KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module 2

Module – 2 (Filled Area Primitives and transformations)

Filled Area Primitives- Scan line polygon filling, Boundary filling and flood filling. Two dimensional transformations-Translation, Rotation, Scaling, Reflection and Shearing, Composite transformations, Matrix representations and homogeneous coordinates. Basic 3D transformations.

SAINTGITS
LEARN.GROW.EXCEL

Polygon Fill Algorithms

Polygon Representation

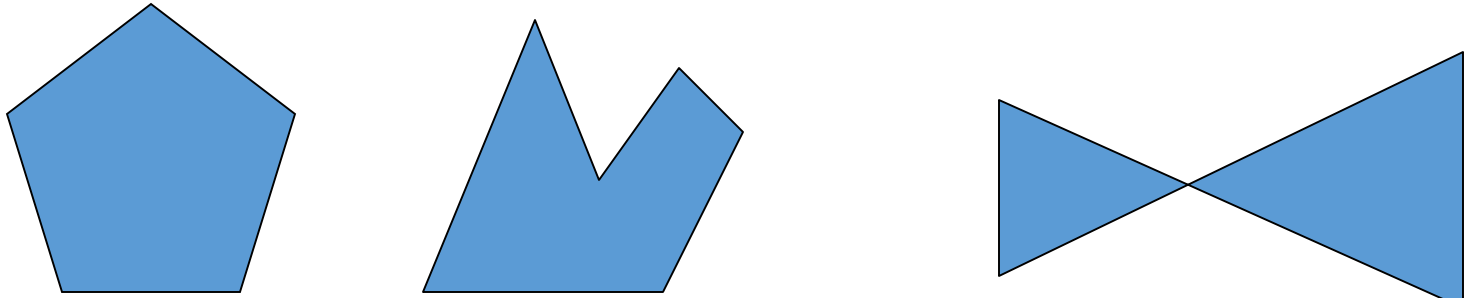
The polygon can be represented by listing its n vertices in an ordered list.

$$P = \{(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_n, y_n)\}.$$

The polygon can be displayed by drawing a line between (x_1, y_1), and (x_2, y_2), then a line between (x_2, y_2), and (x_3, y_3), and so on until the end vertex. In order to close up the polygon, a line between (x_n, y_n), and (x_1, y_1) must be drawn.

Polygon Fill Algorithm

- ***Different types of Polygons***
  - Simple Convex
  - Simple Concave
  - Non-simple : self-intersecting
  - With holes



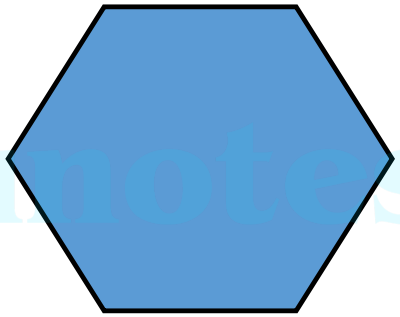
Convex Concave Self-intersecting

Polygon Filling

Types of filling

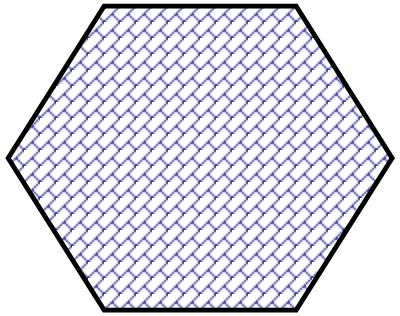
- **Solid-fill**

All the pixels inside the polygon's boundary are illuminated.



- **Pattern-fill**

The polygon is filled with an arbitrary predefined pattern.



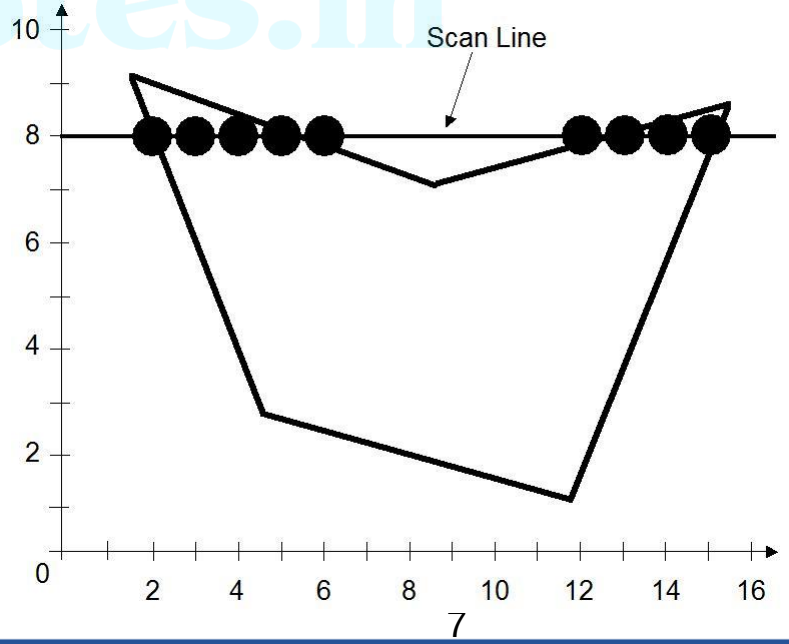
Types of Polygon fill algorithm

- 1.Scan line algorithm
- 2. Boundary Fill algorithm
- 3. Flood fill algorithm

The Scan-Line Polygon Fill Algorithm

The scan-line polygon-filling algorithm involves

- the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,
- identifying which edges intersect the scan-line,
- and finally drawing the interior horizontal lines with the specified fill color. process.



Example

- Consider the following polygon:

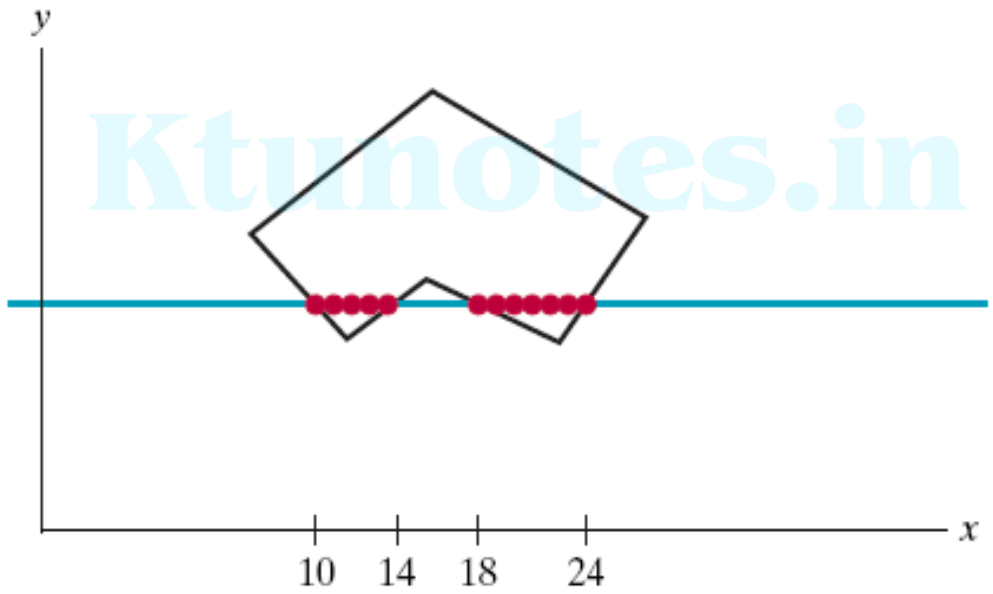


FIGURE 4–20 Interior pixels along a scan line passing through a polygon fill area.

Example

- For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color.

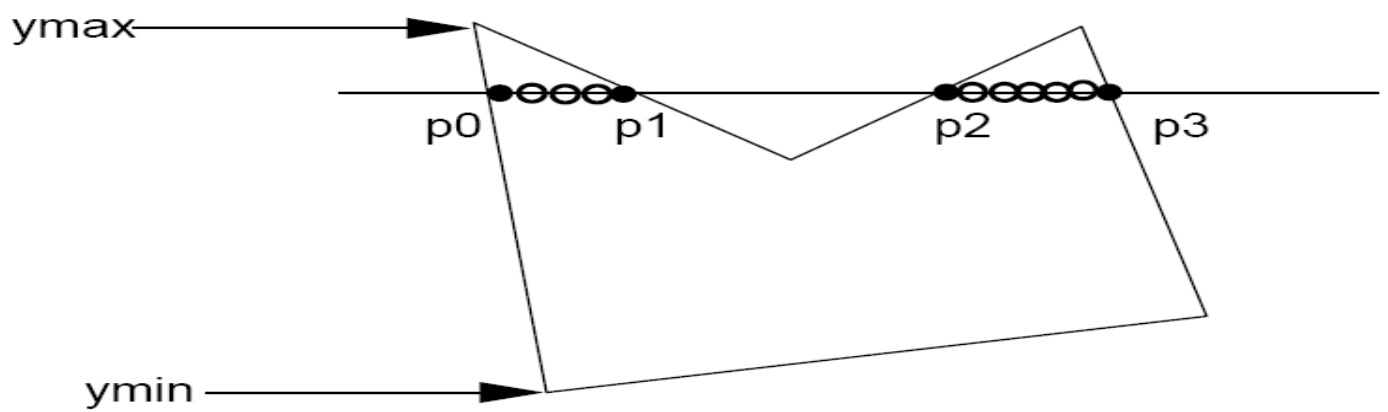
- In the previous Figure, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels.

Scanline Fill Algorithm

- Intersect scanline with polygon edges
- Fill between pairs of intersections
- Basic algorithm:

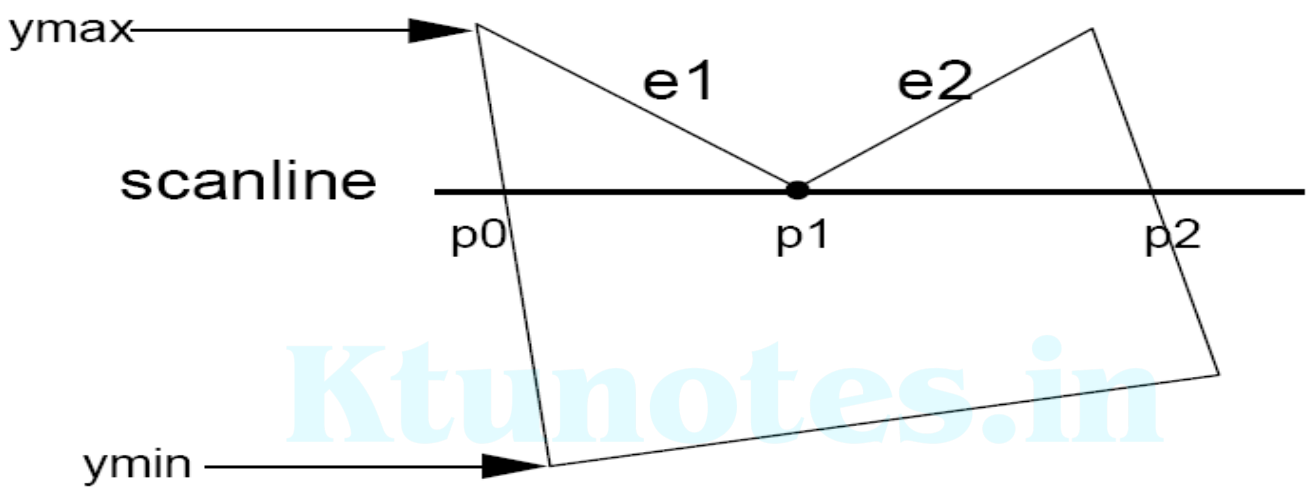
For y = ymin to ymax

1) intersect scanline y with each edge
2) sort interesections by increasing x
   [p0,p1,p2,p3]
3) fill pairwise (p0 –> p1, p2–> p3, ...)



Special handling (cont'd)

c) Intersection is an edge end point



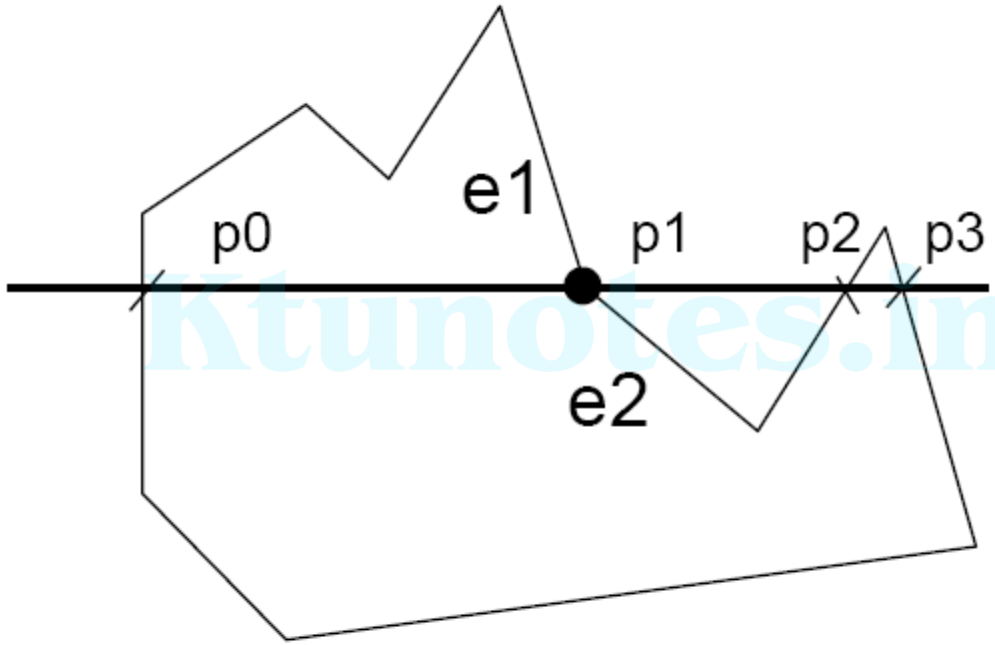
Intersection points: (p0, p1, p2) ???

–> (p0,p1,p1,p2) so we can still fill pairwise

–> In fact, if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point p1 twice. Keep both of the p1.

Special handling (cont'd)

c) Intersection is an edge end point (cont'd)



However, in this case we don't want to count p1 twice (p0,p1,p1,p2,p3), otherwise we will fill pixels between p1 and p2, which is wrong

Polygon Fill Algorithm

- Consider the next Figure.
- It shows two scan lines that cross a polygon fill area and intersect a vertex.
- Scan line *y'* intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.
- But scan line *y* intersects five polygon edges.

Polygon Fill Algorithm

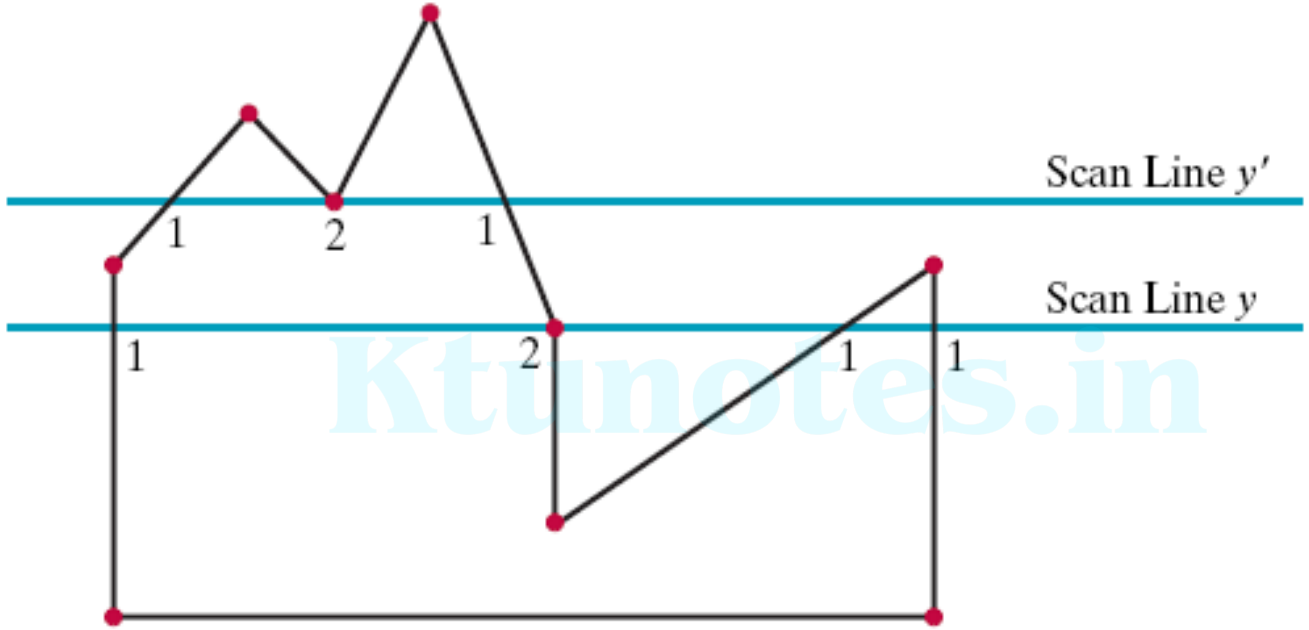


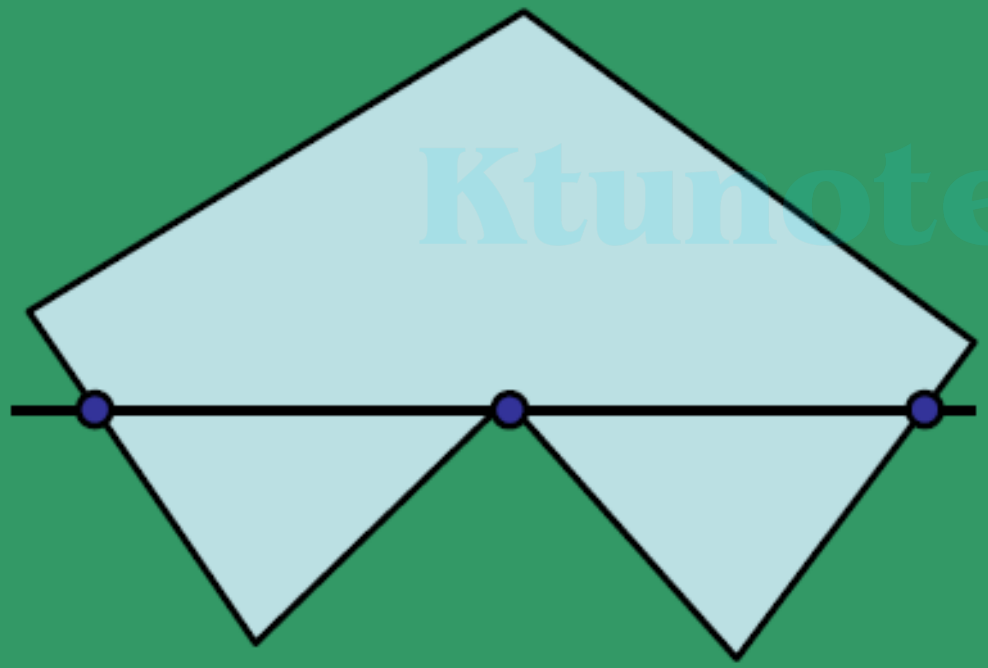
FIGURE 4–21 Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

Polygon Fill Algorithm

- To identify the interior pixels for scan line *y*, we must count the vertex intersection as only one point.

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

Two different cases of scanlines passing through the vertex of a polygon



Case - I

← *Add one more intersection:*

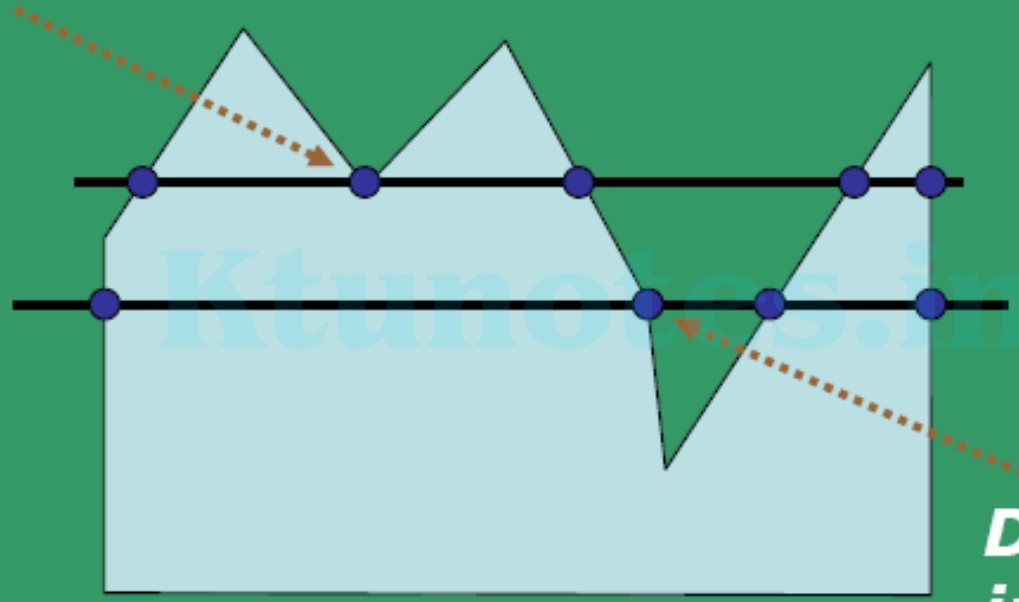
3 -> 4

SAINTGITS
LEARN.GROW.EXCEL

Case - II

Add one more intersection:

5 -> 6;

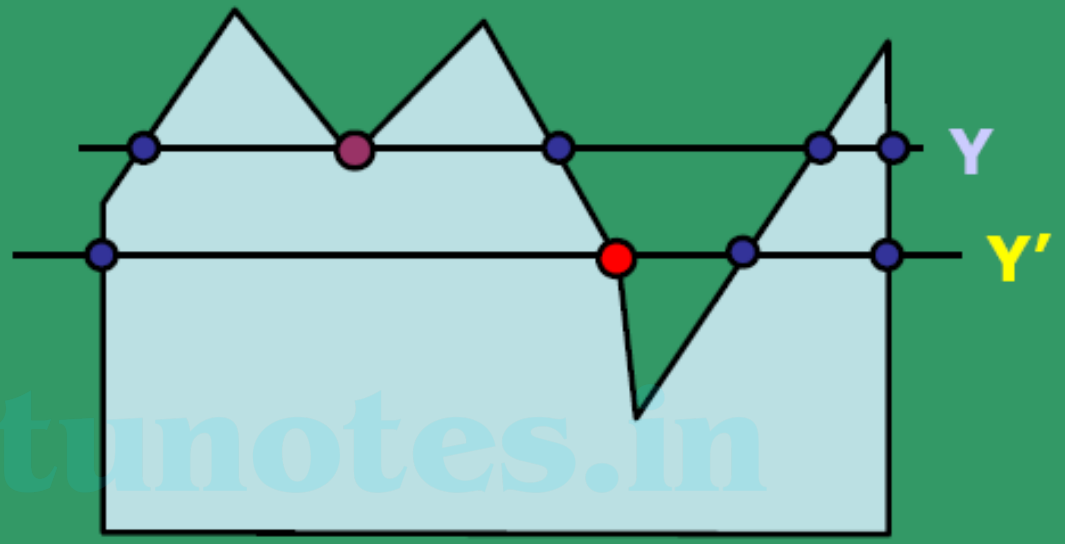


Do not add intersection, keep 4;

HOW ??

SAINTGITS
LEARN.GROW.EXCEL

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

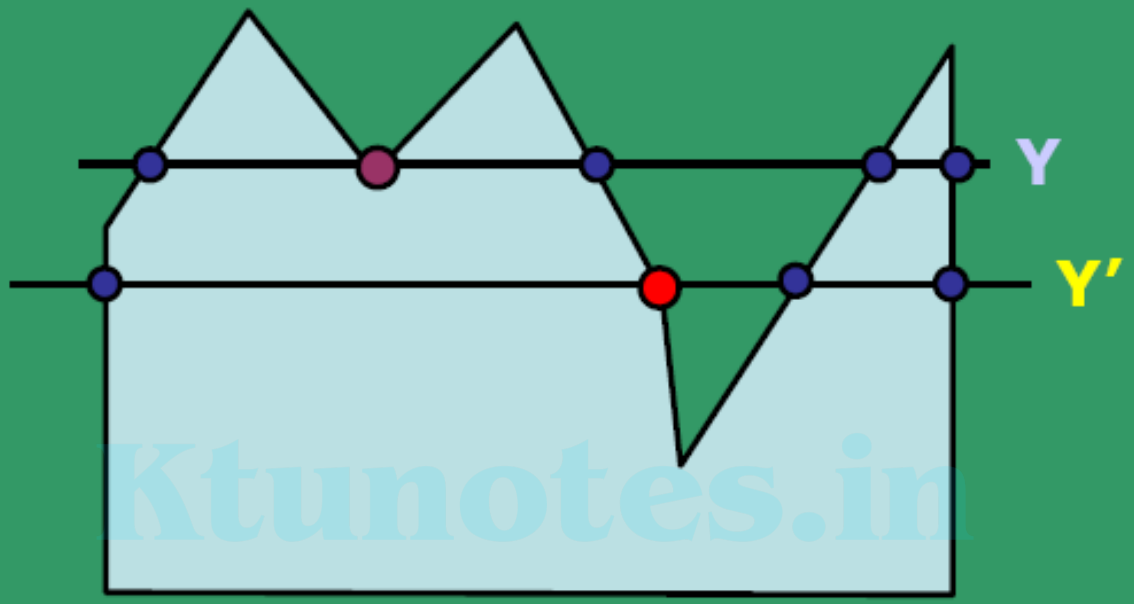


For Y, the edges at the vertex are on the same side of the scanline.

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

In this case, we require additional processing.

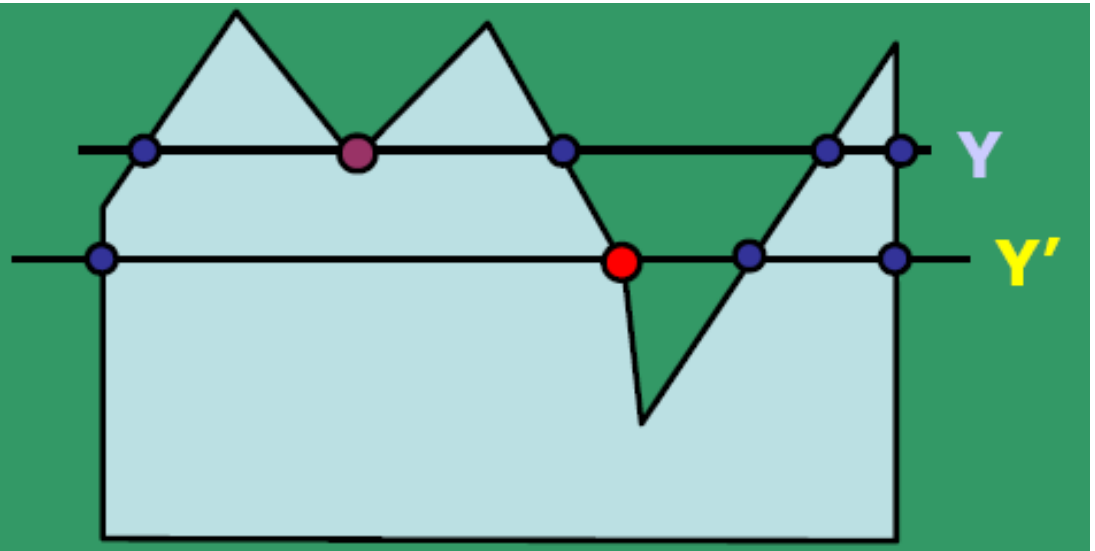
Vertex counting in a scanline



- **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).

SAINTGITS
LEARN.GROW.EXCEL

Vertex counting in a scanline



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 scanline passing through it.

Else the shared vertex represents a *local maximum (or minimum)* on the polygon boundary. *Increment the intersection count*.

Boundary Fill Algorithm

Introduction :

- Boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary.
- This algorithm works **only if** the color with which the region has to be filled and the color of the boundary of the region are different.
- If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region.

Boundary Fill Algorithm

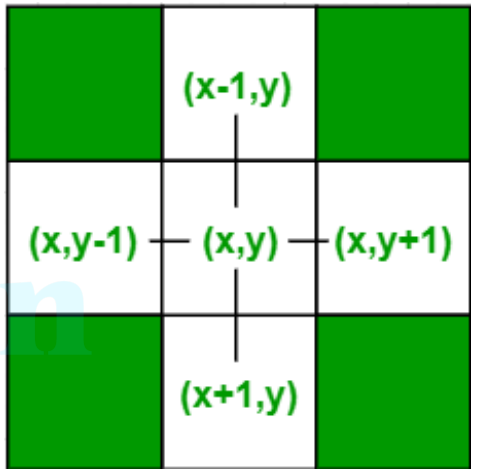
Boundary Fill Algorithm is recursive in nature

- It takes an interior point(x, y), a fill color, and a boundary color as the input.
- The algorithm starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbors of (x, y).
- If a point is found to be of fill color or of boundary color, the function does not call its neighbors and returns.
- This process continues until all points up to the boundary color for the region have been tested.
- The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

Boundary Fill Algorithm

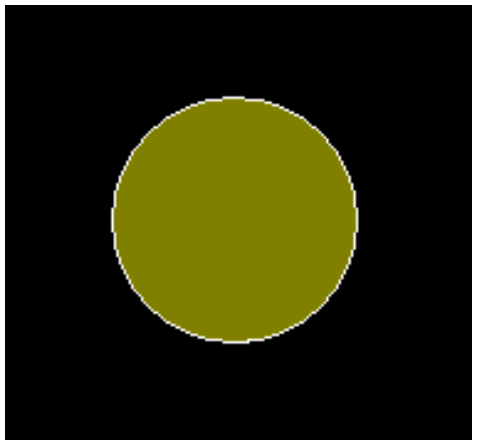
4-connected pixels :

- After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called 4-connected.



- **Algorithm:**

```
void boundaryFill4(int x, int y, int fill_color,int boundary_color)
{
   if(getpixel(x, y) != boundary_color &&
      getpixel(x, y) != fill_color)
   {
      putpixel(x, y, fill_color);
      boundaryFill4(x + 1, y, fill_color, boundary_color);
      boundaryFill4(x, y + 1, fill_color, boundary_color);
      boundaryFill4(x - 1, y, fill_color, boundary_color);
      boundaryFill4(x, y - 1, fill_color, boundary_color);
   }
}
```



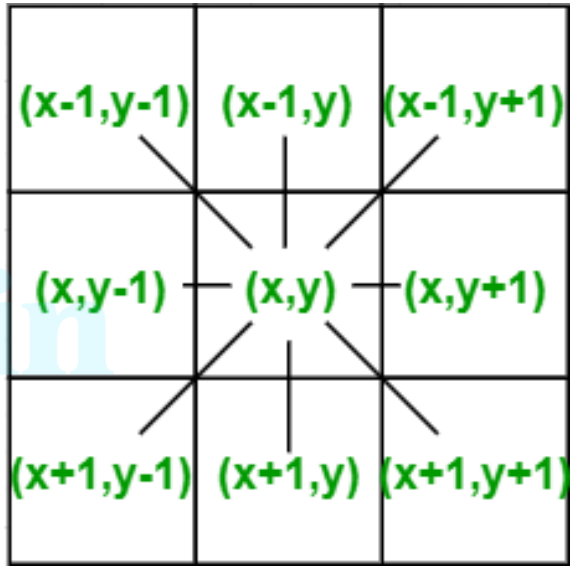
Boundary Fill Algorithm

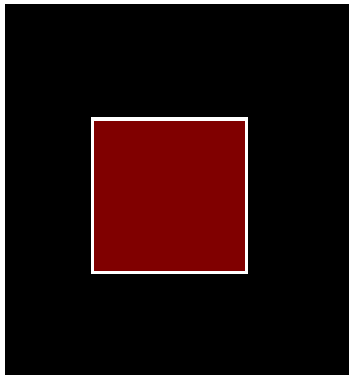
8-connected pixels :

More complex figures are filled using this approach. The pixels to be tested are the 8 neighboring pixels, the pixel on the right, left, above, below and the 4 diagonal pixels. Areas filled by this method are called 8-connected.

Algorithm :

```
void boundaryFill8(int x, int y, int fill_color,int boundary_color)
{
   if(getpixel(x, y) != boundary_color &&
      getpixel(x, y) != fill_color)
   {
      putpixel(x, y, fill_color);
      boundaryFill8(x + 1, y, fill_color, boundary_color);
      boundaryFill8(x, y + 1, fill_color, boundary_color);
      boundaryFill8(x - 1, y, fill_color, boundary_color);
      boundaryFill8(x, y - 1, fill_color, boundary_color);
      boundaryFill8(x - 1, y - 1, fill_color, boundary_color);
      boundaryFill8(x - 1, y + 1, fill_color, boundary_color);
      boundaryFill8(x + 1, y - 1, fill_color, boundary_color);
      boundaryFill8(x + 1, y + 1, fill_color, boundary_color);
   }
}
```

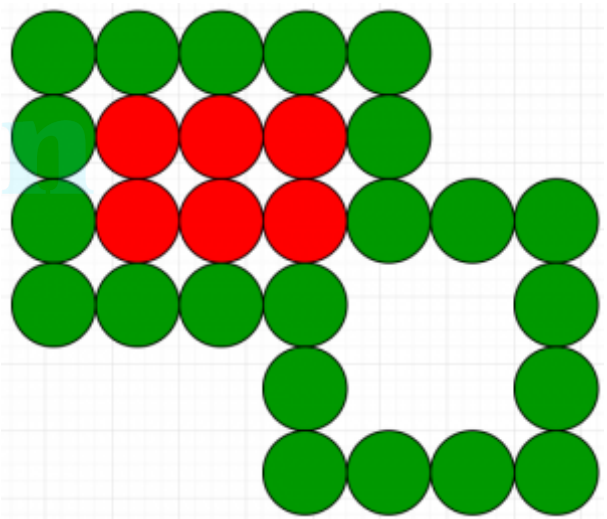




Boundary Fill Algorithm

4-connected pixels Vs 8-connected pixels :

Let us take a figure with the boundary color as GREEN and the fill color as RED. The 4-connected method fails to fill this figure completely. This figure will be efficiently filled using the 8-connected technique.



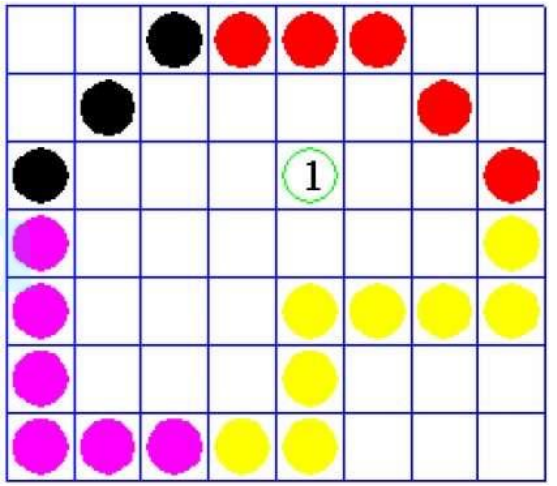
Flood fill Vs Boundary fill :

Though both Flood fill and Boundary fill algorithms color a given figure with a chosen color, they differ in one aspect. In Flood fill, all the connected pixels of a selected color get replaced by a fill color. On the other hand, in Boundary fill, the program stops when a given color boundary is found.

Flood Fill Algorithm

- Sometimes we come across an object where we want to fill the area and its boundary with different colors. We can paint such objects with a specified interior color instead of searching for particular boundary color as in boundary filling algorithm.

Instead of relying on the boundary of the object, it relies on the fill color. In other words, it replaces the interior color of the object with the fill color. When no more pixels of the original interior color exist, the algorithm is completed.

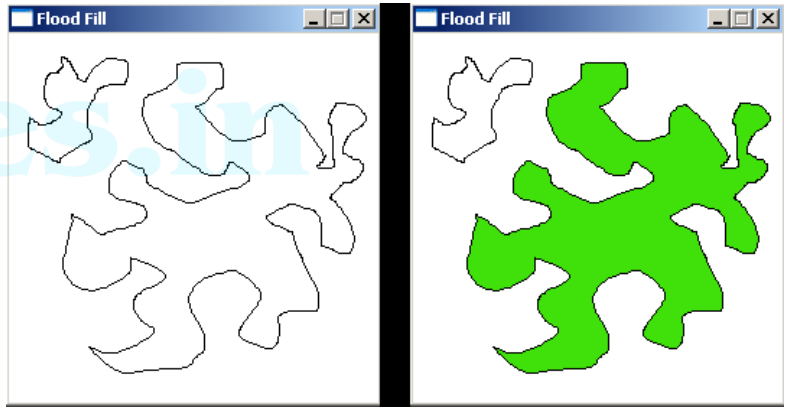


- Once again, this algorithm relies on the Four-connect or Eight-connect method of filling in the pixels. But instead of looking for the boundary color, it is looking for all adjacent pixels that are a part of the interior.

Flood Fill Algorithm

- Algorithm (4-connect):

```
floodfill4 (x, y,fill_ color, old_color: integer)
{
   If (getpixel (x, y)=old_color)
  {
   setpixel (x, y, fill_color);
   fill (x+1, y, fill_color, old_color);
    fill (x-1, y, fill_color, old_color);
   fill (x, y+1, fill_color, old_color);
   fill (x, y-1, fill_color, old_color);
    }
}
```



Flood Fill Algorithm

- Algorithm:

```
floodfill8 (x, y,fill_color, old_color: integer)
{
   If (getpixel (x, y)=old_color)
  {
   setpixel (x, y, fill_color);
   floodfill(x+1,y,old,newcol);
   floodfill(x-1,y,old,newcol);
   floodfill(x,y+1,old,newcol);
   floodfill(x,y-1,old,newcol);
   floodfill(x+1,y+1,old,newcol);
   floodfill(x-1,y+1,old,newcol);
   floodfill(x+1,y-1,old,newcol);
   floodfill(x-1,y-1,old,newcol);
  }
```

2D Transformations

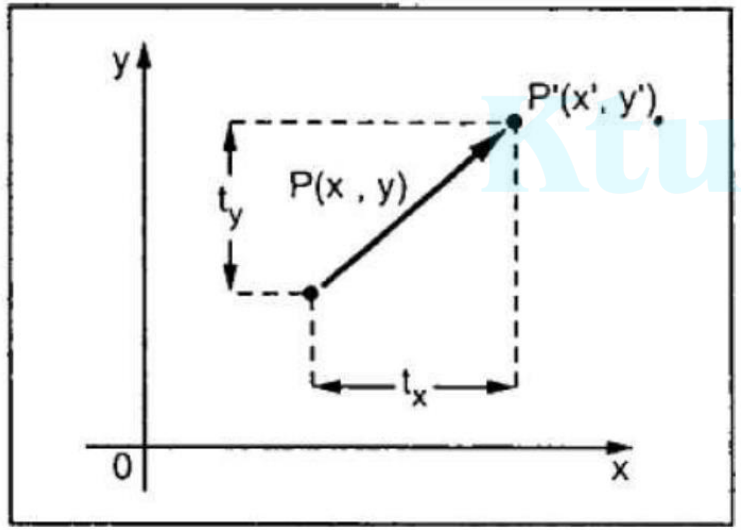
Transformation means changing some graphics into something else by applying rules. We can have various types of transformations such as translation, scaling up or down, rotation, shearing, etc. When a transformation takes place on a 2D plane, it is called 2D transformation.

Transformations play an important role in computer graphics to reposition the graphics on the screen and change their size or orientation.

2D Transformations

Translation

A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (t_x, t_y) to the original coordinate (X, Y) to get the new coordinate (X′, Y′).



From the above figure, you can write that –

$$X' = X + t_x$$

$$Y' = Y + t_y$$

Given:

$$P = (x, y)$$

$$T = (t_x, t_y)$$

We want:

$$x' = x + t_x$$

$$y' = y + t_y$$

Matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$P' = P + T$$

$$P = (-3.7, -4.1)$$

$$T = (7.1, 8.2)$$

$$x' = -3.7 + 7.1$$

$$y' = -4.1 + 8.2$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -3.7 \\ -4.1 \end{bmatrix} + \begin{bmatrix} 7.1 \\ 8.2 \end{bmatrix}$$

$$x' = 3.4$$

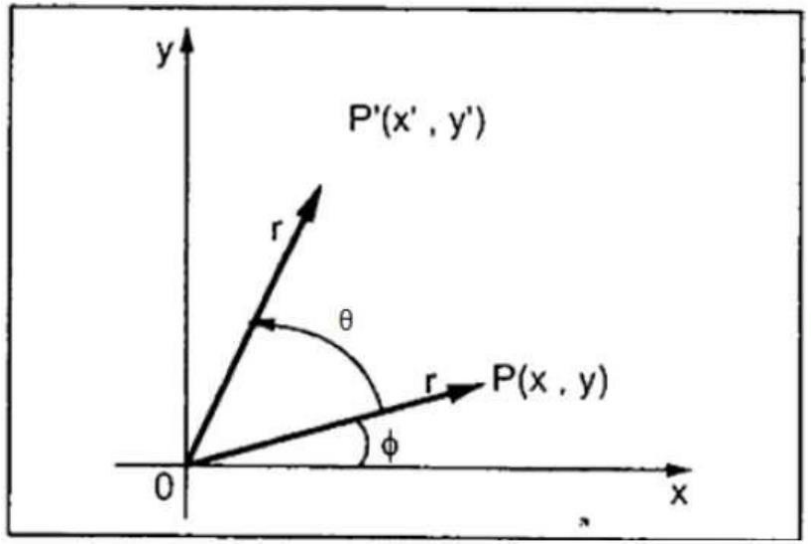
$$y' = 4.1$$

2D Transformations

Rotation

In rotation, we rotate the object at particular angle θ (theta) from its origin. From the following figure, we can see that the point P(X, Y) is located at angle φ from the horizontal X coordinate with distance r from the origin.

Let us suppose you want to rotate it at the angle θ. After rotating it to a new location, you will get a new point P′ (X′, Y′).



Using standard trigonometric the original coordinate of point P(X, Y) can be represented as –

$$X = r\cos\phi \ldots\ldots (1)$$

$$Y = r\sin\phi \ldots\ldots (2)$$

Same way we can represent the point P′ (X′, Y′) as –

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta \ldots\ldots (3)$$

$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta \ldots\ldots (4)$$

2D Transformations

Rotation:

Substituting equation (1) & (2) in (3) & (4) respectively, we will get

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

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

Representing the above equation in matrix form,

$$[X'Y'] = [XY]\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} OR$$

$$P' = P \cdot R$$

Where R is the rotation matrix

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

2D Transformations

Rotation:

The rotation angle can be positive and negative.

For positive rotation angle, we can use the above rotation matrix. However, for negative angle rotation, the matrix will change as shown below –

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

$$= \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} (\because cos(-\theta) = cos\theta \; and \; sin(-\theta) = -sin\theta)$$

2D Transformations

Rotation:

$$P = (x, y)$$

$$R = (\theta)$$

$$x = r\cos\phi$$

$$y = r\sin\phi$$

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

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

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

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

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

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

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

$$P' = R \cdot P$$

2D Transformations

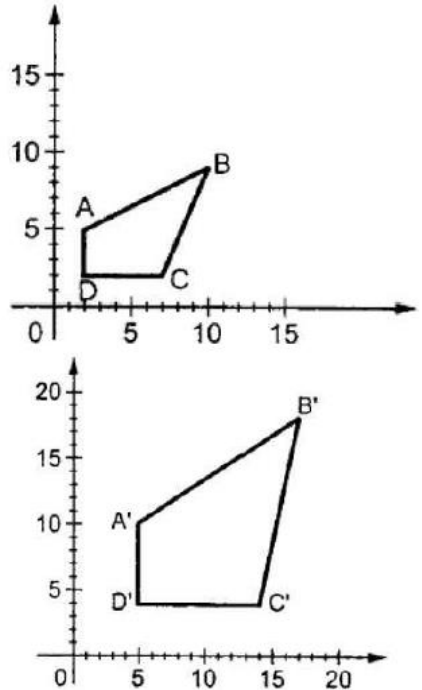
Scaling

To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.

Let us assume that the original coordinates are (X, Y), the scaling factors are (S_X, S_Y), and the produced coordinates are (X', Y'). This can be mathematically represented as shown below –

$$X' = X \cdot S_X \text{ and } Y' = Y \cdot S_Y$$

If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.



2D Transformations - Scaling

- Given:

$$P = (x, y)$$
$$S = (s_x, s_y)$$

- We want:

$$x' = s_x x$$
$$y' = s_y y$$

- Matrix form:

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

$$P = (1.4, 2.2)$$
$$S = (3, 3)$$

$$x' = 3 * 1.4$$
$$y' = 3 * 2.2$$
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1.4 \\ 2.2 \end{bmatrix}$$
$$x' = 4.2$$
$$y' = 6.6$$

Matrix Representations and Homogeneous Coordinates

- Many graphics applications involve sequences of geometric transformations
  - Animations
  - Design and picture construction applications
- We will now consider matrix representations of these operations
  - Sequences of transformations can be efficiently processed using matrices

Matrix Representations and Homogeneous Coordinates

- To produce a sequence of operations, such as scaling followed by rotation then translation, we could calculate the transformed coordinates one step at a time

- A more efficient approach is to combine transformations, without calculating intermediate coordinate values

Matrix Representations and Homogeneous Coordinates

- Multiplicative and translational terms for a 2D geometric transformation can be combined into a single matrix if we expand the representations to 3 by 3 matrices
  - We can use the third column for translation terms, and all transformation equations can be expressed as matrix multiplications

Matrix Representations and Homogeneous Coordinates

- Expand each 2D coordinate (x,y) to three element representation (x_h,y_h,h) called **homogeneous coordinates**
- h is the **homogeneous parameter** such that $x = x_h/h, \quad y = y_h/h,$
- A convenient choice is to choose h = 1

Matrix Representations and Homogeneous Coordinates

- 2D Translation Matrix

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

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

Matrix Representations and Homogeneous Coordinates

- 2D 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**(θ)·**P**

Matrix Representations and Homogeneous Coordinates

- 2D 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, $\boldsymbol{P'} = \boldsymbol{S}(s_x, s_y) \cdot \boldsymbol{P}$

2D Composite Transformations

- Composite 2D Translations
  - If two successive translation are applied to a point P, then the final transformed location P' is calculated as

$$\mathbf{P}' = \mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1}) \cdot \mathbf{P} = \mathbf{T}(t_{x1} + t_{x2}, t_{y1} + t_{y2}) \cdot \mathbf{P}$$

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

2D Composite Transformations

- Composite 2D Rotations

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

$$\begin{bmatrix} \cos\Theta_2 & -\sin\Theta_2 & 0 \\ \sin\Theta_2 & \cos\Theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\Theta_1 & -\sin\Theta_1 & 0 \\ \sin\Theta_1 & \cos\Theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\Theta_1+\Theta_2) & -\sin(\Theta_1+\Theta_2) & 0 \\ \sin(\Theta_1+\Theta_2) & \cos(\Theta_1+\Theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2D Composite Transformations

- Composite 2D Scaling

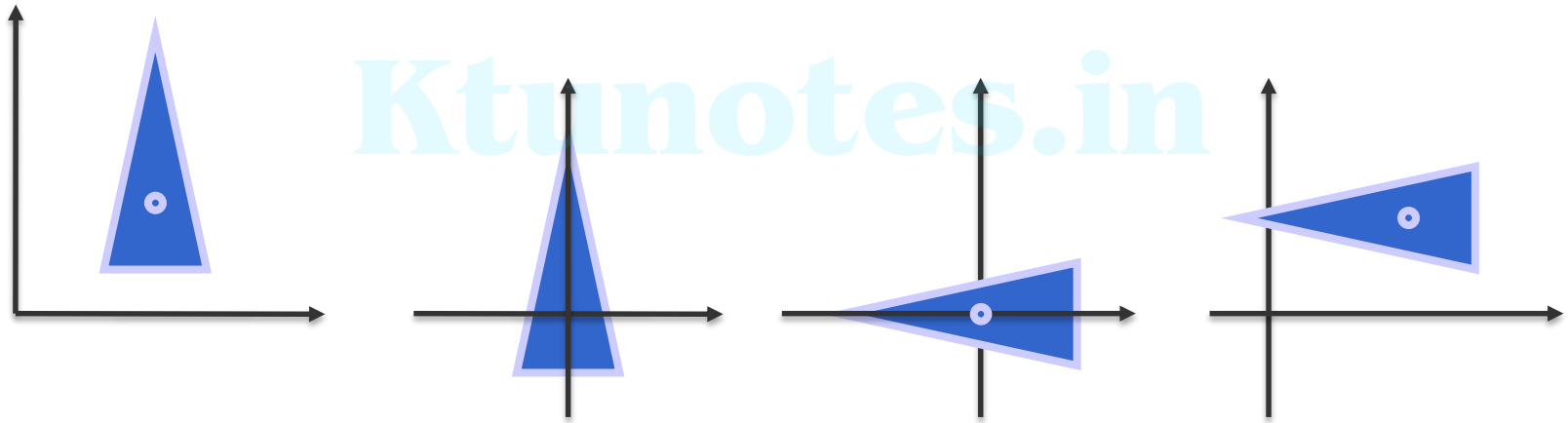
$$\mathbf{S}(s_{x_2}, s_{y_2}) \cdot \mathbf{S}(s_{x_1}, s_{y_1}) = \mathbf{S}(s_{x_1} \cdot s_{x_2},\ s_{y_1} \cdot s_{y_2})$$

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

General Pivot Point Rotation

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

General Pivot Point Rotation



General Pivot Point Rotation

- General 2D Pivot-Point Rotation

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

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

Problem:

Given a triangle ABC A(4,6) B(2,2)C(6,2). Rotate it by 90 degree anticlockwise about the point (3,3).

Solution:

We have $x_r=3$ $y_r=3$

R(Θ)=90˚

1.Translate to origin T(-3,-3).
2. Then Rotate anticlockwise.
3.Translate back to original position T(3,3)

Object Matrix

$$O=\begin{bmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 6 \\ 6 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$O' = T(x_r, y_r) \cdot R(\theta) \cdot T(-x_r, -y_r) \cdot O$$

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

Translation Matrix

$$T(-xr, -yr) = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix

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

Translation Matrix

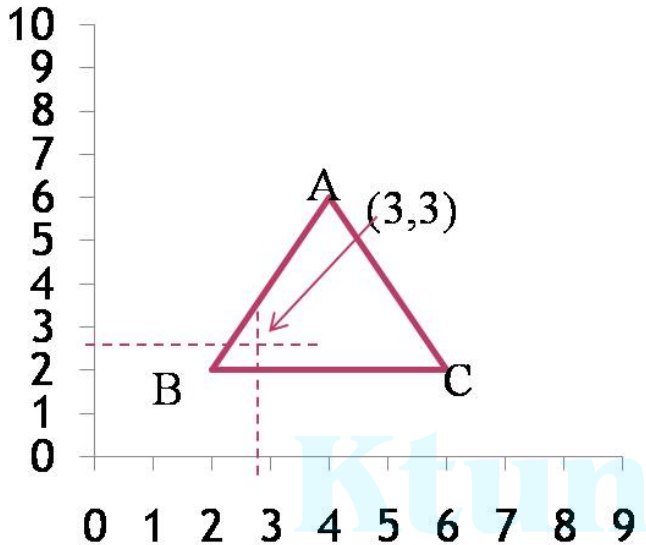
$$T(xr, yr) = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

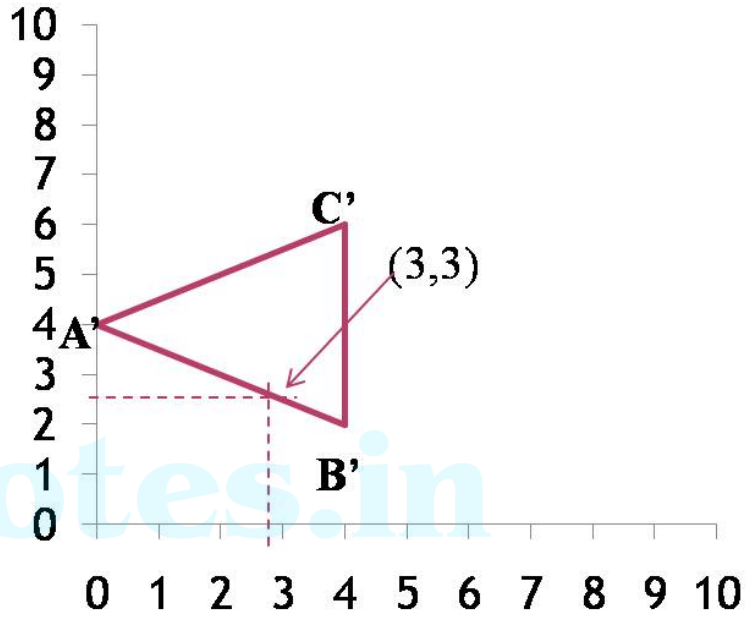
$$o' = \begin{bmatrix} \cos\Theta & -\sin\Theta & x_r(1-\cos\Theta) + y_r \sin\Theta \\ \sin\Theta & \cos\Theta & y_r(1-\cos\Theta) - x_r \sin\Theta \\ 0 & 0 & 1 \end{bmatrix} o$$

$$O' = \begin{bmatrix} 0 & -1 & 6 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 6 \\ 6 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$O' = \begin{bmatrix} 0 & 4 & 4 \\ 4 & 2 & 6 \\ 1 & 1 & 1 \end{bmatrix}$$

A'=(0,4)
B'=(4,2)
C'=(4,6)





ORIGINAL TRIANGLE ABC

FINAL TRIANGLE A'B'C'

A=(4,6)
B=(2,2)
C=(6,2)

A'=(0,4)
B'=(4,2)
C'=(4,6)

So as we have substituted all the values in the corresponding matrices. We will arrange them in sequence. **Always keep in mind matrix multiplication is associative but not commutative**. Arrange them from right to left and object matrix being the rightmost.

$$O' = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 6 \\ 6 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$O' = \begin{bmatrix} 0 & -1 & 3 \\ 1 & 0 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 6 \\ 6 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

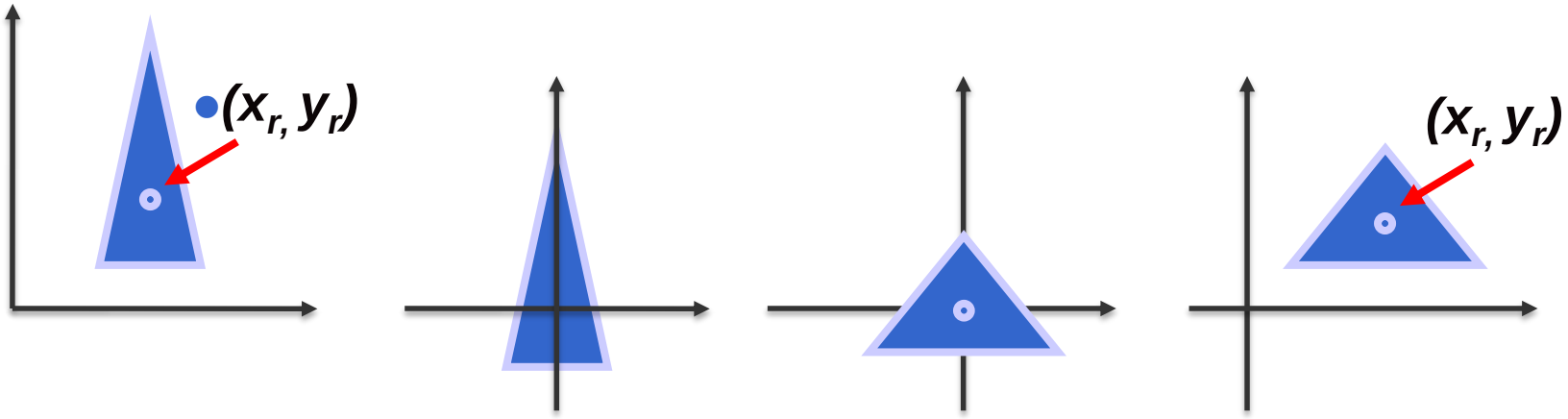
$$O' = \begin{bmatrix} 0 & -1 & 6 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 6 \\ 6 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

$$O' = \begin{bmatrix} 0 & 4 & 4 \\ 4 & 2 & 6 \\ 1 & 1 & 1 \end{bmatrix}$$

A'=(0,4)
B'=(4,2)
C'=(4,6)

General fixed Point Scaling

- Steps:
  1. Translate the object so that the fixed point coincides with the coordinate origin.
  2. Scale the object about the origin.
  3. Translate the object so that the pivot point is returned to its original position.



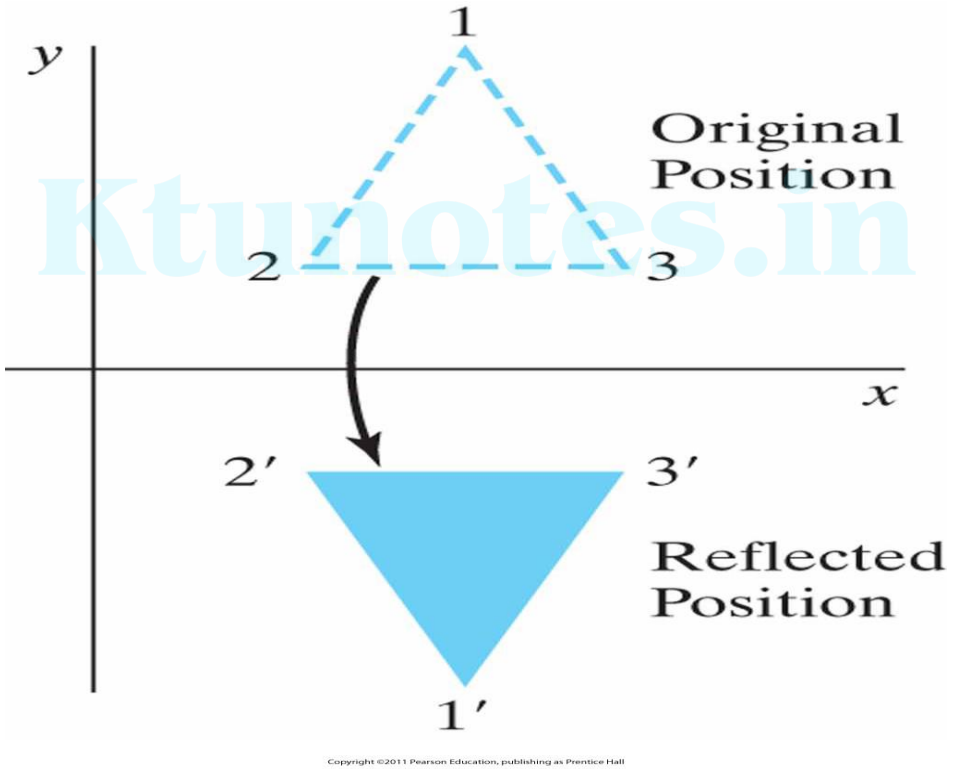
General fixed Point Scaling

- General 2D Fixed-Point Scaling:

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

Other 2D Transformations

- Reflection
  - Transformation that produces a mirror image of an object



Other 2D Transformations

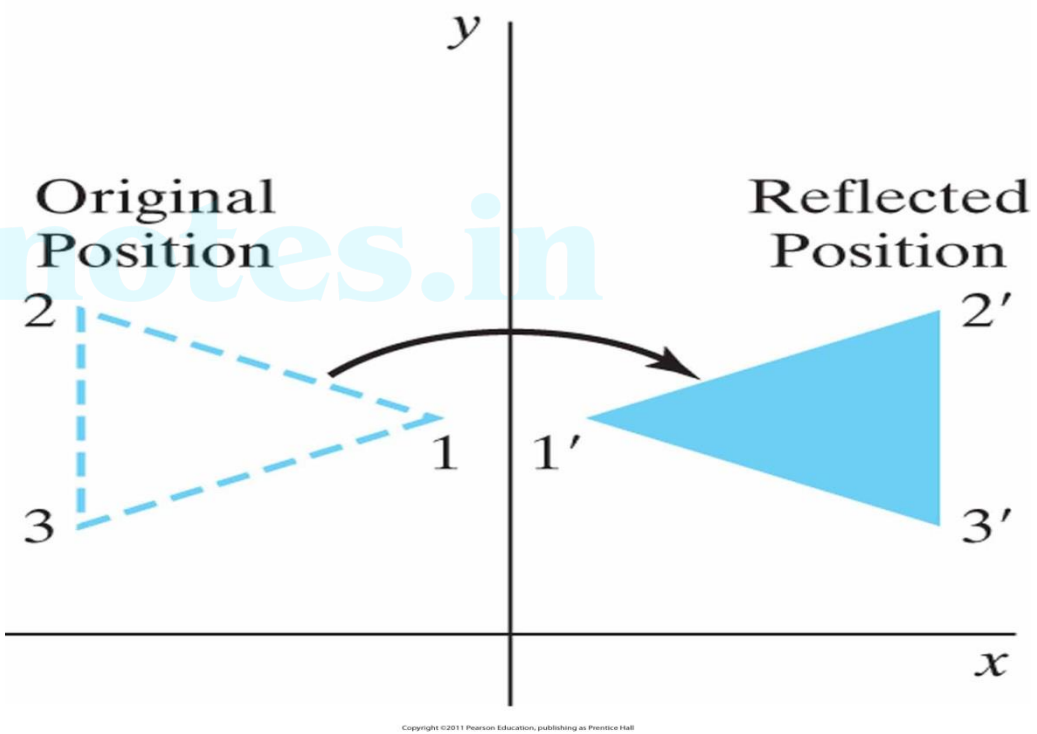
- Reflection
  - Image is generated relative to an axis of reflection by rotating the object 180° about the reflection axis
  - Reflection about the line y=0 (the x axis) (previous slide)

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

Other 2D Transformations

- Reflection
  - Reflection about the line x=0 (the y axis)

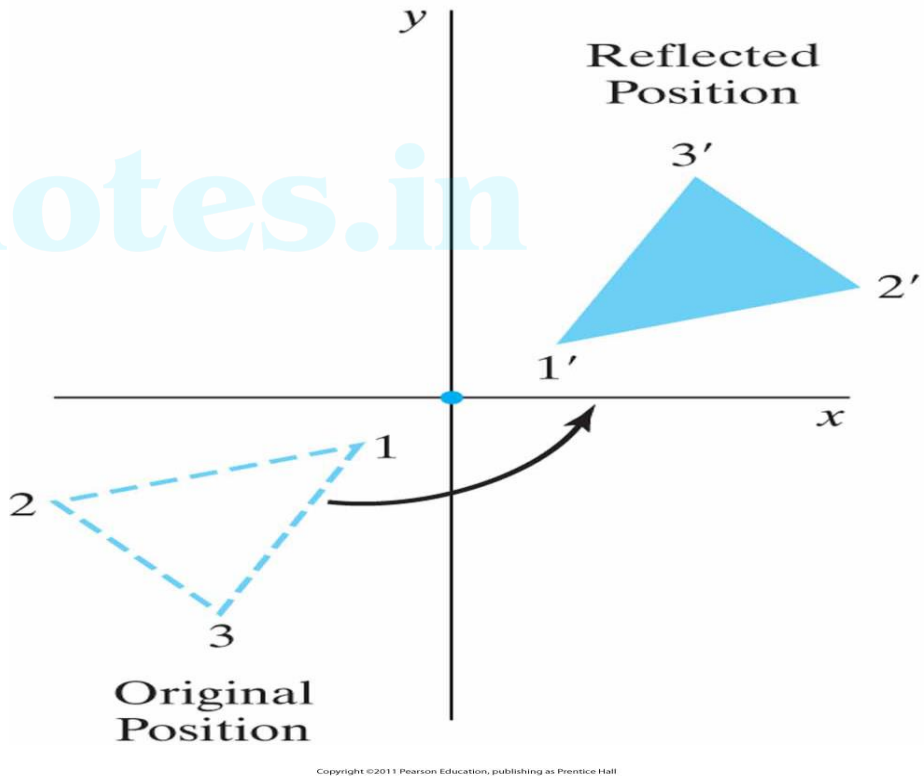
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Other 2D Transformations

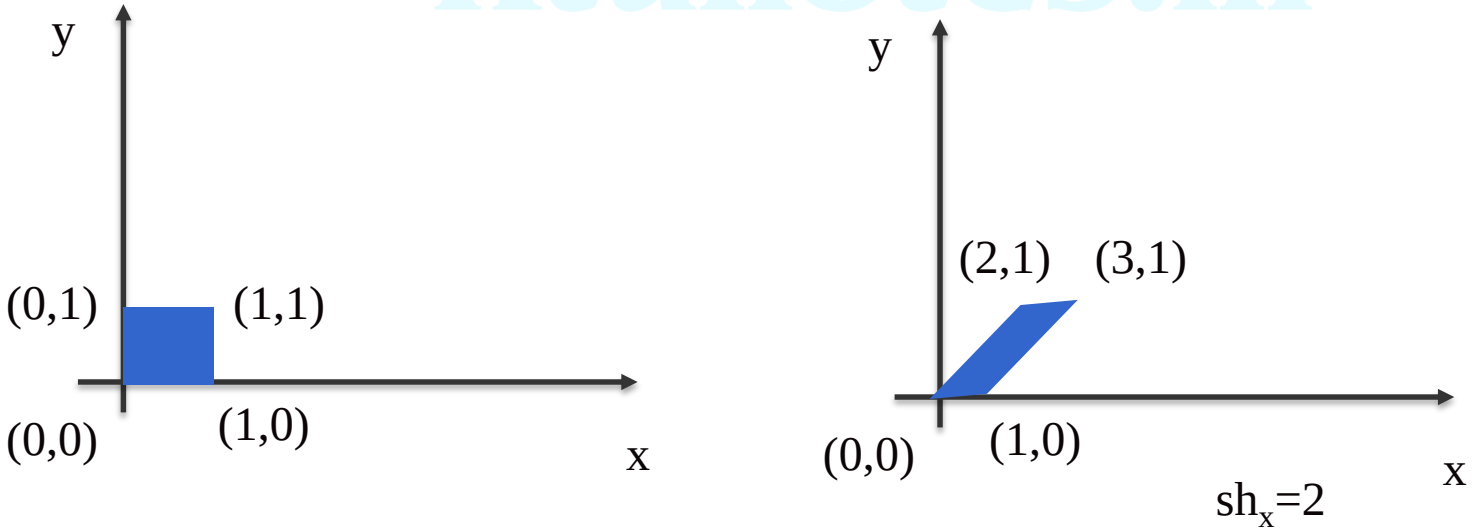
- Reflection about the origin

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



Other 2D Transformations

- Shear
  - Transformation that distorts the shape of an object such that the transformed shape appears as the object was pushed to slide in one (x or y) direction.



Other 2D Transformations

- Shear
  - An x-direction shear relative to the x axis

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad x' = x + sh_x \cdot y \qquad y' = y$$

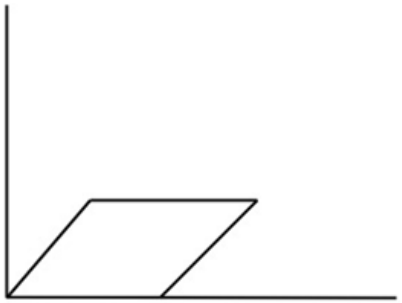
  - An y-direction shear relative to the y axis

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad y' = y + sh_y \cdot x \qquad x' = x$$

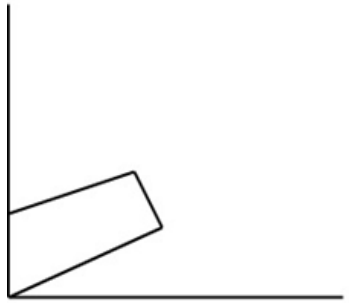
Shearing in X-Y directions: Here layers will be slided in both x as well as y direction. The sliding will be in horizontal as well as vertical direction. The shape of the object will be distorted. The matrix of shear in both directions is given by:

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

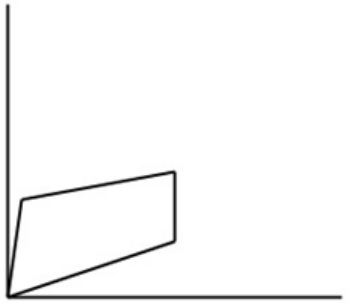
Original Object



Shear in X direction



Shear in Y direction



Shear in both directions

Other 2D Transformations

- Shear
  - x-direction shear relative to other reference lines

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & sh_x & -sh_x * y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$x' = x + sh_x * (y - y_{ref})$$

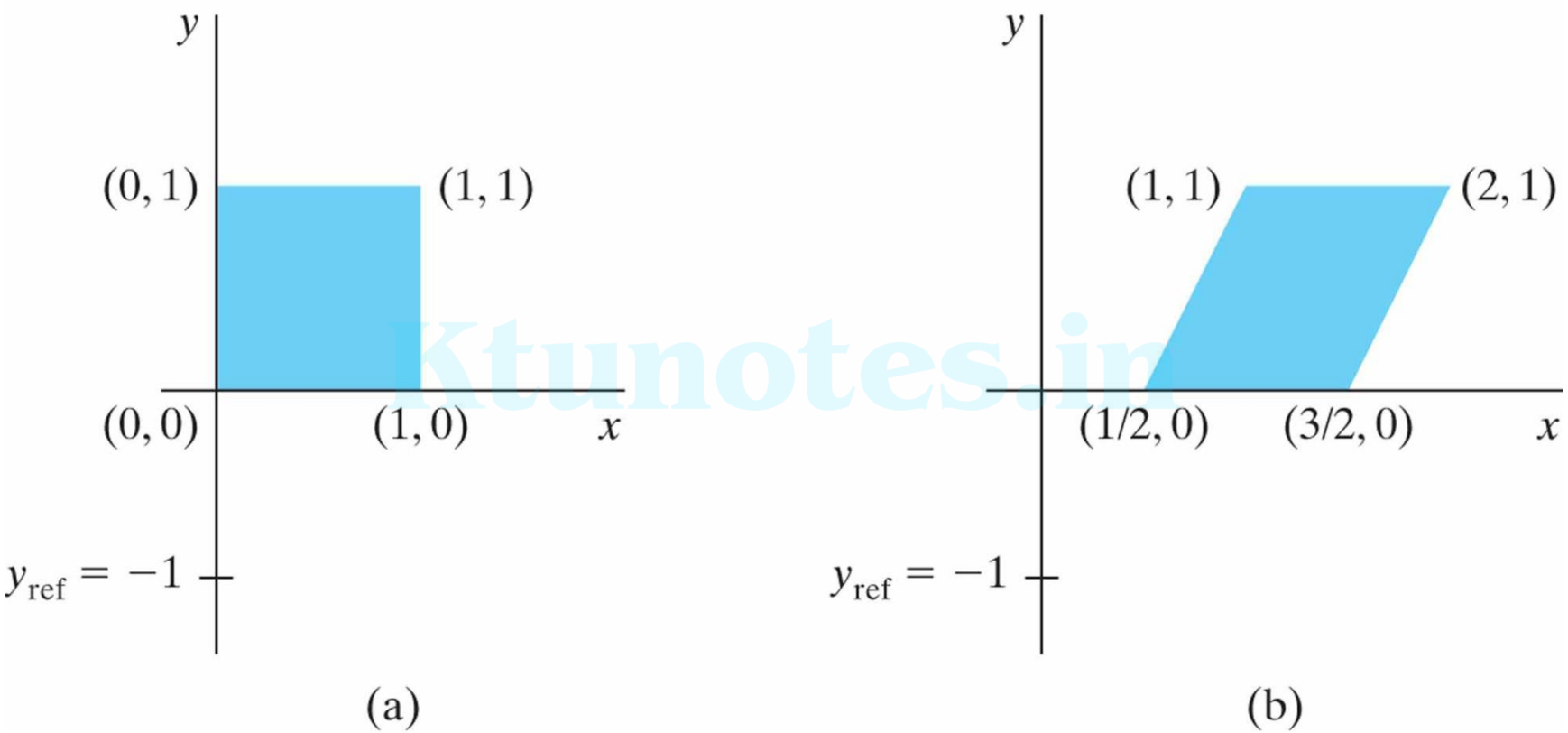
$$y' = y$$

Example

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

$$x' = x + sh_x * (y - y_{ref})$$

$$y' = y$$



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

Other 2D Transformations

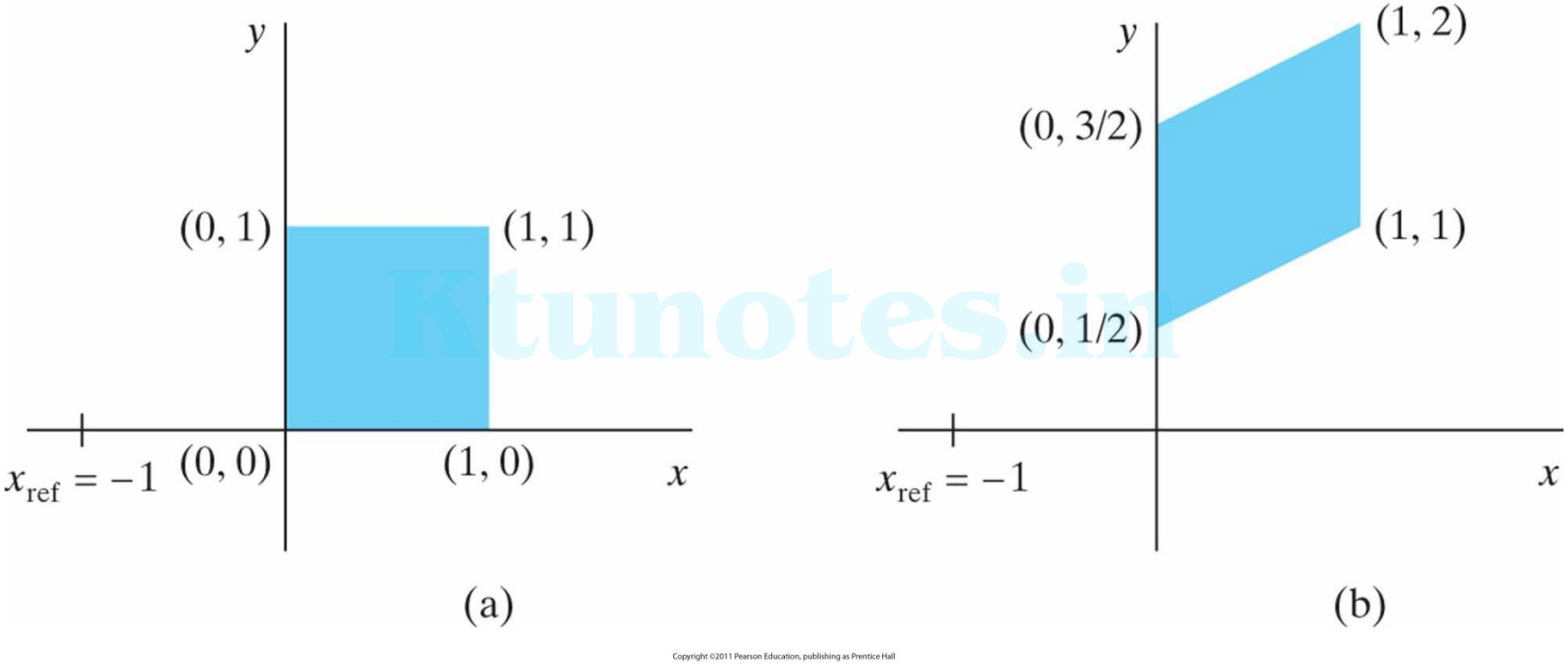
- Shear
  - y-direction shear relative to the line $x = x_{ref}$

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

$$x' = x$$

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

Other 2D Transformations – Shear - Example



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

Basic 3D Transformations

Recap – 2D Transformations

- Translation: $\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$
- Scaling: $\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Rotation: $\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Basic 3D Transformations

- When the transformation takes place on a 3D plane .it is called 3D transformation.
- Generalize from 2D by including **z** coordinate

Straight forward for translation and scale, rotation more difficult

Homogeneous coordinates: 4 components

Transformation matrices: 4×4 elements

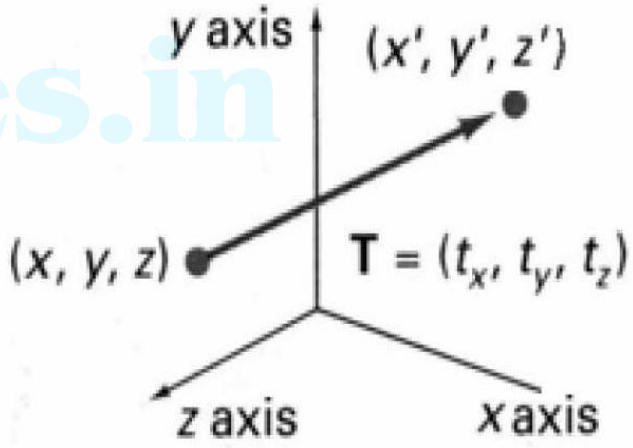
$$\begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Basic 3D Transformations - **Translation**

- Moving of object is called Translation.
- In 3 dimensional homogeneous coordinate representation , a point is transformed from position P = (x, y , z) to P'=(x', y', z')
- This can be written as:-

Using **P' = T . P**

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



- The matrix representation is equivalent to the three equation.

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

Basic 3D Transformations - **Rotation**

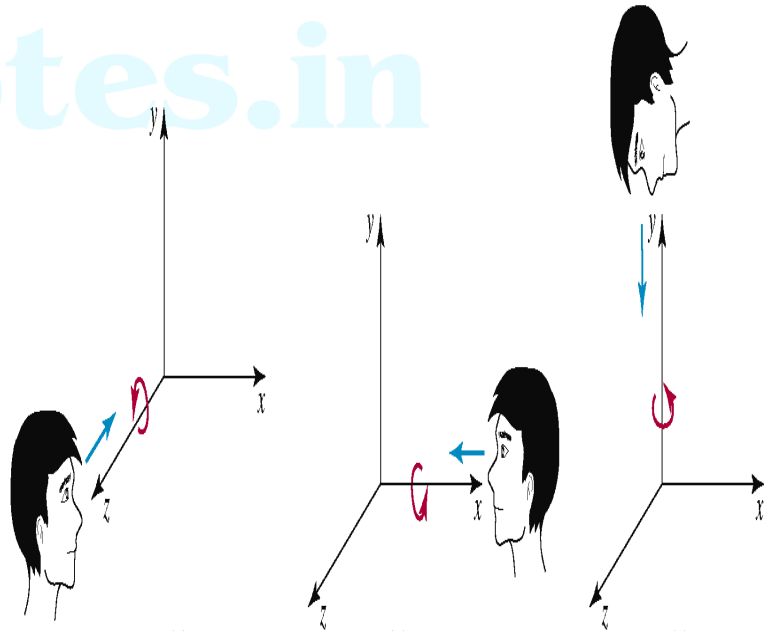
Where an object is to be rotated about an axis that is parallel to one of the coordinate axis, we can obtain the desired rotation with the following transformation sequence.

Coordinate axis rotation

Z- axis Rotation

Y-axis Rotation

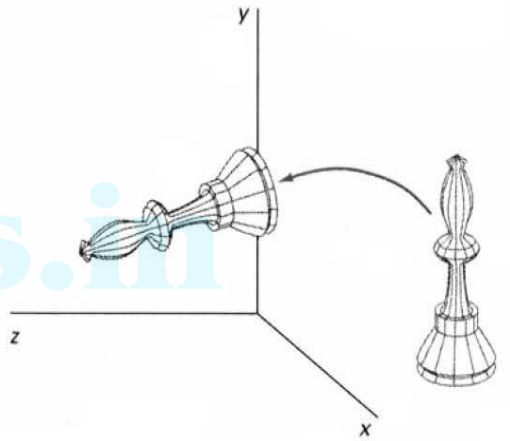
X-axis Rotation



Basic 3D Transformations - **Rotation**

The equation for **X-axis Rotation**

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



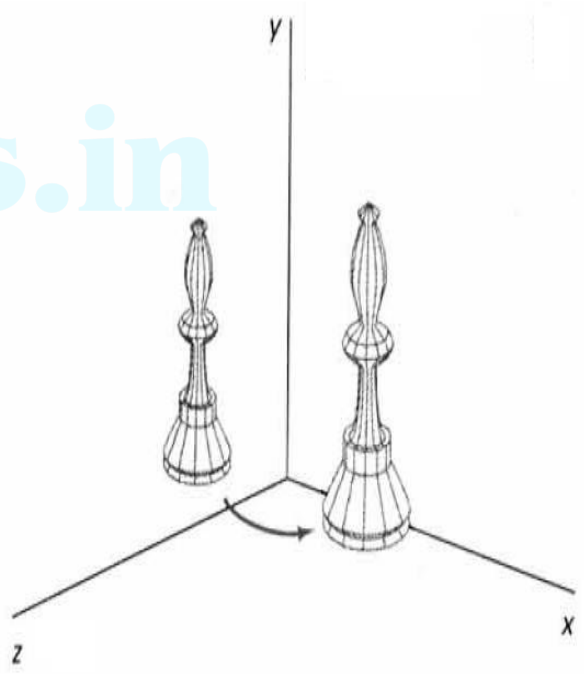
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Basic 3D Transformations - **Rotation**

The equation for **Y-axis Rotaion**

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

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

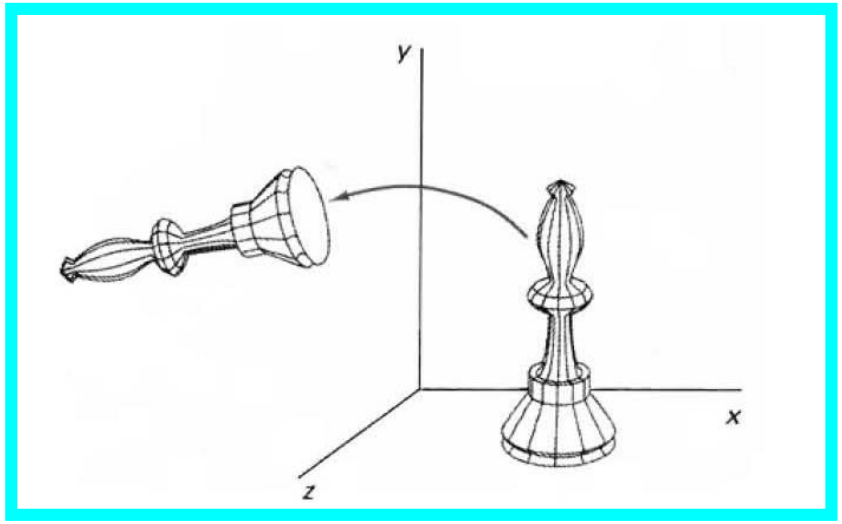


Basic 3D Transformations - **Rotation**

The equation for **Z-axis rotation**

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

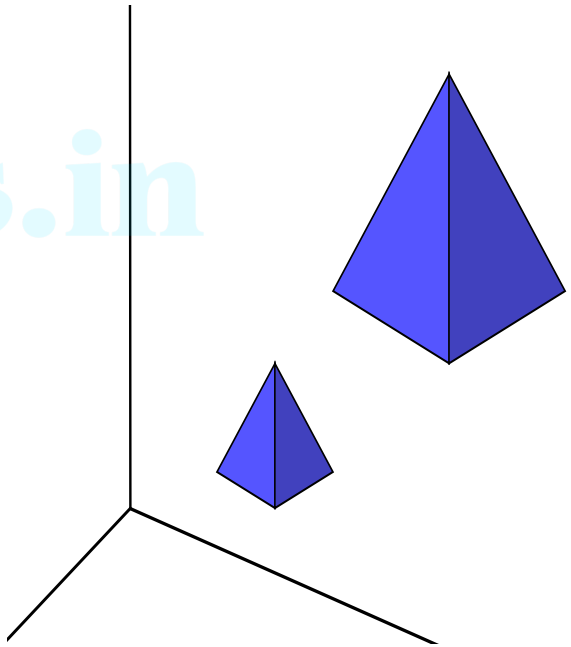
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Basic 3D Transformations - **Scaling**

- Changes the size of the object and repositions the object relative to the coordinate origin.

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



- The equations for scaling

$$\mathbf{x' = x \cdot s_x}$$
$$\mathbf{y' = y \cdot s_y}$$
$$\mathbf{z' = z \cdot s_z}$$

Basic 3D Transformations - **Reflection**

Reflection about **x-axis**:

x'=x y'=-y z'=-z

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

Reflection about **y-axis**:

y'=y x'=-x z'=-z

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

- Reflection about z-axis:-

x'=-x y'=-y z'=z

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