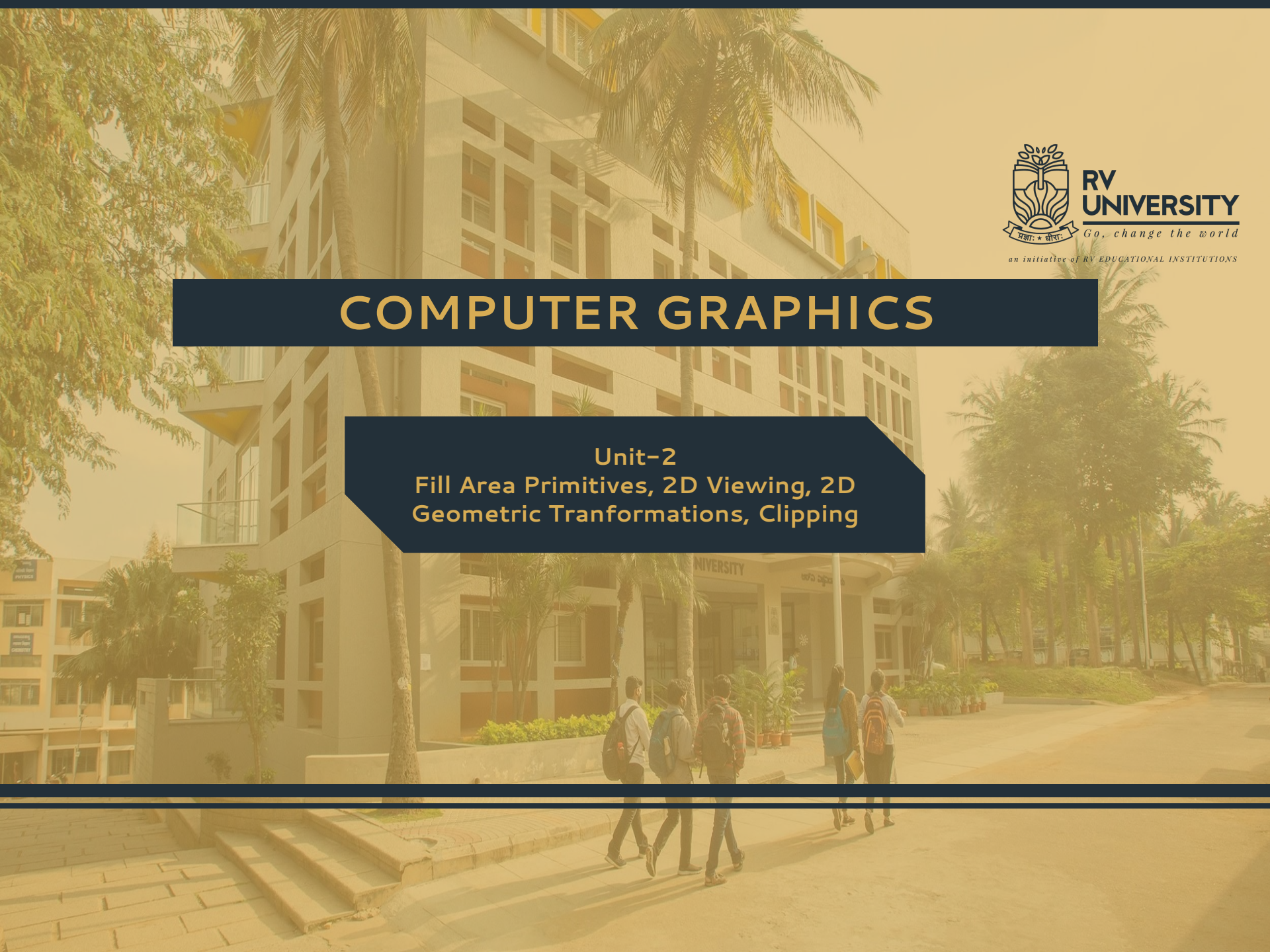


RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

COMPUTER GRAPHICS

Unit-2
Fill Area Primitives, 2D Viewing, 2D Geometric Tranformations, Clipping

Fill area primitives-Polygon Fill Areas

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

polygon

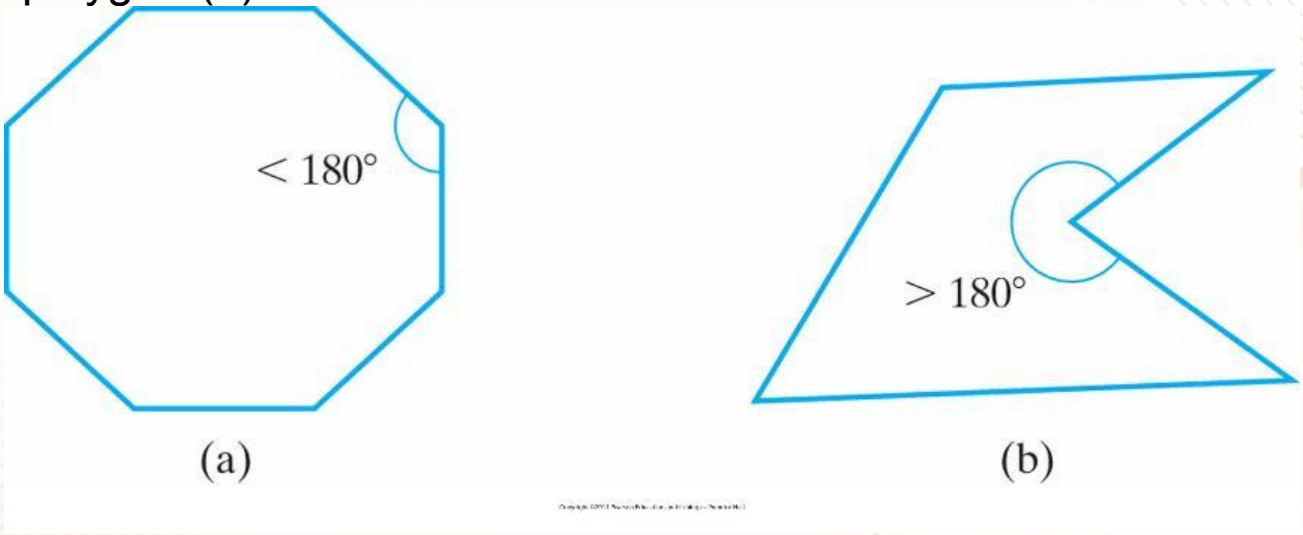
- A plane figure specified by a set of three or more coordinate positions, called vertices, that are connected in sequence by straight-line segments, called the edges or sides of the polygon.

Simple polygon

- All its vertices within a single plane
- No edge crossings.
- Ex: triangles, rectangles, octagons, and decagons

Polygon classifications

Figure : A convex polygon (a), and a concave polygon (b).



Concave polygons may present problems when implementing fill algorithms

Convex

- An interior angle of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges. If all interior angles of a polygon are less than or equal to 180◦
- its interior lies completely on one side of the infinite extension line of any one of its edges.
- 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

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

Identifying concave Polygons

Characteristics

- A concave polygon has at least one interior angle greater than 180◦.
- Extension of some edges of a concave polygon will intersect other edges,
- Some pair of interior points will produce a line segment that intersects the polygon boundary.

- If we set up a vector for each polygon edge, then we can use the cross-product of adjacent edges to test for concavity.

Convex

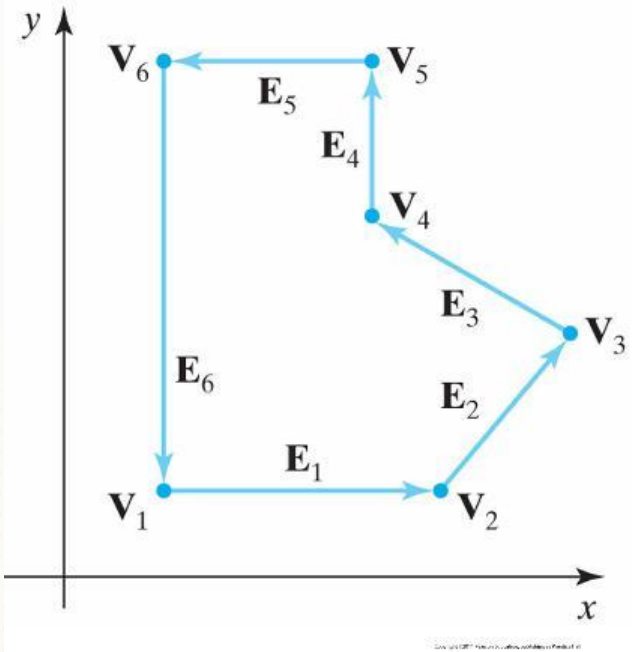
- All such vector products will be of the same sign (positive or negative)

Concave

- if some cross-products yield a positive value and some a negative value, we have a concave polygon..

Identifying concave polygons

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



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

For concave polygons, some cross-products are positive, some are negative

Splitting concave polygons: vector method

Form the edge vectors $E_k = V_{k+1} - V_k$

Calculate the cross-products of successive edge vectors in a counter-clockwise manner

Use the cross-product test to identify concave polygons

If any cross-product has a negative z-component, the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair

Splitting example

EXAMPLE 4 - 1 **The Vector Method for Splitting Concave Polygons**

Figure 4-10 shows a concave polygon with six edges. Edge vectors for this polygon can be expressed as

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

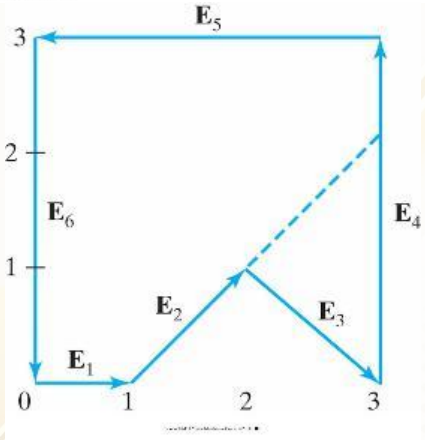
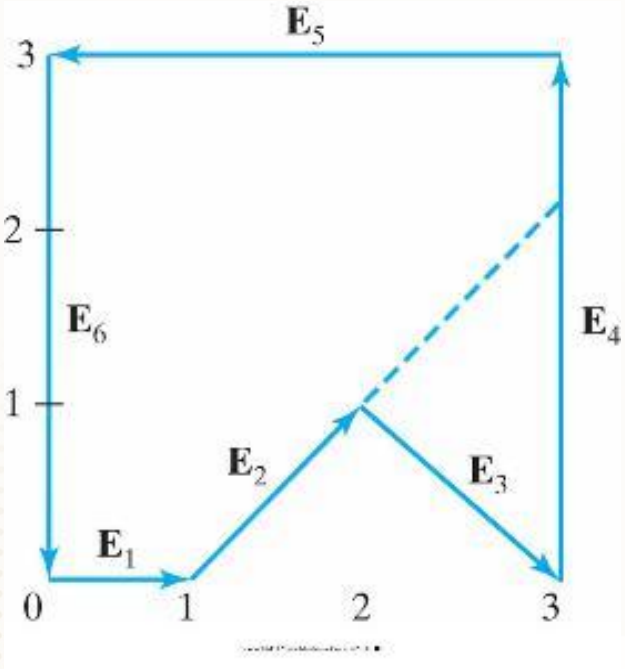


Figure : Splitting a concave polygon using the vector method.

Splitting example



Splitting polygons

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

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

Since the cross-product $\mathbf{E}_2 \times \mathbf{E}_3$ has a negative z component, we split the polygon along the line of vector $\mathbf{E}_2$. The line equation for this edge has a slope of 1 and a y intercept of -1. We then determine the intersection of this line with the other polygon edges to split the polygon into two pieces. No other edge cross-products are negative, so the two new polygons are both convex.

Rotation method

Shift the position of the polygon so that each vertex Vk in turn is at the coordinate origin.

Rotate the polygon about the origin in a clockwise direction so that the next vertex Vk+1 is on the x axis.

If the following vertex, Vk+2, is below the x axis, the polygon is concave.

Split the polygon along the x axis to form two new polygons.

Repeat the concave test for each of the two new polygons.

Repeat until we have tested all vertices in the polygon list.

Rotation method

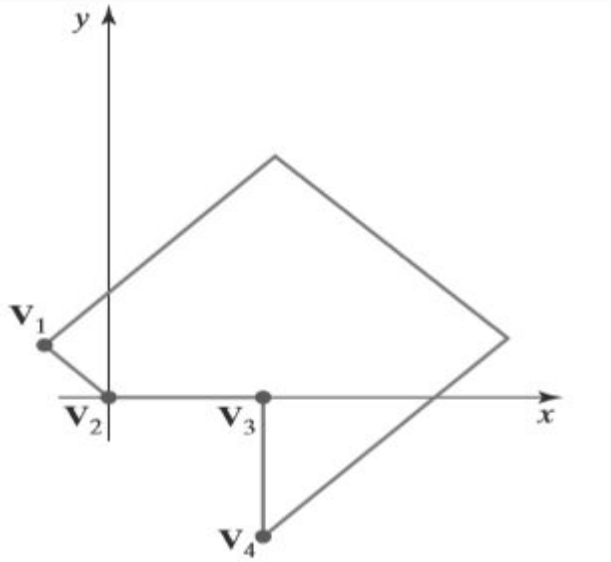


FIGURE 11
Splitting a concave polygon using the rotational method. After moving V_2 to the coordinate origin and rotating V_3 onto the x axis, we find that V_4 is below the x axis. So we split the polygon along the line of $\overline{V_2V_3}$, which is the x axis.

Splitting a convex polygon into a set of triangles

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Triangles make several important processing routines simple

Define any sequence of three consecutive vertices to be a new polygon (triangle)

Delete the middle triangle vertex from the original vertex list

Apply the same procedure to the modified list to strip off another triangle

Continue until the original polygon is reduced to just three vertices (the last triangle)

Polygon Tables

Two groups:

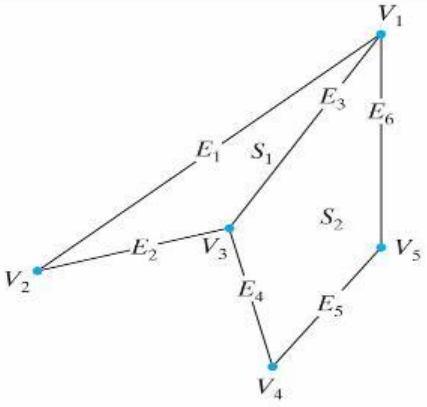
- Geometric table – vertex coordinates
- attribute table- degree of transparency, surface reflectivity, texture characteristics

Geometric table

- Vertex table- coordinate values for each vertex
- Edge table- pointers back to vertex table to identify vertices for each edge
- Surface facet table - pointers back to edge table to identify edges for each polygon.

Polygon tables

Figure : Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.



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

A surface shape can be defined as a mesh of polygon patches. Geometric data for objects can be arranged in 3 lists

Plane equations

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

$$A\,x + B\,y + C\,z + D = 0$$

where (x, y, z) is any point on the plane

A, B and C are constants describing the spatial properties of the plane.

Solutions to plane equations

We obtain A, B, C, and D by solving a set of three plane equations.

Select 3 successive convex polygon vertices in a counter-clockwise manner and solve the following set of simultaneous linear plane equations for A/D, B/D, and C/D

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

Solution to this set of equations can be obtained using **Cramer's rule**

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

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

Solutions

Then

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

Values of A, B , C and D are computed for each polygon and stored in data structure.

Coordinates of polygon face may not be in single plane.

Divide facet into set of triangles

Front and back faces

Back face

- Side of a polygon that faces into object interior.

Front face

- Visible or outward face

- Outside/inside
  - Any point that is not on the plane and that is visible to the front face of a polygon surface section is said to be in front of (or outside) the plane, and, thus, outside the object.
  - And any point that is visible to the back face of the polygon is behind (or inside) the plane. A point that is behind (inside) all polygon surface planes is inside the object.

A point that is behind (inside) all polygon surface planes is inside the object.

We need to keep in mind that this inside/outside classification is relative to the plane containing the polygon.

Any points (x,y,z) not on a plane with parameters A,B,C,D we have

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

if	$A\,x + B\,y + C\,z + D < 0,$	the point (x, y, z) is behind the plane
if	$A\,x + B\,y + C\,z + D > 0,$	the point (x, y, z) is in front of the plane

Any point outside(in front of) satisfies x-1>0
Any point inside the plane has an x coordinate value less than 1

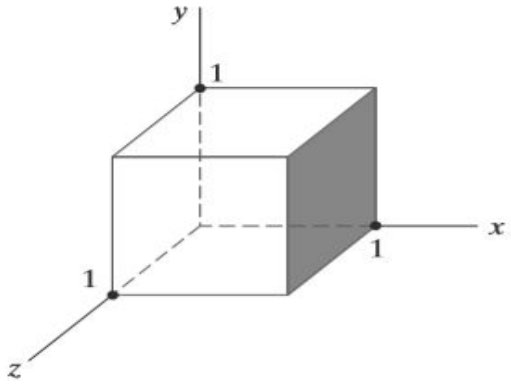


FIGURE 18
The shaded polygon surface of the unit cube has the plane equation $x - 1 = 0$.

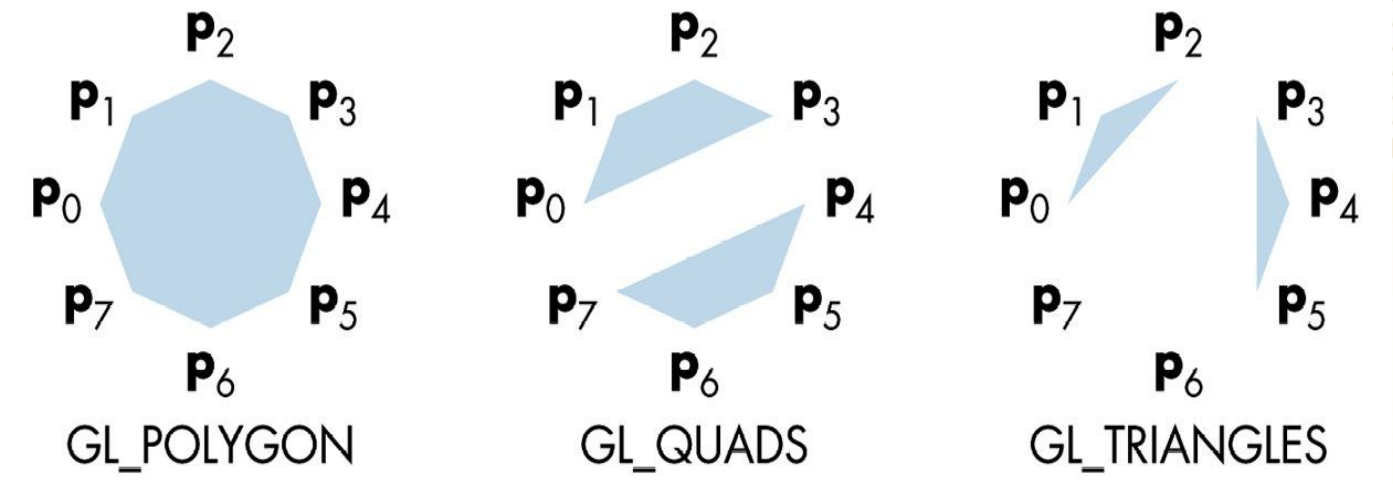
Front and back polygon faces

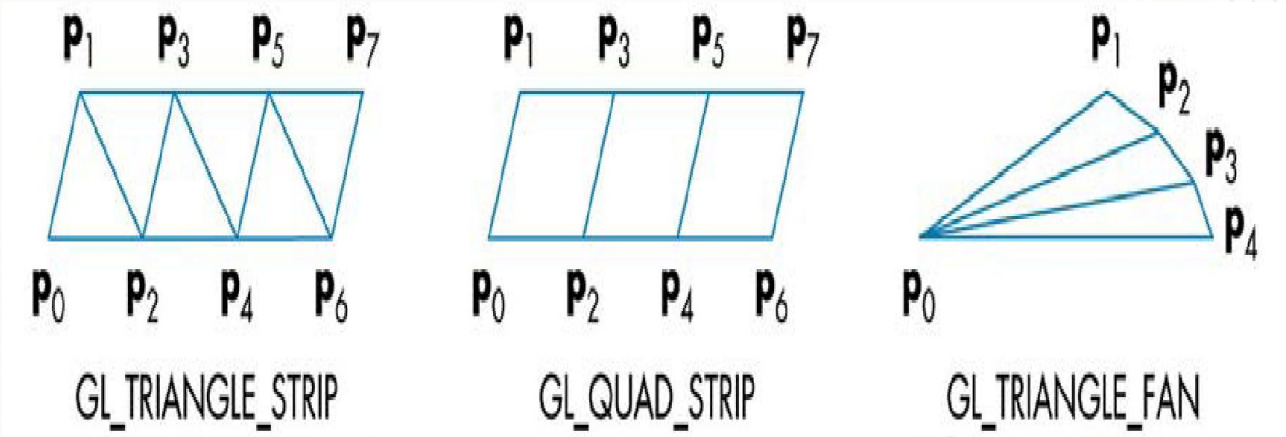
Faces can be determined by the sign of Ax+By+Cz+D

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

$$\text{if} \quad Ax + By + Cz + D < 0, \qquad \text{the point } (x, y, z) \text{ is behind the plane}$$
$$\text{if} \quad Ax + By + Cz + D > 0, \qquad \text{the point } (x, y, z) \text{ is in front of the plane}$$

OpenGL polygon fill-area functions





Strips and Fans(GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN)

groups of triangles or quadrilaterals that share vertices and edges.

Triangle strip

Each additional vertex is combined with the previous two vertices to define a new triangle.

Quadstrip

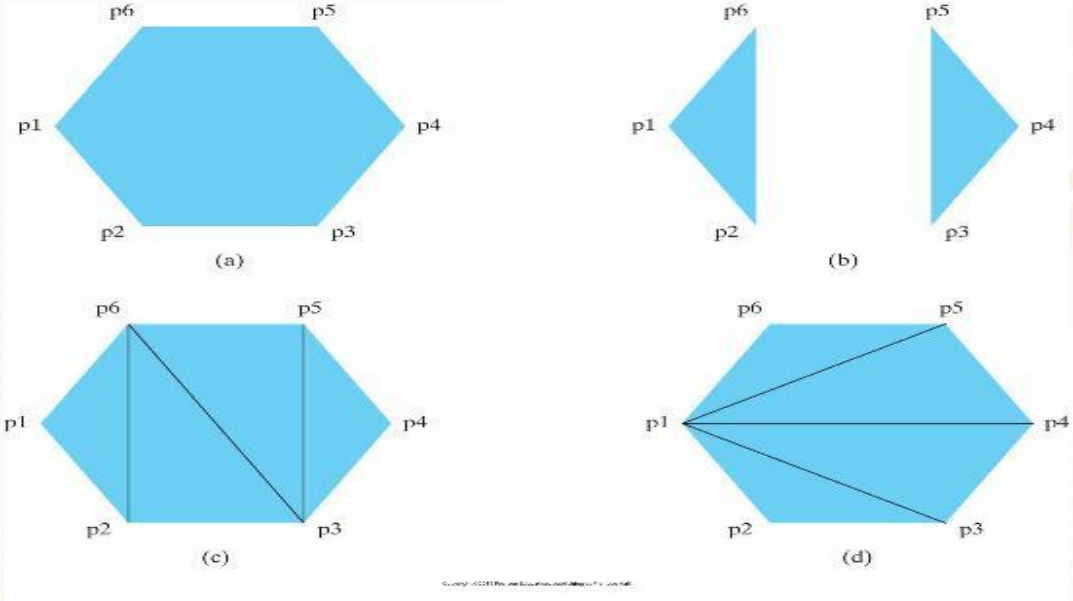
Combine two new vertices with the previous two vertices to define a new quadrilateral.

Triangle fan

Triangle fan is based on the one fixed point.The next two points determine the first triangle, and subsequent triangles are formed from one new point,the previous point and the first

OpenGL polygon fill-area functions

Figure : Displaying polygon fill areas using a list of six vertex positions. (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.



Fill-area examples

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

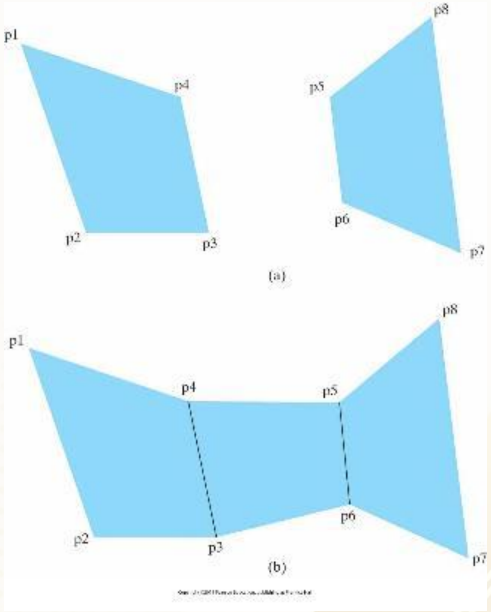
```
glBegin (GL_TRIANGLE_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p4);
glEnd ( );
```

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

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

Quadrilateral fill-areas

Figure : Displaying quadrilateral fill areas using a list of eight vertex positions. (a) Two unconnected quadrilaterals generated with GL_QUADS. (b) Three connected quadrilaterals generated with GL_QUAD_STRIP.




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

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

General scan line polygon fill algorithm

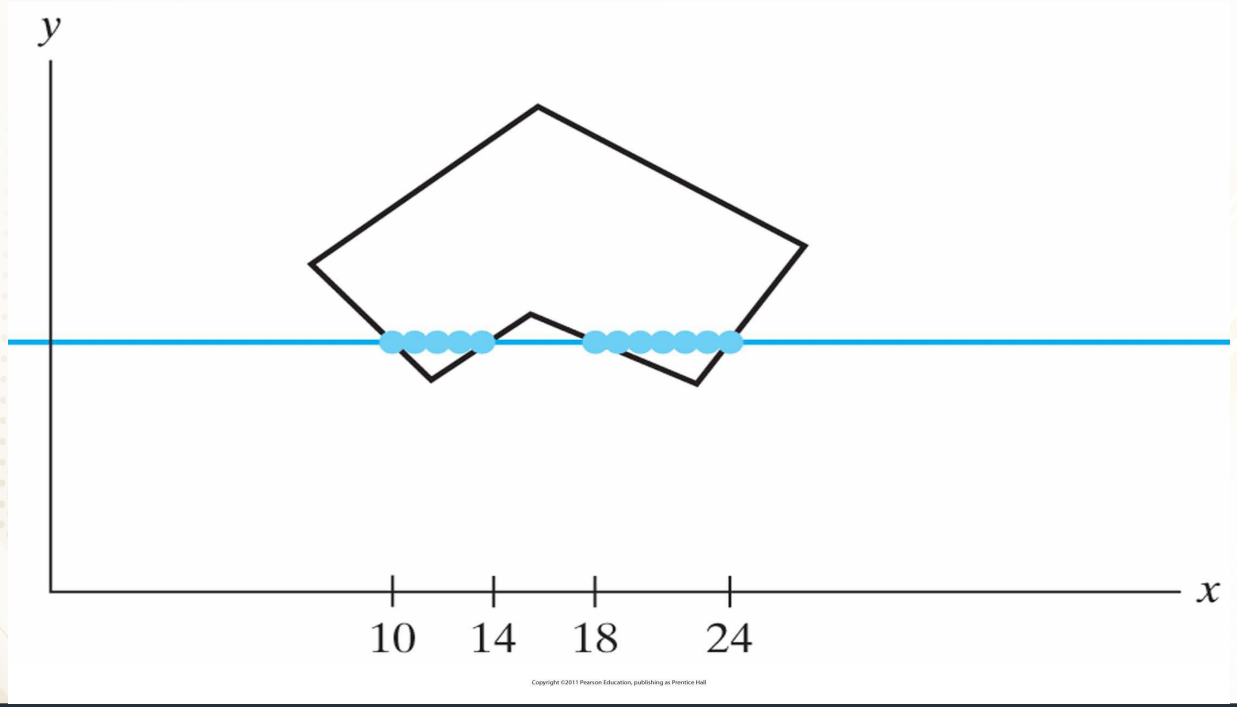
Determine the intersection positions of the boundaries.

For each scanline that crosses the polygon, edge-intersection are sorted from left to right, then pixel positions, b/w including each intersection pair is filled.

Solving pair of simultaneous linear equations.

General scan line polygon fill algorithm

Fig: Interior pixels along a scan line passing through a polygon fill area.



The fill color is applied to five pixels from x=10 to x=14 and seven pixels from x=18 to x=24.

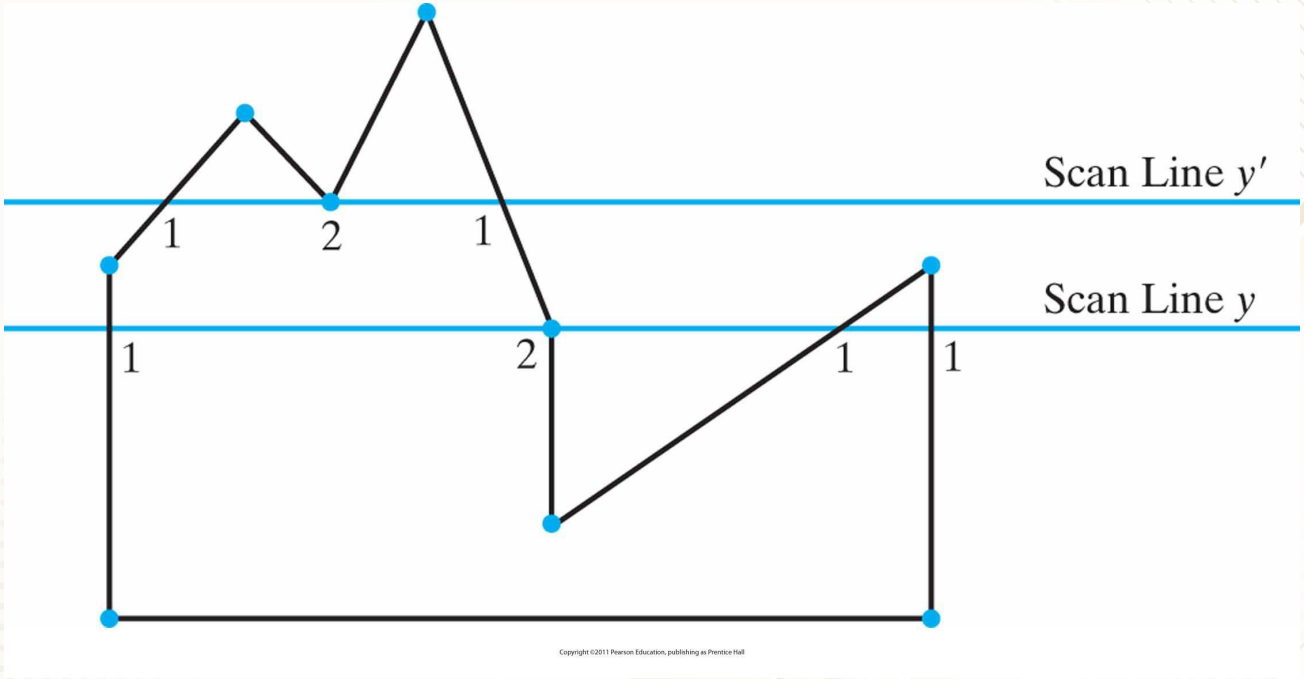
Algorithm is not quite simple.

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

Scan line y'- even number of edges – two pair correctly identify interior pixels

Scanline y- five edges. Here we must count vertex intersection as one point.

Figure : Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.



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

For scan line y, the two intersecting edges are both above the scan line.

vertex that has adjoining edges on **opposite sides** of an intersecting scan line should be **counted as just one.**

Trace around clockwise or counter clockwise and observing the relative changes in y values.

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 the two edge intersections

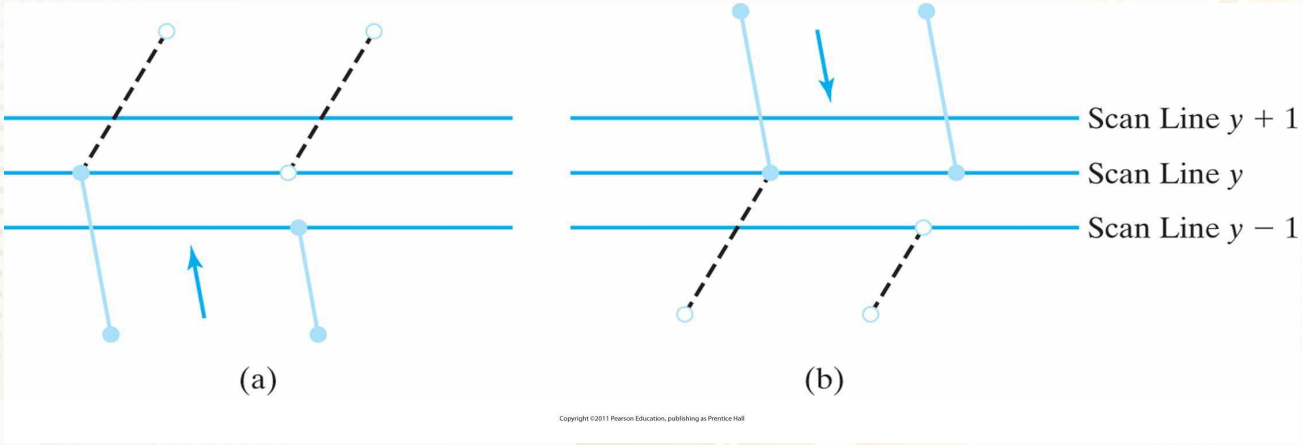
Adjustment to vertex intersection count

shorten some polygon edges to split those vertices that should be counted as one intersection.

edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint y values.

If so, the lower edge can be shortened to ensure that only one intersection point is generated

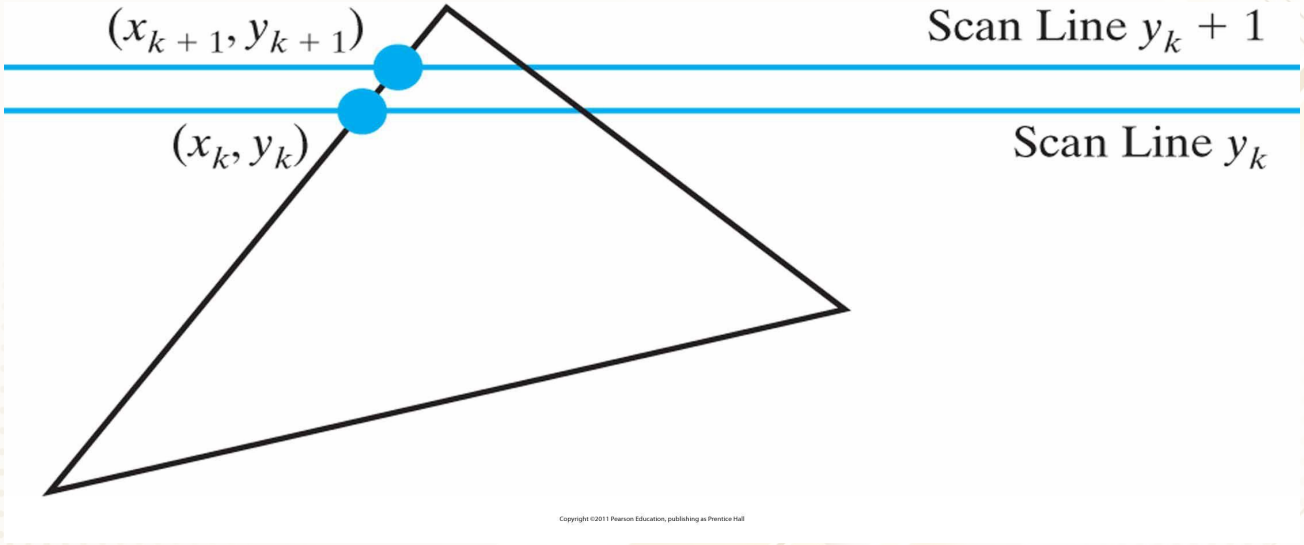
Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.



When the endpoint y coordinates of the two edges are increasing, the y value of the upper end point for the current edge is decreased by 1.

When the endpoint y values are monotonically decreasing, we decrease the y coordinate of the upper endpoint of the edge following the current edge.

Figure 6-49 Two successive scan lines intersecting a polygon boundary.



Slope of the edge is constant from one scanline to other scan line.

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

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \tag{59}$$

Because the change in y coordinates between the two scan lines is simply

$$y_{k+1} - y_k = 1 \tag{60}$$

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

$$x_{k+1} = x_k + \frac{1}{m} \tag{61}$$

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 xk value for scan line k above the initial scan line can be calculated as

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

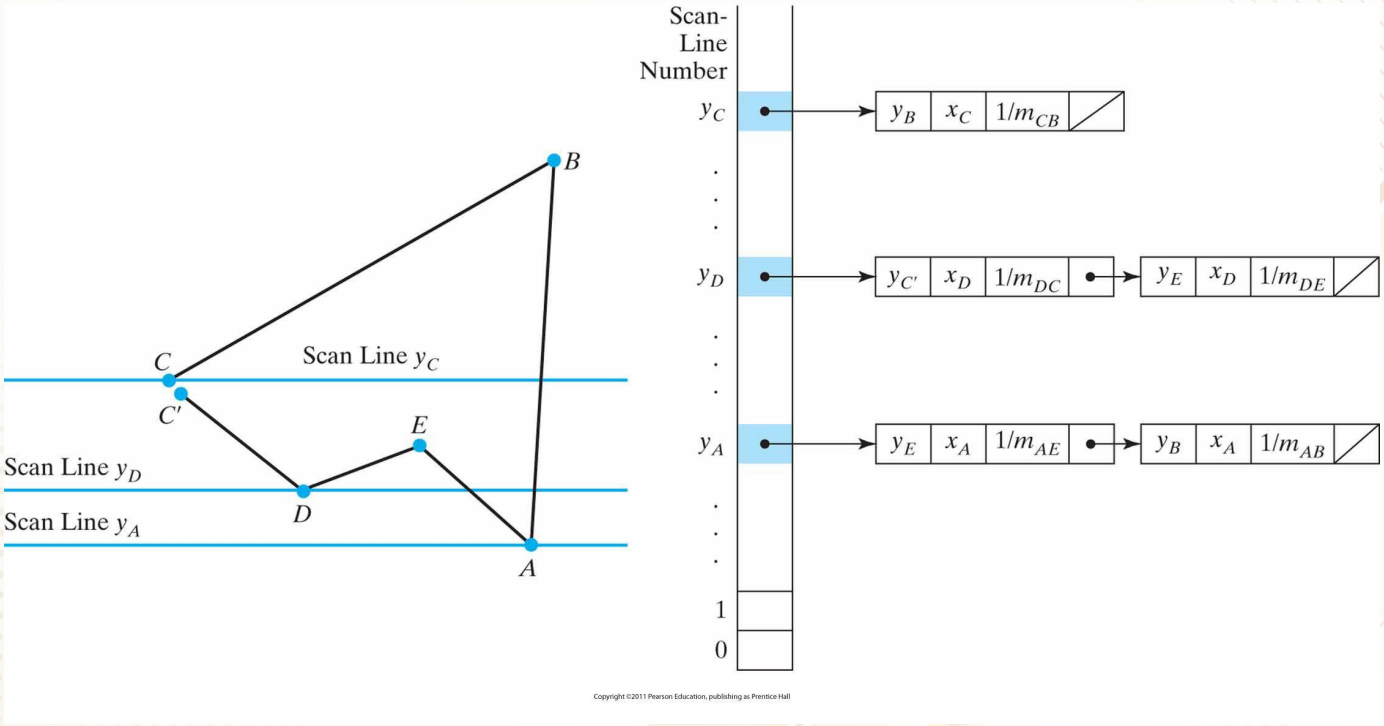
$$m = \frac{\Delta y}{\Delta x}$$

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

Sorted edge table

- Store polygon boundary
- Use bucket sort to store edges, sorted on the smallest y value of each edge
- Only nonhorizontal edges are entered into sorted table

Figure 6-50 A polygon and its sorted edge table, with edge DC shortened by one unit in the y direction.



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 scanline, the edges are in sorted order from left to right.

Active edge list

Process scan lines from bottom to top, produce active edge list for each scan line crossing the polygon boundaries.

Contains all edges crossed by that scan line with iterative coherence calculations for edge intersection

Pop-Up Menu

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Three Steps

- We must define actions corresponding to each entry in the menu.
- We must link the menu to particular mouse button.
- We must register a callback function for each menu

Geometric transformation

Operations applied to the geometric description of an object to change its position, orientation or size

Two-Dimensional Geometric Transformations

Basic Transformations

- Translation
- Rotation
- Scaling

Composite Transformations

Other transformations

- Reflection
- Shear

Basic two-dimensional geometric transformations

- **Two-Dimensional translation**
  - Translate a point by Adding offsets to coordinates to generate new coordinate position.
  - Move original point position along a straight line path to its new location.
  - Set t_x,t_y be the translation distance, we have

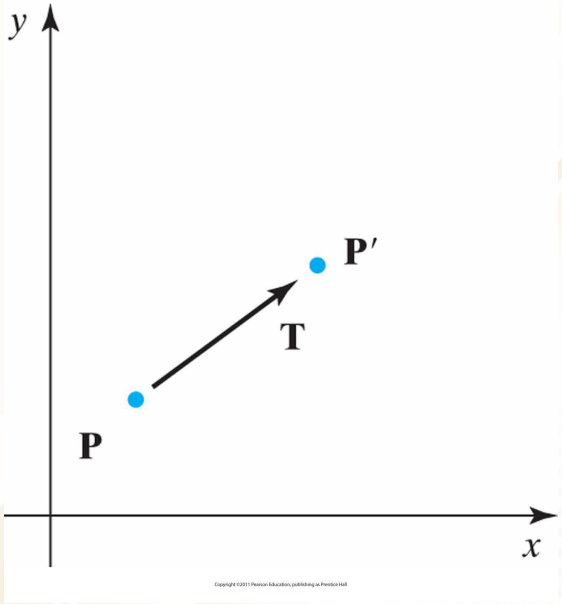
$$x' = x + t_x \qquad y' = y + t_y$$

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad P = \begin{bmatrix} x \\ y \end{bmatrix} \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$P' = P + T$$

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

Translating a point from position **P** to position **P'** using a translation vector **T**.

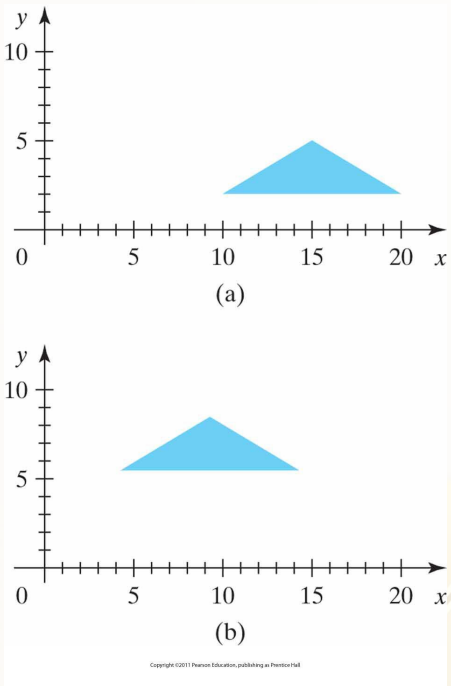


- Translation is a **rigid-body transformation** that moves objects without deformation.
- Every point on the object is translated by the same amount.

Polygon translation

- We add a translation vector to the coordinate position of each vertex and then regenerate the polygon using the new set of vertex coordinates.

Moving a polygon from position (a) to position (b) with the translation vector (−5.50, 3.75).



2D Rotation

Specify a rotation axis and a rotation angle.

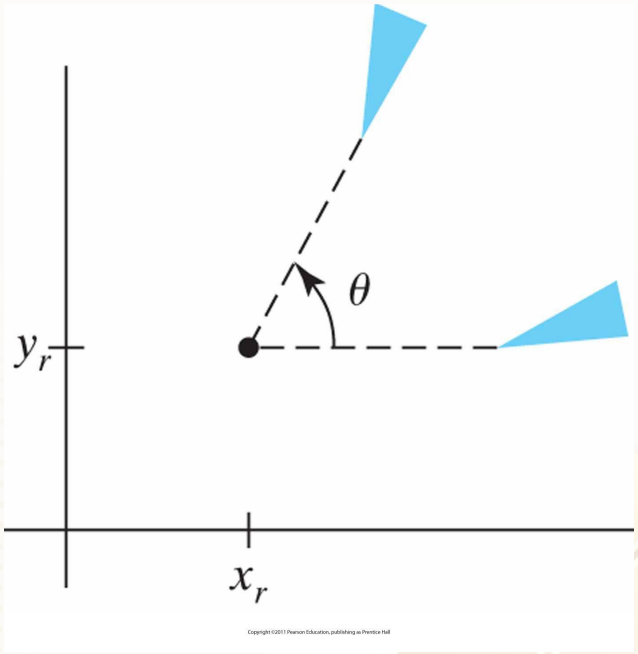
All points of the object are then transformed to new positions.

Repositioning the object along a circular path in the xy plane.

2D rotation- rotation about z axis.

2D Rotation

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

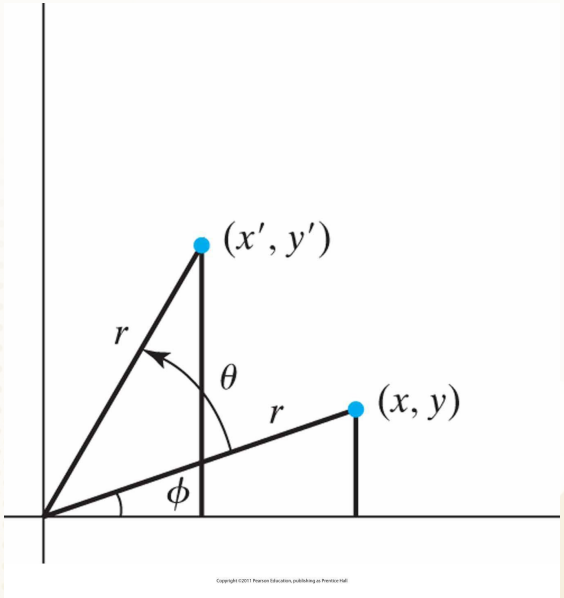


Parameters

- rotation angle θ
- 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 counter clockwise
- A negative value rotates objects in the clockwise direction.

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



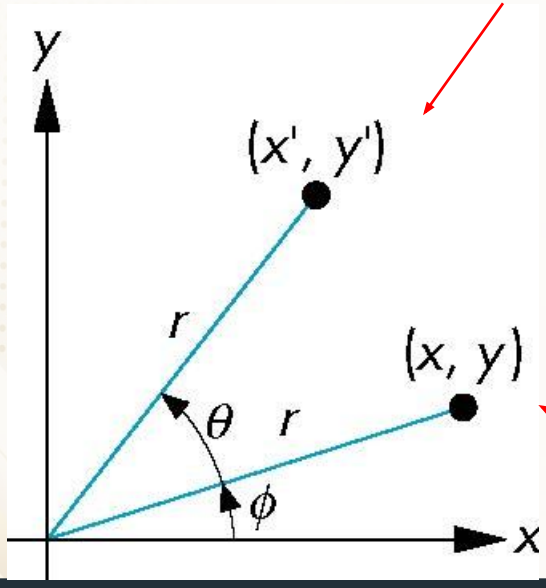
r is the constant distance of the point from the origin.
Φ- original angular position of the point from the horizontal.
Θ- Rotation angle

Rotation (2D)

Consider rotation about the origin by q degrees radius stays the same, angle increases by *q*

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

$$X' = r\cos(\varphi + \theta)$$
$$Y' = r\sin(\varphi + \theta)$$



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

$x = r\cos$

φ

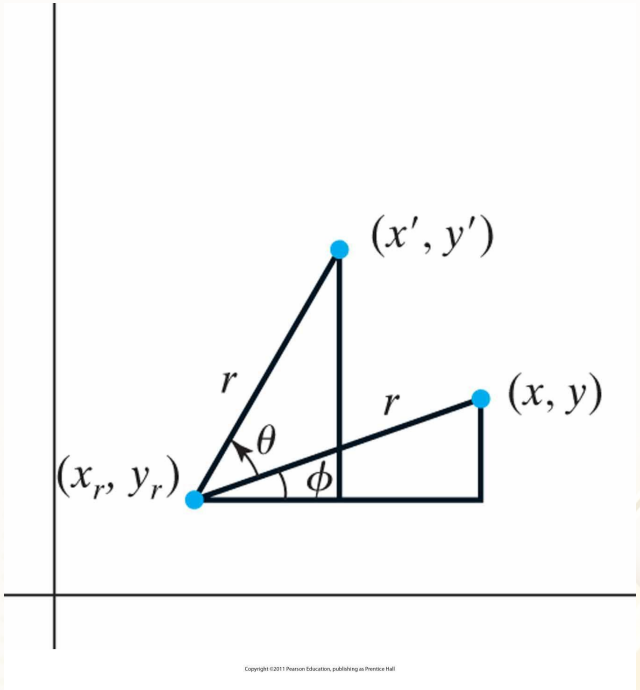
$y = r\sin\varphi$

These equations can be written in matrix form as

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

$$P' = R.P$$

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



Rotation

Translations, rotations are **rigid-body transformations** that move objects without deformation.

Every point on an object is rotated through the same angle.

A polygon is rotated by displacing each vertex using the specified rotation angle and then regenerating the polygon using the new vertices.

2D Scaling

- To alter the size of an object by multiplying the coordinates with scaling factor s_x and s_y

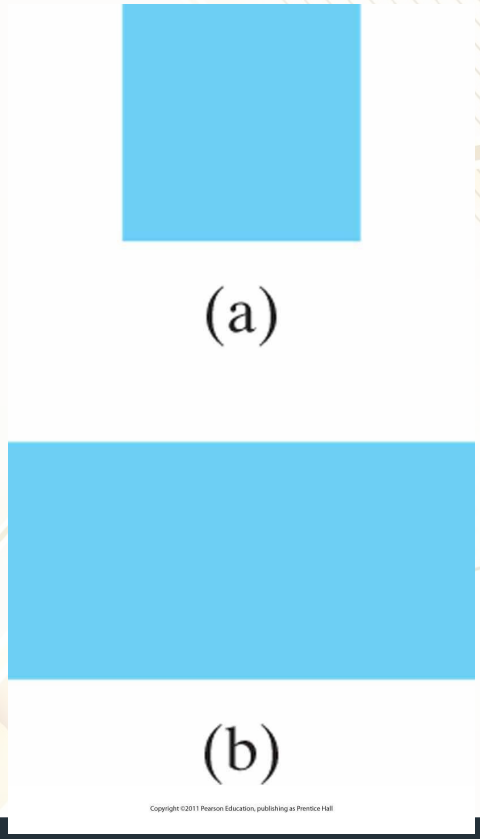
$$x' = x \cdot s_x \qquad y' = y \cdot s_y$$

  - **In matrix format, where S is a 2 by 2 scaling matrix**

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

- scaling factor s_x scales an object in the x direction and s_y scales in the y direction
- Values less than 1 reduce the size of the objects.
- Greater than 1 produce enlargements.
- Value of 1- unchanged object

Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$



Uniform scaling

- s_x and s_y are assigned the same value.
- maintains relative object proportions.

Differential scaling

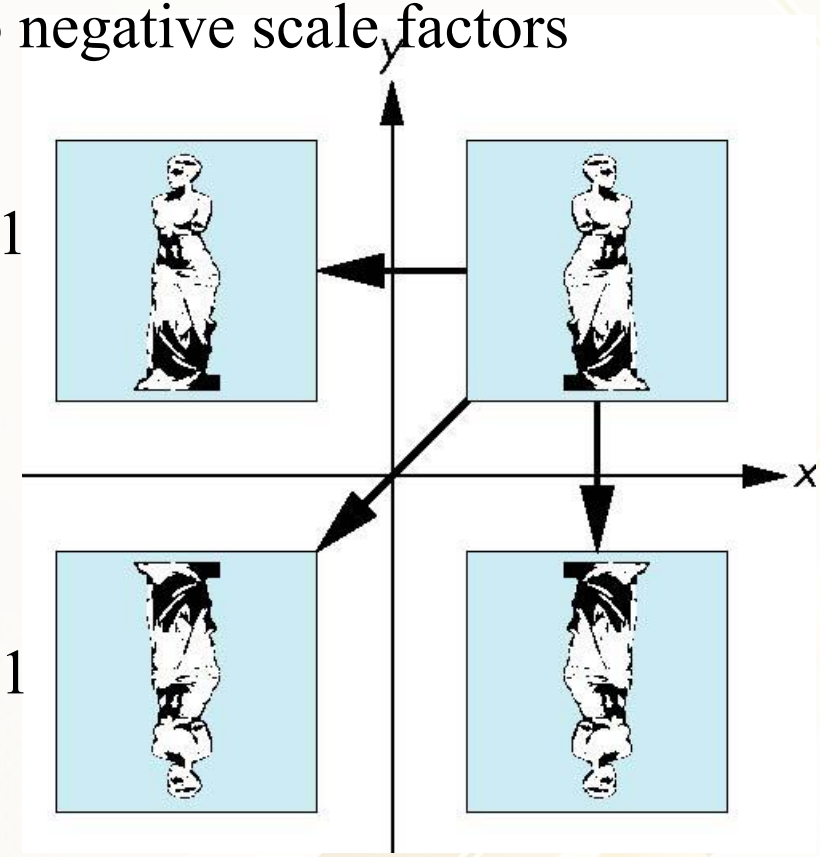
- Unequal values for sx and sy. often used in design applications.

- negative values - reflects it about one or more of the coordinate axes.

Reflection

corresponds to negative scale factors

$s_x = -1$ $s_y = 1$



original

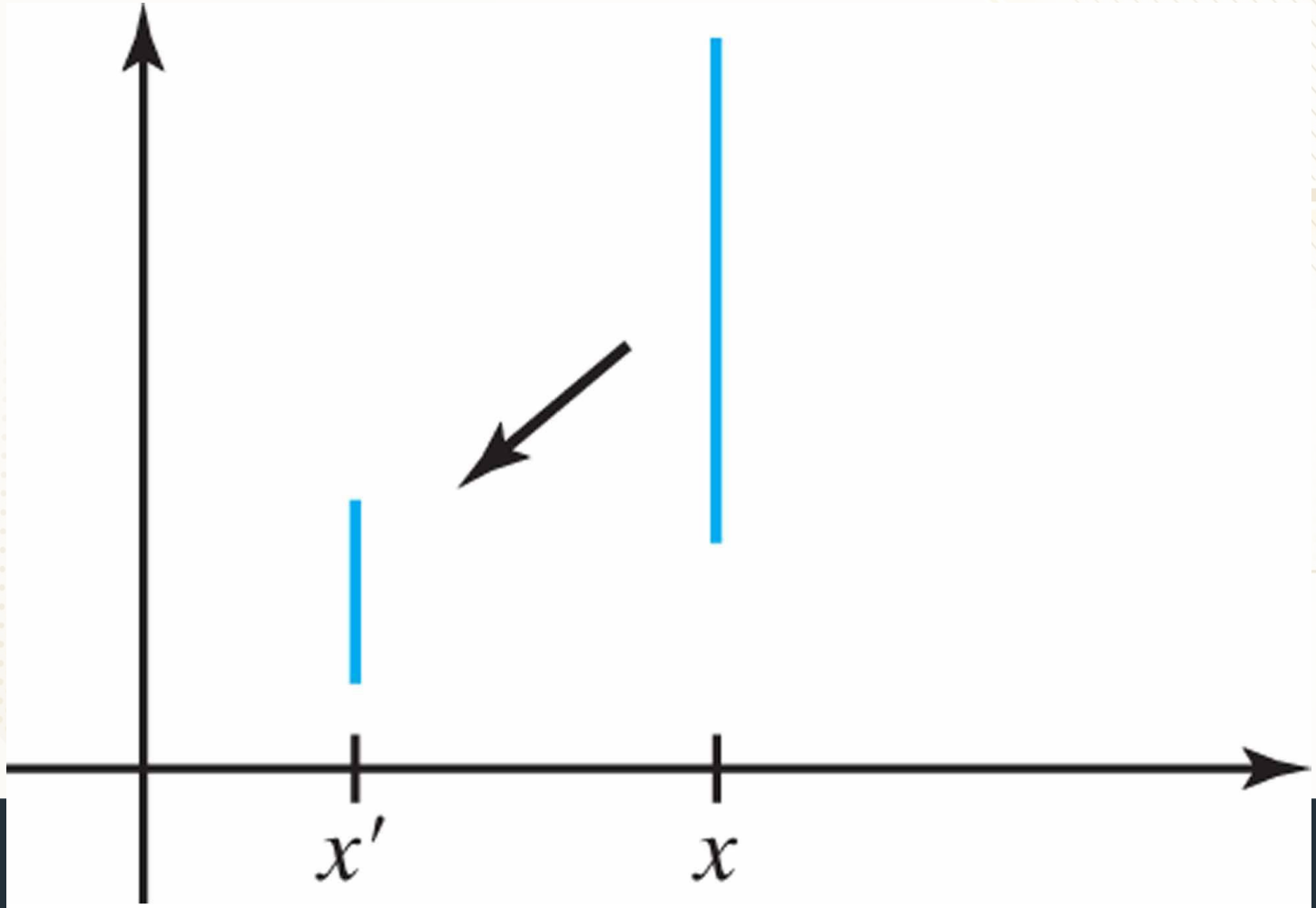
$s_x = -1$ $s_y = -1$

$s_x = 1$ $s_y = -1$

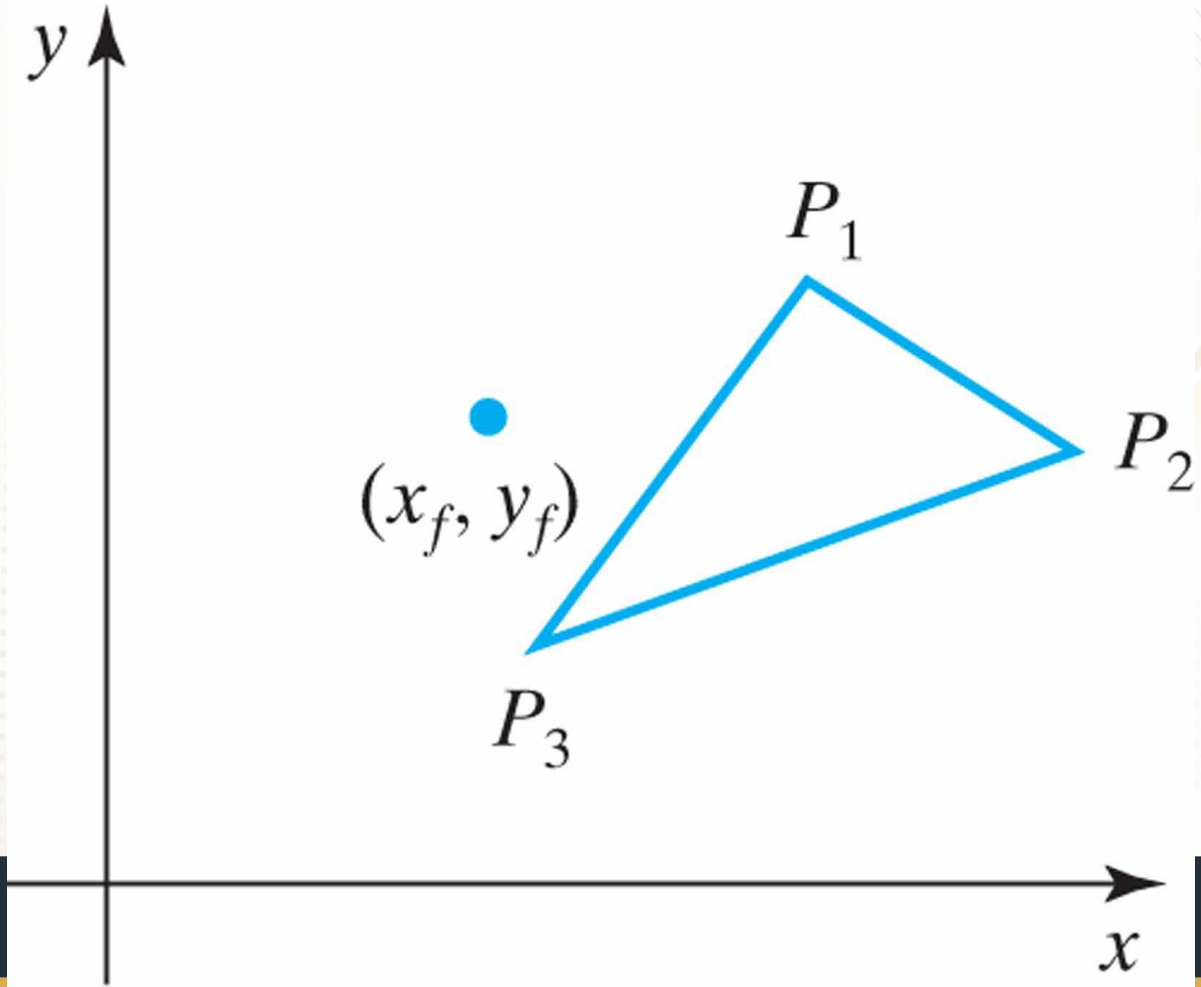
Scaling factors with absolute values less than 1 move objects closer to the coordinate origin.

Absolute values greater than 1 move coordinate positions farther from the origin.

A line scaled with Equation using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.



Scaling relative to a chosen fixed point (x_f, y_f). The distance from each polygon vertex to the fixed point is scaled by



- 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.
- Choosing a fix point (x_f, y_f) as its centroid to perform scaling

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

Matrix representations and Homogeneous Coordinates

- Three basic transformation can be expressed in the general matrix form

$$\mathbf{P' = M_1 . P + M_2}$$

Coordinate positions P and P' represented as column vectors.

M_1 is a 2 by 2 array containing multiplicative factors

M_2 is two element column matrix containing translation terms.

For translation, M_1 is 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 translation terms for 2D geometric transformation can be combined into single matrix if we expand the representations to 3 by 3 matrices.

Use third column of a transformation matrix for translation terms, and all transformation equations can be expressed as matrix multiplication.

Expand 2D coordinate positions to three element column matrix.

Homogeneous Coordinates

- **A point** (x, y) **can be re-written in** homogeneous coordinates **as** (x_h, y_h, h)
- **The** homogeneous parameter h **is a non- zero value such that:**

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

- We can then write homogeneous coordinate representation as (hx, hy, h)
- We can conveniently choose $h = 1$ so that (x, y) becomes $(x, y, 1)$

Homogeneous coordinates allows to represent all geometric transformation equations as matrix multiplications.

Std method used in graphics systems

2D Translation matrix

Using a homogeneous-coordinate approach, we can represent the equations for a 2D translation of a coordinate position using the following matrix multiplication:

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

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

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

2D 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

2D scaling Matrix

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 $S(s_x,s_y)$ is the 3 ×3matrix

Inverse Transformation

We obtain the inverse matrix by negating the translation distances.

if we have two-dimensional translation distances t_x and t_y, the inverse translation matrix is

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

Product of a translation matrix and its inverse produces the identity matrix.

Negative values for rotation angles generate rotations in a clockwise direction

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

Inverse matrix can also be obtained by interchanging rows and columns.

Calculate the inverse of any rotation matrix R by evaluating its transpose ($R^{-1} = R^{T}$).

For two-dimensional scaling with parameters s_x and s_y applied relative to the coordinate origin, the inverse transformation matrix is

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

2D Composite transformation

- we can set up a sequence of transformations as a composite transformation matrix by calculating the product of the individual transformations.

Concatenation, or composition of matrices.

- Forming products of transformation matrices
- First multiply the transformation matrices to form a single composite matrix.
- If we want to apply two transformations to point position P, the transformed location would be calculated as

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

Composite 2D Translation

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

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

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

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

which demonstrates that **two successive translations are additive**

Composite 2D 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:**

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

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

Composite 2D Scalings

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

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

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

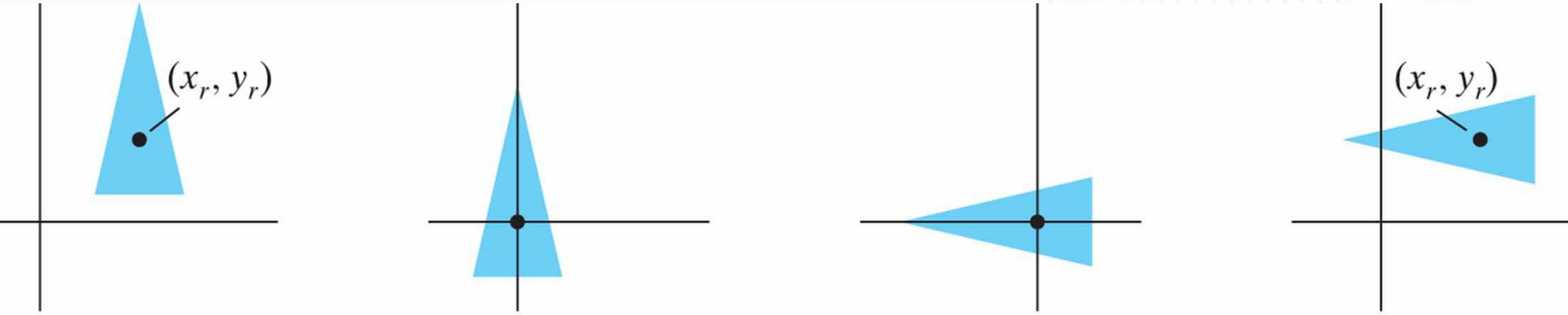
The resulting matrix in this case indicates that successive scaling operations are multiplicative

2D Composite transformation
General two dimensional pivot point Rotation

2D rotation about any other pivot point (xr, yr)

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.

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



(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)

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

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

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

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

$M= T(x_r,y_r).R(\theta).T(-x_r, -y_r) =R(x_r,y_r,\theta)$

$P'=M.P$

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

- A square in two dimensional system is specified by its vertices (6,6), (10,6), (10,10) and (6,10). Implement the following by first finding composite transformation matrix for the sequence of transformations involved. Sketch the original and transformed square.

  i) Rotate the square by 45 degree about its vertex(6,6)

  ii) Scale the original square by a factor of 2 about its center.

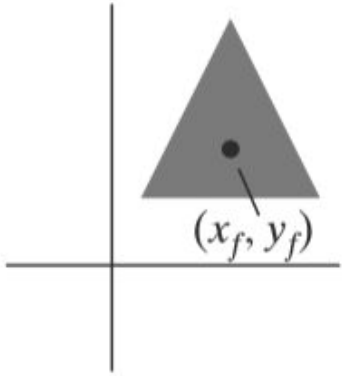
General 2D Fixed point Scalings

1.Translate the object so that the fixed point coincides with the coordinate origin.

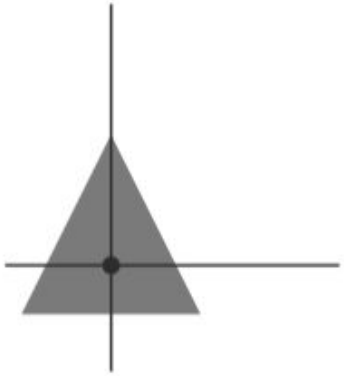
2.Scale the object with respect to the coordinate origin.

3. Use the inverse of the translation in step (1) to return the object to its original position.

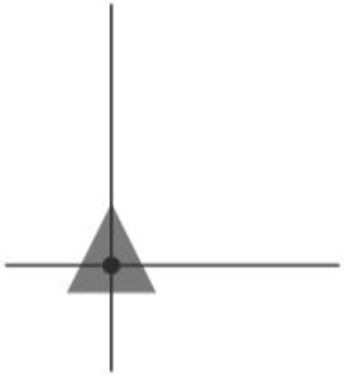
General 2D Fixed point Scalings



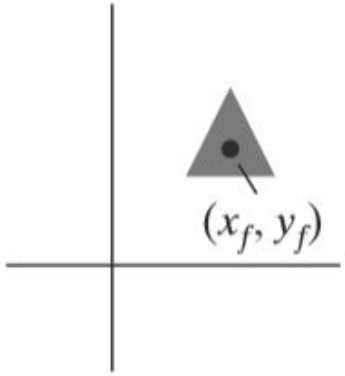
(a)
Original Position of Object and Fixed Point



(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)

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

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

A square in a two dimensional system is specified by its vertices (6, 6), (10, 6), (10, 10) and (6, 10). Implement the following by first finding a composite transformation matrix for the sequence of transformations involved. Sketch the original and transformed square.

i) Rotate the square by 45^0 about its vertex (6, 6)

ii) Scale the original square by a factor of 2 about its center. **(10 Marks)**

Matrix Concatenation Properties

Multiplication of matrices is associative.

For any three matrices,M1,M2,and M3, the matrix product M3 ·M2 ·M1 can be performed by first multiplying M3 and M2 or by first multiplying M2 and M1:

$$\mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1 = (\mathbf{M}_3 \cdot \mathbf{M}_2) \cdot \mathbf{M}_1 = \mathbf{M}_3 \cdot (\mathbf{M}_2 \cdot \mathbf{M}_1)$$

Construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying).

- The last transformation invoked (which is M1 for this example) is the first to be applied, and the first transformation that is called (M3 in this case) is the last to be applied.
- Transformation products may not be commutative.
  - The matrix product M2·M1 is not equal to M1·M2,
  - If we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated.

Transformation products may not be commutative

Be careful about the order in which the composite matrix is evaluated.

Except for some special cases:

- Two successive rotations
- Two successive translations
- Two successive scalings
- rotation and uniform scaling

OpenGL geometric transformation functions

Separate functions for basic geometric transformations.

Are specified in 3D.

4 X 4 matrix is applied to the coordinates of objects.

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, either f (float) or d (double).
- For 2D, we set t_z=0.0;
- 2D position is represented as a four-element column matrix with the z component equal to 0.0.
- The translation matrix generated by this function is used to transform positions of objects defined after this function is invoked.

- **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.

  - **glRotatef (90.0, 0.0, 0.0, 1.0);**
  - sets up the matrix for a 90◦ rotation about the z axis

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

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

The suffix code is again either f or d,and the scaling parameters can be assigned any real-number values.

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

OpenGL matrix Operations

Matrices are part of the state.

glMatrixMode- select matrix to which the operations apply.

- Projection mode
- Modelview mode
- Texture mode
- Color mode

Projection mode

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Projection mode

- This transformation determines how a scene is to be projected onto the screen.

Modelview

- Set up a matrix for the geometric transformations.
- It is used to store and combine the geometric transformations.
- It is also used to combine the geometric transformations with the transformation to a viewing-coordinate system.
- **glMatrixMode (GL_MODELVIEW);**
- Which designates the 4×4 model view matrix as the current matrix.

The OpenGL transformation routines are used to modify the modelview matrix, which is applied to transform coordinate postions in a scene.

So it is important to use glMatrixMode to change to the modelview matrix before applying geometric transformations.

Texture matrix

mapping texture patterns to surfaces

color matrix

convert from one color model to another.

The default argument for the glMatrixMode function is GL MODELVIEW.

Once we are in the model view mode a call to a transformation routine generates a matrix that is multiplied by the current matrix for that mode.

In addition, we can assign values to the elements of the current matrix, and there are two functions in the OpenGL library for this purpose.

With the following function, we assign the identity matrix to the current matrix:

```
glLoadIdentity ( );
```

- we can assign other values to the elements of the current matrix using
  - **glLoadMatrix* (elements16);**

The elements in this array must be specified in column- major order.

```
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);

The current matrix is postmultiplied by the matrix specified in glMultMatrix, and this product replaces the current matrix

M=M.M’

```
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:

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

The first transformation to be applied in this sequence is the last one specified in the code.

OpenGL Matrix stack

For each of the four modes, we can select with the glMatrixMode function, OpenGL maintains a matrix stack.

Initially, each stack contains only the identity matrix.

Top matrix- current matrix

Model view Stack depth- atleast 32

glGetIntegerv **(GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);**

Which returns a single integer value to array stackSize

Other matrix mode stack depth - 2

OpenGL symbolic constants:

- GL_MAX_ PROJECTION_STACK_DEPTH, GL_MAX TEXTURE_STACK_DEPTH, GL_MAX_COLOR_STACK_DEPTH.
- Find out how many matrices are currently in the stack with
- glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);

```
glMatrixMode (GL_MODELVIEW);
glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);
glLoadIdentity();
glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0);
glRecti (50, 100, 200, 150);
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);
glLoadIdentity();
 glScalef (-0.5, 1.0, 1.0);
glRecti (50, 100, 200, 150);
```

- **glPushMatrix ( );** copy the current matrix at the top of the active stack
- **glPopMatrix ( );-** destroys the matrix at the top of the stack

push and pop the current matrix stack

```
glMatrixMode (GL_MODELVIEW);
glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);
glPushMatrix ( );
 glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0);
glRecti (50, 100, 200, 150);
glPopMatrix ( );
glPushMatrix ( );
glRotatef (90.0, 0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150); glPopMatrix ( );glPushMatrix ( );
 glScalef (-0.5, 1.0, 1.0);
glRecti (50, 100, 200, 150);
```

Figure 7-34 Translating a rectangle using the OpenGL function glTranslatef (−200.0, −50.0, 0.0).

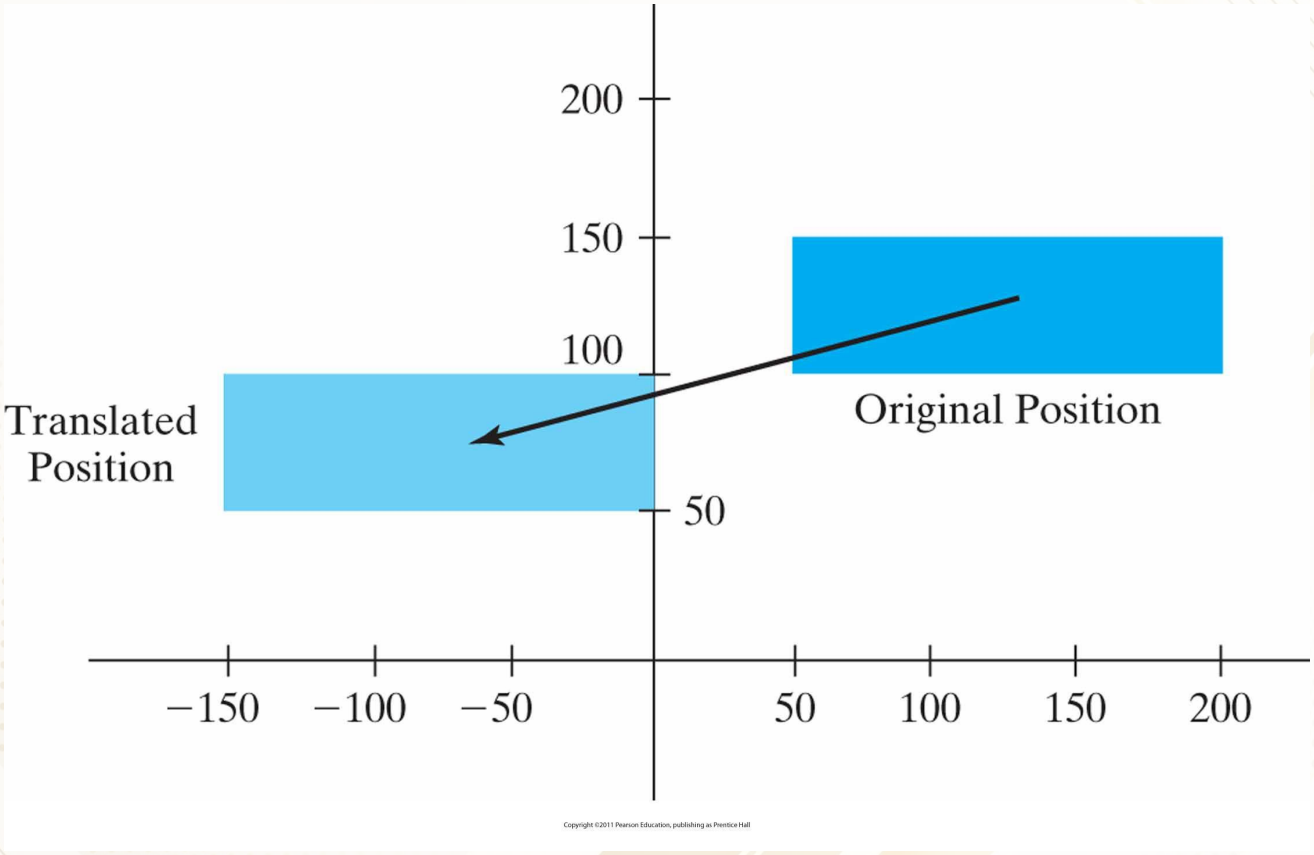


Figure 7-35 Rotating a rectangle about the *z* axis using the OpenGL function glRotatef (90.0, 0.0, 0.0, 1.0).

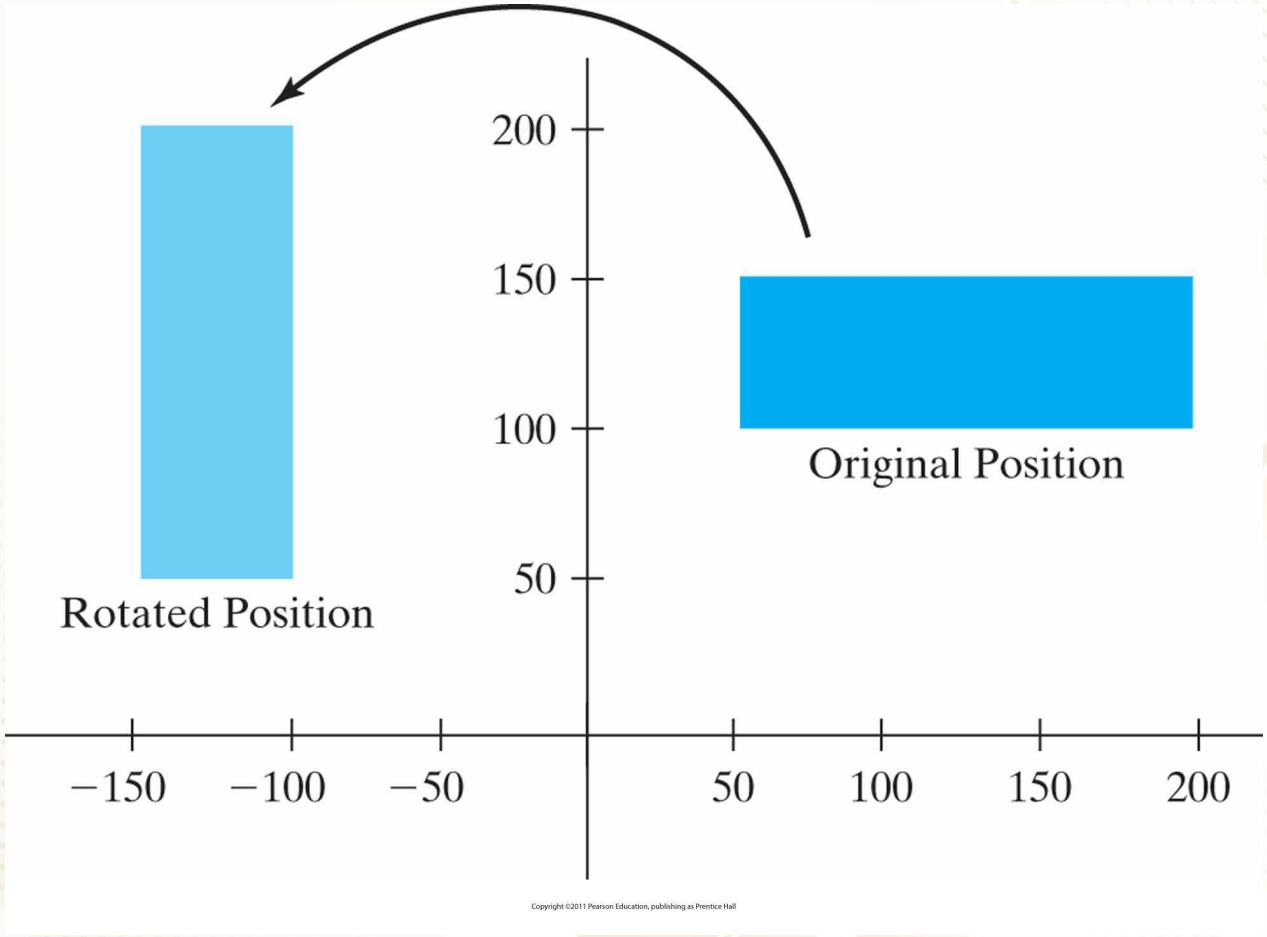


Figure 7-36 Scaling and reflecting a rectangle using the OpenGL function glScalef (−0.5, 1.0, 1.0).

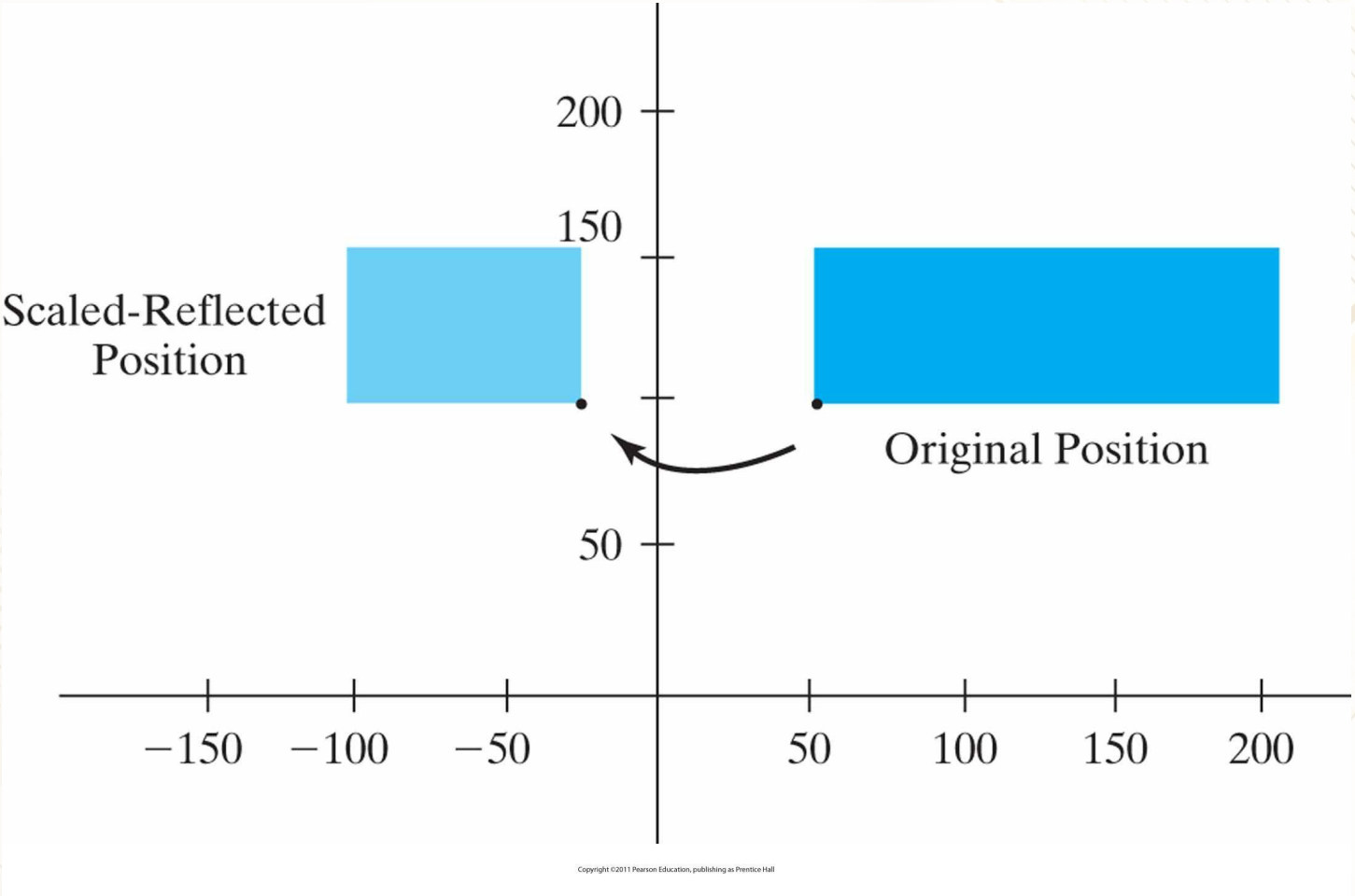


TABLE 7-1

T Summary of OpenGL Geometric Transformation Functions

T

Function	Description
`glTranslate*`	Specifies translation parameters.
`glRotate*`	Specifies parameters for rotation about any axis through the origin.
`glScale*`	Specifies scaling parameters with respect to coordinate origin.
`glMatrixMode`	Specifies current matrix for geometric-viewing transformations, projection transformations, texture transformations, or color transformations.
`glLoadIdentity`	Sets current matrix to identity.
`glLoadMatrix*` (elems);	Sets elements of current matrix.
`glMultMatrix*` (elems);	Postmultiplies the current matrix by the specified matrix.
`glPixelZoom`	Specifies two-dimensional scaling parameters for raster operations.

The 2D Viewing Pipeline

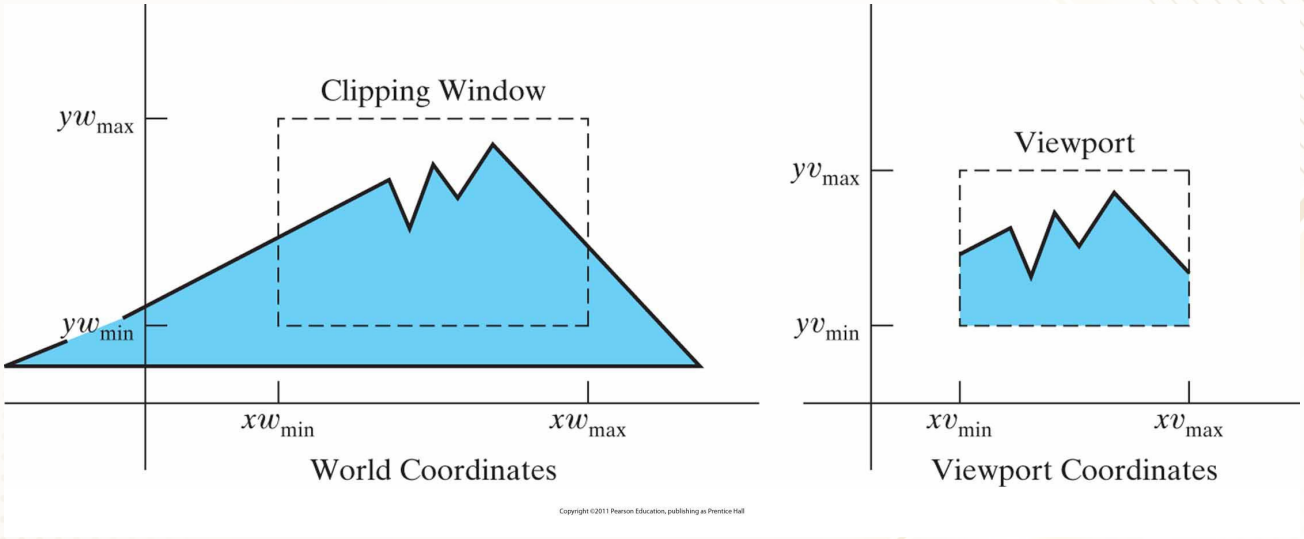
A graphics package allows the user to specify

a) Which part of the defined picture is to be displayed.

b) Where the part is to be placed on the display device.

Much like what we see in real life through a small window or the view finder of a camera.

Figure 8-1 A clipping window and associated viewport, specified as rectangles aligned with the coordinate axes.



Clipping Window

A section of a two-dimensional scene that is selected for display.

All parts of the scene outside the selected section are "**clipped" off.**

The only part of the scene that shows up on the screen is what is inside the clipping window.

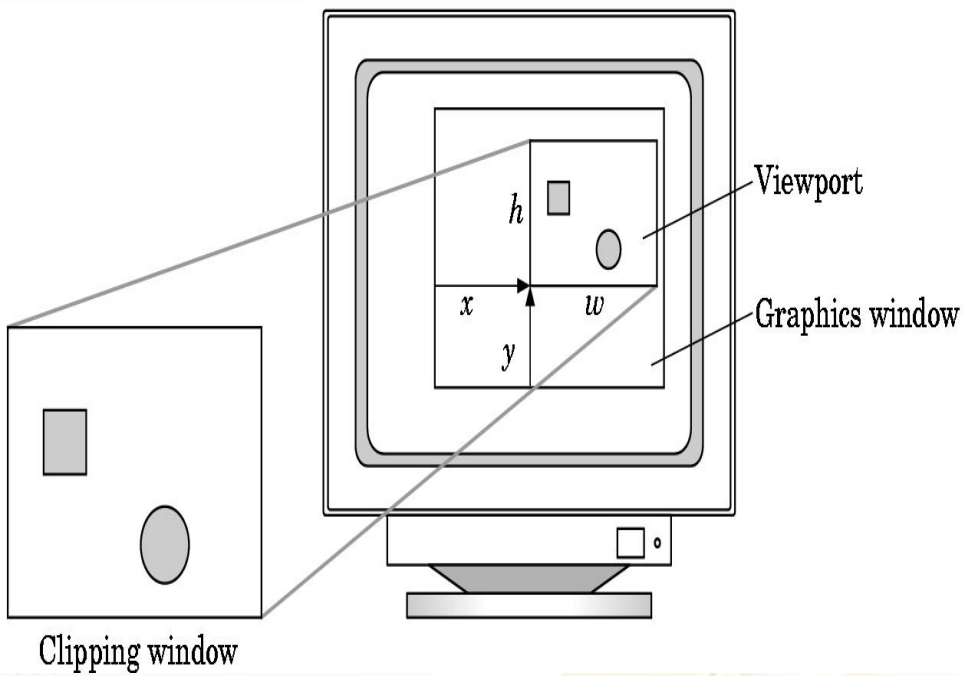
world window or the viewing window

selected section of a scene that is eventually converted to pixel patterns within a display window on the video monitor.

Viewport

- placement within the display window using another "window" called the viewport.
- Objects inside the clipping window are mapped to the viewport.
- The clipping window selects **what we want to see;**
- Viewport indicates **where it is to be viewed** on the output device.

A mapping to the viewport

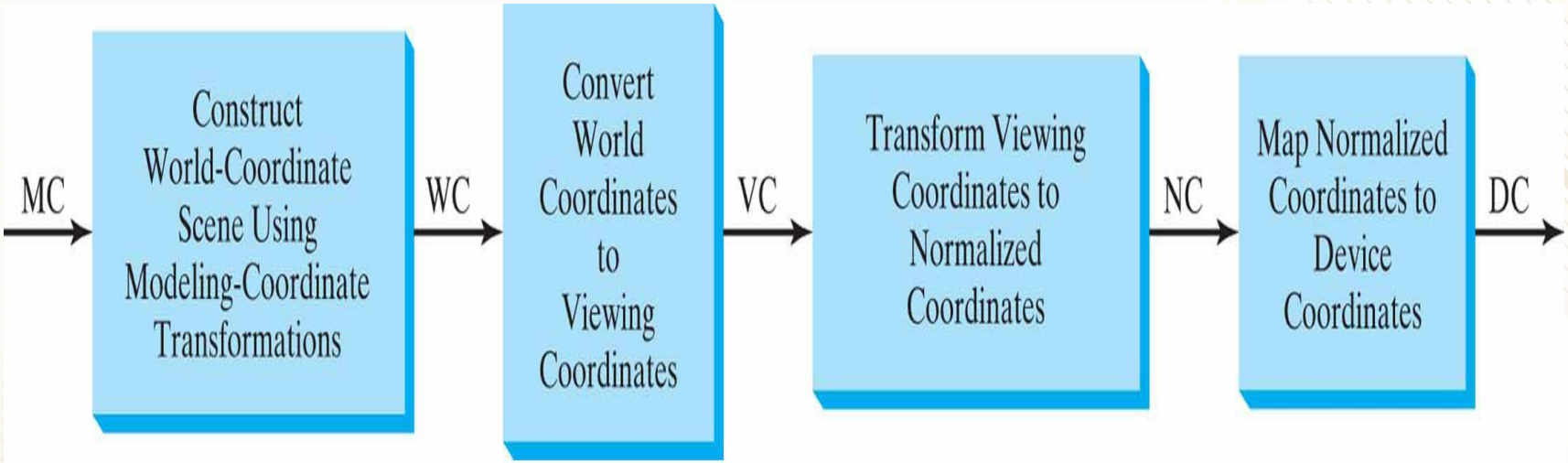


2D viewing transformation or window to viewport transformation or windowing transformation

The mapping of a 2D, world-coordinate scene description to device coordinates is called a two-dimensional viewing transformation.

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

Figure 8-2 Two-dimensional viewing-transformation pipeline



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

Viewing coordinates for two-dimensional applications are the same as world coordinates.

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.

Normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1.

Viewport is defined either in normalized coordinates or in screen coordinates after the normalization process.

OpenGL 2D VIEWING FUNCTIONS

OpenGL projection Mode

- Establish the appropriate mode for constructing the matrix to transform from world coordinates to screen coordinates
- To define clipping windows and viewports, first go to projection mode.
- glMatrixMode(GL_PROJECTION);
- use glLoadIdentity() to initialize the matrix.

GLU Clipping window Function

- A 2D clipping window can be defined with

gluOrtho2D(xwmin, xwmax, ywmin, ywmax)

- If we do not specify a clipping window in an application program, the default coordinates are (xwmin, ywmin) = (−1.0,−1.0) and (xwmax, ywmax) = (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 functions

- To specify the viewport parameters, do

glViewport(xvmin, yvmin, vpWidth, vpHeight)

with parameters given in integer screen coordinates.

- The first two give the position of the lower left corner relative to the lower left corner of the display window.
- By default, the viewport is the entire window.
- You can create multiple viewports within a window.
- **xvmax = xvmin +vpWidth, yvmax = yvmin +vpHeight**

- To get the parameters for the active viewport, use

  glGetIntegerv(GL_VIEWPORT, vpArray);

  Where vpArray is a 4 element integer array.

  - The contents of vpArray after the function returns will be the same as the arguments given to glViewport in the same order.

Creating a GLUT DISPLAY WINDOW

- **glutInitWindowPosition (xTopLeft, yTopLeft);**

  –the integer parameters are relative to the top-left corner of the screen
- **glutInitWindowSize (dwWidth, dwHeight);**
- **glutCreateWindow ("Title of Display Window");**
- Default position is (−1,−1) which allows the window system to choose the position.
- Default size is 300 pixels square.
- glutCreateWindow returns an integer id for the window. You only need this if you are displaying more than one window.

Setting the GLUT display-window Mode and color

- glutInitDisplayMode (mode)

  -to set number of buffer, color mode, depth mode etc.

glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB)

- glClearColor (red, green, blue, alpha)

  –Specify the background color

glClearIndex(index)- set display window color

GLUT Display-Window Identifier

- Multiple display windows may be created and each is assigned with a positive integer **display window identifier, starts with 1**

windowID = glutCreateWindow("Window1");

Deleting a GLUT display window

- glutDestroyWindow (windowID) // to destroy the window

Current GLUT Display Window

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);**
- Query the system to determine which window is the current display window:
- **currentWindowID= glutGetWindow();**
- Value 0 means no display window or destroyed

Relocating and resizing a GLUT Display window

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

glutPositionWindow (xNewTopLeft, yNewTopLeft);

Where the coordinates specify the new position for the upper-left display-window corner, relative to the upper-left corner of the screen.

Resets the size of the current display window:

glutReshapeWindow (dwNewWidth, dwNewHeight);

Expand the current display window to full the screen:

glutFullScreen ( );

glutReshapeFunc(winReshape);

Managing Multiple GLUT Display Windows

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

Convert the current display window to an icon in the form of a small picture or symbol representing the window:

```
glutIconifyWindow ( );
```

Change the icon title

```
glutSetIconTitle ("Icon Name");
```

Change the name of the display window

```
glutSetWindowTitle ("Icon Name");
```

Some windows may overlap or totally obscure other display windows.

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 ( );
```

"push"the current display window to the back so that it is behind all other display windows.

```
glutSetWindow (windowID);
glutPushWindow ( );
```

We can also take the current window off the screen with

```
glutHideWindow ( );
```

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

Partitioning display windows called subwindow

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

- The parameters are given relative to the lower left corner of the display window specified by windowID.
- Sub-windows also have window ids and we can create sub-windows within sub-windows.

Selecting a display window screen cursor shape

glutsetCursor(shape)

Assign symbolic constant GLUT_CURSOR_UP_DOWN, GLUT_CURSOR_WAIT- wristshape
GLUT_CURSOR_DESTROY -crossbone

Viewing graphics objects in a GLUT Display Window

- Display callback functions are called only when GLUT determines that the display content should be renewed.

  -glutDisplayFunction (Display);

- To update the display manually call: glutPostRedisplay();

  glutIdleFunc (functionName) could be used in animations.

Table 8-1 Summary of OpenGL Two-D

RV UNIVERSITY
Go, change the world
an initiative of RV EDUCATIONAL INSTITUTIONS

TABLE 8-1

Summary of OpenGL Two-Dimensional Viewing Functions

Function	Description
`gluOrtho2D`	Specifies clipping-window coordinates as parameters for a two-dimensional orthogonal projection.
`glViewport`	Specifies screen-coordinate parameters for a viewport.
`glGetIntegerv`	Uses arguments `GL_VIEWPORT` and `vpArray` to obtain parameters for the currently active viewport.
`glutInit`	Initializes the GLUT library.
`glutInitWindowPosition`	Specifies coordinates for the top-left corner of a display window.
`glutInitWindowSize`	Specifies width and height for a display window.
`glutCreateWindow`	Creates a display window (which is assigned an integer identifier) and specify a display-window title.
`glutInitDisplayMode`	Selects parameters such as buffering and color mode for a display window.
`glClearColor`	Specifies a background RGB color for a display window.
`glClearIndex`	Specifies a background color for a display window using color-index mode.

Table 8-1 (continued) Summary of OpenGL Two-Dimensional Viewing Functions

`glutDestroyWindow`	Specifies an identifier number for a display window that is to be deleted.
`glutSetWindow`	Specifies the identifier number for a display window that is to be the current display window.
`glutPositionWindow`	Resets the screen location for the current display window.
`glutReshapeWindow`	Resets the width and height for the current display window.
`glutFullScreen`	Sets current display window to the size of the video screen.
`glutReshapeFunc`	Specifies a function that is to be invoked when display-window size is changed.
`glutIconifyWindow`	Converts the current display window to an icon.
`glutSetIconTitle`	Specifies a label for a display-window icon.
`glutSetWindowTitle`	Specifies new title for the current display window.
`glutPopWindow`	Moves current display window to the "top"; i.e., in front of all other windows.

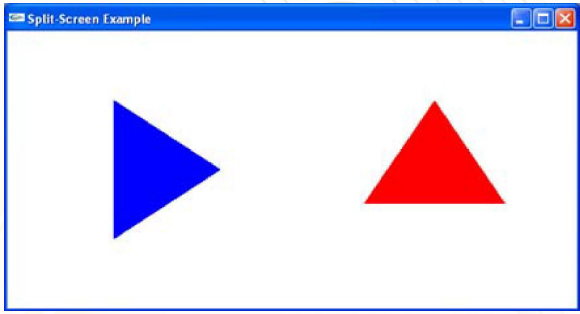
Table 8-1 (continued) Summary of OpenGL Two-Dimensional Viewing Functions

`glutPushWindow`	Moves current display window to the "bottom"; i.e., behind all other windows.
`glutShowWindow`	Returns the current display window to the screen.
`glutCreateSubWindow`	Creates a second-level window within a display window.
`glutSetCursor`	Selects a shape for the screen cursor.
`glutDisplayFunc`	Invokes a function to create a picture within the current display window.
`glutPostRedisplay`	Renews the contents of the current display window.
`glutMainLoop`	Executes the computer-graphics program.
`glutIdleFunc`	Specifies a function to execute when the system is idle.
`glutGet`	Queries the system about a specified state parameter.

OpenGL 2D VIEWING EXAMPLE

- 2 Viewports
- One triangle is displayed in two colors and orientations in 2 viewports
- glClear(GL_COLOR_BUFFER_BIT);

```
glColor3f(0.0, 0.0, 1.0);
glViewport(0, 0, 300, 300);
drawCenteredTriangle();
glColor3f(1.0, 0.0, 0.0);
glViewport(300, 0, 300, 300);
glRotatef(90.0, 0.0, 0.0, 1.0);
drawCenteredTriangle();
```



OpenGL 2D Viewing Functions

- glMatrixMode(GL_PROJECTION);
- use glLoadIdentity() to initialize the matrix.
- **gluOrtho2D(xwmin, xwmax, ywmin, ywmax)- clipping window**
- **glViewport(xvmin, yvmin, vpWidth, vpHeight)**
- glGetIntegerv(GL_VIEWPORT, vpArray);

GLUT Display-Window Identifier

- windowID = glutCreateWindow(“Window1”);
- glutDestroyWindow (windowID) // to destroy the window
- glutSetWindow (windowID);
- currentWindowID= glutGetWindow();
- glutPositionWindow (xNewTopLeft, yNewTopLeft);
- glutReshapeWindow (dwNewWidth, dwNewHeight);

glutFullScreen ( );

glutReshapeFunc(winReshape);

Convert the current display window to an icon

- **glutIconifyWindow ( );**

- Change the icon title
  - glutSetIconTitle ("Icon Name");
- Change the name of the display window
  - glutSetWindowTitle ("Icon Name");

Choose any display window to be in front of all other windows

- glutSetWindow (windowID);
- glutPopWindow ( );

“push”the current display window to the back so that it is behind all other display windows.

- glutSetWindow (windowID);
- glutPushWindow ( );

- glutHideWindow ( );
- glutShowWindow ( );

Partitioning display windows called subwindow

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

  glutsetCursor(shape)

  - Assign symbolic constant GLUT_CURSOR_UP_DOWN, GLUT_CURSOR_WAIT- wristshape GLUT_CURSOR_DESTROY -crossbone

Clipping Algorithms

- **Clipping** is a procedure that identifies those portions of a picture that are either inside or outside of a specified region of space.

- **Clip Window** is the region against which an object is to be clipped.
- It may be a polygon or curved boundaries. But

rectangular clip regions are preferred.

Types of clipping

Based on different output primitives

- Point clipping
- Line clipping
- Fill- Area clipping(polygons)
- Curve clipping
- Text clipping

POINT CLIPPING

- A point p=(x,y) is saved for display, if the following inequalities are satisfied,

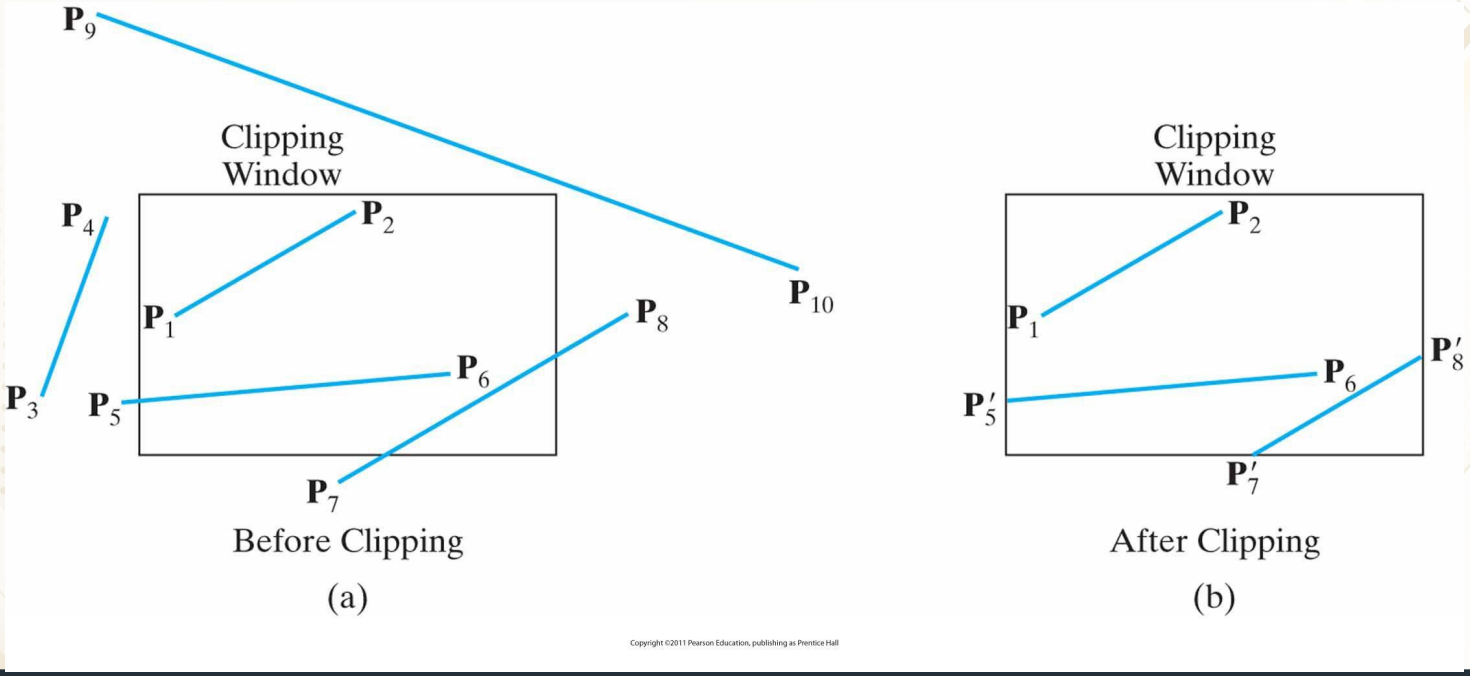
$$xw_{min} \le x \le xw_{max}$$
$$yw_{min} \le y \le yw_{max}$$

If any one of these 4 inequalities is not satisfied, the point is clipped i.e, not saved for display.

Can be applied to scenes involving **explosions or sea foam or cloud or smoke** that are modeled with particles (points) distributed in some region of the scene.

Line Clipping

processes each line in a scene through a series of tests and intersection calculations to determine entire line or part of it is to be saved



2 D Line Clipping

- Expensive part-calculating intersection positions

Goal

- Minimize intersection calculations
- Test whether a line segment is completely inside the clipping window or completely outside
  - Completely inside – easy; both points are inside – P1P2
  - Completely outside – difficult; both points outside any one of the four boundaries – P3P4
- If not possible – Test whether any part of a line crosses the window interior

Line Clipping

- Use the parametric equation of line

$$x = x_0 + u(x_{end} - x_0)$$
$$y = y_0 + u(y_{end} - y_0) \qquad 0 \le u \le 1$$

to determine intersections by assigning the coordinate value for that edge to either *x* or *y* and solving for parameter *u*.

- If the value of *u* is outside the range from 0 to 1
  - line segment does not intersect that window border line
  - Else part of the line is inside the border
- Eg: boundary is *xwmin,* so substitute this value for *x*, solve for *u*, and calculate corresponding *y*-intersection value
- Process inside portion of the line segment against the other clipping boundaries
  - until the section inside the window is found

Line Clipping – Cohen-Sutherland

- One of the earliest algorithms for fast line clipping
- Processing time is reduced by performing more tests before proceeding to the intersection calculations
- Every line endpoint in a picture is assigned a four-bit binary value – **region code or out code**
  - each bit position is used to indicate whether the point is inside or outside one of the clipping-window boundaries
  - 1 (or *true)*=> endpoint is outside that window
  - 0 (or *false)* => endpoint is not outside (it is inside or on) the corresponding window edge

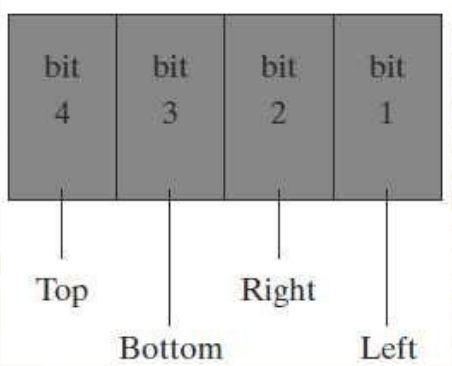
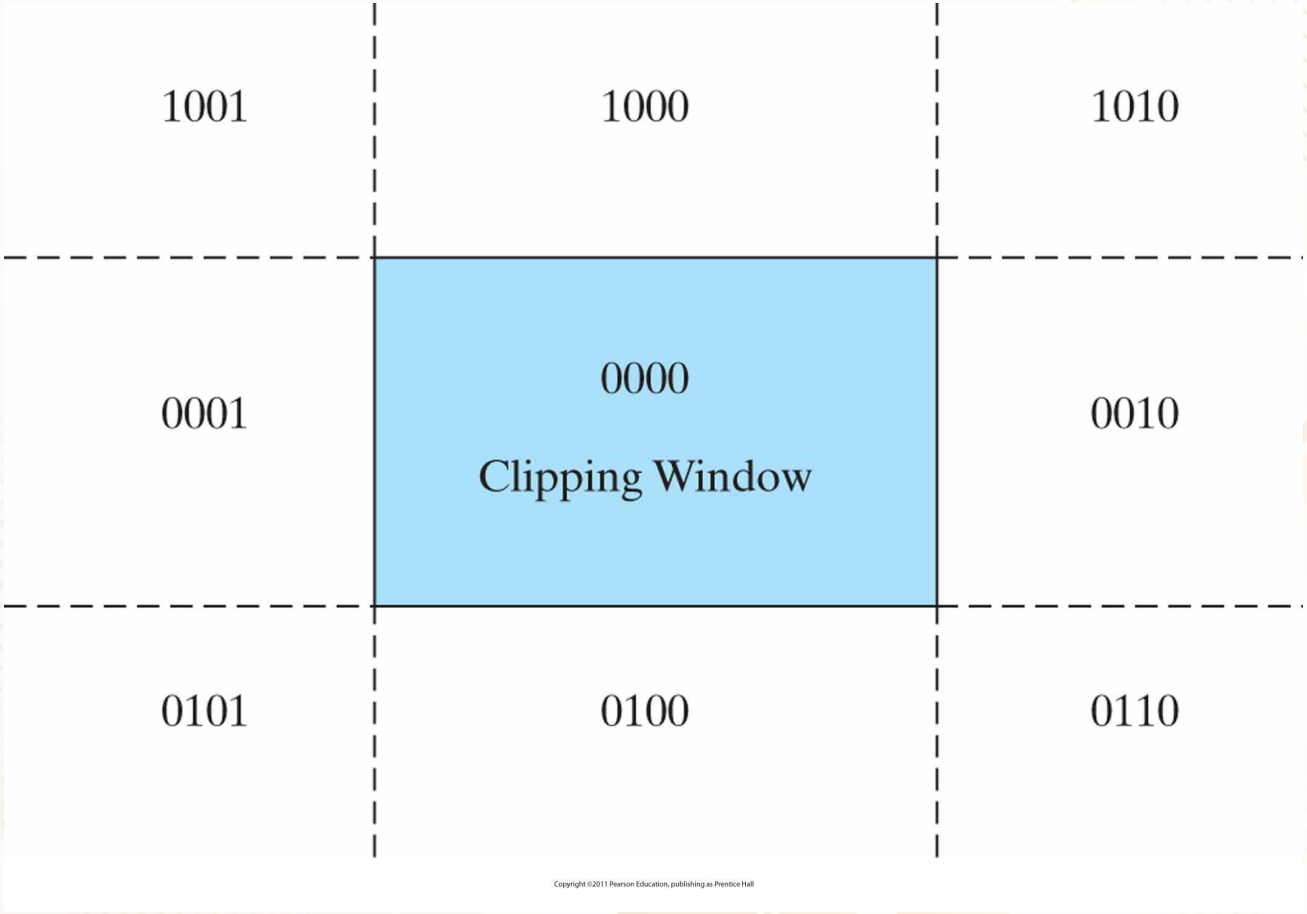


Figure 8-11 The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.



Line Clipping – Cohen-Sutherland

- Bit values in a region code determined by comparing the coordinate values (x, y) endpoints to the clipping boundaries
  - Bit 1 is set to 1 if $x < xwmin$
  - Bit 2 is set to 1 if $x > xwmax$
  - Bit 3 is set to 1 if $y < ywmin$
  - Bit 4 is set to 1 if $y > ywmax$

More efficient

  - Calculate differences between endpoint coordinates and clipping boundaries.
  - Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

Line Clipping – Cohen-Sutherland

- Perform Inside –outside test using logical operators

Case1: (o1=o2=0000)

- When ***or*** operation between two endpoints region code is false(0000), then line is inside, save the line (p1p2)

Case 2 ($.o_1 \& o_2 \neq 0$)

When ***and*** operation between two endpoints region code is true(not 0000), then line is completely outside, eliminate the line.(p3p4)

Case 3 .(o1≠0,o2=0 or vice versa)

one end point is inside the clipping window.one is outside. the line segment must be shortened.(p5p6)

Case 4 o_1&o_2≠0 (p9p10)

both endpoints are outside ,but they are on the outside of different edges of the window. Intersect with one of the sides of the window and to check the outcode of the resulting point.

Several intersection calculations might be necessary to clip a line segment, depending on the order in which we process the clipping boundaries.

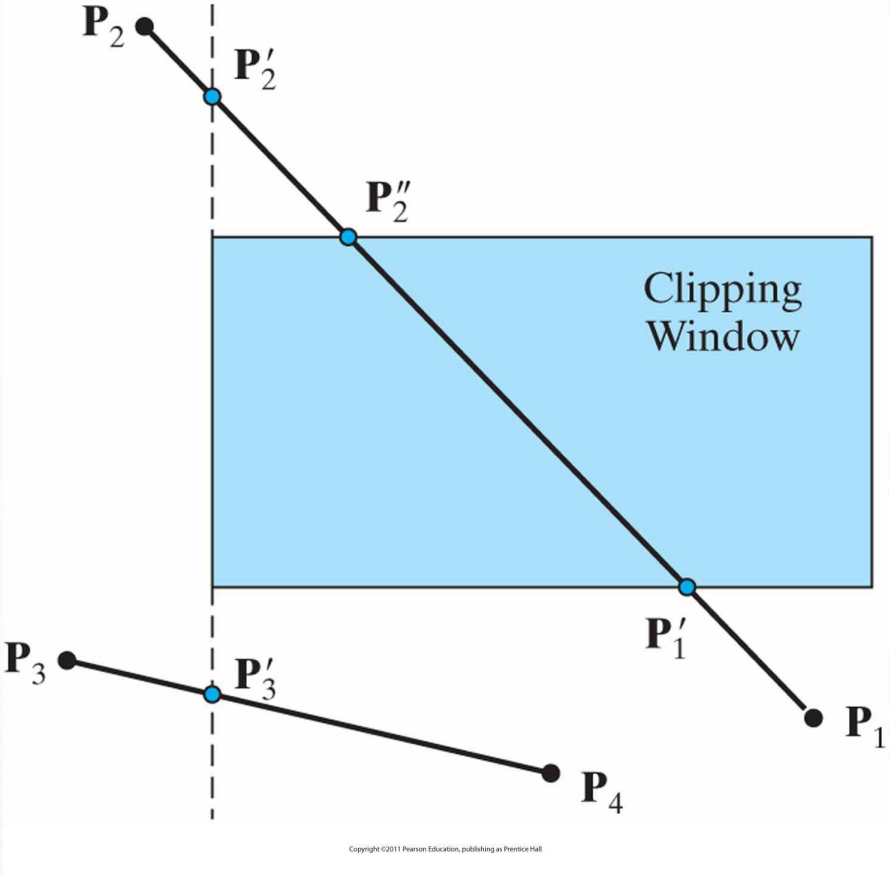
We continue eliminating sections until either the line is totally clipped or the remaining part of the line is inside the clipping window.

Window edges are processed in the following order: left, right, bottom, top.

To determine whether a line crosses a selected clipping boundary, we can check corresponding bit values in the two endpoint region codes.

If one of these bit values is 1 and the other is 0, the line segment crosses that boundary.

Figure 8-12 Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.



- Region codes for the line from P1 to P2 are 0100 and 1001.
- P1 is inside the left clipping boundary and P2 is outside that boundary. We then calculate the intersection position P’2, and we clip off the line section from P2 to P’ 2.

- The remaining portion of the line is inside the right border line, and so we next check the bottom border. Endpoint P1 is below the bottom clipping edge and P’2 is above it, so we determine the intersection position at this boundary(P’1).
- We eliminate the line section from P1 to P’1 .There we determine the intersection position to be P’’2.
- The final step is to clip off the section above the top boundary and save the interior segment from P’1 toP’’2

Line Clipping – Cohen-Sutherland

- To determine a boundary intersections, use the slope intercept form of the line equation
- For a line with endpoint coordinates (*x0, y0)* and (*xend, yend),*
  - For intersection point with a vertical clipping border line, *y*- coordinate is obtained with the calculation-

$$y = y_0 + m(x - x_0) \qquad m = (y_{end} - y_0)/(x_{end} - x_0)$$

where the *x* value is set to either *xwmin* or *xwmax*

  - Similarly, for intersection with a horizontal border, the *x*-coordinate is calculated as

$$x = x_0 + \frac{y - y_0}{m}$$

with *y* set either to *ywmin* or to *ywmax*