Vidyavardhini's College of Engineering and Technology
Department of Artificial Intelligence & Data Science

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	1
Title:	Digital Differential Analyzer Line Drawing Algorithm
Marks:	
Sign of Faculty:	

Experiment No. 1

Aim : Write a program to implement Digital Differential Analyzer Line Drawing Algorithm in C.

Objective: To implement DDA line drawing algorithm for drawing a line segment between two points A (x1, y1) and B (x2, y2)

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

DDA Working Mechanism

Case 1: Slope < 1 (m<1)(theta<45)x coordinates increase fast
dx is set to unit interval dx=1; dy is computed; m = dy/dx therefore; dy = m Calculation for next pixel for line processed from left to right
$x_{k+1} = x_k + dx = x_k + 1$
$y_{k+1} = y_k + dy = y_k + m$
Calculation for next pixel for line processed from right to left $x_{k+1} = x_k + dx = x_k - 1$
$y_{k+1} = y_k + dy = y_k - m$

Case 2: Slope > 1 (m>1) theta>45 y coordinates increase fast
dy is set to unit interval dy=1; dx is computed; m = dy/dx therefore; dx = 1/m
Calculation for next pixel for line processed from left to right
$x_{k+1} = x_k + dx = x_k + 1/m$
$y_{k+1} = y_k + dy = y_k + 1$
Calculation for next pixel for line processed from right to left $x_{k+1} = x_k + dx = x_k - 1/m$
$y_{k+1} = y_k + dy = y_k - 1$

Case 3: Slope =1 (m=1) theta=45
dx and dy is set to unit interval dx=1 and dy=1
Calculation for next pixel for line processed from left to right $x_{k+1} = x_k + dx = x_k + 1$
$y_{k+1} = y_k + dy = y_k + 1$
Calculation for next pixel for line processed from right to left $x_{k+1} = x_k + dx = x_k - 1$
$y_{k+1} = y_k + dy = y_k - 1$

Algorithm

DDA Algorithm:

Step1: Start Algorithm
Step2: Declare x_1, y_1, x_2, y_2, dx, dy, x, y as integer variables.
Step3: Enter value of x_1, y_1, x_2, y_2.

Step4: Calculate dx = x_2-x_1
Step5: Calculate dy = y_2-y_1
Step6: If ABS (dx) > ABS (dy)
Then step = abs (dx)
Else
Step7: x_{inc}=dx/step
y_{inc}=dy/step
assign x = x_1
assign y = y_1
Step8: Set pixel (x, y)
Step9: x = x + x_{inc}
y = y + y_{inc}
Set pixels (Round (x), Round (y)) **Step10:** Repeat step 9 until x = x_2
Step11: End Algorithm

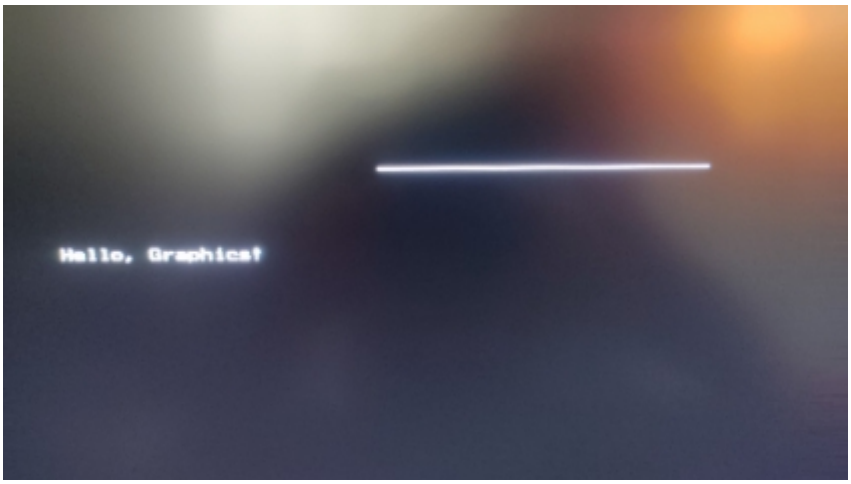
Code :

```c
#include <stdio.h>
#include <graphics.h>
void drawLineDDA(int x1, int y1, int x2, int y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    float steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);
    float xIncrement = dx / steps;
    float yIncrement = dy / steps;
    float x = x1;
    float y = y1;
    putpixel(x, y, WHITE);
    for (int i = 1; i <= steps; i++) {
        x += xIncrement;
        y += yIncrement;

        putpixel(round(x), round(y), WHITE);
    }
}
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "");
    int x1, y1, x2, y2;
    printf("Enter the coordinates of the first point (x1 y1): ");
    scanf("%d %d", &x1, &y1);
    printf("Enter the coordinates of the second point (x2 y2): ");
    scanf("%d %d", &x2, &y2);
    drawLineDDA(x1, y1, x2, y2);
```

```
    delay(5000);
    closegraph();
    return 0;
}
```

Output :



Conclusion :

The Digital Differential Analyzer (DDA) algorithm is a straightforward and efficient method for drawing lines on a computer screen. It is based on linear interpolation using floating-point arithmetic to determine the pixel positions along the line. The algorithm is easy to understand and implement, making it suitable for simple graphics applications.

However, the DDA algorithm has some limitations. It relies on floating-point arithmetic, which may not be ideal for systems with limited computational resources. Moreover, rounding errors during calculations can accumulate, leading to slight deviations from the ideal line. Additionally, the DDA algorithm is sensitive to the choice of the number of steps, and an inappropriate number of steps may result in a loss of precision or graphical artifacts.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
	2
Experiment No.:	Bresenham's Line Drawing Algorithm

Title:	
Marks:	
Sign of Faculty:	

Experiment No. 2

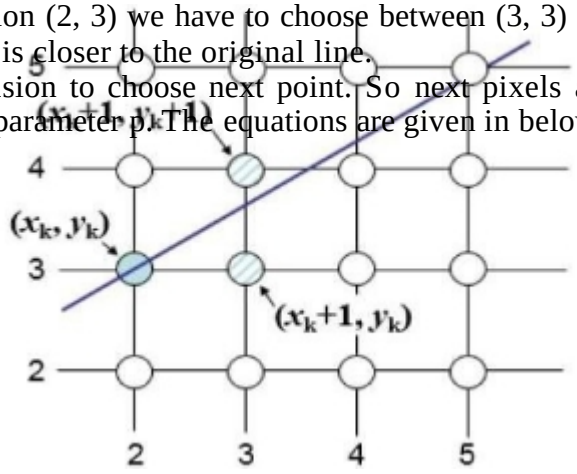
Aim :Write a program to implement Bresenham Line Drawing Algorithm in C.

Objective :To implement Bresenham line drawing algorithm for drawing a line segment between two points A (x1, y1) and B (x2, y2)

Theory :

Bresenham 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

Basic Concept:
Move across the x axis in unit intervals and at each step choose between two different y coordinates.
For example, from position (2, 3) we have to choose between (3, 3) and (3, 4). We would like the point that is closer to the original line.
So we have to take decision to choose next point. So next pixels are selected based on the value of decision parameter p. The equations are given in below algorithm



Algorithm :

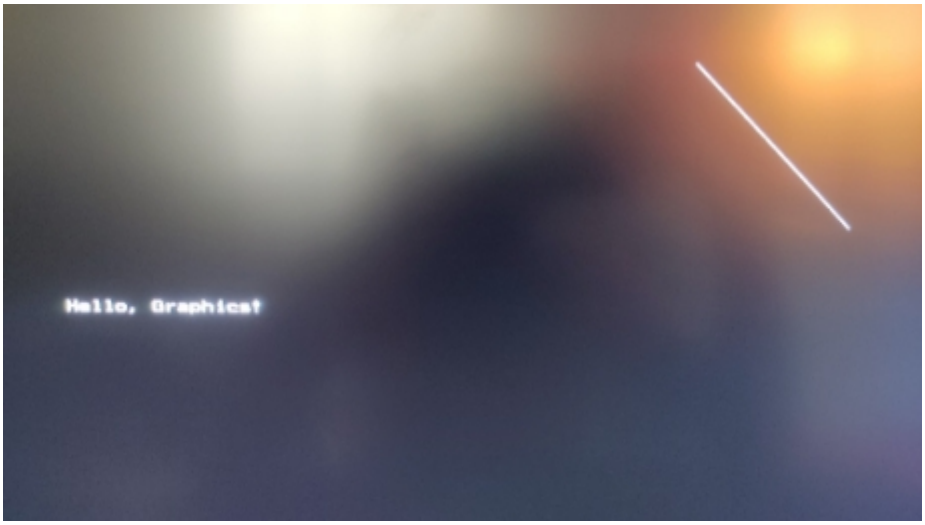
Step1: StartAlgorithm
Step2: Declare variable x1,x2,y1,y2,d,i1,i2,dx,dy
Step3: Enter value of x1,y1,x2,y2
Where x1,y1are coordinates of starting point And x2,y2 are coordinates of Ending point
Step4: Calculate dx = x2-x1 Calculate dy = y2-y1 Calculate i1=2*dy Calculate i2=2*(dy-dx) Calculate d=i1-dx
Step5: Consider (x, y) as starting point and xend as maximum possible value of x.
If dx < 0
Then x = x2 y = y2 xend=x1
If dx > 0 Then x = x1
y = y1 xend=x2
Step6: Generate point at (x,y)coordinates.
Step7: Check if whole line is generated.
If x > = xend Stop.
Step8: Calculate co-ordinates of the next pixel If d < 0
Then d = d + i1 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

Code :

```
#include <stdio.h>
#include <graphics.h>
void drawLineBresenham(int x1, int y1, int x2, int y2) {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int slope_gt_one = 0;
    if (dy > dx) {
        slope_gt_one = 1;
        int temp = x1;
        x1 = y1;
        y1 = temp;
        temp = x2;
        x2 = y2;
        y2 = temp;
```

```
        dx = abs(x2 - x1);
        dy = abs(y2 - y1);
    }
    int p = 2 * dy - dx;
    int twoDy = 2 * dy;
    int twoDyMinusDx = 2 * (dy - dx);
    int x, y, xEnd;
    if (x1 > x2) {
        x = x2;
        y = y2;
        xEnd = x1;
    } else {
        x = x1;
        y = y1;
        xEnd = x2;
    }
    putpixel(x, y, WHITE);
    while (x < xEnd) {
        x++;
        if (p < 0)
            p += twoDy;
        else {
            y++;
            p += twoDyMinusDx;
        }
        if (slope_gt_one)
            putpixel(y, x, WHITE);
        else
            putpixel(x, y, WHITE);
    }
    closegraph();
}
int main() {
    int x1, y1, x2, y2;
    printf("Enter the starting point (x1, y1): ");
    scanf("%d %d", &x1, &y1);
    printf("Enter the ending point (x2, y2): ");
    scanf("%d %d", &x2, &y2);
    drawLineBresenham(x1, y1, x2, y2);
    return 0;
}
```

Output



Conclusion :

The Bresenham Line Drawing Algorithm is a powerful and efficient method for drawing lines on a computer screen using integer arithmetic. It avoids the need for floating-point calculations, making it faster and more suitable for systems with limited computational resources.

One key advantage of the Bresenham algorithm is its ability to make decisions based on integer operations and avoid the overhead associated with floating-point arithmetic. This leads to a faster execution and is particularly beneficial in resource-constrained environments.

The algorithm is also versatile and can be extended to draw lines at any angle efficiently. However, it is specific to drawing lines and doesn't handle other shapes. In contemporary graphics programming, hardware acceleration and more advanced algorithms are often used for efficiency, but the Bresenham algorithm remains a fundamental concept and is widely used in various applications.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	3
Title:	Midpoint Circle Drawing Algorithm
Marks:	
Sign of Faculty:	

Experiment No. 3

Aim :Write a program to implement Midpoint Circle Drawing Algorithm in C.

Objective :To implement midpoint circle drawing algorithm for drawing a circle with radius (R) and centre (Xc, Yc)

Midpoint Circle Algorithm

Circles have the property of being highly symmetrical, which is handy when it comes to drawing them on a display screen.

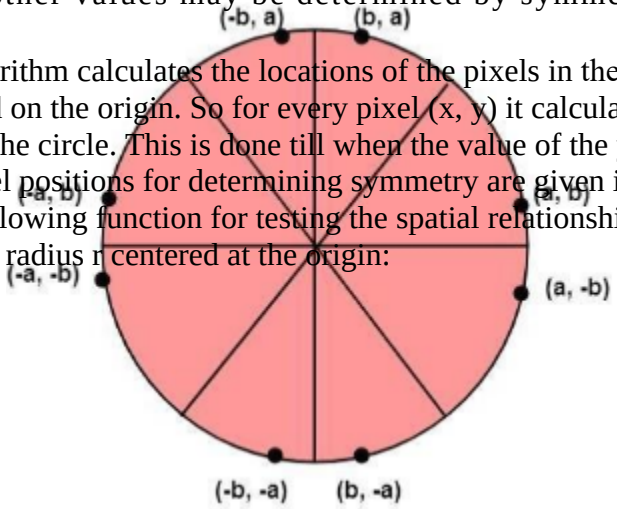
· We know that there are 360 degrees in a circle. First we see that a circle is symmetrical about the x axis, so only the first 180 degrees need to be calculated.

· Next we see that it's also symmetrical about the y axis, so now we

only need to calculate the first 90 degrees.

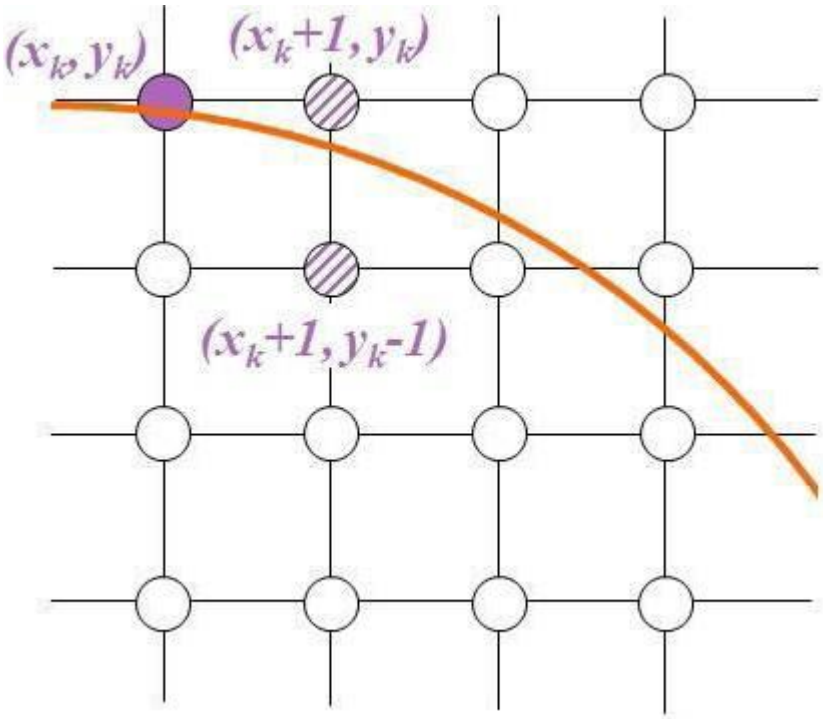
· Finally we see that the circle is also symmetrical about the 45 degree diagonal axis, so we only need to calculate the first 45 degrees.

· We only need to calculate the values on the border of the circle in the first octant. The other values may be determined by symmetry

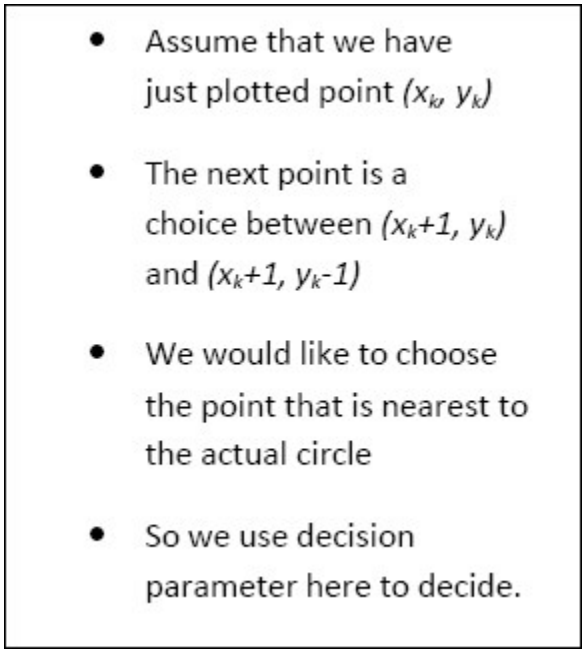


Midpoint circle algorithm calculates the locations of the pixels in thefirst 45 degrees. It assumes that the circle is centered on the origin. So for every pixel (x, y) it calculates, we draw a pixel in each of the eight octants of the circle. This is done till when the value of the y coordinate equals the x coordinate. The pixel positions for determining symmetry are given in the below 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:



.....equation 1



Algorithm :

Plot the initial (x_k , y_k) i.e. $x_k = 0$ and $y_k = r$
$x_{k+1} = x_k + 1$
$y_{k+1} = y_k$
If $P_k >= 0$, then choose $y_{k+1} = y_k$
Repeat 3 and 4 until x becomes greater than or equal to y
To plot the entire circle, use the 8-way symmetry

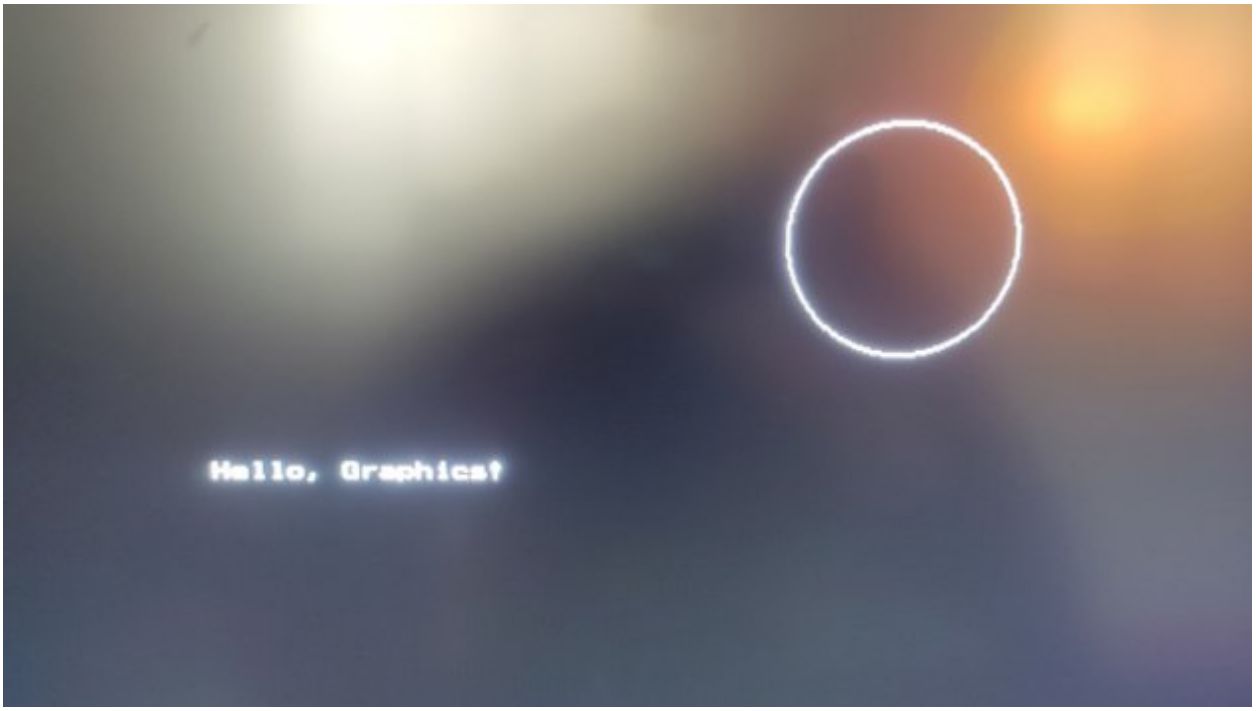
$$P_{k+1} = P_k + 2x_k + 3$$

Code :

```
#include <stdio.h>
#include <graphics.h>
void drawCircleMidpoint(int xc, int yc, int radius) {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    int x = radius;
    int y = 0;
    int p = 1 - radius;
    putpixel(x + xc, y + yc, WHITE);
    if (radius > 0) {
        putpixel(x + xc, -y + yc, WHITE);
        putpixel(y + xc, x + yc, WHITE);
        putpixel(-y + xc, x + yc, WHITE);
    }
    while (x > y) {
        y++;
        if (p <= 0)
            p = p + 2 * y + 1;
        else {
            x--;
            p = p + 2 * y - 2 * x + 1;
        }
        if (x < y)
            break;
        putpixel(x + xc, y + yc, WHITE);
        putpixel(-x + xc, y + yc, WHITE);
        putpixel(x + xc, -y + yc, WHITE);
        putpixel(-x + xc, -y + yc, WHITE);
        if (x != y) {
            putpixel(y + xc, x + yc, WHITE);
            putpixel(-y + xc, x + yc, WHITE);
            putpixel(y + xc, -x + yc, WHITE);
            putpixel(-y + xc, -x + yc, WHITE);
        }
    }
```

```
    delay(5000);
    closegraph();
}
int main() {
    int xc, yc, radius;
    printf("Enter the center of the circle (xc, yc): ");
    scanf("%d %d", &xc, &yc);
    printf("Enter the radius of the circle: ");
    scanf("%d", &radius);
    drawCircleMidpoint(xc, yc, radius);
    return 0;
}
```

Output:



Conclusion:

The Midpoint Circle Drawing Algorithm is a simple and efficient algorithm for drawing circles. It is based on the concept of symmetry and reduces the computational load by taking advantage of the symmetry properties of a circle. The algorithm uses integer arithmetic and avoids the need for expensive floating-point operations.

One notable advantage of the Midpoint Circle Drawing Algorithm is its efficiency in terms of both time and space. It generates circle points using only integer addition and subtraction operations, making it suitable for systems with limited computational resources.

However, it's important to note that this algorithm is specifically designed for integer coordinates and may not produce accurate results for circles with very large radii. In such cases, other algorithms, such as the Bresenham Circle Algorithm, might be preferred. Overall, the Midpoint Circle Drawing Algorithm is a fundamental and widely used technique in computer graphics for rendering circular shapes efficiently.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	4
Title:	Midpoint Ellipse Drawing Algorithm
Date of	
Marks:	
Sign of Faculty:	

Experiment No. 4

Aim :Write a program to implement Midpoint Ellipse Drawing Algorithm in C.

Objective : To implement midpoint ellipse drawing algorithm for drawing anellipse with radii rx and ry.

Midpoint EllipseAlgorithm

This is an incremental method for scan converting an ellipse that is centered at the origin in standard position i.e., with the major and minoraxis parallel to coordinate system axis. It is very similar to the midpointcircle algorithm. Because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant.
Let's first rewrite the ellipse equation and define the function f that can be used to decide

if the midpoint between two candidate pixels is inside or outside the ellipse:

Now divide the elliptical curve from (0, b) to (a, 0) into two parts atpoint Q where the slope of the curve is -1.

Slope of the curve is defined by the f(x, y) = 0 is $\frac{dy}{dx} = -\frac{fx}{fy}$ where fx & fy are partial derivatives of f(x, y) with respect to x & y.

We have fx = 2b2 x, fy=2a2 y & $\frac{dy}{dx} = -\frac{2b^2x}{2a^2y}$ Hence we can monitor the slope value during the scan conversion process to detect Q. Our starting point is (0, b)
Suppose that the coordinates of the last scan converted pixel uponenteringstep i are(xi,yi). We areto select either T (xi+1),yi) or S (xi+1,yi-1)to be the next pixel. The midpoint of T & S is used to define the following decision parameter.

pi = f(xi+1),yi-1/2)

pi=b2 (xi+1)2+a2 (yi-1/2)2-a2 b2
If pi<0, the midpoint is inside the curve and we choose pixel T.
If pi>0, the midpoint is outside or on the curve and we choose pixel S.

Decision parameter for the next step is:

$p_{i+1}=f(x_{i+1}+1,y_{i+1}-1/2)$

$= b^2 (x_{i+1}+1)^2+a^2 (y_{i+1}-1/2)^2-a^2 b^2$

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

$p_{i+1}-pi=b^2[((x_{i+1}+1)^2+a^2 (y_{i+1}-1/2)^2-(y_i -1/2)^2]$

$p_{i+1}= pi+2b^2 x_{i+1}+b^2+a^2 [(y_{i+1}-1/2)^2-(y_i -1/2)^2]$

If T is chosen pixel (pi<0), we have $y_{i+1}=yi$.
If S is chosen pixel (pi>0) we have $y_{i+1}=yi-1$. Thus we can express p_{i+1}in terms of pi and (x_{i+1},y_{i+1}):

$p_{i+1}= pi+2b^2 x_{i+1}+b^2$ if pi<0
$= pi+2b^2 x_{i+1}+b^2-2a^2 y_{i+1}$ if pi>0

The initial value for the recursive expression can be obtained by the evaluating the original definition of pi with (0, b):

$p1 = (b^2+a^2 (b-1/2)^2-a^2 b^2$
$= b^2-a^2 b+a^2/4$

Suppose the pixel $(x_j\ y_j)$ has just been scan converted upon entering step j. The next pixel is either U (x_j ,y_j-1) or V (x_j+1,y_j-1). The midpointof the horizontal line connecting U & V is used to define the decision parameter: $q_j=f(x_j+1/2,y_j-1)$

$q_j=b^2 (x_j+1/2)^2+a^2 (y_j -1)^2-a^2 b^2$

If $q_j<0$, the midpoint is inside the curve and we choose pixel V.
If $q_j \geq 0$, the midpoint is outside the curve and we choose pixel U. Decision parameter for the next step is:

$q_{j+1}=f(x_{j+1}+1/2,y_{j+1}-1)$

$$f(x,y) = b^2x^2 + a^2y^2 - a^2b^2 = \begin{bmatrix} < 0 (x,y) \text{inside} \\ o\ (x,y) \text{on} \\ > 0\ (x,y) \text{outside} \end{bmatrix}$$

$= b^2 (x_{j+1}+1/2)^2+ a^2 (y_{j+1}-1)^2- a^2 b^2$

Since $y_{j+1}=y_j-1$,we have

$q_{j+1}-q_j=b^2 [(x_{j+1}+1/2)^2-(x_j +1/2)^2]+a^2 (y_{j+1}-1)^2-(y_{j+1})^2]$

$q_{j+1}=q_j+b^2 [(x_{j+1}+1/2)^2-(x_j +1/2)^2]-2a^2 y_{j+1}+a^2$

If V is chosen pixel ($q_j<0$), we have $x_{j+1}=x_j$.

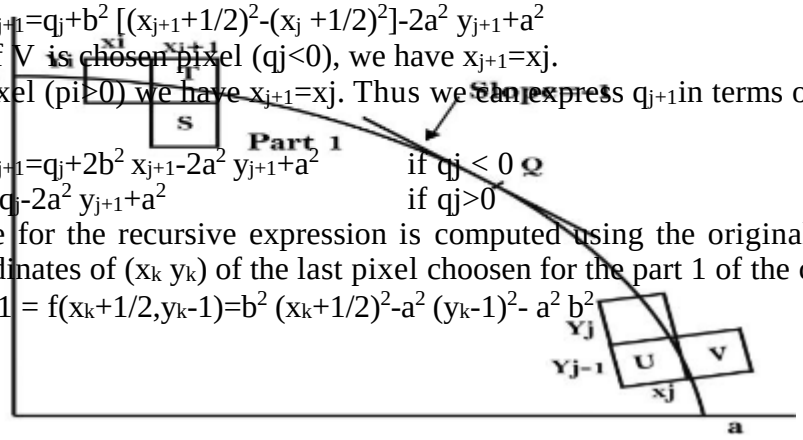
If U is chosen pixel ($p_i\geq 0$) we have $x_{j+1}=x_j$. Thus we can express q_{j+1} in terms of q_j and (x_{j+1},y_{j+1}):

$q_{j+1}=q_j+2b^2 x_{j+1}-2a^2 y_{j+1}+a^2$ if $q_j < 0$

$=q_j-2a^2 y_{j+1}+a^2$ if $q_j>0$

The initial value for the recursive expression is computed using the original definition of qj. And the coordinates of $(x_k\ y_k)$ of the last pixel choosen for the part 1 of the curve:

$q1 = f(x_k+1/2,y_k-1)=b^2 (x_k+1/2)^2-a^2 (y_k-1)^2- a^2 b^2$



Algorithm:

int x=0, y=b; [starting point]
int fx=2x□2, fy=2□2y [initial partial derivatives]
int p = □2-□2 b+□2/4 while (fx<fy); Setpixel (x, y);

if (p<0) fx=fx+2□2;
p = p + fx +□2; else
{
y--; fx=fx+2□2; fy=fy-2□2;
p = p + fx +□2-fy; }
x++;

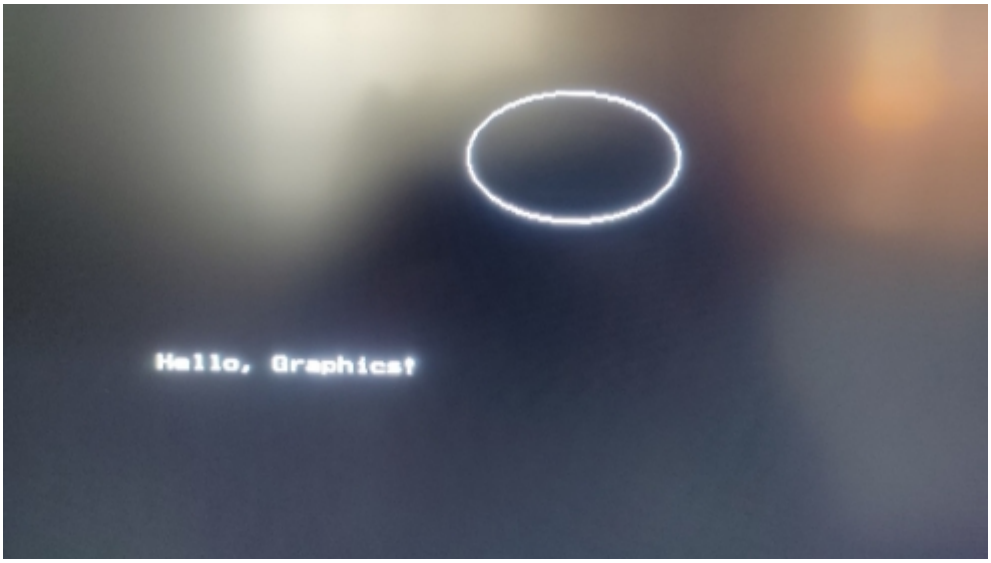
p=□2 (x+0.5)2+ □2 (y-1)2- □2 □2 while (y>0)
Setpixel (x, y); {
y--;
fy=fy-2□2; if (p>=0) p=p-fy+□2 else
{
y--; x++;
fx=fx+2□2 ; fy=fy-2□2; p=p+fx-fy+□2;} }

Code :

```c
#include <stdio.h>
#include <graphics.h>
void drawEllipseMidpoint(int xc, int yc, int rx, int ry) {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    int x = 0, y = ry;
    int rx_sq = rx * rx;
    int ry_sq = ry * ry;
    int two_rx_sq = 2 * rx_sq;
    int two_ry_sq = 2 * ry_sq;
    int p;
    int px = 0, py = two_rx_sq * y;
    putpixel(xc + x, yc - y, WHITE);
    putpixel(xc - x, yc - y, WHITE);
    putpixel(xc + x, yc + y, WHITE);
    putpixel(xc - x, yc + y, WHITE);
    p = round(ry_sq - (rx_sq * ry) + (0.25 * rx_sq));
    while (px < py) {
        x++;
        px += two_ry_sq;
        if (p < 0)
            p += ry_sq + px;
        else {
            y--;
            py -= two_rx_sq;
            p += ry_sq + px - py;
        }
        putpixel(xc + x, yc - y, WHITE);
        putpixel(xc - x, yc - y, WHITE);
        putpixel(xc + x, yc + y, WHITE);
```

```
        putpixel(xc - x, yc + y, WHITE);
    }
    p = round(ry_sq * (x + 0.5) * (x + 0.5) + rx_sq * (y - 1) * (y - 1) - rx_sq *
ry_sq);
    while (y > 0) {
        y--;
        py -= two_rx_sq;
        if (p > 0)
            p += rx_sq - py;
        else {
            x++;
            px += two_ry_sq;
            p += rx_sq - py + px;
        }
        putpixel(xc + x, yc - y, WHITE);
        putpixel(xc - x, yc - y, WHITE);
        putpixel(xc + x, yc + y, WHITE);
        putpixel(xc - x, yc + y, WHITE);
    }
    delay(5000);
    closegraph();
}
int main() {
    int xc, yc, rx, ry;
    printf("Enter the center of the ellipse (xc, yc): ");
    scanf("%d %d", &xc, &yc);
    printf("Enter the major and minor radii (rx, ry): ");
    scanf("%d %d", &rx, &ry);
    drawEllipseMidpoint(xc, yc, rx, ry);
    return 0;
}
```

Output



Conclusion :

The Midpoint Ellipse Drawing Algorithm is an efficient method for drawing ellipses, providing a balance between accuracy and computational complexity. The algorithm is based on selecting points along the ellipse using a decision parameter and updating it at each step to determine the next pixel to be plotted.

One notable advantage of the Midpoint Ellipse Drawing Algorithm is its ability to handle ellipses of various sizes and orientations without significant modifications. It is also relatively efficient compared to more straightforward algorithms, especially when dealing with large ellipses.

However, like many algorithms, the Midpoint Ellipse Drawing Algorithm has its limitations. It may not provide pixel-perfect results for all ellipses, especially when dealing with extreme aspect ratios. Additionally, the algorithm relies on floating-point arithmetic, which may be computationally expensive on systems with limited resources.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	5
Title:	Boundary Fill and Flood Fill Polygon filling Algorithm
Marks:	
Sign of Faculty:	

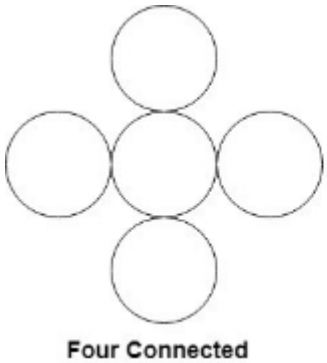
Experiment No. 5

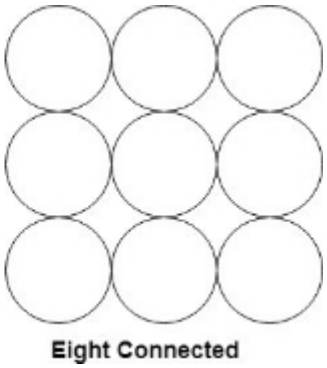
Aim :Write a program to implement Boundary Fill and Flood Fill Polygon filling Algorithm in C.

Obective : **Boundary Fill Algorithm:**
This algorithm uses the recursive method. First of all, a starting pixel called the seed is considered. The algorithm checks whether boundary pixels or adjacent pixels are colored or not. If the adjacent pixel is already filled or colored then leave it, otherwise fill it. The filling is done using four connected or eight connected approaches.

1. Four connected approaches: In this approach, left, right, above, below pixels are tested.
2. Eight connected approaches: In this approach, left, right, above, below and four diagonals are selected.
Boundary can be checked by seeing pixels from left and right first. Then pixels are checked by seeing pixels from top to bottom. The algorithm takes time and memory because some recursive calls are needed.

Flood Fill Algorithm:
In this method, a point or seed which is inside region is selected. This point is called a seed point. Then four connected approaches or eight connected approaches is used to fill with specified color.
The flood fill algorithm has many characters similar to boundary fill. But this method is more suitable for filling multiple colors boundary. When boundary is of many colors and interior is to be filled with one color we use this algorithm.
In fill algorithm, we start from a specified interior point (x, y) and reassign all pixel values are currently set to a given interior color with the desired color. Using either a 4-connected or 8-connected approaches, we then step through pixel positions until all interior points have been repainted.


Four Connected


Eight Connected

Algorithm:

Boundary Fill

boundary_fill_8(x, y, fill_color, boundary_color) int current;
current=getpixel (x, y);
if ((current != fill_color) && (current != boundary_color)) {
setpixel (x, y, fill_color)
boundary_fill_8(x+1, y, fill_color , boundary_color); boundary_fill_8(x-1, y, fill_color , boundary_color); boundary_fill_8(x, y+1, fill_color , boundary_color); boundary_fill_8(x, y-1, fill_color , boundary_color); boundary_fill_8(x+1, y+1, fill_color , boundary_color); boundary_fill_8(x+1, y-1, fill_color , boundary_color); boundary_fill_8(x-1, y+1, fill_color , boundary_color); boundary_fill_8(x-1, y-1, fill_color , boundary_color); }

Flood Fill

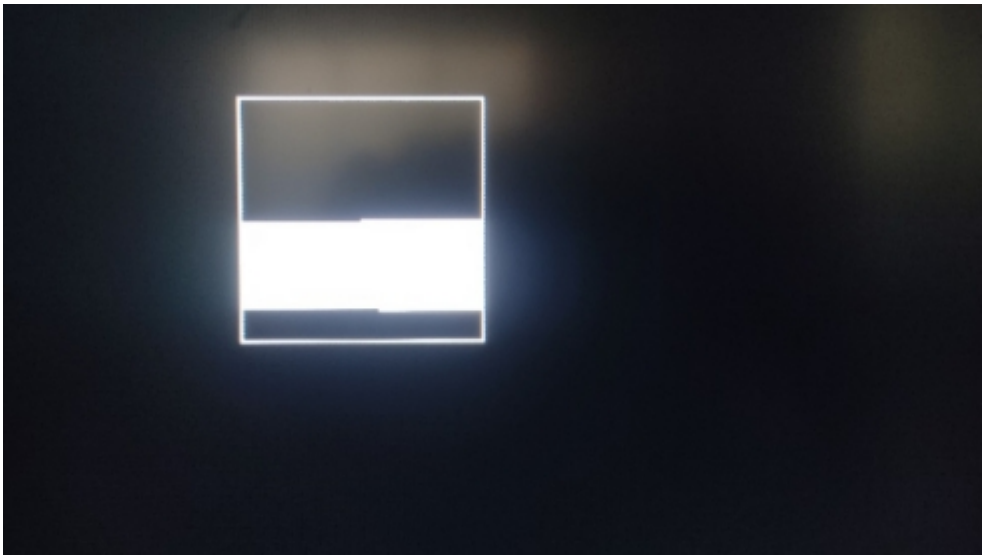
flood_fill_8(x, y, fill_color, default_color)
{
int current; current=getpixel (x, y);
if (current == default_color) {
setpixel (x, y, fill_color)
flood_fill_8(x+1, y, fill_color , default_color); flood_fill_8(x-1, y, fill_color , default_color); flood_fill_8(x, y+1, fill_color , default_color); flood_fill_8(x, y-1, fill_color , default_color); flood_fill_8(x+1, y+1, fill_color , default_color); flood_fill_8(x-1, y+1, fill_color , default_color); flood_fill_8(x+1, y-1, fill_color , default_color); flood_fill_8(x-1, y-1, fill_color , default_color); }
}

Code :

```
#include <stdio.h>
#include <graphics.h>
void boundaryFill(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);
        boundaryFill(x + 1, y, fill_color, boundary_color);
        boundaryFill(x, y + 1, fill_color, boundary_color);
        boundaryFill(x - 1, y, fill_color, boundary_color);
        boundaryFill(x, y - 1, fill_color, boundary_color);
    }
}
void floodFill(int x, int y, int old_color, int new_color) {
    if (getpixel(x, y) == old_color) {
        putpixel(x, y, new_color);
        floodFill(x + 1, y, old_color, new_color);
        floodFill(x, y + 1, old_color, new_color);
        floodFill(x - 1, y, old_color, new_color);
        floodFill(x, y - 1, old_color, new_color);
    }
}
int main() {
    int gd = DETECT, gm;
```

```
    initgraph(&gd, &gm, NULL);
    int poly[] = {100, 100, 300, 100, 200, 300, 100, 100};
    drawpoly(4, poly);
    boundaryFill(200, 200, 6, 15);
    delay(2000);
    cleardevice();
    drawpoly(4, poly);
    floodFill(200, 200, 0, 6);
    delay(2000);
    closegraph();
    return 0;
}
```

Output:



Conclusion:

The Boundary Fill and Flood Fill algorithms are fundamental techniques for filling closed areas in computer graphics.

Boundary Fill:

- Pros: It is easy to implement and understand.
- Cons: It may face issues with performance and efficiency, especially for complex shapes. The algorithm might fill areas that are not intended to be filled due to the similarity of colors.

Flood Fill:

- Pros: It is generally more efficient than Boundary Fill for filling large areas.
- Cons: The recursive nature of the algorithm may lead to a stack overflow for large areas. It can also fill areas that are not enclosed by the boundary.
-

Both algorithms have their strengths and weaknesses, and the choice between them depends on the specific requirements of the application. In practice, more advanced algorithms like scanline fill algorithms or seed fill algorithms are often used for efficient and accurate polygon filling in computer graphics.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	6
Title:	Implement 2D Transformation on an object.
Marks:	
Sign of Faculty:	

Experiment No. 6

Aim :Write a program to implement 2D Transformation on an object -Translation, Rotation, Reflection, Scaling and Shear in C.

Objective :

Description: We have to perform 2D transformations on 2D objects. Here we perform transformations on a line segment. The 2D transformations are:

1. Translation
2. Scaling
3. Rotation
4. Reflection
5. Shear

1. Translation: Translation is defined as moving the object from one position to another position along straight line path.

We can move the objects based on translation distances along x and y axis. tx denotes translation distance along x-axis and ty denotes translation distance along y axis.

Translation Distance: It is nothing but by how much units we should shift the object from one location to another along x, y-axis.

Consider (x,y) are old coordinates of a point. Then the new coordinatesof that same point (x',y') can be obtained as follows:

X'=x+tx

Y'=y+ty

2. **Scaling**: scaling refers to changing the size of the object either by increasing or decreasing. We will increase or decrease the size of the object based on scaling factors along x and y-axis.
If (x, y) are old coordinates of object, then new coordinates of object after applying scaling transformation are obtained as:
x'=x*sx y'=y*sy.
sx and sy are scaling factors along x-axis and y-axis.

3. Rotation: A rotation repositions all points in an object along a circular path in the plane centered at the pivot point. We rotate an objectby an angle theta.
New coordinates after rotation depend on *both x* and y • $x' = x\cos\theta - y\sin\theta$

• $y' = x\sin\theta + y\cos\theta$

4. Reflection: Reflection is nothing but producing mirror image of an object. Reflection can be done just by rotating the object about given axis of reflection with an angle of 180 degrees.

5. Shear:
1. Shear is the translation along an axis by an amount that increases linearly with another axis (**Y**). It produces shape distortions as if objects were composed of layers that are caused to slide over each other.
2. Shear transformations are very useful in creating italic letters and slanted letters from regular letters.
3. Shear transformation changes the shape of the object to a slant position.
4. Shear transformation is of 2 types:
a. X-shear: changing x-coordinate value and keeping y constant x'=x+shx*y
y'=y
b. Y-shear: changing y coordinates value and keeping x constant x'=x
y'=y+shy*x shx and shy are shear factors along x and y-axis.

Algorithm :

1. Start

2. Read the coordinate values to draw the 2D figure and perform the following steps to achieve the necessary transformations
3. Translation : Read the translation vector tx and ty and compute the new position using x1 = x + tx , y1 = y + ty and display the new position
4. Rotation : Read the rotation angle Θ and compute the new pos using the formula x1 = x Cos Θ –

y Sin Θ , y1 = x Sin Θ + y Cos Θ and display the new position

5. Scaling : Read the scaling factor sx and sy and compute the newposition using x1 = x. sx , y1 = y. sy and display the new position

6. Shearing : Shearing is done in x direction and y direction using the factor shx and shy for X direction: x1 = x + shx (y – yref), y1 = y & forY direction x1 = x , y1 = shy (x – xref)

7. Reflection : Reflection depends upon the reflection axis . If reflection axis is y = x , then x1 = y and y1 = x
8. Stop

Code :

```
#include <stdio.h>
#include <graphics.h>
#include <math.h>
void drawObject(int x[], int y[], int edges) {
    for (int i = 0; i < edges - 1; i++) {
        line(x[i], y[i], x[i + 1], y[i + 1]);
    }
    line(x[edges - 1], y[edges - 1], x[0], y[0]);
}
void translate(int x[], int y[], int edges, int tx, int ty) {
    for (int i = 0; i < edges; i++) {
        x[i] = x[i] + tx;
        y[i] = y[i] + ty;
    }
}
void rotate(int x[], int y[], int edges, float angle) {
    float radian = angle * (M_PI / 180.0);
    int cx = 0, cy = 0;
    for (int i = 0; i < edges; i++) {
        int tempX = x[i];
        x[i] = cx + (x[i] - cx) * cos(radian) - (y[i] - cy) * sin(radian);
        y[i] = cy + (tempX - cx) * sin(radian) + (y[i] - cy) * cos(radian);
    }
}
void reflect(int x[], int y[], int edges, int reflectionAxis) {
    for (int i = 0; i < edges; i++) {
        if (reflectionAxis == 0)
            y[i] = -y[i];
```

```
        else if (reflectionAxis == 1)
            x[i] = -x[i];
    }
}
void scale(int x[], int y[], int edges, float sx, float sy) {
    for (int i = 0; i < edges; i++) {
        x[i] = round(x[i] * sx);
        y[i] = round(y[i] * sy);
    }
}
void shear(int x[], int y[], int edges, float shx, float shy) {
    for (int i = 0; i < edges; i++) {
        x[i] = round(x[i] + shx * y[i]);
        y[i] = round(y[i] + shy * x[i]);
    }
}
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    int x[] = {50, 100, 100, 50, 50};
    int y[] = {50, 50, 100, 100, 50};
    int edges = sizeof(x) / sizeof(x[0]);
    drawObject(x, y, edges);
    translate(x, y, edges, 50, 50);
    drawObject(x, y, edges);
    rotate(x, y, edges, 45);
    drawObject(x, y, edges);
    reflect(x, y, edges, 1);
    drawObject(x, y, edges);
    scale(x, y, edges, 1.5, 1.5);
    drawObject(x, y, edges);
    shear(x, y, edges, 0.5, 0);
    drawObject(x, y, edges);
    getch();
    closegraph();
    return 0;
}
```

Output:

Conclusion:

The 2D transformation algorithms implemented in this program provide a foundation for manipulating objects in computer graphics. Each transformation serves a specific purpose:

- Translation: Moves the object by a specified distance in the x and y directions.
- Rotation: Rotates the object by a given angle around the origin.
- Reflection: Mirrors the object with respect to the X-axis or Y-axis.
- Scaling: Enlarges or shrinks the object by scaling its coordinates.
- Shearing: Skews the object by displacing its points along one of the axes.

These transformations are essential for creating dynamic and visually appealing graphics in applications such as computer-aided design, gaming, and simulations. While the provided program demonstrates the transformations individually, combining multiple transformations can produce more complex effects. Understanding and implementing 2D transformations are fundamental skills in computer graphics and are often extended to 3D transformations in more advanced graphics applications.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	7
Title:	Cohen Sutherland Line Clipping Algorithm
Marks:	
Sign of Faculty:	

Experiment No.7

Aim :Write a program to implement Cohen Sutherland Line Clipping Algorithm in C.

Objective :To implement Cohen Sutherland Line Clipping Algorithm for clipping a line x1,y1 and x2, y2 with respect to a clipping window Xmin, Xmax, Ymin, Ymax.

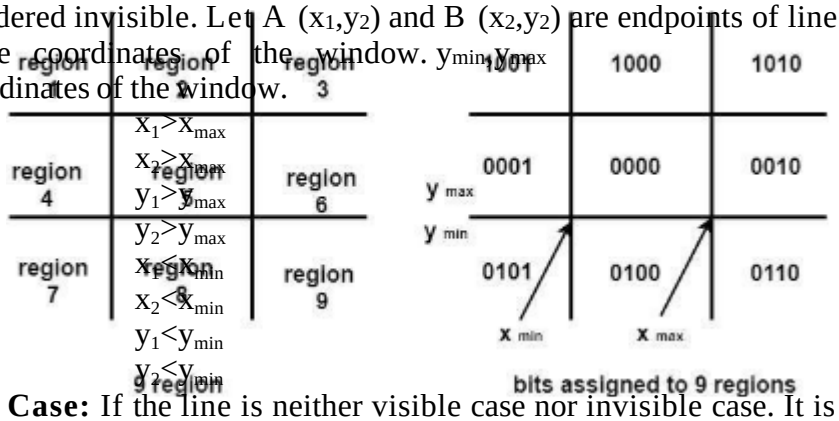
Cohen Sutherland Line Clipping Algorithm:
In the algorithm, first of all, it is detected whether line lies inside the screen or it is outside the screen. All lines come under any one of the following categories:

1. Visible
2. Not Visible
3. Clipping Case

1. Visible: If a line lies within the window, i.e., both endpoints of the line lies within the window. A line is visible and will be displayed as it is.

2. Not Visible: If a line lies outside the window it will be invisible and rejected. Such lines will not display. If any one of the following inequalities is satisfied, then the line is considered invisible. Let A (x_1,y_2) and B (x_2,y_2) are endpoints of line. x_{min},x_{max} are coordinates of the window. y_{min},y_{max} are also coordinates of the window.

$x_1>x_{max}$
$x_2>x_{max}$
$y_1>y_{max}$
$y_2>y_{max}$
$x_1<x_{min}$
$x_2<x_{min}$
$y_1<y_{min}$
$y_2<y_{min}$



3. Clipping Case: If the line is neither visible case nor invisible case. It is considered to be clipped case. First of all, the category of a line is found based on nine regions given below. All nine regions are assigned codes. Each code is of 4 bits. If both endpoints of the line have end bits zero, then the line is considered to be visible.

The center area is having the code, 0000, i.e., region 5 is considered a rectangle window.

Conditions <u>Case I:</u>

Bitwise OR of region of 2 endpoints of line = 0000 → Line lies inside the window

<u>Case II :</u>

1. Completely invisible Bitwise OR ≠ 0000
1. Partially Visible Bitwise AND ≠

0000

Completely invisible **<u>Case III :</u>**

Partially Visible

1. Choose the endpoints of the line.
2. Find the intersection point with the window.
3. Replace the endpoints with the intersection point.

Algorithm :

Step 1: Assign a region code for each end point.

Step 2: Perform Bitwise OR =
0000. Accept line & draw

Step 3: Perform Bitwise AND
If result ≠ 0000 then reject the line.
else clip the line
→ Select the endpoints.
→ Find the intersection point at the window.
→ Replace endpoints with the intersection pt & update the region code.

Code :

```c
#include <stdio.h>
#include <graphics.h>
#define INSIDE 0
#define LEFT 1
#define RIGHT 2
#define BOTTOM 4
#define TOP 8
#define X_MIN 50
#define X_MAX 300
#define Y_MIN 50
#define Y_MAX 200
int computeCode(int x, int y) {
    int code = INSIDE;
    if (x < X_MIN)
        code |= LEFT;
    else if (x > X_MAX)
        code |= RIGHT;
    if (y < Y_MIN)
        code |= BOTTOM;
    else if (y > Y_MAX)
        code |= TOP;
    return code;
}
```

```
void cohenSutherland(int x1, int y1, int x2, int y2) {
    int code1, code2, code;
    int accept = 0, done = 0;
    while (!done) {
        code1 = computeCode(x1, y1);
        code2 = computeCode(x2, y2);
        if (!(code1 | code2)) {
            accept = 1;
            done = 1;
        } else if (code1 & code2) {
            done = 1;
        } else {
            int x, y;
            code = code1 ? code1 : code2;
            if (code & TOP) {
                x = x1 + (x2 - x1) * (Y_MAX - y1) / (y2 - y1);
                y = Y_MAX;
            } else if (code & BOTTOM) {
                x = x1 + (x2 - x1) * (Y_MIN - y1) / (y2 - y1);
                y = Y_MIN;
            } else if (code & RIGHT) {
                y = y1 + (y2 - y1) * (X_MAX - x1) / (x2 - x1);
                x = X_MAX;
            } else if (code & LEFT) {
                y = y1 + (y2 - y1) * (X_MIN - x1) / (x2 - x1);
                x = X_MIN;
            }
            if (code == code1) {
                x1 = x;
                y1 = y;
            } else {
                x2 = x;
                y2 = y;
            }
        }
    }
    if (accept) {
        int gd = DETECT, gm;
        initgraph(&gd, &gm, NULL);
        setcolor(WHITE);
        rectangle(X_MIN, Y_MIN, X_MAX, Y_MAX);
        setcolor(YELLOW);
        line(x1, y1, x2, y2);
        delay(5000);
        closegraph();
    }
}
```

```c
int main() {
    int x1, y1, x2, y2;
    printf("Enter the starting point (x1, y1): ");
    scanf("%d %d", &x1, &y1);
    printf("Enter the ending point (x2, y2): ");
    scanf("%d %d", &x2, &y2);
    cohenSutherland(x1, y1, x2, y2);
    return 0;
}
```

Output :

Conclusion :

The Cohen-Sutherland Line Clipping Algorithm provides a simple and efficient method for clipping line segments against a rectangular window. It uses region codes to quickly determine whether an endpoint is inside or outside the window and then applies appropriate clipping operations to obtain the visible portion of the line.

This algorithm is particularly useful in computer graphics for optimizing the rendering process by eliminating the portions of lines that lie outside the viewing area. While the Cohen-Sutherland algorithm is effective for clipping against axis-aligned rectangles, it may not perform as well for more complex clipping scenarios. In such cases, more advanced algorithms like the Liang-Barsky algorithm or the Cyrus-Beck algorithm might be considered.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	8
Title:	Sutherland Hodgeman Polygon Clipping Algorithm
Marks:	
Sign of Faculty:	

Experiment No.8

Aim :Write a program to implement Sutherland Hodgeman Polygon Clipping Algorithm in C.

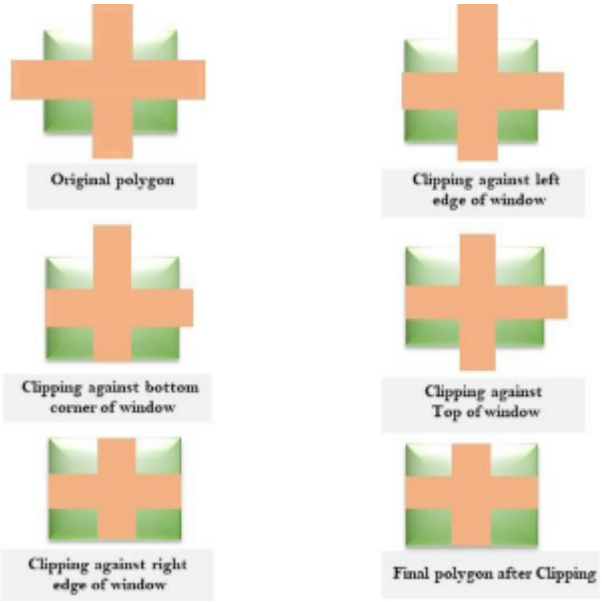
Objective :To implement Sutherland Hodgeman Polygon Clipping algorithm for clipping a subject polygon with respect to a clip polygon.

Sutherland-Hodgeman Polygon Clipping:
It is performed by processing the boundary of polygon against each window corner or edge. First of all entire polygon is clipped against one edge, then resulting polygon is considered, then the polygon is considered against the second edge, so on for all four edges.

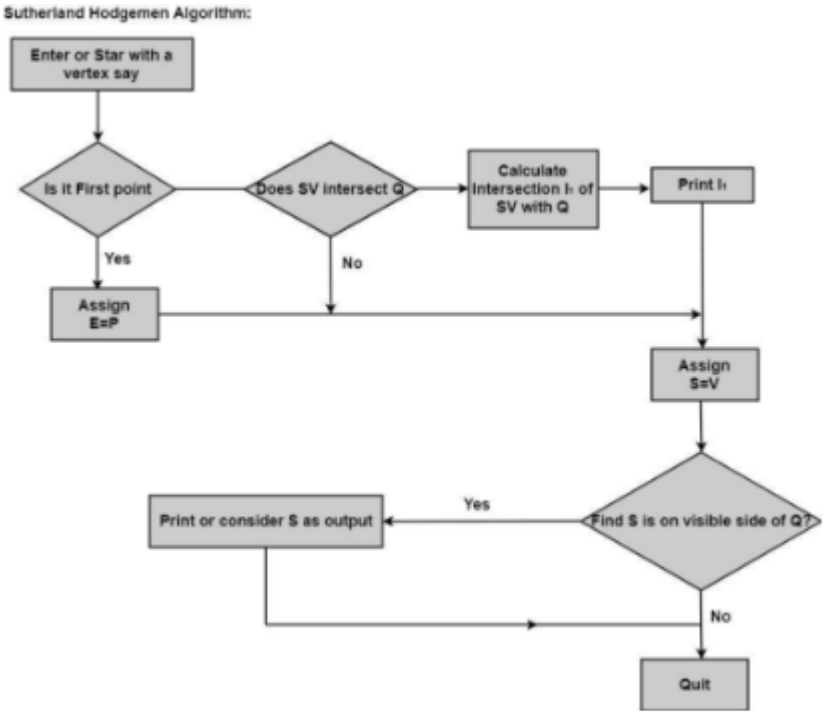
Four possible situations while processing
If the first vertex is an outside the window, the second vertex is inside the window. Then second vertex is added to the output list. The point of intersection of window boundary and polygon side (edge) is also added to the output line.

If both vertexes are inside window boundary. Then only second vertex is added to the output list.
If the first vertex is inside the window and second is an outside window. The edge which intersects with window is added to output list.
If both vertices are the outside window, then nothing is added to output list.

Following figures shows original polygon and clipping of polygon against four windows.



Algorithm:



Code :

```
#include <stdio.h>
#include <graphics.h>
int subjectPolygon[][2] = {{50, 150}, {150, 150}, {150, 50}, {50, 50}};
int clipPolygon[][2] = {{100, 100}, {200, 100}, {200, 0}, {100, 0}};
int inside(int p[2], int clipEdge[2][2], int edge) {
    return (clipEdge[1][0] - clipEdge[0][0]) * (p[1] - clipEdge[0][1]) >
           (clipEdge[1][1] - clipEdge[0][1]) * (p[0] - clipEdge[0][0]);
}
void computeIntersection(int p1[2], int p2[2], int clipEdge[2][2], int edge, int
intersection[2]) {
    int dx = p2[0] - p1[0];
    int dy = p2[1] - p1[1];
    int numerator = (clipEdge[0][1] - p1[1]) * dx - (clipEdge[0][0] - p1[0]) * dy;
    int denominator = (clipEdge[1][0] - clipEdge[0][0]) * dy - (clipEdge[1][1] -
clipEdge[0][1]) * dx;
    intersection[0] = p1[0] + numerator / denominator;
    intersection[1] = p1[1] + numerator / denominator;
}
void sutherlandHodgman() {
    int outputPolygon[4][2];
    int inputPolygonSize = 4, clipEdgeSize = 4;
    for (int edge = 0; edge < clipEdgeSize; edge++) {
        int inputSize = inputPolygonSize;
        int inputPolygon[inputPolygonSize][2];
        for (int i = 0; i < inputPolygonSize; i++) {
            inputPolygon[i][0] = subjectPolygon[i][0];
            inputPolygon[i][1] = subjectPolygon[i][1];
        }
        int outputSize = 0;
        int clipEdge[2][2] = {{clipPolygon[edge][0], clipPolygon[edge][1]},
                              {clipPolygon[(edge + 1) % clipEdgeSize][0],
clipPolygon[(edge + 1) % clipEdgeSize][1]}};
        for (int i = 0; i < inputSize; i++) {
            int p1[2] = {inputPolygon[i][0], inputPolygon[i][1]};
            int p2[2] = {inputPolygon[(i + 1) % inputSize][0], inputPolygon[(i + 1)
% inputSize][1]};
            if (inside(p2, clipEdge, edge)) {
                if (!inside(p1, clipEdge, edge)) {
                    computeIntersection(p1, p2, clipEdge, edge,
outputPolygon[outputSize]);
                    outputSize++;
                }
                outputPolygon[outputSize][0] = p2[0];
                outputPolygon[outputSize][1] = p2[1];
                outputSize++;
            } else if (inside(p1, clipEdge, edge)) {
```

```
                computeIntersection(p1, p2, clipEdge, edge,
outputPolygon[outputSize]);
                outputSize++;
            }
        }
        inputPolygonSize = outputSize;
        for (int i = 0; i < outputSize; i++) {
            subjectPolygon[i][0] = outputPolygon[i][0];
            subjectPolygon[i][1] = outputPolygon[i][1];
        }
    }
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    setcolor(RED);
    drawpoly(inputPolygonSize, (int *)subjectPolygon);
    setcolor(GREEN);
    drawpoly(clipEdgeSize, (int *)clipPolygon);
    delay(5000);
    closegraph();
}
int main() {
    sutherlandHodgman();
    return 0;
}
```

Output :

Conclusion :

The Sutherland-Hodgman polygon clipping algorithm is a simple and effective method for clipping polygons against an arbitrary convex clip window. The algorithm works by iteratively clipping the polygon against each edge of the clip window.

The implementation involves checking whether each vertex of the polygon is inside or outside the clip window. If a vertex is inside, it is included in the output. If it is outside, the algorithm computes the intersection point of the polygon edge with the clip window edge and includes the intersection point in the output.

This algorithm is particularly useful in computer graphics for removing parts of a polygon that lie outside a specified viewing area, which is common in rendering and display processes. While Sutherland-Hodgman is effective for convex clip windows, it may require modification for handling concave clip windows or more complex cases. Additionally, more advanced algorithms, such as the Cyrus-Beck algorithm or the Weiler-Atherton algorithm, are employed for handling more general cases of polygon clipping.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	9 Bezier Curve for n control points
Title:	
Marks:	
Sign of Faculty:	

Experiment No. 9

Aim :Write a program to implement Bezier Curve for n control points in C.

Objective:

A Bezier curve is a parametric curve used in computer graphics and related fields. The curve, which is related to the Bernstein polynomial, is named after Pierre Bezier, who used it in the 1960s for designing curves for the bodywork of Renault cars.

P(u) = Pi Bi,n(u)

0 ≤ u ≤ 1

where, Pi = control points

Bi,n / BEZ i,n = Bezier function or Berstein Polynomials.

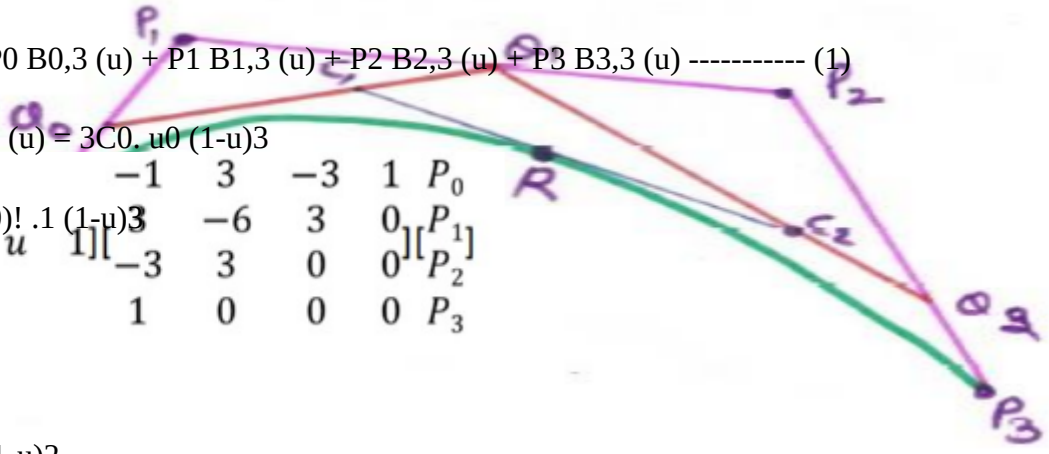
The Bernstein polynomial or the Bezier function is very important function will dictate the smoothness of this curve & the weight will be dictated by boundary conditions.

BEZ i,n (u) = nCi. ui (1-u)n-i

$$\frac{n!}{i!(n-i)!}$$

where nCi = [Binomial Coefficient]

P(u) = Pi Bi,n(u)



P(u) = = P0 B0,3 (u) + P1 B1,3 (u) + P2 B2,3 (u) + P3 B3,3 (u) ---------- (1)

where, B 0,3 (u) = 3C0. u0 (1-u)3

= 3! / 0! (3-0)! .1 (1-u)3

$$[u^3 \quad u^2 \quad u \quad 1][\begin{matrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{matrix}][\begin{matrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{matrix}]$$

= (1-u)3

Similarly,

B1,3 = 3u. (1-u)2

B2,3 = 3u2. (1-u)

B3,3 = u3

Now substituting these value in equation 1 we get

P(u) = P0 (1-u)3 + P1 3u. (1-u)2 + P2 3u2. (1-u) + P3 u3

x(u) = (1-u)3 x0 + 3u(1-u)2 x1 + 3u2(1-u) x2 + u3 x3

y(u) = (1-u)3 y0 + 3u(1-u)2 y1 + 3u2(1-u) y2 + u3 y3

z(u) = (1-u)3 z0 + 3u(1-u)2 z1 + 3u2(1-u) z2 + u3 z3

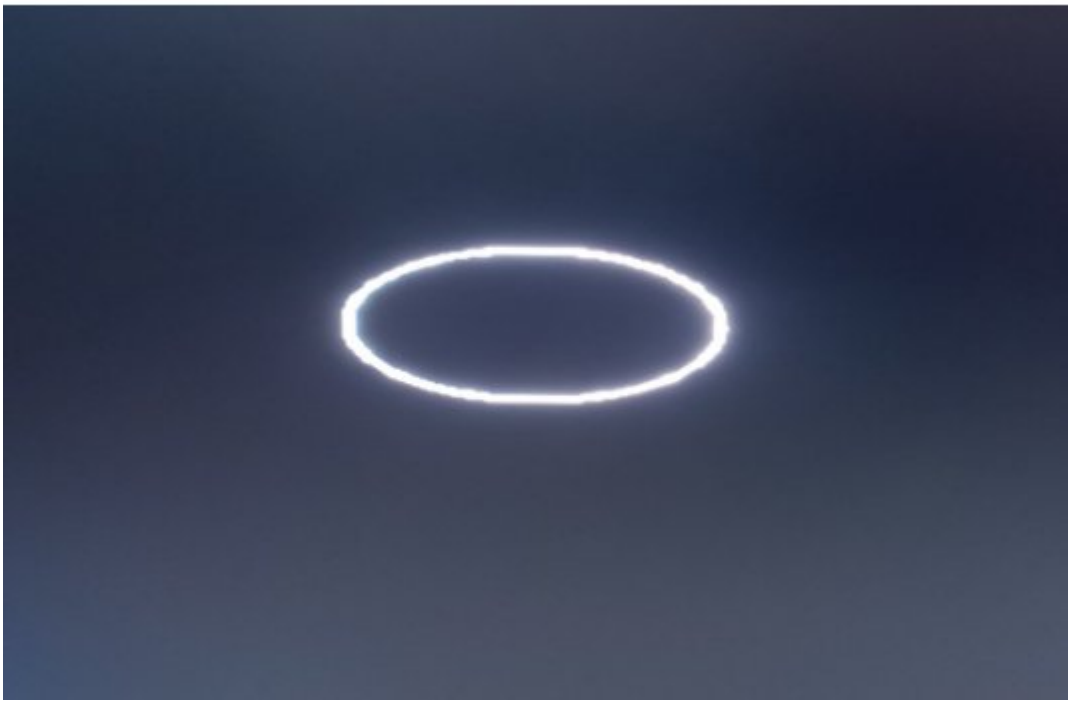
Bezier curve can fit any number of control points Reversing the order of control points yields the same bezier curve The curve begins at P0 and ends at Pn this is the so-called endpoint interpolation property.
The curve is a straight line if and only if all the control points are collinear.

Code :

```
#include <stdio.h>
#include <graphics.h>
int binomialCoefficient(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    } else {
        return binomialCoefficient(n - 1, k - 1) + binomialCoefficient(n - 1, k);
    }
}
float bezierPoint(int n, int i, float t, int *ctrlPoints) {
    return binomialCoefficient(n, i) * pow(1 - t, n - i) * pow(t, i) *
ctrlPoints[i];
}
void drawBezierCurve(int n, int *ctrlPoints) {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);
    float t;
    for (t = 0.0; t <= 1.0; t += 0.01) {
        float x = 0.0, y = 0.0;
        for (int i = 0; i <= n; i++) {
            x += bezierPoint(n, i, t, ctrlPoints);
        }
        t += 0.01;
        for (int i = 0; i <= n; i++) {
            y += bezierPoint(n, i, t, ctrlPoints);
        }
        putpixel(round(x), round(y), WHITE);
    }
    delay(5000);
    closegraph();
}
int main() {
    int n;
    printf("Enter the number of control points: ");
    scanf("%d", &n);
    int ctrlPoints[n + 1];
    printf("Enter the control points:\n");
    for (int i = 0; i <= n; i++) {
        printf("P%d: ", i);
```

```
        scanf("%d", &ctrlPoints[i]);
    }
    drawBezierCurve(n, ctrlPoints);
    return 0;
}
```

Output



Conclusion :

The Bezier curve is a versatile and widely used method for defining smooth curves in computer graphics. This implementation allows users to input an arbitrary number of control points, making it suitable for creating curves of varying complexity.

One key advantage of Bezier curves is their ability to represent a wide range of shapes, from simple curves to more intricate forms. The recursive nature of the binomial coefficient calculation provides an elegant and efficient solution for evaluating points along the curve.

However, it's essential to note that Bezier curves, particularly with a large number of control points, can be computationally expensive. In practice, optimization techniques and more specialized algorithms, such as de Casteljau's algorithm, are often employed to enhance performance.

Despite potential computational complexities, Bezier curves remain a fundamental concept in computer graphics and find applications in areas such as 3D modeling, animation, and font design. Their mathematical elegance and flexibility make them a valuable tool for representing and manipulating curves in digital environments.

Name:	ARPAN MAHADIK
Roll No:	
Class/Sem:	SE/III
Experiment No.:	10
Title:	Program to implement Simple Animation
Date of	

Marks:	
Sign of Faculty:	

Experiment No. 10

Aim :Program to implement Simple Animation (Perform Animation (such as Rising Sun, Moving Vehicle, Smileys, Screen saver etc.) in C

Objective :

To perform a simple animation using graphics.h header file

settextstyle settextstyle function in c
Settextstyle function is used to change the way in which text appears, using it we can modify the size of text, change direction of text and change the font oftext.
Declaration :- void settextstyle(int font, int direction, int charsize); font argument specifies the font of text, Direction can be HORIZ_DIR(Left to right) or VERT_DIR (Bottom to top).

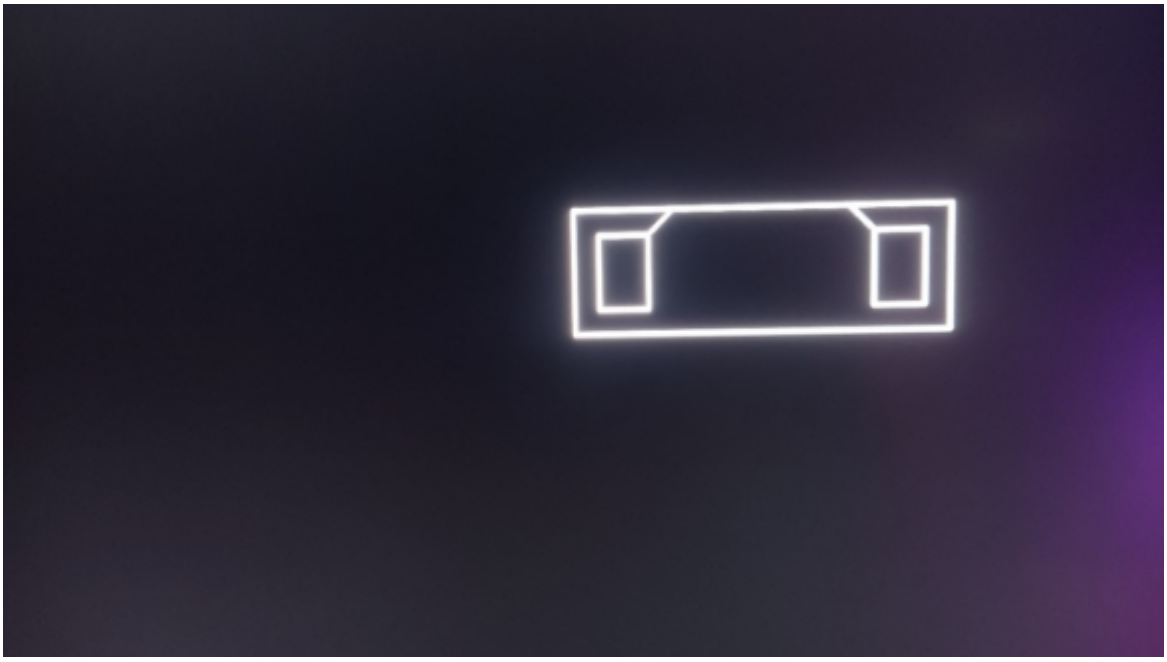
1.Different fonts enum font_names { DEFAULT_FONT, TRIPLEX_FONT, SMALL_FONT, SANS_SERIF_FONT, GOTHIC_FONT, SCRIPT_FONT,
SIMPLEX_FONT, TRIPLEX_SCR_FONT, COMPLEX_FONT, EUROPEAN_FONT,
BOLD_FONT
};

outtextxy
outtextxy function display text or string at a specified point(x,y) on the screen.
Declaration :- void outtextxy(int x, int y, char *string);
x, y are coordinates of the point and third argument contains the address of string to be displayed.

Code :

```c
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void drawCar(int x, int y) {
    rectangle(x, y, x + 150, y + 50);
    rectangle(x + 10, y + 10, x + 30, y + 40);
    rectangle(x + 120, y + 10, x + 140, y + 40);
    line(x + 10, y + 10, x + 30, y + 10);
    line(x + 30, y + 10, x + 40, y);
    line(x + 40, y, x + 110, y);
    line(x + 110, y, x + 120, y + 10);
    line(x + 120, y + 10, x + 140, y + 10);
    line(x, y + 50, x + 150, y + 50);
}
int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\Turboc3\\BGI");
    int x = 0, y = 200;
    for (int i = 0; i <= getmaxx() - 150; i += 5) {
        cleardevice();
        drawCar(x + i, y);
        delay(50);
    }
    getch();
    closegraph();
    return 0;
}
```

Output:



Conclusion :

Creating simple animations in C using graphics libraries provides an introduction to basic computer graphics concepts. In this example, the program uses the graphics.h library to draw and move a basic vehicle on the screen. The animation is achieved by clearing the screen in each iteration of the loop and redrawing the vehicle at a new position.

While this approach provides a basic understanding of animation in a graphical environment, it's important to note that graphics.h is specific to certain compilers and may not be available on all systems. Modern graphics programming often involves using more advanced libraries and frameworks, such as OpenGL, DirectX, or game development engines like Unity or Unreal Engine.

Additionally, the animation in this example is simple, and more complex animations often require sophisticated techniques, such as frame interpolation, sprite sheets, or skeletal animation. Aspiring graphics programmers may explore these advanced concepts and tools to create more intricate and visually appealing animations.