S J P N Trust's

Hirasugar Institute of Technology, Nidasoshi.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka, Affiliated to VTU Belagavi.

Accredited at "A" Grade by NAAC and Recognized Under Section 2(f) of UGC Act, 1956.

Module-2

18CS62

Fill area Primitives, 2D Geometric Transformations and 2D viewing

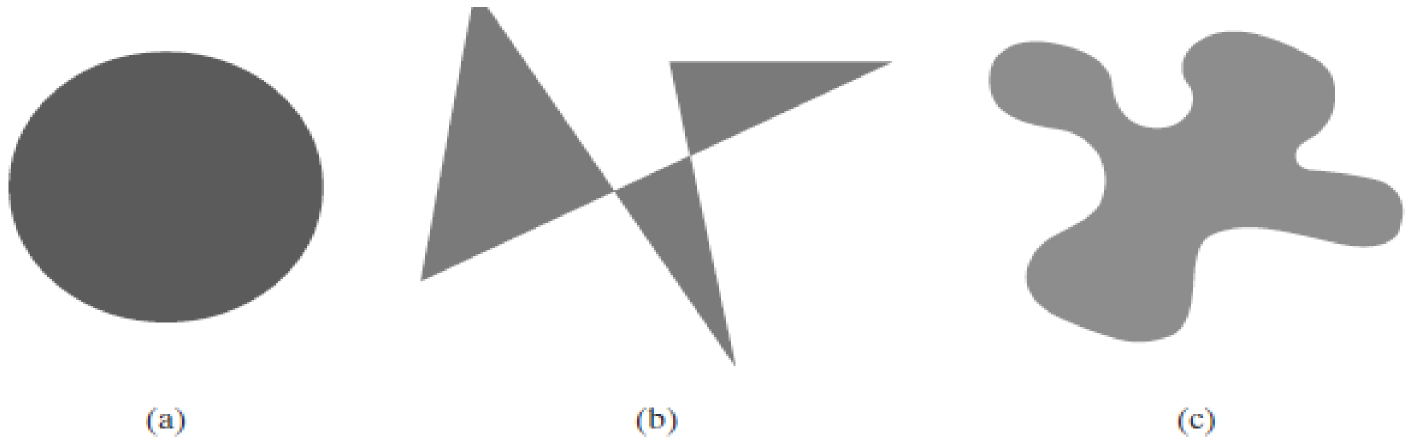
Prof. Rahul Palakar

Fill-Area Primitives

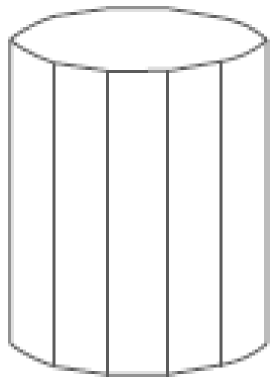
Another useful construct, besides points, straight-line segments, and curves, for describing components of a picture is an area that is filled with some solid color or pattern.

A picture component of this type is typically referred to as a **fill area** or a **filled area.**

- are used to describe surfaces of solid objects.
- fill regions are usually planar surfaces, mainly polygons.
- in general, there are many possible shapes for a region in a picture that we might wish to fill with a color option



- graphics libraries generally do not support specifications for arbitrary fill shapes.
- Most library routines require that a fill area be specified as a polygon.
- most curved surfaces can be approximated reasonably well with a set of polygon patches.
- Approximating a curved surface with polygon facets is sometimes referred to as *surface tessellation*, or fitting the surface with a *polygon mesh*.



- Objects described with a set of polygon surface patches are usually referred to as **standard graphics objects**, or just **graphics objects**.
- In general, we can create fill areas with any boundary specification, such as a circle or connected set of spline-curve sections.

Polygon Fill Areas

Mathematically defined, 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.

- a polygon must have all its vertices within a single plane and there can be no edge crossings.

Ex: triangles, rectangles, octagons, and decagons.

- any plane figure with a closed-polyline boundary is called as a polygon, and one with no crossing edges is referred to as a *standard polygon* or a *simple polygon*.

- For a computer-graphics application, it is possible that a designated set of polygon vertices do not all lie exactly in one plane.
- One way to rectify this problem is simply to divide the specified surface mesh into triangles.
- methods have been devised for approximating a nonplanar polygonal shape with a plane figure.

Polygon Classifications

Polygon are classified in to two types:

1. Convex polygon.
2. Concave polygon.

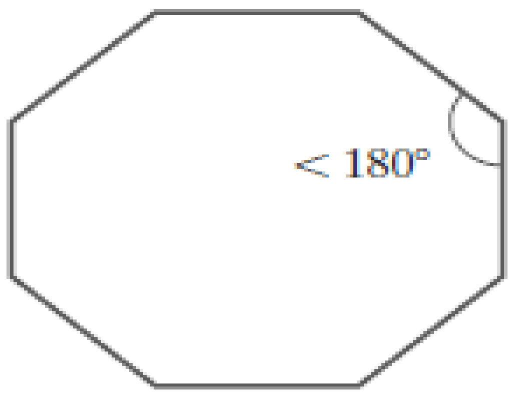
Convex polygon: (identification convex polygon)

1. If all interior angles of a polygon are less than or equal to 180°, the polygon is **convex.**

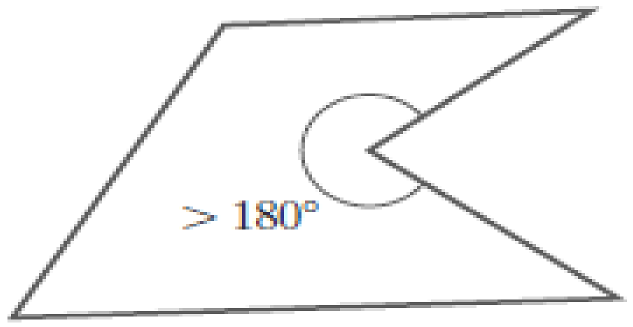
2. Convex polygon is that its interior lies completely on one side of the infinite extension line of any one of its edges.

3. 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.



(a)

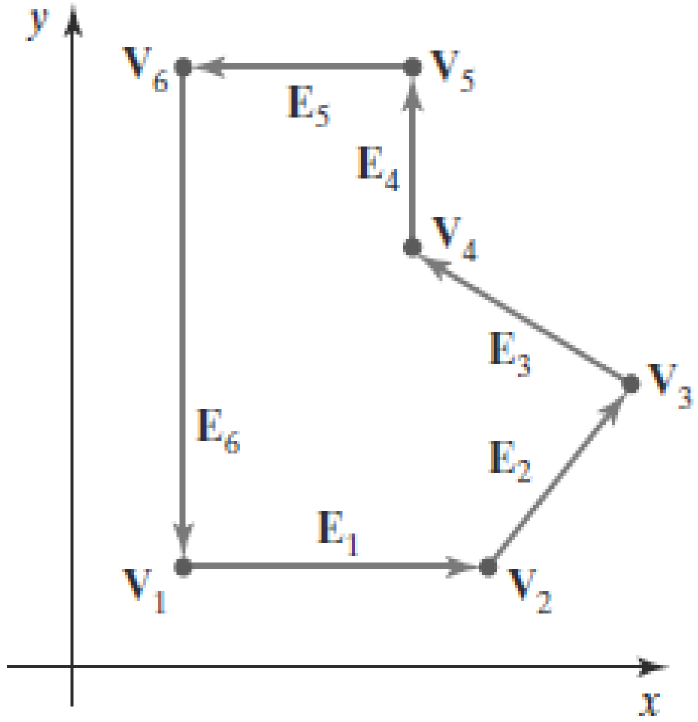


(b)

- Implementations of fill algorithms and other graphics routines are more complicated for concave polygons.
- so it is generally more efficient to split a concave polygon into a set of convex polygons before processing.
- Some graphics packages, including OpenGL, require all fill polygons to be convex.

Identifying Concave Polygons

- 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.
- 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***.



$(\mathbf{E}_1 \times \mathbf{E}_2)_z > 0$

$(\mathbf{E}_2 \times \mathbf{E}_3)_z > 0$

$(\mathbf{E}_3 \times \mathbf{E}_4)_z < 0$

$(\mathbf{E}_4 \times \mathbf{E}_5)_z > 0$

$(\mathbf{E}_5 \times \mathbf{E}_6)_z > 0$

$(\mathbf{E}_6 \times \mathbf{E}_1)_z > 0$

- Another way to identify a concave polygon is to 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

- This can be accomplished using edge vectors and edge cross-products; or, we can use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other.
- we assume that all polygons are in the ***xy*** plane.
- we first need to form the edge vectors.
- Given two consecutive vertex positions, $\mathbf{V}_k$ and $\mathbf{V}_{k+1}$, we define the edge vector between them as

$$\mathbf{E}_k = \mathbf{V}_{k+1} - \mathbf{V}_k$$

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

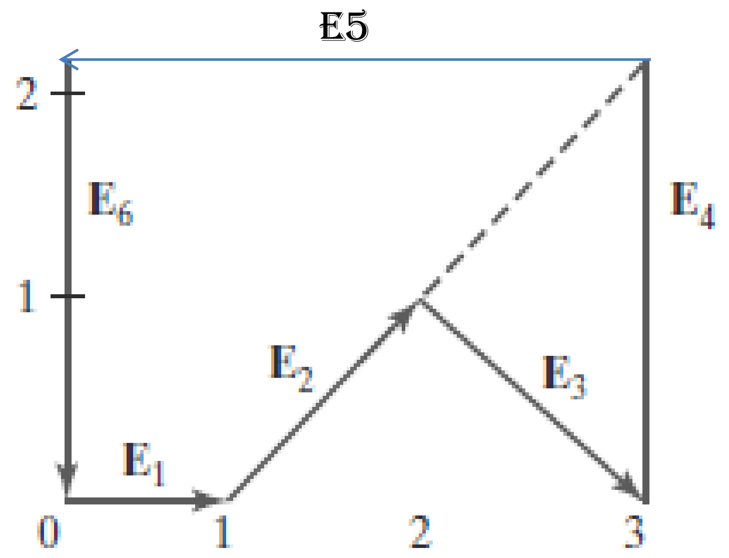
Ex:

$$\vec{A} \times \vec{B} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix}$$

$$= (A_y B_z - A_z B_y)\vec{i} - (A_x B_z - A_z B_x)\vec{j} + (A_x B_y - A_y B_x)\vec{k}$$

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

Example for identifying type of polygon using vector cross product



Calculate edge vector using formula $\mathbf{E}_k = \mathbf{V}_{k+1} - \mathbf{V}_k$.

E1=(1,0,0)-(0,0,0)   V1=(1,0,0)& V0=(0,0,0)

=(1,0,0)

E2=(2,1,0)-(1,0,0)   V2=(2,1,0) & V1=(1,0,0)

=(1,1,0)

Similarly

E3 = (1, −1, 0)   **E**4 = (0, 2, 0)

E5 = (−3, 0, 0)   **E**6 = (0, −2, 0)

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 $E_{jx}E_{ky} - E_{kx}E_{jy}$:

Calculate cross product using formula:

$$\underline{(A_yB_z - A_zB_y)\vec{i} - (A_xB_z - A_zB_x)\vec{j} + (A_xB_y - A_yB_x)\vec{k}}$$

Ex: E1 X E2=(1,0,0) X (1,1,0)

=(0x0-0x1)i-(1x0-0x1)j-(1x1-0x1)k

=(0-0)i-(0-0)j-(1-0)k

=(0)i-(0)j-(1)k

=(0,0,1)

Similarly

$\mathbf{E}2 \times \mathbf{E}3 = (0, 0, -2)$

$\mathbf{E}3 \times \mathbf{E}4 = (0, 0, 2)$

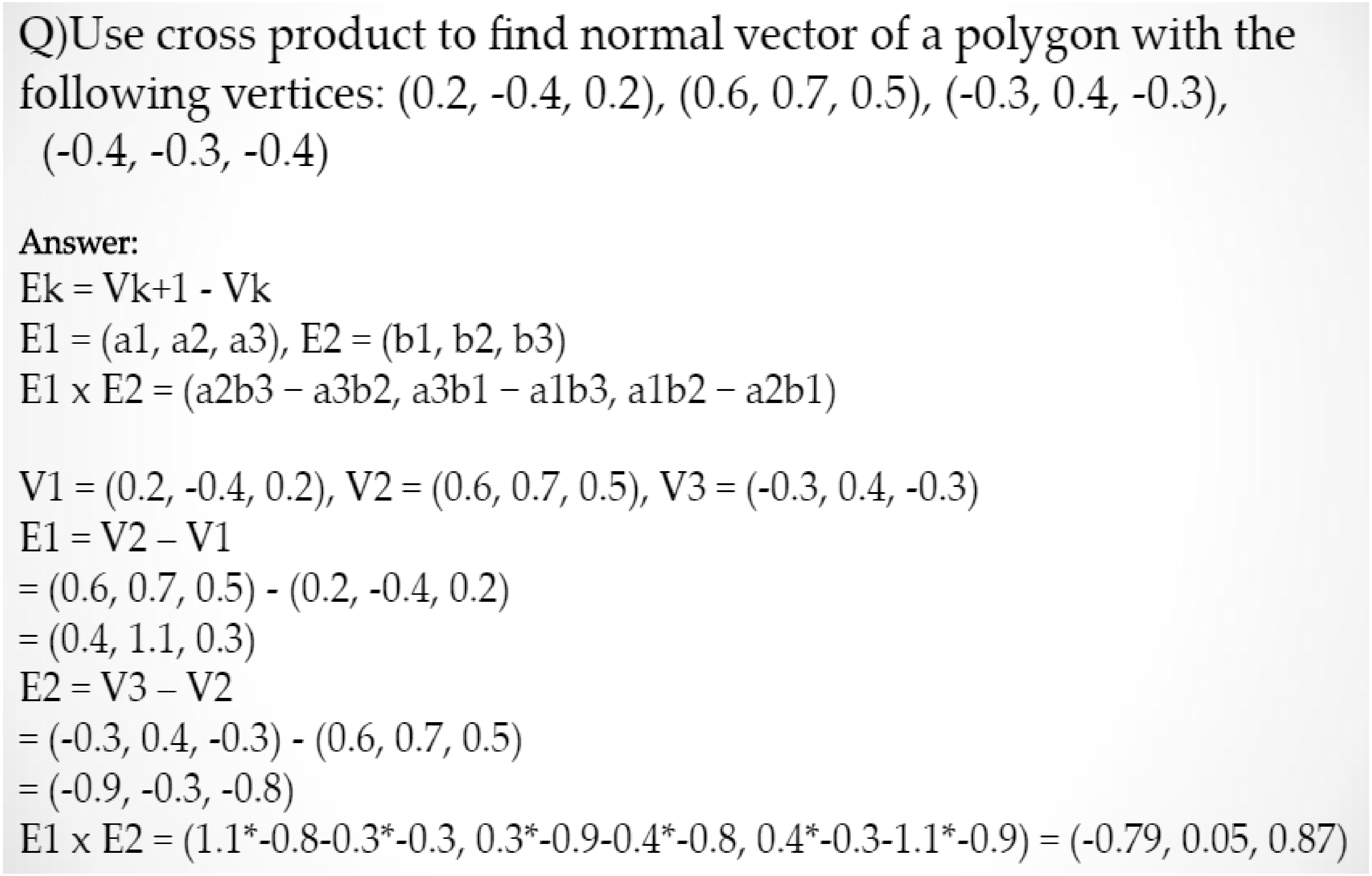
$\mathbf{E}4 \times \mathbf{E}5 = (0, 0, 6)$

$\mathbf{E}5 \times \mathbf{E}6 = (0, 0, 6)$

$\mathbf{E}6 \times \mathbf{E}1 = (0, 0, 2)$

- 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 **E**2.

Example



Q)Use cross product to find normal vector of a polygon with the following vertices: (0.2, -0.4, 0.2), (0.6, 0.7, 0.5), (-0.3, 0.4, -0.3), (-0.4, -0.3, -0.4)

Answer:

Ek = Vk+1 - Vk

E1 = (a1, a2, a3), E2 = (b1, b2, b3)

E1 x E2 = (a2b3 − a3b2, a3b1 − a1b3, a1b2 − a2b1)

V1 = (0.2, -0.4, 0.2), V2 = (0.6, 0.7, 0.5), V3 = (-0.3, 0.4, -0.3)

E1 = V2 – V1

= (0.6, 0.7, 0.5) - (0.2, -0.4, 0.2)

= (0.4, 1.1, 0.3)

E2 = V3 – V2

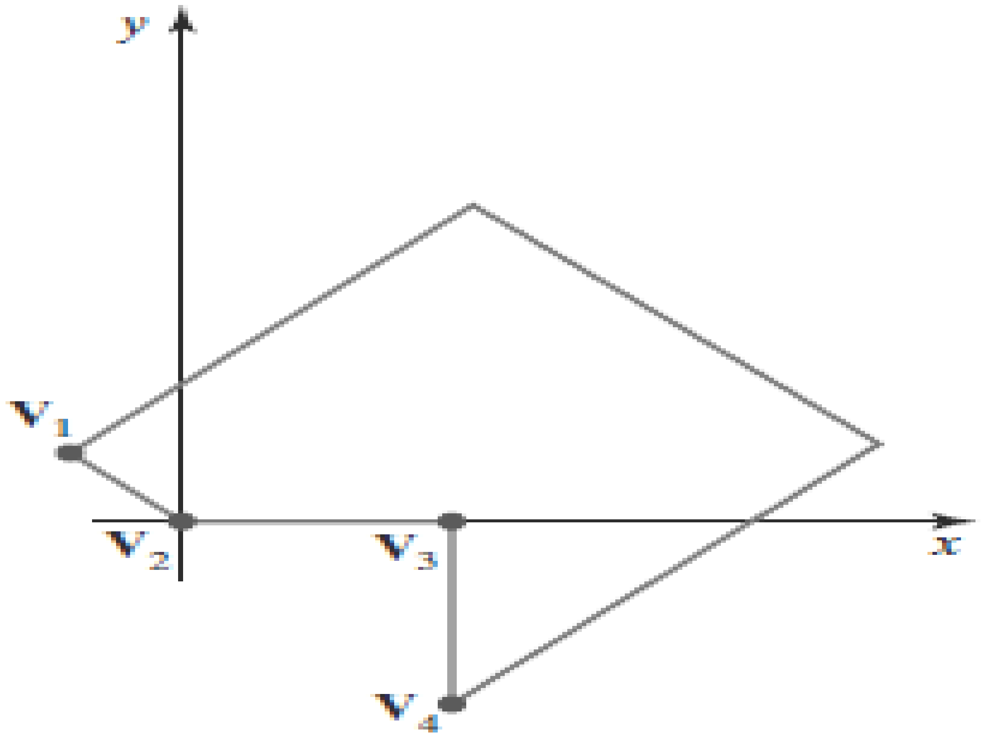
= (-0.3, 0.4, -0.3) - (0.6, 0.7, 0.5)

= (-0.9, -0.3, -0.8)

E1 x E2 = (1.1*-0.8-0.3*-0.3, 0.3*-0.9-0.4*-0.8, 0.4*-0.3-1.1*-0.9) = (-0.79, 0.05, 0.87)

We can also split a concave polygon using a **rotational method.**

- Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex $\mathbf{V}_k$ in turn is at the coordinate origin.
- Then, we rotate the polygon about the origin in a clockwise direction so that the next vertex $\mathbf{V}_{k+1}$ is on the *x* axis.
- If the following vertex, $\mathbf{V}_{k+2}$, 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.



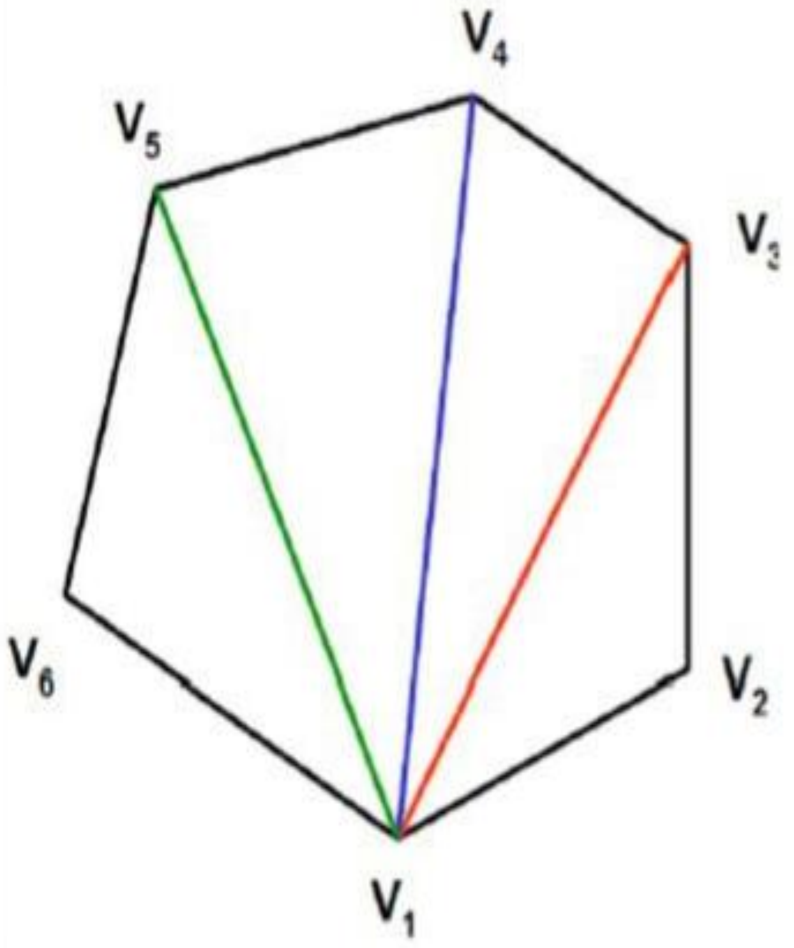
- These steps are repeated until we have tested all vertices in the polygon list.

Splitting a Convex Polygon into a Set of Triangles

Once we have a vertex list for a convex polygon, we could transform it into a set of triangles.

- 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.
- Then the same procedure is applied to this modified vertex list to strip off another triangle.
- We continue forming triangles in this manner until the original polygon is reduced to just three vertices, which define the last triangle in the set.

Example



- $V_1, V_2, V_3, V_4, V_5, V_6$
- V_1, V_2, V_3 = 1st triangle
- V_1, V_3, V_4, V_5, V_6
- V_1, V_3, V_4 = 2nd triangle
- V_1, V_4, V_5, V_6
- V_1, V_4, V_5 = 3rd triangle
- V_1, V_5, V_6 = 4th triangle

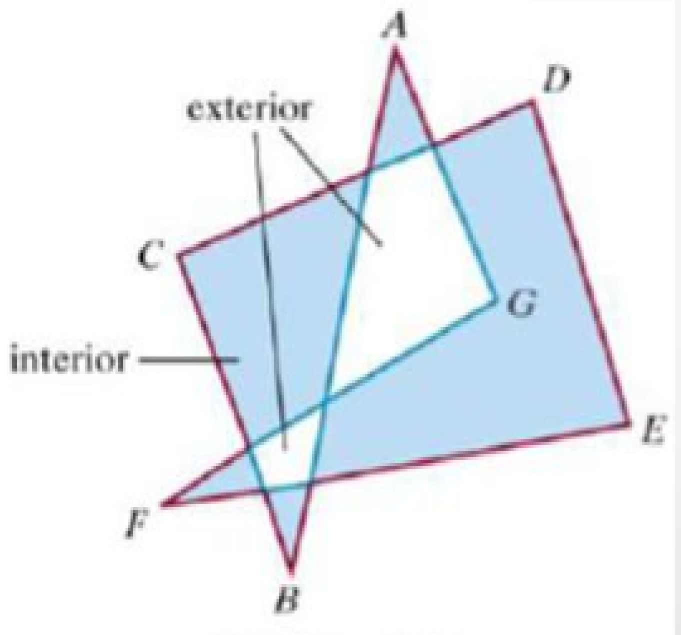
Inside-Outside Tests

- It is not clear which regions of the *xy* plane we should call interior and which regions we should designate as exterior for a complex polygon with intersecting regions.

- algorithms:
  - ❑ odd-even rule
  - ❑ nonzero winding-number rule

odd-even rule

- Draw a ***reference line***from any position to a distant point outside a closed polyline
- The line must not pass through any endpoints
- Count the number of line segments crossed along this line
- If the number is odd then the region considered to be ***interior***
- Otherwise, the region is ***exterior***



Odd-Even Rule

Nonzero Winding Number Rule :

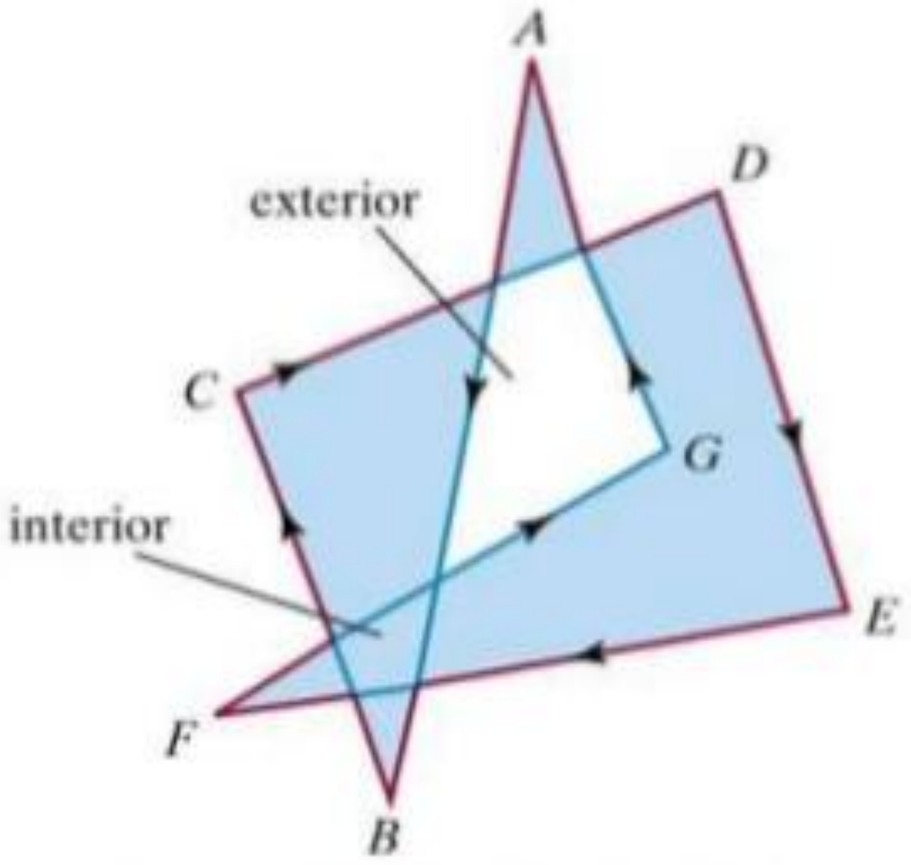
- Another method of finding whether a point is inside or outside of the polygon. In this every point has a winding number, and the interior points of a two-dimensional object are defined to be those that have a nonzero value for the winding number.

1. Initializing the winding number to 0.
2. Imagine a line drawn from any position P to a distant point beyond the coordinate extents of the object.
3. Count the number of edges that cross the line in each direction. We add 1 to the winding number every time we intersect a polygon edge that crosses the line from right to left, and we subtract 1 every time we intersect an edge that crosses from left to right.
4. If the winding number is **nonzero**, then

   P is defined to be an **interior** point

   Else

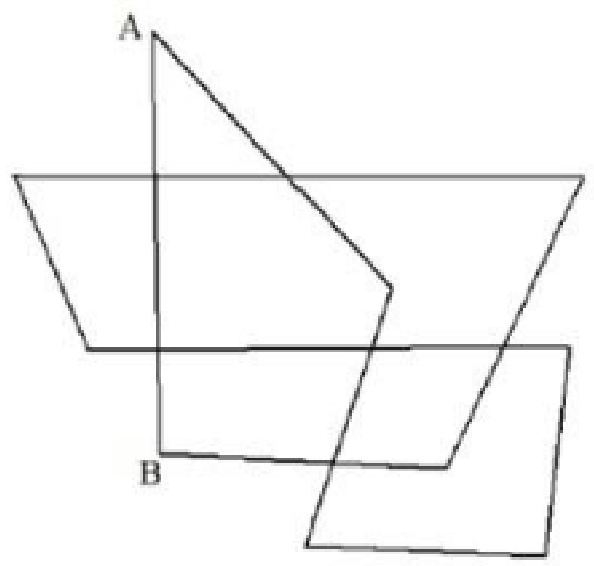
   P is taken to be an **exterior** point.



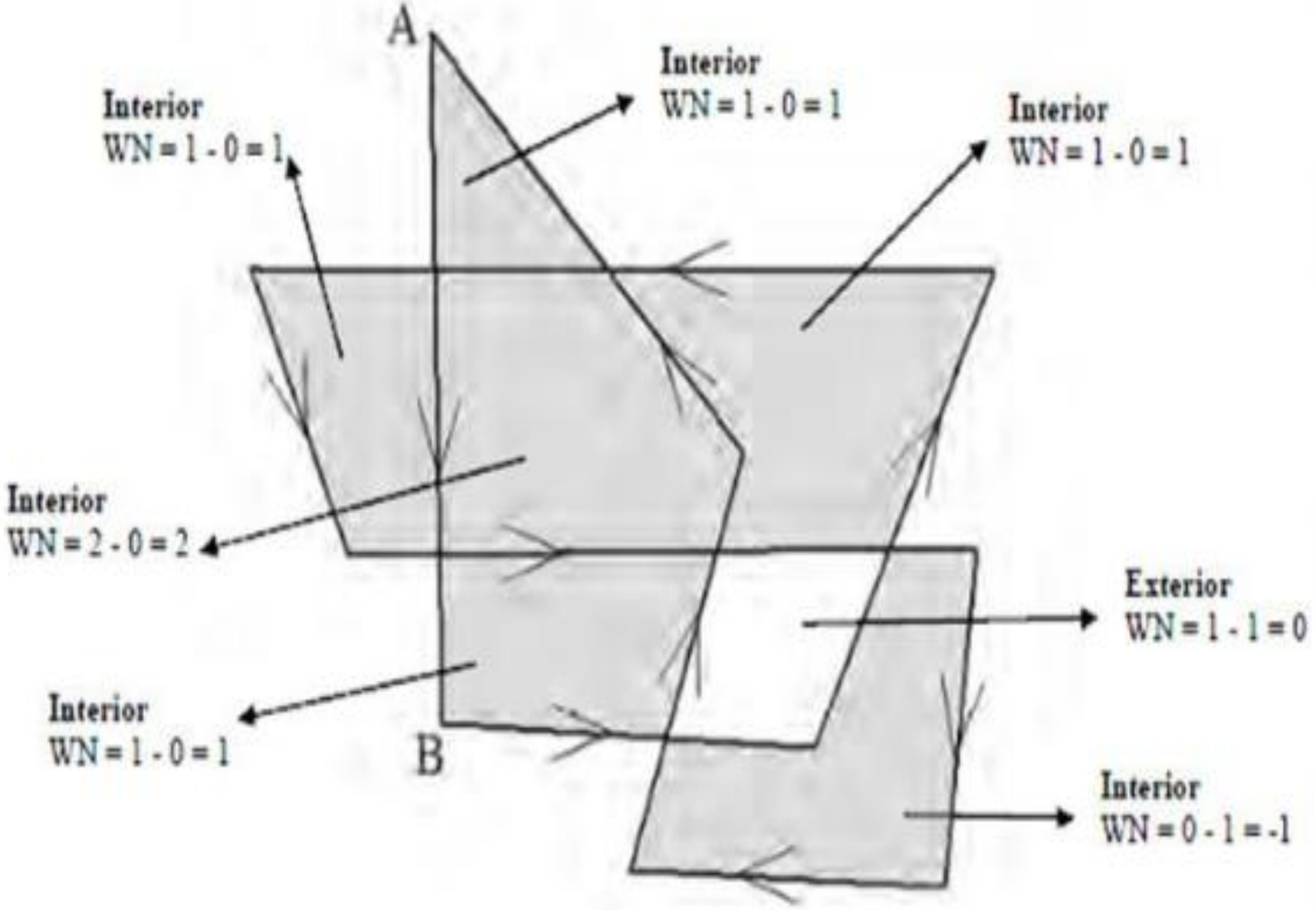
Nonzero Winding-Number Rule

Example

Q) Use nonzero winding number rule to determine the interior and exterior regions of the following polygon ?



Answer:


A
Interior
WN = 1 - 0 = 1
Interior
WN = 1 - 0 = 1
Interior
WN = 1 - 0 = 1
Interior
WN = 2 - 0 = 2
Exterior
WN = 1 - 1 = 0
Interior
WN = 1 - 0 = 1
B
Interior
WN = 0 - 1 = -1

- One way to determine directional boundary crossings is to set up vectors along the object edges (or boundary lines) and along the reference line.
- Then we compute the vector cross-product of the vector **u**, along the line from **P** to a distant point, with an object edge vector **E** for each edge that crosses the line.
- Assuming that we have a two-dimensional object in the *xy* plane, the direction of each vector cross-product will be either in the +*z* direction or in the −*z* direction.

- If the *z* component of a cross-product **u** × **E** for a particular crossing is positive, that segment crosses from right to left and we add 1 to the winding number.
- Otherwise, the segment crosses from left to right and we subtract 1 from the winding number.

Polygon Tables

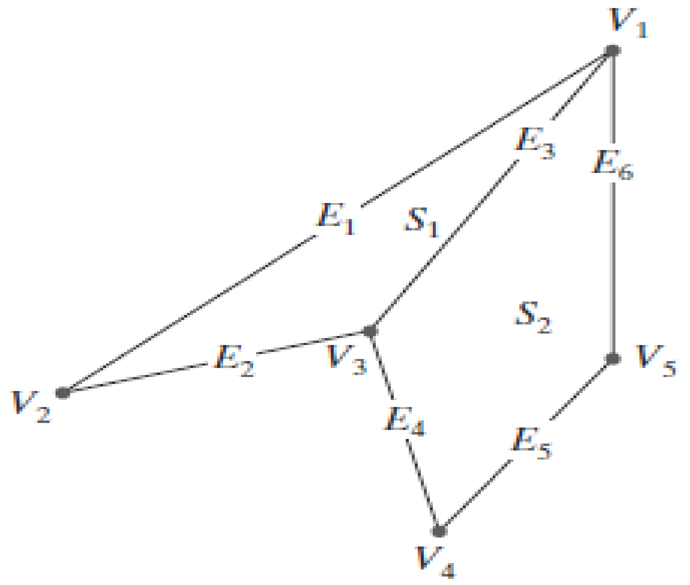
- the objects in a scene are described as sets of polygon surface facets.
- graphics packages often provide functions for defining a surface shape as a mesh of polygon patches.
- The description for each object includes coordinate information specifying the geometry for the polygon facets and other surface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

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

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

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

- 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.**



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

- Additional geometric information that is usually stored in the data tables includes the slope for each edge and the coordinate extents for polygon edges, polygon facets, and each object in a scene.
- The more information included in the data tables, the easier it is to check for errors.

Some of the tests that could be performed by a graphics package are:

1. that every vertex is listed as an endpoint for at least two edges.
2. that every edge is part of at least one polygon.

3. that every polygon is closed.

4. that each polygon has at least one shared edge.

5. that if the edge table contains pointers to polygons.

Plane Equations

Graphics system processes the input data through several procedures:

1. Transformation of the modeling and world-coordinate.
2. identification of visible surfaces.
3. application of rendering routines to the individual surface facets.

For some of these processes, information about the spatial orientation of the surface components of objects is needed.

This information is obtained from the vertex coordinate values and the equations that describe the polygon surfaces.

- 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, and the coefficients *A*, *B*, *C*, and *D*(called *plane parameters*) are constants describing the spatial properties of the plane.

- *A*, *B*, *C*, and *D can be obtained* by solving a set of three plane equations using the coordinate values for three noncollinear points in the plane.

For this purpose, we can select three successive convex-polygon vertices, (*x*1, *y*1, *z*1), (*x*2, *y*2, *z*2), and (*x*3, *y*3, *z*3), in a counterclockwise order and solve the following set of simultaneous linear plane equations for the ratios *A/D*, *B/D*, and *C/D*:

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

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

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

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

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

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$

$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$

$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1)$$

- When vertex coordinates and other information are entered into the polygon data structure, values for *A*, *B*, *C*, and *D* can be computed for each polygon facet and stored with the other polygon data.

Example

$$x + 2y + 3z = -5$$

$$3x + y - 3z = 4$$

$$-3x + 4y + 7z = -7$$

- **coefficient matrix D=** $\begin{vmatrix} 1 & 2 & 3 \\ 3 & 1 & -3 \\ -3 & 4 & 7 \end{vmatrix}$
- **X – matrix A =** $\begin{vmatrix} -5 & 2 & 3 \\ 4 & 1 & -3 \\ -7 & 4 & 7 \end{vmatrix}$

- Y – matrix $B = \begin{vmatrix} 1 & -5 & 3 \\ 3 & 4 & -3 \\ -3 & -7 & 7 \end{vmatrix}$
- Z – matrix $C = \begin{vmatrix} 1 & 2 & -5 \\ 3 & 1 & 4 \\ -3 & 4 & 7 \end{vmatrix}$

- **Determinants of each matrix**

  $|D| = 40 \quad |A| = -40 \quad |B| = 40 \quad |C| = -80$

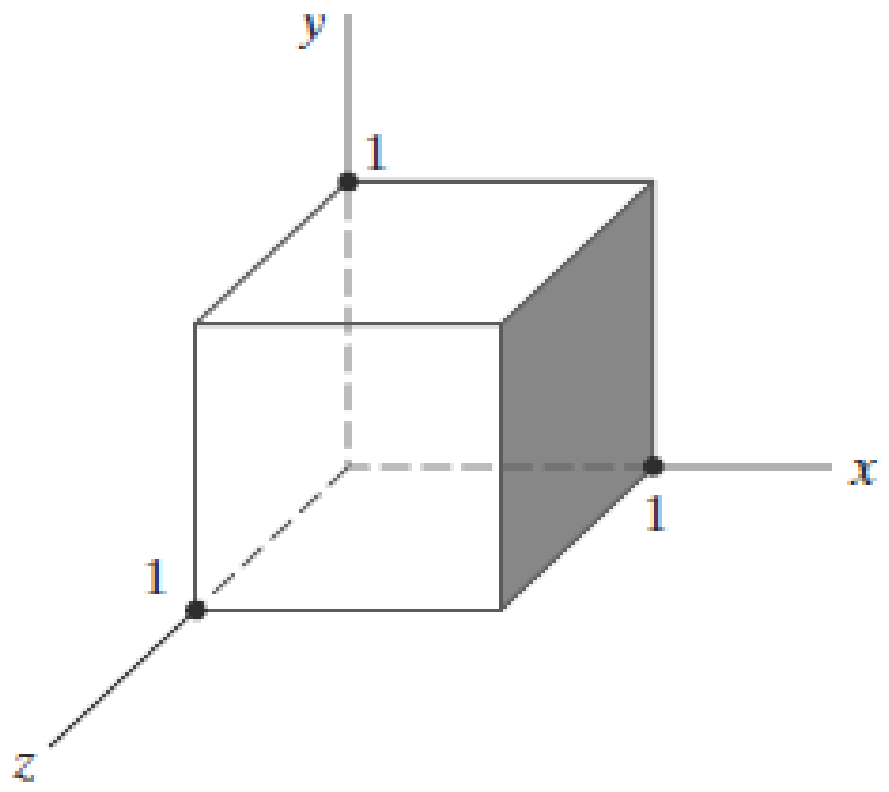
Front and Back Polygon Faces

- The side of a polygon that faces into the object interior is called the **back face.**
- The visible, or outward, side is the **front face.**
- 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.

- For any point (x, y, z) not on a plane with parameters A, B, C, D, we have

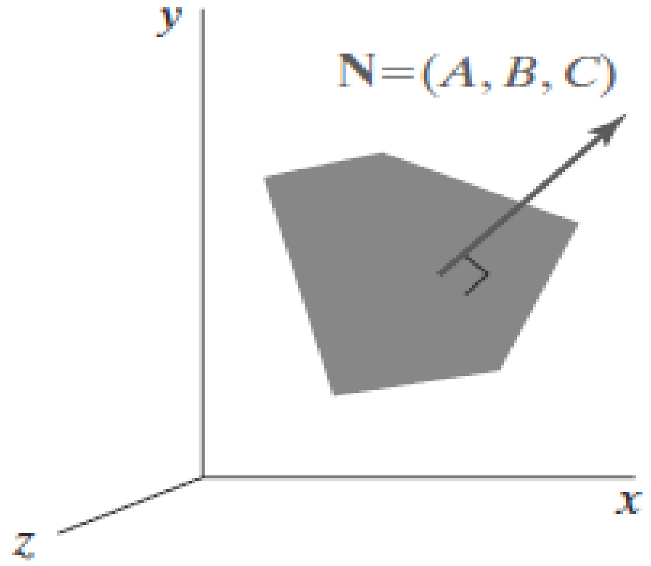
$$Ax + B y + C z + D != 0$$

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



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

- Orientation of a polygon surface in space can be described with the **normal vector** for the plane containing that polygon.



- This surface normal vector is perpendicular to the plane and has Cartesian components (*A*, *B*, *C*), where parameters *A*, *B*, and *C* are the plane coefficients calculated.
- **DIRECTION IS from the back face of the polygon to the front face.**

- The elements of a normal vector can also be obtained using a vector crossproduct calculation.
- we again select any three vertex positions,**V**1,**V**2, and **V**3, taken in counterclockwise order when viewing from outside the object toward the inside.
- Form two vectors, one from **V**1 to **V**2 and the second from **V**1 to **V**3, we calculate **N** as the vector cross-product:

$$\mathbf{N} = (\mathbf{V}2 - \mathbf{V}1) \times (\mathbf{V}3 - \mathbf{V}1)$$

This generates values for the plane parameters *A*, *B*, and *C.*

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

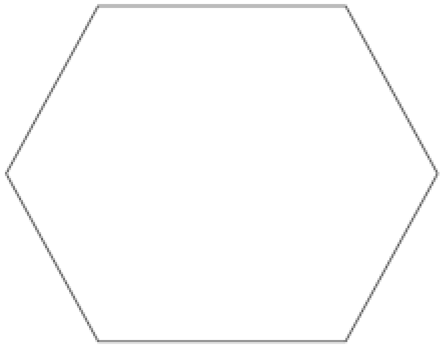
$$\mathbf{N} \cdot \mathbf{P} = -D$$

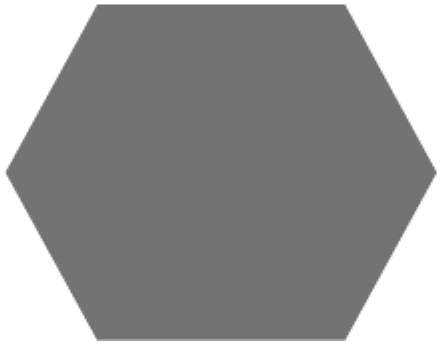
Fill-Area Attributes

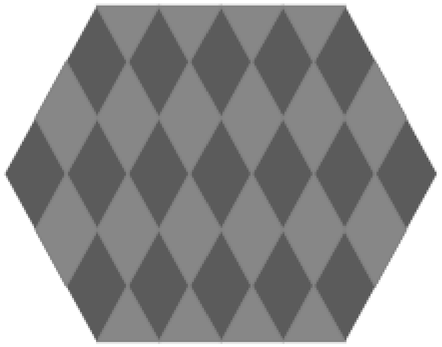
Most graphics packages limit fill areas to polygons because they are described with linear equations.

- **Fill Styles**

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.


Hollow


Solid


Patterned

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.

- 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.

OpenGL Fill-Pattern Function

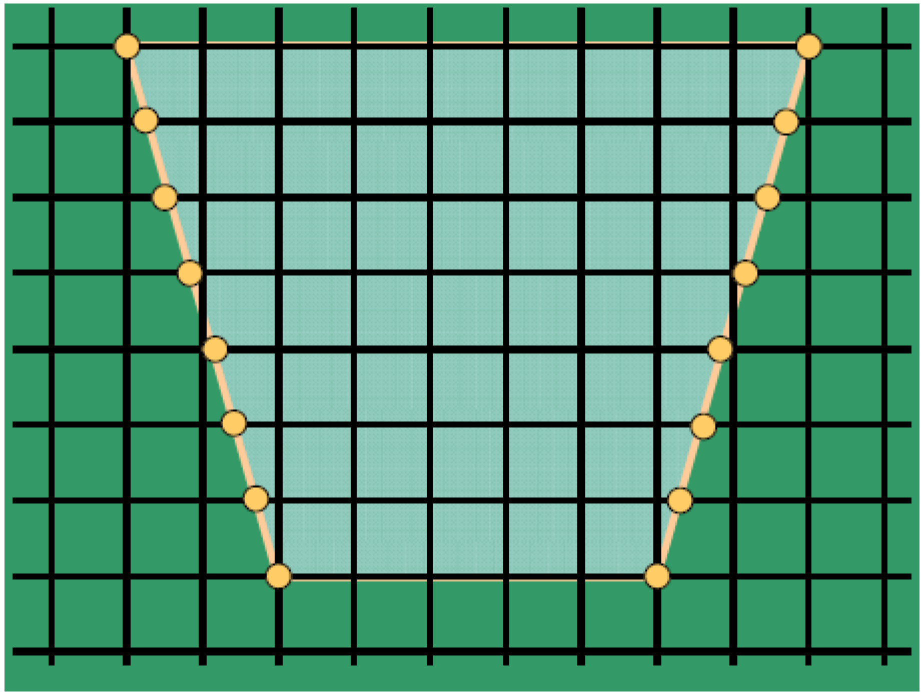
- To fill the polygon with a pattern in OpenGL, we use a 32 × 32 bit mask.
- A value of 1 in the mask indicates that the corresponding pixel is to be set to the current color, and a 0 leaves the value of that frame-buffer position unchanged.

example,

- **GLubyte fillPattern [ ] = { 0xff, 0x00, 0xff, 0x00, ... };**
- The bits must be specified starting with the bottom row of the pattern, and *con*tinuing up to the topmost row (32) of the pattern.

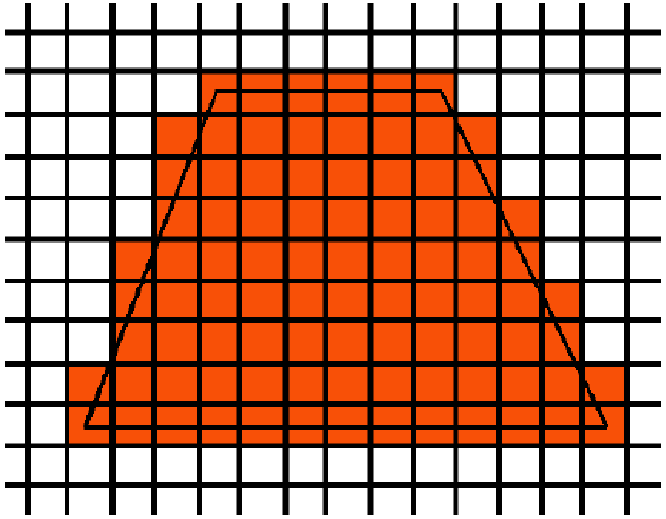
- **glPolygonStipple (fillPattern);**
- **glEnable (GL_POLYGON_STIPPLE);**
- **glDisable (GL_POLYGON_STIPPLE);**

SCANLINE POLYFILL ALGORITHM



Assume:

Pixels are not at the center of the grid, but at the intersection of two orthogonal scan lines (on the grid intersection points).

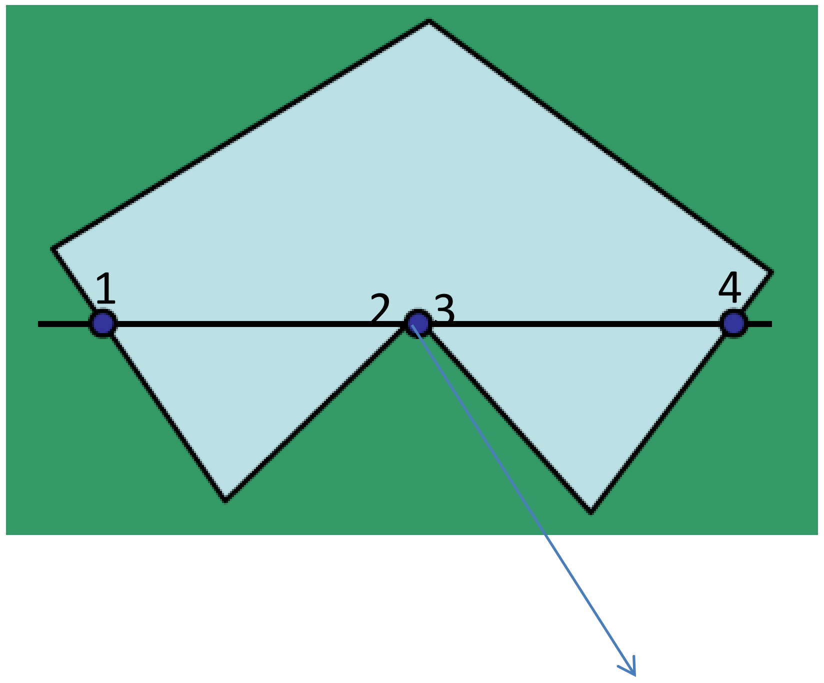


Conceptual Scan Line Polygon Fill Algorithm:

- Find minimum enclosed rectangle
- No. of scan lines = $Y_{max} – Y_{min} + 1$
- For each scan line do
    - Obtain intersection points of scan line with polygon edges.
    - Sort intersections from left to right
- Form pairs of intersections from the list§
- Fill within pairs
- Intersection points are updated for each scan line
- Stop when scan line has reached Y_{max}

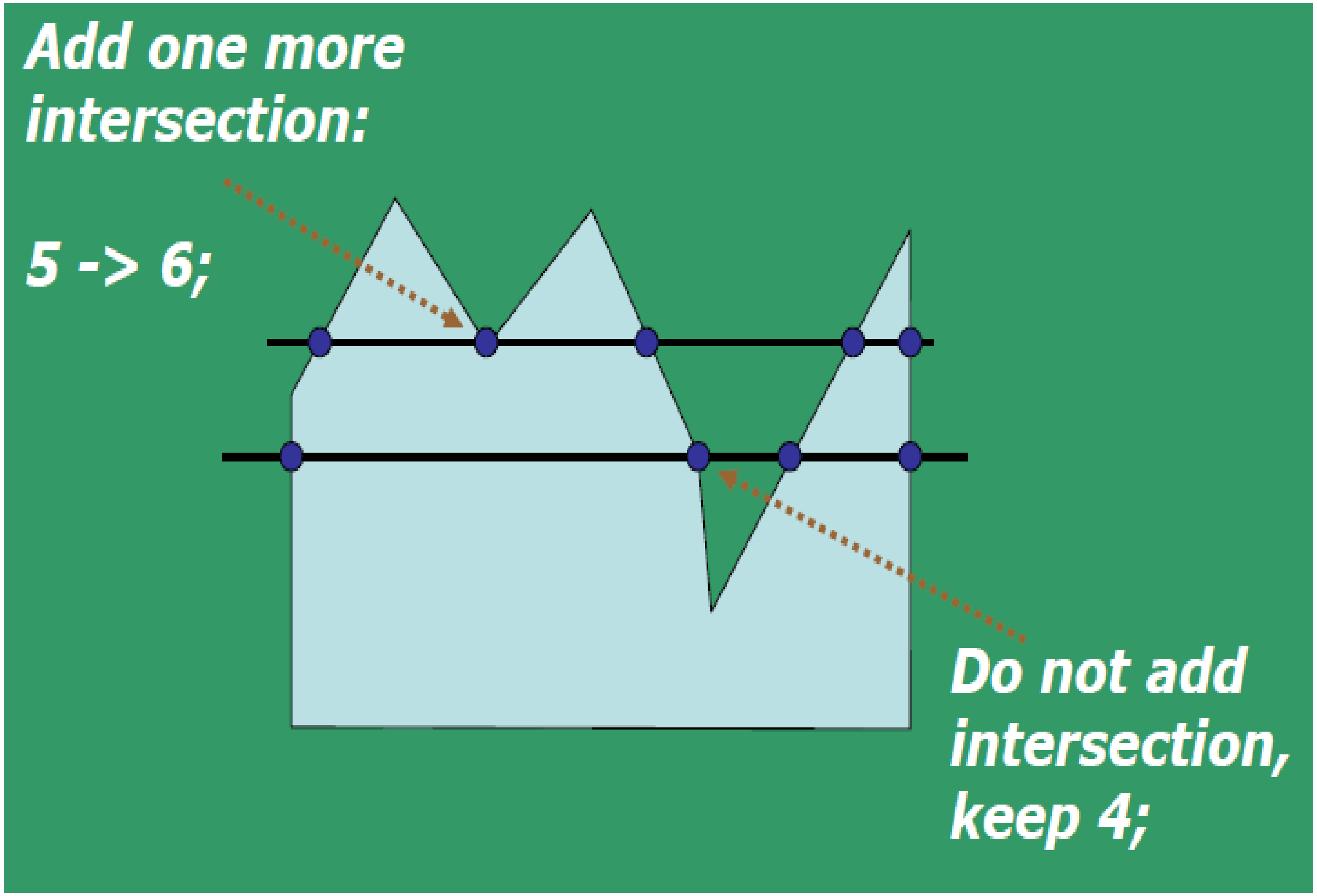
Two different cases of scan lines passing through the vertex of a polygon

<u>CASE-1:</u>



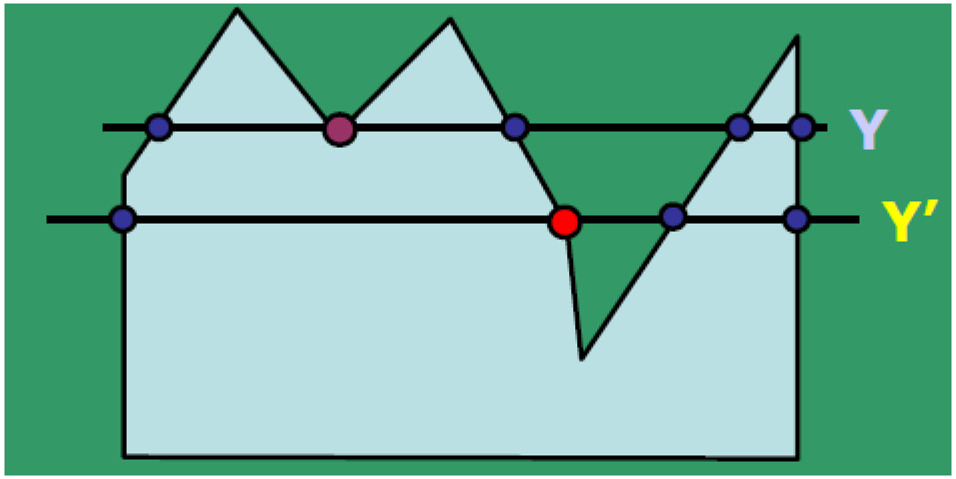
- ***Add one more intersection: 3 -> 4***

CASE-2:



How?

What is the difference between the intersection of the scan lines Y and Y', with the vertices?

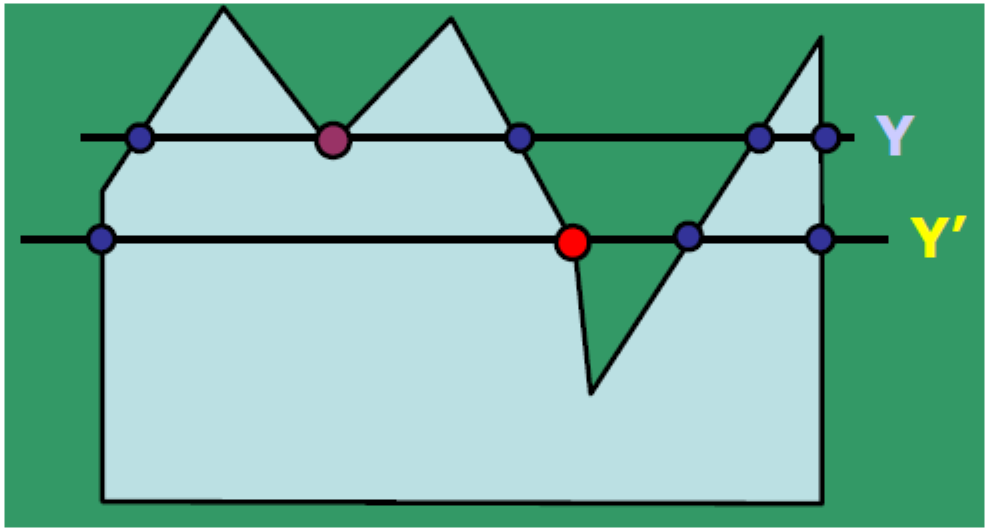


For Y, the edges at the vertex are on the same side of the scan line.

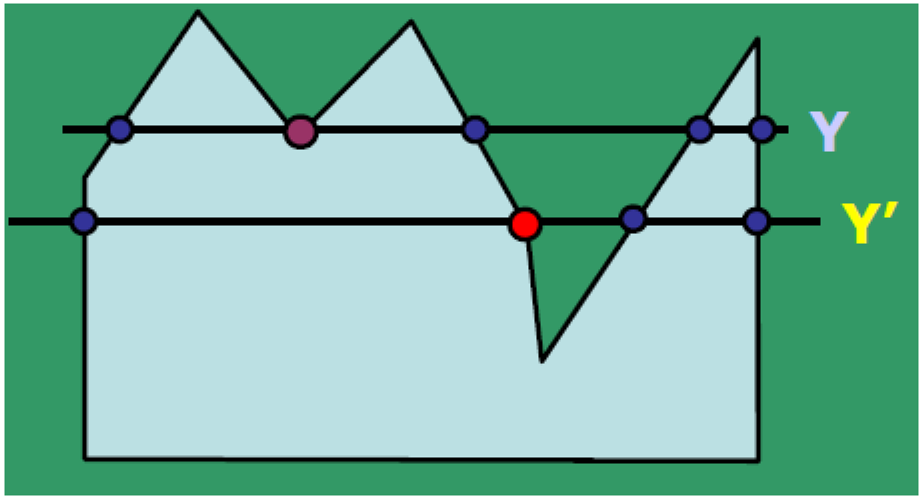
Whereas for Y', the edges are on either/both sides of the vertex.

In this case, we require additional processing.

Vertex counting in a scan line



- Traverse along the polygon boundary clockwise (or counter- clockwise) and
- Observe the *relative change in Y-value* of the edges on either side of the vertex (i.e. as we move from one edge to another).

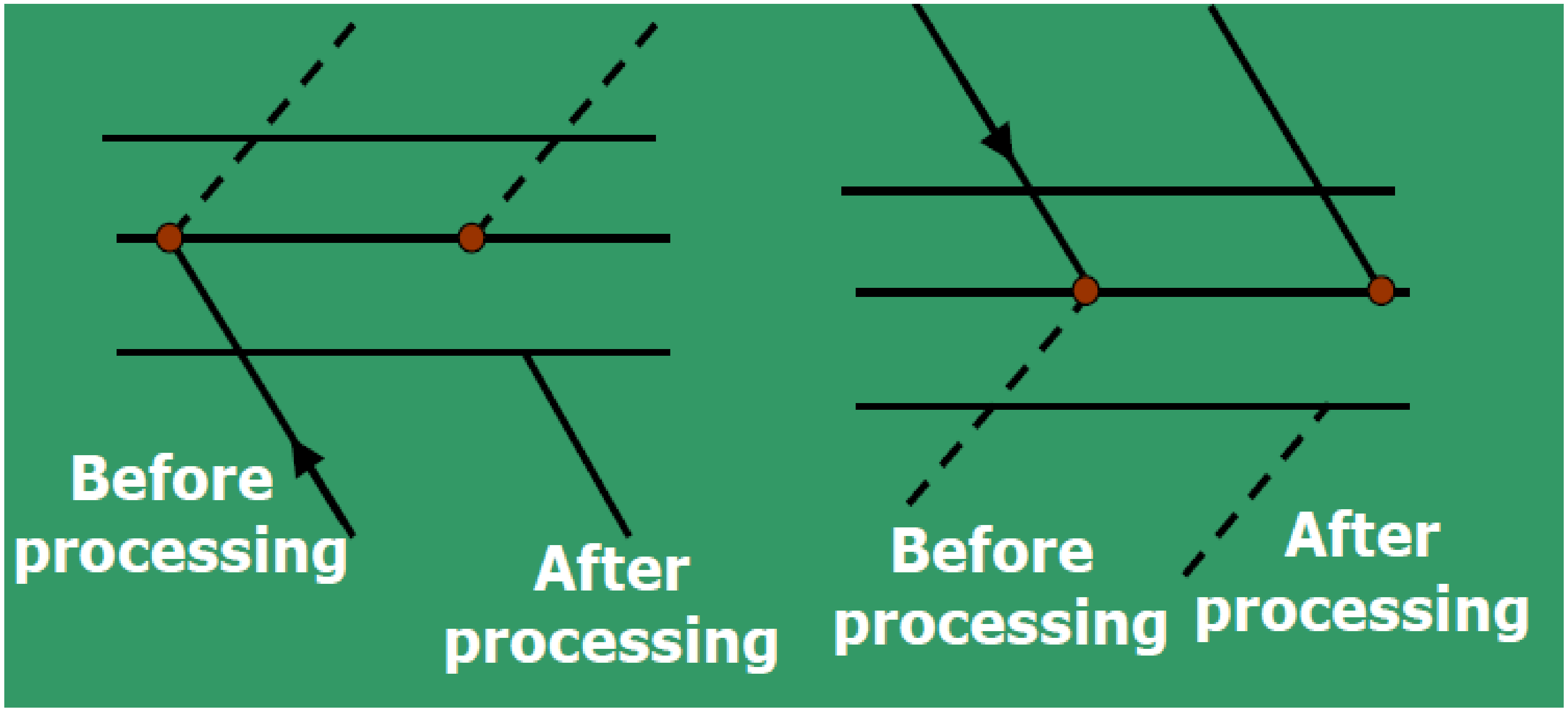


Check the condition:

- If *end-point Y values* of two consecutive edges *monotonically increase or decrease*, count the middle vertex as a single intersection point for the scan line passing through it.
- Else the shared vertex represents a *local maximum (or minimum)* on the polygon boundary. *Increment the intersection count*.

To implement the above:

- Shorten the lower edge to ensure only one intersection point at the vertex.



Scan line Poly Fill Algorithm

Intersect scan line with polygon edges.

Fill between pairs of intersections

Basic Structure:

For $y = Y_{min}$ to Y_{max}

1) intersect scan line with each edge

2) sort intersections by increasing X

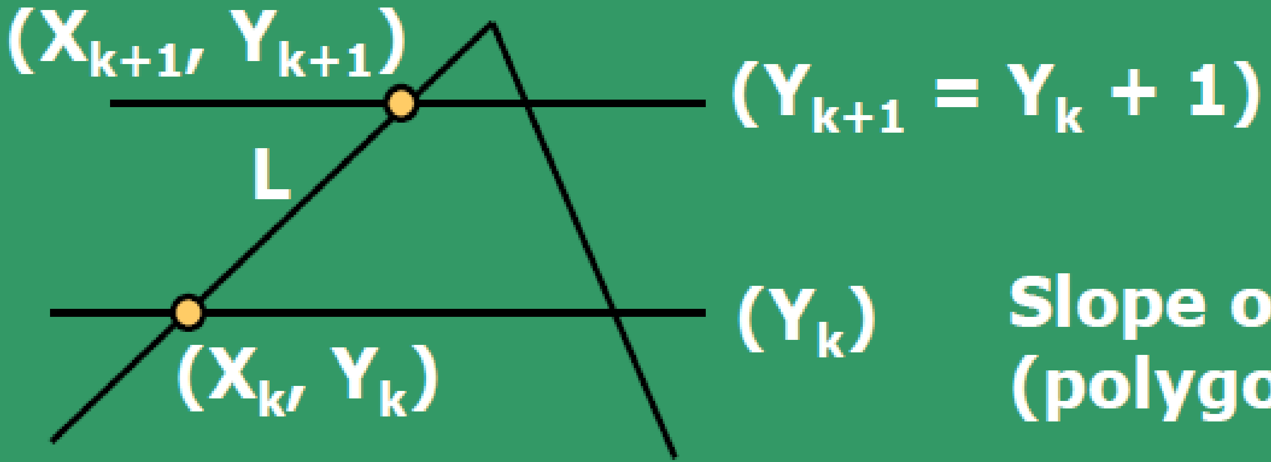
3) fill pairwise (int0 -> int1, int2 -> int3, ...)

4) Update intersections for next scan line

This is the basic structure, but we are going to handle some special cases to make sure it works correctly and fast.

Two important features of scan line-based polygon filling are:

- ***scanline coherence*** - values don't change much from one scanline to the next – the coverage (or visibility) of a face on one scanline typically differs little from the previous one.
- ***edge coherence*** - edges intersected by scanline “i” are typically intersected by scanline “i+1”.



Slope of the line L (polygon edge) is:

$$m = \frac{Y_{k+1} - Y_k}{X_{k+1} - X_k}$$

If, $Y_{k+1} = Y_k + 1$;

Then, $X_{k+1} = X_k + 1/m$

Thus the intersection for the next scanline is obtained as:

$X_{k+1} = \text{round}(X_k + 1/m)$, where $m = \Delta Y/\Delta X$.

Data Structure Used (typical example):

SET (Sorted Edge table):

- **Contains all information necessary to process the scanlines efficiently.**
- **SET is typically built using a bucket sort, with a many buckets as there are scan lines.**
- **All edges are sorted by their Y_{min} coordinate, with a separate Y bucket for each scanline.**
- **Within each bucket, edges sorted by increasing X of Y_{min} point.**
- **Only non-horizontal edges are stored. Store these edges at the scanline position in the SET.**

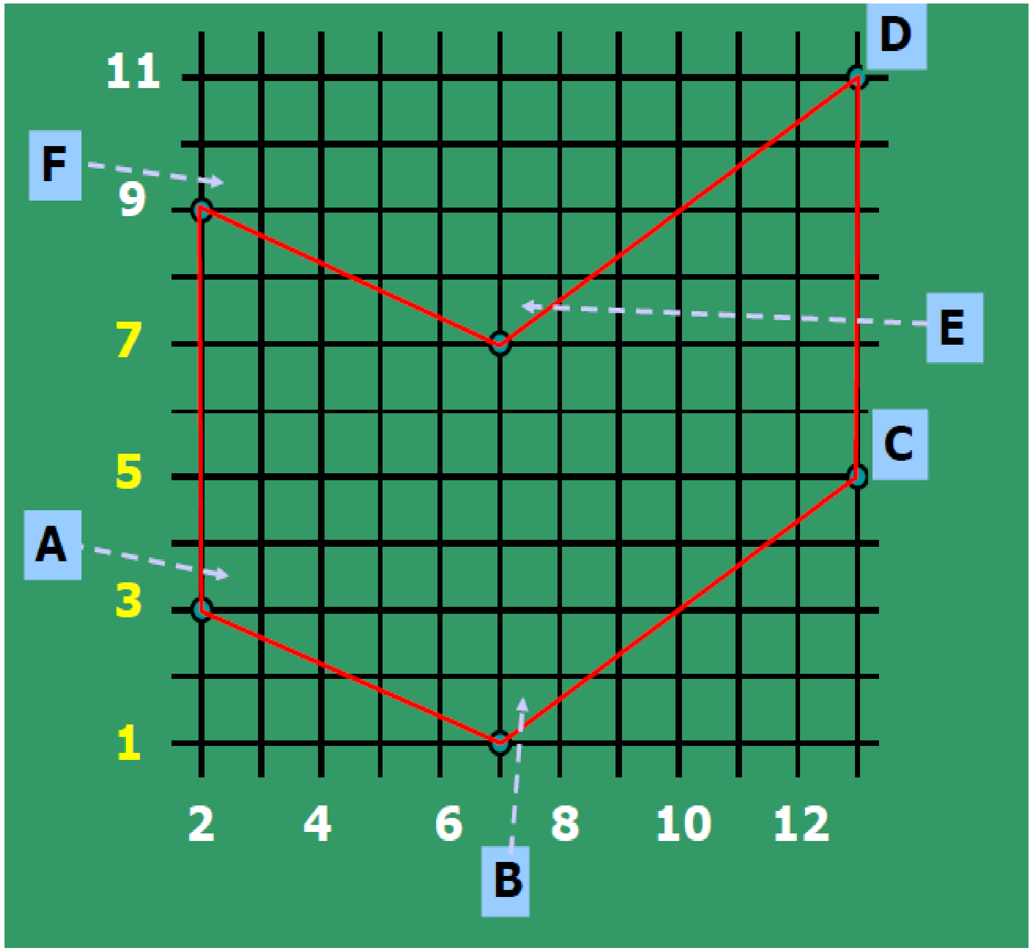
Edge structure

(sample record for each scanline):

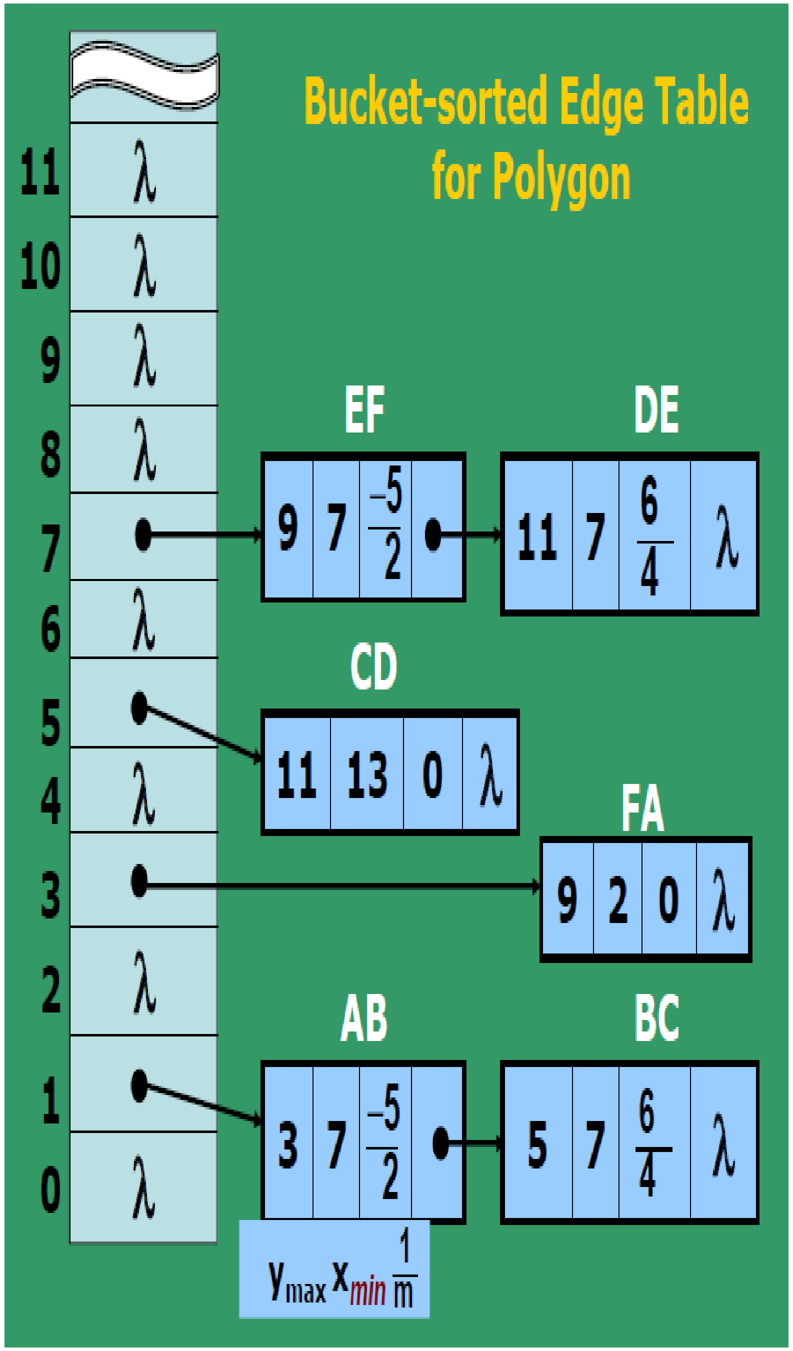
(Y_{max}, X_{min}, $\Delta X/\Delta Y$, pointer to next edge)

AEL (Active edge List):

- **Contains all edges crossed by a scanline at the current stage of iteration.**
- **This is a list of edges that are active for this scanline, sorted by increasing X intersections.**
- **Also called: Active Edge Table (AET).**



Bucket-sorted Edge Table for Polygon



How to implement this using integer arithmetic ?

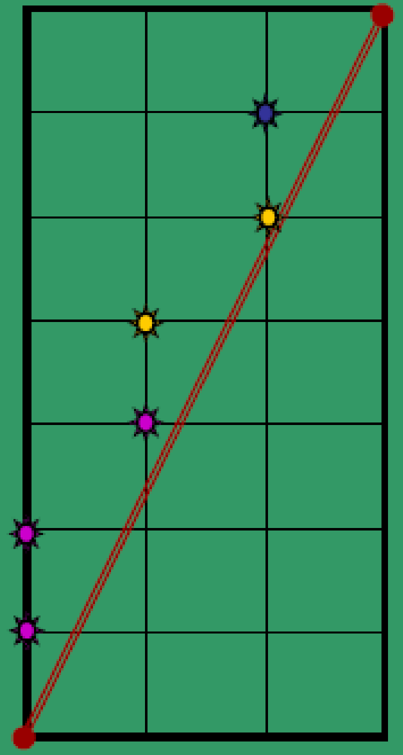
Take an example: $m = \Delta Y/\Delta X = 7/3$.

Set Counter, $C = 0$

and counter-increment, $\Delta C = \Delta X = 3$;

For the next three scan lines,
successive values of C are : 3, 6, 9.

Thus only at 3rd scanline $C >= \Delta Y$.



Then, X_k is incremented by 1 only at 3rd scanline and set as: $C \leftarrow C - \Delta Y = 9 - 7 = 2$.

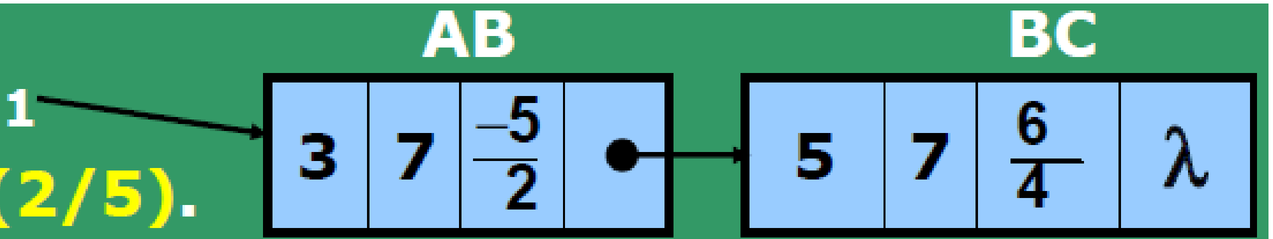
Repeat the above step(s) till Y_k reaches Y_{max} .

After 2 more scanlines: 2 + 3 + 3 = 8; 8 – 7 = 1;

After 2 more scanlines: 1 + 3 + 3 = 7;

AB:

$m = \Delta Y/\Delta X = -(2/5)$.



Set Counter, $C = 0$; and

counter-increment, $\Delta C = \min(\Delta X, \Delta Y) = 2\ (= \Delta Y)$;

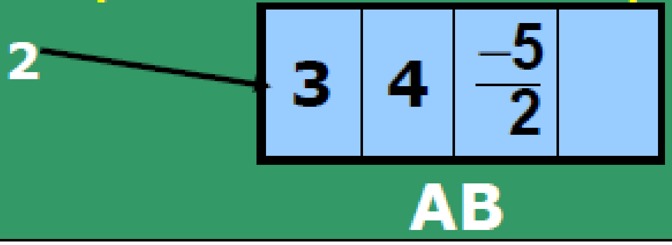
Update for AB (-ve m), when $Y_K = 2$; $Y = 1$:

For the next three left (-ve) vertical (Y) scan lines, successive values of C are : 2, 4, 6; $X = 7 - 3 = 4$;

Thus only at 3rd iteration: $C >= \Delta X$.

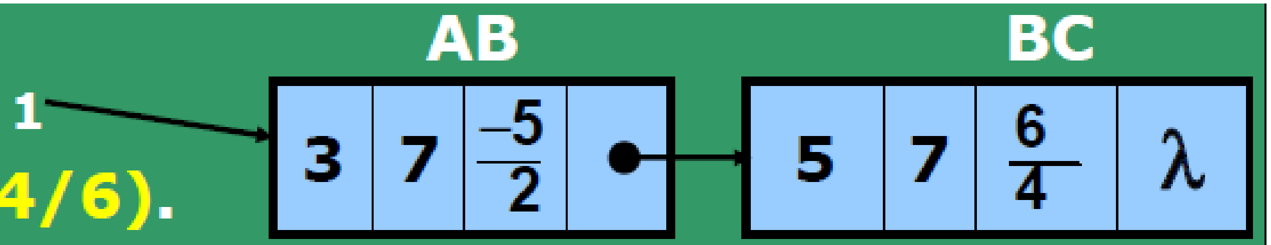
Then, Y is incremented by 1 only at 3rd scanline and set: $C \leftarrow C - \Delta X = 6 - 5 = 1$; $Y = 1 + 1 = 2$;

Stop as $Y = Y_K$.



BC:

$m = \Delta Y/\Delta X = (4/6)$.

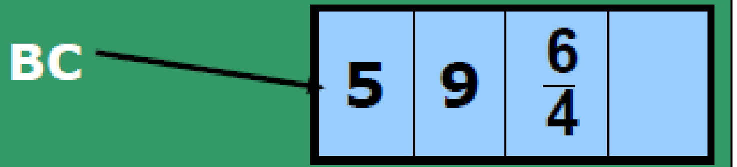


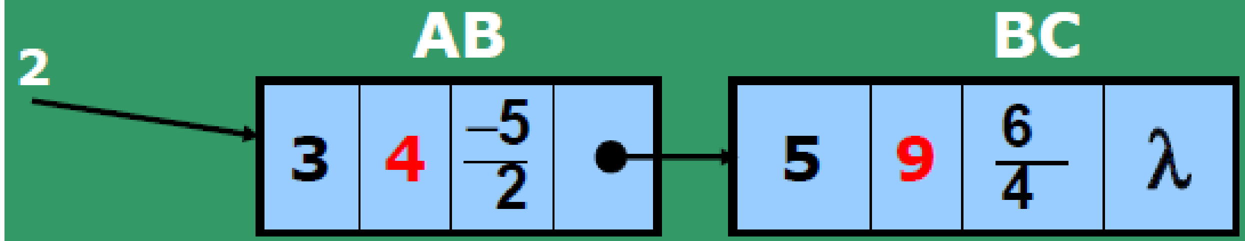
Set Counter , $C = 0$; and

counter-increment, $\Delta C = \min(\Delta X, \Delta Y) = 4$ ($= \Delta Y$);

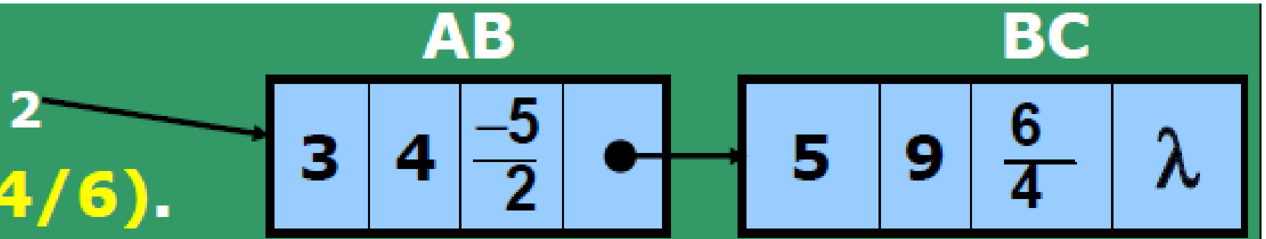
Update for BC (+ve m), when $Y_K = 2$; $Y = 1$:
For the next two right vertical (Y) scan lines,
successive values of C are : 4, 8; $X = 7 + 2 = 9$;

Thus only at 2nd iteration: $C >= \Delta X$.
Then, Y is incremented by 1 only at 2nd scanline
and set : $C \leftarrow C - \Delta X = 8 - 6 = 2$; $Y = 1 + 1 = 2$;
Stop as $Y = Y_K$.





BC:



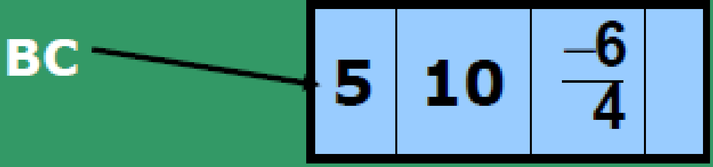
$m = \Delta Y/\Delta X = (4/6)$.

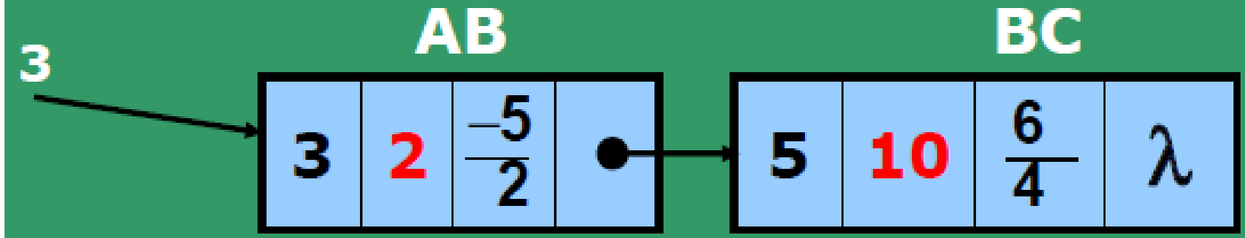
Counter (from earlier iteration) , $C = 2$;   and

counter-increment, $\Delta C = \min(\Delta X, \Delta Y) = 4 \; (= \Delta Y)$;

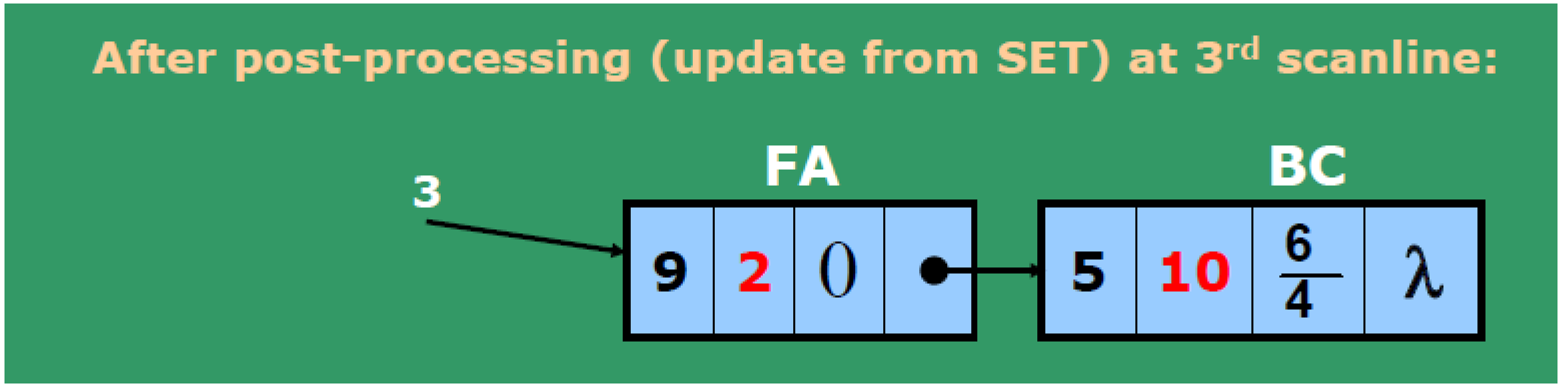
Update for BC (+ve m), when $Y_K = 3$; $Y = 2$:
For the next **right** vertical (Y) scan line, the
successive value of C is : 6; $X = 9 + 1 = 10$;

Thus only at 1st iteration: $C >= \Delta X$.
Then, Y is incremented by 1 only at 1st scanline
and set : $C \leftarrow C - \Delta X = 6 - 6 = 0$; $Y = 2 + 1 = 3$;
Stop as $Y = Y_K = ??$.

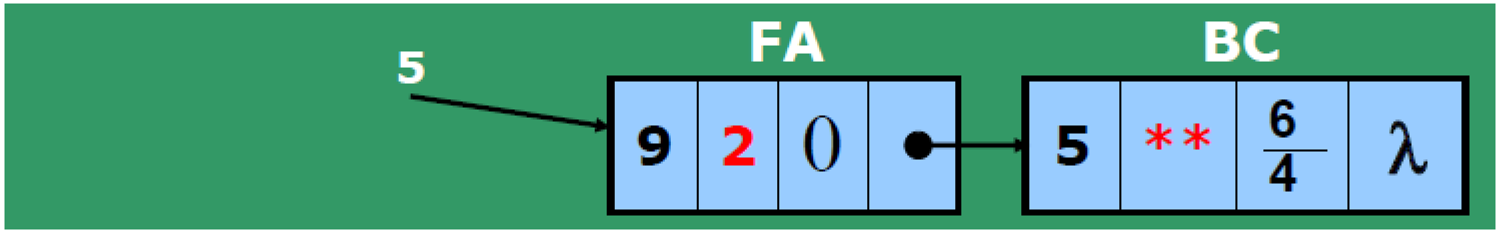


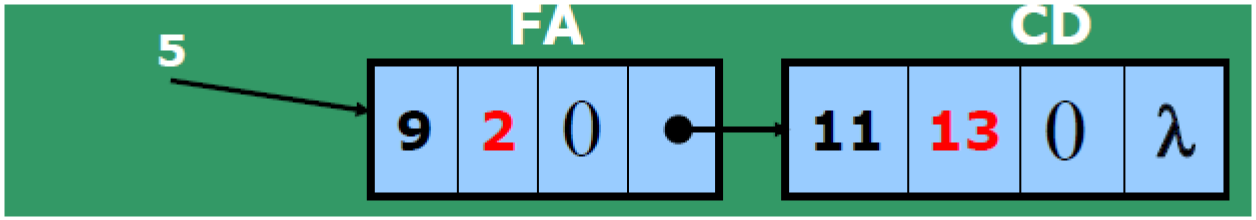


<< - Is this OK ??



- What are the vales at scan line 5



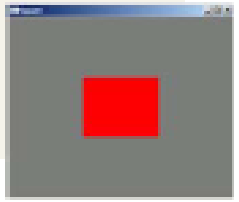


OpenGL Polygon Fill-area Functions

- In OpenGL, specifying fill polygons are similar to those for describing a point or polyline.

```
glBegin(SYMBOLIC_CONSTANT)
    glVertex*(…);
    glVertex*(…);
    …
glEnd();
```

```
//set red color
glColor3f(1.0, 0.0, 0.0);
//specify 2D square
 glBegin(GL_QUADS);
    glVertex2i(-2, 2);
    glVertex2i(2, 2);
    glVertex2i(2, -2);
    glVertex2i(-2, -2);
glEnd();
```

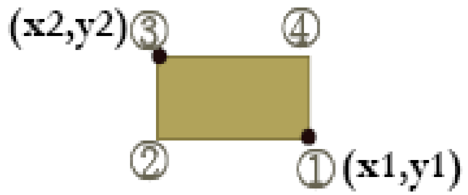


- By default, a polygon interior is displayed in a **solid** color, determined by the current color settings

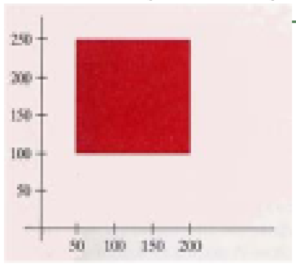
Define Rectangle in OpenGL

- **A special rectangle function of OpenGL**

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



'*': the coordinate data type: i (integer), s (short), f (float), d (double), and v (vector)



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

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

This function is equivalent to :

```
glBegin (GL_POLYGON);
        glVertex2* (x1, y1);
        glVertex2* (x2, y1);
        glVertex2* (x2, y2);
        glVertex2* (x1, y2);
glEnd ();
```

and

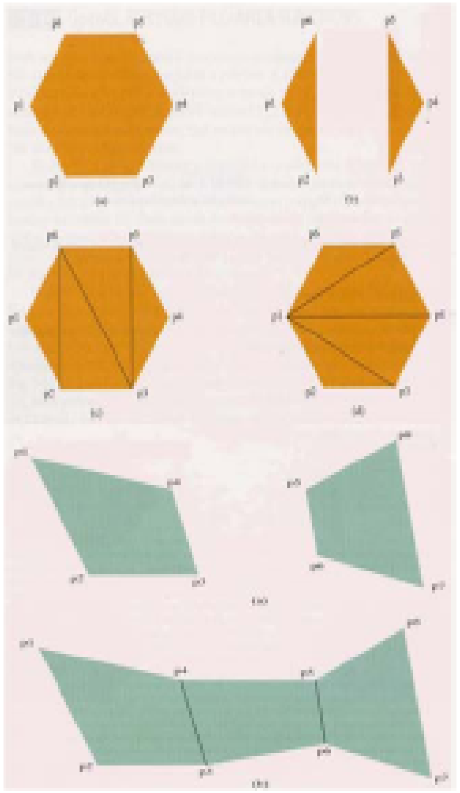
```
glBegin (GL_QUADS);
        glVertex2* (x1, y1);
        glVertex2* (x2, y1);
        glVertex2* (x2, y2);
        glVertex2* (x1, y2);
glEnd ();
```

glRect* is more efficient than using the above glVertex specifications

Six OpenGL Polygon Fill Primitives

To use the symbolic constant in the glBegin function, along with a list of glVertex commands.

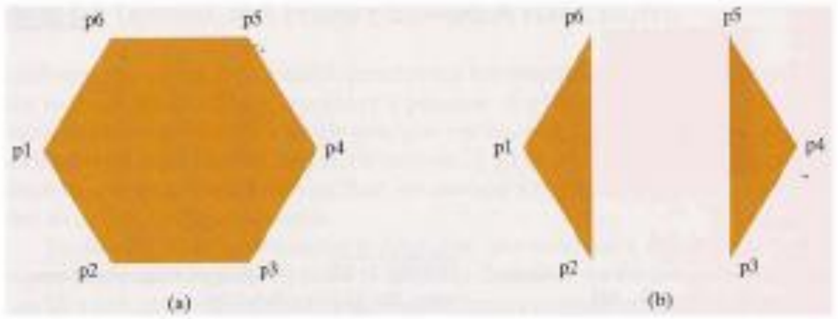
- GL_POLYGON -- closed ploygon
- GL_TRIANGLES -- disconnected triangles
- GL_TRIANGLE_STRIP -- connected triangles
- GL_TRIANGLE_FAN -- triangles sharing common point
- GL_QUADS -- disconnected quadrilaterals
- GL_QUAD_STRIP -- connected quadrilaterals



Same vertices in different order, and with different symbolic constant

OpenGL Polygon Fill-area Functions

- GL_POLYGON and GL_TRIANGLES



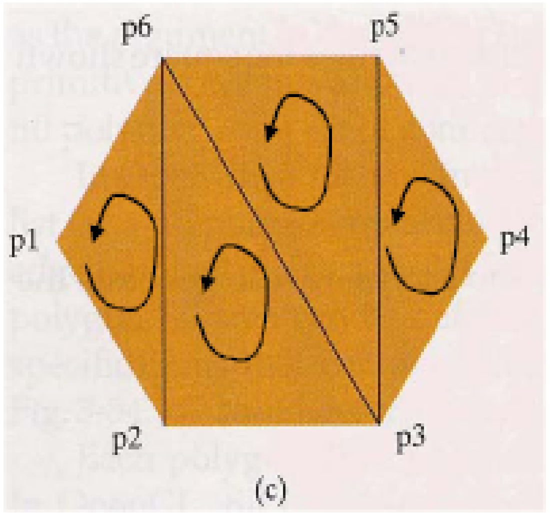
```
glBegin (GL_POLYGON);
   glVertex2iv (p1);
   glVertex2iv (p2);
   glVertex2iv (p3);
   glVertex2iv (p4);
   glVertex2iv (p5);
   glVertex2iv (p6);
glEnd ();
```

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

The orders of the vertices in (a) and (b) between the glBegin() and glEnd() pair are different.

OpenGL Polygon Fill-area Functions

- GL_TRIANGLE_STRIP

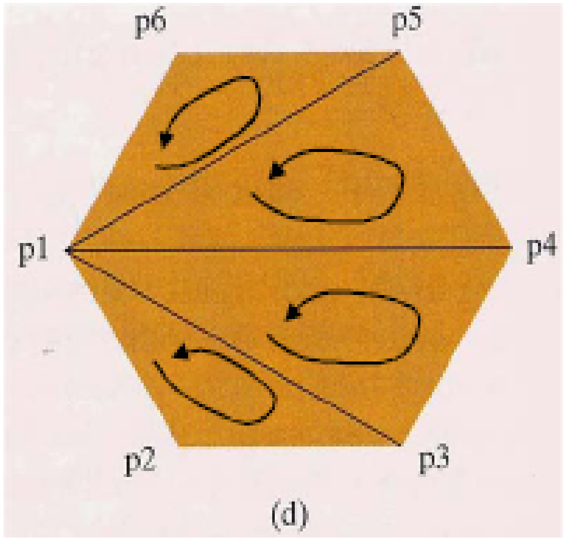


```
glBegin (GL_TRIANGLE_STRIP);
    glVertex2iv (p1);  -> n=1
    glVertex2iv (p2);  -> n=2
    glVertex2iv (p6);  -> n=3
    glVertex2iv (p3);  -> n=4
    glVertex2iv (p5);
    glVertex2iv (p4);
glEnd ();
```

- For N vertices, we obtain N-2 triangles. Each successive triangle shares an edge with the previously defined triangle.
- The **first three points** form the first triangle (**counterclockwise** viewing from the outside), points 2-4 form the second, points 3-5 form the third, and so on.
- The ordering of the vertex list is important to ensure a consistent display.
  - Define each position n in the vertex list in the order 1, 2, …, N-2.
    - If n is odd, the triangle vertices are in the order: n, n+1, n+2; -> (p1, p2, p6)
    - If n is even, the triangle vertices are in the order: n+1, n, n+2. -> (p6, p2, p3)

OpenGL Polygon Fill-area Functions

- GL_TRIANGLE_FAN

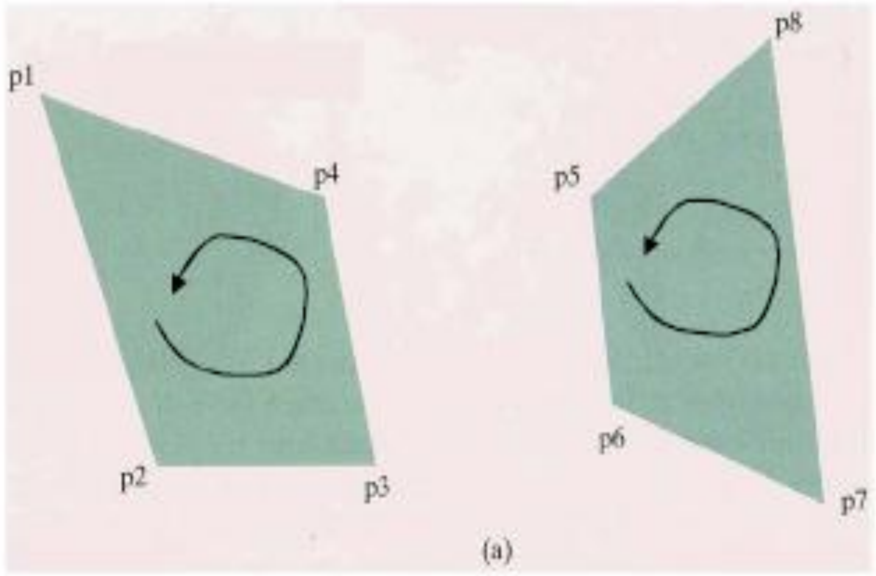


```
glBegin (GL_TRIANGLE_FAN);
    glVertex2iv (p1);  -> n=1
    glVertex2iv (p2);  -> n=2
    glVertex2iv (p3);  -> n=3
    glVertex2iv (p4);  -> n=4
    glVertex2iv (p5);
    glVertex2iv (p6);
glEnd ();
```

- For N vertices, we obtain N-2 triangles.
- The first point is shared by every triangle. Points 1&2&3 define the first one, points 1&3&4 define the second, and so on.
- Define each position n in the vertex list in the order 1, 2, …, N-2.
  - Vertices1, n+1, n+2 define nth triangle; -> (p1, p2, p3); (p1, p3, p4); …

OpenGL Polygon Fill-area Functions

- GL_QUADS

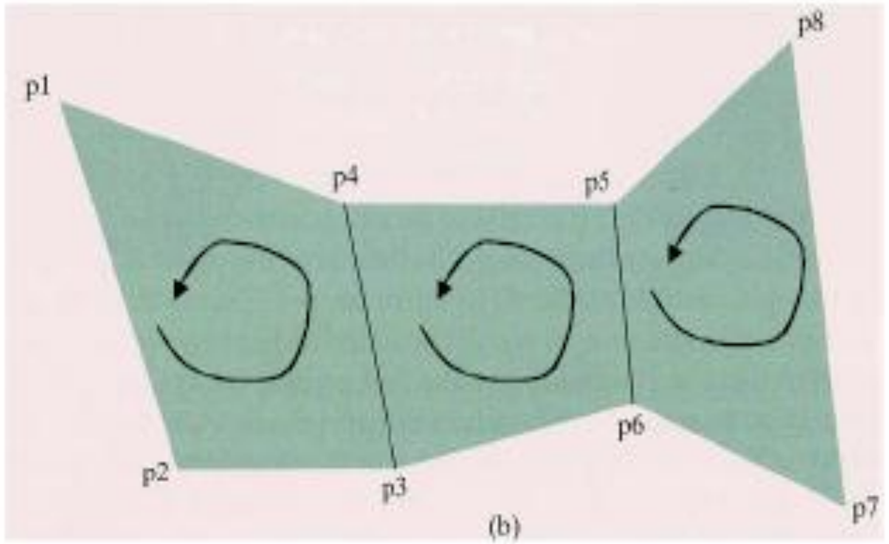


```
glBegin (GL_QUADS);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p7);
    glVertex2iv (p8);
glEnd ();
```

- The first four points form a quadrilateral, the next four points form the second, and so on.
- At least four points should be listed, otherwise, nothing is displayed.

OpenGL Polygon Fill-area Functions

- GL_QUAD_STRIP



```
glBegin (GL_QUAD_STRIP);
    glVertex2iv (p1); -> n=1
    glVertex2iv (p2); -> n=2
    glVertex2iv (p4); -> n=3
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p8);
    glVertex2iv (p7);
glEnd ();
```

➢ Rearrange the vertex list, we can obtain the set of connected quadrilaterals.

➢ Define each position n in the vertex list in the order n=1, n=2, …, n=(N/2)-1.

➢ The vertices 2n-1, 2n, 2n+2, 2n+1 define nth quadrilateral -> (p1, p2, p3, p4); (p4, p3, p6, p5)

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.**

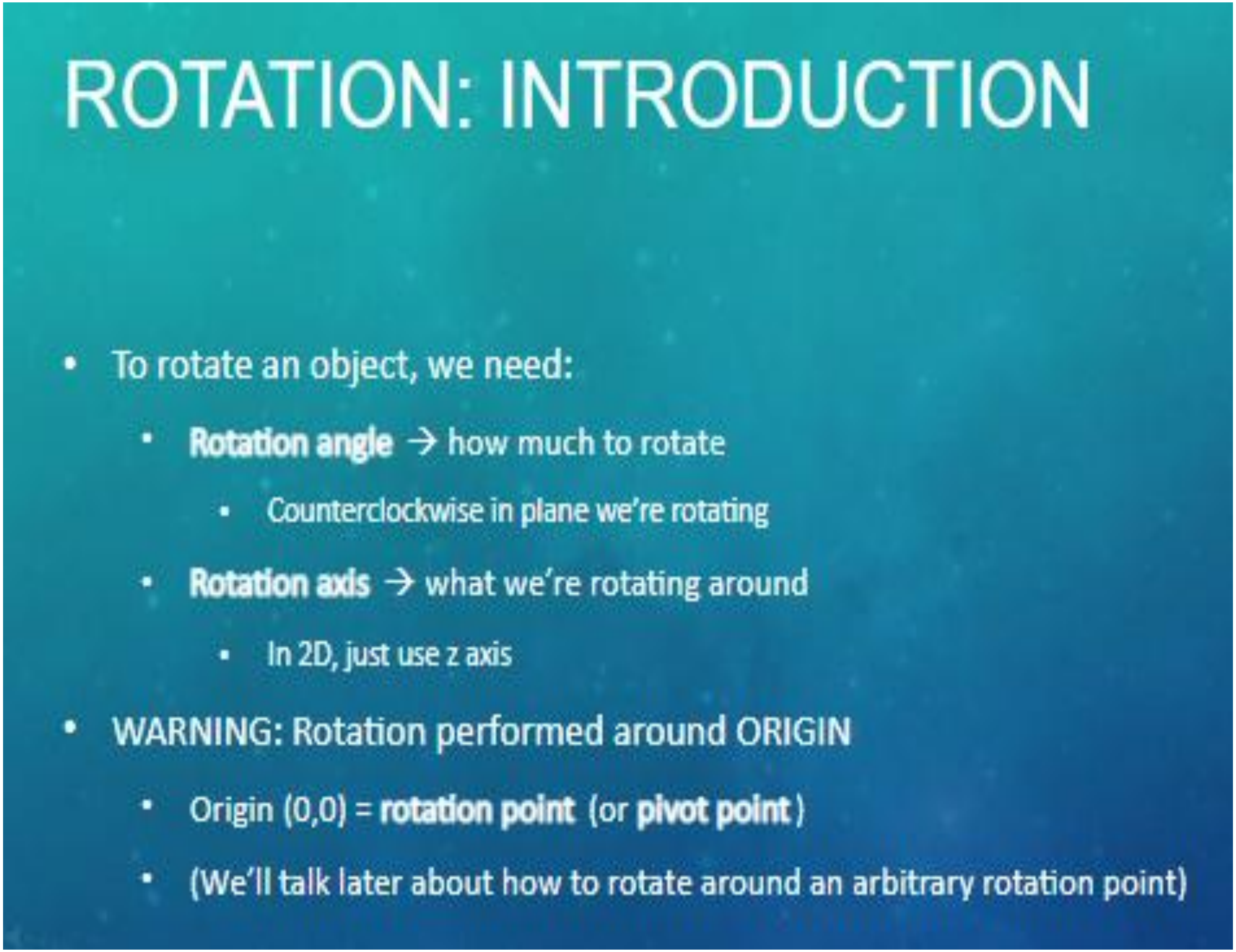
Geometric transformations, on the other hand, can be used to describe how objects might move around in a scene during an animation sequence or simply to view them from another angle.

OVERVIEW

- Three most basic transformations:
  - Rotation
  - Scaling (i.e., resizing the object)
  - Translation (i.e., moving/placing the object)
- *Other transformations*: shear, reflection, etc.

- We're going to start with 2D transformations first
- Then, later, we'll move on to 3D transformations

- Keep in mind, we will be building simple transformations (with certain implicit assumptions)
  - However, we will combine these to make more powerful transformations



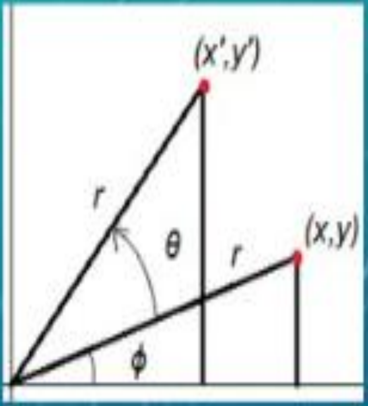
ROTATION: INTRODUCTION

- To rotate an object, we need:
    - **Rotation angle** → how much to rotate
        - Counterclockwise in plane we're rotating
    - **Rotation axis** → what we're rotating around
        - In 2D, just use z axis
- WARNING: Rotation performed around ORIGIN
    - Origin (0,0) = **rotation point** (or **pivot point**)
    - (We'll talk later about how to rotate around an arbitrary rotation point)

ROTATION: DERIVING

Recall:

$$\cos(A+B) = \cos A \cos B - \sin A \sin B$$
$$\sin(A+B) = \cos A \sin B + \sin A \cos B$$



- P = (x,y) → original point
- P' = (x',y') → transformed point

- φ = original angle of point (x,y) from x axis
- θ = difference in angle between old and new point
- So, our original point (x,y) and transformed point (x',y') in polar coordinates are as follows →

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

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

- After substitution, we can express the transformed point in terms of θ only:

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

ROTATION MATRIX

- We have:

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

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

- If our rotation matrix transform is R, then:

$$P' = R \cdot P$$

- Therefore, our **2D rotation matrix** is:

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

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

SCALING: INTRODUCTION

- **Scaling** an object = altering the size of an object
- The scaling we will be doing here → simply multiplying each coordinate by a **scaling factor**: $$x' = x \cdot s_x$$ $$y' = y \cdot s_y$$
- The corresponding scaling matrix transformation → $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

SCALING: FACTORS

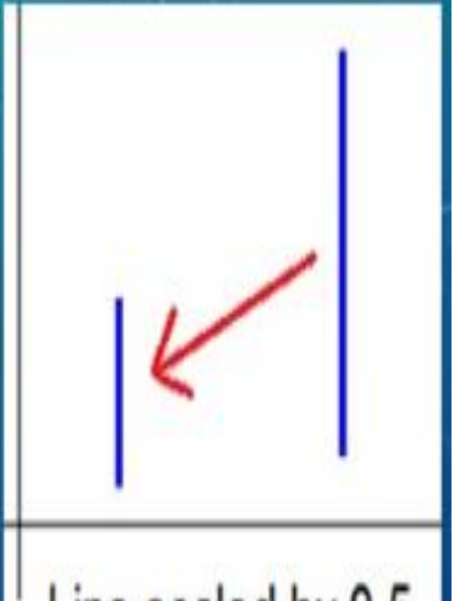
- Scaling factor > 1.0 → enlarge
- Scaling factor < 1.0 → shrink

- Scaling factor < 0 → negative scaling → resizes AND reflects object

- **Uniform scaling** = scaling factors are all the same (e.g., $s_x = s_y$)
  - Otherwise, called **differential scaling**

SCALING: ASSUMPTIONS

- WARNING: Because of the way we are doing scaling:
  - Only scaling in X or Y direction (or both), but NOT in arbitrary direction!
  - Scaling relative to ORIGIN!
    - ORIGIN = **fixed point** (point unaffected by scaling)
    - (We'll talk later about how to use a different fixed point)



Line scaled by 0.5 in x and y: changes size AND moves line closer to origin

TRANSLATION: INTRODUCTION

- **Translation** = moving a point by a certain distance (t_x, t_y)
  - (t_x, t_y) = translation distances = translation vector = shift vector

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

- If we're stuck with 2x2 matrices and 2x1 vectors, we have to add vectors to perform a translation:

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

TRANSLATION: PROBLEM

- At some point, we would like to be able to combine multiple transformations into a single matrix:

$$\begin{aligned} P' &= F \cdot E \cdot D \cdot C \cdot B \cdot A \cdot P \\ &= (F \cdot E \cdot D \cdot C \cdot B \cdot A) \cdot P \\ &= M \cdot P \end{aligned}$$

- This means we can multiply all our transformations together first (M), and then apply it to each point we want
- HOWEVER, because translation is handled as addition, we need to compute intermediate steps:

$$P' = (R \cdot P) + T$$

HOMOGENEOUS COORDINATES

- To fix this, we will extend our 2x2 matrices (and our 2x1 vectors) to 3x3 matrices (and to 3x1 vectors)
- **Homogeneous coordinates** = for 2D coordinates, extension to (x_h , y_h , h)
  - h = **homogeneous parameter** → nonzero value such that:

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

  - Often just set h = 1 → (x, y) becomes (x, y, 1)
  - Often use "w" instead of "h" (especially for 3D vectors → (x,y,z,w))
- As we'll see, this allows us to represent translation as a matrix multiplication!

TRANSLATION MATRIX WITH HOMOGENEOUS COORDINATES

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t_x \\ y+t_y \\ 1 \end{bmatrix}$$

- The 2D translation matrix is sometimes represented as $T(t_x , t_y)$

ROTATION MATRIX WITH HOMOGENEOUS COORDINATES

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

- The 2D rotation matrix is sometimes represented as R(θ)

SCALING MATRIX WITH HOMOGENEOUS COORDINATES

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

- The 2D scaling matrix is sometimes represented as $S(s_x , s_y)$

PATTERN WITH HOMOGENEOUS COORDINATE MATRICES

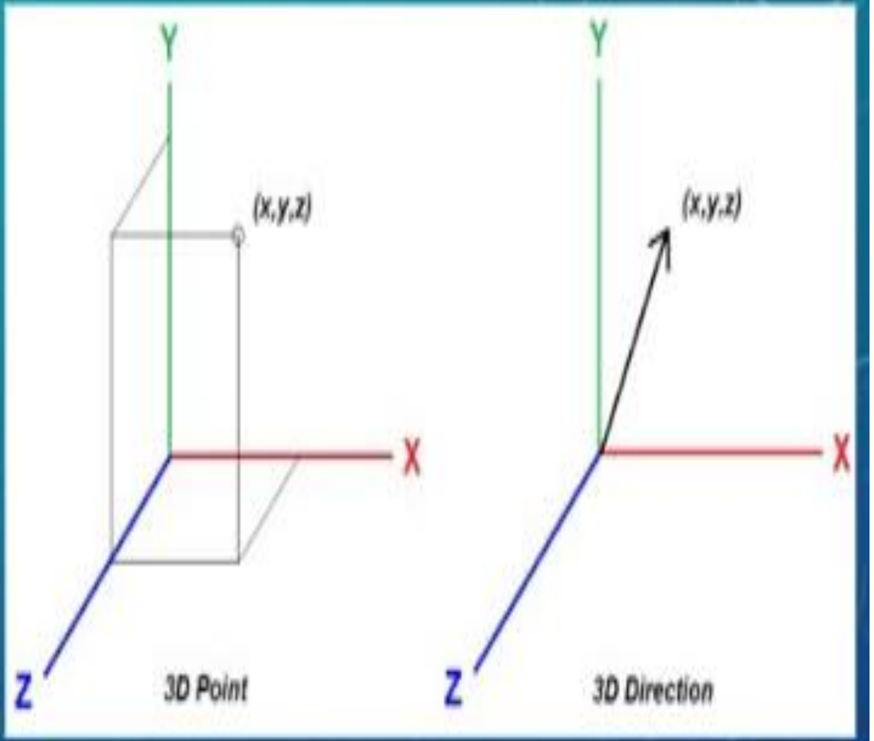
- With the translation matrix, we purposely use the additional elements of the matrix (in this case, the extra column):

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

- For rotation, scaling, and shear matrices (discussed later), the original matrix is augmented with an extra row and column of zeros (except for the last (row,column) position, which is set to 1):

$$M = \begin{bmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad M = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

HOMOGENEOUS COORDINATES: POINTS VS. VECTORS



- *Recall*: a vector can also be interpreted as:
  - **Location** (w = 1)
  - **Direction** (w = 0)
- …in space
  - *Note*: sometimes, location → called "point" and direction → called "vector"
- Depending on how we want to interpret the vector, we will set a different value for w

HOMOGENEOUS COORDINATES: POINTS VS. VECTORS

- Points → all transformations should have an effect (translation, rotation, scaling, etc.)
  - w set to 1

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

- Direction → translation has no meaning (other transformation should work though)
  - w set to 0

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = T \cdot \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Composing Transformation

- Composing Transformation – the process of applying several transformation in succession to form one overall transformation
- If we apply transforming a point P using M1 matrix first, and then transforming using M2, and then M3, then we have:

$(M3 \times (M2 \times (M1 \times P)))$

$$\underset{\text{(pre-multiply)}}{(M3 \times (M2 \times (M1 \times P)))} = \underbrace{M3 \times M2 \times M1}_{M} \times P$$

- Matrix multiplication is associative

  M3 x M2 x M1 = (M3 x M2) x M1 = M3 x (M2 x M1)

- Transformation products may not be commutative A x B != B x A
- Some cases where A x B = B x A

A	B
translation	translation
scaling	scaling
rotation	rotation
uniform scaling (sx = sy)	rotation

S J P N Trust's

Hirasugar Institute of Technology, Nidasoshi.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

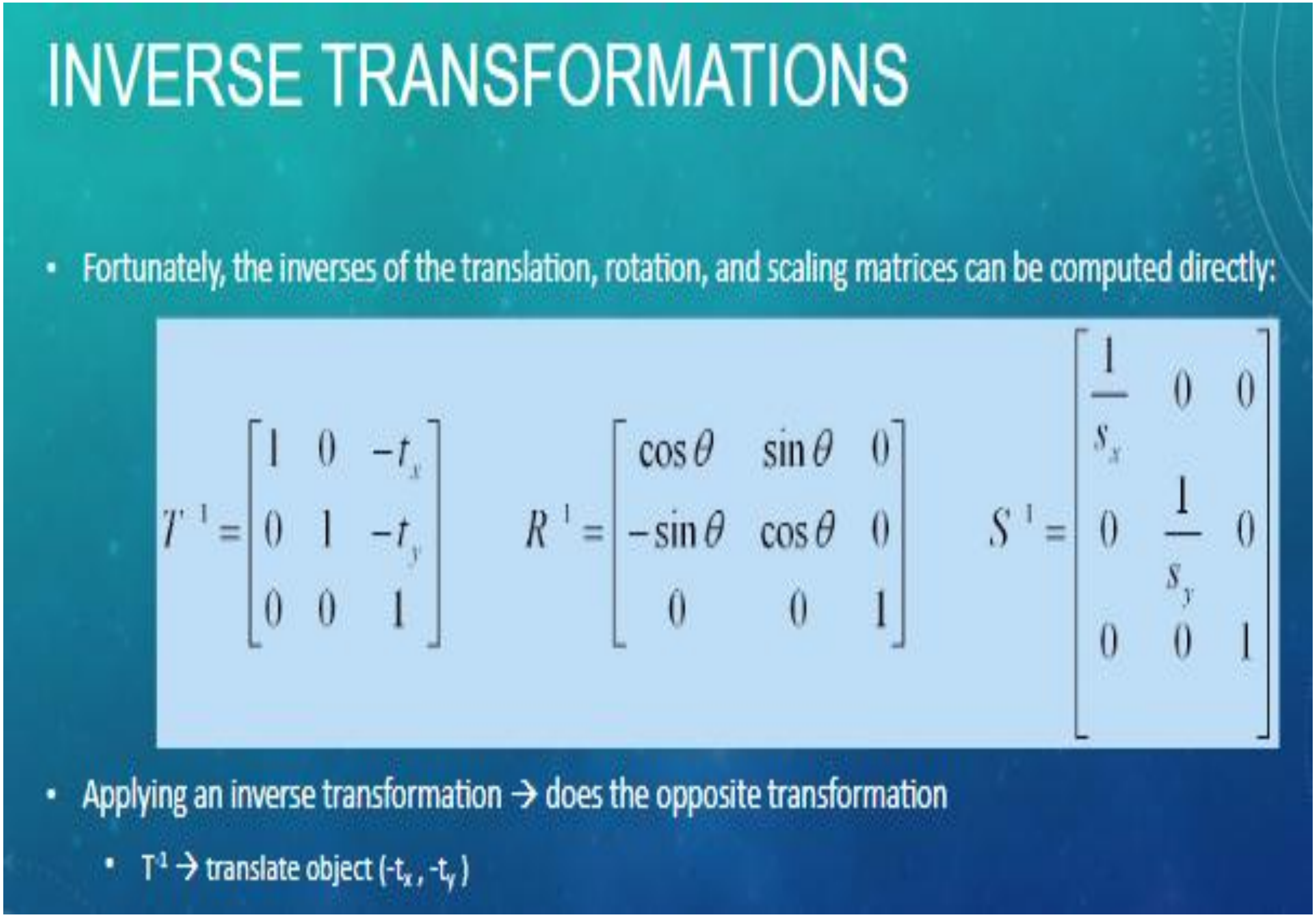
Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

Module-2
18CS62

Fill area Primitives, 2D Geometric Transformations and 2D viewing

Prof. Rahul Palakar



INVERSE TRANSFORMATIONS

- Fortunately, the inverses of the translation, rotation, and scaling matrices can be computed directly:

$$T^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \qquad R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad S^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

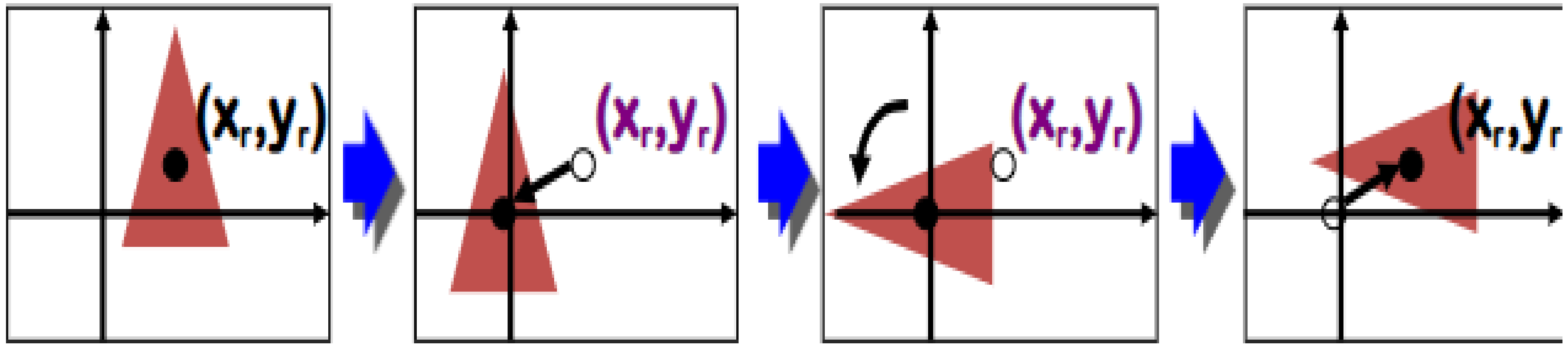
- Applying an inverse transformation → does the opposite transformation
  - T^{-1} → translate object $(-t_x, -t_y)$

QUICK ASIDE: INVERSE OF ROTATION MATRIX

- We computed the inverse directly by using the negative angle (-θ) → only sine was affected by this
- It turns out, any rotation matrix is ORTHOGONAL → inverse = transpose = swapping rows and columns

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

2D Pivot(Arbitrary) point rotation



- So we can generate a 2D rotation about any other pivot point (x, y) by performing the following sequence of translate-rotate-translate operations

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

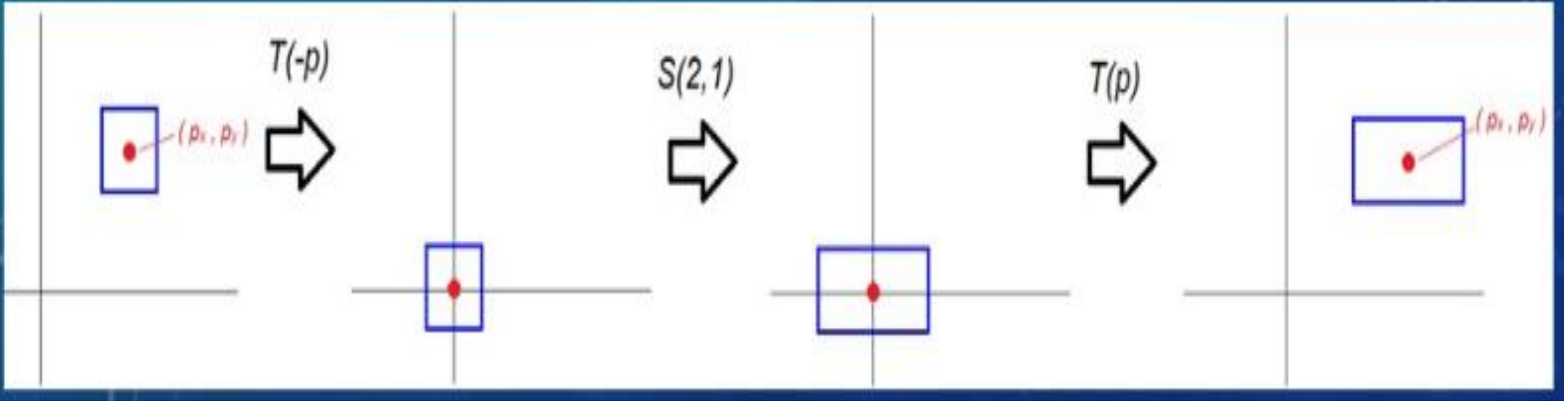
$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta)+y_r\sin\theta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

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

SCALING AROUND AN ARBITRARY FIXED POINT

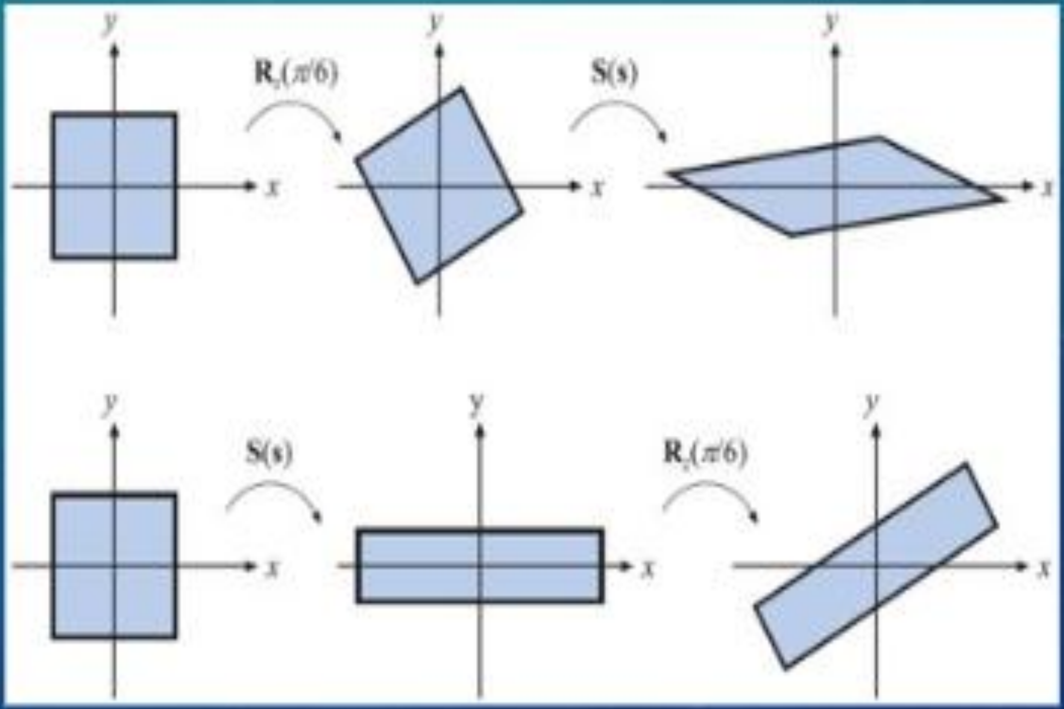
- Say we want to scale an object relative a fixed point (p_x , p_y)
- *Basic idea*:
  - Translate ($-p_x$, $-p_y$) → (p_x , p_y) is now at origin
  - Scale points
  - Translate back to (p_x , p_y)

$$T(p_x, p_y)\cdot S(s_x, s_y)\cdot T(-p_x, -p_y)$$
$$= \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{bmatrix}$$



ORDER MATTERS!

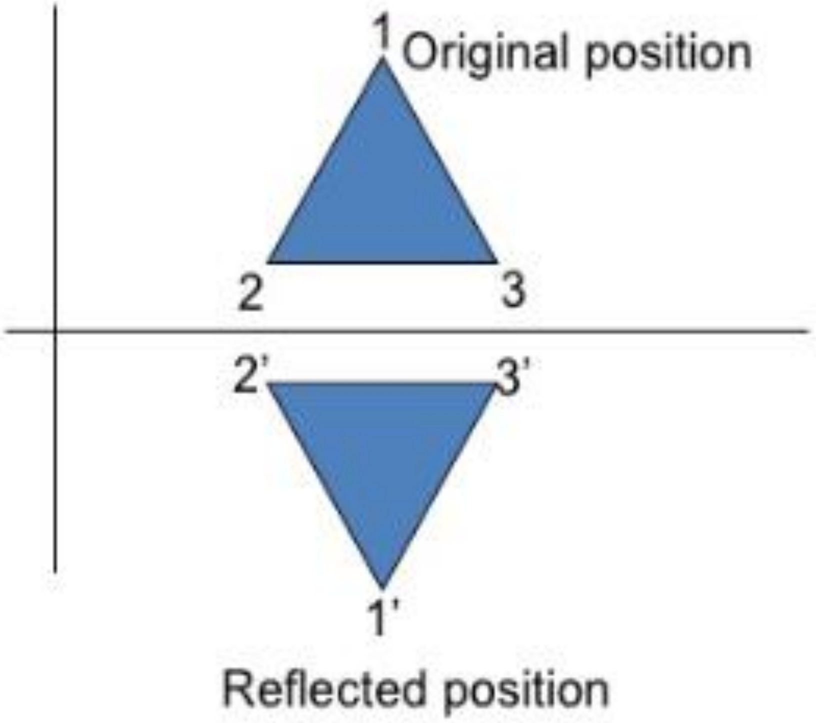
- What order you apply your matrices will affect what transformations you perform!!!
  - *Example*: rotation then scale vs. scale then rotation



- First transformation → RIGHT-most matrix when multiplying!

Other transformations

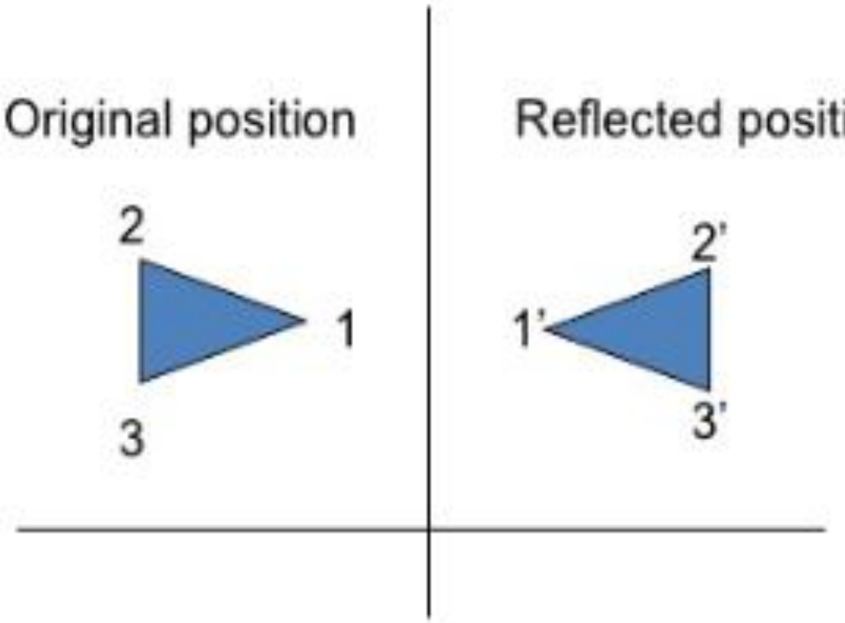
- **Reflection** is a transformation that produces a mirror image of an object. It is obtained by rotating the object by 180 deg about the reflection axis



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

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

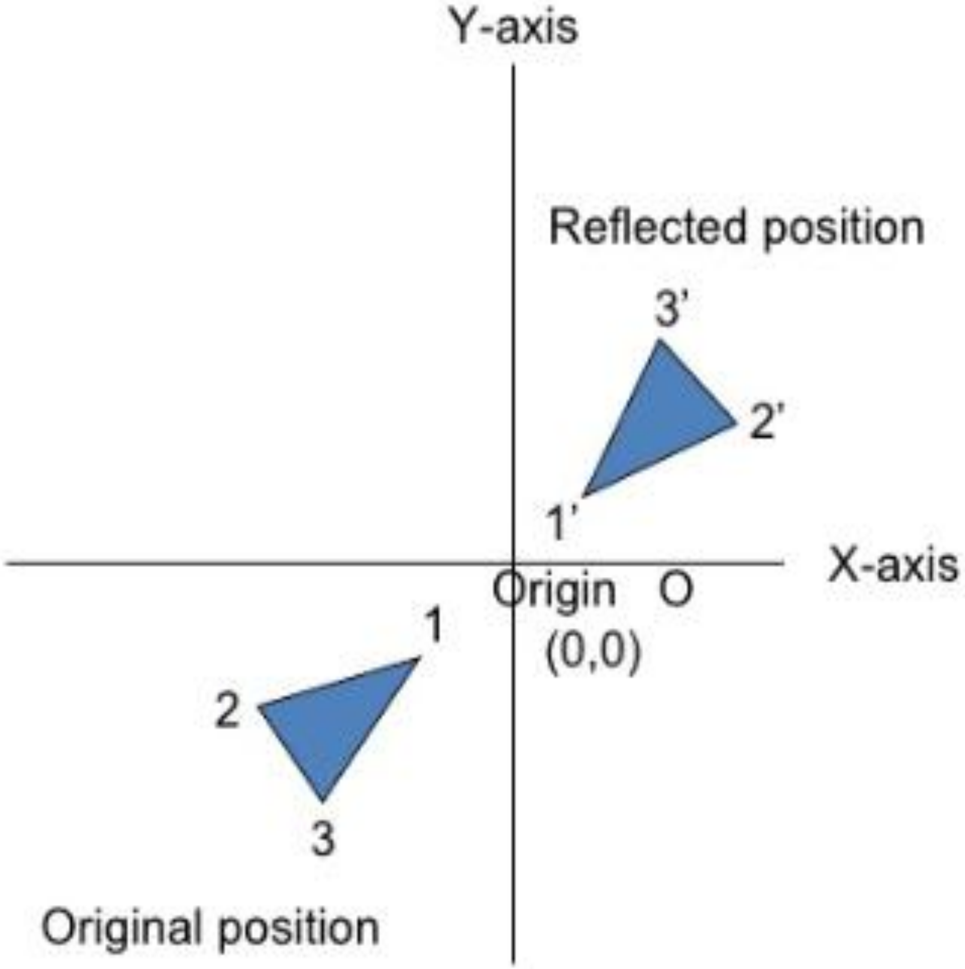
Reflection



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

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

Reflection of an object relative to an axis perpendicular to the xy plane and passing through the coordinate origin

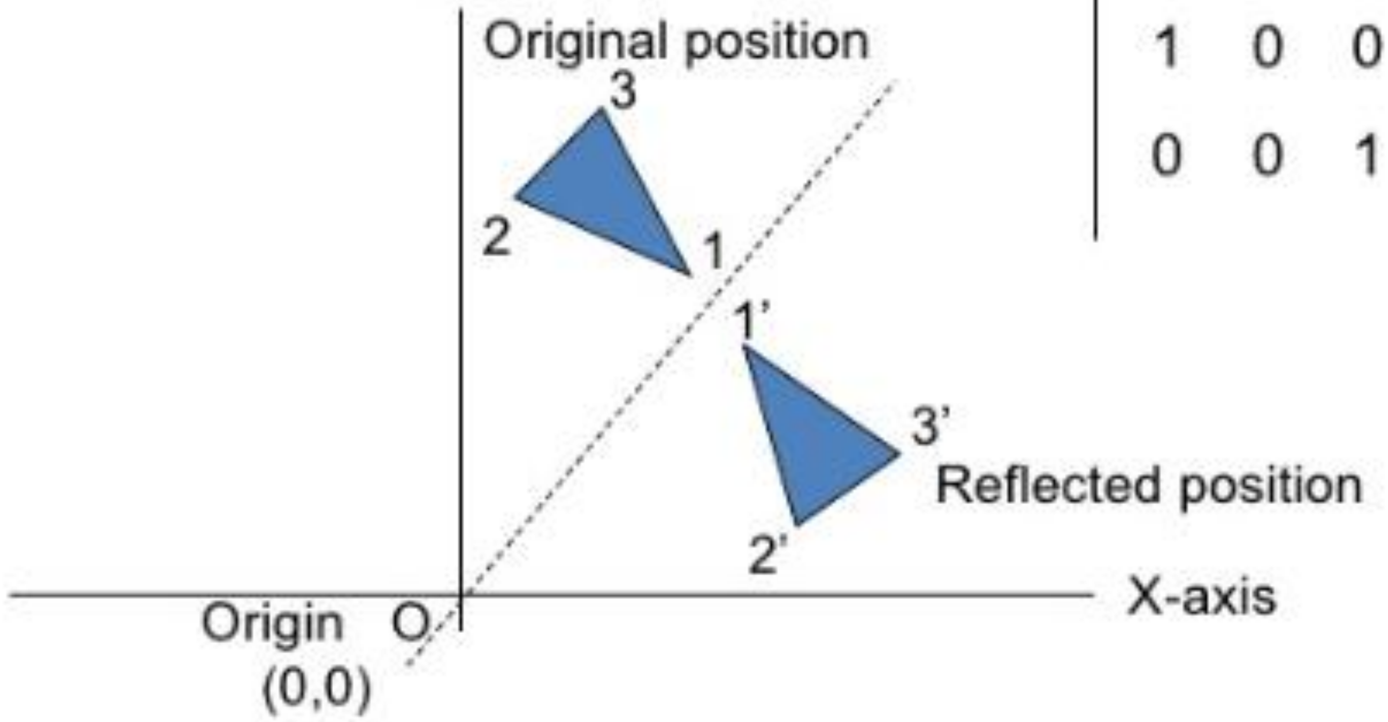


$$\begin{vmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The above reflection matrix is the rotation matrix with angle=180 degree.

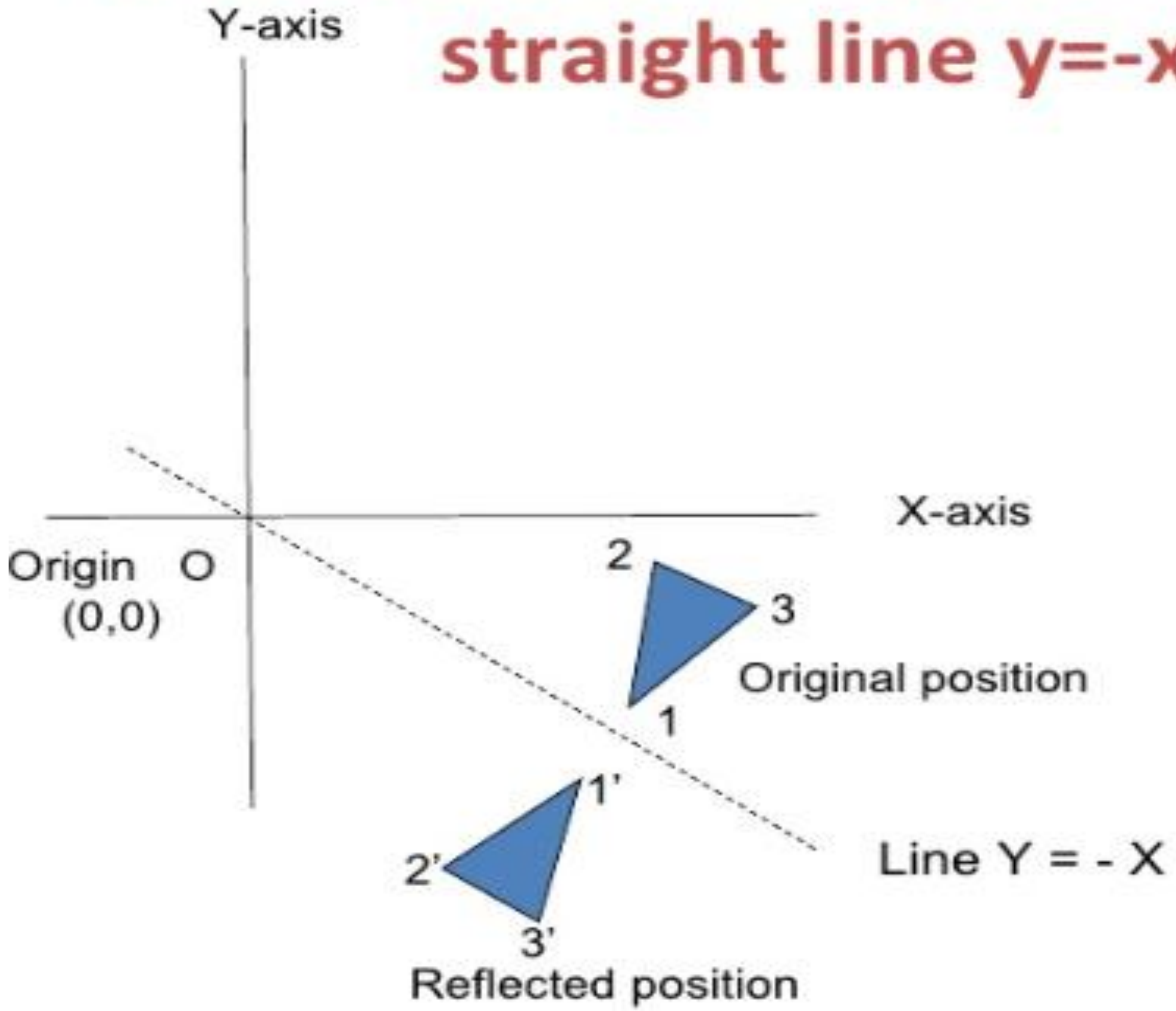
This can be generalized to any reflection point in the xy plane. This reflection is the same as a 180 degree rotation in the xy plane using the reflection point as the pivot point.

Reflection of an object w.r.t the straight line y=x



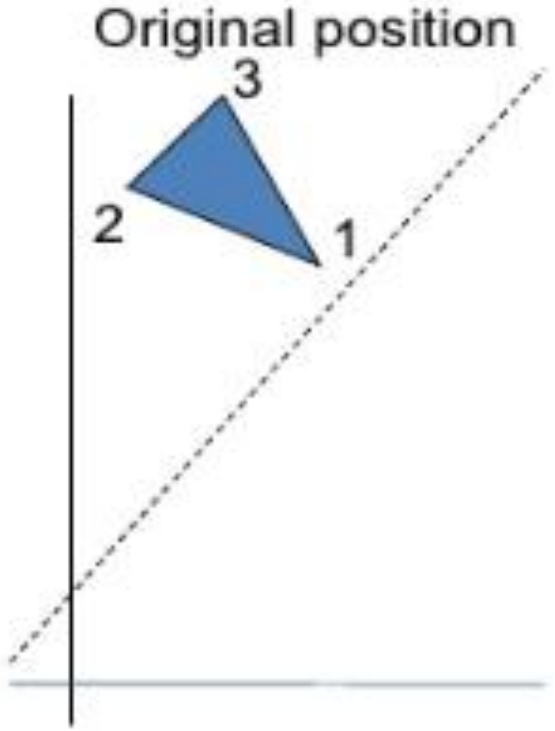
$$\begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

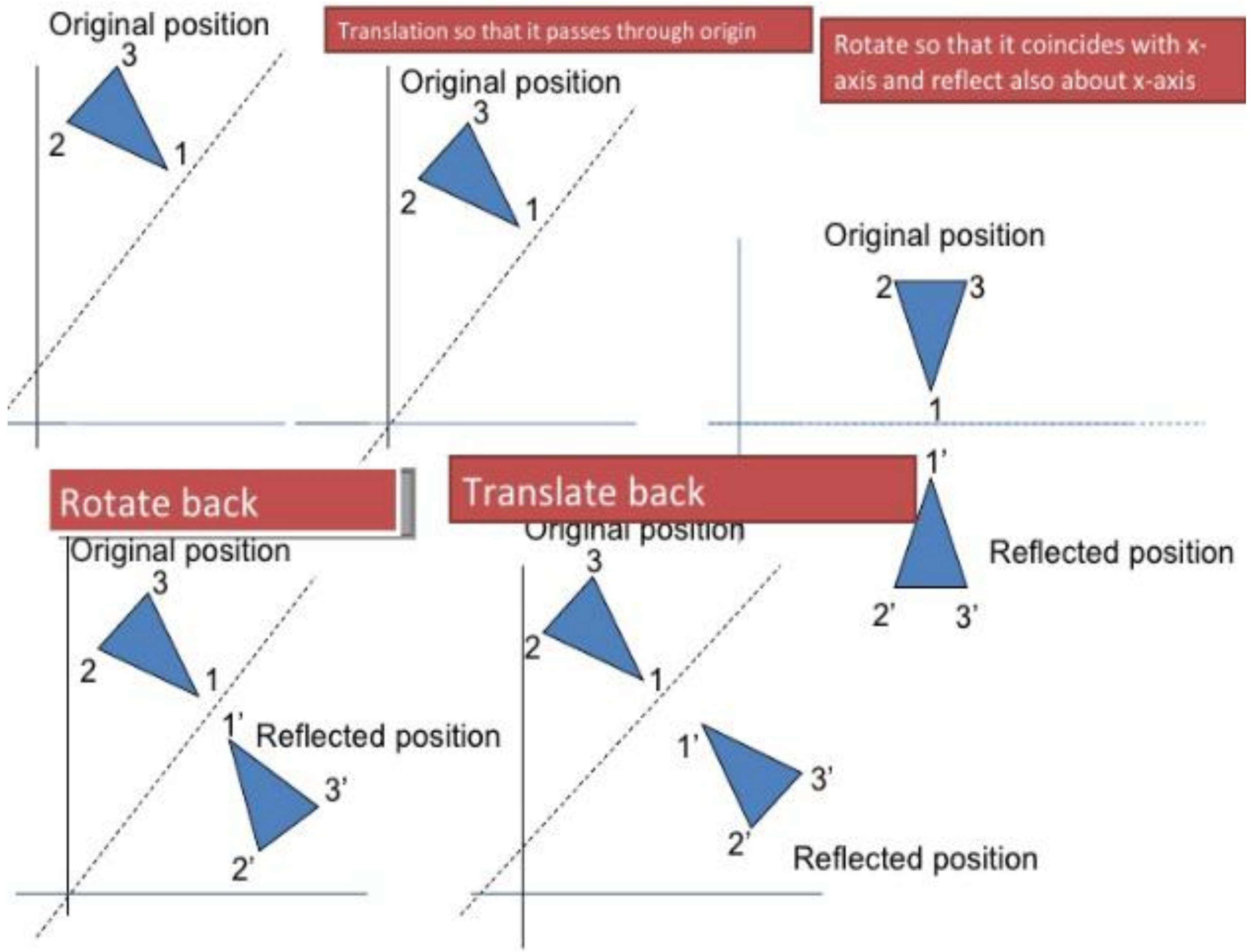
Reflection of an object w.r.t the straight line y=-x



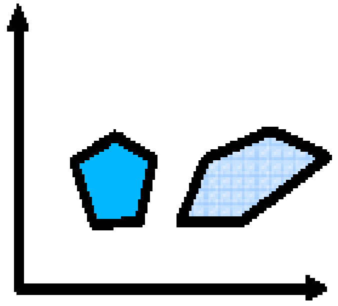
$$\begin{vmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

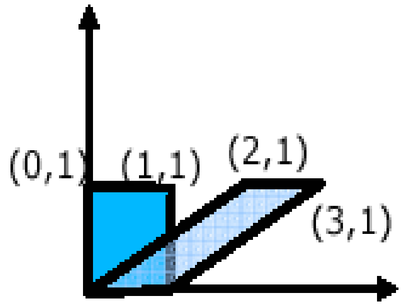
Reflection of an arbitrary axis y=mx+b




Original position
3
2
1
Translation so that it passes through origin
Original position
3
2
1
Rotate so that it coincides with x-axis and reflect also about x-axis
Original position
2
3
1
1'
Reflected position
2'
3'
Rotate back
Original position
3
2
1
1'
Reflected position
3'
2'
Translate back
Original position
3
2
1
1'
3'
2'
Reflected position

- Shear: Deform the shape like shifted slices.





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

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

Shear in x:

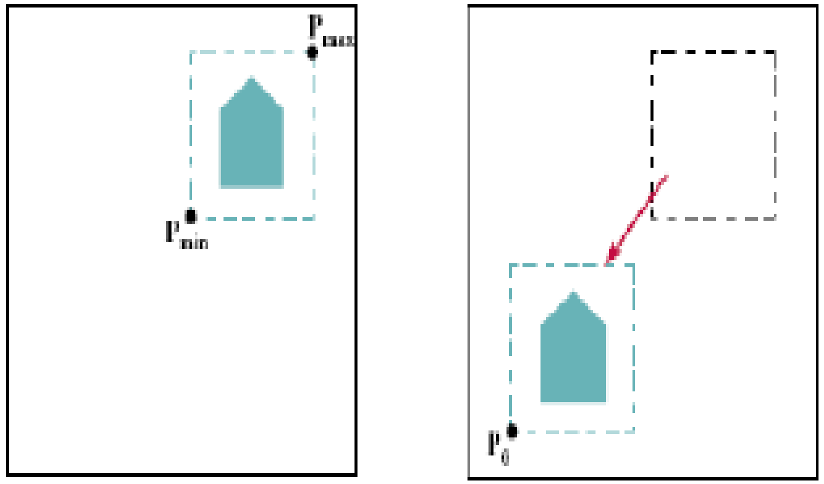
$$Sh_x = \begin{bmatrix} 1 & sh_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + sh_x y \\ y \end{bmatrix}$$

Shear in y:

$$Sh_y = \begin{bmatrix} 1 & 0 \\ sh_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ sh_y . x + y \end{bmatrix}$$

Raster Methods for Geometric Transformations

- All bit settings in the rectangular area shown are copied as a block into another part of the frame buffer



(a) (b)

- 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

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

(a) (b) (c)

- For array rotations that are not multiples of 90°, we need to do some extra processing
- Similar methods to scale a block of pixels

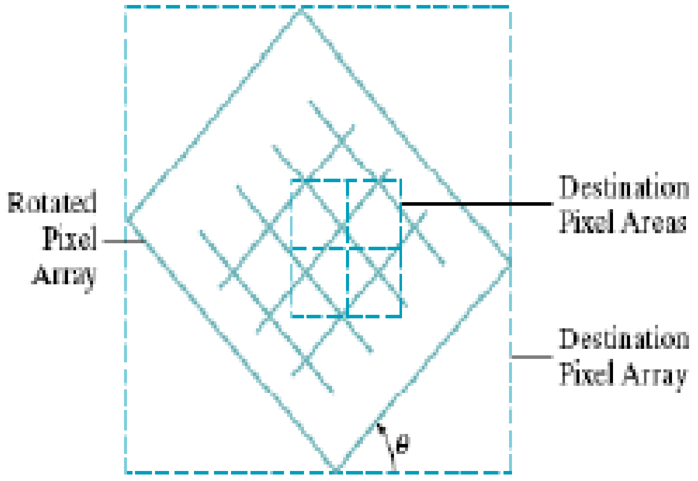


FIGURE 5-28 A raster rotation for a rectangular block of pixels can be accomplished by mapping the destination pixel areas onto the rotated block.

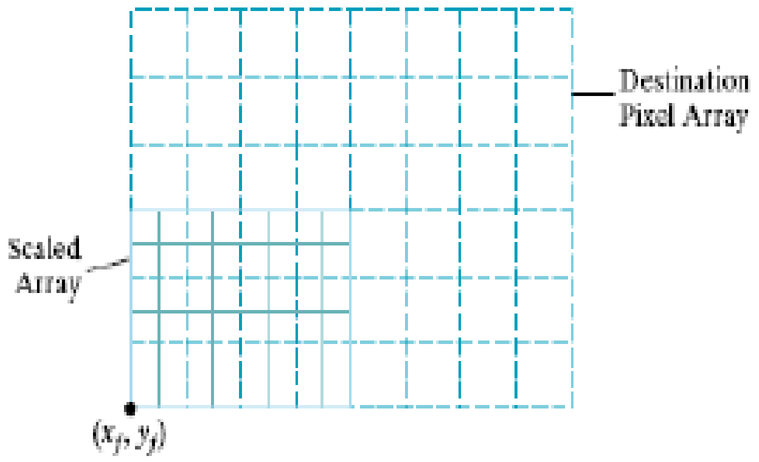


FIGURE 5-29 Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point (x_f, y_f).

OpenGL Raster Transformations

- Copying pixels from one buffer area to another can be accomplished with

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

- GL_COLOR says what is to be copied (color values)
- Copied to refresh buffer at same loc

To read into an array:

glReadPixels(xmin, ymin, width, height,

GL_RGB, GL_UNSIGNED_BYTE, colorArray);

- To do a 90 degree rotation could rearrange rows and columns of array, then place back to refresh buffer at current raster position

glDrawPixels(width, height, GL_RGB,

GL_UNSIGNED_BYTE, colorArray);

To scale an area use:

glPixelZoom(sx,sy);

- where sx and sy are any nonzero floating-point values. (Negative values cause reflections).

Then use **glCopyPixels** or **glDrawPixels** to get/draw the pixels with the given scaling.

OpenGL Transformations **Functions**

Translation

Offset (tx, ty, tz) is applied to all subsequent coordinates. Effectively moves the origin of coordinate system.

$x' = x + \text{tx}$, $y' = y + \text{ty}$, $z' = z + \text{tz}$

- OpenGL function is glTranslate

glTranslatef(tx, ty, tz);

Rotation

- Expressed as rotation through angle θ about an axis direction (*x,y,z*) .
- OpenGL function – **glRotatef (θ, *x,y,z*)**. So

glRotatef(30.0, 0.0, 1.0, 0.0) rotates by 30° about *y*-axis.

- Note carefully: – glRotate wants angles in degrees. C math library (sin, cos etc.) wants angles in radians.

***degs* = *rads* * 180/π;   *rads* = *degs* * π / 180**

- Positive angle? Right hand rule: if the thumb points along the vector of rotation, a positive angle has the fingers curling towards the palm.

Scaling

- Multiply subsequent coordinates by scale factors sx, sy, sz. (Note: these are not a point, not a vector, just 3 numbers)

$x' = sx * x$, $y' = sy * y$, $z' = sz * z$

- Often sx = sy = sz for a *uniform* scaling effect. If the factors are different, the scaling is called *anamorphic*.
- OpenGL function – glScale For example,

glScalef(0.5,0.5,0.5);

- would cause all objects drawn subsequently to be half as big.

OpenGL 2D Viewing Functions

- OpenGL Projection Mode

  glMatrixMode (GL_PROJECTION); //projection matrix

  glLoadIdentity ();

- GLU Clipping-Window Function

  gluOrtho2D (xwmin, xwmax, ywmin, ywmax);

  2D parallel projection.

- OpenGL Viewport Function

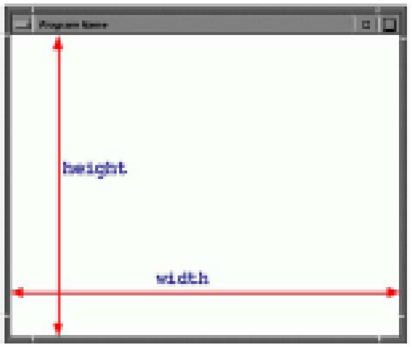
  glViewport (xvmin, yvmin, vpWidth, vpHeight);

  glGetIntegerv (GL_VIEWPORT, vpArray);

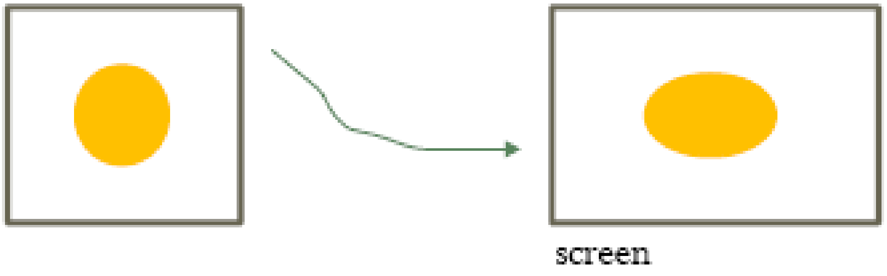
  To obtain the parameters for the currently active viewport: xvmin, yvmin, vpWidth, vpHeight.

OpenGL Viewport Function

- The rectangle area has an aspect ratio: width / height
  - Windows
    - glutInitWindowSize (width, height);
  - 2D clipping window
    - gluOrtho2D (left, right, bottom, top);
  - Viewport
    - glViewport (x, y, width, height);



- In general, the clipping window (viewing volume) and viewport need to have the same ratio.



Thank you