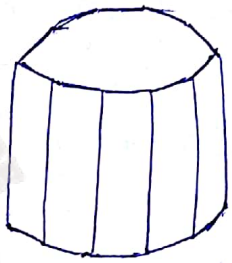
## Fill-Area Primitives

* Describing components of a picture - is an area that is filled with some solid color or pattern. A picture component of this type is refered to as fill area or filled area.
* Approximating a curved surface with polygon facets is sometimes tessellation or fitting the surface with polygon mesh.



Wire frame representation for a cylinder

## Polygon Fill Areas

Polygon → is a plane figure specified by a set of 3 or more co-ordinate positions called vertices., that are connected in sequence by straight-line segments called edges or sides of the polygon.

Polygon must have all its vertices within a single plane and there can be no edge crossing (standard polygon or simple polygon).

Problems → for a computer-graphics appln, it is possible that all the set of polygon vertices do not lie exactly in one plane. due to round off error in the calculation of numerical values, to errors in selecting co-ordinate positions of vertices or approximating a curved surfaced with a set of polygonal patches

## Polygon classifications

* An interior angle → is an angle inside the polygon boundary that is formed by two adjacent edges.
* If the interior angle of a polygon are less than or = 180°, the polygon is convex
* A convex polygon is that its interior lies completely on one side of the infinite extension line of any one of its edges.
* A polygon that is not convex is called concave polygon
* The term degenerate polygon is often used to describe a set of vertices that are collinear (generating a line segment) or that have repeated co-ordinate positions - can generate a polygon shape with extraneous lines (overlapping edges or edges that have a length equal to 0. (rejected by graphics packages)


Convex polygon  Concave polygon

## Problems with concave polygons

1) Implementing fill algorithms & other graphics routines are more complicated → so it is generally more efficient to split a concave polygon into a set of convex polygons before processing.
2) concave polygon splitting is often not included in graphics library

Some graphics packages, including OpenGL, requires all fill polygons to be convex.

## Identifying Concave Polygons.

→ A concave polygon has at 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 a polygon boundary.

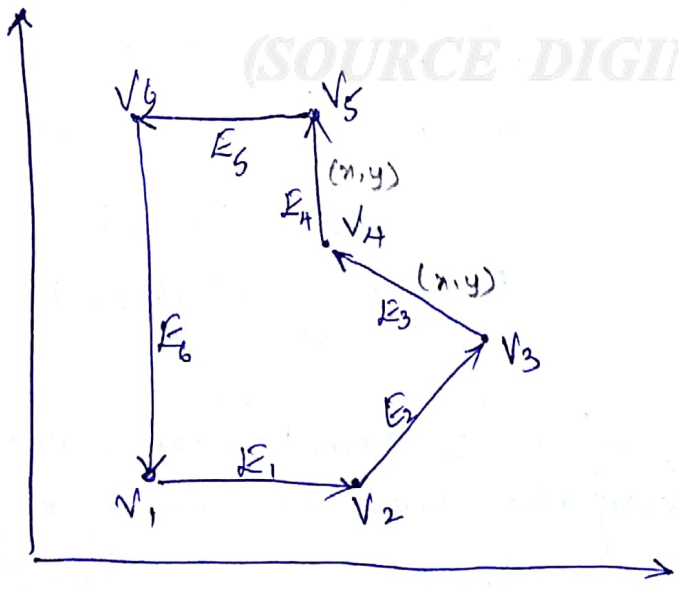
→ Set up vertex for all edges

→ Perform cross product to adjacent vectors to test for concavity.

→ Perform dot product if we want to determine the angle between two edges.

→ All vector products will be same value (+ve or -ve) for convex polygon.

→ If there are some cross products yield a positive & some yield negative value, we have a concave polygon.




$(E_1 \times E_2)_z > 0$

$(E_2 \times E_3)_z > 0$

$(E_3 \times E_4)_z < 0$

$(E_4 \times E_5)_z \not> 0$

$(E_5 \times E_6)_z > 0$

$(E_6 \times E_1)_z > 0$

$E_{3x} E_{4y} - E_{3y} E_{4x}$

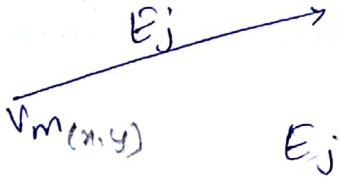
## Splitting Concave Polygons

→ Splitting can be accomplished using edge vectors & 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 & which are on the other

→ Vector method for splitting a concave polygon.




Dot product

$E_j = (V_{nx} - V_{mx}, V_{ny} - V_{my})$

$E_j \times E_k = E_{jx} E_{ky} - E_{jy} E_{kx}$

for 2 successive edges vectors is a vector perpendicular to xy plane with z component equal to

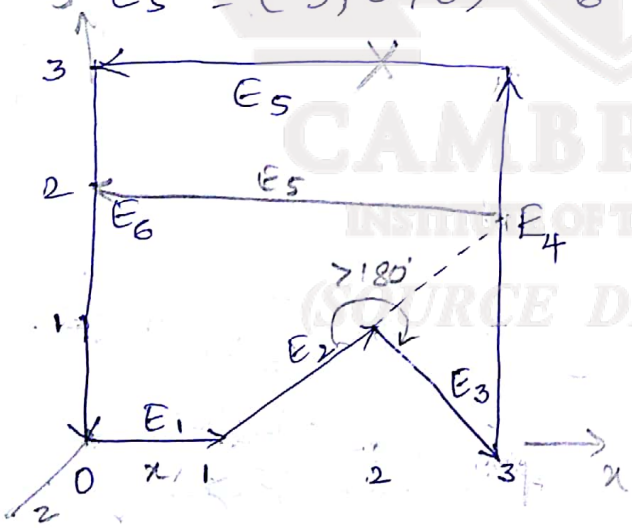
Ex: Vector method. – concave polygon with six edges

$E_1 = (1, 0, 0)$ $E_2 = (1, 1, 0)$

$E_3 = (1, -1, 0)$ $E_4 = (0, 2, 0)$

$E_5 = (-3, 0, 0)$ $E_6 = (0, -2, 0)$

since z component is '0', all edges are in xy plane




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

$E_1 \times E_2 = (0, 0, 1)$ 1 − 0 = 1

$E_2 \times E_3 = (0, 0, -2)$ 1 − 1 = −2

$E_3 \times E_4 = (0, 0, 2)$ 2 − 0 = 2

$E_4 \times E_5 = (0, 0, 6)$

$E_5 \times E_6 = (0, 0, 6)$

$E_6 \times E_1 = (0, 0, 2)$

→ Since $E_2 \times E_3$ is negative z component, we split the polygon ~~line~~ along the line of vector $E_2$

→ line eqn of this edge has slope 1 & y intercept −1

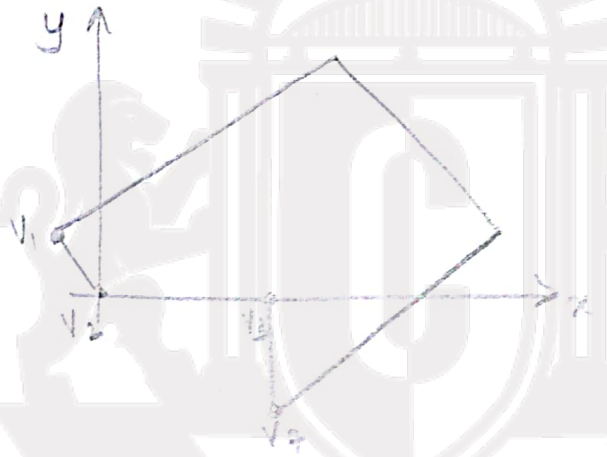
→ hence polygon edges split the polygon into Two pieces.

→ No other edge cross product is negative, so Two new polygons are convex.

## Splitting a convex polygon into a set of triangles

→ Once a convex polygon is obtained with a vertex list, we could transform into a set of triangles.

→ any sequence of three consecutive vertices are formed, to be a new polygon (a triangle)




Rotational Method - for concave polygons

→ shift the position of polygon, vertex $V_k$ at the co-ordinate origin.

→ rotate the polygon about the origin in ~~anticlock~~ wise so that the next vertex $V_{k+1}$ is on the x axis.

→ if polygon ~~is~~ has vertex $V_{k+2}$ below x-axis the polygon is concave.

## Inside - Outside Tests

→ With complex objects, we may have to specify a complex fill region with intersecting edges.

→ for such shapes, we must decide how to determine whether a given point is inside or outside the polygon.

→ conceptually, the process of filling the inside of a polygon with a color or pattern is equivalent to deciding which points in the plane of the polygon are interior (inside) points or exterior (outside).

There are two types of test.

i) Crossing (or) Odd - even test

2) Winding number test

### 1. Odd - Even test

This test is most widely used for making inside - outside decisions.

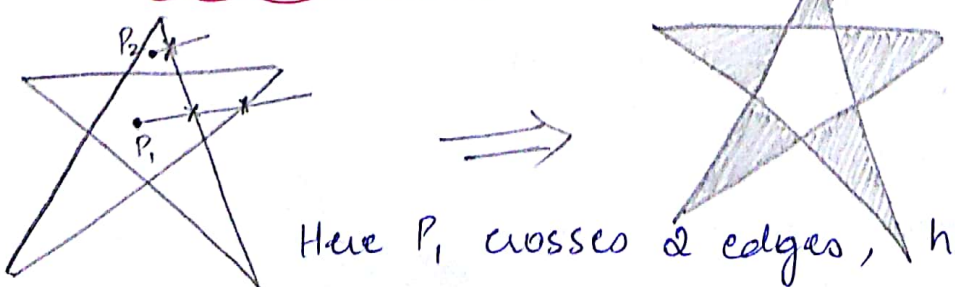
→ Drawing a line from any position 'P' (inside) to a distant point outside the co-ordinate extents of the closed polyline.

→ count the number of line-segment crossings along this line

→ If the number of segments crossed by this line is odd - P is considered to be interior point

→ otherwise P is exterior.

Ex:-



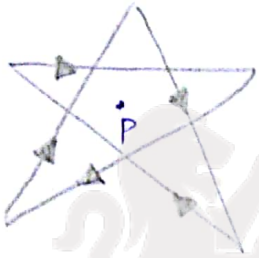

Here $P_1$ crosses 2 edges, hence outside or exterior. $P_2$ crosses 1 edge, hence interior / inside

2) Winding number Test [Non-zero winding number rule]

This test fills the complete star rather than in previous test

→ To implement this test, we consider traversing the edges of the polygon from any starting vertex and going around the edge in a particular direction (which any direction) until we reach the starting point.

→ we illustrate the path by labeling the edges, as shown in the below fig.




→ labeling the edges

→ consider an arbitrary point. Initial the winding number is set to 'zero'.

→ Winding number, which counts the number of ~~objects~~ times the boundary of an object "winds" around a particular point in counterclockwise direction.

→ count clockwise as positive (+1) or add 1 to windowing number when it intersects a segment that crosses the line in clockwise direction.

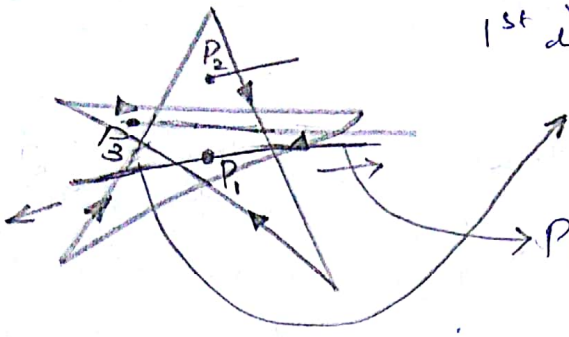
→ count counter anti clockwise as -1. negative.

→ If windowing number is non-zero, P is Interior.
If windowing number is zero, P is exterior

→ All ~~edges~~ points must cross edges not vertices

(cont)

Ex 1.




⇒ $P_1$ crosses 2 edges,
1st is from right to left.

$P_1 = +1+1 = 2$ (Inside)
from vertex $P_1$ to left

$P_1 = -1-1 = -2$ (Inside)
from vertex $P_1$ to right

⇒ $P_2$ = crosses 1 edge from right to left

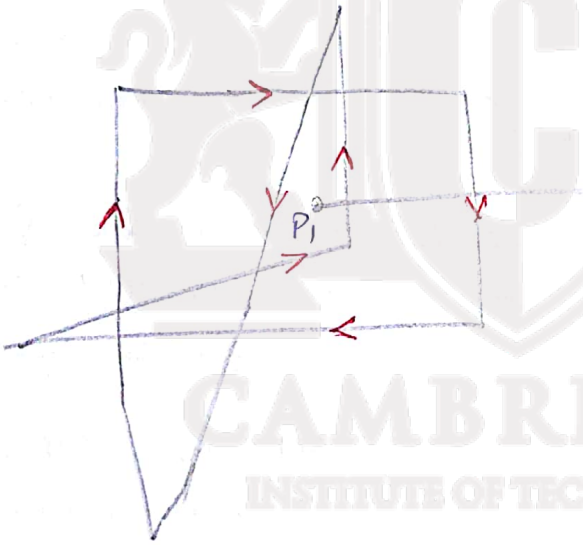
$P_2 = +1$ (Inside)

⇒ $P_3$ = crosses 3 edge.

$= -1+1+1$

$P_3 = 1$ (Inside)

Ex 2.




$P_1 = -1+1$

$P_1 = 0$ (outside)



## Polygon tables

→ The description of objects includes co-ordinate information specifying the geometry for the polygon facets and other surface parameters such as color, transparency, light-reflection properties

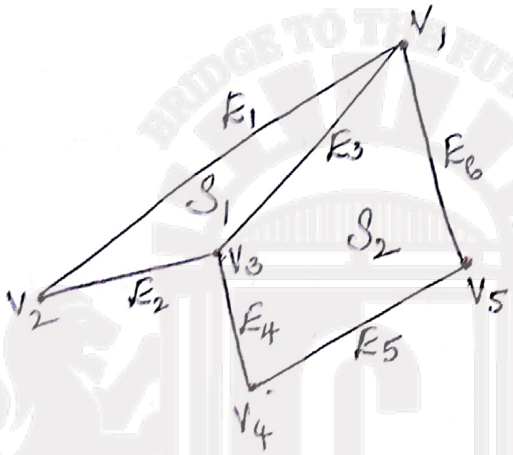
→ As information for each polygon is input, the data are placed into polygon tables for further processing,

display & manipulation of objects

Polygon data tables are organised into 2 groups

① Geometric tables contain vertex co-ordinates & parameters to identify the spatial orientation of polygon surfaces

* a Vertex table * edge table * 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

$S_1$ : $E_1\ E_2\ E_3$
$S_2$ : $E_3\ E_4\ E_5\ E_6$

Vertex table + Surface-facet table → is less convinient & some edges could be drawn twice in a wire-frame display.

Surface-facet table → duplicates co-ordinate information

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

Edge table expanded to include pointers into surface-facet table

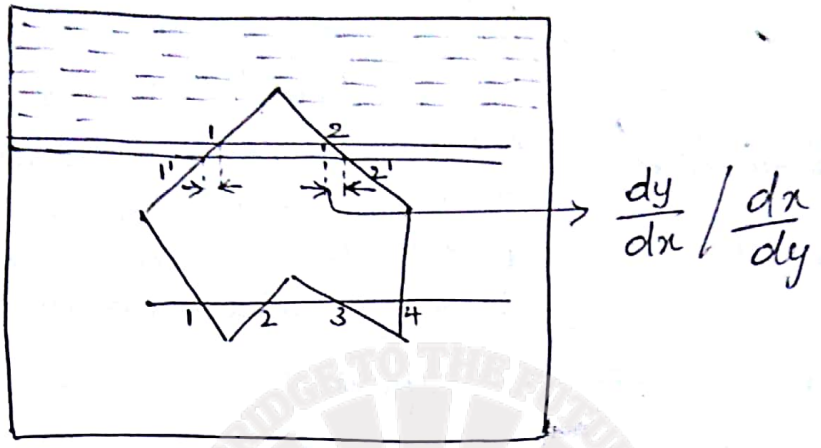
There is importance to check the consistency & completeness of data.

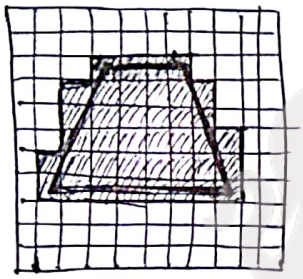
Some tests that could be performed by graphics packages are

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

# Scanline Algorithm / scan conversion

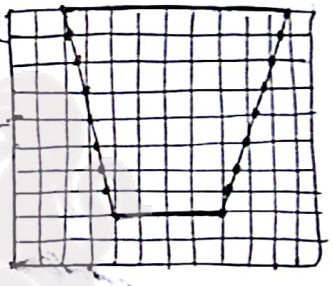
## General Scan-Line Polygon-fill Algorithm



4 Vertices forming polygon, aliasing is seen Pixels are at the center of the grid

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



→ Not drawing lines, but filling regions

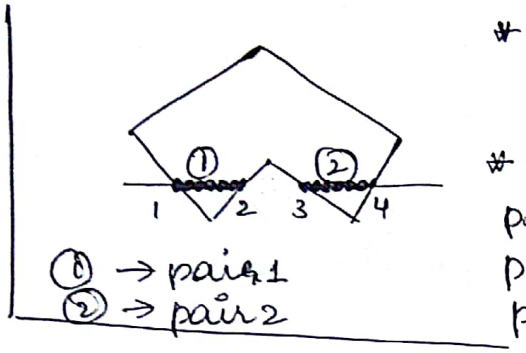
Steps

1) When a polygon is defined, find a minimum enclosed rectangle(binds a polygon) - find the $x_{min}$ & $x_{max}$, $y_{min}$ & $y_{max}$ where all the vertices falls with minimum area that covers a polygon

2) find the no of scanlines = $y_{max} - y_{min} + 1$

3) for each of scanlines do
   * obtain the intersection points of scanline with polygon edges
   * sort the intersection points from left to right, identify interior regions
   * as the odd-even rule
   * form pair of intersections from the list

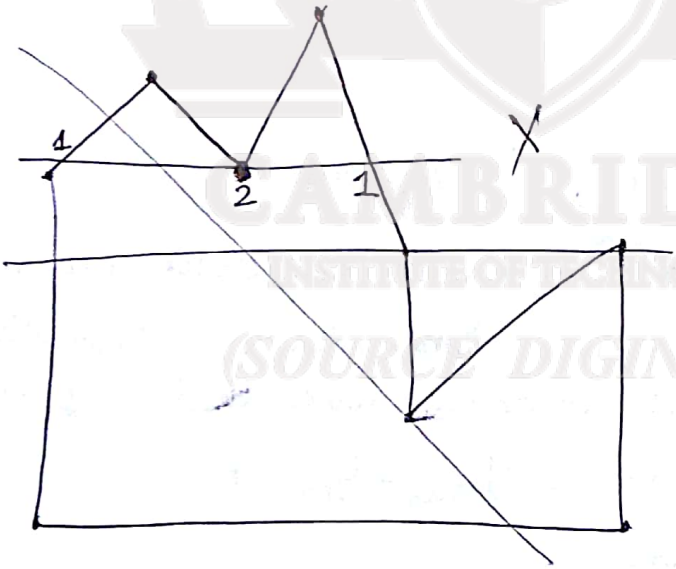
* fill colors are applied to each section (pair) of a scanline that lies within the interior of fill region



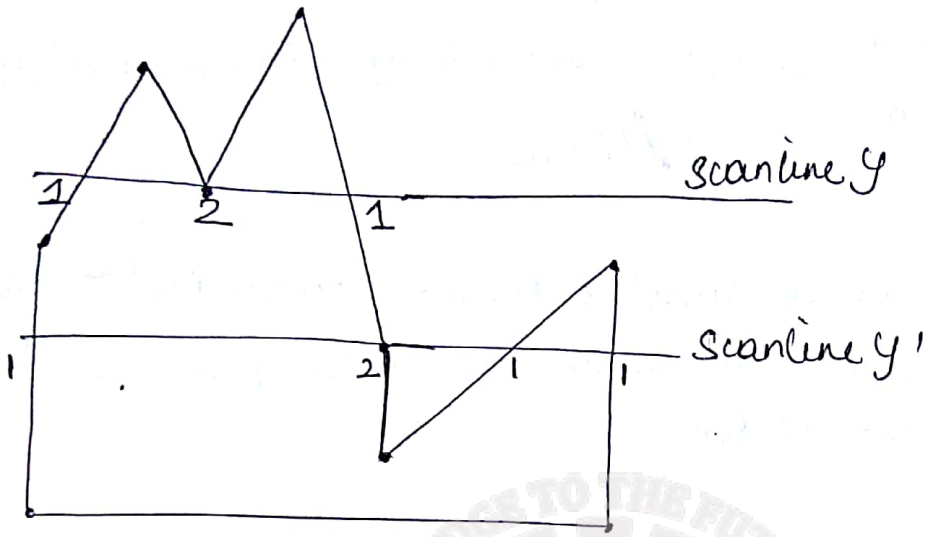

* No of intersections are Even no's forming pairs
* Look left or right (pixel inside the pair), count the no of intersection points of the scanline with edge of polygon, odd → interior else exterior
* Intersection points are updated for each scanline
* Stop when scanline has reached $y_{max}$

## Special issues

Whenever a scan line passes through a vertex, it intersects two polygon edges at that point.
This can results in an odd number of boundary intersections for a scanline.




* Introduction - when we move from one scanline to another i) $\frac{dy}{dx}$, intersection points change by $\frac{\Delta y}{dx}$.
  ii) New edge may start / end.
* If we know the intersection points, then all the next points are calculated by the slope of the line




for y, the edges at the vertex are on the same side of the scanline ∨ bottom ∧ top

Whereas for y', the edges are on either / both sides of the vertex (opposite sides of scanline)

- one top
- one bottom

In such case additional preprocessing is required

## Vertex counting in a scanline

* Traverse along the polygon boundary clockwise or counterclockwise
* 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 for condition

If end-point y values of two consecutive edges monotonically increase or decrease, → decreasing clockwise / increasing anticlock

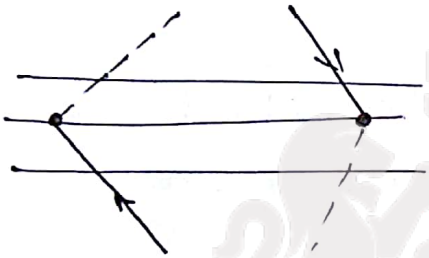
count the middle vertex as a single intersection point for the scanline passing thought it.

Else the shared vertex represents a local maximum (or minimum) on the polygon boundary, Increament the intersection count

decrease ↘ ↗ increase

if the vertex is a local extrema, consider (or add) 2 intersections for the scan line corresponding to such a shared vertex.

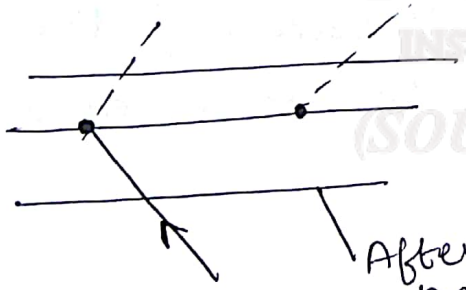
Must avoid such cases listed below.



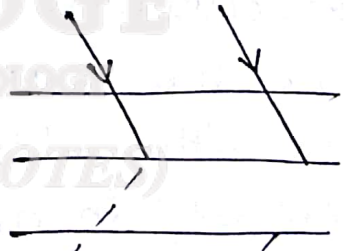
should not add intersection point.

To implement the above.

while processing non-horizontal edges along a polygon boundary in any order, check to determine the condition of monotonically changing (increasing or decreasing) endpoints y values



Before processing After processing



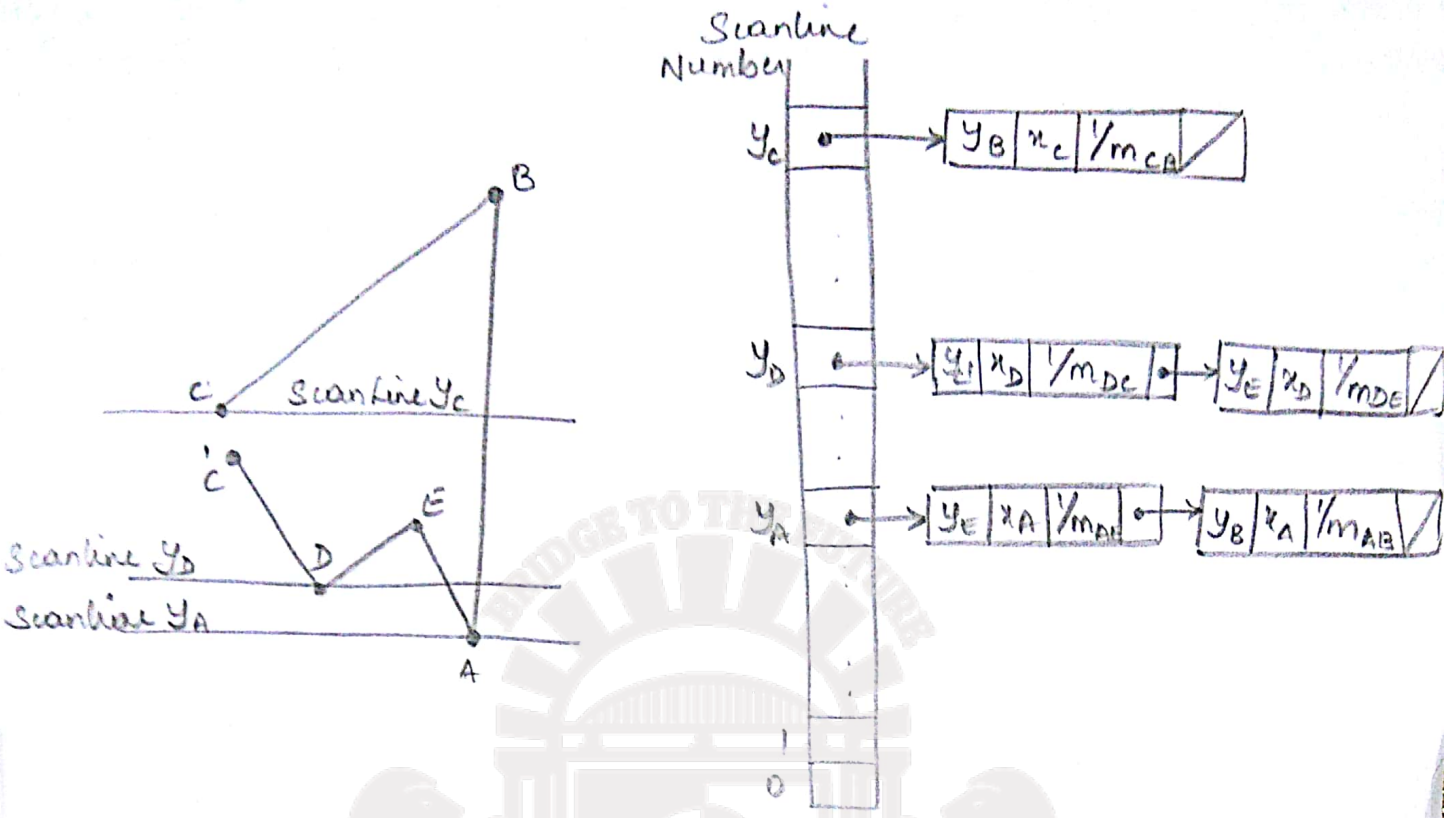
Before processing After processing

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

y-coordinate b/w 2 scan lines $y_{k+1} - y_k = 1$.

x-intersection value $x_{k+1}$ can be determined by

$$x_{k+1} = x_k + 1/m$$




* To efficiently perform a polygon fill, store the polygon boundaries in a sorted edge table containing all information necessary to process the scanlines efficiently
* Use Buckets sort to store edges, sorted on smallest y value of each edge.
* Only Non-horizontal edges are entered into the sorted edge table.
* As edges are processed, we can also shorten certain edges to resolve the vertex-intersection question.
* each table entry contains (for a particular scanline) the maximum y value for that edge, the x-intercept value (at lower vertex) for the edge, & inverse slope of the edge.
* we process the scan lines later from the bottom of the polygon to its top producing an {active edge list} for each scan line crossing the polygon boundaries.

will contain all edges crossed by scanline, used to obtain edge intersections.

# OpenGL Polygon fill-Area functions

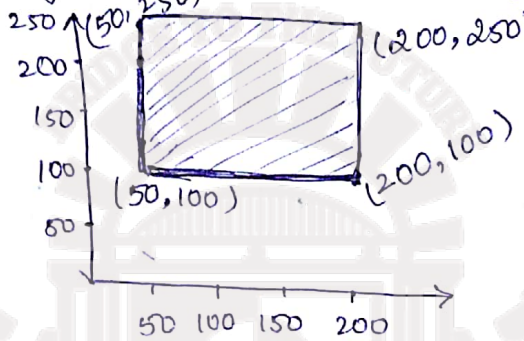
→ glRect * (x1, y1, x2, y2)

One corner of rectangle is at co-ordinate position (x1, y1) and the opposite corner of the rectangle at position (x2, y2).

→ Suffix codes for glRect specifies the co-ordinate data type and whether co-ordinates are to be expressed as array elements

i - integer
s - short
f - float
d - double
v - vector

glRecti(200, 100, 50, 250);




if we put co-ordinate values for this rectangle into Arrays we generate a same square with following code

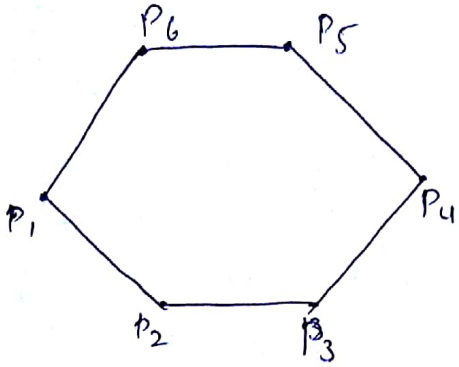
```
int vertex1[ ] = (200, 100)   (x1, y1)
int vertex2[ ] = (50, 250)    (x2 y2)
glRectiv(vertex1, vertex2);
```

when glRect is used, the polygon edges are formed between the vertices in order (x1, y1), (x2, y1), (x2, y2) (x1, y2)

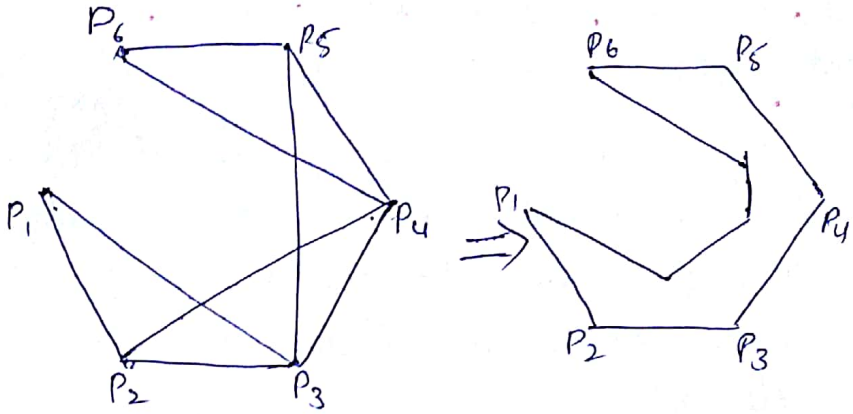
(200, 100), (50, 100), (50, 250), (200, 250)

```
* glBegin (GL_POLYGON):
    glVertex2iv (P1);
    glVertex2iv (P2);
    glVertex2iv (P3);
    glVertex2iv (P4);
    glVertex2iv (P5);
  glEnd ();
```

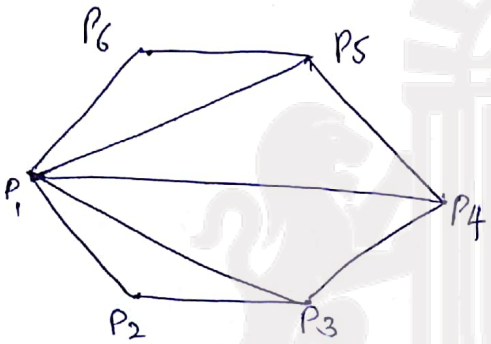
* polygon vertex list must contain at least three vertices, otherwise nothing will be displayed.



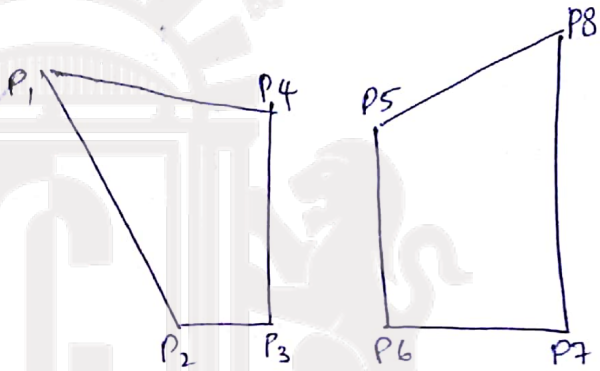

GL_POLYGON



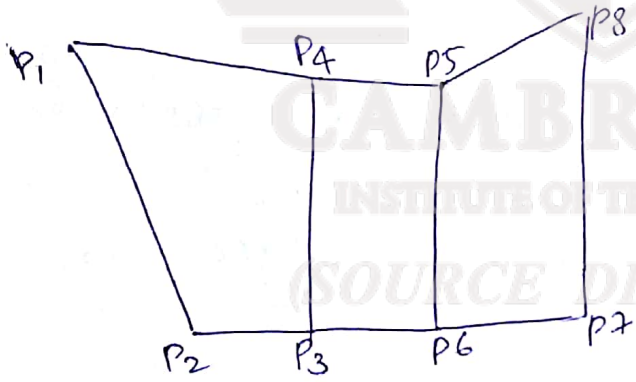

GL_TRIANGLE_STRIP




GL_TRIANGLE_FAN




GL_QUADS



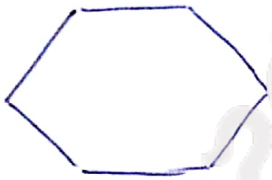

GL_QUAD_STRIP

## Fill area Attributes

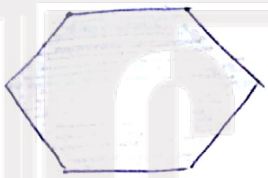
There are two basic procedures for filling an area on raster systems.

1. determine the overlap intervals for scan lines that cross the area, then pixel positions along these overlap intervals are set to fill color.
2. start from interior position & paint outward, pixel by pixel, from this point until we encounter specified boundary conditions

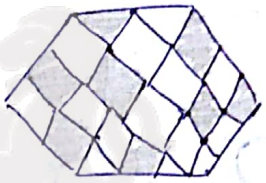
## Fill Styles



a) Hollow



b) Solid

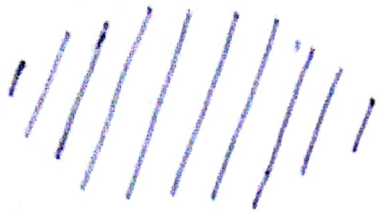


c) Pattern

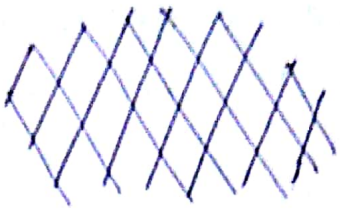
We can also fill selected regions of scene using brush styles, color-blending combinations or textures.

we can also list different colors for different positions in the array / OR fill pattern could specify the bit array that indicates which relative positions are to be displayed in a single selected color.

* An array specifying a fill pattern is a mask that is to be applied to display area.
* the process of filling an area with a rectangular pattern is called ~~tiling~~ "tiling" & the rectangular fill pattern is sometimes refered as "tiling pattern"



Diagonal Hatch fill



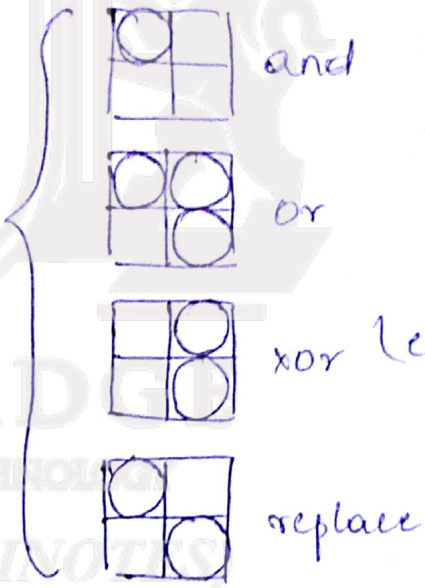
Diagonal crosshatch fill

Spacing & slope for the hatchlines could be set as parameters in hatch table or specified as pattern array that produces sets of diagonal lines.

Color-Blended fill regions

* combine a fill pattern with background colors (Pattern) using a transparency factor that determines how much of background should be mixed with object color.

Pattern Background




Pixel values

Application.

Fill methods using blended colors have been referred to as soft-fill or tint fill algorithms.

① Usage is to soften the fill colors at object borders that have been blurred To antialias the edge

② is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of brush color & background color "behind" the area.

Linear soft-fill algorithm repaints an area merging a foreground color F with a single background color B, where F≠B.

Assume we know the values of F and B, we can check the contents of frame buffer to determine how these colors were combined.

current RGB color P of each pixel

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

t → transparency factor between 0 & 1 for each pixel

$t < 0.5$, background color contributes more to interior color of region than the fill color does.

$$P = (P_R, P_G, P_B) \quad F = (F_R, F_G, F_B) \quad B = (B_R, B_G, B_B)$$

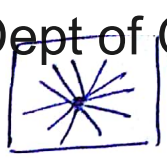
$$t = \frac{P_K - B_K}{F_K - B_K}$$

- color blending procedures can be applied to an area, foreground color merged with multiple background color areas,

When two background color $B_1$ & $B_2$ are mixed with foreground color F, resulting pixel color P

$$P = t_0 F + t_1 B_1 + (1 - t_0 - t_1) B_2$$

color co-efficient $t_0, t_1$ & $(1 - t_0 - t_1)$ must be equal to 1.



to extent

## Plane Equations.

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

$$Ax + By + Cz + D = 0 \quad \rightarrow (1)$$

where $x, y, z$ is any point on the plane.

$A, B, C, D$ are plane parameters — describing spatial properties of plane.

To obtain A, B, C & D values., three plane equations are solved by using co-ordinate values with 3 non collinear points $(x_1, y_1, z_1)$ $(x_2 y_2 z_2)$ $(x_3 y_3 z_3)$

$\div$ by D $\rightarrow (1)$

$$\begin{vmatrix} + & - & + \\ - & + & - \\ + & - & + \end{vmatrix}$$

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

Solution to eqn is obtained using Cramer's rule.

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

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

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

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

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

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

Expanding determinants we get the below

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

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

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

$$D = -x_1(y_2 z_3 - y_3 z_2) - x_2(y_3 z_1 - y_1 z_3) - x_3(y_1 z_2 - y_2 z_1)$$

## Front and Back Polygon faces

* The side of a polygon that faces into the object interior is called backface.
* the visible, or outward side is front face.
* Identifying the position of points in space relative to front & back faces of a polygon – Basic task.

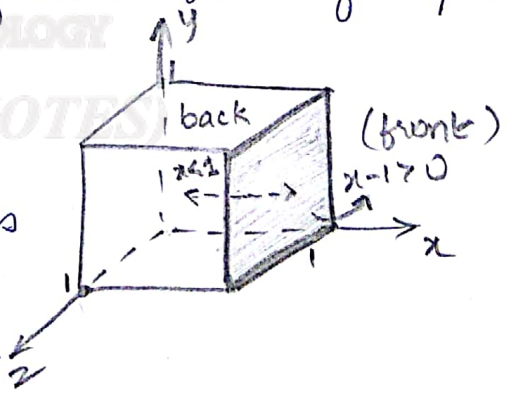
A polygon with infinite plane.

* Any point that is not on the plane & that is visible to the front face → in front of (outside the object)
* Any point that is visible to the back face of polygon is behind (inside) the plane.
* Inside / outside is relative to the plane containing the polygon.

Plane equation → $Ax+By+Cz+D \neq 0$. (Not on plane)

if $Ax+By+Cz+D < 0$ the point $(x,y,z)$ is behind the plane.

If $Ax+By+Cz+D > 0$ the point $(x,y,z)$ is in front of the plane

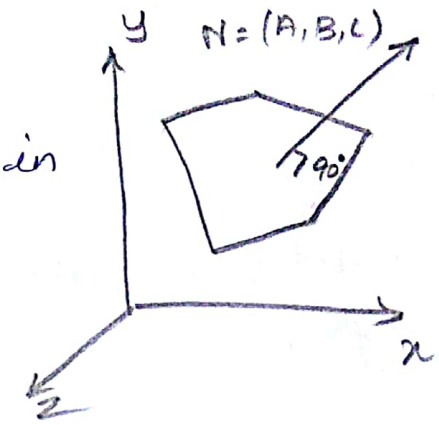



Any point outside (front of) the plane of shaded polygon satisfies the inequality $x-1>0$ while any point inside the plane (in back of) has x co-ordinate value less than 1

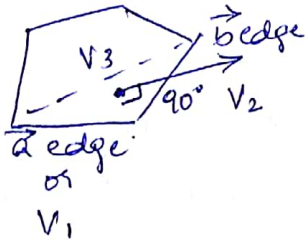
* Orientation of a polygon surface in space can be described with normal vector (perpendicular to plane) & has cartesian components A, B, C (plane co-efficients)

* Normal vector points in direction from inside the plane to outside i.e from back face of polygon to front face.

* The normal vector $N = (1, 0, 0)$ – is in the direction of positive x-axis's




Suppose.




$$N = (V_2 - V_1) \times (V_3 - V_1)$$

* This generates values for plane parameters A, B & C
* Elements of normal vector can be obtained using vector cross product calculation.
* Plane equation in vector form is $N.P = -D$
  N → normal vector, P is any point in the plane.

## OpenGL Fill Area Attribute functions

Displays of filled convex polygon in 4 steps.

1. Define a fill pattern.

```
GLubyte fillPattern[] = { 0xFF, 0x00, 0xFF, 0x00...};
```

* to fill the patter in OpenGL, we use a 32bit x 32 bit mask.
* Value 1 in mask indicates the corresponding pixels is to be set to the current color.
* Value 0 → leaves the value of that frame buffer position unchanged.

2. Invoke the polygon fill routine

```
glPolygonStipple(fillPattern);
```

we need to enable the fill routines before we specify the vertices for the polygons that are to be filled with the current pattern. hence (3).

3. we activate the polygon-fill feature of OpenGL

```
glEnable(GL_POLYGON_STIPPLE);
```

we turn off pattern filling with

```
glDisable(GL_POLYGON_STIPPLE);
```

4. Describe the polygons to be filled

## OpenGL Texture & Interpolation Patterns

* Use texture patterns to fill polygons.
* Similar to Simulate the surface appearances of wood, brick, brushed steel...
* Interpolation fill of a polygon interior is used to produce realistic displays of shaded surface under various lighting conditions.

```
glShapeModel(GL_SMOOTH);
glBegin(GL_TRIANGES);
   glColor3f(0.0, 0.0, 1.0);
   glVertex2i(50, 50);
   glColor3f(0.0, 1.0, 0.0);
   glVertex2i(150, 50);
   glColor3f(1.0, 0.0, 0.0);
   glVertex2i(75, 150);
glEnd();
```

GL_FLAT → fills the polygon with one color.
GL_SMOOTH → default shading.

OpenGL Wire-frame Methods.
- to show only polygon edges. (produces wire frame or hollow display of polygon).

glPolygonMode(face, displayMode);

* parameter 'face' which face of polygon we want to show edges. GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.
* DisplayMode → GL_LINE.
  GL_POINTS (polygon vertex points)

- Stitching → methods for displaying the edges of a filled polygon may produce gaps along the edges. due to scanline fill or edge line-drawing algo calculation.
- to eliminate the gap - shift the depth values calcu-

-lated by the fill routine so that they do not overlap with edge depth values for that polygon

```
glColor3f (0.0, 1.0, 0.0);
glEnable( GL_POLYGON_OFFSET_FILL);   - set routine for scanline filling
glPolygonOffset (1.0, 1.0);
glDisable ( GL_POLYGON_OFFSET_FILL);
```

glPolygonOffset (factor1, factor2);

calculate amount of depth offset.

depthoffset = factor1 . maxSlope + factor2 . const

→ GL_POLYGON_OFFSET_LINE
→ GL_POLYGON_OFFSET_POINT

* To eliminate selected edges from wire-frame display — glEdgeFlag (flag);

This indicates that a vertex does not proceed a boundary edge, GL_FALSE to parameter flag.

OpenGL Front face function.

Although the ordering of polygon vertices controls the identification of front & back faces.

We can label the selected faces in the scene independently as front or back with the function.

glFrontFace (vertexorder);

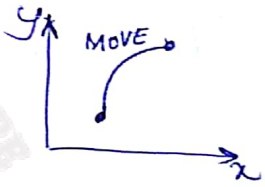
The vertexOrder in OpenGL when set to GL_CW (clockwise ordering) for its vertices will be considered to the front face.

If the vertex order in OpenGL, GL-CCW (counter clockwise Ordering) of polygon vertices as front-facing which is the default Ordering.

## Basic Two-Dimensional Geometric Transformations

Representation of points:

$2 \times 1$ matrix (R c) $\begin{bmatrix} x \\ y \end{bmatrix}$




General Method of applying transformation.

geometric transform matrix / Affine transformation matrix

$$[B] = [T][A]$$

Transformed co-ordinates x prime, y prime $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$ $\quad \begin{bmatrix} x' \\ y' \end{bmatrix}^T = \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$x' = ax + cy$ $\quad$ $x' = ax + cy$

$y' = bx + dy$ $\quad$ $y' = bx + dy$

Premultiplication $\quad$ Post Multiplication.

## Special cases of 2D Transformations

1) $T$ = identity matrix.

$a = d = 1, \; b = c = 0 \Rightarrow x' = x, \; y' = y.$

2) Scaling & Reflections:

$b = 0, \; c = 0 \Rightarrow x' = a \cdot x, \; y' = d \cdot y;$

This is scaling by $a$ in $x$, $d$ in $y$.

if $a = d > 1$, we have enlargement (zooms)

if $0 < a = d < 1$, we have compression (reduction)

Scaling is uniform. if $a$ & $d$ value are same

" non-uniform; if $a$ & $d$ do not have same identity value

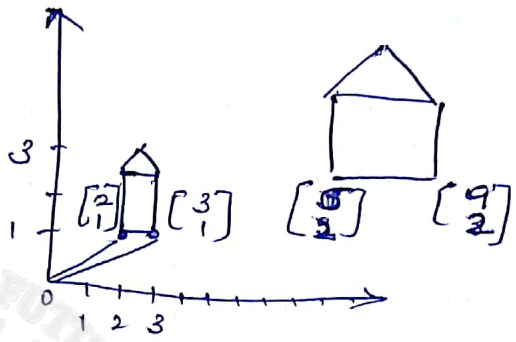
i.e if $a = d \Rightarrow$ Uniform scaling. else non-uniform scaling.

Scale Matrix : let $s_x = a$, $s_y = d$ $\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$

Example of scaling Non-Uniform

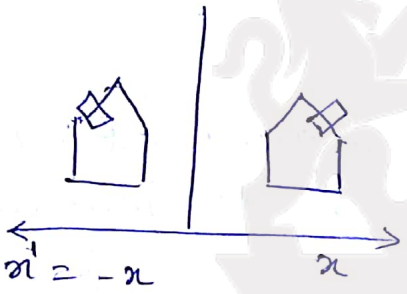
$s_x = 3$

$s_y = 2$




Only the diagonal terms are involved in scaling and reflections

Some more examples. (Reflection)




we will be considering the diagonal value. $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ $b \& c = 0$,

In this case $x$ will be negative $y$ remains same +ve.

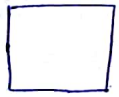
$x' = -1$

$y = 1$

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

off diagonal elements are involved in Shearing

Force is applied →



shears along the axis

Very huge dictionary

$a = d = 1$;

let $c = 0$, $b = 2$

$x' = x$

$y' = 2x + y$

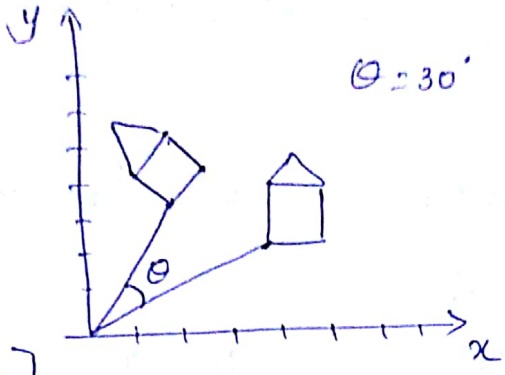
$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$

$x' = ax + cy$

$y' = bx + dy$.

$y'$ depends linearly on $x$ : This effect is called shear.

## Rotation




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

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

matrix form is

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Clockwise → -ve rotation

Anticlockwise → +ve rotation.

For rotation determinant $|T| = 1$

$\cos^2\theta + \sin^2\theta = 1$

& $|T|^T = |T|^{-1}$ (Transpose of matrix is inverse.)

* Rotation matrices are orthogonal.

## Two-Dimensional Translation

- A translation (Perform) on a single co-ordinate point by adding offsets to co-ordinates so as to generate a new co-ordinate position.
- In effect, we are moving the point along a straight-line path to its new location.
- If an entire object is moved (multiple co-ordinates) by rotating all the co-ordinate positions by same displacement along parallel paths. Then the complete object is displayed at new location.
- translation distance $t_x$ and $t_y$ to original co-ordinates $(x, y)$ to obtain new co-ordinate positions $(x', y')$

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

$(t_x, t_y)$ is translation vector or shift vector.

matrix format:

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

$$P' = P + T$$

Translation is a rigid-body transformation that moves objects without deformation (every point on object is translated by same amount).

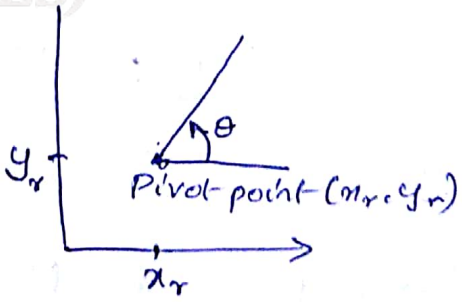
## Two Dimentional Rotation.

* Rotation transformation of an object is done by specifying a rotation axis and a rotation angle
* All points of objects are then transformed to new positions by rotating points through specified angle about the rotation axis.
* Parameters of 2D rotation are the rotation angle $\theta$, position $(x_r, y_r)$ called rotation point or pivot point (intersection point/position of rotation axis ~~at~~ with xy plane).
* +ve value for angle $\theta$ → counterclockwise rotation
* -ve value - clockwise.

Formula in Trignometry

$$\cos\theta = \frac{\text{adjacent side}}{\text{hypotenuse}}$$

$$\sin\theta = \frac{\text{opposite side}}{\text{hypo}}$$



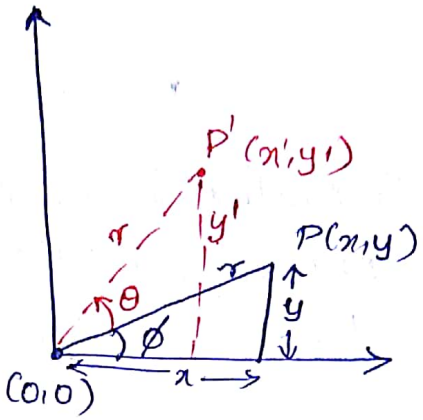

$$\cos(A+B) = \cos A \cos B - \sin A \sin B$$
$$\sin(A+B) = \sin A \cos B + \cos A \sin B$$

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

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

$\cos\phi = \frac{x}{r} \Rightarrow x = r\cos\phi$

$\sin\phi = \frac{y}{r} \Rightarrow y = r\sin\phi$




Rotation in Anticlockwise

The new angle after rotation from $P+P' = (\phi+\theta)$

so $\cos(\phi+\theta) = \frac{x'}{r}$

$$\boxed{x' = r.\cos(\phi+\theta)} = r[\cos\phi\cos\theta - \sin\phi.\sin\theta]$$

$$= r\cos\phi\cos\theta - r.\sin\phi.\sin\theta.$$

$$\boxed{x' = x\cos\theta - y\sin\theta} \rightarrow (1)$$

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

Also $\sin(\phi+\theta) = \frac{y'}{r}$

$$\boxed{y' = r.\sin(\phi+\theta)} = r.\sin\phi\cos\theta + \cos\phi.\sin\theta$$

$$= r.\sin\phi\cos\theta + r.\cos\phi.\sin\theta.$$

$$\boxed{y' = y\cos\theta + x\sin\theta} \rightarrow (2)$$

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

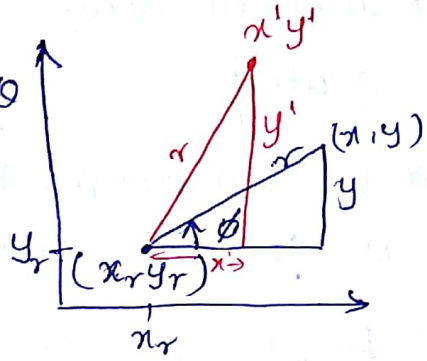
$\therefore P' = R.P$

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

If we solve this matrix we will get eqn (1) & (2)

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

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




## Two-Dimensional Scaling

* To alter size of the object, we apply a scaling transformation
* Scaling operation is performed by multiplying object positions $(x, y)$ by scaling factors $S_x$ and $S_y$ to produce

Let us consider
$P = (x, y)$ → before scaling &
$P'(x', y')$ → after scaling

$$x' = x \cdot S_x$$
$$y' = y \cdot S_y$$

$$\therefore \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{matrix} x' = x \cdot S_x \\ y' = y \cdot S_y \end{matrix}$$

$$P' = S \cdot P.$$

* if $S_x$ & $S_y$ is in between 0 & 1, then point is closer to origin which means the size of an object will decrease
* If $S_x$ & $S_y$ are greater than 1, then the point is away from the origin, which means the size of an object increases.
* If $S_x$ & $S_y$ are equal, assigned to same value,

Uniform scaling is performed.

* Unequal values for $S_x$ & $S_y$ result in differential scaling

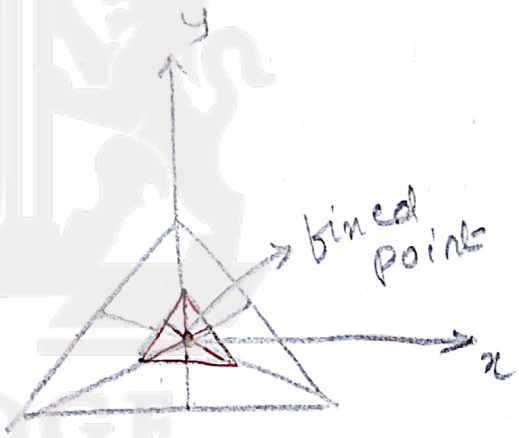
* we can control the location of a scaled object by choosing a position, called fixed position/point

fixed point $(x_f, y_f)$ – centroid

objects are now ~~rotated~~ resized by scaling the distances b/w object points & fixed points.

$$x' = x_f + (x - x_f) S_x, \quad y' = y_f + (y - y_f) S_y$$

$$x' = x \cdot S_x + x_f (1 - S_x)$$
$$y' = y \cdot S_y + y_f (1 - S_y)$$



# Matrix Representations and Homogeneous Co-ordinates

Three Basic two-Dimensional transformations

1. Translation
2. Rotation
3. Scaling

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

P' & P represents column vectors.

Matrix $M_1$ → 2 × 2 array containing multiplicative factors.

$M_2$ → 2 element array column matrix containing translational terms. $\begin{bmatrix} x_t \\ y_t \end{bmatrix}$

For translation, $M_1$ is identity matrix.

$P' = P + T$ where $T = M_2$

For rotation and scaling, $M_2$ is 0 contains translational terms associated with pivot point or scaling fixed point.

$P' = P + T$, $P' = R \cdot T$ multiplicative & Translational term are to be combined into a single matrix form.

## Homogeneous Co-ordinates

A standard technique to expand the matrix representation for a 2D co-ordinate $(x, y)$ position to a three-element (column matrix) representation $(x_h, y_h, h)$ → called Homogeneous co-ordinates,

$h$ → homogeneous parameter h (non zero value)

i.e $(x, y)$ is converted into new co-ordinate values as $(x_h, y_h, h)$

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

$$x_h = x \cdot h \quad y_h = y \cdot h$$

$$\Rightarrow (x \cdot h, y \cdot h, h)$$

Suppose $x=2$, $y=3$ & if $h=1$

$(x_h, y_h, h) = (2, 3, 1)$

If $h=2$ $(x_h, y_h, h) = (x.h, y.h, h)$

$= (2\times 2, 3\times 2, 2)$

$= (\underline{4}, \underline{6}, \underline{2})$

Again if you want to convert from homogenous co-ordinates to 2D then

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

for the case $h=2$.

$x = \frac{4}{2} = 2$ $y = \frac{6}{2} = 3$. h is considered as '1'. i.e $h=\underline{1}$.

$(x, y) = (2, 3)$

If $h=1$, the old & the new co-ordinate values will not change

If $h=0$, then the co-ordinate system will be set to infinity.

## Two-Dimensional Translation Matrix.

Using homogenous approach,

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

This translation operation can be written as

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

$T(t_x, t_y)$ is $^{3\times 3}$ translation matrix.

## Two Dimensional Rotation Matrix:

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

(OR)

$$P' = R(\theta) \cdot P$$

## Two Dimensional Scaling Matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

(OR)

$$P' = S(S_x, S_y) \cdot P$$

## Inverse Transformations

* For translation, we obtain the inverse matrix by negating the translation distances.

Inverse translation Matrix is

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

* An Inverse Rotation is accomplished by replacing the rotation angle by its negative.

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ equivalent to its transpose.

$$R^{-1} = R^T$$

* Inverse matrix for scaling transformation - by replacing the scaling parameters with their reciprocals

$$S^{-1} = \begin{bmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

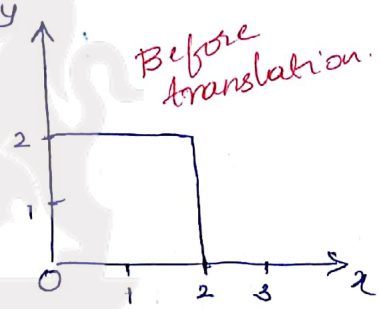
- The inverse matrix generates an opposite scaling transformation, so any scaling matrix multiply X its Inverse produces Identity matrix.

Translation Example.

1. Consider the square with co-ordinates (0,0) (2,0) (0,2) (2,2)

Translate square '2' units w.r.t. x-axis, '3' units with respect to 'y' axis.

$t_x = 2$, $t_y = 3$




translate each point w.r.t $t_x$ & $t_y$.

(1) point (0,0)

$x = 0, y = 0$

$x' = x + t_x = 0 + 2 = 2$

$y' = y + t_y = 0 + 3 = 3$

(0,0) is translated to new point (2,3).

(2) point (0,2)

$x = 0, y = 2$

$x = 0 + 2 = 2$

$y = 2 + 3 = 5$

$(0,2) \rightarrow (2,5)$

(3) point (2,2)

$x = 2, y = 2$

$x = 2 + 2 = 4$

$y = 2 + 3 = 5$

$(2,2) \rightarrow (4,5)$

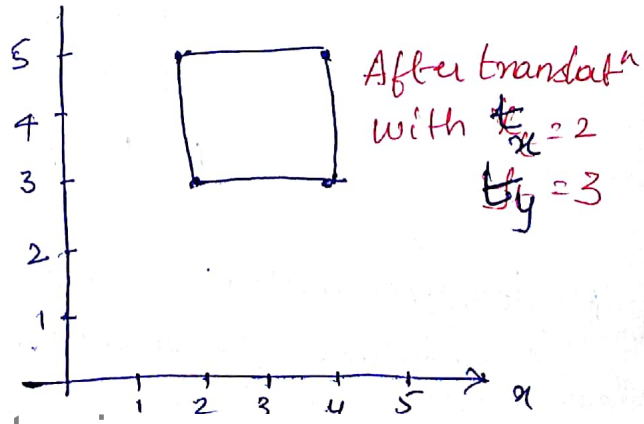
(4) point (2,0)

$x = 2, y = 0$

$x = 2 + 2 = 4$

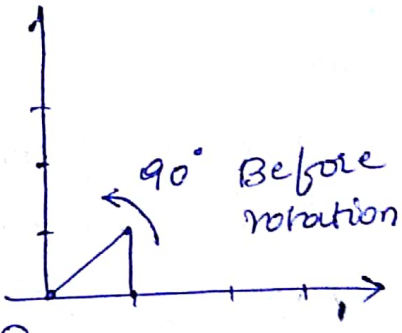
$y = 0 + 3 = 3$

$(2,0) \rightarrow (4,3)$



## Rotation Example

1. a) Consider a triangle (0,0) (1,0) & (1,1), Rotate 90° with Anticlockwise.




Soln :- formula for Anticlockwise.

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

We know that $\sin 90° = 1$ $\cos 90 = 0$

① Consider 1st co-ordinate

$(0,0) \Rightarrow x = 0, y = 0$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Rightarrow (0,0) \rightarrow (0,0)$$ does not change even after rotation

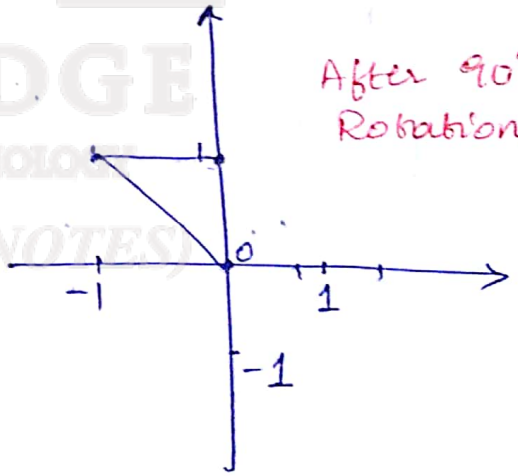
② 2nd co-ordinate

$(1,0) \Rightarrow x = 1, y = 0$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$(1,0) \rightarrow (0,1)$




③ 3rd co-ordinate

$(1,1) \Rightarrow x = 1, y = 1$

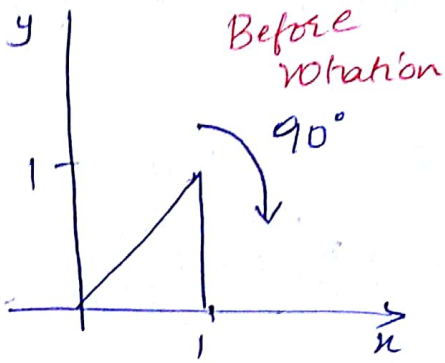
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$(1,1) \rightarrow (-1,1)$

b) Rotating 90° in clockwise direction

triangle (0,0) (1,0) (1,1)




Sol^n

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

1) (0,0) - 1st co-ordinate $x=0, y=0$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$(0,0) \rightarrow (0,0)$

② 2nd co-ordinate (1,0)

$x=1, y=0$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

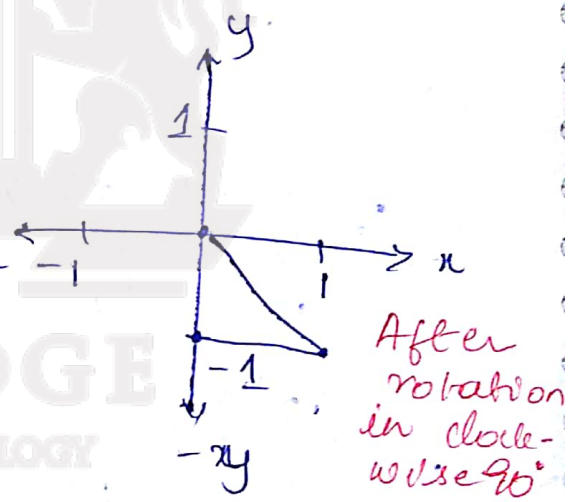
$(1,0) \rightarrow (0,-1)$

③ 3rd co-ordinate (1,1)

$x=1, y=1$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$(1,1) = (1,-1)$



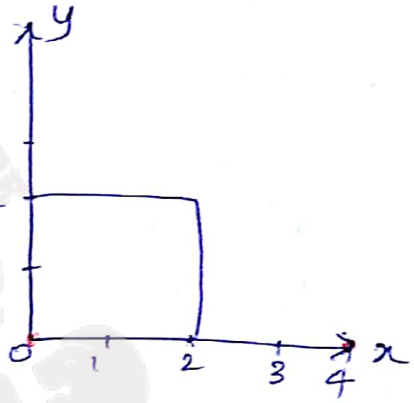
# Example of Scaling

1. a) Consider a square with co-ordinates
(0,0) (2,0) (0,2) (2,2)
Scale w.r.t. $S_x = 2$, $S_y = 3$

Soln: Since $S_x$ & $S_y$ 70 → the size of the square increases
also since $S_x \neq S_y$ → hence the shape of the square also changes.

formula :- $x' = x * S_x$
$y' = y * S_y$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$




a. ① 1st coordinate (0,0)
$x = 0, y = 0$

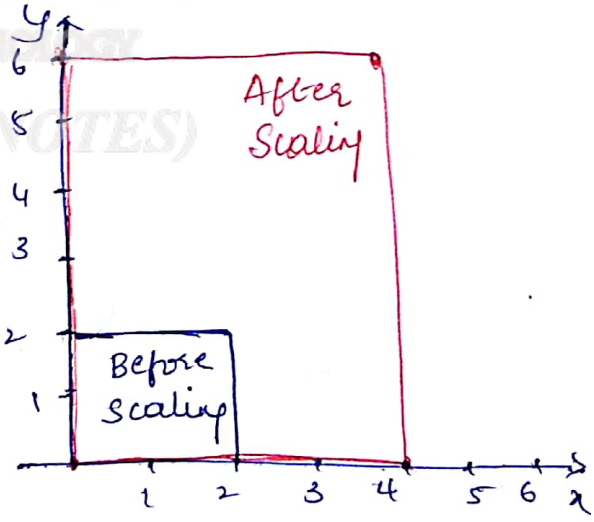
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$x' = x \cdot S_x$
$= 0 \cdot 2 = 0$
$y' = 0 \cdot 3 = 0$
(0,0) → (0,0)

② 2nd co-ordinate (2,0)
$x = 2, y = 0$
$x' = 2 \times 2 = 4$
$y' = 3 \times 0 = 0$
(2,0) → (4,0)

③ 3rd co-ordinate (0,2)
$x = 0, y = 2$
$x' = 0 \times 2 = 0$
$y' = 3 \times 2 = 6$
(0,2) → (0,6)

④ 4th co-ordinate (2,2)
$x = 2, y = 2$
$x' = 2 \times 2 = 4$
$y' = 2 \times 3 = 6$
(2,2) → (4,6)



b) Scaling parameter $S_x = 0.5$, $S_y = 0.5$ for the same square (0,0) (2,0) (0,2) (2,2)

Soln. ① 1st co-ordinate

$x = 0, y = 0$

$x' = 0.5 \times 0 = 0$

$y' = 0 \times 0.5 = 0$

$(0,0) \rightarrow (0,0)$

② 2nd co-ordinate

$x = 2, y = 0$

$x' = 2 \times 0.5 = 1$

$y' = 0 \times 0.5 = 0$

$(2,0) \rightarrow (1,0)$

③ 3rd co-ordinate

$(0,2) \rightarrow x = 0, y = 2$

$x' = 0 \times 0.5 = 0$

$y = 2 \times 0.5 = 1$

$(0,2) \rightarrow (0,1)$

④ 4th co-ordinate

$(2,2) \rightarrow x = 2, y = 2$

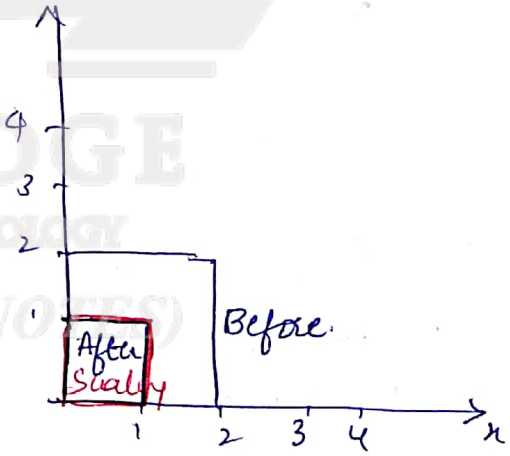
$x' = 2 \times 0.5 = 1$

$y' = 2 \times 0.5 = 1$

$(2,2) \rightarrow (1,1)$

Here $S_x$ & $S_y$ are of same values, hence the size of the object changes uniformly.

Also, $S_x$ & $S_y$ is in b/w 0 & 1, so the size will decrease.



## Two-Dimensional composite Transformations.

* we can set up a sequence of transformations as a composite transformation matrix by calculating the product of individual transformation refered to as concatination or composition

* If we want to apply two transformations to point position P,

$$P' = M_2 \cdot M_1 \cdot P$$
$$= M \cdot P \qquad \Rightarrow M = M_2 \cdot M_1$$

## Composite two-dimensional translations

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

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

$$T(t_{2x}, t_{2y}) \cdot T(t_{1x}, t_{1y}) = \begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix}$$

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

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

which demonstrates 2 successive translations are additive.

## Composite 2D Rotations

* Two successive rotation's applied to a point P produce the transformed position

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

## Composite 2D Scalings

Concatenating transformation matrices for 2 successive scaling operations produces.

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

(OR)

$$S(S_{2x}, S_{2y}) \cdot S(S_{1x}, S_{1y}) = S(S_{1x} \cdot S_{2x}, S_{1y} \cdot S_{2y})$$

## General two-Dimensional Pivot-Point Rotation.

* when a graphics package provides only a rotate function w.r.t co-ordinate origin, we can generate a rotation about any pivot-point $(x_r, y_r)$ by performing the sequence of translate-rotate-translate operations.

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

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

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

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

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

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

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

which can be expressed as

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

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

## General Two-Dimensional Fixed point scaling

This sequence is

1) Translate the object so that the fixed point coincides with the co-ordinate origin.
2) Scale the object with respect to co-ordinate origin.
3) Use the inverse of translation in step (1) to return the object to its original position.

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

(OR)

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

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

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

## General Two-Dimensional Scaling Directions

Parameters $s_x$ & $s_y$ scale objects along the x & y directions. We can scale an object in other directions by rotating the object to align the desired scaling directions.

To accomplish scaling without changing the orientation effect of the object, we first perform the rotation so that the direction for $s_1$ & $s_2$ co-inside with x & y axis, then scaling transformation $S(s_1, s_2)$ is applied.

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

matrix concatenation properties

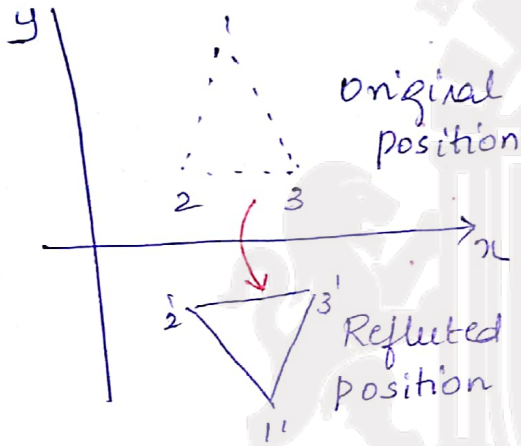
Associative : $M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1)$

# Other 2D transformations.

* Reflection
* Shear

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

* Image is generated relative to an axis of reflection by rotating the object 180° about the reflection axis.

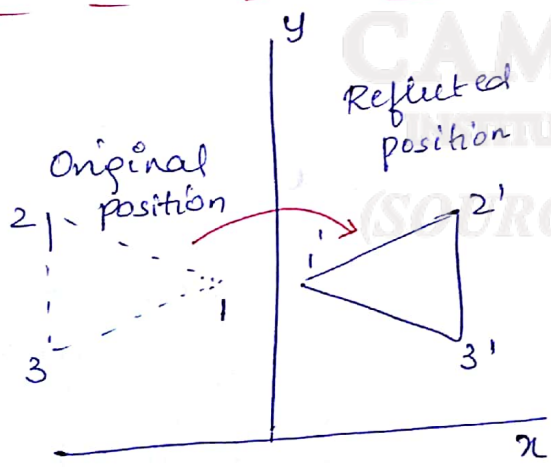



Reflection of an object about the x-axis.

**Reflection about y = 0**

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

transformation retains x values, but "flips" the y values of co-ordinate posith. remains

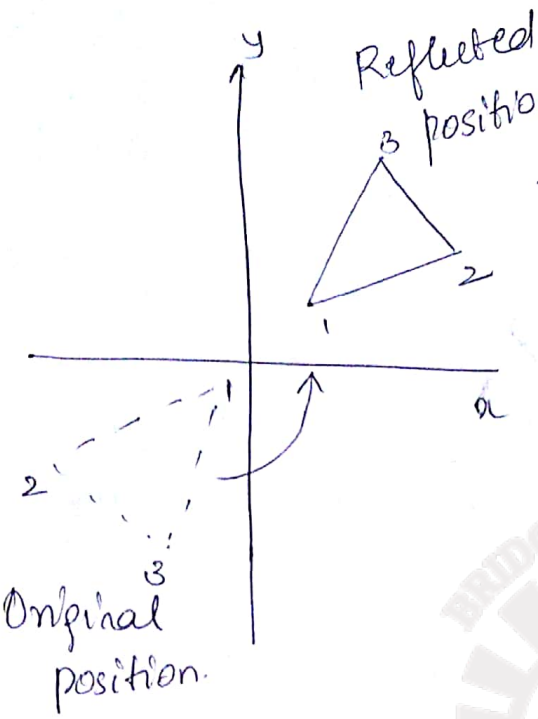



Reflection of an object about the y-axis.

**A reflection about x = 0** flips x co-ordinate while keeping y co-ordinate the same.

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

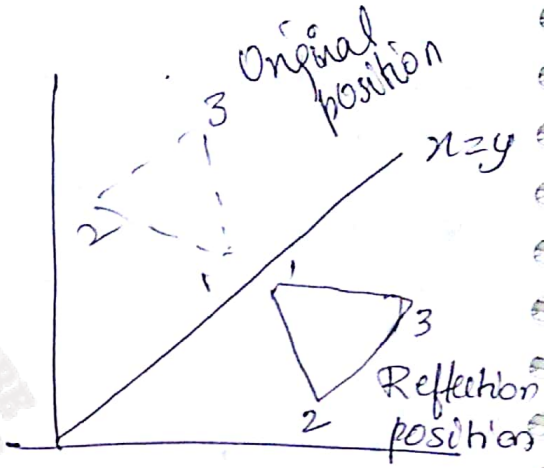
equivalent rotation = 180




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

Reflection about the Origin (rotate about the xy plane)

180° R(θ) = 180°



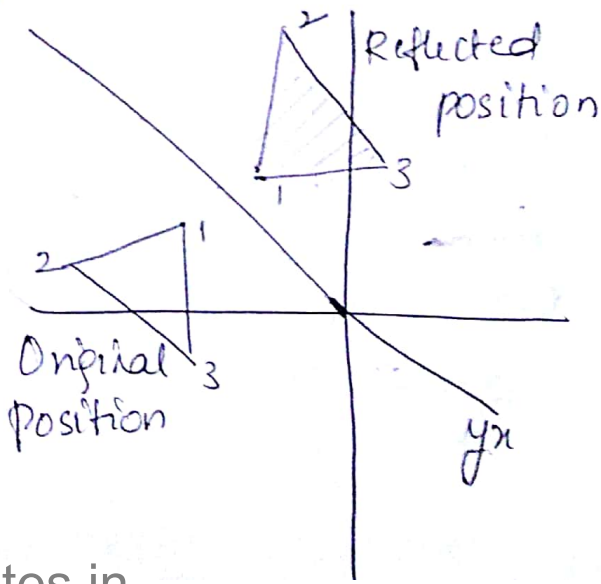

Reflection of an object with respect to line $y = x$

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

To obtain a transformation matrix for reflection about the diagonal $y = -x$, sequence is

1. clockwise rotation by 45°
2. Reflection about y-axis ($y = x$)
3. Counter clockwise rotation by 45°

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



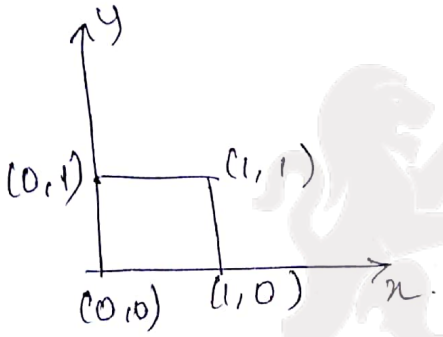
## Shear

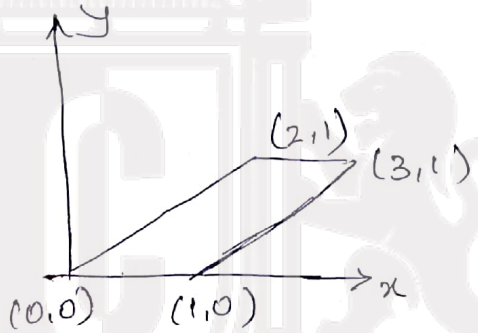
A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called shear.

An x-direction shear relative to x-axis is

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow \boxed{\begin{aligned} x &= x + sh_x \cdot y \\ y &= y \end{aligned}}$$

$sh_x \rightarrow$ Shear parameter




Using x-direction shear with $sh_x = 2$.

① 1st co-ordinate $x=0, y=0$

$x = 0+0 = 0$

$y = 0$ $\quad (0,0) \rightarrow (0,0)$

② 2nd co-ordinate $x=1, y=0$

$x = 1+0 = 1$

$y = 0$ $\quad (1,0) \rightarrow (1,0)$

③ 3rd co-ordinate

$x=1, y=1$

$x = (1*2)+1 = 3$

$y = 1$

$(1,1) \rightarrow (3,1)$

④ $x=0, y=1$

$x = 1*2 = 2$

$y = 1$ $\quad (0,1) \rightarrow (2,1)$

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

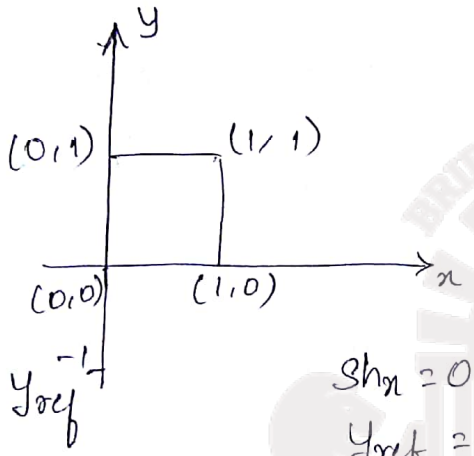
$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boxed{\begin{aligned} x &= x + sh_x(y - y_{ref}) \\ y &= y \end{aligned}}$$

$\{y_{ref} = -1\}$

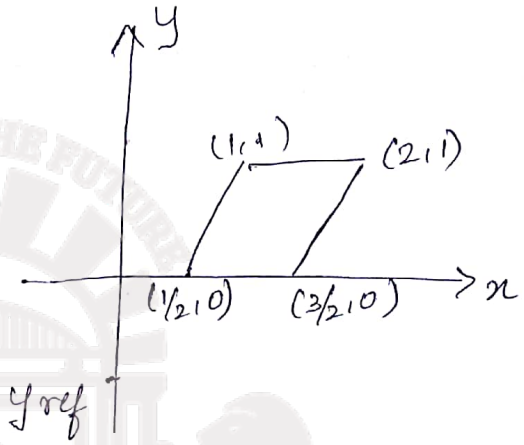
A y-direction shear relative to line $x = x_{ref}$ is generated with the transformation matrix

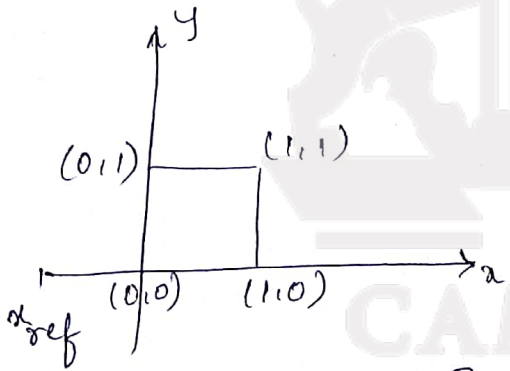
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

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

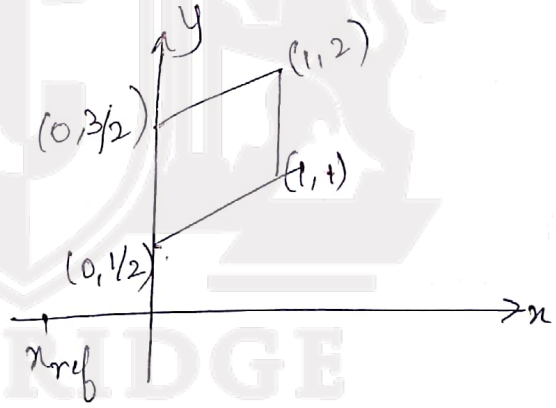



$sh_x = 0.5$
$y_{ref} = -1$
in x-direction




$sh_y = 0.5$
$x_{ref} = -1$ in y direction




## Rigid body transformation matrix

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

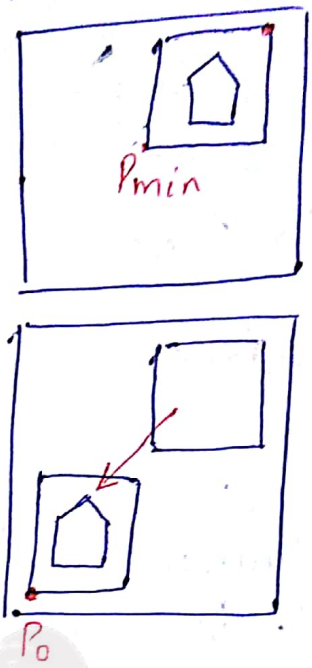
$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1$$

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

# Raster Methods for Geometric Transformations

* Raster systems store picture information as color patterns in the frame buffer.

Functions that manipulate rectangular pixel arrays are called raster operations and moving a block of pixel values from one position to another is termed as block transfer, a bitblt or a pixblt




* Rotate a two-dimensional object or pattern 90° counterclockwise by reversing the pixel values in each row of the array, then interchanging rows & columns.

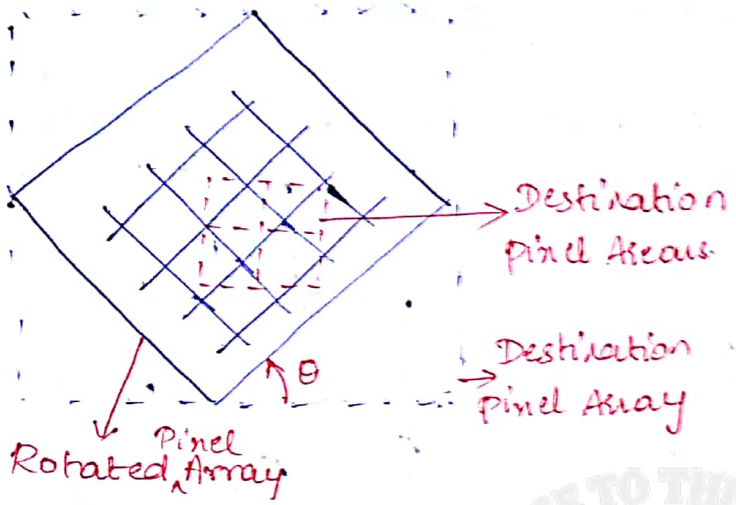
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \xrightarrow[\text{anti clock}]{\text{Reverse 90°}} \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \xrightarrow[90°]{\text{Reverse}} \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

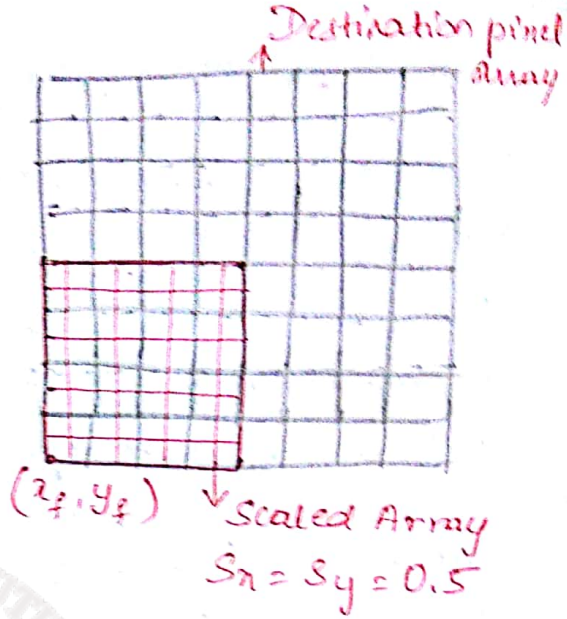
(Overall 180°) 90+90

* 180° rotation is obtained by reversing the order of elements in each row of the array, then reversing the order of rows.

* For rotations not multiples of 90°, additional processing is required.

Each destination pixel area is mapped onto the rotated array & amount of overlap with the rotated pixel area is calculated.




## OpenGL Raster Transformations

* A translation of a rectangular array of pixel-color values from one buffer area to another by

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

  locations & dimensions of pixel block

  Specified that it is the color value that are to be copied.

* Both the regions to be copied (the source) & destination area should lie completely within the bounds of screen co-ordinates

* We can rotate a block 90° by first saving the block in an array, then rearranging the elements of the array & placing it back in refresh buffer.

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

placing back → glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);

## Two Dimensional Scaling transformation

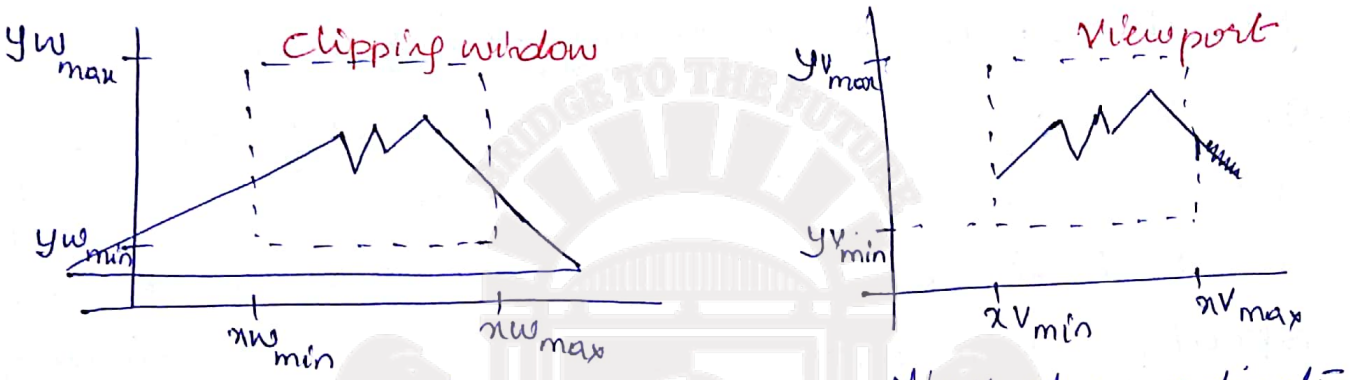
glPixelZoom(Sx, Sy) - scaling factors & then invoking glCopyPixels or glDrawPixels

## 2D Viewing Pipeline.

A section of 2D scene that is selected for display is called clipping window.

Clipping window selects what we want to see.

Viewport indicates where it is to be viewed on the output devices.




* By changing positions of a viewport, objects can be viewed at different positions on display area.
* Multiple viewport windows can be used to display different ~~solid~~ sections of scene at different screen positions.
* In most applications, we tend to specify or define the objects with a convenient size, orientation, and location in a seperate co-ordinate reference frame is called model or object frame.
* Each object must be brought into an application that might contain hundreds or thousands of individual objects. constructing a scene by placing all the objects into appropriate locations within a scene reference frame is called world frame and the values are world co-ordinates.

* This step involves a transformation of individual object (modeling coordinates frames) to specified to position and orientations within the world frame.
* The mapping of 2D, world coordinate scene discription to device co-ordinates is called a 2D viewing transformation. [window to viewport transformation or windowing transformation]
* In 2D; clipping window is often just defined in world co-ordinates (Viewing co-ord. are same as world co-ord)
* In 3D: A seperate viewing frame is required to specify the parameter for viewing position, direction and orientation.
* Transform viewing co-ordinates into Normalized co-ordinates where each co-ordinate value is in the range from 0 to 1, or -1 to 1.
* At the final → Map Normalized co-ordinates to Device co-ordinates, the contents of viewport are transferred to positions within the display window.

Refer figure 6-3 from textbook.

# OpenGL 2D Viewing functions.

Refer text book 6-4