUT routine could reference a background function or a procedure to update parameters for an animation when no other processes are taking place.
Finally, we can use the following function to query the system about some of the current state parameters:

**glutGet (stateParam);**
This function returns an integer value corresponding to the symbolic constant we select for its argument. For example, we can obtain the *x*-coordinate position for the top-left corner of the current display window, relative to the top-left corner of the screen, with the constant **GLUT WINDOW X;** and we can retrieve the current display-window width or the screen width with **GLUT WINDOW WIDTH** or **GLUT SCREEN WIDTH.**

**OpenGL Two-Dimensional Viewing Program Example**
As a demonstration of the use of the OpenGL viewport function, we use a splitscreen effect to show two views of a triangle in the *xy* plane with its centroid at the world-coordinate origin.
First, a viewport is defined in the left half of the display window, and the original triangle is displayed there in blue. Using the same clipping window, we then define another viewport for the right half of the display window, and the fill color is changed to red.
The triangle is then rotated about its centroid and displayed in the second viewport.

```
#include <GL/glut.h>
class wcPt2D {
public:
GLfloat x, y;
};

void init (void)
{
        /* Set color of display window to white. */
        glClearColor (1.0, 1.0, 1.0, 0.0);
        /* Set parameters for world-coordinate clipping window. */
        glMatrixMode (GL_PROJECTION);
        gluOrtho2D (-100.0, 100.0, -100.0, 100.0);
        /* Set mode for constructing geometric transformation matrix. */
        glMatrixMode (GL_MODELVIEW);
}
void triangle (wcPt2D *verts)
{
        GLint k;
        glBegin (GL_TRIANGLES);
        for (k = 0; k < 3; k++)
        glVertex2f (verts [k].x, verts [k].y);
        glEnd ( );
}
void displayFcn (void)
```

```
{
        /* Define initial position for triangle. */
        wcPt2D verts [3] = { {-50.0, -25.0}, {50.0, -25.0}, {0.0, 50.0} };
        glClear (GL_COLOR_BUFFER_BIT); // Clear display window.
        glColor3f (0.0, 0.0, 1.0); // Set fill color to blue.
        glViewport (0, 0, 300, 300); // Set left viewport.
        triangle (verts); // Display triangle.
        /* Rotate triangle and display in right half of display window. */
        glColor3f (1.0, 0.0, 0.0); // Set fill color to red.
        glViewport (300, 0, 300, 300); // Set right viewport.
        glRotatef (90.0, 0.0, 0.0, 1.0); // Rotate about z axis.
        triangle (verts); // Display red rotated triangle.
        glFlush ( );
}
void main (int argc, char ** argv)
{
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition (50, 50);
        glutInitWindowSize (600, 300);
        glutCreateWindow ("Split-Screen Example");
        init ( );
        glutDisplayFunc (displayFcn);
        glutMainLoop ( );
}
```