Experiment No : 1

Aim: **Implementation of DDA (Digital Differential Analyzer) algorithm.**

Description:

The DDA starts by calculating the smaller of dy or dx for a unit increment of the other. A line is then sampled at unit intervals in one coordinate and corresponding integer values nearest the line path are determined for the other coordinate.

Considering a line with positive slope, if the slope is less than or equal to 1, we sample at unit x intervals (dx=1) and compute successive y values as

$$y_{k+1} = y_k + m$$

Subscript k takes integer values starting from 0, for the 1st point and increases by 1 until endpoint is reached. y value is rounded off to nearest integer to correspond to a screen pixel.

For lines with slope greater than 1, we reverse the role of x and y i.e. we sample at dy=1 and calculate consecutive x values as

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

Similar calculations are carried out to determine pixel positions along a line with negative slope. Thus, if the absolute value of the slope is less than 1, we set dx=1 if $x_{start} < x_{end}$i.e. the starting extreme point is at the left.

DDA algorithm:

1. Define the nodes, i.e. end points in form of (x_1,y_1) and (x_2,y_2).
2. Calculate the distance between the two end points vertically and horizontally, i.e $dx=|x_1-x_2|$ and $dy=|y_1-y_2|$.
3. Define new variable name ‘steps’, and compare dx and dy values,
   if dx > dy then
   steps=dx
   else
   steps =dy.
4. dx=dx/steps
   and dy=dy/steps
5. $x_k=x_1$;
   $y_k=y_1$;
6. while (i<=steps) compute the pixel and plot the pixel with $x_{k+1}=x_k+dx$ and $y_{k+1}=y_k+dy$.

Experiment No: 2

Aim: **Implementation of Bresenham Line Drawing algorithm.**

Description:

Bresenham's line algorithm is an algorithm that determines the points of an *n*-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is one of the earliest algorithms developed in the field of computer graphics. An extension to the original algorithm may be used for drawing circles.

Bresenham Line Drawing Algorithm For Slope Less Than 1:

1: Get the line endpoints from the user.

(X_1, Y_1) and (X_2, Y_2) are the endpoints of a line.

2: Calculate dx, dy, 2dy and (2dy - 2dx)

$dx = X_2 – X_1$

$dy = Y_2 – Y_1$

3: Find the initial value of the decision parameter(P).

$P_0 = 2dy - dx$

4: If $P_k < 0$, then the next points to plot are(k starts at 0)

$X_{k+1} = X_k + 1$

$Y_{k+1} = Y_k$

and $P_{k+1} = P_k + 2dy$

Otherwise, the next points to plot are

$X_{k+1} = X_k + 1$

$Y_{k+1} = Y_k + 1$

and $P_{k+1} = P_k + 2dy - 2dx$

5: Repeat step 4 dx times.

Experiment No : 3

Aim: Implementation of mid-point circle generation algorithm.

Description:

In Midpoint Circle Algorithm, the decision parameter at the k^{th} step is the circle function evaluated using the coordinates of the midpoint of the two pixel centres which are the next possible pixel position to be plotted.

Let us assume that we are giving unit increments to x in the plotting process and determining the y position using this algorithm. Assuming we have just plotted the k^{th} pixel at (X_k,Y_k), we need next to determine whether the pixel at the position (X_{k+1},Y_k) or the one at (X_{k+1} , Y_{k-1}) is closer to the circle.

MID – POINT CIRCLE ALGORITHM

Mid-Point Circle (Xc, Yc, R):

```
1. Accept (Xc,Yc) and radius R.
2. Set $X_K = 0$ and $Y_K = R$
3. Set $P_K = 1 – R$
4. Repeat While ($X_K < Y_K$)
5. Call Draw Circle(Xc, Yc, $X_K$, $Y_K$)
6. Set $X_{K+1} = X_K + 1$
7. If ($P_K < 0$) Then
8. $P_{K+1} = P_K + 2X_K + 3$
9. Else
10. Set $Y_{K+1} = Y_K – 1$
11. $P_{K+1} = P_K + 2(X_K – Y_K) + 5$
     [End of If]
12. Call Draw Circle(Xc, Yc, $X_K$, $Y_K$)
           [End of While]
13. Exit
```

Draw Circle (Xc, Yc, X, Y):

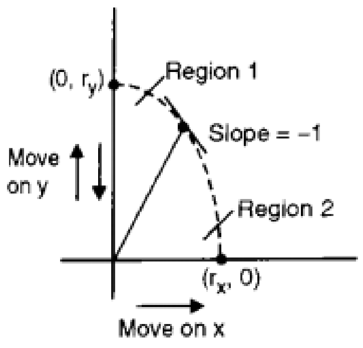
```
1. Call PutPixel(Xc + X, Yc, + Y)
2. Call PutPixel(Xc - X, Yc, + Y)
3. Call PutPixel(Xc + X, Yc, - Y)
4. Call PutPixel(Xc - X, Yc, - Y)
5. Call PutPixel(Xc + Y, Yc, + X)
6. Call PutPixel(Xc - Y, Yc, + X)
7. Call PutPixel(Xc + Y, Yc, - X)
8. Call PutPixel(Xc - Y, Yc, - X)
9. Exit
```

Experiment No : 4

Aim: **Implementation of mid-point ellipse drawing algorithm.**

Description:

The midpoint ellipse method is applied throughout the first quadrant in two parts i.e. region- I and region-2. We are forming this regions by considering the slope of the curve. If the slope of the curve is less than - I then we are in region-l and when the slope becomes greater than - 1 then in region-2.



$$\frac{dy}{dx} = -1.$$

The slope of the ellipse is calculated as

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

Fig. 2.7.3

At the boundary between region-1 & 2,

$$\frac{dy}{dx} = -1 \quad \text{and} \quad 2r_y^2 x = 2 r_x^2 y$$

So, we move out of region-1 when

$$2 r_y^2 x \geq 2 r_x^2 y$$

Mid-Point Elliplse (X_C, Y_C, R_X, R_Y):

Description: Here $\mathbf{X_C}$ and $\mathbf{Y_C}$ denote the x – coordinate and y – coordinate of the center of the ellipse and $\mathbf{R_X}$ and $\mathbf{R_Y}$ denote the x – radius and y – radius respectively.

1. Accept R_X , R_Y , (X_C,Y_C).
2. Set RXSq = R_X * R_X
3. Set RYSq = R_Y * R_Y
4. Set $X_K = 0$ and $Y_K = R_Y$
5. Set $P_X = 0$ and P_Y = 2 * RXSq * Y
6. Call Draw Elliplse(X_C, Y_C, X_K, Y_K)
7. Set P_K = RYSq – (RXSq * RY) + (0.25 * RXSq) [Region 1]
8. Repeat While ($P_X < P_Y$)
9. Set $X_{K+1} = X_K + 1$

```
10.  $P_X = P_X + 2 * RYSq$
11.  If ($P_K < 0$) Then
12.  Set $P_{K+1} = P_K + RYSq + P_X$
13.  Else
14.  Set $Y_{K+1} = Y_K – 1$
15.  Set $P_Y = P_Y – 2 * RXSq$
16.  Set $P_{K+1} = P_K + RYSq + P_X – P_Y$
[End of If]
17.  Call Draw Elliplse($X_C$, $Y_C$, $X_K$, $Y_K$)
[End of Step 7 While]
18.  Set $P_{K+1} = RYSq*(X_K + 0.5)^2 + RXSq*(Y_K – 1)^2 – RXSq*RYSq$ [Region 2]
19.  Repeat While ($Y > 0$)
20.  Set $Y_{K+1} = Y_K – 1$
21.  Set $P_Y = P_Y – 2 * RXSq$
22.  If ($P > 0$) Then
23.  Set $P_{K+1} = P_K + RXSq – P_Y$
24.  Else
25.  Set $X_{K+1} = X_K + 1$
26.  Set $P_X = P_X + 2 * RYSq$
27.  Set $P_{K+1} = P_K + RXSq – P_Y + P_X$
 [End of If]
28.  Call Draw Ellipse($X_C$, $Y_C$, $X_K$, $Y_K$)
[End of Step 18 While]
29.  Exit
```

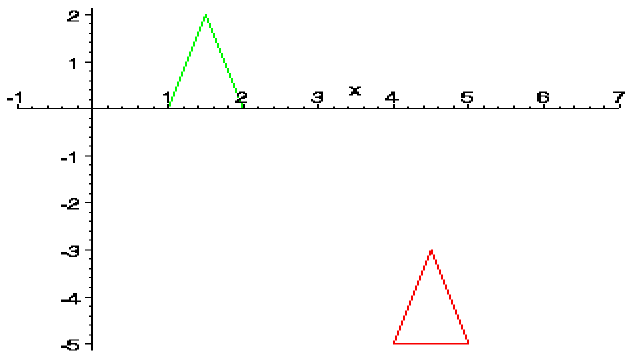
Draw Ellipse (X_C, Y_C, X_K, Y_K):

```
1.  Call PutPixel($X_C + X_K$, $Y_C + Y_K$)
2.  Call PutPixel($X_C - X_K$, $Y_C + Y_K$)
3.  Call PutPixel($X_C + X_K$, $Y_C - Y_K$)
4.  Call PutPixel($X_C - X_K$, $Y_C - Y_K$)
5.  Exit
```

Experiment No: 5

Aim: To Perform Translation of an Object.

Theory:



Let us look at the procedure for c arrying out basic transformations, which are based on matrix operation. A transformation can be expressed as

$[P^*] = [P]\,[T]$

Where, $[P^*]$ is the new coordinates matrix
[P] Is the original coordinates matrix, or points matrix
[T] Is the transformation matrix

In translation, every point on an object translates exactly the same distance. The effect of a translation transformation is that the original coordinate values increase or decrease by the amount of the translation along the x, y, and z-axes.

The transformation matrix has the form:

$$[T] = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

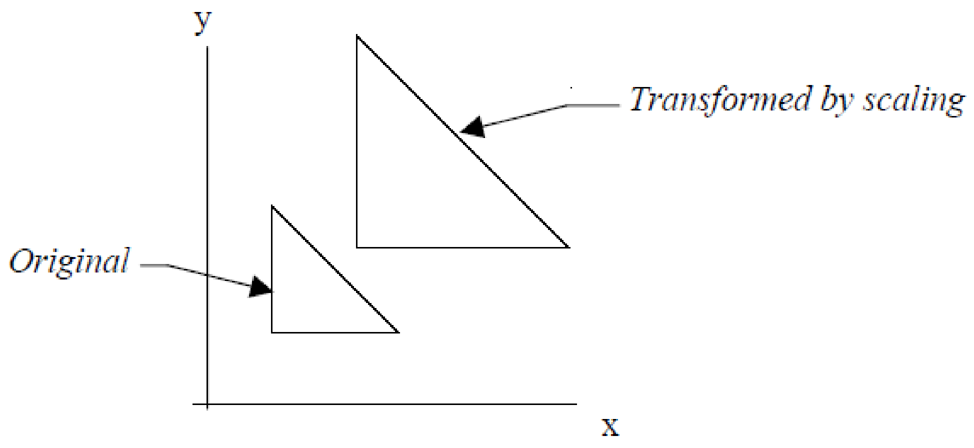
The resultant transformed matrix is given by

$$[P^*] = \begin{pmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ z1 & z2 & z3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Experiment No: 6

Aim: **To perform Scaling of an Object.**

Theory:



n scaling transformation, the original coordinates of an object are multiplied by the given scale factor. There are two types of scaling transformations: uniform and non-uniform. In the uniform scaling, the coordinate values change uniformly along the x, y, and z coordinates, whereas, in non-uniform scaling, the change is not necessarily the same in all the coordinate directions.

Matrix equation of a non-uniform scaling has the form:

$$[T] = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The resultant transformed matrix is given by

$$[P^*] = \begin{pmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ z1 & z2 & z3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Experiment No: 7

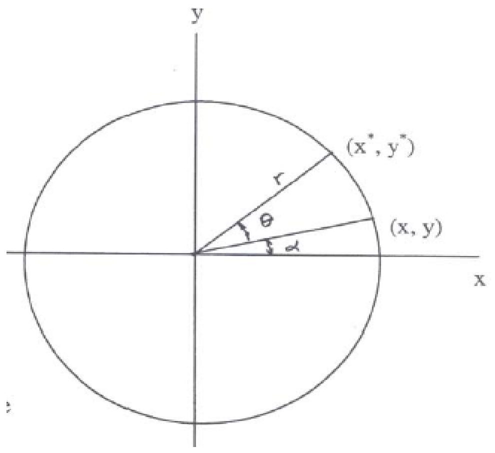
Aim: To Perform Rotation of an Object

Theory:

Using trigonometric relations, as given below, we can derive the rotation transformation matrix. Let the point P(x, y) be on the circle, located at an angle α, as shown. If the point P is rotated an additional angle θ, the new point will have the coordinates (x*, y*). The angle and the original coordinate relationship is found as follows.

$$\left.\begin{aligned} x &= r\cos\alpha \\ y &= r\sin\alpha \end{aligned}\right\} \textit{Original coordinates of point P.}$$

$$\left.\begin{aligned} x^* &= r\cos(\alpha+\theta) \\ y^* &= r\sin(\alpha+\theta) \end{aligned}\right\} \textit{The new coordinates.}$$



where, α is the angle between the line joining the initial position of the point and the x-axis, and θ is the angle between the original and the new position of the point.

Using the trigonometric relations,
we get,
$x^* = r(\cos\alpha\cos\theta - \sin\alpha\sin\theta) = x\cos\theta - y\sin\theta$
$y^* = r(\cos\alpha\sin\theta + \sin\alpha\cos\theta) = x\sin\theta + y\cos\theta$

In matrix form we can write these equations as*

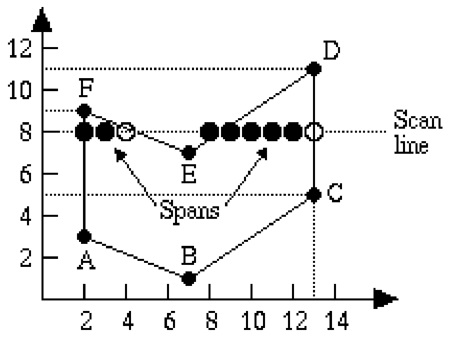
$$[P^*] = \begin{pmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

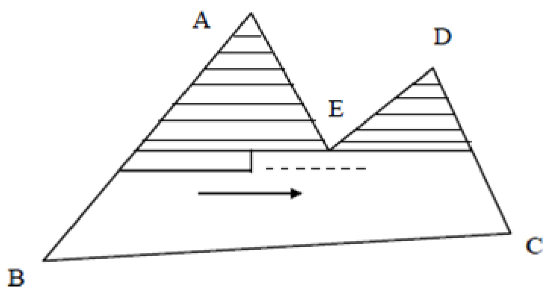
Experiment No: 8

AIM: To implement scan line polygon filling algorithm.

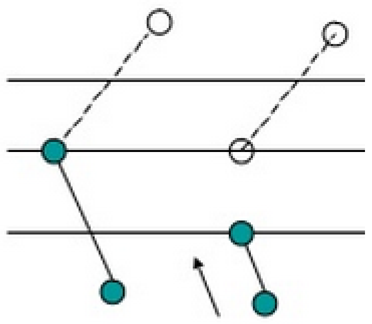
THEORY:

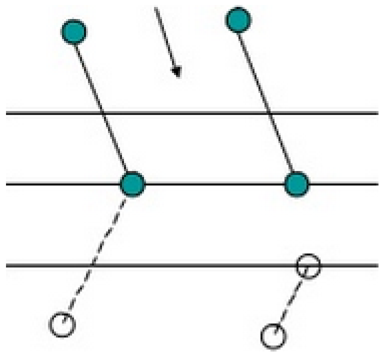
The scan line fill algorithm is an ingenious way of filling in irregular polygons. This algorithm starts with first scan line and proceeds line by line toward the last scan line and checks whether every pixel on that scan line satisfies our inside point test or not i.e. it checks which points on that scan line are inside the polygon. This method avoids the need for seed point. Hence, it can fill figures without knowing boundary colour as well as figures which have unfilled regions in them. Here important point is how to find out the intersection point of scan line with a particular edge.
For example, consider the polygon in the figure. Here, algorithm begins at the first scan line at D and proceeds line by line till last scan line is encountered at B. If the scan line intersects an edge then there are even intersection points of the edges and scan line so they are paired from left to right and the region between each pair is filled. In this figure scan line intersects edge AF at p1, edge EF at point p2, edge ED at point p3 and edge CD at point p4. p1 and p2 are paired and p3 and p4 form the other pair. The region between these paired points is coloured. The region between p2 and p3 is left uncoloured as they already belong to some pair.





But vertices (edge end points) come under a different category. If a scan line encounters a vertex then we have two cases. One would be where the edges are monotonically increasing or decreasing. In this case we shorten the lower edge and thus get two different points from that vertex.





Thus, if the polygon is moving in the same direction from edge e1 to edge e2, p1 is not counted twice (figure 1). If the direction is reversed at the vertex, we need to duplicate the point p1 and form pairs (by counting p1 twice) as shown in figure2.

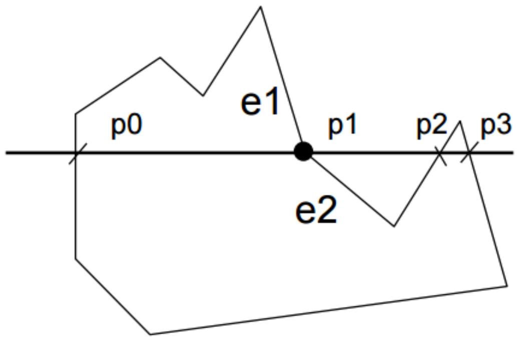


figure 1

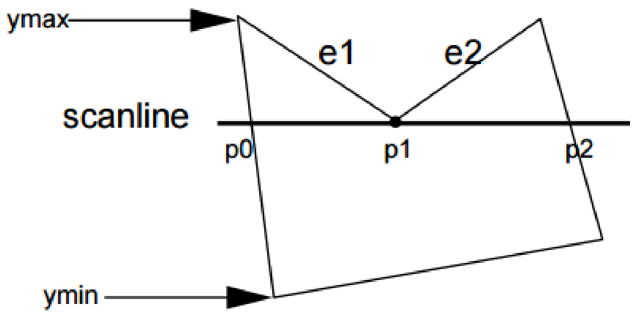


figure 2

Thus, we accordingly take the decision in the case of reversal whether the region above the point will be colored or below. As we are not using 4 or 8 connected algorithm, we avoid recursion. We can fill large objects faster and can also fill regions where edges cross each other.

ALGORITHM:

Basic:

1. Accept the points p0, p1, p2… that are the vertices of the polygon.
2. Find the ymax and ymin values from the given vertices.
3. For y = ymin to ymax .
4. Intersect scanline y with each edge.(finding active edge list)
5. Sort intersections (active edges) by increasing x [p0, p1, p2, p3….].
6. Fill pairwise (p0 –> p1, p2–> p3 ...).

Special cases (found using slope):

A) Intersection is an edge end point with direction reversed:
Intersection points: (p0, p1, p2….)
–> (p0,p1,p1,p2…..) so we can still fill pairwise.

B) Intersection is an edge end point in same direction:
No duplication will take place, lower edge will be shortened.

Experiment No: 9

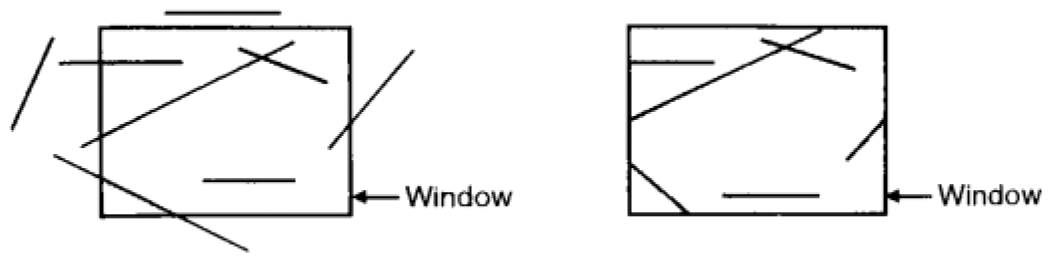
Aim: To implement Cohen – Sutherland Line Clipping algorithm

Theory:

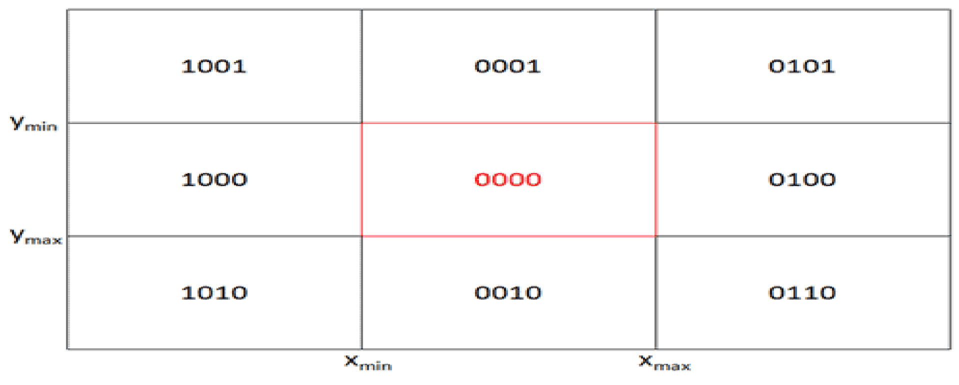
The algorithm includes, excludes or partially includes the line based on where:

- Both endpoints are in the viewport region (bitwise OR of endpoints == 0): trivial accept.
- Both endpoints share at least one non-visible region which implies that the line does not cross the visible region. (Bitwise AND of endpoints ! = 0): trivial reject.
- Both endpoints are in different regions: In case of this nontrivial situation the algorithm finds one of the two points that is outside the viewport region (there will be at least one point outside). The intersection of the outpoint and extended viewport border is then calculated (i.e. with the parametric equation for the line)
  Use formula $y = y_0 + slope * (x - x_0)$,
  $x = x_0 + (1 / slope) * (y - y_0)$
- And this new point replaces the outpoint. The algorithm repeats until a trivial accepts or reject occurs.

The numbers in the figure below are called outcodes. An outcode is computed for each of the two points in the line. The outcode will have four bits for two-dimensional clipping, or six bits in the three-dimensional case. The first bit is set to 1 if the point is above the viewport. The bits in the 2D outcode represent: Top, Bottom, Right, Left. For example the outcode 1010 represents a point that is top-right of the viewport. Note that the outcodes for endpoints **must** be recalculated on each iteration after the clipping occurs.



The Cohen–Sutherland algorithm can be used only on a rectangular clipping area.



Experiment No: 10

Aim: To implement Liang-Barsky Line Clipping algorithm

Theory:

The Liang–Barsky algorithm uses the parametric equation of a line and inequalities describing the range of the clipping window to determine the intersections between the line and the clipping window. With these intersections it knows which portion of the line should be drawn. This algorithm is significantly more efficient than Cohen–Sutherland.

The idea of the Liang-Barsky clipping algorithm is to do as much testing as possible before computing line intersections. Consider first the usual parametric form of a straight line:

$$x = x_0 + u(x_1 - x_0) = x_0 + u\Delta x$$
$$y = y_0 + u(y_1 - y_0) = y_0 + u\Delta y$$

A point is in the clip window, if

$$x_{\min} \le x_0 + u\Delta x \le x_{\max}$$

And

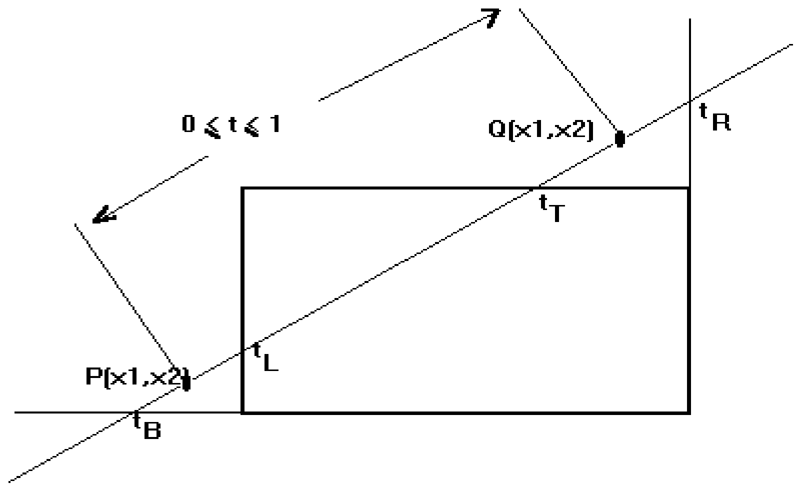
$$y_{\min} \le y_0 + u\Delta y \le y_{\max},$$

This can be expressed as the 4 inequalities

$$up_k \le q_k, \quad k = 1, 2, 3, 4,$$

Where

$p_1 = -\Delta x, q_1 = x_0 - x_{\min}$(Left)
$p_2 = \Delta x, q_2 = x_{\max} - x_0$(Right)
$p_3 = -\Delta y, q_3 = y_0 - y_{\min}$(Bottom)
$p_4 = \Delta y, q_4 = y_{\max} - y_0$(Top)



To compute the final line segment:

1. A line parallel to a clipping window edge has $p_k = 0$ for that boundary.
2. If for that k, $q_k < 0$, the line is completely outside and can be eliminated.
3. When $p_k < 0$ the line proceeds outside to inside the clip window and when $p_k > 0$, the line proceeds inside to outside.
4. For nonzero p_k, $u = \frac{q_k}{p_k}$ gives the intersection point.
5. For each line, calculate u_1 and u_2. For u_1, look at boundaries for which $p_k < 0$ (i.e. outside to inside). Take u_1 to be the largest among $\left\{0, \frac{q_k}{p_k}\right\}$. For u_2, look at boundaries for which $p_k > 0$ (i.e. inside to outside). Take u_2 to be the minimum of $\left\{1, \frac{q_k}{p_k}\right\}$. If $u_1 > u_2$, the line is outside and therefore rejected.
6. The resultant coordinates of the line (x',y') and (x'',y'') are given by :

   $x' = x_0 + (u_1 * dx);$
   $y' = y_0 + (u_1 * dy);$

   $x'' = x_0 + (u_2 * dx);$
   $y'' = y_0 + (u_2 * dy);$

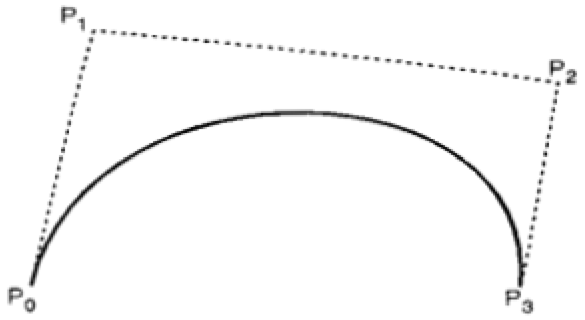
Experiment No: 11

AIM: **Implementing Bezier curves**

THEORY:

Cubic curves are commonly used in graphics because curves of lower order Commonly have too little flexibility, while curves of higher order are usually Considered unnecessarily complex and make it easy to introduce undesired wiggles. To Join multiple curves into one curve smoothly, we need to specify the positions and their tangent-vectors of both ends, i.e. the continuity requirements.

Four points Po, PI, P2 and P3 in the plane or in three-dimensional space define a cubic Bezier curve. The curve starts at Po going toward PI and arrives at P3 coming from the direction of P2. Usually, it will not pass through PI or P2; these points are only there to provide directional information. The distance between Po and PI determines "how long" the curve moves into direction P2 before turning towards



A cubic Bezier curve indirectly specifies the endpoint tangent-vector by Specifying two intermediate points that are not on the curve.

Points on the curve are given by
$X = X_4{*}a^3 + 3{*}X_3{*}a^2 *(1 - a) + 3{*}X_2{*}a *(1 - a)^2 + X_l *(1 - a)^3$

$Y = Y_4{*}a^3 + 3{*}Y_3{*}a^{2*} (1- a) + 3{*}Y_2{*}a *(1- a)^2 + Y_1{*}(1- a)^3$

Algorithm:

1. Accept the number of control points ‘n’
2. Accept the control points $(x_1, y_1), (x_2, y_2)\ldots (x_n ,y_n)$
3. Repeat steps 4 to 6 while ‘a’ goes from 0 -1
4. Calculate $x' = \sum({}^nC_k{*}x_k{*}a^k(1-a)^{(n-k)})$
5. Calculate $y' = \sum({}^nC_k{*}y_k{*}a^k(1-a)^{(n-k)})$
6. Plot (x’,y’)

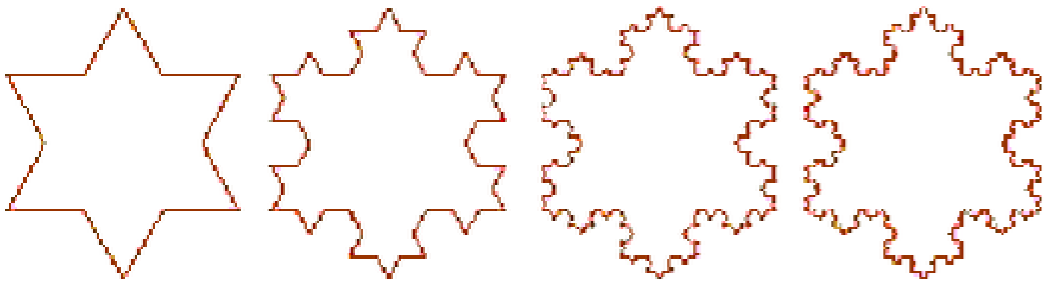
Experiment No: 12

AIM: Fractal generation

THEORY:

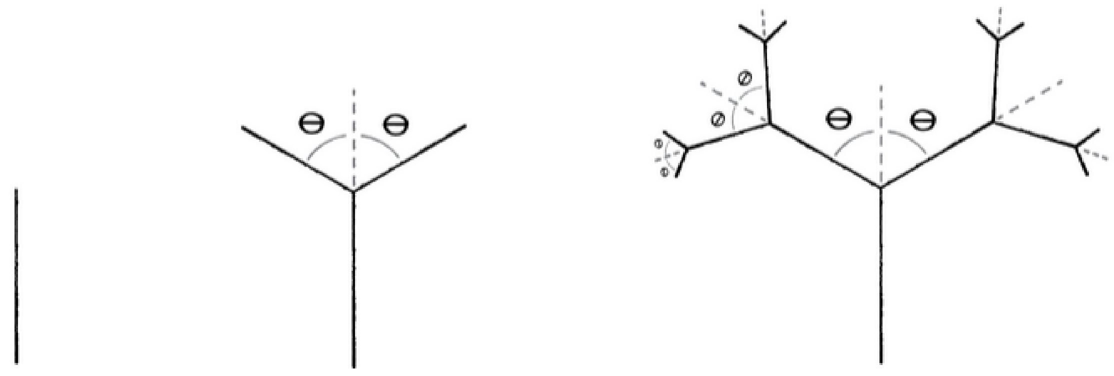
A fractal is a natural phenomenon or a mathematical set that exhibits a repeating pattern that displays at every scale. If the replication is exactly the same at every scale, it is called a self-similar pattern.

One of the basic properties that characterize fractals is self-similarity. The self-similarity property of an object can take different forms, depending on the choice of fractal representation. Self-similarity means if we zoom into a piece of a fractal we will keep seeing the same structures repeated over and over.



Algorithm:

1. Draw a line.
2. At the end of the line, (a) rotate to the left and draw a shorter line and (b) rotate to the right and draw a shorter line.
3. Repeat step 2 for the new lines, again and again and again.



Experiment No: 13

AIM: Polygon generation using mouse interaction

THEORY:

glutMouseFunc sets the mouse callback for the *current window*.

Usage

```
void glutMouseFunc(void (*func)(int button, int state,
                                int x, int y));
```

func

The new mouse callback function.

Description

glutMouseFunc sets the mouse callback for the *current window*. When a user presses and releases mouse buttons in the window, each press and each release generates a mouse callback. The button parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. For systems with only two mouse buttons, it may not be possible to generate GLUT_MIDDLE_BUTTON callback. For systems with a single mouse button, it may be possible to generate only a GLUT_LEFT_BUTTON callback. The state parameter is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively. The x and y callback parameters indicate the window relative coordinates when the mouse button state changed. If a GLUT_DOWN callback for a specific button is triggered, the program can assume a GLUT_UP callback for the same button will be generated (assuming the window still has a mouse callback registered) when the mouse button is released even if the mouse has moved outside the window.

If a menu is attached to a button for a window, mouse callbacks will not be generated for that button.

During a mouse callback, glutGetModifiers may be called to determine the state of modifier keys when the mouse event generating the callback occurred.

Passing NULL to glutMouseFunc disables the generation of mouse callbacks.