COMPUTER GRAPHICS

UNIT-2

Raster Graphics:

In computer graphics, a raster graphic represents a two-dimensional picture as a rectangular matrix or grid of square pixels, viewable via a computer display, paper, or other display medium.

A raster is technically characterized by the width and height of the image in pixels and by the number of bits per pixel.

Raster images are stored in image files with varying dissemination, production, generation, and acquisition formats.

Raster Graphics Algorithms:

1. Line drawing algorithms:

In computer graphics, a line drawing algorithm is an algorithm for approximating a line segment on discrete graphical media, such as pixel-based displays and printers. On such media, line drawing requires an approximation. Basic algorithms rasterize lines in one color. A better representation with multiple color gradations requires an advanced process, spatial anti-aliasing.

There are two types of Line drawing algorithms:

A. DDA Algorithm:

DDA stands for Digital Differential Analyzer. It is an incremental method of scan conversion of line. In this method calculation is performed at each step but by using results of previous steps.

Suppose at step i, the pixels is (xi, yi)

The line of equation for step i

yi=mxi+b.....................equation 1

Next value will be

yi+1=mxi+1+b................equation 2

m =

yi+1-yi=Δy......................equation 3

yi+1-xi=Δx.....................equation 4

yi+1=yi+Δy

Δy=mΔx
yi+1=yi+mΔx
Δx=Δy/m
xi+1=xi+Δx
xi+1=xi+Δy/m

Case1: When |M|<1 then (assume that x12)
x= x1, y=y1 set Δx=1
yi+1=y1+m,   x=x+1
Until x = x2

Case2: When |M|<1 then (assume that y12)
x= x1, y=y1 set Δy=1
xi+1=,   y=y+1
Until y → y2

Advantage:

- It is a faster method than method of using direct use of line equation.
- This method does not use multiplication theorem.
- It allows us to detect the change in the value of x and y, so plotting of same point twice is not possible.
- This method gives overflow indication when a point is repositioned.
- It is an easy method because each step involves just two additions.

Disadvantage:

- It involves floating point additions rounding off is done. Accumulations of round off error cause accumulation of error.
- Rounding off operations and floating-point operations consumes a lot of time.
- It is more suitable for generating line using the software. But it is less suited for hardware implementation.

Algorithm:

Step1: Start Algorithm

Step2: Declare x1,y1,x2,y2,dx,dy,x,y as integer variables.

Step3: Enter value of x1,y1,x2,y2.

Step4: Calculate dx = x2-x1

Step5: Calculate dy = y2-y1

```
Step6: If ABS (dx) > ABS (dy)
    Then step = abs (dx)
    Else

Step7: xinc=dx/step
    yinc=dy/step
    assign x = x1
    assign y = y1

Step8: Set pixel (x, y)

Step9: x = x + xinc
    y = y + yinc
    Set pixels (Round (x), Round (y))

Step10: Repeat step 9 until x = x2

Step11: End Algorithm
```

Example:

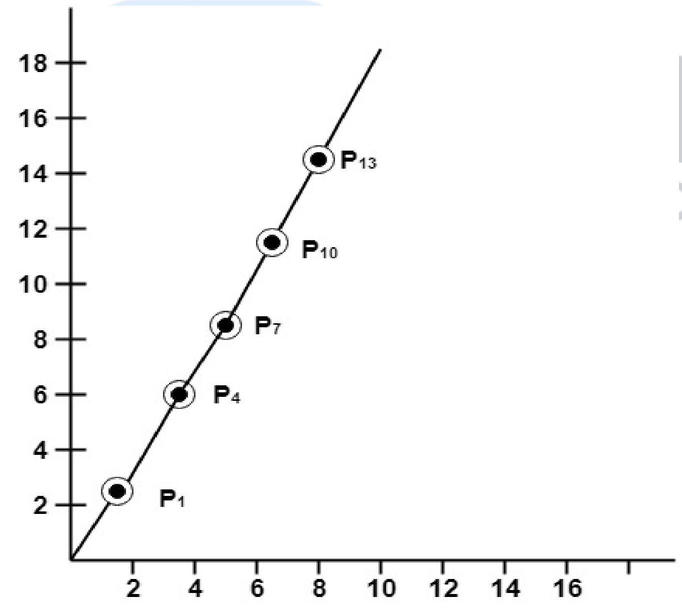
If a line is drawn from (2, 3) to (6, 15) with use of DDA. How many points will need to generate such line?

Solution: P1 (2,3)   P11 (6,15)

x1=2
y1=3
x2= 6
y2=15
dx = 6 - 2 = 4
dy = 15 - 3 = 12
m = $\frac{dy}{dx} = \frac{12}{4}$

For calculating next value of x takes x = x + $\frac{1}{m}$

$P_1(2,3)$	point plotted
$P_2(2\frac{1}{3},4)$	point plotted
$P_3(2\frac{2}{3},5)$	point not plotted
$P_4(3,6)$	point plotted
$P_5(3\frac{1}{3},7)$	point not plotted
$P_6(3\frac{2}{3},8)$	point not plotted
$P_7(4,9)$	point plotted
$P_8(4\frac{1}{3},10)$	point not plotted
$P_9(4\frac{2}{3},11)$	point not plotted
$P_{10}(5,12)$	point plotted
$P_{11}(5\frac{1}{3},13)$	point not plotted
$P_{12}(5\frac{2}{3},14)$	point not plotted
$P_{13}(6,15)$	point plotted



Symmetrical DDA:

The Digital Differential Analyzer (DDA) generates lines from their differential equations. The equation of a straight line is

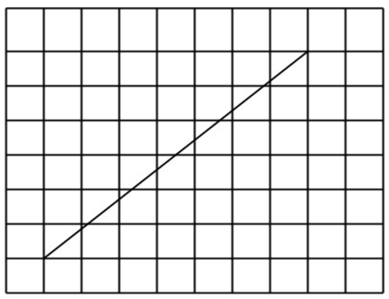
$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x}$$

The DDA works on the principle that we simultaneously increment x and y by small steps proportional to the first derivatives of x and y. In this case of a straight line, the first derivatives are constant and are proportional to Δx and Δy. Therefore, we could generate a line by incrementing x and y by $\epsilon\ \Delta x$ and $\epsilon\ \Delta y$, where ϵ is some small quantity. There are two ways to generate points

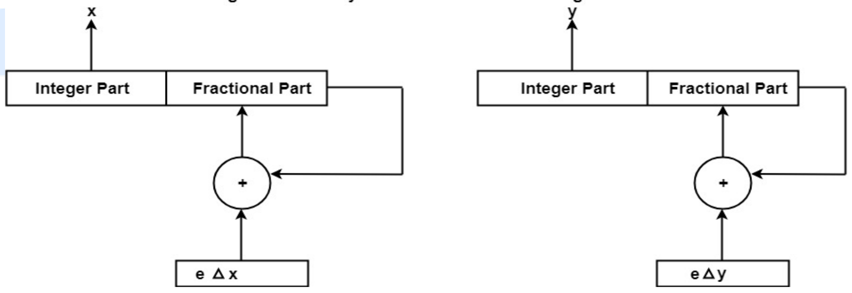
1. By rounding to the nearest integer after each incremental step, after rounding we display dots at the resultant x and y.

2. An alternative to rounding the use of arithmetic overflow: x and y are kept in registers that have two parts, integer and fractional. The incrementing values, which are both less than unity, are repeatedly added to the fractional parts and whenever the results overflow, the corresponding integer part is incremented. The integer parts of the x and y registers are used in plotting the line. In the case of the symmetrical DDA, we choose $\varepsilon=2^{-n}$,where $2^{n-1}\leq \max(|\Delta x|,|\Delta y|)<2^{\pi}$

A line drawn with the symmetrical DDA is shown in fig:



Organization of Symmetrical DDA shown in fig:



Example:

If a line is drawn from (0, 0) to (10, 5) with a symmetrical DDA

1. How many iterations are performed?

2. How many different points will be generated?

Solutions: Given: P1 (0,0) P2 (10,5)

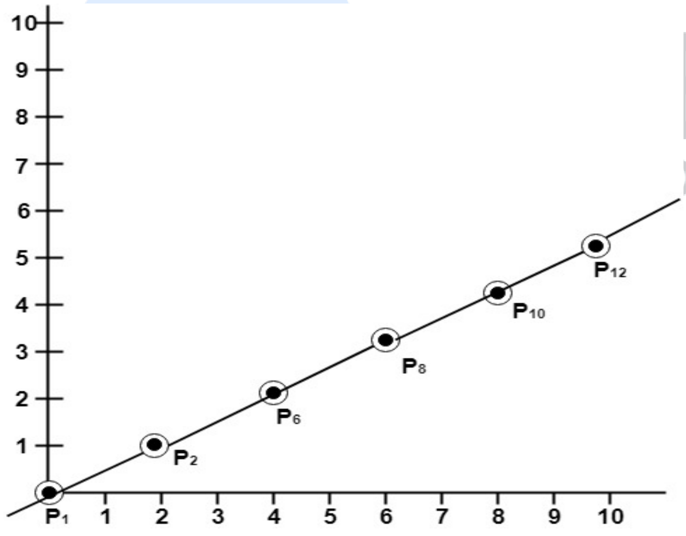
x1=0
y1=0
x2=10
y2=5

dx = 10 - 0 = 10
dy = 5 - 0 = 0

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{5-0}{10-0} = \frac{5}{10} = 0.5$$

P^1	(0,0)	will be considered starting points
P^3	(1,0.5)	point not plotted
P^4	(2, 1)	point plotted
P^5	(3, 1.5)	point not plotted
P^6	(4,2)	point plotted
P^7	(5,2.5)	point not plotted
P^8	(6,3)	point plotted
P^9	(7,3.5)	point not plotted
P^{10}	(8, 4)	point plotted
P^{11}	(9,4.5)	point not plotted
P^{12}	(10,5)	point plotted

Following Figure show line plotted using these points.



B. Bresenham's Line Algorithm

This algorithm is used for scan converting a line. It was developed by Bresenham. It is an efficient method because it involves only integer addition, subtractions, and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly.

In this method, next pixel selected is that one who has the least distance from true line.

Algorithm:

Step1: Start Algorithm

Step2: Declare variable x_1,x_2,y_1,y_2,d,i_1,i_2,dx,dy

Step3: Enter value of x_1,y_1,x_2,y_2
Where x_1,y_1 are coordinates of starting point
And x_2,y_2 are coordinates of Ending point

Step4: Calculate dx = x_2-x_1
Calculate dy = y_2-y_1
Calculate $i_1=2*dy$
Calculate $i_2=2*(dy-dx)$
Calculate $d=i_1-dx$

Step5: Consider (x, y) as starting point and x_{end} as maximum possible value of x.
If dx < 0
Then x = x_2
y = y2
$x_{end}=x_1$
If dx > 0
Then x = x1
y = y1
$x_{end}=x_2$

Step6: Generate point at (x,y)coordinates.

Step7: Check if whole line is generated.
If x > = x_{end}
Stop.

Step8: Calculate co-ordinates of the next pixel
If d < 0
Then d = d + i_1
If d ≥ 0
Then d = d + i2
Increment y = y + 1

Step9: Increment x = x + 1

Step10: Draw a point of latest (x, y) coordinates

Step11: Go to step 7

Step12: End of Algorithm

Example:

Starting and Ending position of the line are (1, 1) and (8, 5). Find intermediate points.

Solution:

x1=1

y1=1

x2=8

y2=5

dx= x2-x1=8-1=7

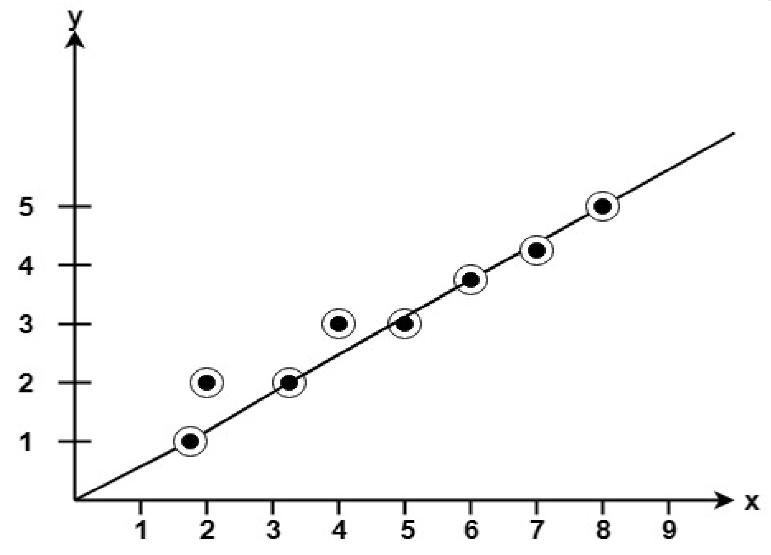
dy=y2-y1=5-1=4

I1=2* Δy=2*4=8

I2=2*(Δy-Δx)=2*(4-7)=-6

d = I1-Δx=8-7=1

x	y	d=d+I_1 or I_2
1	1	d+I_2=1+(-6)=-5
2	2	d+I_1=-5+8=3
3	2	d+I_2=3+(-6)=-3
4	3	d+I_1=-3+8=5
5	3	d+I_2=5+(-6)=-1
6	4	d+I_1=-1+8=7
7	4	d+I_2=7+(-6)=1
8	5	



Differentiate between DDA Algorithm and Bresenham's Line Algorithm:

DDA Algorithm	Bresenham's Line Algorithm
1. DDA Algorithm use floating point, i.e., Real Arithmetic.	1. Bresenham's Line Algorithm use fixed point, i.e., Integer Arithmetic
2. DDA Algorithms uses multiplication & division its operation	2.Bresenham's Line Algorithm uses only subtraction and addition its operation
3. DDA Algorithm is slowly than Bresenham's Line Algorithm in line drawing because it uses real arithmetic (Floating Point operation)	3. Bresenham's Algorithm is faster than DDA Algorithm in line because it involves only addition & subtraction in its calculation and uses only integer arithmetic.
4. DDA Algorithm is not accurate and efficient as Bresenham's Line Algorithm.	4. Bresenham's Line Algorithm is more accurate and efficient at DDA Algorithm.
5.DDA Algorithm can draw circle and curves but are not accurate as Bresenham's Line Algorithm	5. Bresenham's Line Algorithm can draw circle and curves with more accurate than DDA Algorithm.

Algorithm to draw circles and ellipses:

This algorithm is based on the parametric form of the circle equation. For more see Parametric equation of a circle.

Recall that this looks like

$x = h + r\cos\theta$
$y = k + r\sin\theta$

where r is the radius of the circle, and h,k are the coordinates of the center.

What these equations do is generate the x,y coordinates of a point on the circle given an angle θ (theta). The algorithm starts with theta at zero, and then loops adding an increment to theta each time round the loop. It draws straight line segments between these successive points on the circle. The circle is thus drawn as a series of straight lines. If the increment is small enough, the result looks like a circle to the eye, even though in strict mathematical terms is is not.

The algorithm:

Below is the algorithm in pseudocode showing the basic idea.

```
theta = 0;  // angle that will be increased each loop
h = 12      // x coordinate of circle center
k = 10      // y coordinate of circle center
step = 15;  // amount to add to theta each time (degrees)
```

```
repeat until theta >= 360;
  { x = h + r*cos(theta)
    y = k + r*sin(theta)
    draw a line to x,y
    add step to theta
  }
```

The decision about how big to make the step size is a tradeoff. If it is very small, many lines will be drawn for a smooth circle, but there will be more computer time used to do it. If it is too large the circle will not be smooth and be visually ugly.

Example:

Below is the algorithm written in JavaScript using the HTML5 canvas element to draw into.

```
var ctx = canvas.getContext("2d");

 var step = 2*Math.PI/20;  // see note 1
 var h = 150;
 var k = 150;
 var r = 50;
 ctx.beginPath();  //tell canvas to start a set of lines

 for(var theta=0;  theta < 2*Math.PI;  theta+=step)
  { var x = h + r*Math.cos(theta);
    var y = k - r*Math.sin(theta);    //note 2.
    ctx.lineTo(x,y);
  }
 ctx.closePath();     //close the end to the start point
   ctx.stroke();        //actually draw the accumulated lines
```

Like most graphics systems, the canvas element differs from the usual mathematical coordinate plane:

1. The origin is in the top left corner. The code above compensates by assuming that h, and k are actually relative to the top left.
2. The y axis is inverted. Positive y is down the screen, not up. To correct for this, the k variable (the y coordinate of the center) must be positive to place the center some way down the screen. Also, the y calculation has to subtract the sin(x) term instead of add. Marked in the code as Note 1.

Note 2. The step size is set to an exact division of 2π to avoid gaps or over-runs in the circle. This code divides the circle into exactly 20 segments. Note too that as in most computer languages, the trig functions operate in radians, not degrees.

Ellipses:

The circles can be made into ellipses by simply "squashing" them in one direction or the other. For an ellipse that is wider than it is tall, be divide the y coordinate by a number (here 2) to reduce its height. The two inner calculations would then look like:

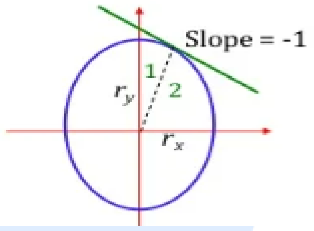
```
      var x = h +       r*Math.cos(theta) ;
       var y = k - 0.5 * r*Math.sin(theta) ;
Changing the 0.5 will alter how much the vertical size is squashed. Multiply
the y term this way will make an ellipse that is tall and narrow.
```

Ellipse Algorithms

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

- Decision parameter:

$$f_{ellipse}(x, y) = \begin{cases} < 0 & \text{if } (x, y) \text{ is inside the ellipse} \\ = 0 & \text{if } (x, y) \text{ is on the ellipse} \\ > 0 & \text{if } (x, y) \text{ is outside the ellipse} \end{cases}$$



$$Slope = \frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y}$$

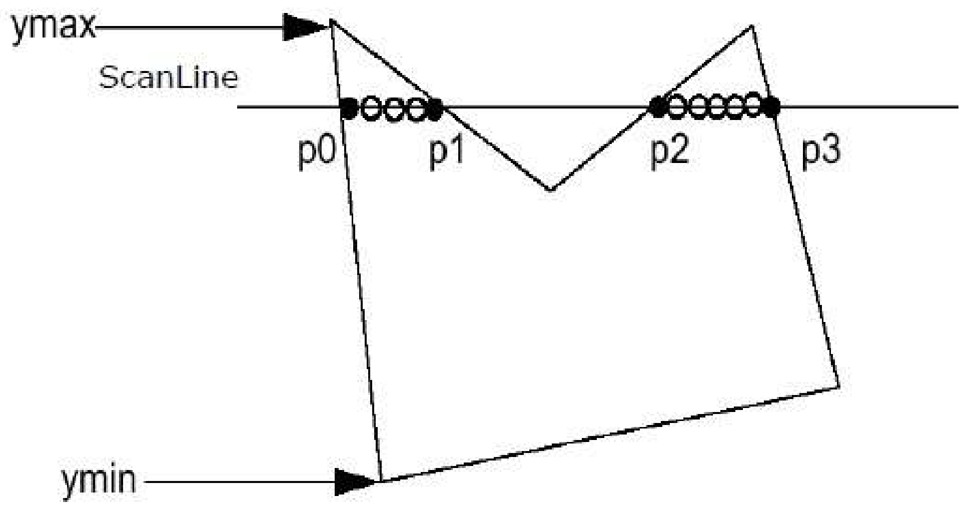
Scan-converting Polygon filling:

Polygon: Polygon is an ordered list of vertices as shown in the following figure. For filling polygons with particular colors, you need to determine the pixels falling on the border of the polygon and those which fall inside the polygon. In this chapter, we will see how we can fill polygons using different techniques.

Scan Line Algorithm:

This algorithm works by intersecting scanline with polygon edges and fills the polygon between pairs of intersections. The following steps depict how this algorithm works.

Step 1 − Find out the Ymin and Ymax from the given polygon.



Step 2 − ScanLine intersects with each edge of the polygon from Ymin to Ymax. Name each intersection point of the polygon. As per the figure shown above, they are named as p0, p1, p2, p3.

Step 3 − Sort the intersection point in the increasing order of X coordinate i.e. p0,p1$p0,p1$, p1,p2$p1,p2$, and p2,p3$p2,p3$.

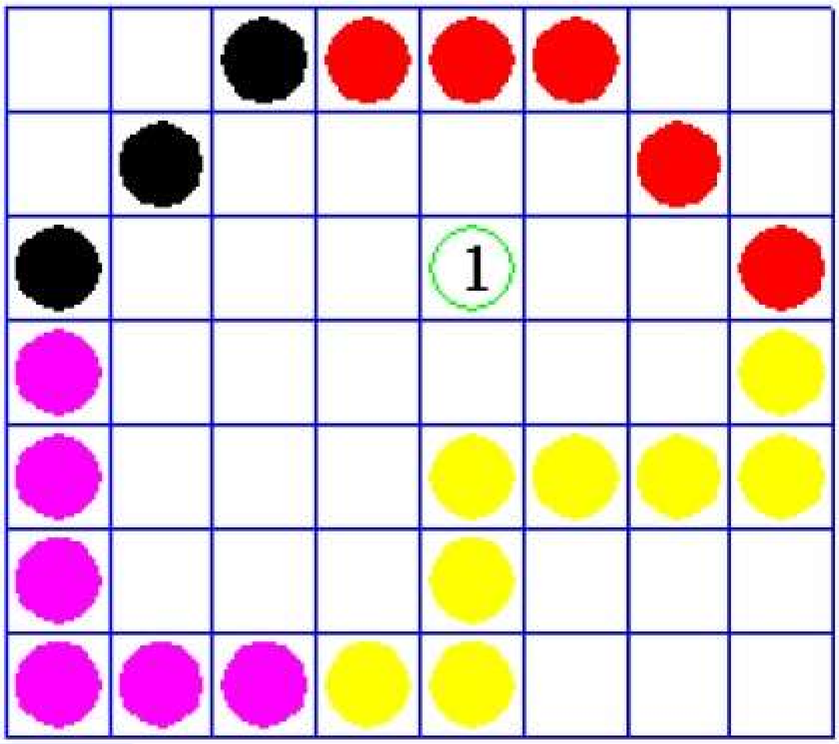
Step 4 − Fill all those pair of coordinates that are inside polygons and ignore the alternate pairs.

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.



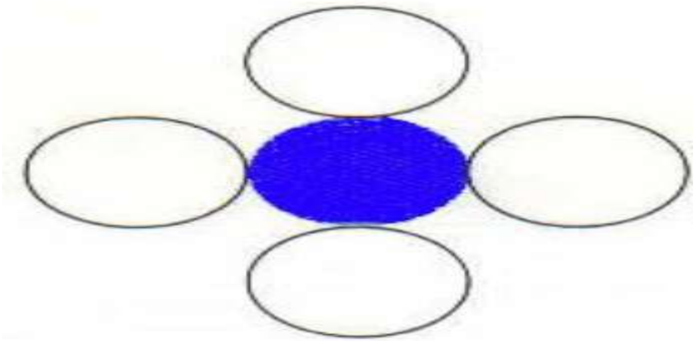
Boundary Fill Algorithm(Area Fill Algorithm):

The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The color of the boundary and the color that we fill should be different for this algorithm to work.

In this algorithm, we assume that color of the boundary is same for the entire object. The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

4-Connected Polygon

In this technique 4-connected pixels are used as shown in the figure. We are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different color.



Algorithm

Step 1 − Initialize the value of seed point seedx,seedy*seedx,seedy*, fcolor and dcol.

Step 2 − Define the boundary values of the polygon.

Step 3 − Check if the current seed point is of default color, then repeat the steps 4 and 5 till the boundary pixels reached.

```
If getpixel(x, y) = dcol then repeat step 4 and 5
```

Step 4 − Change the default color with the fill color at the seed point.

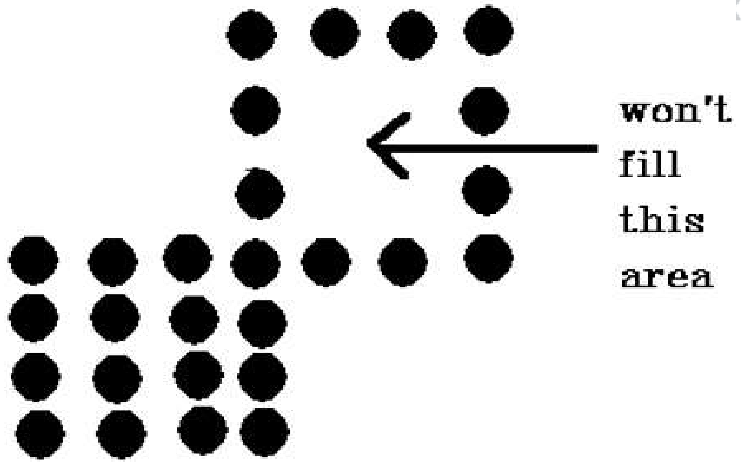
```
setPixel(seedx, seedy, fcol)
```

Step 5 − Recursively follow the procedure with four neighborhood points.

```
FloodFill (seedx – 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy - 1, fcol, dcol)
FloodFill (seedx – 1, seedy + 1, fcol, dcol)
```

Step 6 − Exit

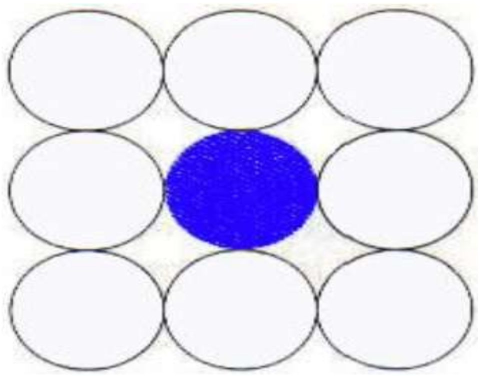
There is a problem with this technique. Consider the case as shown below where we tried to fill the entire region. Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.



8-Connected Polygon

In this technique 8-connected pixels are used as shown in the figure. We are putting pixels above, below, right and left side of the current pixels as we were doing in 4-connected technique.

In addition to this, we are also putting pixels in diagonals so that entire area of the current pixel is covered. This process will continue until we find a boundary with different color.



Algorithm

Step 1 − Initialize the value of seed point seedx,seedy*seedx,seedy*, fcolor and dcol.

Step 2 − Define the boundary values of the polygon.

Step 3 − Check if the current seed point is of default color then repeat the steps 4 and 5 till the boundary pixels reached

```
If getpixel(x,y) = dcol then repeat step 4 and 5
```

Step 4 − Change the default color with the fill color at the seed point.

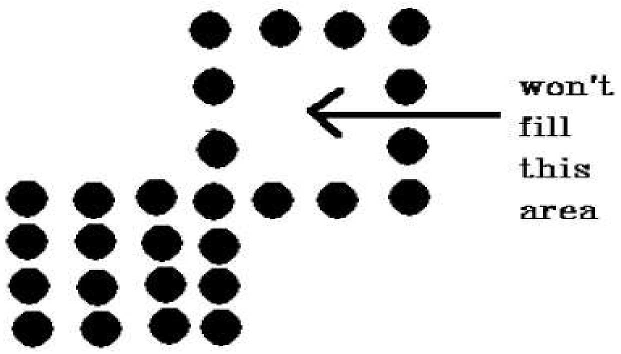
```
setPixel(seedx, seedy, fcol)
```

Step 5 − Recursively follow the procedure with four neighbourhood points

```
FloodFill (seedx – 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy - 1, fcol, dcol)
FloodFill (seedx, seedy + 1, fcol, dcol)
FloodFill (seedx – 1, seedy + 1, fcol, dcol)
FloodFill (seedx + 1, seedy + 1, fcol, dcol)
FloodFill (seedx + 1, seedy - 1, fcol, dcol)
FloodFill (seedx – 1, seedy - 1, fcol, dcol)
```

Step 6 − Exit

The 4-connected pixel technique failed to fill the area as marked in the following figure which won’t happen with the 8-connected technique.



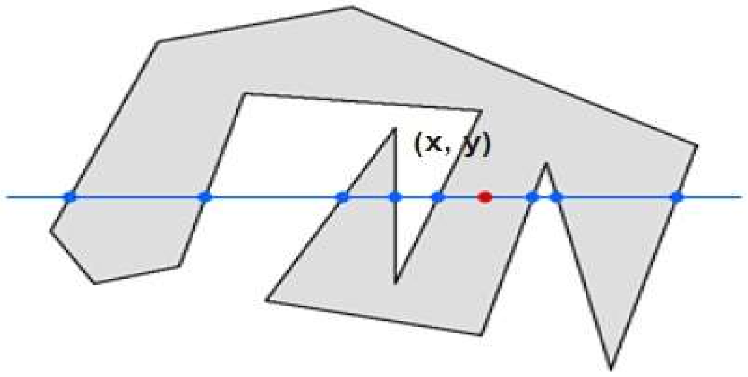
Inside-outside Test

This method is also known as **counting number method**. While filling an object, we often need to identify whether particular point is inside the object or outside it. There are two methods by which we can identify whether particular point is inside an object or outside.

- Odd-Even Rule
- Nonzero winding number rule

Odd-Even Rule

In this technique, we will count the edge crossing along the line from any point x,y*x,y* to infinity. If the number of interactions is odd, then the point x,y*x,y* is an interior point; and if the number of interactions is even, then the point x,y*x,y* is an exterior point. The following example depicts this concept.

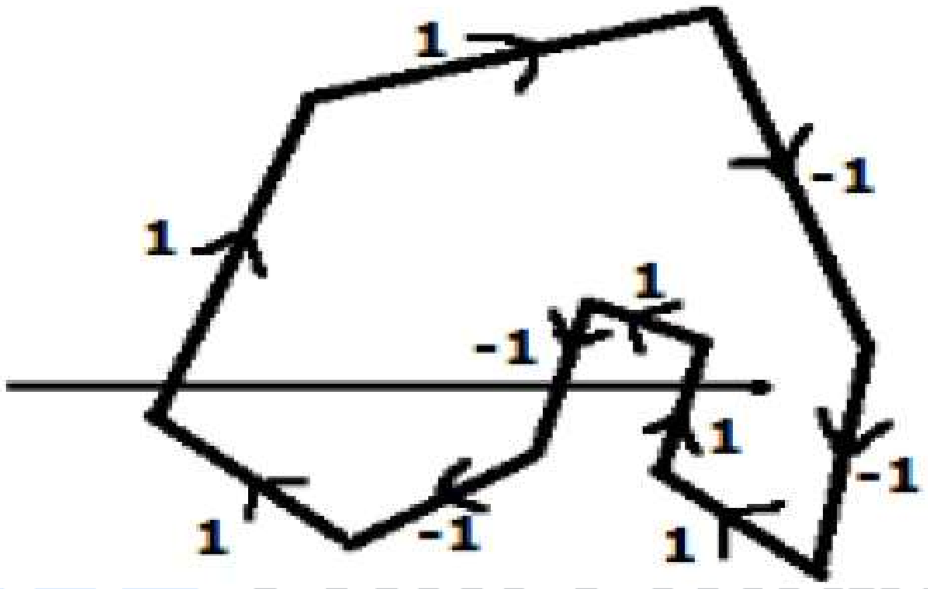


From the above figure, we can see that from the point x,y*x,y*, the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.

Nonzero Winding Number Rule

This method is also used with the simple polygons to test the given point is interior or not. It can be simply understood with the help of a pin and a rubber band. Fix up the pin on one of the edges of the polygon and tie-up the rubber band in it and then stretch the rubber band along the edges of the polygon.

When all the edges of the polygon are covered by the rubber band, check out the pin which has been fixed up at the point to be test. If we find at least one wind at the point consider it within the polygon, else we can say that the point is not inside the polygon.



In another alternative method, give directions to all the edges of the polygon. Draw a scan line from the point to be test towards the left most of X direction.

- Give the value 1 to all the edges which are going to upward direction and all other -1 as direction values.
- Check the edge direction values from which the scan line is passing and sum up them.
- If the total sum of this direction value is non-zero, then this point to be tested is an **interior point,** otherwise it is an **exterior point**.
- In the above figure, we sum up the direction values from which the scan line is passing then the total is 1 – 1 + 1 = 1; which is non-zero. So, the point is said to be an interior point.

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.

Homogenous Coordinates

To perform a sequence of transformation such as translation followed by rotation and scaling, we need to follow a sequential process −

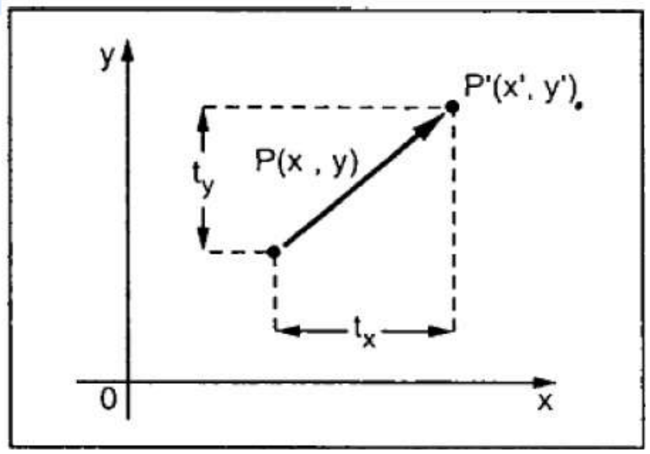
- Translate the coordinates,
- Rotate the translated coordinates, and then
- Scale the rotated coordinates to complete the composite transformation.

To shorten this process, we have to use 3×3 transformation matrix instead of 2×2 transformation matrix. To convert a 2×2 matrix to 3×3 matrix, we have to add an extra dummy coordinate W.

In this way, we can represent the point by 3 numbers instead of 2 numbers, which is called **Homogenous Coordinate** system. In this system, we can represent all the transformation equations in matrix multiplication. Any Cartesian point PX,Y*X,Y* can be converted to homogenous coordinates by P' (X_h, Y_h, h).

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*X,Y* to get the new coordinate X′,Y′*X′,Y′*.



From the above figure, you can write that −

$$\mathbf{X' = X + t_x}$$

$$\mathbf{Y' = Y + t_y}$$

The pair (t_x, t_y) is called the translation vector or shift vector. The above equations can also be represented using the column vectors.

P=[X][Y]*P=[X][Y]* p' = [X′][Y′]*[X′][Y′]*T = [t_x][t_y]*[tx][ty]*

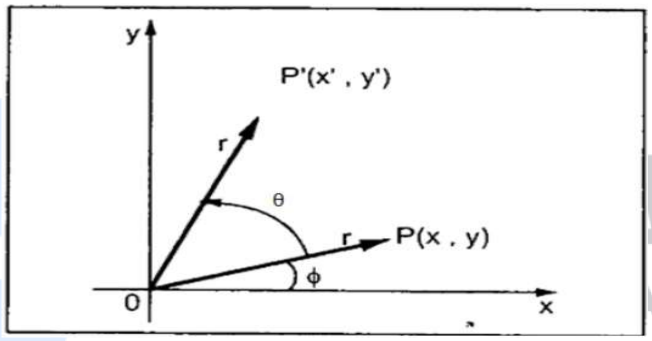
We can write it as −

P' = P + T

Rotation

In rotation, we rotate the object at particular angle θ theta*theta* from its origin. From the following figure, we can see that the point PX,Y*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′*X′,Y′*.



Using standard trigonometric the original coordinate of point PX,Y*X,Y* can be represented as −

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

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

Same way we can represent the point P' X′,Y′*X′,Y′* as −

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

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

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

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

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

Representing the above equation in matrix form,

$$[X′Y′]=[XY][cosθ−sinθsinθcosθ]OR\textit{[X′Y′]=[XY][cosϑsinϑ−sinϑcosϑ]OR}$$

P’ = P . R

Where R is the rotation matrix

$$R=[cosθ−sinθsinθcosθ]\textit{R=[cosϑsinϑ−sinϑcosϑ]}$$

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 −

$$R=[cos(−θ)−sin(−θ)sin(−θ)cos(−θ)]\textit{R=[cos(−ϑ)sin(−ϑ)−sin(−ϑ)cos(−ϑ)]}$$

$$=[cosθsinθ−sinθcosθ](\quad cos(−θ)=cosθandsin(−θ)=−sinθ)\textit{=[cosϑ−sinϑsinϑcosϑ](\quad cos(−ϑ)=cosϑandsin(−ϑ)=−sinϑ)}$$

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*X,Y*, the scaling factors are (S_X, S_Y), and the produced coordinates are X′,Y′*X′,Y′*. This can be mathematically represented as shown below −

$$\mathbf{X' = X \,.\, S_X \text{ and } Y' = Y \,.\, S_Y}$$

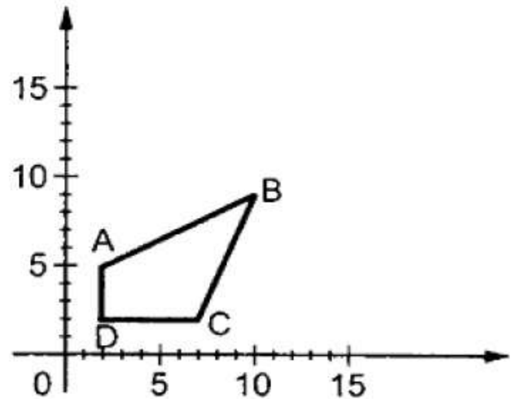
The scaling factor S_X, S_Y scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below −

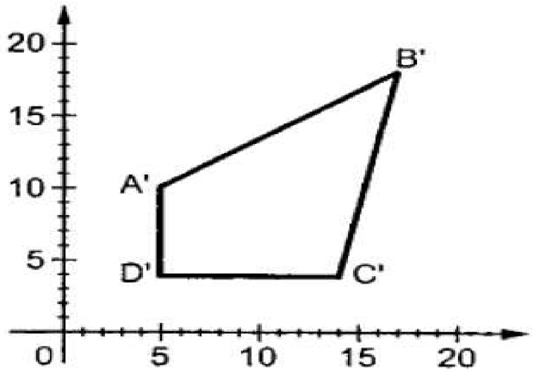
$$(X′Y′)=(XY)[S_x00S_Y]\textit{(X′Y′)=(XY)[Sx00Sy]}$$

OR

P’ = P . S

Where S is the scaling matrix. The scaling process is shown in the following figure.



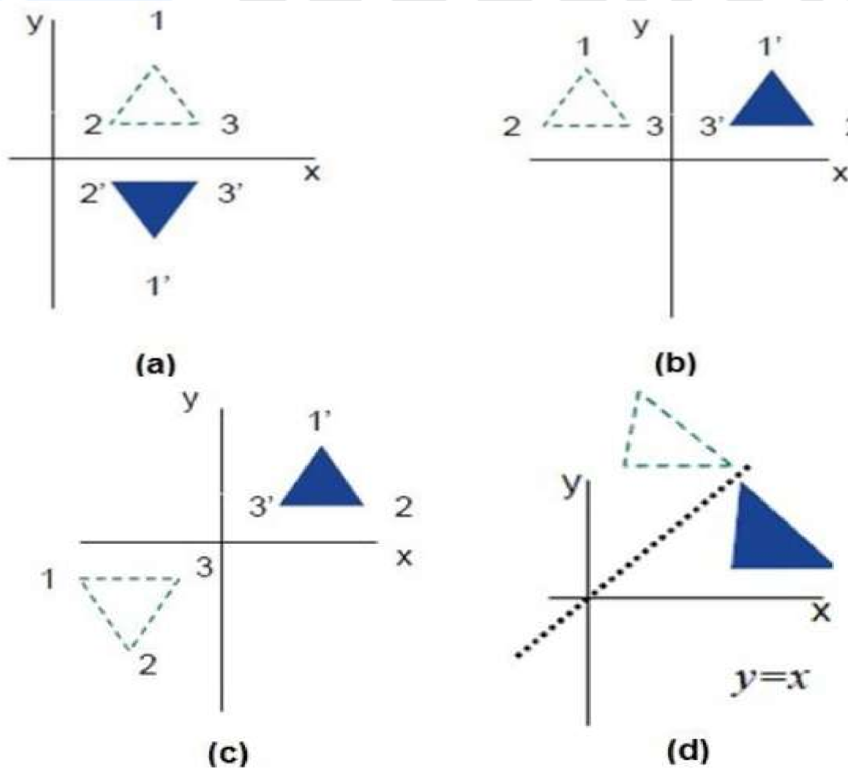


If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

Reflection

Reflection is the mirror image of original object. In other words, we can say that it is a rotation operation with 180°. In reflection transformation, the size of the object does not change.

The following figures show reflections with respect to X and Y axes, and about the origin respectively.

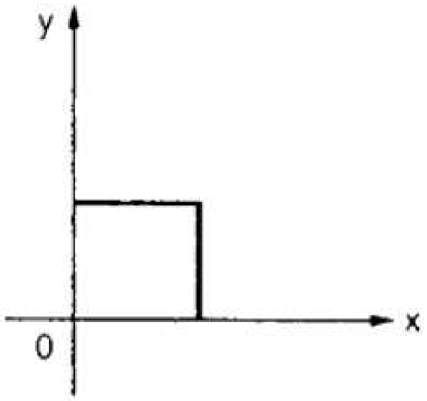


Shear

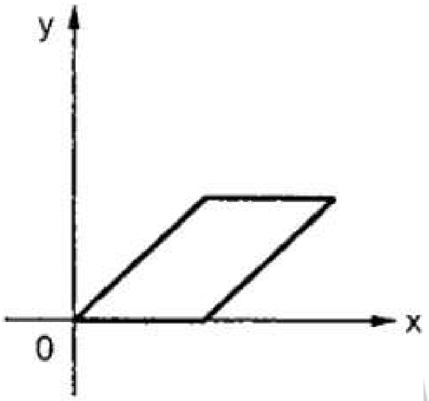
A transformation that slants the shape of an object is called the shear transformation. There are two shear transformations **X-Shear** and **Y-Shear**. One shifts X coordinates values and other shifts Y coordinate values. However; in both the cases only one coordinate changes its coordinates and other preserves its values. Shearing is also termed as **Skewing**.

X-Shear

The X-Shear preserves the Y coordinate and changes are made to X coordinates, which causes the vertical lines to tilt right or left as shown in below figure.

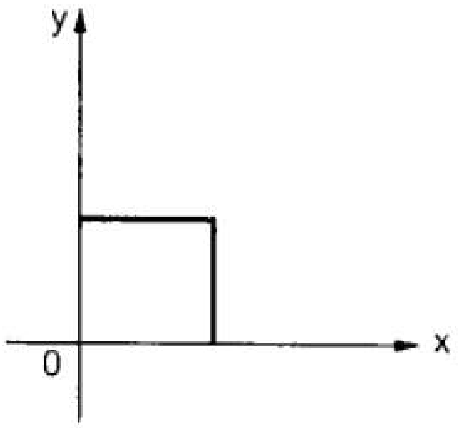


(a) Original object

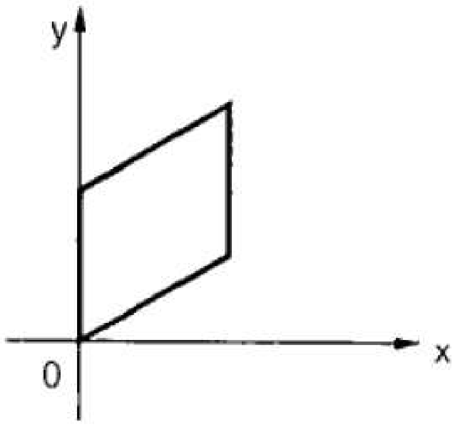


(b) Object after x shear

The Y-Shear preserves the X coordinates and changes the Y coordinates which causes the horizontal lines to transform into lines which slopes up or down as shown in the following figure.



(a) Original object



(b) Object after y shear

Composite Transformation

If a transformation of the plane T1 is followed by a second plane transformation T2, then the result itself may be represented by a single transformation T which is the composition of T1 and T2 taken in that order. This is written as T = T1·T2.

Composite transformation can be achieved by concatenation of transformation matrices to obtain a combined transformation matrix.

A combined matrix −

[T][X] = [X] [T1] [T2] [T3] [T4] …. [Tn]

Where [Ti] is any combination of

- Translation
- Scaling
- Shearing
- Rotation
- Reflection

The change in the order of transformation would lead to different results, as in general matrix multiplication is not cumulative, that is [A] . [B] ≠ [B] . [A] and the order of multiplication. The basic purpose of composing transformations is to gain efficiency by applying a single composed transformation to a point, rather than applying a series of transformation, one after another.

For example, to rotate an object about an arbitrary point (X_p, Y_p), we have to carry out three steps −

- Translate point (X_p, Y_p) to the origin.
- Rotate it about the origin.
- Finally, translate the center of rotation back where it belonged.

3D – Transformation:

In very general terms a 3D model is a mathematical representation of a physical entity that occupies space. In more practical terms, a 3D model is made of a description of its shape and a description of its color appearance.3-D Transformation is the process of manipulating the view of a three-D object with respect to its original position by modifying its physical attributes through various methods of transformation like Translation, Scaling, Rotation, Shear, etc.

Properties of 3-D Transformation:

- Lines are preserved,
- Parallelism is preserved,

- Proportional distances are preserved.

One main categorization of a 3D object’s representation can be done by considering whether the surface or the volume of the object is represented:

Boundary-based: the surface of the 3D object is represented. This representation is also called b-rep. Polygon meshes, implicit surfaces, and parametric surfaces, which we will describe in the following, are common representations of this type.

Volume-based: the volume of the 3D object is represented. Voxels and Constructive Solid Geometry (CSG) Are commonly used to represent volumetric data.

Rotation

3D rotation is not same as 2D rotation. In 3D rotation, we have to specify the angle of rotation along with the axis of rotation. We can perform 3D rotation about X, Y, and Z axes.

Scaling

You can change the size of an object using scaling transformation. 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.

In 3D scaling operation, three coordinates are used. Let us assume that the original coordinates are X,Y,ZX,Y,Z, scaling factors are (S_X,S_Y,S_Z)(SX,SY,Sz) respectively, and the produced coordinates are X′,Y′,Z′$X′,Y′,Z′$.

Shear

A transformation that slants the shape of an object is called the **shear transformation**. Like in 2D shear, we can shear an object along the X-axis, Y-axis, or Z-axis in 3D.

Transformation Matrices

Transformation matrix is a basic tool for transformation. A matrix with n x m dimensions is multiplied with the coordinate of objects. Usually 3 x 3 or 4 x 4 matrices are used for transformation. For example, consider the following matrix for various operation.