Delhi Technological University

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi - 110042

Department of Computer Science and Engineering

Computer Graphics

Course Code: CO313

Lab Manual

Submitted to: Submitted by:

Mr. Nipun Bansal Prakul Gupta

2K22/CO/332

INDEX

S.NO.	Experiment	Date Sign
1.	Write a program to implement the DDA Line Algorithm	
2.	Write a program to implement the Bresenham's Line Algorithm	
3.	Write a program to implement the Mid-Point Circle Drawing Algorithm	
4.	Write a program to implement the Mid-Point Ellipse Drawing Algorithm.	
5.	Write a program To write a program and implement Flood Fill Algorithm	
6.	Write a program To implement Boundary Fill Algorithm	
7.	Write a program to implement 2D Translation of an object	
8.	Write a program to implement 2D Scaling of an object.	
9.	Write a program to implement 2D Rotation of an object.	
10.	Write a program To implement Cohen-Sutherland Line Clipping Algorithm	
11.	Write a program to implement Liang Barsky Line Clipping algorithm	

Experiment 1

Aim: Write a program to implement the DDA Line Algorithm

Theory: DDA (Digital Differential Analyzer) is a line drawing algorithm used in computer graphics to generate a line segment between two specified endpoints. It is a simple and efficient algorithm that works by using the incremental difference between the x-coordinates and y-coordinates of the two endpoints to plot the line.

Algorithm: The steps involved in DDA line generation algorithm are:

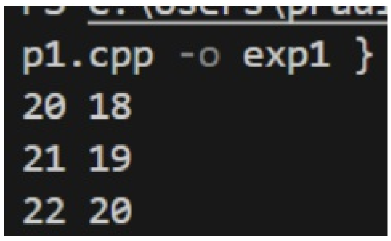
1. Input the two endpoints of the line segment, (x1,y1) and (x2,y2).
2. Calculate the difference between the x-coordinates and y-coordinates of the endpoints as dx and dy respectively.
3. Calculate the slope of the line as m = dy/dx.
4. Set the initial point of the line as (x1,y1).
5. Loop through the x-coordinates of the line, incrementing by one each time, and calculate the corresponding y-coordinate using the equation y = y1 + m(x – x1).

6. Plot the pixel at the calculated (x,y) coordinate.
7. Repeat steps 5 and 6 until the endpoint (x2,y2) is reached.

Code:

```
#include <bits/stdc++.h>

using namespace std;

void DDALine(int x0, int y0, int x1, int y1) {

    int dx = x1 - x0;

    int dy = y1 - y0;

    int step;

    if (abs(dx) > abs(dy))

        step = abs(dx);

    else

        step = abs(dy);

    float x_incr = (float)dx / step;

    float y_incr = (float)dy / step;

    float x = x0;

    float y = y0;

    for (int i = 0; i <= step; i++){

        cout << x << " " << y << "\n";

        x += x_incr;

        y += y_incr;

    }

}

int main() {

    int x0 = 20, y0 = 18, x1 = 22, y1 = 20;

    DDALine(x0, y0, x1, y1);

    return 0;

}
```

Output:



```
p1.cpp -o exp1 }
20 18
21 19
22 20
```

Experiment 2

Aim: Write a program to implement the Bresenham's Line Algorithm

Theory: 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 operationsThe idea of Bresenham's algorithm is to avoid floating point multiplication and addition to compute mx + c, and then compute the round value of (mx + c) in every step. In Bresenham's algorithm, we move across the x-axis in unit intervals.

Algorithm:

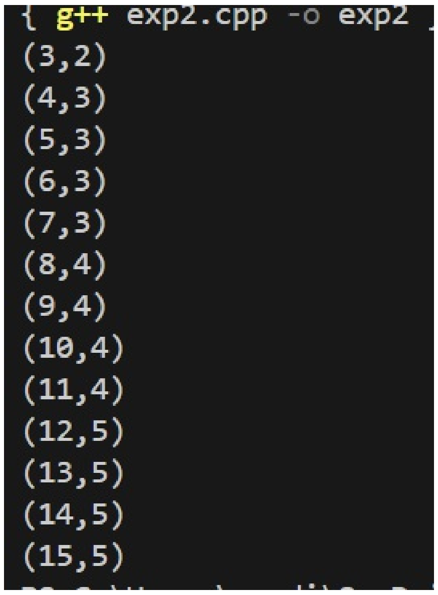
1. Input the two endpoints of the line, save the left endpoint in (x_0, y_0)

2. Plot the point (x_0, y_0)
3. Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and obtain the first value for the decision parameter as:
   $p_0 = 2\Delta y - \Delta x$
4. Perform the following test at each x_k along the line, beginning at $k = 0$. If $p_k < 0$, then the next point to the plot is (x_{k+1}, y_k) and:
   $p_{k+1} = p_k + 2\Delta y$
   Otherwise, the next point to plot is (x_{k+1}, y_{k+1}) and:
   $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
5. Repeat step 4, $(\Delta x - 1)$ times

Code:

```
#include <bits/stdc++.h>

using namespace std;

void bresenham(int x1, int y1, int x2, int y2) {

  int m_new = 2 * (y2 - y1);

  int slope_error_new = m_new - (x2 - x1);

  for (int x = x1, y = y1; x <= x2; x++) {

    cout << "(" << x << "," << y << ")\n";

    slope_error_new += m_new;

    if (slope_error_new >= 0) {

      y++;

      slope_error_new -= 2 * (x2 - x1);

    }

  }

}

int main(){

  int x1 = 3, y1 = 2, x2 = 15, y2 = 5;

  bresenham(x1, y1, x2, y2);

  return 0;

}
```

Output:



```
{ g++ exp2.cpp -o exp2
(3,2)
(4,3)
(5,3)
(6,3)
(7,3)
(8,4)
(9,4)
(10,4)
(11,4)
(12,5)
(13,5)
(14,5)
(15,5)
```

EXPERIMENT 3

AIM : Write a program to implement the Mid-Point Circle Drawing Algorithm

THEORY: The midpoint circle drawing algorithm is an algorithm used to determine the points needed for rasterizing a circle. We use the midpoint algorithm to calculate all the perimeter points of the circle in the first octant and then print them along with their mirror points in the other octants. This will work because a circle is symmetric about its center.

ALGORITHM :

```
1. We assign the starting point coordinates ($X_0$, $Y_0$) as $X_0 = 0$, $Y_0 = R$
2. We calculate decision parameter $P_0$, $P_0 = 1 – R$
3. We calculate $P_k$, ($X_k$, $Y_k$), and ($X_{k+1}$, $Y_{k+1}$)
    1. If $P_k < 0$
    2. $P_{k+1} = P_k + 2X_{k+1} + 1$, $X_{k+1} = X_k + 1$, $Y_{k+1} = Y_k$
    3. Else
    4. $P_{k+1} = P_k + 2X_{k+1} + 2Y_{k+1} + 1$, $X_{k+1} = X_k + 1$, $Y_{k+1} = Y_k - 1$
```

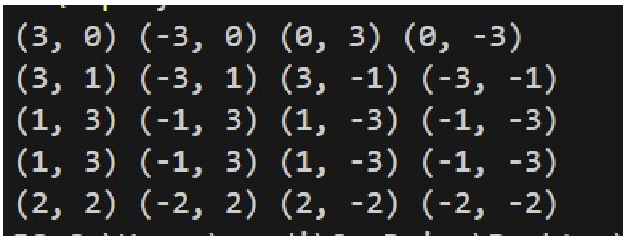
This will give points in the first quadrant. We will then change signs and get the points in other quadrants.

Code:

```
#include<iostream>

using namespace std;

void midPointCircleDraw(int x_centre, int y_centre, int r){

  int x = r, y = 0;

  cout << "(" << x + x_centre << ", " << y + y_centre << ") ";

  if (r > 0){

    cout << "(" << -x + x_centre << ", " << y + y_centre << ") ";

    cout << "(" << y + x_centre << ", " << x + y_centre << ") ";

    cout << "(" << -y+ x_centre << ", " << -x + y_centre << ")\n";

  }

  int P = 1 - r;

  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;

    cout << "(" << x + x_centre << ", " << y + y_centre << ") ";

    cout << "(" << -x + x_centre << ", " << y + y_centre << ") ";

    cout << "(" << x + x_centre << ", " << -y + y_centre << ") ";

    cout << "(" << -x + x_centre << ", " << -y + y_centre << ")\n";

    if (x != y){

      cout << "(" << y + x_centre << ", " << x + y_centre << ") ";

      cout << "(" << -y + x_centre << ", " << x + y_centre << ") ";

      cout << "(" << y + x_centre << ", " << -x + y_centre << ") ";
```

```
            cout << "(" << -y + x_centre << ", " << -x + y_centre << ")\n";

        }

    }

}

int main(){

    midPointCircleDraw(0, 0, 3);

    return 0;

}
```

Output:



```
(3, 0) (-3, 0) (0, 3) (0, -3)
(3, 1) (-3, 1) (3, -1) (-3, -1)
(1, 3) (-1, 3) (1, -3) (-1, -3)
(1, 3) (-1, 3) (1, -3) (-1, -3)
(2, 2) (-2, 2) (2, -2) (-2, -2)
```

Experiment 4

Aim: Write a program to implement the Mid-Point Ellipse Drawing Algorithm.

Theory: Midpoint ellipse algorithm plots(finds) points of an ellipse on the first quadrant by dividing the quadrant into two regions. Each point(x, y) is then projected into the other three quadrants (-x, y), (x, -y), (-x, -y).

Algorithm:

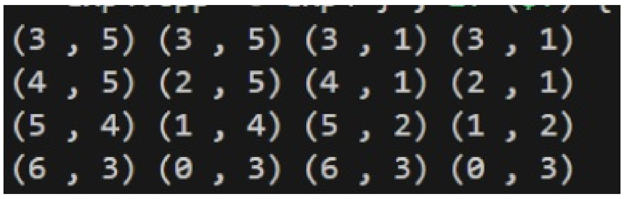
1. Take the input radius along the x axis and y axis and obtain the center of the ellipse.
2. Initially, we assume the ellipse to be centered at origin and the first point as : $(x, y_0)= (0, r_y)$.
3. Obtain the initial decision parameter for region 1 as: $p1_0=r_y^2+1/4r_x^2-r_x{}^2r_y$
4. For every x_k position in region 1 :
   If $p1_k<0$ then the next point along the is (x_{k+1}, y_k) and $p1_{k+1}=p1_k+2r_y{}^2x_{k+1}+r_y{}^2$
   Else, the next point is (x_{k+1}, y_{k-1})
   And $p1_{k+1}=p1_k+2r_y{}^2x_{k+1} - 2r_x^2y_{k+1}+r_y{}^2$
5. Obtain the initial value in region 2 using the last point (x_0, y_0) of region 1 as: $p2_0=r_y{}^2(x_0+1/2)^2+r_x^2 (y_0-1)^2-r_x^2r_y{}^2$
6. At each y_k in region 2 starting at k =0 perform the following task.
   If $p2_k>0$ the next point is (x_k, y_{k-1}) and $p2_{k+1}=p2_k-2r_x^2y_{k+1}+r_x^2$
7. Else, the next point is (x_{k+1}, y_{k-1}) and $p2_{k+1}=p2_k+2r_y{}^2x_{k+1} -2r_x^2y_{k+1}+r_x^2$
8. Now obtain the symmetric points in the three quadrants and plot the coordinate value as: x=x+xc, y=y+yc
9. Repeat the steps for region 1 until $2r_y{}^2x$&gt=$2r_x^2y$

Code:

```
#include <bits/stdc++.h>

using namespace std;

void midptellipse(int rx, int ry, int xc, int yc){

    float dx, dy, d1, d2, x, y;

    x = 0;

    y = ry;

    d1 = (ry * ry) - (rx * rx * ry) +

                    (0.25 * rx * rx);

    dx = 2 * ry * ry * x;

    dy = 2 * rx * rx * y;

    while (dx < dy) {
        cout << "("<<x + xc << " , " << y + yc << ")"<<" ";
        cout << "("<<-x + xc << " , " << y + yc << ")"<<" ";
        cout << "("<<x + xc << " , " << -y + yc << ")"<<" ";
        cout << "("<< -x + xc << " , " << -y + yc <<")"<<endl;
        if (d1 < 0){
            x++;
            dx = dx + (2 * ry * ry);
            d1 = d1 + dx + (ry * ry);
        }
        else{
            x++;
            y--;
            dx = dx + (2 * ry * ry);
            dy = dy - (2 * rx * rx);
            d1 = d1 + dx - dy + (ry * ry);
        }
    }
    d2 = ((ry * ry) * ((x + 0.5) * (x + 0.5))) +
        ((rx * rx) * ((y - 1) * (y - 1))) -
        (rx * rx * ry * ry);
    while (y >= 0){
        cout << "("<< x + xc << " , " << y + yc << ")"<<" ";
        cout <<"("<< -x + xc << " , " << y + yc << ")"<<" ";
        cout <<"("<< x + xc << " , " << -y + yc << ")"<<" ";
        cout << "("<< -x + xc << " , " << -y + yc << ")"<<" ";
        if (d2 > 0) {
            y--;
            dy = dy - (2 * rx * rx);
            d2 = d2 + (rx * rx) - dy;
        }
        else{
            y--;
            x++;
            dx = dx + (2 * ry * ry);
            dy = dy - (2 * rx * rx);
            d2 = d2 + dx - dy + (rx * rx);
        }
    }
}
```

```
int main(){
    midptellipse(3, 2, 3, 3);
    return 0;
}
```

Output:



```
(3 , 5) (3 , 5) (3 , 1) (3 , 1)
(4 , 5) (2 , 5) (4 , 1) (2 , 1)
(5 , 4) (1 , 4) (5 , 2) (1 , 2)
(6 , 3) (0 , 3) (6 , 3) (0 , 3)
```

Experiment 5

Aim: Write a program To write a program and implement Flood Fill Algorithm

Theory: Given a 2D screen arr[][] where each arr[i][j] is an integer representing the color of that pixel, also given the location of a pixel (X, Y) and a color C, the task is to replace the color of the given pixel and all the adjacent same-colored pixels with the given color.

The values in the given 2D screen indicate colors of the pixels. X and Y are coordinates of the brush, C is the color that should replace the previous color on screen[X][Y] and all surrounding pixels with the same color. Hence all the 2 are replaced with 3.

Algorithm: The idea is to use BFS traversal to replace the color with the new color.

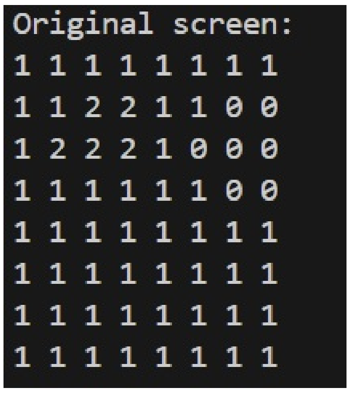
- Create an empty queue lets say Q.
- Push the starting location of the pixel as given in the input and apply replacement color to it. • Iterate until Q is not empty and pop the front node (pixel position).

Check the pixels adjacent to the current pixel and push into the queue if valid (had not been colored with replacement color and have the same color as the old color).

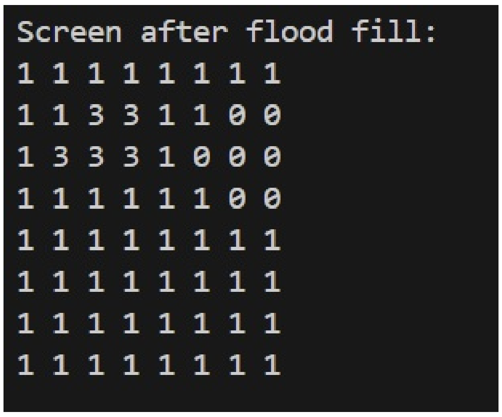
Code:

```
#include <iostream>

using namespace std;

#define M 8

#define N 8

void printScreen(int screen[M][N]) {

    for (int i = 0; i < M; i++) {

        for (int j = 0; j < N; j++) {

            cout << screen[i][j] << " ";

        }

        cout << endl;

    }

    cout << endl;

}

void floodFill(int screen[M][N], int x, int y, int newColor, int oldColor) {

    if (x < 0 || x >= M || y < 0 || y >= N) return;

    if (screen[x][y] != oldColor) return;

    screen[x][y] = newColor;

    floodFill(screen, x + 1, y, newColor, oldColor);

    floodFill(screen, x - 1, y, newColor, oldColor);
    floodFill(screen, x, y + 1, newColor, oldColor);

    floodFill(screen, x, y - 1, newColor, oldColor);

}

void fill(int screen[M][N], int x, int y, int newColor) {

    int oldColor = screen[x][y];

    if (oldColor != newColor) {

        floodFill(screen, x, y, newColor, oldColor);

    }

}

int main() {

    int screen[M][N] = {

        {1, 1, 1, 1, 1, 1, 1, 1},

        {1, 1, 2, 2, 1, 1, 0, 0},

        {1, 2, 2, 2, 1, 0, 0, 0},

        {1, 1, 1, 1, 1, 1, 0, 0},

        {1, 1, 1, 1, 1, 1, 1, 1},

        {1, 1, 1, 1, 1, 1, 1, 1},

        {1, 1, 1, 1, 1, 1, 1, 1},

        {1, 1, 1, 1, 1, 1, 1, 1}

    };

    int x = 2, y = 2, newColor = 3;

    cout << "Original screen:\n";

    printScreen(screen);

    fill(screen, x, y, newColor);

    cout << "Screen after flood fill:\n";

    printScreen(screen);

    return 0;

}
```

Output:



```
Original screen:
1 1 1 1 1 1 1 1
1 1 2 2 1 1 0 0
1 2 2 2 1 0 0 0
1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```



```
Screen after flood fill:
1 1 1 1 1 1 1 1
1 1 3 3 1 1 0 0
1 3 3 3 1 0 0 0
1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Experiment 6

Aim: Write a program to implement Boundary Fill Algorithm

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

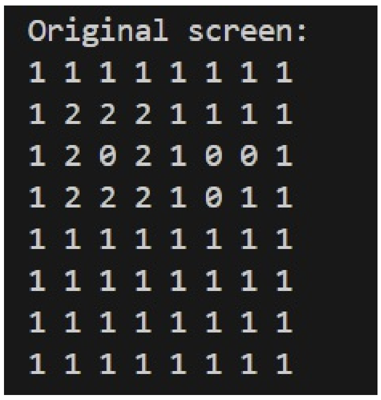
Algorithm:

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

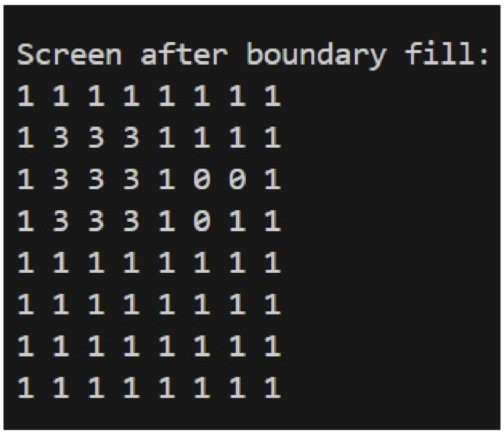
Code:

```
#include <iostream>
using namespace std;
#define M 8
#define N 8
void printScreen(int screen[M][N]) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            cout << screen[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}
void boundaryFill(int screen[M][N], int x, int y, int fillColor, int boundaryColor) {
    if (x < 0 || x >= M || y < 0 || y >= N) return;
    if (screen[x][y] == boundaryColor || screen[x][y] == fillColor) return;
    screen[x][y] = fillColor;
    boundaryFill(screen, x + 1, y, fillColor, boundaryColor);
    boundaryFill(screen, x - 1, y, fillColor, boundaryColor);
    boundaryFill(screen, x, y + 1, fillColor, boundaryColor);
    boundaryFill(screen, x, y - 1, fillColor, boundaryColor);
}
int main() {
    int screen[M][N] = {
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 2, 2, 2, 1, 1, 1, 1},
        {1, 2, 0, 2, 1, 0, 0, 1},
        {1, 2, 2, 2, 1, 0, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1}
    };
    int x = 2, y = 2, fillColor = 3, boundaryColor = 1;
    cout << "Original screen:\n";
    printScreen(screen);
    boundaryFill(screen, x, y, fillColor, boundaryColor);
    cout << "Screen after boundary fill:\n";
    printScreen(screen);
    return 0;
}
```

Output:



```
Original screen:
1 1 1 1 1 1 1 1
1 2 2 2 1 1 1 1
1 2 0 2 1 0 0 1
1 2 2 2 1 0 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

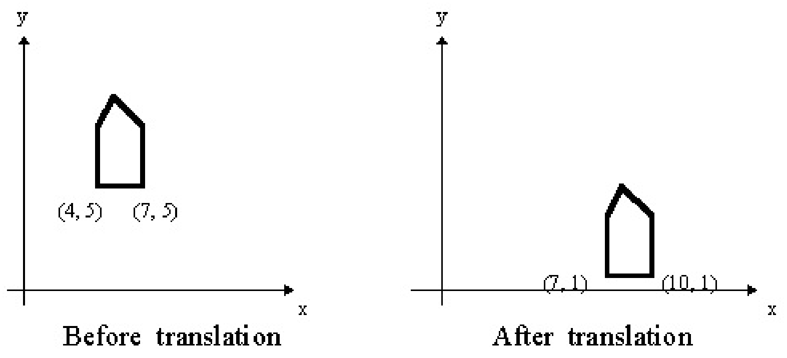


```
Screen after boundary fill:
1 1 1 1 1 1 1 1
1 3 3 3 1 1 1 1
1 3 3 3 1 0 0 1
1 3 3 3 1 0 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Experiment 7

Aim: Write a program to implement 2D Translation of an object

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



We can move the objects based on translation distances along the 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 many units we should shift the object from one location to another along the x, y-axis.

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

X'=x+tx

Y’=y+ty

We denote translation transformation as P. we express above equations in matrix form as:

$$P' = P + T$$

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

x,y---old coordinates

x’,y’—new coordinates after translation

tx,ty—translation distances

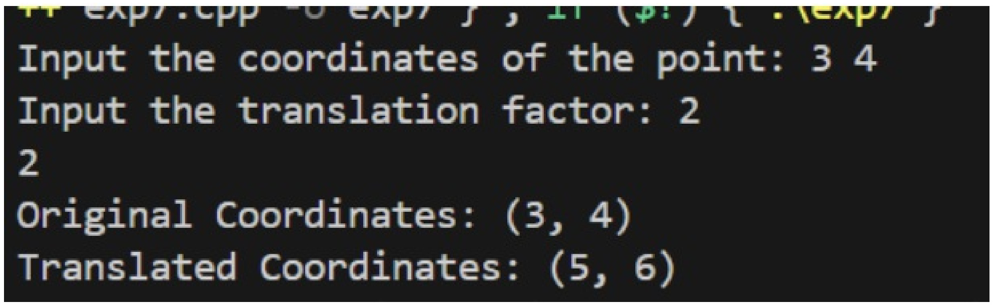
Algorithm: The steps involved in 2D translation are:

1. Input the coordinates of the point and the translation factor of the point as x,y and tx, ty
2. Set xt=x+tx and yt=y+ty
3. Output the results

Code:

```
#include <iostream>

#include <vector>

using namespace std;

void translatePoint(vector<int>& P, const vector<int>& T) {

    cout << "Original Coordinates: (" << P[0] << ", " << P[1] << ")" << endl;

    P[0] += T[0];

    P[1] += T[1];

    cout << "Translated Coordinates: (" << P[0] << ", " << P[1] << ")" << endl;

}

int main() {

    vector<int> P = {5, 8};

    vector<int> T = {2, 1};

    cout<<"Input the coordinates of the point: ";

    cin>>P[0]>>P[1];

    cout<<"Input the translation factor: ";

    cin>>T[0]>>T[1];

    translatePoint(P, T);

    return 0;

}
```

Output:

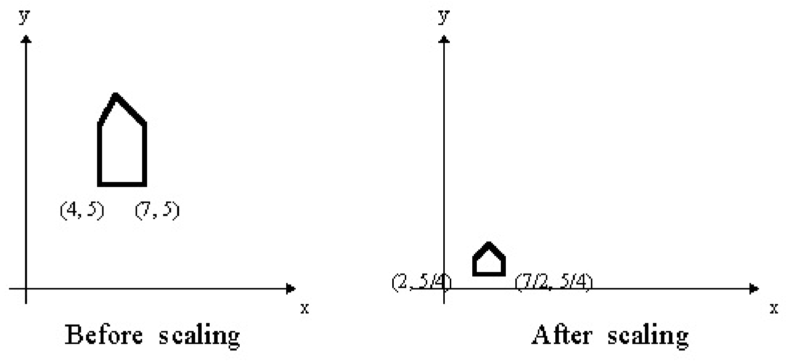


```
++ exp7.cpp -o exp7 } ; if ($?) { .\exp7 }
Input the coordinates of the point: 3 4
Input the translation factor: 2
2
Original Coordinates: (3, 4)
Translated Coordinates: (5, 6)
```

Experiment 8

Aim: Write a program to implement 2D Scaling of an object

Theory: 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. we express the above equations in matrix form as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}}_{\boxed{\text{Scaling Matrix}}} \begin{bmatrix} x \\ y \end{bmatrix}$$

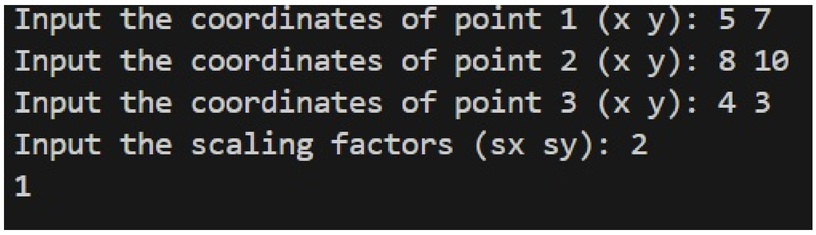
Algorithm: The steps involved in 2D scaling are:

1. Input the coordinates of the points and the scaling factors as x,y and sx,sy
2. Set xnew=x*sx and y new=y*sy
3. Output the result

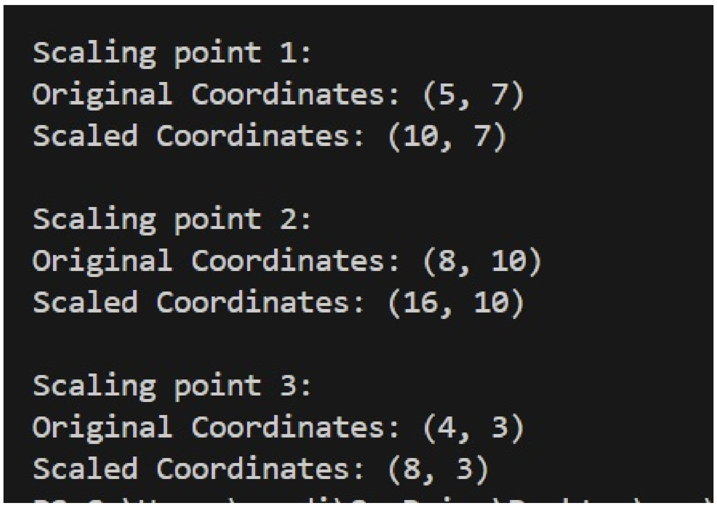
Code:

```
#include <iostream>

#include <vector>

using namespace std;

void scalePoint(vector<int>& P, const vector<int>& S) {

    cout << "Original Coordinates: (" << P[0] << ", " << P[1] << ")" << endl;

    P[0] *= S[0];

    P[1] *= S[1];

    cout << "Scaled Coordinates: (" << P[0] << ", " << P[1] << ")" << endl;

}

int main() {

    vector<vector<int>> points(3, vector<int>(2));

    vector<int> scale(2);

    for (int i = 0; i < 3; ++i) {

        cout << "Input the coordinates of point " << i + 1 << " (x y): ";

        cin >> points[i][0] >> points[i][1];
    }

    cout << "Input the scaling factors (sx sy): ";

    cin >> scale[0] >> scale[1];

    for (int i = 0; i < 3; ++i) {

        cout << "\nScaling point " << i + 1 << ":" << endl;

        scalePoint(points[i], scale);

    }

    return 0;

}
```

Output:



```
Input the coordinates of point 1 (x y): 5 7
Input the coordinates of point 2 (x y): 8 10
Input the coordinates of point 3 (x y): 4 3
Input the scaling factors (sx sy): 2
1
```



```
Scaling point 1:
Original Coordinates: (5, 7)
Scaled Coordinates: (10, 7)

Scaling point 2:
Original Coordinates: (8, 10)
Scaled Coordinates: (16, 10)

Scaling point 3:
Original Coordinates: (4, 3)
Scaled Coordinates: (8, 3)
```

Experiment 9

Aim: Write a program to implement 2D Rotation of an object

Theory: A rotation repositions all points in an object along a circular path in the plane centered at the pivot point. We rotate an object by an angle theta.

New coordinates after rotation depend on both x and y

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

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

or in matrix form:

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

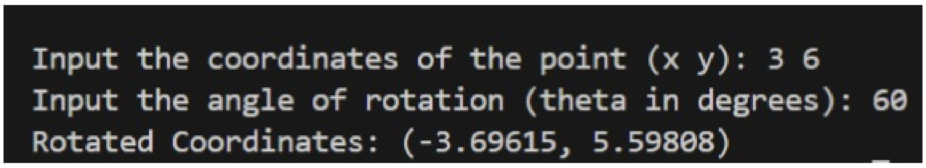
Algorithm: The steps involved in 2D rotation are:

1. Input the coordinates of the point and the angle of rotation as x,y and theta
2. Set xnew=xcos(theta) -ysin(theta) and ynew=xsin(theta) + ycos(theta)
3. Output the results

Code:

```
#include <iostream>

#include <cmath>

using namespace std;

void rotatePoint(double &x, double &y, double theta) {

    double radians = theta * M_PI / 180.0;

    double x_new = x * cos(radians) - y * sin(radians);
    double y_new = x * sin(radians) + y * cos(radians);

    cout << "Rotated Coordinates: (" << x_new << ", " << y_new << ")" << endl;

}

int main() {

    double x, y, theta;

    cout << "Input the coordinates of the point (x y): ";

    cin >> x >> y;

    cout << "Input the angle of rotation (theta in degrees): ";

    cin >> theta;

    rotatePoint(x, y, theta);

    return 0;

}
```

Output:



```
Input the coordinates of the point (x y): 3 6
Input the angle of rotation (theta in degrees): 60
Rotated Coordinates: (-3.69615, 5.59808)
```

Experiment 10

Aim: Write a program To implement Cohen-Sutherland Line Clipping Algorithm

Theory: 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.

Algorithm:

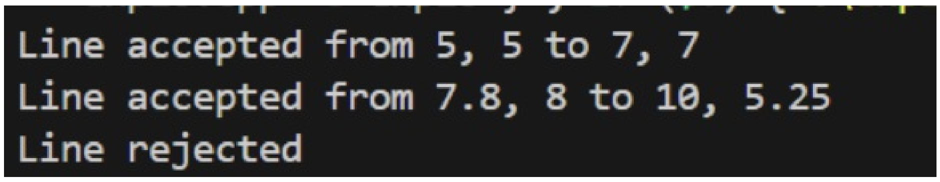
```
Step 1 : Assign a region code for two endpoints of a given line.

Step 2 : If both endpoints have a region code 0000

then the given line is completely inside.

Step 3 : Else, perform the logical AND operation for both region codes.

Step 3.1 : If the result is not 0000, then given line is completely

outside.

Step 3.2 : Else line is partially inside.

Step 3.2.1 : Choose an endpoint of the line

that is outside the given rectangle.

Step 3.2.2 : Find the intersection point of the

rectangular boundary (based on region code).

Step 3.2.3 : Replace endpoint with the intersection point

and update the region code.

Step 3.2.4 : Repeat step 2 until we find a clipped line either

trivially accepted or trivially rejected.

Step 4 : Repeat step 1 for other lines.
```

Code:

```
#include <iostream>
```

```
using namespace std;

const int INSIDE = 0, LEFT = 1, RIGHT = 2, BOTTOM = 4, TOP = 8;

const int x_max = 10;

const int y_max = 8;

const int x_min = 4;

const int y_min = 4;

int computeCode(double x, double 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 cohenSutherlandClip(double x1, double y1,double x2, double y2){

    int code1 = computeCode(x1, y1);

    int code2 = computeCode(x2, y2);

    bool accept = false;

    while (true) {

        if ((code1 == 0) && (code2 == 0)) {

            accept = true;

            break;

        }

        else if (code1 & code2) {

            break;

        }

        else {

            int code_out;

            double x, y;

            if (code1 != 0)

                code_out = code1;

            else

                code_out = code2;

            if (code_out & TOP) {

                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
                y = y_max;
            }
            else if (code_out & BOTTOM) {
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
                y = y_min;
            }
            else if (code_out & RIGHT) {
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
                x = x_max;
            }
            else if (code_out & LEFT) {
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
                x = x_min;
            }
            if (code_out == code1) {
                x1 = x;
                y1 = y;
                code1 = computeCode(x1, y1);
            }
            else {
                x2 = x;
                y2 = y;
                code2 = computeCode(x2, y2);
            }
        }
    }
    if (accept) {
        cout << "Line accepted from " << x1 << ", "
             << y1 << " to " << x2 << ", " << y2 << endl;
    }
    else
        cout << "Line rejected" << endl;
}
int main(){
    cohenSutherlandClip(5, 5, 7, 7);
    cohenSutherlandClip(7, 9, 11, 4);
    cohenSutherlandClip(1, 5, 4, 1);
    return 0;
}
```

Output:



```
Line accepted from 5, 5 to 7, 7
Line accepted from 7.8, 8 to 10, 5.25
Line rejected
```

Experiment 11

Aim: Write a program to implement Liang Barsky Line Clipping algorithm

Theory: The Liang-Barsky algorithm is a line clipping algorithm. This algorithm is more efficient than Cohen–Sutherland line clipping algorithm and can be extended to 3-Dimensional clipping. This algorithm is considered to be the faster parametric line-clipping algorithm. The following concepts are used in this clipping. The parametric equation of the line. The inequalities describing the range of the clipping window which is used to determine the intersections between the line and the clip window.

Algorithm:

1. Set tmin=0, tmax=1.

2. Calculate the values of t (t(left), t(right), t(top), t(bottom)),

(i) If t < tminignore that and move to the next edge.

(ii) else separate the t values as entering or exiting values using the inner product.

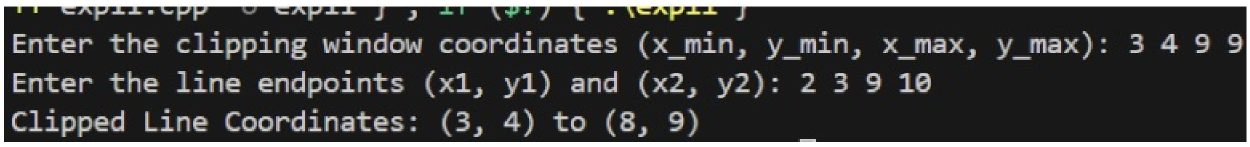
(iii) If t is entering value, set tmin = t; if t is existing value, set tmax = t.

3. If tmin < tmax, draw a line from (x1 + tmin(x2-x1), y1 + tmin(y2-y1)) to (x1 + tmax(x2-x1), y1 + tmax(y2-y1)) 4. If the line crosses over the window, (x1 + tmin(x2-x1), y1 + tmin(y2-y1)) and (x1 + tmax(x2-x1), y1 + tmax(y2- y1)) are the intersection point of line and edge.

Code:

```
#include <iostream>

#include <algorithm>

using namespace std;

float x_min, y_min, x_max, y_max;

void liangBarskyClip(float x1, float y1, float x2, float y2) {

    float p[4], q[4];

    p[0] = -(x2 - x1);

    p[1] = x2 - x1;

    p[2] = -(y2 - y1);

    p[3] = y2 - y1;

    q[0] = x1 - x_min;

    q[1] = x_max - x1;

    q[2] = y1 - y_min;

    q[3] = y_max - y1;

    float u1 = 0.0, u2 = 1.0;

    for (int i = 0; i < 4; i++) {

        if (p[i] == 0) {

            if (q[i] < 0) {

                cout << "Line is outside the clipping window.\n";

                return;

            }
        } else {
            float u = q[i] / p[i];
            if (p[i] < 0) {
                u1 = max(u1, u);
            } else {
                u2 = min(u2, u);
            }
        }
    }
    if (u1 > u2) {
        cout << "Line is outside the clipping window.\n";
    } else{
        float clipped_x1 = x1 + u1 * (x2 - x1);
        float clipped_y1 = y1 + u1 * (y2 - y1);
        float clipped_x2 = x1 + u2 * (x2 - x1);
        float clipped_y2 = y1 + u2 * (y2 - y1);
        cout << "Clipped Line Coordinates: (" << clipped_x1 << ", " << clipped_y1 << ") to ("
             << clipped_x2 << ", " << clipped_y2 << ")\n";
    }
}
int main() {
    cout << "Enter the clipping window coordinates (x_min, y_min, x_max, y_max): ";
    cin >> x_min >> y_min >> x_max >> y_max;
    float x1, y1, x2, y2;
    cout << "Enter the line endpoints (x1, y1) and (x2, y2): ";
    cin >> x1 >> y1 >> x2 >> y2;
    liangBarskyClip(x1, y1, x2, y2);
    return 0;
}
```

Output:



```
Enter the clipping window coordinates (x_min, y_min, x_max, y_max): 3 4 9 9
Enter the line endpoints (x1, y1) and (x2, y2): 2 3 9 10
Clipped Line Coordinates: (3, 4) to (8, 9)
```