DDA Line 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 (x_i, y_i)

The line of equation for step i

$$y_i = m_{i+b} \dots\dots\dots\dots\dots (1)$$

Next value will be

$$y_{i+1} = m_{xi+1} + b \dots\dots\dots\dots (2)$$

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

$$y_{i+1} - y_i = \Delta y \dots\dots (3) \qquad x_{i+1} - x_i = \Delta x \dots\dots (4)$$

$$y_{i+1} = y_i + \Delta y$$

$$\Delta y = m\Delta x$$

$$y_{i+1} = y_i + m\Delta x$$

$$\Delta x = {}^{\Delta y}/_{m}$$

$$x_{i+1} = x_i + \Delta x$$

$$x_{i+1} = x_i + {}^{\Delta y}/_{m}$$

Case1: When |M| < 1 then

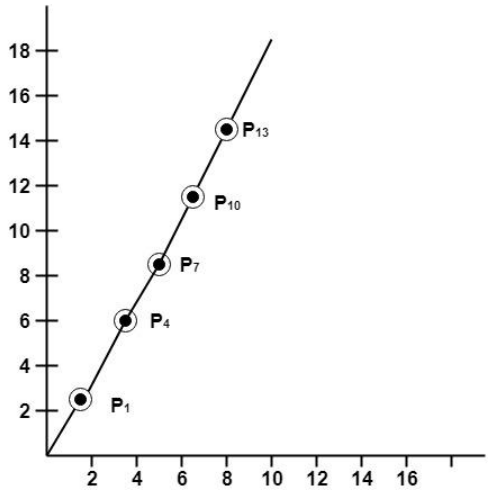
$$x_{i+1} = 1 + x_i; \qquad y_{i+1} = m + y_i$$

Case2: When |M| = 1 then

$$x_{i+1} = 1 + x_i; \qquad y_{i+1} = 1 + y_i$$

Case3: When |M| > 1 then

$$x_{i+1} = {}^{1}/_{m} + x_i; \qquad y_{i+1} = 1 + y_i$$



Algorithms:

Given-

- Starting coordinates = (X_0, Y_0)
- Ending coordinates = (X_n, Y_n)

The points generation using DDA Algorithm involves the following steps-

Step-01:

Calculate ΔX, ΔY and M from the given input.

These parameters are calculated as-

- $\Delta X = X_n - X_0$
- $\Delta Y = Y_n - Y_0$
- $M = \Delta Y / \Delta X$

Step-02:

Find the number of steps or points in between the starting and ending coordinates.

if (absolute (ΔX) > absolute (ΔY))

Steps = absolute (ΔX);

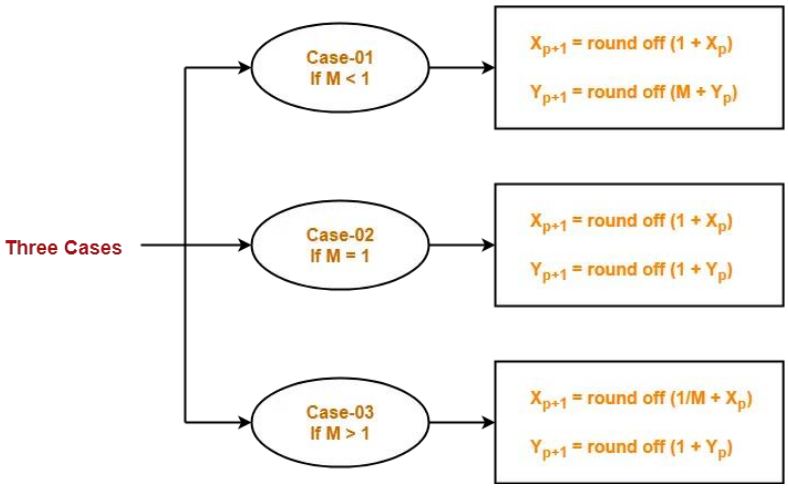
else

Steps = absolute (ΔY);

Step-03:

Suppose the current point is (X_p, Y_p) and the next point is (X_{p+1}, Y_{p+1}).

Find the next point by following the below three cases-



Step-04:

Keep repeating Step-03 until the end point is reached or the number of generated new points (including the starting and ending points) equals to the steps count.

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.

The method works as follows:

Assume a pixel $P_1'(x_1',y_1')$,then select subsequent pixels as we work our may to the night, one pixel position at a time in the horizontal direction toward $P_2'(x_2',y_2')$.

Once a pixel in choose at any step

The next pixel is

1. Either the one to its right (lower-bound for the line)
2. One top its right and up (upper-bound for the line)

The line is best approximated by those pixels that fall the least distance from the path between P_1',P_2'.

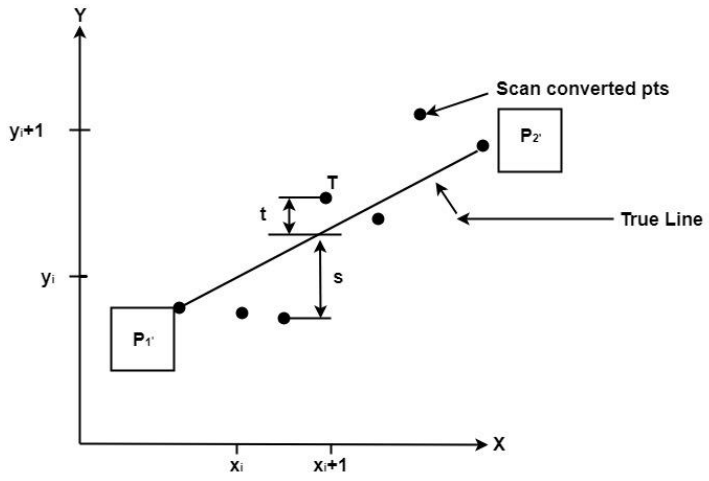


Fig: Scan Converting a line.

To chooses the next one between the bottom pixel S and top pixel T.

If S is chosen

We have $x_{i+1} = x_i+1$ and $y_{i+1} = y_i$

If T is chosen

We have $x_{i+1} = x_i+1$ and $y_{i+1} = y_i+1$

The actual y coordinates of the line at $x = x_{i+1}$ is

$y = mx_{i+1} + b$

$$y = m(x_i + 1) + b$$

The distance from S to the actual line in y direction $s = y - y_i$

The distance from T to the actual line in y direction $t = (y_i+1) – y$

Now consider the difference between these 2-distance values $s - t$

When $(s - t) < 0 \Longrightarrow s < t$

The closest pixel is S

When $(s - t) \geq 0 \Longrightarrow s < t$

The closest pixel is T

This difference is $s\text{-}t = (y - y_i) - [(y_i+1) - y] = = 2y - 2y_i - 1$

$$\boxed{s - t = 2m(x_i + 1) + 2b - 2y_i - 1}$$ [Putting the value of (1)]

Substituting m by $\frac{\Delta y}{\Delta x}$ and introducing decision variable

$d_i = \Delta x\,(s - t)$

$d_i = \Delta x\,(2\frac{\Delta y}{\Delta x}(x_i+1) + 2b - 2y_i - 1)$

$= 2\Delta x y_i - 2\Delta y - 1\Delta x.2b - 2y_i\Delta x - \Delta x$

$d_i = 2\Delta y.x_i - 2\Delta x.y_i + c$

Where $c = 2\Delta y + \Delta x\,(2b - 1)$

We can write the decision variable d_{i+1} for the next slip on

$d_{i+1} = 2\Delta y.x_{i+1} - 2\Delta x.y_{i+1} + c$

$d_{i+1} - d_i = 2\Delta y.(x_{i+1}-x_i) - 2\Delta x(y_{i+1}-y_i)$

Since $x\_(i+1) = x_i+1$, we have

$d_{i+1} + d_i = 2\Delta y.(x_i+1-x_i) - 2\Delta x\,(y_{i+1} - y_i)$

Special Cases,

If chosen pixel is at the top pixel T (i.e., $d_i \geq 0$) $\Longrightarrow y_{i+1} = y_i + 1$

$d_{i+1} = d_i + 2\Delta y - 2\Delta x$

If chosen pixel is at the bottom pixel T (i.e., $d_i < 0$) $\Longrightarrow y_{i+1} = y_i$

$d_{i+1} = d_i + 2\Delta y$

Finally, we calculate d_1

$d_1 = \Delta x\,[2m(x_1+1) + 2b - 2y_1 - 1]$

$d_1 = \Delta x\,[2(mx_1+b-y_1) + 2m - 1]$

Since $mx_1+b - y_i = 0$ and $m = \frac{\Delta y}{\Delta x}$, we have

$d_1 = 2\Delta y - \Delta x$

Algorithm's:

Given-

- Starting coordinates = (X_0, Y_0)
- Ending coordinates = (X_n, Y_n)

The points generation using Bresenham Line Drawing Algorithm involves the following steps-

Step-01:

Calculate ΔX and ΔY from the given input.

These parameters are calculated as-

- $\Delta X = X_n - X_0$
- $\Delta Y = Y_n - Y_0$

Step-02:

Calculate the decision parameter P_k.

It is calculated as-

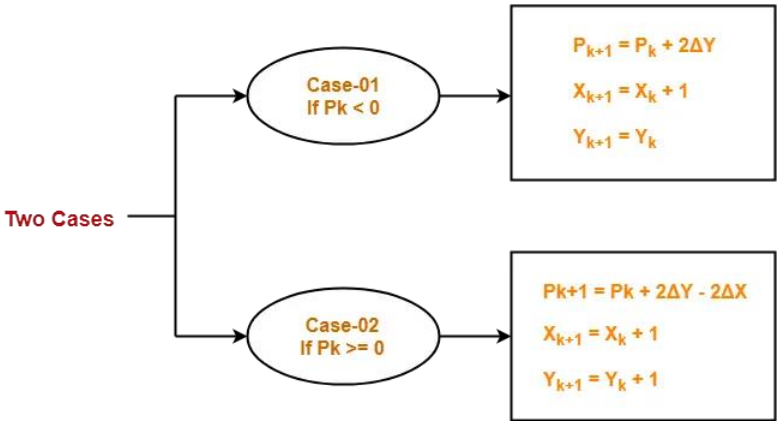
$P_k = 2\Delta Y - \Delta X$

Step-03:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).

Find the next point depending on the value of decision parameter P_k.

Follow the below two cases-

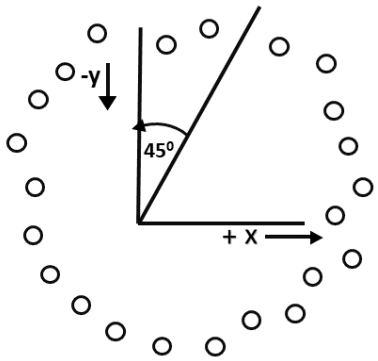


Step-04:

Keep repeating Step-03 until the end point is reached or number of iterations equals to $(\Delta X-1)$ times.

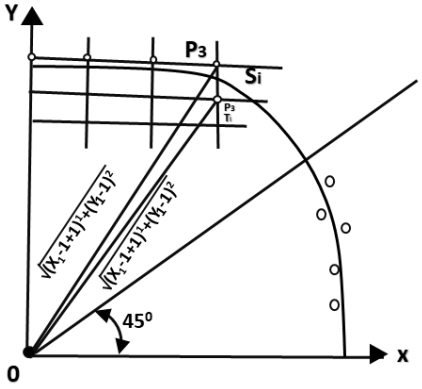
Bresenham's Circle Algorithm:

Scan-Converting a circle using Bresenham's algorithm works as follows: Points are generated from 90° to 45°, moves will be made only in the +x & -y directions as shown in fig:



The best approximation of the true circle will be described by those pixels in the raster that falls the least distance from the true circle. We want to generate the points from

90° to 45°. Assume that the last scan-converted pixel is P_1 as shown in fig. Each new point closest to the true circle can be found by taking either of two actions.



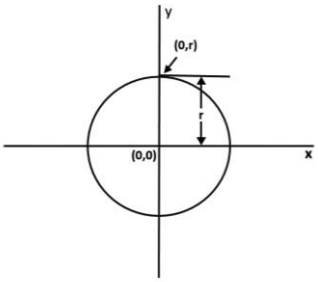
1. Move in the x-direction one unit or
2. Move in the x- direction one unit & move in the negative y-direction one unit.

Let D (S_i) is the distance from the origin to the true circle squared minus the distance to point P_3 squared. D (T_i) is the distance from the origin to the true circle squared minus the distance to point P_2 squared. Therefore, the following expressions arise.

$$D(S_i) = (x_{i-1}+1)^2 + y_{i-1}^2 - r^2$$

$$D(T_i) = (x_{i-1}+1)^2 + (y_{i-1}-1)^2 - r^2$$

Since D (S_i) will always be +ve & D (T_i) will always be -ve, a decision variable d may be defined as follows:



$$d_i = D(S_i) + D(T_i)$$

Therefore,

$$d_i = (x_{i-1}+1)^2 + y_{i-1}^2 - r^2 + (x_{i-1}+1)^2 + (y_{i-1}-1)^2 - r^2$$

From this equation, we can drive initial values of d_i as

If it is assumed that the circle is centered at the origin, then at the first step x = 0 & y = r.

Therefore, $d_i = (0+1)^2 + r^2 - r^2 + (0+1)^2 + (r-1)^2 - r^2 = 1 + 1 + r^2 - 2r + 1 - r^2 = 3 - 2r$

Thereafter, if d_i < 0, then only x is incremented.

$x_{i+1} = x_{i+1}$ $\quad d_{i+1} = d_i + 4x_i + 6$

& if $d_i \geq 0$, then x & y are incremented

$x_{i+1} = x_{i+1}$ $\quad y_{i+1} = y_i + 1$

$d_{i+1} = d_i + 4(x_i - y_i) + 10$

Algorithms:

Given-

- Centre point of Circle = (X_0, Y_0)
- Radius of Circle = R

The points generation using Bresenham Circle Drawing Algorithm involves the following steps-

Step-01:

Assign the starting point coordinates (X_0, Y_0) as-

- $X_0 = 0$
- $Y_0 = R$

Step-02:

Calculate the value of initial decision parameter P_0 as-

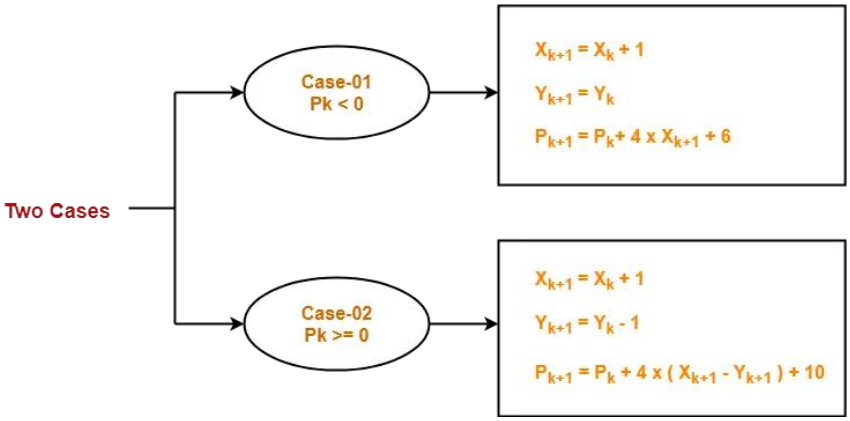
$P_0 = 3 - 2 \times R$

Step-03:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).

Find the next point of the first octant depending on the value of decision parameter P_k.

Follow the below two cases-



Step-04:

If the given centre point (X_0, Y_0) is not (0, 0), then do the following and plot the point-

- $X_{plot} = X_c + X_0$
- $Y_{plot} = Y_c + Y_0$

Here, (X_c, Y_c) denotes the current value of X and Y coordinates.

Step-05:

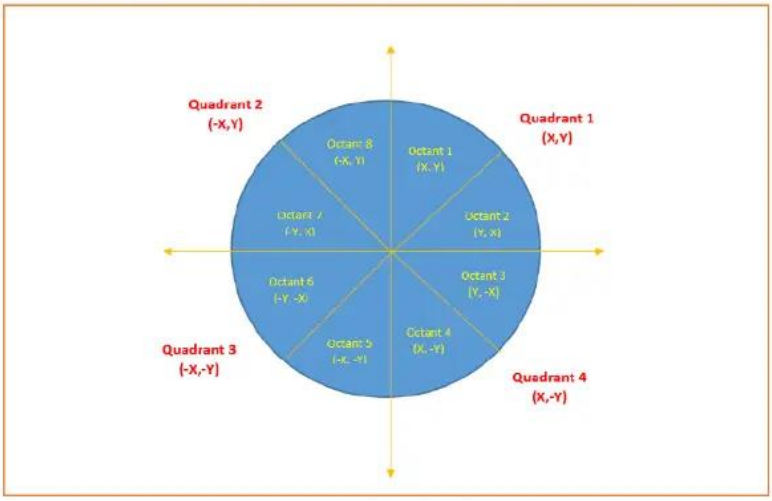
Keep repeating Step-03 and Step-04 until $X_{plot} => Y_{plot}$.

Step-06:

Step-05 generates all the points for one octant.

To find the points for other seven octants, follow the eight-symmetry property of circle.

This is depicted by the following figure-



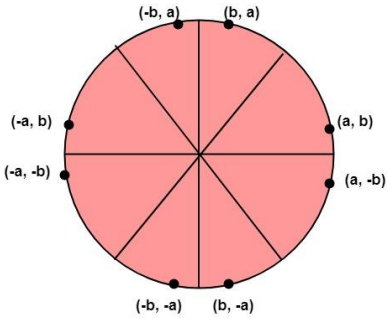
MidPoint Circle Algorithm

It is based on the following function for testing the spatial relationship between the arbitrary point (x, y) and a circle of radius r centered at the origin:

Now, consider the coordinates of the point halfway between pixel T and pixel S

This is called midpoint $(x_{i+1}, y_i - \frac{1}{2})$ and we use it to define a decision parameter:

$$P_i = f(x_{i+1}, y_i - \frac{1}{2}) = (x_{i+1})^2 + (y_i - \frac{1}{2})^2 - r^2 \text{ .....equation 2}$$



If P_i is -ve $\Longrightarrow$ midpoint is inside the circle and we choose pixel T

If P_i is +ve $\Longrightarrow$ midpoint is outside the circle (or on the circle) and we choose pixel S.

The decision parameter for the next step is:

$$P_{i+1} = (x_{i+1}+1)^2 + (y_{i+1} - \frac{1}{2})^2 - r^2 \text{.....equation 3}$$

Since $x_{i+1} = x_{i+1}$, we have

$$P_{i+1} - P_i = ((x_i + 1) + 1)^2 - (x_i + 1)^2 + (y_{i+1} - \tfrac{1}{2})^2 - (y_i - \tfrac{1}{2})^2$$

$$= x_i^2 + 4 + 4x_i - x_i^2 + 1 - 2x_i + y_{i+1}^2 + \tfrac{1}{4} - y_{i+1} - y_i^2 - \tfrac{1}{4} - y_i$$

$$= 2(xi+1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i)$$

$$P_{i+1} = P_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i) \text{.....equation 4}$$

If pixel T is chosen $\Longrightarrow P_i < 0$

We have $y_{i+1} = y_i$

If pixel S is chosen $\Longrightarrow P_i \geq 0$

We have $y_{i+1} = y_i - 1$

Thus, $$P_{i+1} = \begin{bmatrix} P_i + 2(x_i + 1) + 1, & \text{if } P_i < 0 \\ P_i + 2(x_i + 1) + 1 - 2(y_i - 1), & \text{if } P_i \geq 0 \end{bmatrix} \quad \text{.....equation 5}$$

We can continue to simplify this in n terms of (x_i, y_i) and get

$$P_{i+1} = \begin{bmatrix} P_i + 2x_i + 3, & \text{if } P_i < 0 \\ P_i + 2(x_i - y_i) + 5, & \text{if } P_i \geq 0 \end{bmatrix} \quad \text{.....equation 6}$$

Now, initial value of P_i (0, r) from equation 2

$$P_1 = (0 + 1)^2 + (r - \tfrac{1}{2})^2 - r^2$$

$$= 1 + \tfrac{1}{4} - r^2 = \tfrac{5}{4} - r$$

We can put $\frac{5}{4} \cong 1$

∴r is an integer

So, $P_1 = 1 - r$

Algorithms:

Given-

- Centre point of Circle = (X_0, Y_0)
- Radius of Circle = R

The points generation using Mid-Point Circle Drawing Algorithm involves the following steps-

Step-01:

Assign the starting point coordinates (X_0, Y_0) as-

- $X_0 = 0$
- $Y_0 = R$

Step-02:

Calculate the value of initial decision parameter P_0 as-

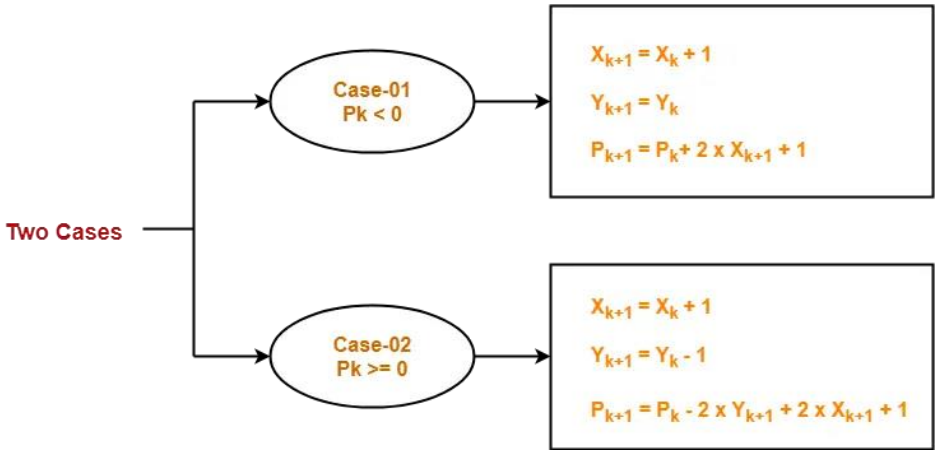
$P_0 = 1 - R$

Step-03:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).

Find the next point of the first octant depending on the value of decision parameter P_k.

Follow the below two cases-



Step-04:

If the given centre point (X_0, Y_0) is not (0, 0), then do the following and plot the point-

- $X_{plot} = X_c + X_0$
- $Y_{plot} = Y_c + Y_0$

Here, (X_c, Y_c) denotes the current value of X and Y coordinates.

Step-05:

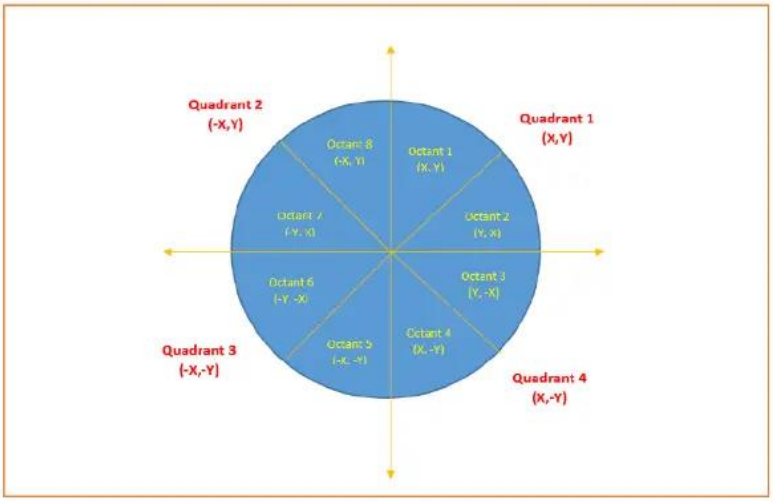
Keep repeating Step-03 and Step-04 until $X_{plot} >= Y_{plot}$.

Step-06:

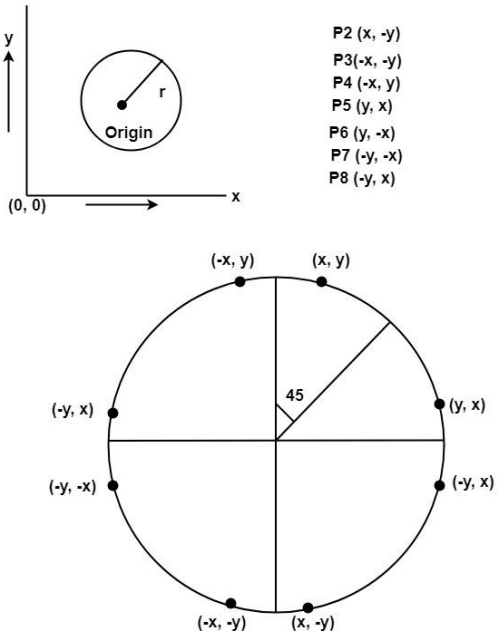
Step-05 generates all the points for one octant.

To find the points for other seven octants, follow the eight-symmetry property of circle.

This is depicted by the following figure-



Eight-Way Symmetry



Circle is an eight-way symmetric figure. The shape of circle is the same in all quadrants. In each quadrant, there are two octants. If the calculation of the point of one octant is done, then the other seven points can be calculated easily by using the concept of eight-way symmetry.

For drawing, circle considers it at the origin. If a point is $P_1(x, y)$, then the other seven points will be

So, we will calculate only 45°arc. From which the whole circle can be determined easily.

If we want to display circle on screen then the putpixel function is used for eight points as shown below:

putpixel (x, y, color)

putpixel (x, -y, color)

putpixel (-x, y, color)

putpixel (-x, -y, color)

putpixel (y, x, color)

putpixel (y, -x, color)

putpixel (-y, x, color)

putpixel (-y, -x, color)

Two-Dimensional Transformation

Translation

It is the straight-line movement of an object from one position to another is called Translation. Here the object is positioned from one coordinate location to another.

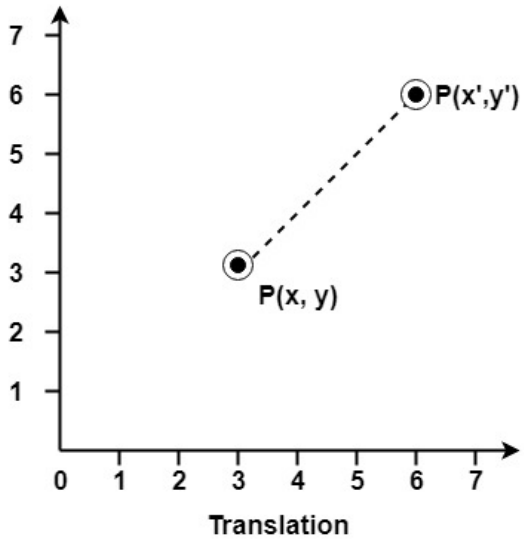
Translation of point:

To translate a point from coordinate position (x, y) to another (x_1 y_1), we add algebraically the translation distances T_x and T_y to original coordinate.

$x_1 = x + T_x$

$y_1 = y + T_y$

The translation pair (T_x,T_y) is called as shift vector.



Translation is a movement of objects without deformation. Every position or point is translated by the same amount. When the straight line is translated, then it will be drawn using endpoints.

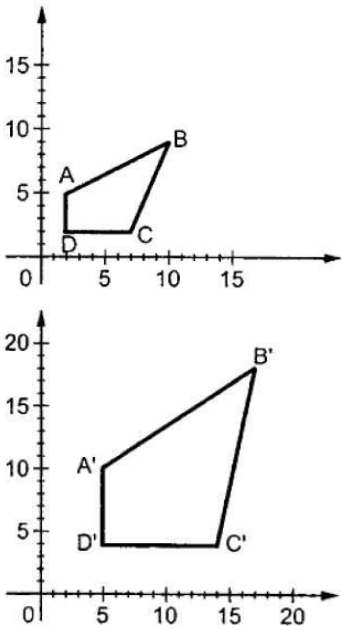
For translating polygon, each vertex of the polygon is converted to a new position. Similarly, curved objects are translated. To change the position of the circle or ellipse its centre coordinates are transformed, then the object is drawn using new coordinates.

Let P is a point with coordinates (x, y). It will be translated as (x^1 y^1).

So, the matrix $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$

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 . S_X 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 –

Matrix- $\begin{bmatrix} x` \\ y` \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} * \begin{bmatrix} S_x \\ S_y \end{bmatrix}$ OR

P' = P . S

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

Rotation:

It is a process of changing the angle of the object. Rotation can be clockwise or anticlockwise. For rotation, we have to specify the angle of rotation and rotation point. Rotation point is also called a pivot point. It is print about which object is rotated.

Types of Rotation:

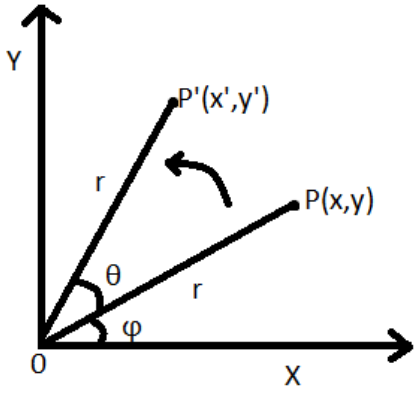
1. Anticlockwise
2. Counterclockwise

The positive value of the pivot point (rotation angle) rotates an object in a counter-clockwise (anti-clockwise) direction.

The negative value of the pivot point (rotation angle) rotates an object in a clockwise direction.

When the object is rotated, then every point of the object is rotated by the same angle.

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



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

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

Same way we can represent the point P' (X', Y') as −

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

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

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,

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

$$P` = P.R$$

Where R is the rotation matrix

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

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 −

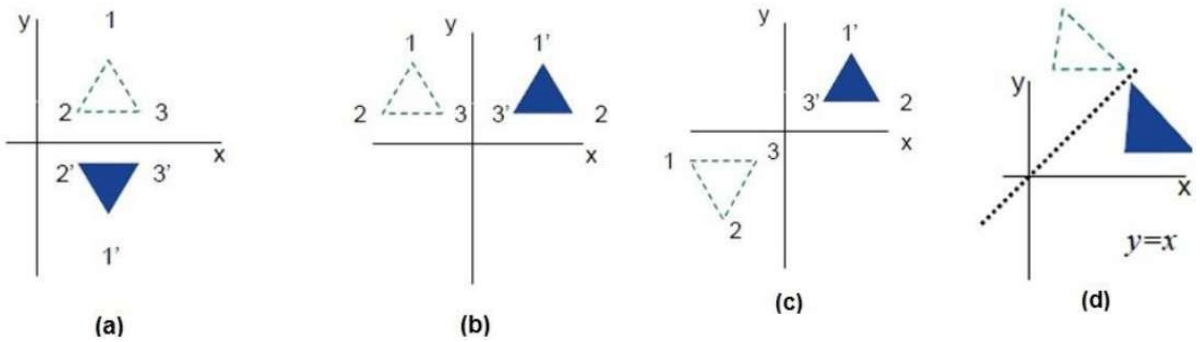
$$R = \begin{bmatrix}\cos(-\theta) & \sin(-\theta)\\ -\sin(-\theta) & \cos(-\theta)\end{bmatrix}$$

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

Reflection/Mirroring

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.

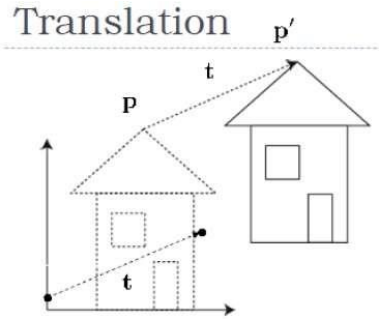


Three-Dimensional Transformation

Translation

In 3D translation, we transfer the Z coordinate along with the X and Y coordinates. The process for translation in 3D is similar to 2D translation. A translation moves an object into a different position on the screen.

The following figure shows the effect of translation −



A point can be translated in 3D by adding translation coordinate (tx, ty, tz)(tx, ty, tz) to the original coordinate (X, Y, Z) to get the new coordinate (X', Y', Z').

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

$$P` = P \cdot T$$

$$[X` \quad Y` \quad Z` \quad 1] = [X \quad Y \quad Z \quad 1] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

$$= [X + t_x \quad Y + t_y \quad Z + t_z \quad 1]$$

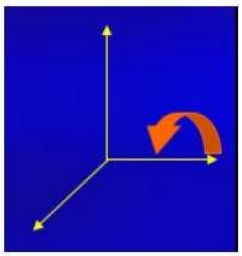
Rotation

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. They are represented in the matrix form as below –

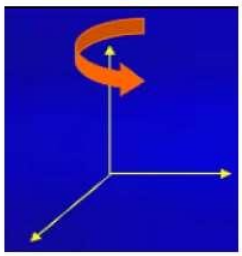
$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

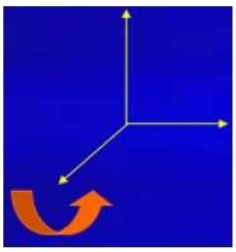
The following figure explains the rotation about various axes –



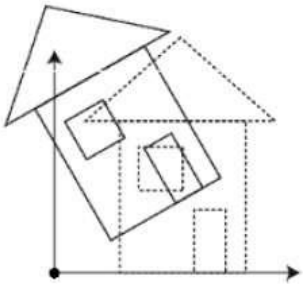
Rotation about x-axis



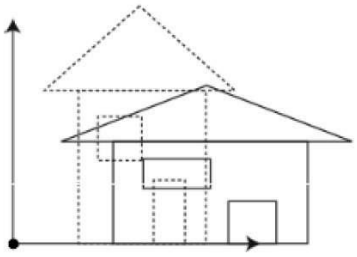
Rotation about y-axis



Rotation about z-axis



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. The following figure shows the effect of 3D scaling −

In 3D scaling operation, three coordinates are used. Let us assume that the original coordinates are (X, Y, Z), scaling factors are (S_X, S_Y, Sz) (S_X, S_Y, Sz) respectively, and the produced coordinates are (X`, Y`, Z`). This can be mathematically represented as shown below –

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{OR} \qquad P` = P \,.\, S$$

$$[X` \quad Y` \quad Z` \quad 1] = [X \quad Y \quad Z \quad 1] \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= [X.S_x \quad Y.S_y \quad Z.S_z \quad 1]$$

Composite Transformation

Composite transformation involves combining these basic transformations into a single operation. This is particularly useful in graphical applications where multiple transformations are often required to achieve the desired effect.

- **Combining Transformations**

Transformations can be combined by applying them in sequence. The order of application matters because different sequences will yield different results. For instance, rotating an object and then translating it will produce a different outcome than translating it first and then rotating it.

Example: Translation and Rotation

To how composite transformation understand Ons work, consider an example where an object is first rotated and then translated. By combining these transformations, we can move the rotated object to a

new location in one step. This simplifies the process and ensures the transformations are applied consistently.

- **Hierarchical Transformations**

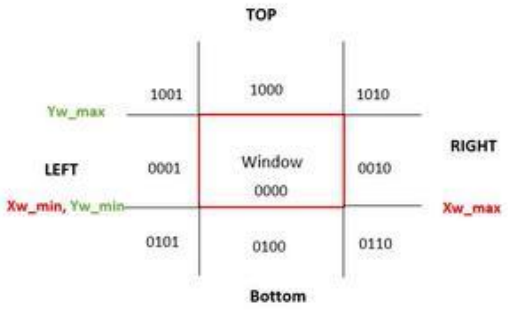
Composite transformations are particularly powerful in hierarchical modelling, where objects are composed of multiple sub-objects. For instance, consider a robot arm with multiple joints. Each segment of the arm can have its own transformation relative to its parent segment. By using composite transformations, we can propagate transformations from the base of the arm to the end-effector, creating a cohesive and coordinated movement.

$$\begin{cases} x` = x - a \\ y` = y - b \\ z` = z - c \end{cases}$$

Cohen–Sutherland Algorithm

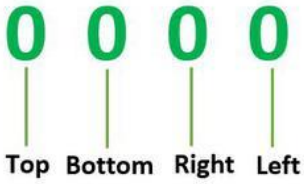
In this algorithm, we are given 9 regions on the screen. Out of which one region is of the window and the rest 8 regions are around it given by 4 digit binary. The division of the regions are based on (x_max, y_max) and (x_min, y_min).

The central part is the viewing region or window, all the lines which lie within this region are completely visible. A region code is always assigned to the endpoints of the given line.



To check whether the line is visible or not.

Formula to check binary digits: TBRL which can be defined as top, bottom, right, and left accordingly.



Algorithm

Steps

1) Assign the region codes to both endpoints.

2) Perform OR operation on both of these endpoints.

3) if OR = 0000,

then it is completely visible (inside the window).

else

Perform AND operation on both these endpoints.

i) if AND ? 0000,

then the line is invisible and not inside the window. Also, it can't be considered for clipping.

ii) else

AND = 0000, the line is partially inside the window and considered for clipping.

4) After confirming that the line is partially inside the window, then we find the intersection with the boundary of the window. By using the following formula:-

Slope:- m= (y2-y1)/(x2-x1)

a) If the line passes through top or the line intersects with the top boundary of the window.

x = x + (y_wmax – y)/m

y = y_wmax

b) If the line passes through the bottom or the line intersects with the bottom boundary of the window.

x = x + (y_wmin – y)/m

y = y_wmin

c) If the line passes through the left region or the line intersects with the left boundary of the window.

y = y+ (x_wmin – x)*m

x = x_wmin

d) If the line passes through the right region or the line intersects with the right boundary of the window.

y = y + (x_wmax -x)*m

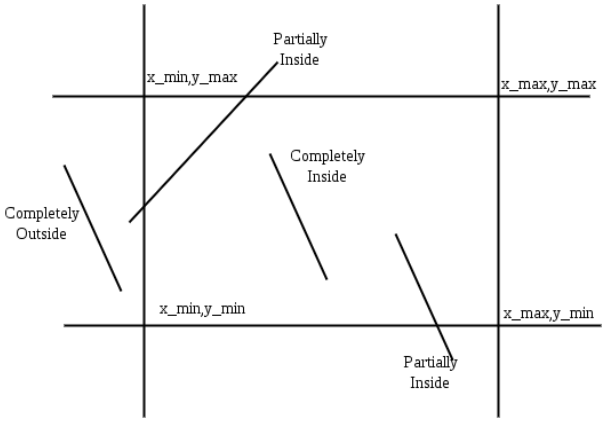
x = x_wmax

5) Now, overwrite the endpoints with a new one and update it.

6) Repeat the 4th step till your line doesn't get completely clipped

There are three possible cases for any given line.

1. **Completely inside the given rectangle:** Bitwise OR of region of two end points of line is 0 (Both points are inside the rectangle)
2. **Completely outside the given rectangle:** Both endpoints share at least one outside region which implies that the line does not cross the visible region. (bitwise AND of endpoints!= 0).
3. **Partially inside the window:** Both endpoints are in different regions. In this case, the algorithm finds one of the two points that is outside the rectangular region. The intersection of the line from outside point and rectangular window becomes new corner point and the algorithm repeats.



Z-Buffer Algorithm

It is also called a **Depth Buffer Algorithm**. Depth buffer algorithm is simplest image space algorithm. For each pixel on the display screen, we keep a record of the depth of an object within the pixel that lies closest to the observer. In addition to depth, we also record the intensity that should be displayed to

show the object. Depth buffer is an extension of the frame buffer. Depth buffer algorithm requires 2 arrays, intensity and depth each of which is indexed by pixel coordinates (x, y).

Algorithm

For all pixels on the screen, set depth [x, y] to 1.0 and intensity [x, y] to a background value.

For each polygon in the scene, find all pixels (x, y) that lie within the boundaries of a polygon when projected onto the screen. For each of these pixels:

(a) Calculate the depth z of the polygon at (x, y)

(b) If z < depth [x, y], this polygon is closer to the observer than others already recorded for this pixel. In this case, set depth [x, y] to z and intensity [x, y] to a value corresponding to polygon's shading. If instead z > depth [x, y], the polygon already recorded at (x, y) lies closer to the observer than does this new polygon, and no action is taken.

3. After all, polygons have been processed; the intensity array will contain the solution.

4. The depth buffer algorithm illustrates several features common to all hidden surface algorithms.

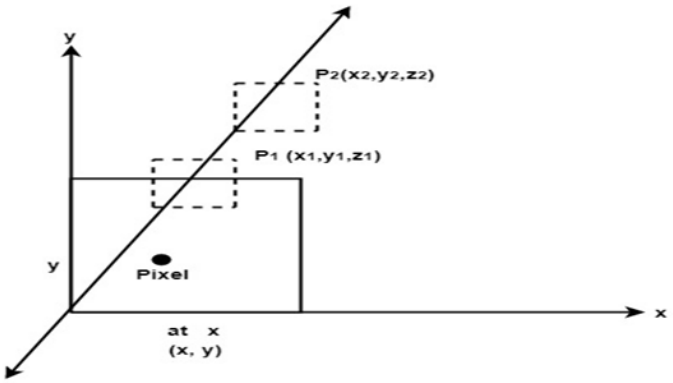
5. First, it requires a representation of all opaque surface in scene polygon in this case.

6. These polygons may be faces of polyhedral recorded in the model of scene or may simply represent thin opaque 'sheets' in the scene.

7. The IInd important feature of the algorithm is its use of a screen coordinate system. Before step 1, all polygons in the scene are transformed into a screen coordinate system using matrix multiplication.

Limitations of Depth Buffer

1. The depth buffer Algorithm is not always practical because of the enormous size of depth and intensity arrays.
2. Generating an image with a raster of 500 x 500 pixels requires 2, 50,000 storage locations for each array.
3. Even though the frame buffer may provide memory for intensity array, the depth array remains large.
4. To reduce the amount of storage required, the image can be divided into many smaller images, and the depth buffer algorithm is applied to each in turn.



5. For example, the original 500 x 500 faster can be divided into 100 rasters each 50 x 50 pixels.
6. Processing each small raster requires array of only 2500 elements, but execution time grows because each polygon is processed many times.
7. Subdivision of the screen does not always increase execution time instead it can help reduce the work required to generate the image. This reduction arises because of coherence between small regions of the screen.