MODULE 2: FILL AREA PRIMITIVES, 2D GEOMETRIC TRANSFORMATIONS AND 3D VIEWING.

Fill area Primitives:

Introduction

Polygon fill-areas,

OpenGL polygon Fill Area Functions,

Fill area attributes,

General scan line polygon fill algorithm,

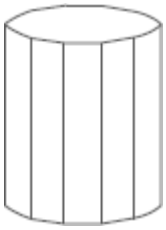
OpenGL fill-area Attribute functions.

Introduction

- An useful construct for describing components of a picture is an area that is filled with some solid color or pattern.
- A picture component of this type is typically referred to as a **fill area or a filled area**.
- Any fill-area shape is possible, graphics libraries generally do not support specifications for arbitrary fill shapes
- Figure below illustrates a few possible fill-area shapes.

- Graphics routines can more efficiently process polygons than other kinds of fill shapes because polygon boundaries are described with linear equations.
- When lighting effects and surface-shading procedures are applied, an approximated curved surface can be displayed quite realistically.
- Approximating a curved surface with polygon facets is sometimes referred to as ***surface tessellation,*** **or fitting the surface with a** ***polygon mesh.***

- Below figure shows the side and top surfaces of a metal cylinder approximated in an outline form as a polygon mesh.



- Displays of such figures can be generated quickly as *wire-frame* views, showing only the polygon edges to give a general indication of the surface structure
- Objects described with a set of polygon surface patches are usually referred to as standard graphics objects, or just graphics objects.

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.
- ✓ It is required that the polygon edges have no common point other than their endpoints.
- ✓ Thus, 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
- ✓ Any plane figure with a closed-polyline boundary is alluded to as a polygon, and one with no crossing edges is referred to as a *standard polygon* or a *simple polygon*

Problem:

- ➢ For a computer-graphics application, it is possible that a designated set of polygon vertices do not all lie exactly in one plane
- ➢ This is due to roundoff error in the calculation of numerical values, to errors in selecting coordinate positions for the vertices, or, more typically, to approximating a curved surface with a set of polygonal patches

Solution:

- ➢ To divide the specified surface mesh into triangles

Polygon Classifications

- ✓ Polygons are classified into two types
  1. Convex Polygon and
  2. Concave Polygon

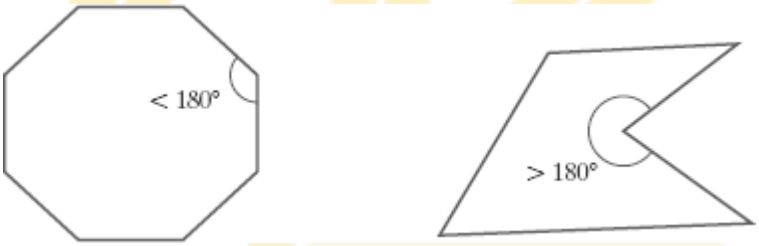
Convex Polygon:

- ✓ The polygon is convex if all interior angles of a polygon are less than or equal to 180◦, where an interior angle of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges
- ✓ 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.

Concave Polygon:

- ✓ A polygon that is not convex is called a concave polygon.

Te below figure shows convex and concave polygon



- ✓ The term degenerate polygon is often used to describe a set of vertices that are collinear or that have repeated coordinate positions.

Problems in concave polygon:

➔ Implementations of fill algorithms and other graphics routines are more complicated

Solution:

➔ It is generally more efficient to split a concave polygon into a set of convex polygons before processing

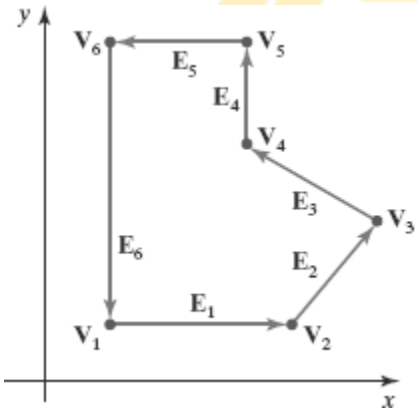
Identifying Concave Polygons

Characteristics:

- A concave polygon has at least one interior angle greater than 180◦.
- 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

Identification algorithm 1

- Identifying a concave polygon by calculating cross-products of successive pairs of edge vectors.
- 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-products yield a positive value and some a negative value, we have a concave polygon



$(\mathbf{E}_1 \times \mathbf{E}_2)_z > 0$
$(\mathbf{E}_2 \times \mathbf{E}_3)_z > 0$
$(\mathbf{E}_3 \times \mathbf{E}_4)_z < 0$
$(\mathbf{E}_4 \times \mathbf{E}_5)_z > 0$
$(\mathbf{E}_5 \times \mathbf{E}_6)_z > 0$
$(\mathbf{E}_6 \times \mathbf{E}_1)_z > 0$

Identification algorithm 2

- 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

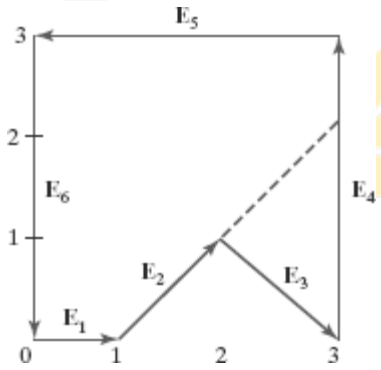
- ✓ Split concave polygon it into a set of convex polygons using edge vectors and edge cross-products; or, we can use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other.

Vector method

- ➔ First need to form the edge vectors.
- ➔ Given two consecutive vertex positions, V_k and V_{k+1}, we define the edge vector between them as

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

- ➔ 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 a negative *z* component, the polygon is concave.
- ➔ We can apply the vector method by processing edge vectors in counterclockwise order If any cross-product has a negative *z* component (as in below figure), the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair



E1 = *(*1, 0, 0*)* E2 = *(*1, 1, 0*)*
E3 = *(*1, −1, 0*)* E4 = *(*0, 2, 0*)*
E5 = *(*−3, 0, 0*)* E6 = *(*0, −2, 0*)*

- ➔ Where the *z* component is 0, since all edges are in the *xy* plane.
- ➔ The crossproduct $E_j \times E_k$ for two successive edge vectors is a vector perpendicular to the *xy* plane with *z* component equal to $E_{jx}E_{ky} - E_{kx}E_{jy}$:
- ➔ The values for the above figure is as follows

E1 × E2 = *(*0, 0, 1*)* E2 × E3 = *(*0, 0, −2*)*
E3 × E4 = *(*0, 0, 2*)* E4 × E5 = *(*0, 0, 6*)*

$E5 \times E6 = (0, 0, 6)$ $E6 \times E1 = (0, 0, 2)$

➔ Since the cross-product E2 × E3 has a negative *z* component, we split the polygon along the line of vector E2.

➔ The line equation for this edge has a slope of 1 and a *y* intercept of −1 . No other edge cross-products are negative, so the two new polygons are both convex.

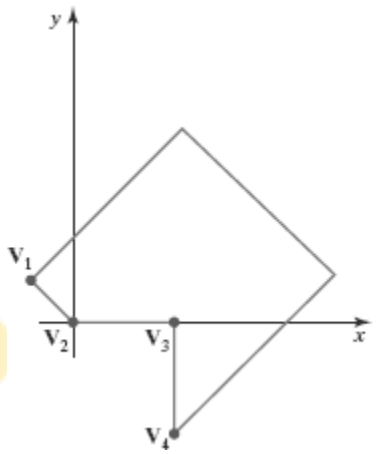
Rotational method

➔ Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex V_k in turn is at the coordinate origin.

➔ We rotate the polygon about the origin in a clockwise direction so that the next vertex V_{k+1} is on the *x* axis.

➔ If the following vertex, V_{k+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



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.
- First define 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 .
- 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.
- Concave polygon can also be divided into a set of triangles using this approach, although care must be taken that the new diagonal edge formed by joining the first and third selected vertices does not cross the concave portion of the polygon, and that the three selected vertices at each step form an interior angle that is less than 180◦

Identifying interior and exterior region of polygon

- We may want to specify a complex fill region with intersecting edges.
- For such shapes, it is not always clear which regions of the *xy* plane we should call "interior" and which regions.
- We should designate as "exterior" to the object boundaries.
- Two commonly used algorithms
    1. Odd-Even rule and
    2. The nonzero winding-number rule.

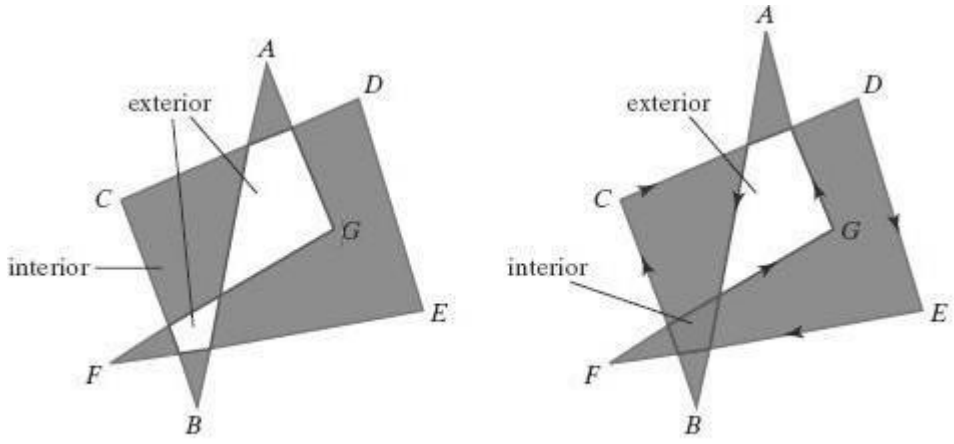
Inside-Outside Tests

- ✓ Also called the *odd-parity rule* or the *even-odd rule.*
- ✓ Draw a line from any position P to a distant point outside the coordinate extents of the closed polyline.
- ✓ Then we count the number of line-segment crossings along this line.
- ✓ If the number of segments crossed by this line is odd, then P is considered to be an *interior* point Otherwise, P is an *exterior* point
- ✓ We can use this procedure, for example,to fill the interior region between two concentric circles or two concentric polygons with a specified color.

Nonzero Winding-Number rule

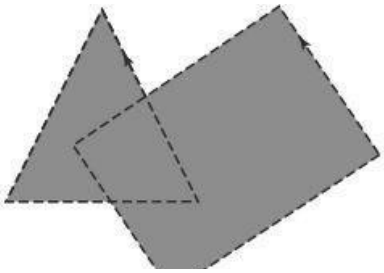
- ✓ This counts the number of times that the boundary of an object "winds" around a particular point in the counterclockwise direction termed as winding number,
- ✓ Initialize the winding number to 0 and again imagining a line drawn from any position P to a distant point beyond the coordinate extents of the object.
- ✓ The line we choose must not pass through any endpoint coordinates.
- ✓ As we move along the line from position P to the distant point, we count the number of object line segments that cross the reference line in each direction
- ✓ We add 1 to the winding number every time we intersect a segment that crosses the line in the direction from right to left, and we subtract 1 very time we intersect a segment that crosses from left to right

- ✓ If the winding number is nonzero, P is considered to be an interior point. Otherwise, P is taken to be an exterior point

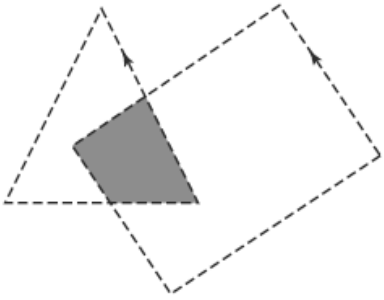


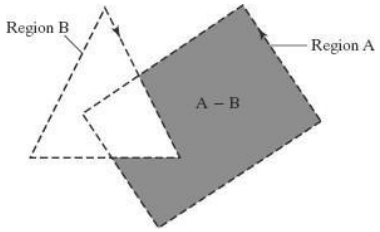
Odd-Even Rule Nonzero Winding-Number Rule

- ✓ The nonzero winding-number rule tends to classify as interior some areas that the odd-even rule deems to be exterior.
- ✓ Variations of the nonzero winding-number rule can be used to define interior regions in other ways define a point to be interior if its winding number is positive or if it is negative; or we could use any other rule to generate a variety of fill shapes
- ✓ Boolean operations are used to specify a fill area as a combination of two regions
- ✓ One way to implement Boolean operations is by using a variation of the basic winding-number rule consider the direction for each boundary to be counterclockwise, the union of two regions would consist of those points whose winding number is positive



- ✓ The intersection of two regions with counterclockwise boundaries would contain those points whose winding number is greater than 1,

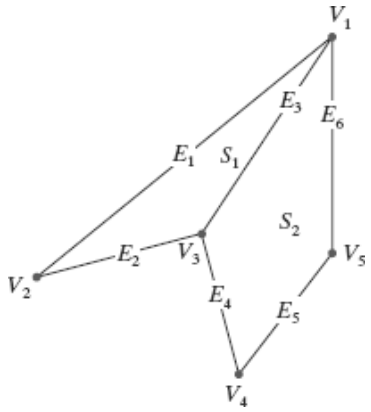




- To set up a fill area that is the difference of two regions (say, A − B), we can enclose region A with a counterclockwise border and B with a clockwise border

Polygon Tables

- ✓ The objects in a scene are described as sets of polygon surface facets
- ✓ 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.
- ✓ The data of the polygons 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:
  1. Geometric tables and
  2. 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
- ✓ 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



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

- ✓ The object can be displayed efficiently by using data from the edge table to identify polygon boundaries.
- ✓ An alternative arrangement is to use just two tables: a vertex table and a surface-facet table this scheme is less convenient, and some edges could get drawn twice in a wire-frame display.
- ✓ Another possibility is to use only a surface-facet table, but this duplicates coordinate information, since explicit coordinate values are listed for each vertex in each polygon facet. Also the relationship between edges and facets would have to be reconstructed from the vertex listings in the surface-facet table.
- ✓ We could expand the edge table to include forward pointers into the surface-facet table so that a common edge between polygons could be identifiedmore rapidly the vertex table could be expanded to reference corresponding edges, for faster information retrieval

E_1:	V_1, V_2, S_1
E_2:	V_2, V_3, S_1
E_3:	V_3, V_1, S_1, S_2
E_4:	V_3, V_4, S_2
E_5:	V_4, V_5, S_2
E_6:	V_5, V_1, S_2

- ✓ Because 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.
- ✓ 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, and

(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

$$Ax + B y + C z + D = 0$$

Where,

- ➔ (x, y, z) is any point on the plane, and
- ➔ The coefficients A, B, C, and D (called *plane parameters*) 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 the three successive convex-polygon vertices, $(x1, y1, z1)$, $(x2, y2, z2)$, and $(x3, y3, z3)$, in a counterclockwise order and solve the following set of simultaneous linear plane equations for the ratios A/D, B/D, and C/D:

$$(A/D)xk + (B/D)yk + (C/D)zk = -1, \; k = 1, 2, 3$$

- 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

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

- It is possible that the coordinates defining a polygon facet may not be contained within a single plane.
- We can solve this problem by dividing the facet into a set of triangles; or we could find an approximating plane for the vertex list.
- One method for obtaining an approximating plane is to divide the vertex list into subsets, where each subset contains three vertices, and calculate plane parameters *A, B, C, D*for each subset.

Front and Back Polygon Faces

- The side of a polygon that faces into 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.
- Plane equations can be used to identify the position of spatial points relative to the polygon facets of an object.
- For any point (*x*, *y*, *z*) not on a plane with parameters *A*, *B*, *C*, *D*, we have

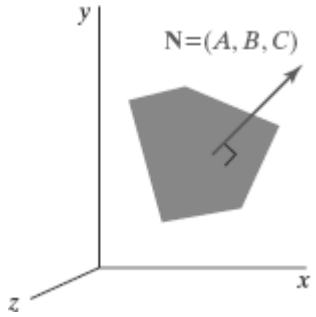
$$Ax + B\,y + C\,z + D \mathrel{!}= 0$$

- Thus, we can identify the point as either behind or in front of a polygon surface contained within that plane according to the sign (negative or positive) of

  $Ax + By + Cz + D$:

  if $Ax + B\,y + C\,z + D < 0$, the point (*x*, *y*, *z*) is behind the plane

  if $Ax + B\,y + C\,z + D > 0$, the point (*x*, *y*, *z*) is in front of the plane
- Orientation of a polygon surface in space can be described with the normal vector for the plane containing that polygon



- The normal vector points in a direction from inside the plane to the outside; that is, from the back face of the polygon to the front face.
- Thus, the normal vector for this plane is N = *(*1, 0, 0*)*, which is in the direction of the positive *x* axis.
- That is, the normal vector is pointing from inside the cube to the outside and is perpendicular to the plane $x = 1$.
- The elements of a normal vector can also be obtained using a vector crossproduct Calculation.
- We have a convex-polygon surface facet and a right-handed Cartesian system, we again select any three vertex positions,V1,V2, and V3, taken in counterclockwise order when viewing from outside the object toward the inside.
- Forming two vectors, one from V1 to V2 and the second from V1 to V3, we calculate N as the vector cross-product:

$$N = (V2 - V1) \times (V3 - V1)$$

- 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 in

$$Ax + By + Cz + D = 0$$

- The plane equation can be expressed in vector form using the normal N and the position P of any point in the plane as

$$N \cdot P = -D$$

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.
- ✓ By default, a polygon interior is displayed in a solid color, determined by the current color settings 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
- ✓ In some implementations of OpenGL, the following routine can be more efficient than generating a fill rectangle using glVertex specifications:

  glRect* (x1, y1, x2, y2);
- ✓ One corner of this rectangle is at coordinate position (*x*1, *y*1), and the opposite corner of the rectangle is at position (*x*2, *y*2).
- ✓ 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).
- ✓ Example

  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);

Polygon

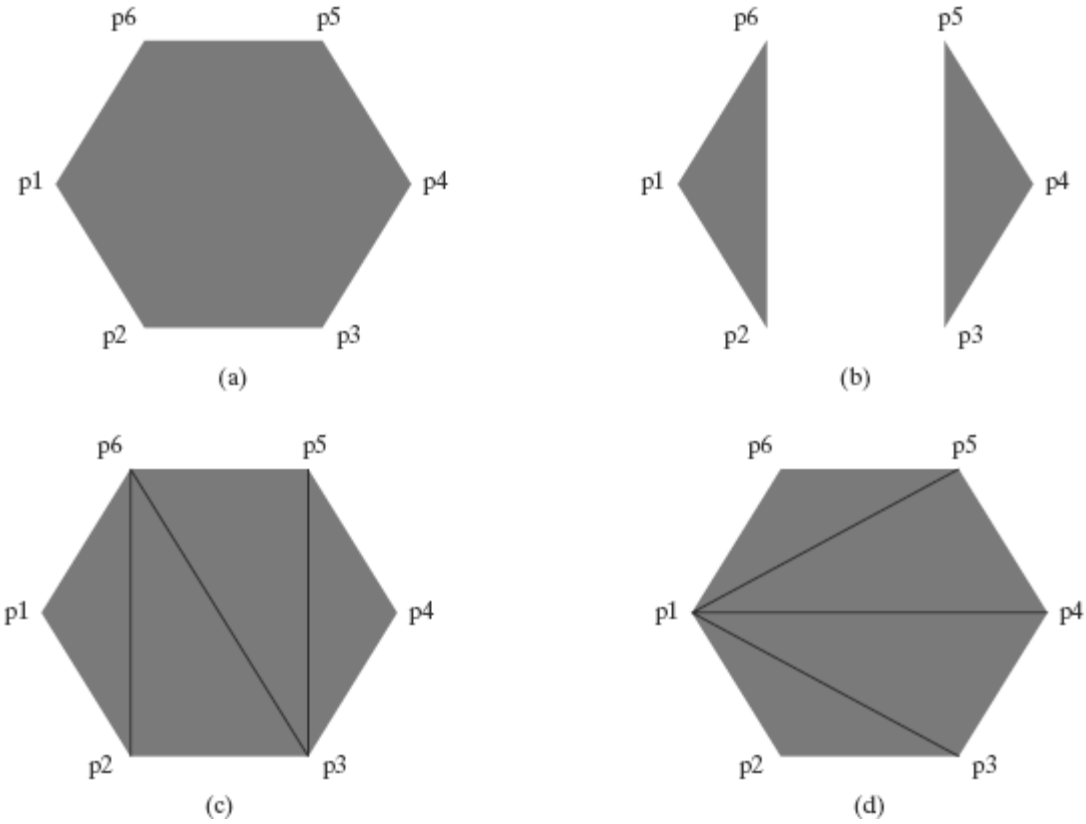
- ❖ With the OpenGL primitive constant GL POLYGON, we can display a single polygon fill area.
- ❖ Each of the points is represented as an array of (*x*, *y*) coordinate values:

  glBegin (GL_POLYGON);

  glVertex2iv (p1);

```
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
    glEnd ( );
```

- A polygon vertex list must contain at least three vertices. Otherwise, nothing is displayed.



(a) A single convex polygon fill area generated with the primitive constant GL POLYGON. (b) Two unconnected triangles generated with GL TRIANGLES.

(c) Four connected triangles generated with GL TRIANGLE STRIP.

(d) Four connected triangles generated with GL TRIANGLE FAN.

Triangles

- Displays the trianlges.

- Three primitives in triangles, GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP

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

- In this 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 triangle strip

```
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 $N \geq 3$ or nothing is displayed.
- 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.
- Example, our first triangle ($n = 1$) would be listed as having vertices (p1, p2, p6). The second triangle ($n = 2$) would have the vertex ordering (p6, p2, p3). Vertex ordering for the third triangle ($n = 3$) would be (p6, p3, p5). And the fourth triangle ($n = 4$) would be listed in the polygon tables with vertex ordering (p5, p3, p4).

Triangle Fan

- Another way to generate a set of connected triangles is to use the "fan" Approach

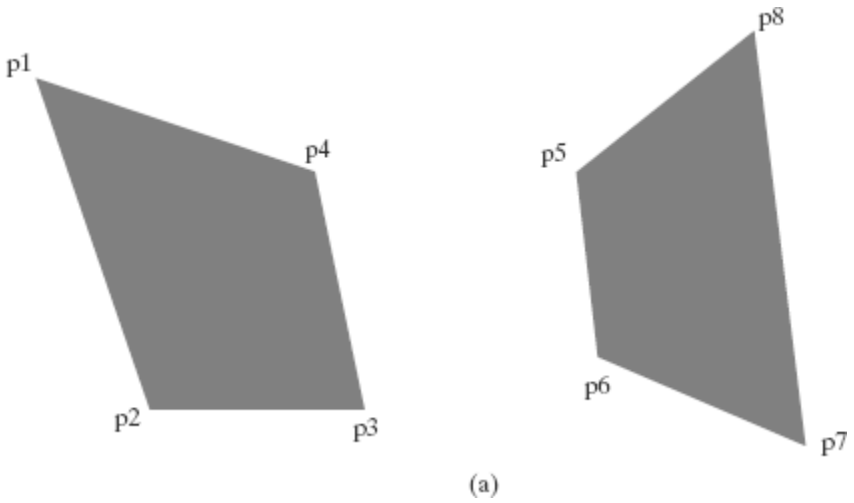
```
glBegin (GL_TRIANGLE_FAN);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
glEnd ( );
```

- For *N* vertices, we again obtain *N*−2 triangles, providing no vertex positions are repeated, and we must list at least three vertices be specified in the proper order to define front and back faces for each triangle correctly.
- Therefore, triangle 1 is defined with the vertex list (p1, p2, p3); triangle 2 has the vertex ordering (p1, p3, p4); triangle 3 has its vertices specified in the order (p1, p4, p5); and triangle 4 is listed with vertices (p1, p5, p6).

Quadrilaterals

- ✓ OpenGL provides for the specifications of two types of quadrilaterals.
- ✓ With the GL QUADS primitive constant and the following list of eight vertices, specified as two-dimensional coordinate arrays, we can generate the display shown in Figure (a):

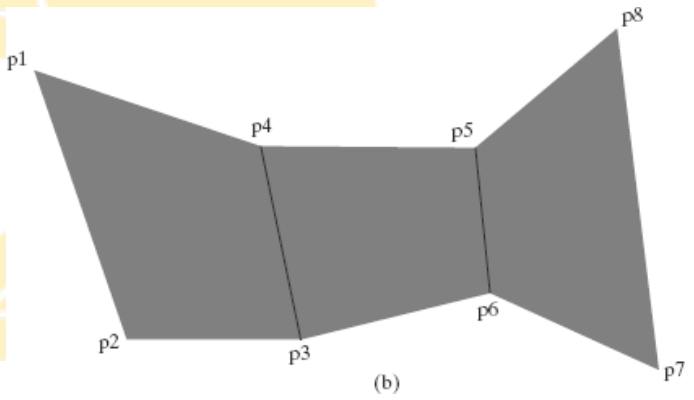
```
glBegin (GL_QUADS);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
        glVertex2iv (p7);
        glVertex2iv (p8);
glEnd ( );
```



(a)

- ✓ Rearranging the vertex list in the previous quadrilateral code example and changing the primitive constant to GL QUAD STRIP, we can obtain the set of connected quadrilaterals shown in Figure (b):

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



(b)

- ✓ For a list of N vertices, we obtain $N/2-1$ quadrilaterals, providing that $N \geq 4$. Thus, our first quadrilateral ($n = 1$) is listed as having a vertex ordering of (p1, p2, p3, p4). The second quadrilateral (n=2) has the vertex ordering (p4, p3, p6, p5), and the vertex ordering for the third quadrilateral (n=3) is (p5, p6, p7, p8).

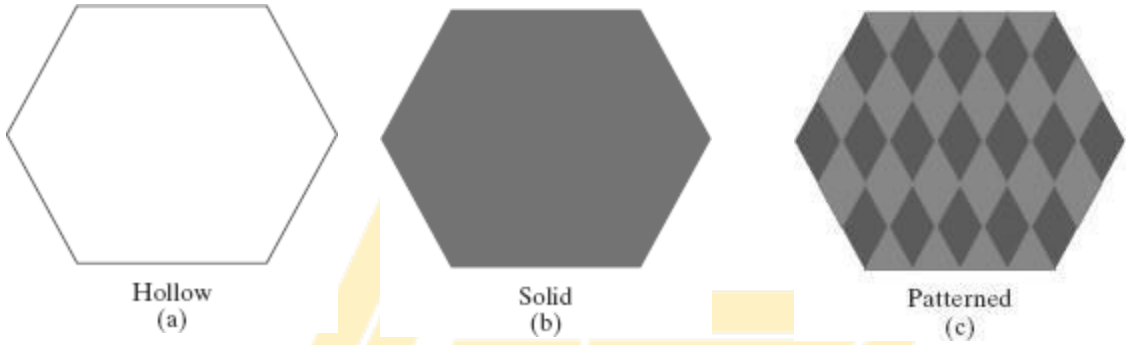
Fill-Area Attributes

➔ We can fill any specified regions, including circles, ellipses, and other objects with curved boundaries

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 fill pattern, or in a "hollow" style by showing only the boundary of the region



➔ We can also fill selected regions of a scene using various brush styles, color-blending combinations, or textures.

➔ For polygons, we could show the edges in different colors, widths, and styles; and we can 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.

➔ An array specifying a fill pattern is a *mask* that is to be applied to the display area.

➔ The mask is replicated in the horizontal and vertical directions until the display area is filled with nonoverlapping copies of the pattern.

➔ This process of filling an area with a rectangular pattern is called tiling, and a rectangular fill pattern is sometimes referred to as a tiling pattern predefined fill patterns are available in a system, such as the *hatch* fill patterns

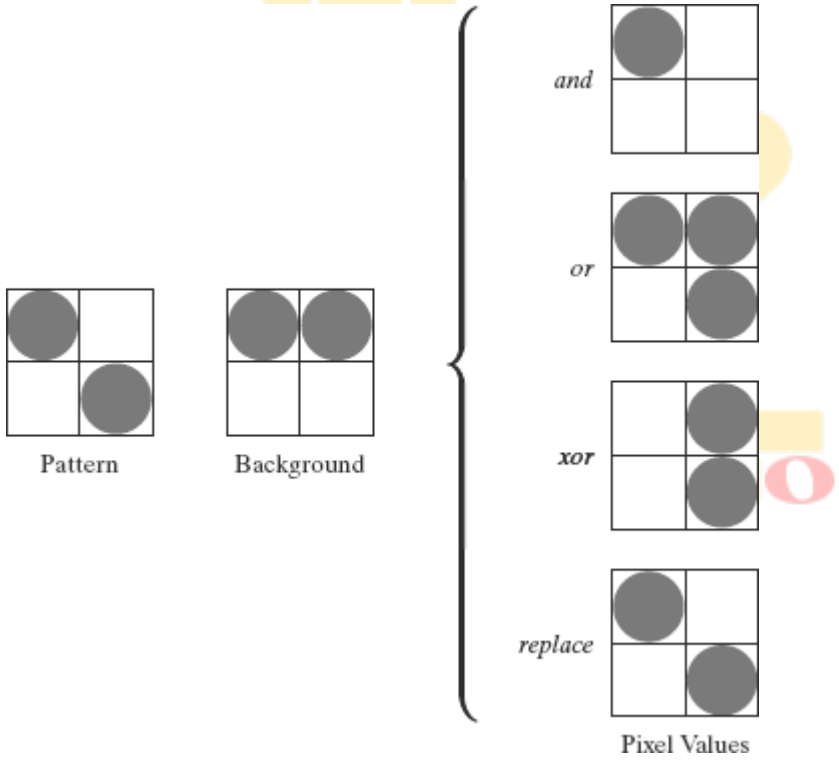
Diagonal
Hatch Fill

Diagonal
Crosshatch Fill

➔ Hatch fill could be applied to regions by drawing sets of line segments to display either single hatching or crosshtching

Color-Blended Fill Regions

- Color-blended regions can be implemented using either transparency factors to control the blending of background and object colors, or using simple logical or replace operations as shown in figure



- 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.
- The current color P of each pixel within the area to be refilled is some linear combination of F and B:

$$P = tF + (1 - t)B$$

- 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.
- If our color values are represented using separate red, green, and blue (RGB) components, each component of the colors, with

  $P = (P_R, P_G, P_B)$, $F = (F_R, F_G, F_B)$, $B = (B_R, B_G, B_B)$ *is used*
- We can thus calculate the value of parameter *t* using one of the RGB color components as follows:

$$t = \frac{P_k - B_k}{F_k - B_k}$$

  Where *k* = *R*, *G*, or *B*; and *Fk* != *Bk* .
- When two background colors *B*1 and *B*2 are mixed with foreground color F, the resulting pixel color P is

$$P = t0F + t1B1 + (1 - t0 - t1)B2$$

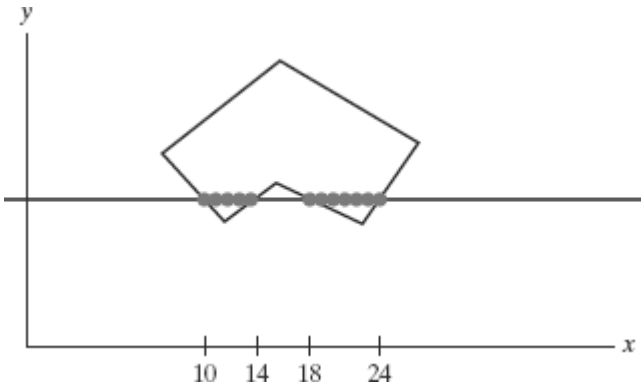
- Where the sum of the color-term coefficients *t*0, *t*1, and $(1 - t0 - t1)$ must equal 1.
- With three background colors and one foreground color, or with two background and two foreground colors, we need all three RGB equations to obtain the relative amounts of the four colors.

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 a scan line that lies within the interior of the fill region.

➔ The simplest area to fill is a polygon because each scanline intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations, where the equation for the scan line is simply y = constant.

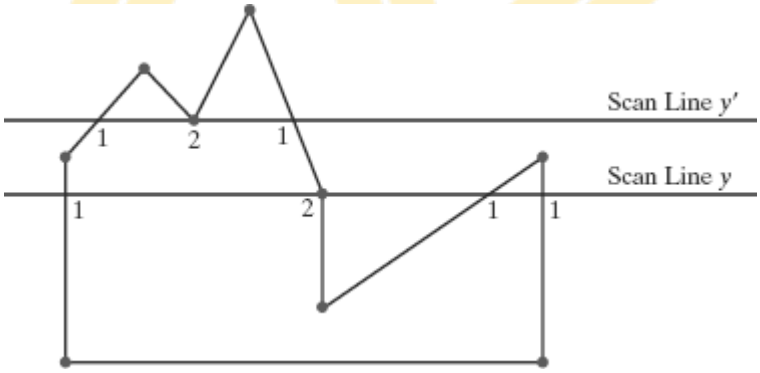


➔ Figure above illustrates the basic scan-line procedure for a solid-color fill of a polygon.

➔ For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color the fill color is applied to the five pixels from $x = 10$ to $x = 14$ and to the seven pixels from $x = 18$ to $x = 24$.

➔ Whenever a scan line passes through a vertex, it intersects two polygon edges at that point.

➔ In some cases, this can result in an odd number of boundary intersections for a scan line.



➔ Scan line *y*' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.

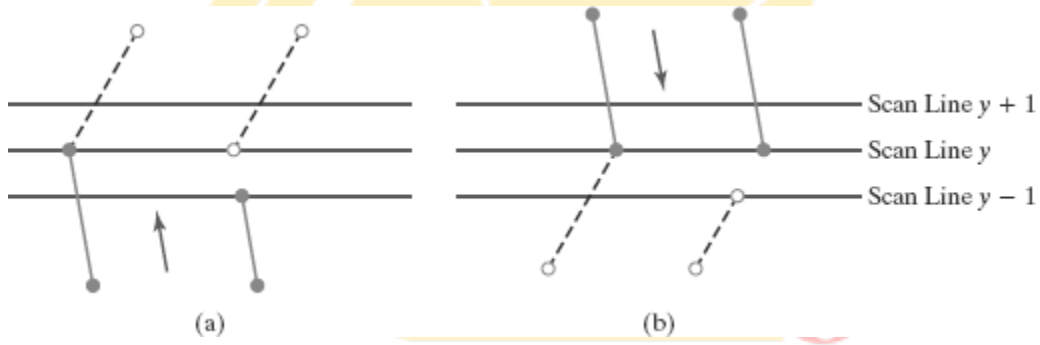
➔ But scan line *y* intersects five polygon edges.

➔ Thus, as we process scan lines, we need to distinguish between these cases.

➔ For scan line *y*, the two edges sharing an intersection vertex are on opposite sides of the scan line.

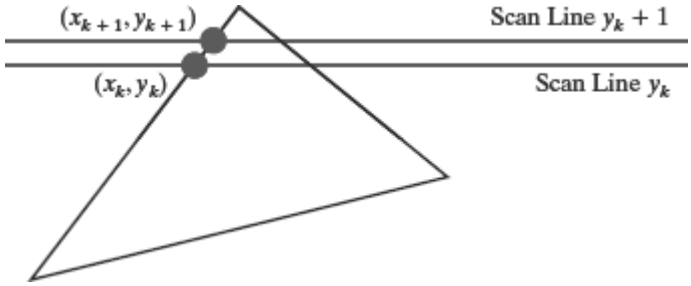
➔ But for scan line *y*', the two intersecting edges are both above the scan line.

- ➔ Thus, a vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point.
- ➔ If the three endpoint *y* values of two consecutive edges monotonically increase or decrease, we need to count the shared (middle) vertex as a single intersection point for the scan line passing through that vertex.
- ➔ Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.
- ➔ One method for implementing the adjustment to the vertex-intersection count 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.
- ➔ Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line



In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

- ➔ Coherence properties can be used in computer-graphics algorithms to reduce processing.
- ➔ Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines



➔ The slope of this edge can be expressed in terms of the scan-line intersection coordinates:

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

➔ Because 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.

➔ Along an edge with slope *m*, the intersection x_k value for scan line *k* above the initial scan line can be calculated as

$$x_k = x_0 + k/m$$

Where, 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

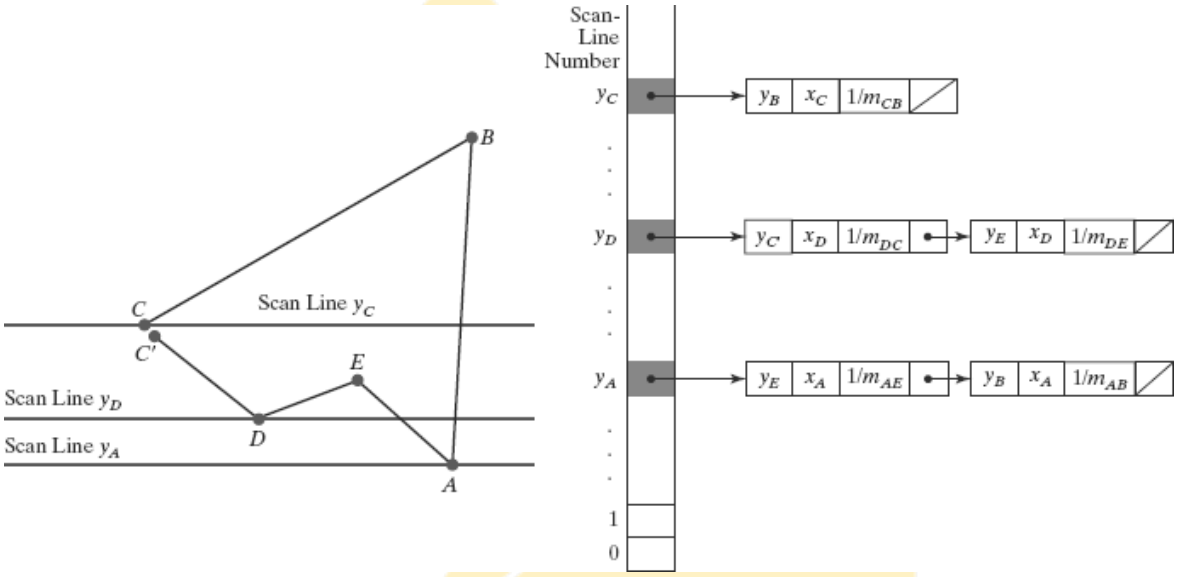
$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$$

➔ To perform a polygon fill efficiently, 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.

➔ 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 fromleft to right



➔ 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

➔ Implementation of edge-intersection calculations can be facilitated by storing Δx and Δy values in the sorted edge list

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.

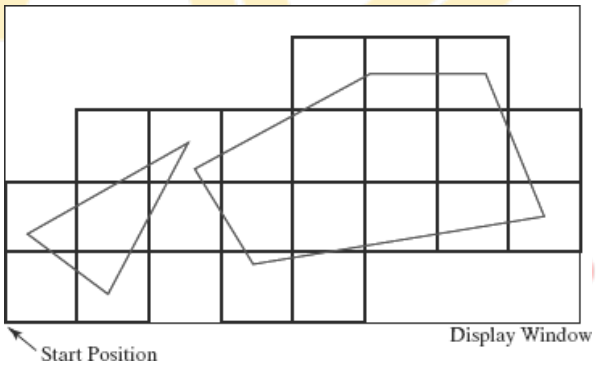
➔ A polygon fill pattern is displayed up to and including the polygon edges. Thus, there are no boundary lines around the fill region unless we specifically add them to the display

OpenGL Fill-Pattern Function

- To fill the polygon with a pattern in OpenGL, we use a 32 × 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.
- The fill pattern is specified in unsigned bytes using the OpenGL data type Glubyte

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



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

glPolygonStipple (fillPattern);

- We need to enable the fill routines before we specify the vertices for the polygons that are to be filled with the current pattern

glEnable (GL_POLYGON_STIPPLE);

- Similarly, we turn off pattern filling with

glDisable (GL_POLYGON_STIPPLE);

OpenGL Texture and Interpolation Patterns

- Another method for filling polygons is to use texture patterns.
- This can produce fill patterns that simulate the surface appearance of wood, brick, brushed steel, or some other material.
- 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.
- 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 (150, 50);
        glColor3f (0.0, 1.0, 0.0);
        glVertex2i (75, 150);
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 plotting a set of points only at the vertex positions.

➔ These options are selected with the function

glPolygonMode (face, displayMode);

➔ We use parameter face to designate which face of the polygon that 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.

➔ 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.

➔ Another option is to display a polygon with both an interior fill and a different color or pattern for its edges.

➔ The following code section fills a polygon interior with a green color, and then the edges are assigned a red color:

```
glColor3f (0.0, 1.0, 0.0);
/* Invoke polygon-generating routine. */
glColor3f (1.0, 0.0, 0.0);
glPolygonMode (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 referred to as **stitching.**

➔ 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.

➔ The calculation for this depth offset is

$$\textbf{depthOffset} = \textbf{factor1} \cdot \textbf{maxSlope} + \textbf{factor2} \cdot \textbf{const}$$

Where,

maxSlope is the maximum slope of the polygon and

const is an implementation constant

➔ As an example of assigning values to offset factors, we can modify the previous code segment as follows:

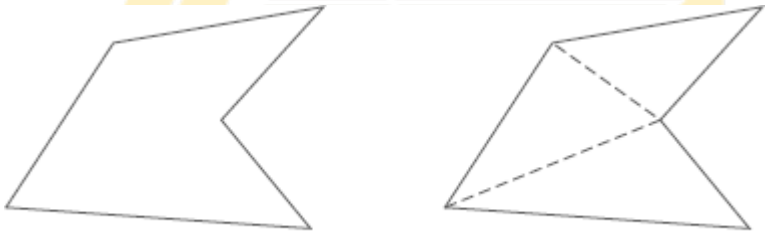
```
glColor3f (0.0, 1.0, 0.0);
```

glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
/* Invoke polygon-generating routine. */
glDisable (GL_POLYGON_OFFSET_FILL);
glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */

➔ Another method for eliminating the stitching effect along polygon edges is to use the OpenGL stencil buffer to limit the polygon interior filling so that it does not overlap the edges.

➔ 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. Then we could display the triangles.



Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

➔ Fortunately, OpenGL provides a mechanism that allows us to eliminate selected edges from a wire-frame display.

➔ So all we need do is 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 on again, which is the default.

➔ As an illustration of the use of an edge flag, the following code displays only two edges of the defined triangle

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

➔ Polygon edge flags can also be specified in an array that could be combined or associated with a vertex array.

➔ The statements for creating an array of edge flags are

```
glEnableClientState (GL_EDGE_FLAG_ARRAY);
glEdgeFlagPointer (offset, edgeFlagArray);
```

OpenGL Front-Face Function

- We can label selected surfaces in a scene independently 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.
- The constant GL_CCW labels a counterclockwise ordering of polygon vertices as front-facing, which is the default ordering.r its vertices is considered to be front-facing

2DGeometric Transformations:

Basic 2D Geometric Transformations,

Matrix representations and homogeneouscoordinates.

Inverse transformations,

2DComposite transformations,

Other 2D transformations,

Raster methods for geometric transformations,

OpenGL raster transformations

OpenGL geometric transformations function,

Two-Dimensional Geometric Transformations

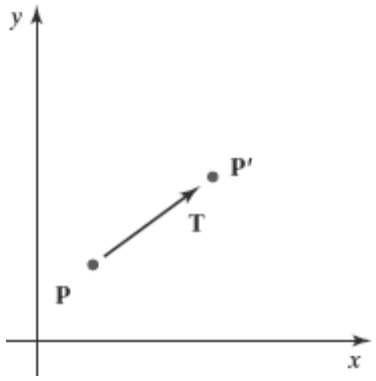
Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations.**

Basic Two-Dimensional Geometric Transformations

The geometric-transformation functions that are available in all graphics packages are those for translation, rotation, and scaling.

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.
- We are moving the original point position along a straight-line path to its 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 (*x*', *y*') as shown in Figure



- The translation values of x' and y' is calculated as

$$x' = x + t_x, \qquad y' = y + t_y$$

- The translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector Column vector representation is given as**

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

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

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

- Translation is a *rigid-body transformation* that moves objects without deformation.

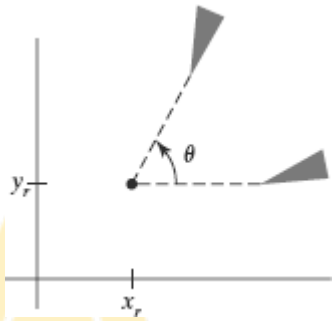
Code:

```
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 ( );
}
```

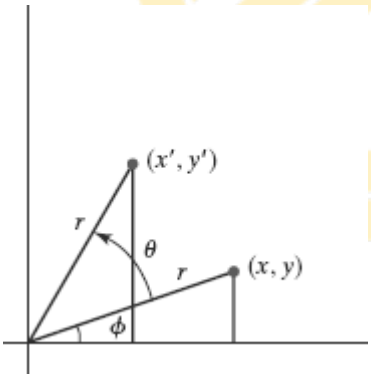
Two-Dimensional Rotation

- ✓ We generate a **rotation** transformation of an object by specifying a **rotation axis** and a **rotation angle.**

- ✓ 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 (parallel to the coordinate *z* axis)**.**
- ✓ Parameters for the two-dimensional rotation are the rotation angle θ and a position (*xr*, *yr*), called the **rotation point** (or **pivot point**), about which the object is to be rotated



- ✓ A positive value for the angle θ defines a counterclockwise rotation about the pivot point, as in above Figure , and a negative value rotates objects in the clockwise direction.
- ✓ The angular and coordinate relationships of the original and transformed point positions are shown in Figure



- ✓ In this figure, *r* is the constant distance of the point from the origin, angle φ is the original angular position of the point from the horizontal, and θ is the rotation angle.
- ✓ we can express the transformed coordinates in terms of angles θ and φ as

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

- ✓ The original coordinates of the point in polar coordinates are

$$x = r\cos\phi, \qquad y = r\sin\phi$$

- ✓ Substituting expressions of x and y in the eaquations of x' and y' we get

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

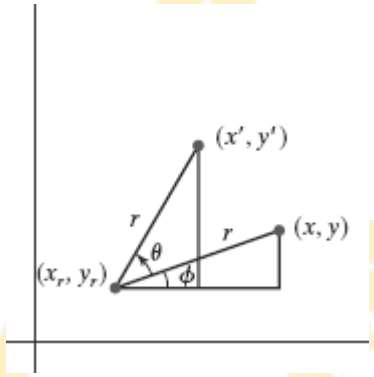
- ✓ We can write the rotation equations in the matrix form

P' = **R**· **P**

Where the rotation matrix is,

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

- ✓ Rotation of a point about an arbitrary pivot position is illustrated in Figure



- ✓ The transformation equations for rotation of a point about any specified rotation position (*xr* , *yr*):

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

Code:

```
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 ( );
}
```

Two-Dimensional Scaling

- ✓ To alter the size of an object, we apply a **scaling** transformation.
- ✓ A simple twodimensional scaling operation is performed by multiplying object positions (x, y) by **scaling factors** sx and sy to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y$$

- ✓ The basic two-dimensional scaling equations can also be written in the following matrix form

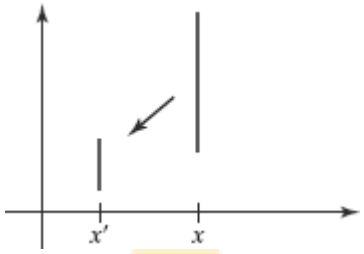
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

Where **S** is the 2 × 2 scaling matrix

- ✓ Any positive values can be assigned to the scaling factors sx and sy.
- ✓ Values less than 1 reduce the size of objects
- ✓ Values greater than 1 produce enlargements.
- ✓ Specifying a value of 1 for both sx and sy leaves the size of objects unchanged.
- ✓ When sx and sy are assigned the same value, a **uniform scaling** is produced, which maintains relative object proportions.

- ✓ Unequal values for *sx* and *sy* result in a **differential scaling** that is often used in design applications.
- ✓ In some systems, negative values can also be specified for the scaling parameters. This not only resizes an object, it reflects it about one or more of the coordinate axes.
- ✓ Figure below illustrates scaling of a line by assigning the value 0.5 to both *sx* and *sy*



- ✓ 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* , *yf*), are often chosen at some object position, such as its centroid but any other spatial position can be selected.
- ✓ For a coordinate position (*x*, *y*), the scaled coordinates (*x'*, *y'*) are then calculated from the following relationships:

$$x' - x_f = (x - x_f)s_x, \qquad y' - y_f = (y - y_f)s_y$$

- ✓ We can rewrite Equations to separate the multiplicative and additive terms as

$$x' = x \cdot s_x + x_f(1 - s_x)$$
$$y' = y \cdot s_y + y_f(1 - s_y)$$

- ✓ Where the additive terms *x f (*1 − *sx)* and *yf (*1 − *sy)* are constants for all points in the object.

Code:

```
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 ( );
}
```

Matrix Representations and Homogeneous Coordinates

- ✓ Each of the three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed in the general matrix form

$$P' = M_1 \cdot P + M_2$$

- ✓ With coordinate positions **P** and **P**' represented as column vectors.
- ✓ Matrix **M**1 is a 2 × 2 array containing multiplicative factors, and **M**2 is a two-element column matrix containing translational terms.
- ✓ For translation, **M**1 is the identity matrix.
- ✓ For rotation or scaling, **M**2 contains the translational terms associated with the pivot point or scaling fixed point.

Homogeneous Coordinates

- ➢ Multiplicative and translational terms for a two-dimensional geometric transformation can be combined into a single matrix if we expand the representations to 3 × 3 matrices
- ➢ We can use the third column of a transformation matrix for the translation terms, and all transformation equations can be expressed as matrix multiplications.
- ➢ We also need to expand the matrix representation for a two-dimensional coordinate position to a three-element column matrix

- A standard technique for accomplishing this is to expand each twodimensional coordinate-position representation (*x*, *y*) to a three-element representation (*xh*, *yh*, *h*), called **homogeneous coordinates,** where the **homogeneous parameter** *h* is a nonzero value such that

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

- A general two-dimensional homogeneous coordinate representation could also be written as (*h*·*x*, *h*·*y*, *h*).
- A convenient choice is simply to set *h* = 1. Each two-dimensional position is then represented with homogeneous coordinates (*x*, *y*, 1).
- The term *homogeneous coordinates* is used in mathematics to refer to the effect of this representation on Cartesian equations.

Two-Dimensional Translation Matrix

- ✓ The homogeneous-coordinate for translation is given by

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

- ✓ This translation operation can be written in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P}$$

with **T**(*tx*, *ty*) as the 3 × 3 translation matrix

Two-Dimensional Rotation Matrix

- ✓ Two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

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

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P}$$

- ✓ The rotation transformation operator **R**(θ) is the 3 × 3 matrix with rotation parameter θ.

Two-Dimensional Scaling Matrix

- ✓ A scaling transformation relative to the coordinate origin can now be expressed as the matrix multiplication

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

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P}$$

- ✓ The scaling operator S(sx, sy) is the 3 × 3 matrix with parameters sx and sy

Inverse Transformations

- ❖ For translation,we obtain the inverse matrix by negating the translation distances. Thus, if we have two-dimensional translation distances *tx* and *ty*, the inverse translation matrix is

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- ❖ An inverse rotation is accomplished by replacing the rotation angle by its negative.
- ❖ A two-dimensional rotation through an angle θ about the coordinate origin has the inverse transformation matrix

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ❖ We form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals. the inverse transformation matrix is

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Two-Dimensional Composite Transformations

- ✓ Forming products of transformation matrices is often referred to as a **concatenation,** or **composition,** of matrices if we want to apply two transformations to point position **P**, the transformed location would be calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P} \end{aligned}$$

- ✓ The coordinate position is transformed using the composite matrix**M**, rather than applying the individual transformations **M**1 and then**M**2.

Composite Two-Dimensional Translations

- ✓ If two successive translation vectors (*t*1*x*, *t*1*y*) and (*t*2*x*, *t*2*y*) are applied to a twodimensional coordinate position **P**, the final transformed location **P'** is calculated as

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

where **P** and **P'** are represented as three-element, homogeneous-coordinate column vectors

- ✓ Also, the composite transformation matrix for this sequence of translations is

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

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y})$$

Composite Two-Dimensional Rotations

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

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned}$$

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

$$\mathbf{R}(\theta 2) \cdot \mathbf{R}(\theta 1) = \mathbf{R}(\theta 1 + \theta 2)$$

- ✓ So that the final rotated coordinates of a point 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 in two dimensions produces the following composite scaling matrix

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

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

General Two-Dimensional Pivot-Point Rotation

(a) Original Position of Object and Pivot Point

(b) Translation of Object so that Pivot Point (x_r, y_r) is at Origin

(c) Rotation about Origin

(d) Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

- ✓ We can generate a two-dimensional rotation about any other pivot point (*xr , yr*) 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.

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

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

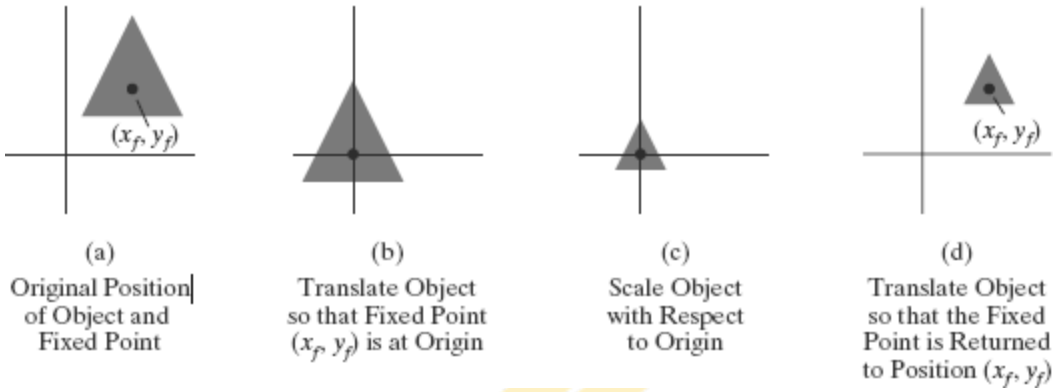
$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

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 $\mathbf{T}(-xr, -yr) = \mathbf{T}^{-1}(xr, yr)$.

General Two-Dimensional Fixed-Point Scaling



(a) Original Position of Object and Fixed Point

(b) Translate Object so that Fixed Point (x_f, y_f) is at Origin

(c) Scale Object with Respect to Origin

(d) Translate Object so that the Fixed Point is Returned to Position (x_f, y_f)

- ✓ To produce a two-dimensional scaling with respect to a selected fixed position ($x f$, yf), when we have a function that can scale relative to the coordinate origin only. This sequence is

  1. Translate the 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 of the translation in step (1) to return the object to its original position.

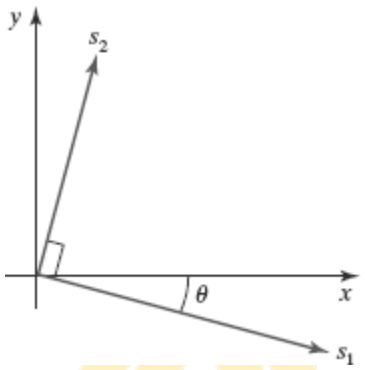
- ✓ Concatenating the matrices for these three operations produces the required scaling matrix:

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

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

General Two-Dimensional Scaling Directions

- ✓ Parameters *sx* and *sy* scale objects along the *x* and *y* directions.
- ✓ We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.
- ✓ Suppose we want to apply scaling factors with values specified by parameters *s*1 and *s*2 in the directions shown in Figure



- ✓ The composite matrix resulting from the product of these three transformations is

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

Matrix Concatenation Properties

Property 1:

- ✓ Multiplication of matrices is associative.
- ✓ For any three matrices,**M**1,**M**2, and**M**3, the matrix product **M**3 · **M**2 · **M**1 can be performed by first multiplying **M**3 and **M**2 or by first multiplying**M**2 and **M**1:

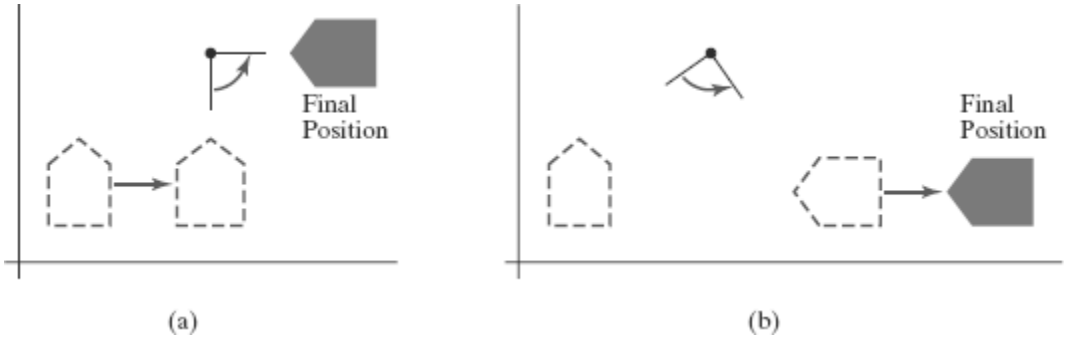
  M3 ·**M**2 · **M**1 = *(***M**3 · **M**2*)* ·**M**1 = **M**3 · *(***M**2 · **M**1*)*

- ✓ We can construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying)

Property 2:

- ✓ Transformation products, on the other hand, may not be commutative. The matrix product**M**2 · **M**1 is not equal to**M**1 · **M**2,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



- ✓ 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 in the x direction, then rotated counterclockwise through an angle of 45◦. In (b), the object is first rotated 45◦ counterclockwise, then translated in the x direction.

General Two-Dimensional Composite Transformations and Computational Efficiency

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

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

- ✓ The four elements *rsjk* are the multiplicative rotation-scaling terms in the transformation, which involve only rotation angles and scaling factors if an object is to be scaled and rotated about its centroid coordinates (*xc* , *yc*) and then translated, the values for the elements of the composite transformation matrix are

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y)$$

$$= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & x_c(1 - s_x \cos\theta) + y_c s_y \sin\theta + t_x \\ s_x \sin\theta & s_y \cos\theta & y_c(1 - s_y \cos\theta) - x_c s_x \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ Although the above matrix requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

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

- ✓ We need actually perform only four multiplications and four additions to transform coordinate positions.
- ✓ Because rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations
- ✓ If we are rotating in small angular steps about the origin, for instance, we can set cos θ 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 rotation calculations are

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

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

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

where the four elements *r jk* are the multiplicative rotation terms, and the elements *trx* and *try* are the translational terms

➔ A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion** transformation.

➔ In addition, the above matrix has the property that its upper-left 2 × 2 submatrix is an *orthogonal matrix*.

➔ If we consider each row (or each column) of the submatrix as a vector, then the two row vectors (*rxx*, *rxy*) and (*ryx*, *ryy*) (or the two column vectors) form an orthogonal set of unit vectors.

➔ Such a set of vectors is also referred to as an *orthonormal* vector set. Each vector has unit length as follows

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1$$

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

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0$$

➔ Therefore, if these unit vectors are transformed by the rotation submatrix, then the vector (*rxx*, *rxy*) is converted to a unit vector along the *x* axis and the vector (*ryx*, *ryy*) is transformed into a unit vector along the *y* axis of the coordinate system

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

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

➔ For example, the following rigid-body transformation first rotates an object through an angle θ about a pivot point (*xr* , *yr*) and then translates the object

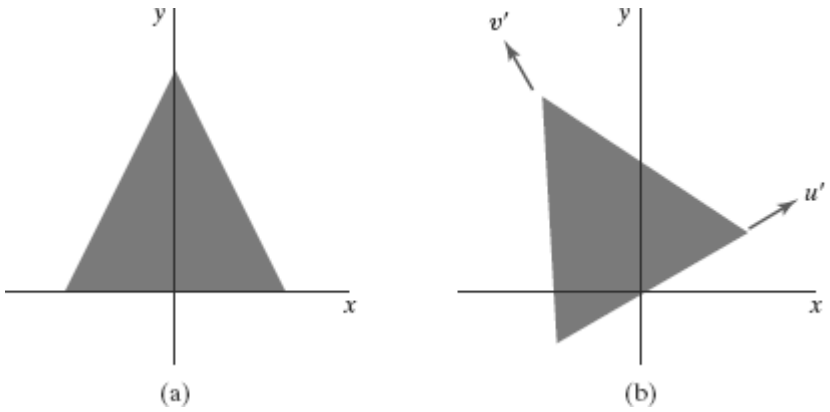
$$\mathrm{T}(t_x, t_y) \cdot \mathrm{R}(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

➔ Here, orthogonal unit vectors in the upper-left 2×2 submatrix are (cos θ, −sin θ) and (sin θ, cos θ).

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

Constructing Two-Dimensional Rotation Matrices

- ✓ The orthogonal property of rotation matrices is useful for constructing the 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.
- ✓ We might want to rotate an object to align its axis of symmetry with the viewing (camera) direction, or we might want to rotate one object so that it is above another object.
- ✓ Figure shows an object that is to be aligned with the unit direction vectors **u**_ and **v**



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

Other Two-Dimensional Transformations

Two such transformations

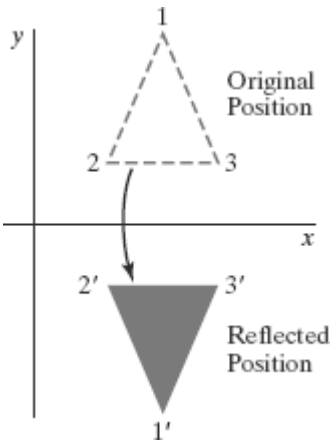
1. Reflection and
2. Shear.

Reflection

- ✓ A transformation that produces a mirror image of an object is called a **reflection.**
- ✓ For a two-dimensional reflection, this image is generated relative to an **axis of reflection** by rotating the object 180◦ about the reflection axis**.**
- ✓ Reflection about the line $y = 0$ (the x axis) is accomplished with the transformation Matrix

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

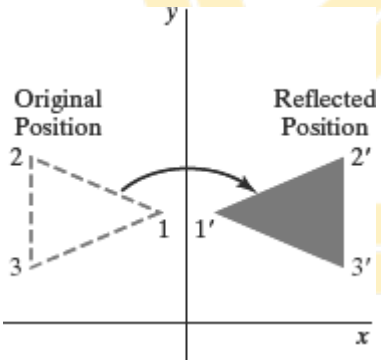
- ✓ This transformation retains x values, but "flips" the y values of coordinate positions.
- ✓ The resulting orientation of an object after it has been reflected about the x axis is shown in Figure



✓ A reflection about the line $x = 0$ (the y axis) flips x coordinates while keeping y coordinates the same. The matrix for this transformation is

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

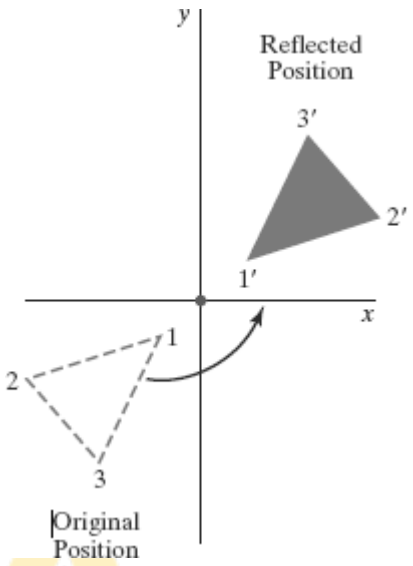
✓ Figure below illustrates the change in position of an object that has been reflected about the line $x = 0$.



✓ 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 the matrix representation for this reflection is

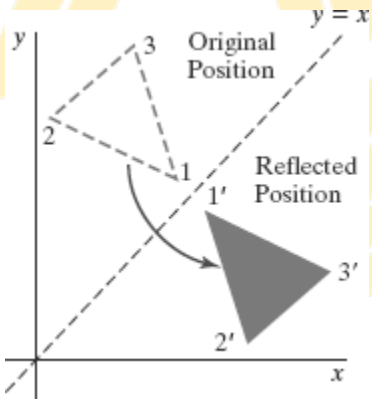
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

✓ An example of reflection about the origin is shown in Figure



✓ If we choose the reflection axis as the diagonal line $y = x$ (Figure below), the reflection matrix is

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



✓ 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

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

Shear

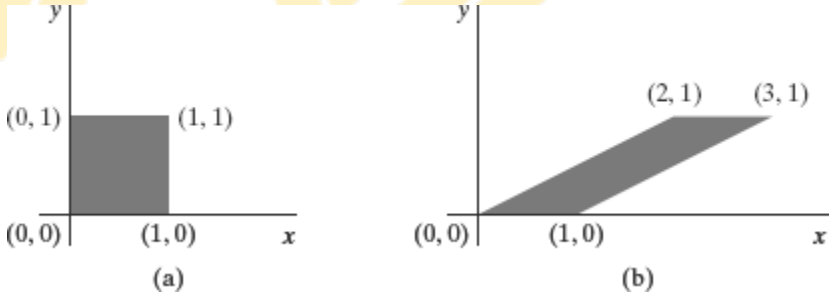
- ✓ A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear.**
- ✓ Two common shearing transformations are those that shift coordinate *x* values and 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 *shx* Setting parameter *shx* to the value 2, for example, changes the square into a parallelogram is shown below. Negative values for *shx* shift coordinate positions to the left.



A unit square (a) is converted to a parallelogram (b) using the x -direction shear with shx = 2.

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

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

Now, coordinate positions are transformed as

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

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

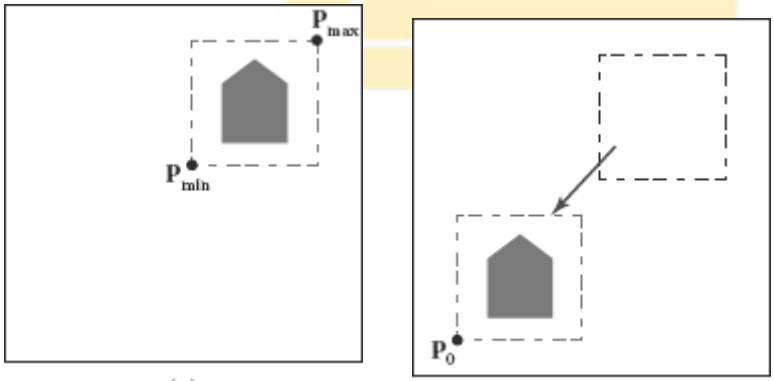
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

which generates the transformed coordinate values

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

Raster Methods for Geometric Transformations

- ✓ 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
- ✓ Few arithmetic operations are needed, so the pixel transformations are particularly efficient.
- ✓ Functions that manipulate rectangular pixel arrays are called *raster operations* and moving a block of pixel values from one position to another is termed a *block transfer*, a *bitblt*, or a *pixblt*.
- ✓ Figure below illustrates a two-dimensional translation implemented as a block transfer of a refresh-buffer area



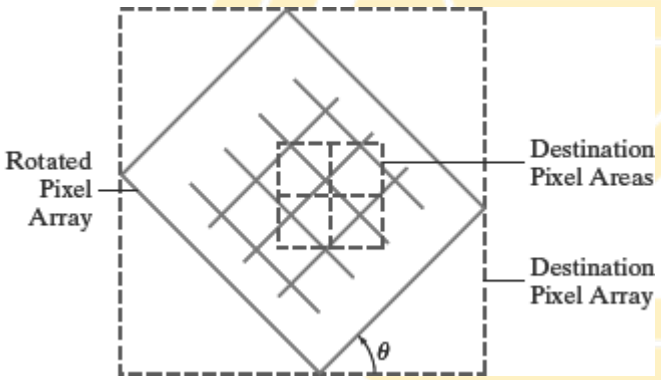
Translating an object from screen position (a) to the destination position shown in (b) by moving a rectangular block of pixel values. Coordinate positions Pmin and Pmax specify the limits of the rectangular block to be moved, and P0 is the destination reference position.

- ✓ Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
- ✓ We can rotate a two-dimensional object or pattern 90◦ counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.
- ✓ A 180◦ rotation is obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows.
- ✓ Figure below demonstrates the array manipulations that can be used to rotate a pixel block by 90◦ and by 180◦.

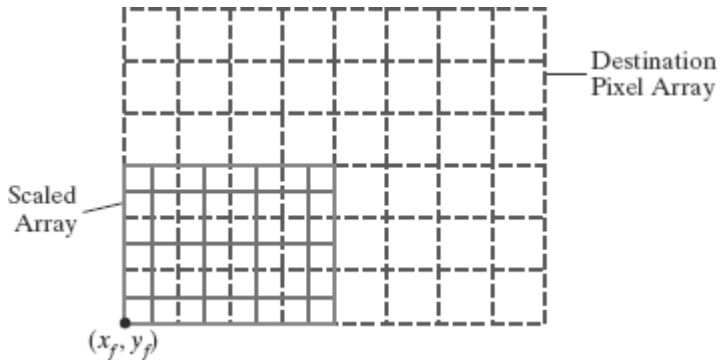
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \quad \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \quad \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(a) (b) (c)

- ✓ For array rotations that are not multiples of 90◦, we need to do some extra processing.
- ✓ The general procedure is illustrated in Figure below.



- ✓ 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.
- ✓ Pixel areas in the original block are scaled, using specified values for *sx* and *sy*, 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



OpenGL Raster Transformations

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

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

- The first four parameters in this function give the location and dimensions of the pixel block; and the OpenGL symbolic constant **GL_COLOR** specifies that it is color values are to be copied.

- A block of RGB color values in a buffer can be saved in an array with the function

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

- If color-table indices are stored at the pixel positions, we replace the constant GL RGB with GL_COLOR_INDEX.

- To rotate the color values, we rearrange the rows and columns of the color array, as described in the previous section. Then we put the rotated array back in the buffer with

  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);

- ❖ We can also combine raster transformations with logical operations to produce various effects with the *exclusive or* operator

OpenGL Functions for Two-Dimensional Geometric Transformations

- ✓ To perform a translation, we invoke the translation routine and set the components for the three-dimensional translation vector.
- ✓ In the rotation function, we specify the angle and the orientation for a rotation axis that intersects the coordinate origin.
- ✓ In addition, a scaling function is used to set the three coordinate scaling factors relative to the coordinate origin. In each case, the transformation routine sets up a 4 × 4 matrix that is applied to the coordinates of objects that are referenced after the transformation call

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.
- ✓ example: **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 degree.
- ✓ For example, the statement: **glRotatef (90.0, 0.0, 0.0, 1.0);**

➔ 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
- ✓ **Example: glScalef (2.0, -3.0, 1.0);**

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.*
- ✓ We specify the *modelview mode* with the statement

glMatrixMode (GL_MODELVIEW);

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

- ✓ 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
- ✓ 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 as follows:

  glMultMatrix* (otherElements16);

- ✓ Again, 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.
- ✓ Thus, assuming that the current matrix is the modelview matrix, which we designate as **M**, then the updated modelview matrix is computed as

  M = **M**· **M**'
- ✓ The **glMultMatrix** function can also be used to set up any transformation sequence with individually defined matrices.
- ✓ For example,

  glMatrixMode (GL_MODELVIEW);

  glLoadIdentity ( ); // Set current matrix to the identity.

  glMultMatrixf (elemsM2); // Postmultiply identity with matrix M2.

  glMultMatrixf (elemsM1); // Postmultiply M2 with matrix M1.

produces the following current modelview matrix:

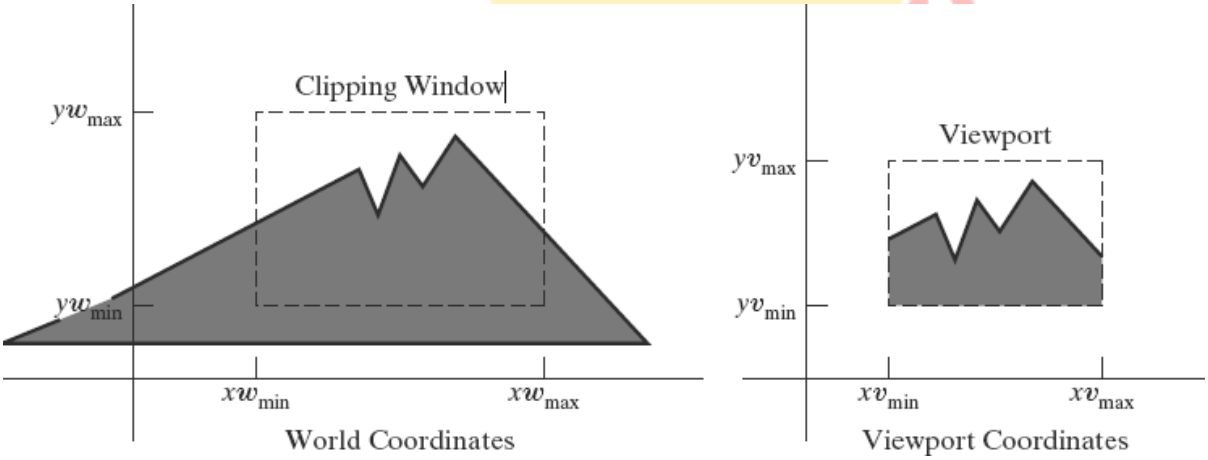
M = **M**2 · **M**1

2.3 Two Dimensional Viewing

2.3.1 2D viewing pipeline
2.3.1 OpenGL 2D viewing functions.

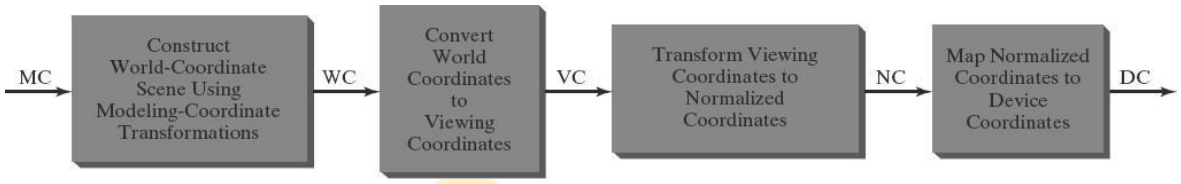
The Two-Dimensional Viewing Pipeline

- A section of a two-dimensional scene that is selected for display is called a clipping Window.
- Sometimes the clipping window is alluded to as the *world window* or the *viewing window*
- Graphics packages allow us also to control the placement within the display window using another "window" called the **viewport**.
- The clipping window selects *what* we want to see; the viewport indicates *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
- Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
- We first consider only rectangular viewports and clipping windows, as illustrated in Figure



Viewing Pipeline

- The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a **two-dimensional viewing transformation**.
- This transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation*
- We can describe the steps for two-dimensional viewing as indicated in Figure



- Once aworld-coordinate scene has been constructed,wecould set up a separate two-dimensional, **viewingcoordinate 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.
- Systems use normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1.
- 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

OpenGL Two-Dimensional Viewing Functions

- The GLU library provides a function for specifying a two-dimensional clipping window, and we have GLUT library functions for handling display windows.

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.

- ✓ We must set the parameters for the clipping window as part of the projection transformation.
- ✓ Function:

  glMatrixMode (GL_PROJECTION);
- ✓ 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);
- ✓ This function specifies an orthogonal projection for mapping the scene to the screen the orthogonal projection has no effect on our two-dimensional scene other than to convert object positions to normalized coordinates.
- ✓ Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping routines.
- ✓ 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,

  ➔ **xvmin** and **yvmin** specify the position of the lowerleft corner of the viewport relative to the lower-left corner of the display window,

➔ **vpWidth** and **vpHeight** are pixel width and height of the viewport

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

$$xv_{max} = xv_{min} + vpWidth, \qquad yv_{max} = yv_{min} + vpHeight$$

- ✓ 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.

Creating a GLUT Display Window

- ✓ 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.
- ✓ We first need to initialize GLUT with the following function:

glutInit (&argc, argv);

- ✓ We have three functions inGLUTfor defining a display window and choosing its dimensions and position:

1. **glutInitWindowPosition (xTopLeft, yTopLeft);**

➔ gives the integer, screen-coordinate position for the top-left corner of the display window, relative to the top-left corner of the screen

2. **glutInitWindowSize (dwWidth, dwHeight);**

➔ 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

3. **glutCreateWindow ("Title of Display Window");**

➔ 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

Setting the GLUT Display-Window Mode and Color

✓ Various display-window parameters are selected with the GLUT function

1. **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.

2. **glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);**

➔ The color mode specification **GLUT_RGB** is equivalent to **GLUT_RGBA.**

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

➔ A background color for the display window is chosen in RGB mode with the OpenGL routine

4. **glClearIndex (index);**

➔ This function sets the display window color using color-index mode,

➔ 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 the first window that is created.

✓ Function:

windowID = glutCreateWindow ("A Display Window");

Deleting a GLUT Display Window

- ✓ 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);
- ✓ 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 the function

  glutPositionWindow (xNewTopLeft, yNewTopLeft);
- ✓ 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 ( );
- ✓ Whenever the size of a display window is changed, its aspect ratio may change and objects may be distorted from their original shapes. We can adjust for a change in display-window dimensions using the statement

  glutReshapeFunc (winReshapeFcn);

Managing Multiple GLUT Display Windows

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

- ✓ 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");
- ✓ We also can change the name of the display window with a similar command:

  glutSetWindowTitle ("New Window Name");
- ✓ 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.*
- ✓ 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.

- ✓ 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.
- ✓ Each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows

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);

  where, shape can be

  ➔ **GLUT_CURSOR_UP_DOWN :** an up-down arrow.

  ➔ **GLUT_CURSOR_CYCLE:** A rotating arrow is chosen

  ➔ **GLUT_CURSOR_WAIT:** a wristwatch shape.

  ➔ **GLUT_CURSOR_DESTROY:** a skull and crossbones

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
- ✓ Then we invoke the following function to assign something to that window:

  glutDisplayFunc (pictureDescrip);

- ✓ 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.
- ✓ 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 ( );

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 ( );

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);

- ✓ 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, for the stateParam we can have the values
  - ➔ **GLUT_WINDOW_X**: obtains the *x*-coordinate position for the top-left corner of the current display window
  - ➔ **GLUT_WINDOW_WIDTH** or **GLUT_SCREEN_WIDTH** : retrieve the current display-window width or the screen width with**.**