

Department of Computer Science & Engineering
Chittagong University Of Engineering & Technology
Chittagong-4349

Course Code: CSE - 458
Course Title: Graphics Design (Sessional)

Fariha Iffath, ID: 1304120

March 22, 2019

Contents

1	DDA Line Drawing Algorithm	3
1.1	Objective	3
1.2	Introduction	3
1.3	Source Code	3
1.4	Sample Input-Output	5
1.5	Discussion	6
2	Bresenhams Line drawing Algorithm	7
2.1	Objective	7
2.2	Description	7
2.3	Source Code	8
2.4	Sample Input-Output	10
2.5	Discussion	10
3	Bresenhams Circle algorithm	11
3.1	Objective	11
3.2	Introduction	11
3.3	Source Code	11
3.4	Sample Input-Output	13
3.5	Discussion	13
4	Midpoint Circle Algorithm	14
4.1	Objective	14
4.2	Introduction	14
4.3	Source Code	14
4.4	Input-Output	15
4.5	Conclusion	16
5	Midpoint Ellipse Algorithm	17
5.1	Objective	17
5.2	Introduction	17
5.3	Source Code	17
5.4	Input-Output	19
5.5	Discussion	20
6	Boundary Fill Algorithm	21
6.1	Objective	21
6.2	Introduction	21
6.3	Description	21
6.4	Algorithm	21
6.5	Source Code	21
6.6	Sample Input-Output	24
6.7	Discussion	24
7	Flood fill algorithm	25
7.1	Objective	25
7.2	Introduction	25
7.3	Description	25
7.4	Source Code	25
7.5	Output	27
7.6	Discussion	28

8	C Curve Algorithm	29
8.1	Objective	29
8.2	Introduction	29
8.3	Source Code	29
8.4	Input-Output	31
8.5	Discussion	31
9	Koch Curve Algorithm	32
9.1	Objective	32
9.2	Introduction	32
9.3	Source Code	32
9.4	Input-Output	34
9.5	Discussion	34
10	Cohen-Sutherland Algorithm	35
10.1	Objective	35
10.2	Introduction	35
10.3	Algorithm	35
10.4	Source Code	35
10.5	Input-Output	38
10.6	Discussion	39
11	Sutherland-Hodgman Algorithm	40
11.1	Objective	40
11.2	Introduction	40
11.3	Source Code	40
11.4	Output	43
11.5	Discussion	43

1 DDA Line Drawing Algorithm

1.1 Objective

The main objective of this experiment is to Scan convert a line digital differential analyzer(DDA) using digital differential analyzer(DDA) line drawing algorithm.

1.2 Introduction

In computer graphics we can not directly join any two coordinate points, for that we should calculate intermediate points coordinate and put a pixel for each intermediate point, of the desired color with help of functions like putpixel(x, y, K) in C, where (x,y) is our co-ordinate and K denotes some color. for generating any line segment we need intermediate points and for calculating them we have can use a basic algorithm called DDA(Digital differential analyzer) line generating algorithm.A DDA algorithm is the simple line generation algorithm which is explained step by step here.

Step 1 Get the input of two end points (X0,Y0) and (X1,Y1).

Step 2 Calculate the difference between two end points.

$$dx = X1 - X0$$

$$dy = Y1 - Y0$$

Step 3 Based on the calculated difference in step-2, you need to identify the number of steps to put pixel. If $dx > dy$, then you need more steps in x-coordinate; otherwise in y-coordinate.

```
if (absolute(dx) > absolute(dy))
```

```
    Steps = absolute(dx);
```

```
else
```

```
    Steps = absolute(dy);
```

Step 4 Calculate the increment in x coordinate and y coordinate.

$$Xincrement = dx / (\text{float}) \text{ steps};$$

$$Yincrement = dy / (\text{float}) \text{ steps};$$

Step 5 Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

1.3 Source Code

```
#include <windows.h>
#include <iostream>
#include <bits/stdc++.h>
#include <GL/glut.h>

GLint x0,y0,xEnd,yEnd;
inline GLint round(const GLfloat a)
{
    return GLint(a+0.5);
}

void myInit(void)
```

```

{
    glClearColor(0.0,1.0,1.0,0.0);
    glColor3f(1.0f,0.0f,0.0f);
    glPointSize(3.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,640.0,0.0,480.0);
}
void readInput()
{
    printf("Enter x0, y0, x1, y1: \n");
    scanf("%i %i %i %i",&x0,&y0,&xEnd,&yEnd);
}
void setPixel(GLint xcoordinate, GLint ycoordinate)
{
    glBegin(GL_POINTS);
    glVertex2i(xcoordinate,ycoordinate);
    glEnd();
    glFlush();
}
void lineDDA(GLint x0,GLint y0,GLint xEnd,GLint yEnd)
{
    GLint dx = abs(xEnd-x0);
    GLint dy = abs(yEnd-y0);
    GLint steps,k;
    GLfloat xIncrement,yIncrement,x=x0,y=y0;

    if(dx>dy)
        steps = dx;
    else
        steps = dy;
    xIncrement = GLfloat(dx) / GLfloat(steps);
    yIncrement = GLfloat(dy) / GLfloat(steps);
    setPixel(round(x),round(y));

    for(k=1;k<steps;k++)
    {
        x+= xIncrement;
        y+= yIncrement;
        setPixel(round(x),round(y));
    }
}
void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    lineDDA(x0,y0,xEnd,yEnd);
}
int main(int argc,char *argv[])
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600,600);
    glutInitWindowPosition(50,50);
    glutCreateWindow("DDA Line Algorithm");
}

```

```

readInput();
glutDisplayFunc(Display);
myInit();
glutMainLoop();
return EXIT_SUCCESS;
}

```

1.4 Sample Input-Output

```

D:\L4t1_14\graphics_lab\practicee\DDA\bin\Debug\DDA.exe
Enter x0, y0, x1, y1:
100 200 300 400

Process returned 0 (0x0)   execution time : 286.658 s
Press any key to continue.

```

Figure 1: User Input

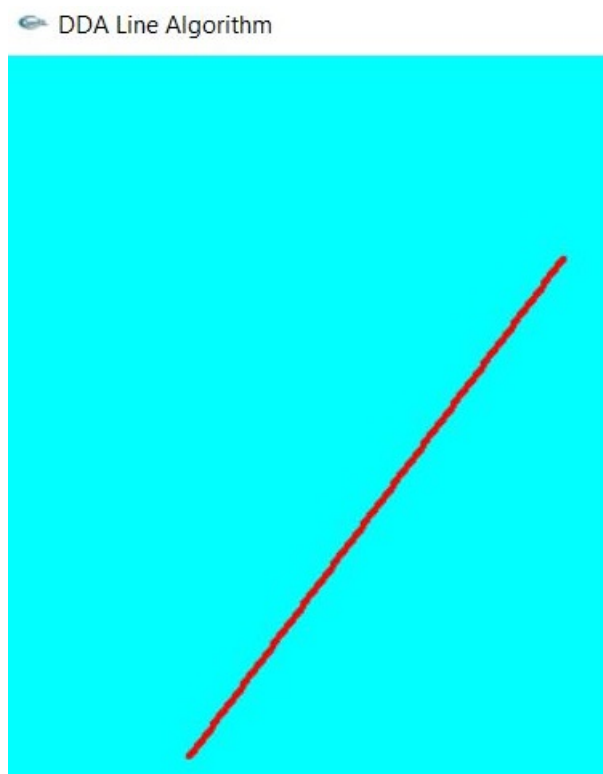


Figure 2: DDA Line Drawing Algorithm.

1.5 Discussion

From this experiment, we knew about the DDA Line Drawing Algorithm and its implementation.

-We find a line that create according to the codes are generated in the opengl.

-We know about opengl and its implantation.

2 Bresenham's Line drawing Algorithm

2.1 Objective

To implement Bresenham's line drawing algorithm.

2.2 Description

The Bresenham algorithm is an incremental scan conversion algorithm across the x axis in unit intervals and at each step choose between two different y coordinates.

For example, as shown in the following illustration, from position (2, 3) we need to choose between (3, 3) and (3, 4). We would like the point that is closer to the original line.

At sample position $X_k + 1$ the vertical separations from the mathematical line are labelled as d_{upper} and d_{lower} .

From the above illustration, the y coordinate on the mathematical line at $x_k + 1$ is $Y = m(X_k + 1) + b$

So, d_{upper} and d_{lower} are given as follows

$$d_{lower} = y - y_k = m(X_k + 1) + b - Y_k$$

$$\text{and } d_{upper} = (y_k + 1) - y = Y_k + 1 - m(X_k + 1) - b = Y_k + 1 - m(X_k + 1) - b$$

We use these to make a simple decision about which pixel is closer to the mathematical line. This simple decision is based on the difference between the two pixel positions.

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

Let us substitute m with $\frac{dy}{dx}$ where dx and dy are the differences between the end-points.

$$dx(d_{lower} - d_{upper}) = dx(2\frac{dy}{dx}(x_k + 1) - 2y_k + 2b - 1)$$

$$= 2dy.x_k - 2dx.y_k + 2dy + 2dx(2b - 1)$$

$$= 2dy.x_k - 2dx.y_k + C$$

So, a decision parameter P_k for the k^{th} step along a line is given by $p_k = dx(d_{lower} - d_{upper}) = 2dy.x_k - 2dx.y_k + C$

The sign of the decision parameter P_k is the same as that of $d_{lower} - d_{upper}$. If p_k is negative, then choose the lower pixel, otherwise choose the upper pixel. At step $k+1$, the decision parameter is given as

$$p_{k+1} = 2dy.x_{k+1} - 2dx.y_{k+1} + C$$

Subtracting p_k from this we get

$$p_{k+1} - p_k = 2dy(x_{k+1} - x_k) - 2dx(y_{k+1} - y_k)$$

But, x_{k+1} is the same as $(x_k) + 1$. So

$p_{k+1} = p_k + 2dy - 2dx(y_{k+1} - y_k)$ Where, $Y_{k+1} - y_k$ is either 0 or 1 depending on the sign of p_k . The first decision parameter p_0 is evaluated at (x_0, y_0) is given as $p_0 = 2dy - dx$. Now, keeping in mind all the above points and calculations, here is the Bresenham algorithm for slope $m < 1$.

Step 1: Input the two end-points of line, storing the left end-point in (x_0, y_0) .

Step 2: Plot the point (x_0, y_0) .

Step 3: Calculate the constants dx, dy, 2dy, and $(2dy - 2dx)$ and get the first value for the decision parameter as,

$$p_0 = 2dy - dx$$

Step 4: At each X_k along the line, starting at $k = 0$, perform the following test,

If $p_k < 0$, the next point to plot is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2dy$$

Otherwise, $(x_k, y_k + 1)$ and

$$p_{k+1} = p_k + 2dy - 2dx$$

Step 5: Repeat step 4 (dx - 1) times. For $m > 1$, find out whether we need to increment x while incrementing y each time.

2.3 Source Code

```
#include<windows.h>
#include<GL/glut.h>
#include<stdio.h>
#include<stdlib.h>

GLint x0,y0,xEnd,yEnd;

void init()
{
    glClearColor(1.0,1.0,1.0,0.0);
    glColor3f(1.0f,0.0f,0.0f);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,600.0,0.0,600.0);
}

void setPixel(GLint xcoordinate, GLint ycoordinate)
{
    glBegin(GL_POINTS);
    glVertex2i(xcoordinate,ycoordinate);
    glEnd();
    glFlush();
}

void lineBA(GLint x0,GLint y0,GLint xEnd,GLint yEnd)
{
    GLint dx = xEnd-x0;
    GLint dy = yEnd-y0;
    GLint steps,k;
    steps=dx;
    GLint x,y,p0=(2*dy)-dx;
    setPixel(x0,y0);
    x=x0;
    y=y0;
    for(k=0;k<steps;k++)
    {
        if(p0<0)
        {
            p0=p0+(2*dy);
            x+=1;
        }
        else
        {
            p0=p0+(2*dy)-(2*dx);
        }
    }
}
```

```

        x+=1;
        y+=1;
    }
    setPixel(x,y);
}

```

```

void readInput()
{
    printf("\nEnter x0, y0, x1, y1 : \n");
    scanf("%i %i %i %i",&x0,&y0,&xEnd,&yEnd);
}

```

```

void Display(void)
{
    //static int i=1;
    glClear(GL_COLOR_BUFFER_BIT);

    lineBA(x0,y0,xEnd,yEnd);
}

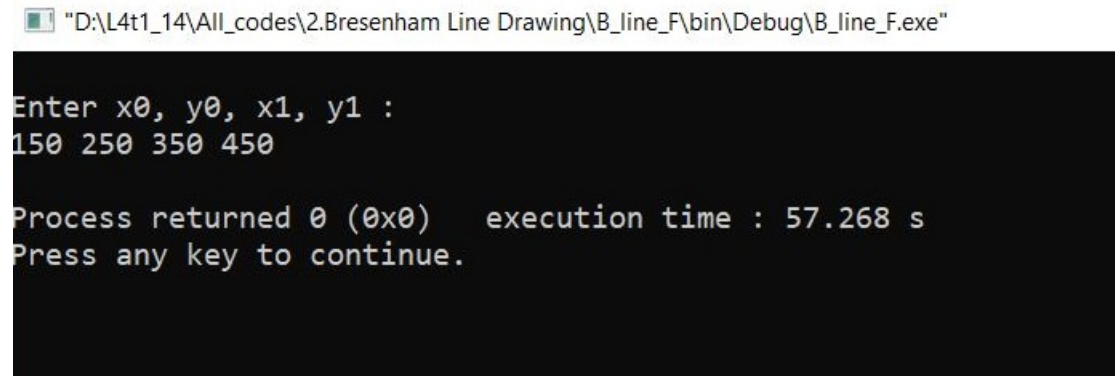
```

```

int main(int argc,char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(600,600);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Bresenham's Line Drawing Algorithm");
    readInput();
    glutDisplayFunc(Display);
    init();
    glutMainLoop();
    return 0;
}

```

2.4 Sample Input-Output



```
"D:\L4t1_14\All_codes\2.Bresenham Line Drawing\B_line_F\bin\Debug\B_line_F.exe"

Enter x0, y0, x1, y1 :
150 250 350 450

Process returned 0 (0x0)   execution time : 57.268 s
Press any key to continue.
```

Figure 3: User Input

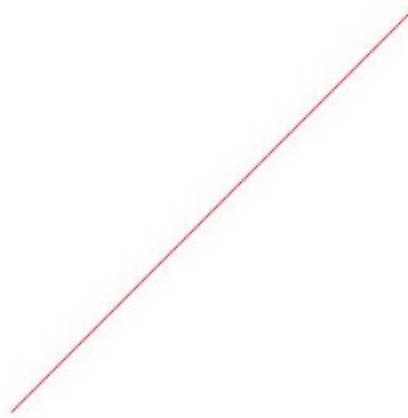


Figure 4: Bresenham Line Drawing Algorithm.

2.5 Discussion

1. The advantage of using this algorithm to draw a line is, it provides more efficient result, because it only use addition, subtraction and multiplication by 2 operation.
2. Since the logic of this program is very easy to understand, it was easy for me to implement it.
3. To implement bresenham line algorithm I have used matlab programming language.
4. The main advantage of this algorithm is, it is less expensive and require less time.
5. Bresenham line algorithm uses fixed points.

3 Bresenham's Circle algorithm

3.1 Objective

The main objective of this experiment is to find a circle using Bresenham's Circle algorithm.

3.2 Introduction

Bresenham's circle algorithm is derived from the midpoint circle algorithm. Here, equation of the circle is,

$$x^2 + y^2 = r^2 \quad (1)$$

The purpose of Bresenham's circle algorithm is to generate the set of points that approximate a circle on a pixel-based display. We generate the points in the first octant of the circle and then use symmetry to generate the other seven octants. We start at (r, 0). This pixel is chosen trivially. Next we need to find the pixel for the y = 1 row. We must make a decision between two candidate points. 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 8 octants of the circle. Now, we will see how to calculate the next pixel location from a previously known pixel location (x,y). We have two options either to choose the next pixel in the east i.e, (x+1,y) or in the south east i.e, (x+1,y+1). This can be decided by using the decision parameter, D. Here,

where, $D = 3 - (2 * \text{radius})$

X=0;

Y=radius

3.3 Source Code

```
#include<windows.h>
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

int cen_x,cen_y,r1;

void plot_point(int x, int y)
{
    glBegin(GL_POINTS);
    glVertex2i(cen_x+x, cen_y+y);
    glVertex2i(cen_x+x, cen_y-y);
    glVertex2i(cen_x+y, cen_y+x);
    glVertex2i(cen_x+y, cen_y-x);
    glVertex2i(cen_x-x, cen_y-y);
    glVertex2i(cen_x-y, cen_y-x);
    glVertex2i(cen_x-x, cen_y+y);
    glVertex2i(cen_x-y, cen_y+x);
    glEnd();
}

void bress_circle(int r1)
{
    int x=0,y=r1;
    int d=3-2*r1;
```

```

    plot_point(x,y);

    while(x < y)
    {
        x = x + 1;
        if(d<0)
            d=d+(4*x)+6;
        else{
            d=d+(4*(x-y))+10;
            y--;
        }

        plot_point(x,y);
    }
    glFlush();
}

void draw_circles(void)
{

    glClear(GL_COLOR_BUFFER_BIT);
    bress_circle(r1);

}

void Init()
{

    glClearColor(1.0,1.0,1.0,0);
    glPointSize(6);
    glColor3f(0.0,0.7,0.5);
    gluOrtho2D(-500, 500 ,-500 , 500);
}

int main(int argc, char **argv)
{
    cen_x=20;
    cen_y=30;
    //printf("\n");
    printf("enter radius  :\n");
    scanf("%d",&r1);

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(0,0);
    glutInitWindowSize(500,500);
    glutCreateWindow("Bresenham_Circle");

    Init();
    glutDisplayFunc(draw_circles);

    glutMainLoop();
}

```

3.4 Sample Input-Output

```
D:\L4t1_14\All_codes\3.bresenham\B_circle_F\bin\Debug\B_circle_F.exe
enter radius :
200

Process returned 0 (0x0)   execution time : 52.468 s
Press any key to continue.
```

Figure 5: User Input

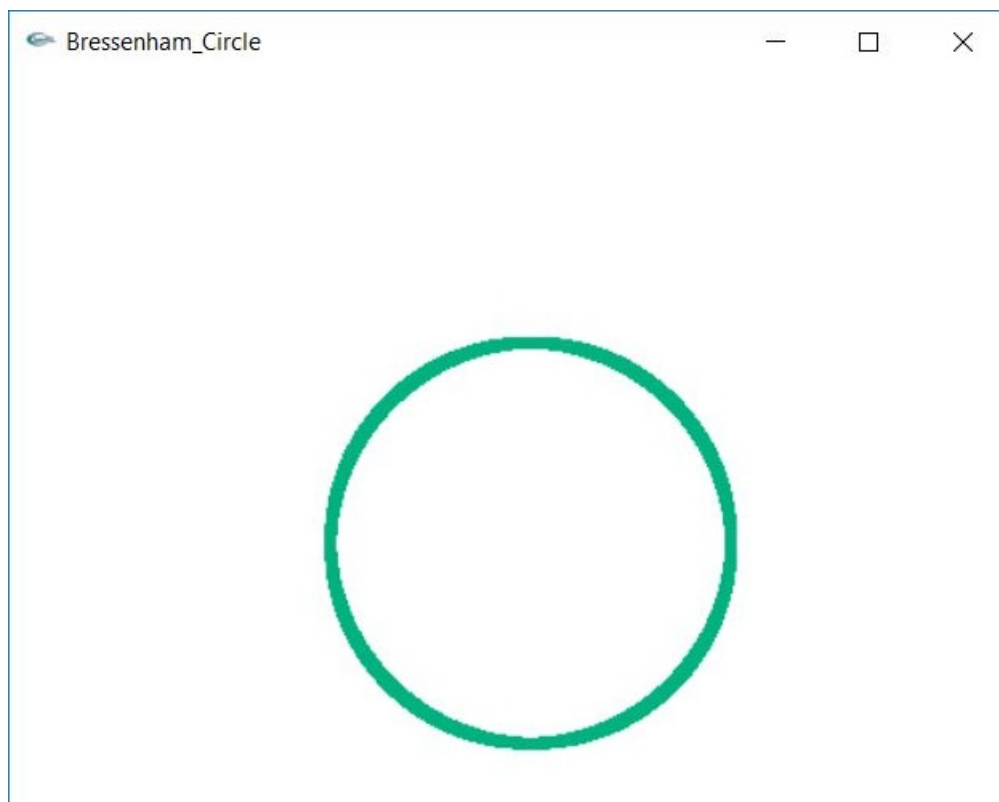


Figure 6: Bresenham Circle Drawing Output.

3.5 Discussion

- From this experiment we knew about the Bresenham's circle algorithm and its implementation.
- We find a circle that create according to the codes are generated in the opengl.
- We know about opengl and its implantation.

4 Midpoint Circle Algorithm

4.1 Objective

To find a circle using Midpoint Circle Algorithm.

4.2 Introduction

The mid-point circle algorithm is an incremental algorithm for drawing circles. In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points. Lets re-jig the equation of the circle slightly to give us:

$$f_{cir}(x, y) = x^2 + y^2 - r^2 \quad (2)$$

4.3 Source Code

```
#include<windows.h>
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

int r;

void Display(void)
{
    int p,pk,d,i,x,y;
    x= 0;
    y=r;
    p=1-r;

    while(x<=y)
    {
        glBegin(GL_POINTS);
        glVertex2i(x,y);
        glVertex2i(y,x);
        glVertex2i(y,-x);
        glVertex2i(x,-y);
        glVertex2i(-x,-y);
        glVertex2i(-y,-x);
        glVertex2i(-y,x);
        glVertex2i(-x,y);
        glEnd();

        if (p<0)
        {
            p=p+2*x+3;
            x=x+1;
        }

    else
        {
            p=p+2*(x-y)+ 5;
            x=x+1;
        }
    }
```

```

        y=y-1;
    }
}

glFlush();
}

void init(void)
{
    glClearColor(0.0,0.0,0.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-100,100,-100,100);
}

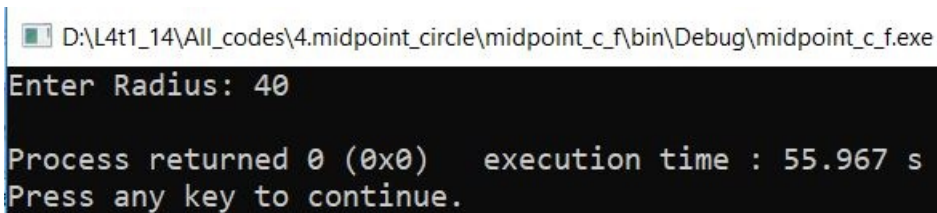
int main(int ac,char** av)
{
    printf("Enter Radius: ");
    scanf("%d",&r);

    glutInit(&ac, av);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Circle ");

    init();
    glutDisplayFunc(Display);
    glutMainLoop();
    return 0;
}

```

4.4 Input-Output



```

D:\L4t1_14\All_codes\4.midpoint_circle\midpoint_c_f\bin\Debug\midpoint_c_f.exe
Enter Radius: 40

Process returned 0 (0x0)   execution time : 55.967 s
Press any key to continue.

```

Figure 7: User Input

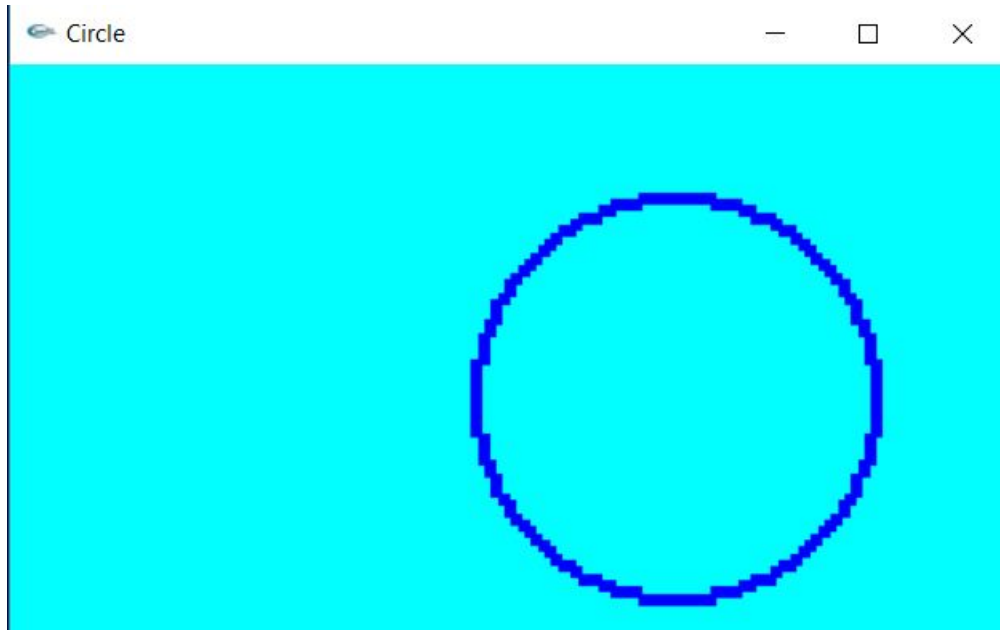


Figure 8: Midpoint Circle Drawing Output.

4.5 Conclusion

From this experiment, we learnt about the Midpoint circle algorithm and its implementation.

- We find a circle that create according to the codes are generated in the opengl.
- We observed the difference between midpoint and bresenham circle.
- We know about opengl and its implantation.

5 Midpoint Ellipse Algorithm

5.1 Objective

The main objective of this experiment is to Scan convert an ellipse using Midpoint Ellipse Algorithm.

5.2 Introduction

The ellipse, like the circle, shows symmetry. In the case of ellipse, symmetry is four rather than eight-way. There are two methods of mathematically defining an ellipse.

1) Polynomial Method of Defining an Ellipse

2) Trigonometric Method of Defining an Ellipse

Since the ellipse shows four-ways symmetry, it can easily be rotated 90° . The new equation is found by trading a and b, the values which describe the major and minor axis. When the polynomial method is used, the equation becomes :

$$(x - h)^2/b^2 + (y - k)^2/a^2 = 1 \quad (3)$$

where, (h,k) = ellipse center a = length of major axis b = length of minor axis when, the trigonometric method is used, the equation becomes,

$$x = b\cos\theta + h \quad (4)$$

$$y = a\sin\theta + k \quad (5)$$

The mid-point ellipse algorithm is an incremental algorithm for scan converting an ellipse that is centered at the origin in standard position. It works in a way that is very similar to the midpoint circle algorithm. However, because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant. Here the ellipse equation and function f that can be used to decide if the midpoint between candidate pixels is inside or outside the ellipse:

$$f_{cir}(x, y) = b^2x^2 + a^2y^2 - a^2b^2 \quad (6)$$

5.3 Source Code

```
#include<windows.h>
#include<bits/stdc++.h>
#include<GL/glut.h>
#include<stdlib.h>
#include<stdio.h>
using namespace std;

int xc,yc,a,b;
void pp(int x,int y)
{
    glBegin(GL_POINTS);
        glVertex2i(x,y);
    glEnd();
}
void getpixel(int x,int y)
{
    pp(xc+x,yc+y);pp(xc-x,yc+y);pp(xc-x,yc-y);pp(xc+x,yc-y);
}

void display ()
{

```

```

//int p0;//=1-Radius;
int x=0;
int y=b;
int e=a*a;
int g=b*b;
int fx=0;
int fy=2*e*b;

int p0=g-e*b+0.25*e;
while(fx<fy){
    getpixel(x,y);
    x++;
    fx=fx+2*g;
    if(p0<0){
        p0=p0+fx+g;
    }
    else{
        y--;
        fy=fy-2*e;
        p0=p0+fx+g-fy;
    }

}
getpixel(x,y);
p0=g*(x+0.5)*(x+0.5)+e*(y-1)*(y-1)-e*g;
while(y>0)
{
    y--;
    fy=fy-2*e;
    if(p0>=0)
        p0=p0-fy+e;
    else{
        x++;
        fx=fx+2*e;
        p0=p0+fx-fy+e;
    }
    getpixel(x,y);
}
glFlush();
}

void init(void)
{
    glClearColor(0.7,0.7,0.7,0.7);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-100,100,-100,100);
}

int main (int argc, char **argv)
{
    cout<<"Give Center Co-ordinates"<<endl;
    cin>>xc>>yc;

```

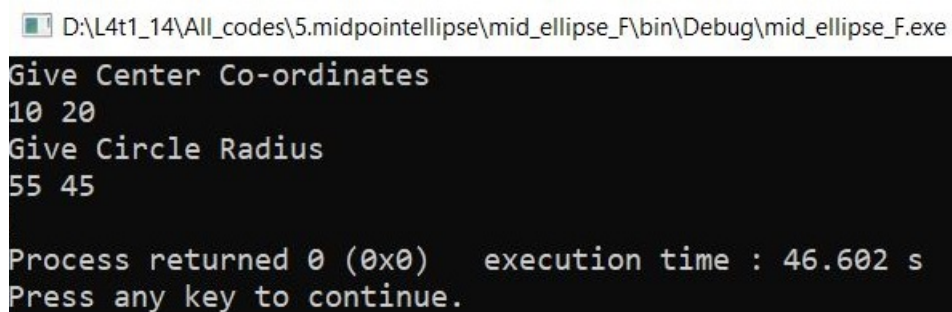
```

cout<<"Give Circle Radius"<<endl;
cin>>a>>b;

glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(100,100);
glutCreateWindow("MidPoint Ellipse Algorithm");
init();
glutDisplayFunc(display);
glutMainLoop();
}

```

5.4 Input-Output



```

D:\L4t1_14\All_codes\5.midpointellipse\mid_ellipse_F\bin\Debug\mid_ellipse_F.exe
Give Center Co-ordinates
10 20
Give Circle Radius
55 45

Process returned 0 (0x0)   execution time : 46.602 s
Press any key to continue.

```

Figure 9: User Input

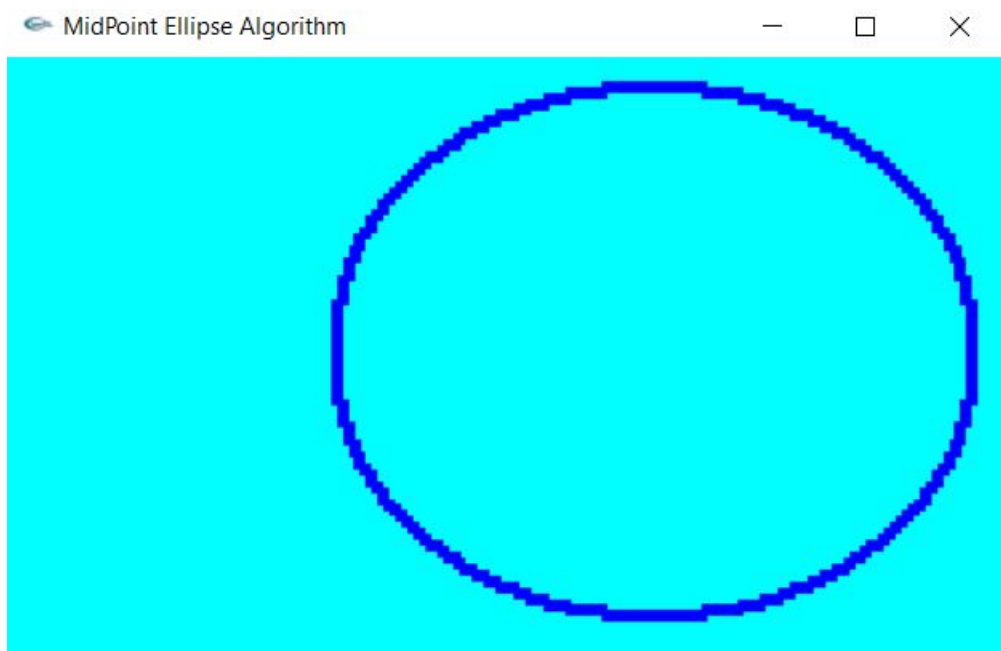


Figure 10: Midpoint Circle Drawing Output.

5.5 Discussion

From this experiment we knew about the Midpoint Ellipse Algorithm and its implementation.

-We find a ellipse that create according to the codes are generated in the opengl.

-We know about opengl and its implantation.

6 Boundary Fill Algorithm

6.1 Objective

A seed point is given inside an object which is referred as polygon. From this point coordinate in our lab of Computer Graphics, we have to fill the object using Boundary fill algorithm.

6.2 Introduction

There are many types of algorithms that are made use of in computer graphics for the purpose of painting figures. Boundary filling is one of them.

In Boundary filling a seed point is fixed, and then neighboring pixels are checked to match with the boundary color. Then, color filling is done until boundary is reached. A region may be 4 connected or 8 connected.

6.3 Description

Boundary fill algorithm is an area filling algorithm. Here, the approach is to start at a point inside a region and paint the interior outward toward the boundary. If the boundary is specified in a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered. A boundary-fill procedure accepts as input the coordinate of the interior point (x, y), a fill color, and a boundary color.

This algorithm works elegantly on an arbitrarily shaped region by chasing and filling all non boundary pixel that are connected to the seed directly or indirectly through a chain of neighboring relation.

6.4 Algorithm

The following steps illustrate the idea of the recursive boundary-fill algorithm:

Step 1: We start from an interior point referred as seep point.

Step 2: If the current pixel is not already filled and if it is not an edge point, then set the pixel with the fill color, and store its neighboring pixels (4 or 8-connected) in the stack for processing. Store only neighboring pixel that is not already filled and is not an edge point.

Step 3: We select the next pixel from the stack, and continue with step 2.

The order of pixels that should be added to stack using 4-connected is above, below, left, and right. For 8-connected is above, below, left, right, above-left, above-right, below-left, and below-right.

6.5 Source Code

```
#include<windows.h>
#include <iostream>
#include <math.h>
#include <time.h>
#include <GL/glut.h>

using namespace std;

struct Point {
    GLint x;
    GLint y;
};

struct Color {
    GLfloat r;
    GLfloat g;
    GLfloat b;
};
```

```

void init() {
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glColor3f(0.0, 0.0, 0.0);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 640, 0, 480);
}

Color getPixelColor(GLint x, GLint y) {
    Color color;
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, &color);
    return color;
}

void setPixelColor(GLint x, GLint y, Color color) {
    glColor3f(color.r, color.g, color.b);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}

void BoundaryFill(int x, int y, Color fillColor, Color boundaryColor) {
    Color currentColor = getPixelColor(x, y);
    if(currentColor.r != boundaryColor.r && currentColor.g != boundaryColor.g &&
    currentColor.b != boundaryColor.b) {
        setPixelColor(x, y, fillColor);
        BoundaryFill(x+1, y, fillColor, boundaryColor);
        BoundaryFill(x-1, y, fillColor, boundaryColor);
        BoundaryFill(x, y+1, fillColor, boundaryColor);
        BoundaryFill(x, y-1, fillColor, boundaryColor);
    }
}

void onMouseClick(int button, int state, int x, int y)
{
    Color fillColor = {1.0f, 0.0f, 0.0f};
    Color boundaryColor = {0.0f, 0.0f, 0.0f};
    Point p = {321, 241};
    BoundaryFill(p.x, p.y, fillColor, boundaryColor);
}

void draw_dda(Point p1, Point p2) {
    GLfloat dx = p2.x - p1.x;
    GLfloat dy = p2.y - p1.y;
    GLfloat x1 = p1.x;
    GLfloat y1 = p1.y;
    GLfloat step = 0;

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

```

```

step = abs(dx);
}

        else {
            step = abs(dy);
        }

GLfloat xInc = dx/step;
GLfloat yInc = dy/step;

for(float i = 1; i <= step; i++) {
    glVertex2i(x1, y1);
    x1 += xInc;
    y1 += yInc;
}
}

void draw_square(Point a, GLint length) {
    Point b = {a.x + length, a.y},
            c = {b.x, b.y+length},
            d = {c.x-length, c.y};

    draw_dda(a, b);
    draw_dda(b, c);
    draw_dda(c, d);
    draw_dda(d, a);
}

void display(void) {

    Point pt = {320, 240};
    GLfloat length = 100;
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
        draw_square(pt, length);
    glEnd();
    glFlush();
}

int main(int argc, char** argv) {

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(400, 400);
    glutCreateWindow("Boundary Fill");
    init();
    glutDisplayFunc(display);
    glutMouseFunc(onMouseClicked);
    glutMainLoop();
    return 0;
}

```


6.6 Sample Input-Output



Figure 11: Boundary Fill Output(Before Filling)



Figure 12: Boundary Fill Output(During Filling)

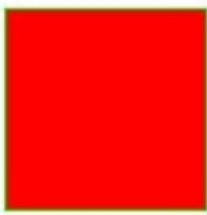


Figure 13: Boundary Fill Output(After Filling)

6.7 Discussion

In this experiment,

- used 4 connected approach to fill the rectangle.
- drew a rectangle to fill.
- The boundary of the rectangle is green colored and the filling color is red.
- For implementating this algorithm, used opengl.
- time consuming algorithm.

7 Flood fill algorithm

7.1 Objective

A seed point is given inside an object which is referred as polygon. From this point coordinate in our lab of Computer Graphics, we have to fill the object using Flood fill algorithm.

7.2 Introduction

Flood fill algorithm helps in visiting each and every point in a given area. It determines the area connected to a given cell in a multi-dimensional array. Following are some famous implementations of flood fill algorithm are Bucket Fill in Paint, Solving a Maze, Minesweeper etc.

This method shares great similarities in its operating principle with Boundary fill algorithm. It is useful when the region to be filled has no uniformly colored boundary.

7.3 Description

Sometimes we come across an object where we want to fill the area and its boundary with different colors. We can paint such objects with a specified interior color instead of searching for particular boundary color as in boundary filling algorithm. Instead of relying on the boundary of the object, it relies on the fill color. In other words, it replaces the interior color of the object with the fill color. When no more pixels of the original interior color exist, the algorithm is completed. Once again, this algorithm relies on the Four-connect or Eight-connect method of filling in the pixels. But instead of looking for the boundary color, it is looking for all adjacent pixels that are a part of the interior.

7.4 Source Code

```
#include<windows.h>
#include <iostream>
#include <math.h>
#include <time.h>
#include <GL/glut.h>

using namespace std;

struct Point {
    GLint x;
    GLint y;
};

struct Color {
    GLfloat r;
    GLfloat g;
    GLfloat b;
};

void draw_dda(Point p1, Point p2) {

    GLfloat dx = p2.x - p1.x;
    GLfloat dy = p2.y - p1.y;
    GLfloat x1 = p1.x;
    GLfloat y1 = p1.y;
    GLfloat step = 0;
    if(abs(dx) > abs(dy)) {
```

```

step = abs(dx);
}
    else {
step = abs(dy);
}
GLfloat xInc = dx/step;
GLfloat yInc = dy/step;
for(float i = 1; i <= step; i++) {

glVertex2i(x1, y1);
x1 += xInc;
y1 += yInc;
}
}

void init() {

glClearColor(1.0, 1.0, 1.0, 0.0);
glColor3f(0.0, 0.0, 0.0);
glPointSize(1.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, 640, 0, 480);
}

Color getPixelColor(GLint x, GLint y) {

Color color;
glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, &color);
return color;
}

void setPixelColor(GLint x, GLint y, Color color) {

glColor3f(color.r, color.g, color.b);
glBegin(GL_POINTS);
glVertex2i(x, y);
glEnd();
glFlush();
}

void floodFill(GLint x, GLint y, Color oldColor, Color newColor) {

Color color;
color = getPixelColor(x, y);
if(color.r == oldColor.r && color.g == oldColor.g && color.b == oldColor.b)
{
setPixelColor(x, y, newColor);
floodFill(x+1, y, oldColor, newColor);
floodFill(x, y+1, oldColor, newColor);
floodFill(x-1, y, oldColor, newColor);
floodFill(x, y-1, oldColor, newColor);
}
return;
}

```

```

}

void onMouseClick(int button, int state, int x, int y) {

Color newColor = {1.0f, 0.0f, 0.0f};
Color oldColor = {1.0f, 1.0f, 1.0f};
floodFill(101, 199, oldColor, newColor);
}

void display(void) {

Point p1 = {100, 100},
p2 = {200, 100},
p3 = {200, 200},
p4 = {100, 200};
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
draw_dda(p1, p2);
draw_dda(p2, p3);
draw_dda(p3, p4);
draw_dda(p4, p1);
glEnd();
glFlush();
}

int main(int argc, char** argv) {

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(640, 480);
glutInitWindowPosition(400, 400);
glutCreateWindow("Flood Fill");
init();
glutDisplayFunc(display);
glutMouseFunc(onMouseClick);
glutMainLoop();
return 0;
}

```

7.5 Output



Figure 14: Flood Fill Output(Before Filling)

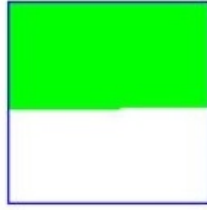


Figure 15: Flood Fill Output(During Filling)

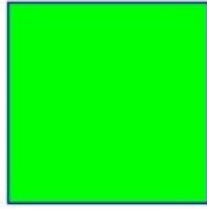


Figure 16: Flood Fill Output(After Filling)

7.6 Discussion

1. During this experiment we learned about boundary fill algorithm and how it fills the any type of boundary.
2. I used 4 connected approach to fill the polygon.
3. I drew a rectangle to fill.
4. I used a rectangular interpolation with this algorithm step by step to compute the rectangle.
5. For implementation this algorithm, I used opengl.
6. It is time consuming algorithm.

8 C Curve Algorithm

8.1 Objective

The main objective of this experiment is to learn about C Curve Algorithm and implement it by finding c curve in opengl platform.

8.2 Introduction

In mathematics, the Lvy C curve is a self-similar fractal that was first described and whose differentiability properties were analysed by Ernesto Cesro in 1906 and Georg Faber in 1910, but now bears the name of French mathematician Paul Lvy, who was the first to describe its self-similarity properties, as well as to provide a geometrical construction showing it as a representative curve in the same class as the Koch curve. If using a Lindenmayer system then the construction of the C curve starts with a straight line. An isosceles triangle with angles of 45^0 , 90^0 and 135^0 is built using this line as its hypotenuse. The original line is then replaced by the other two sides of this triangle.

At the second stage, the two new lines each form the base for another right-angled isosceles triangle, and are replaced by the other two sides of their respective triangle. So, after two stages, the curve takes the appearance of three sides of a rectangle with the same length as the original line, but only half as wide.

At each subsequent stage, each straight line segment in the curve is replaced by the other two sides of a right-angled isosceles triangle built on it. After n stages the curve consists of $2n$ line segments, each of which is smaller than the original line by a factor of $2n/2$.

This L-system can be described as follows,

Variables: F

Constants: +

Start: F

Rules: F \rightarrow FF+

8.3 Source Code

```
#include<windows.h>
#include<GL/glut.h>
#include<bits/stdc++.h>

using namespace std;
float x, y, len, alpha;
int n;

void line (float x1, float y1, float x2, float y2)
{
    glVertex2f(x1,y1);
    glVertex2f(x2,y2);
}

void c_curve (float x, float y, float len, float alpha, int n) {
    if(n > 0){

        len = len / sqrt(2.0);
        c_curve(x, y, len, alpha+45, n-1);
        x += len*cos(alpha+45);
        y += len*sin(alpha+45);
        c_curve(x, y, len, alpha-45, n-1);
    }
}
```

```

    }
    else{
        line(x, y, x+len*cos(alpha), y+len*sin(alpha));
    }
}

void myDisplay(void) {

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 0.0, 0.0);
    glPointSize(1);
    glBegin(GL_LINES);

    c_curve(x, y, len, alpha, n);

    glEnd();
    glFlush ();
}

void init (void) {

    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(0.7,0.7,0.7,0.7);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-200,500,-200,500);
}

int main(int argc, char** argv) {

    cout<<"Co-ordinate of c_curve: ";
    cin>>x>>y;
    cout<<"\nLength: ";
    cin>>len;
    cout<<"\nAngle: ";
    cin>>alpha;
    cout<<"\nValue of n: ";
    cin>>n;

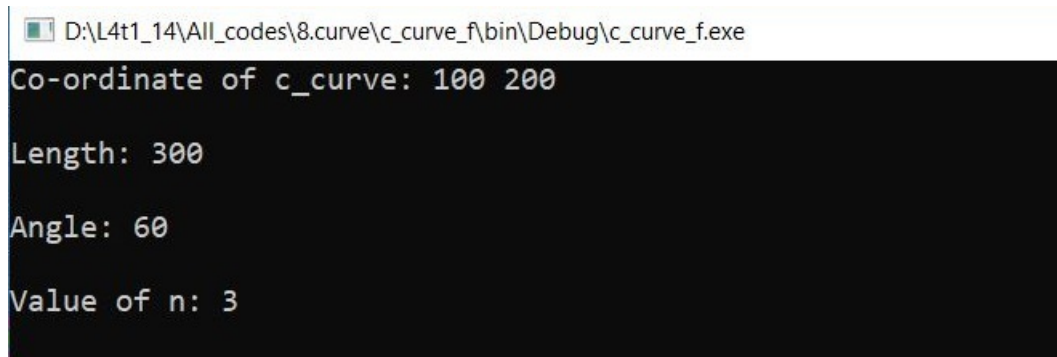
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("C_CURVE");

    init();
    glutDisplayFunc(myDisplay);
    glutMainLoop();

    return 0;
}

```

8.4 Input-Output



```
D:\L4t1_14\All_codes\8.curve\c_curve_f\bin\Debug\c_curve_f.exe
Co-ordinate of c_curve: 100 200
Length: 300
Angle: 60
Value of n: 3
```

Figure 17: User Input

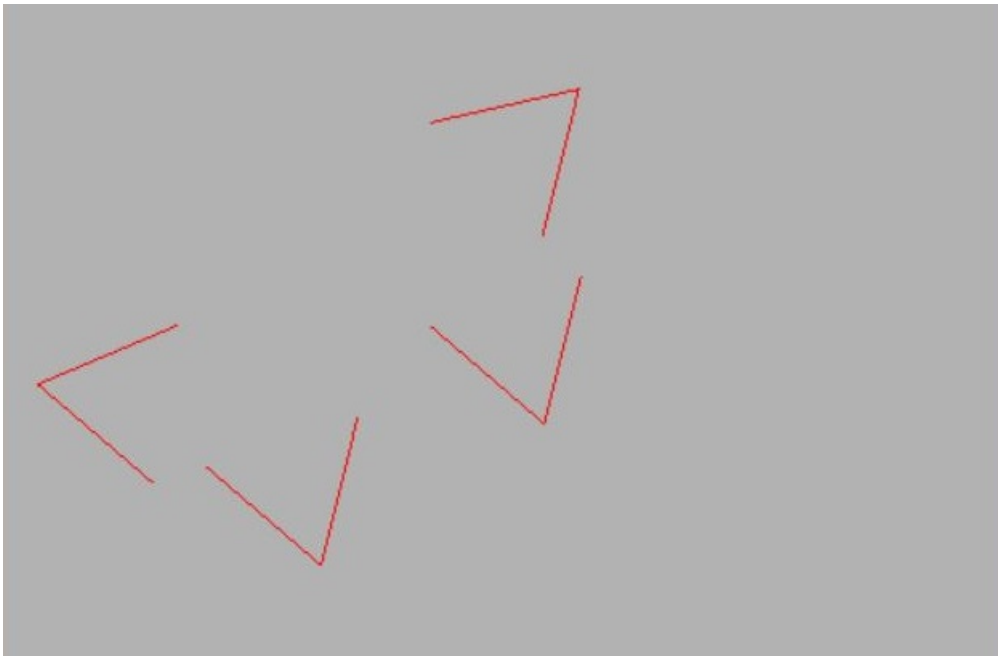


Figure 18: C Curve Output

8.5 Discussion

From this experiment, we knew about the C curve algorithm and its implementation.

- We find a circle that create according to the codes are generated in the opengl.

- We know about opengl and its implantation.

9 Koch Curve Algorithm

9.1 Objective

The main objective of this experiment is to implement the Koch curve algorithm.

9.2 Introduction

The Koch snowflake is a mathematical curve and one of the earliest fractal curves to have been described. The Koch can be constructed by starting with an equilateral triangle, then recursively altering each line segment as follows:

1. divide the line segment into three segments of equal length.
2. draw an equilateral triangle that has the middle segment from step 1 as its base and points outward.
3. remove the line segment that is the base of the triangle from step 2.

After one iteration of this process, the resulting shape is the outline of a hexagram. The perimeter of the Koch curve is increased by $1/4$. That implies that the perimeter after an infinite number of iterations is infinite.

The length of the intermediate curve at the n th iteration of the construction is $(4/3)^n$, where $n = 0$ denotes the original straight line segment. Therefore the length of the Koch curve is infinite. Moreover, the length of the curve between any two points on the curve is also infinite since there is a copy of the Koch curve between any two points. Three copies of the Koch curve placed outward around the three sides of an equilateral triangle form a simple closed curve that forms the boundary of the Koch curve.

9.3 Source Code

```
#include<windows.h>
#include<GL/glut.h>
#include<bits/stdc++.h>

using namespace std;

float x, y, len, alpha;
int n;

void line (float x1, float y1, float x2, float y2) {
    glVertex2f(x1,y1);
    glVertex2f(x2,y2);
}

void koch_curve (float x, float y, float len, float alpha, int n) {

    if(n > 0){
        len = len / 3;
        koch_curve(x, y, len, alpha, n-1);

        x += len*cos(alpha);
        y += len*sin(alpha);
        koch_curve(x, y, len, alpha-120, n-1);
        x += len*cos(alpha-120);
        y += len*sin(alpha-120);
        koch_curve(x, y, len, alpha+120, n-1);
    }
}
```

```

        x += len*cos(alpha+120);
        y += len*sin(alpha+120);
        koch_curve(x, y, len, alpha, n-1);
    }
    else {
        line(x, y, x+len*cos(alpha), y+len*sin(alpha));
    }
}

void myDisplay(void) {

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 0.0, 0.0);
    glPointSize(1);
    glBegin(GL_LINES);

    koch_curve(x, y, len, alpha, n);
    glEnd();
    glFlush ();
}

void init (void) {

    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(0.7,0.7,0.7,0.7);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,250,0,250);
}

int main(int argc, char** argv) {

    cout<<"Co-ordinate of koch_curve: ";
    cin>>x>>y;
    cout<<"\nLength: ";
    cin>>len;
    cout<<"\nAngle: ";
    cin>>alpha;
    cout<<"\nValue of n: ";
    cin>>n;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Koch_CURVE");

    init();
    glutDisplayFunc(myDisplay);
    glutMainLoop();

    return 0;
}

```

9.4 Input-Output

```
D:\L4t1_14\All_codes\9.koch\Koch_f\bin\Debug\Koch_f.exe
Co-ordinate of koch_curve: 50 60
Length: 100
Angle: 45
Value of n: 3
Process returned 0 (0x0)   execution time : 37.507 s
Press any key to continue.
```

Figure 19: User Input

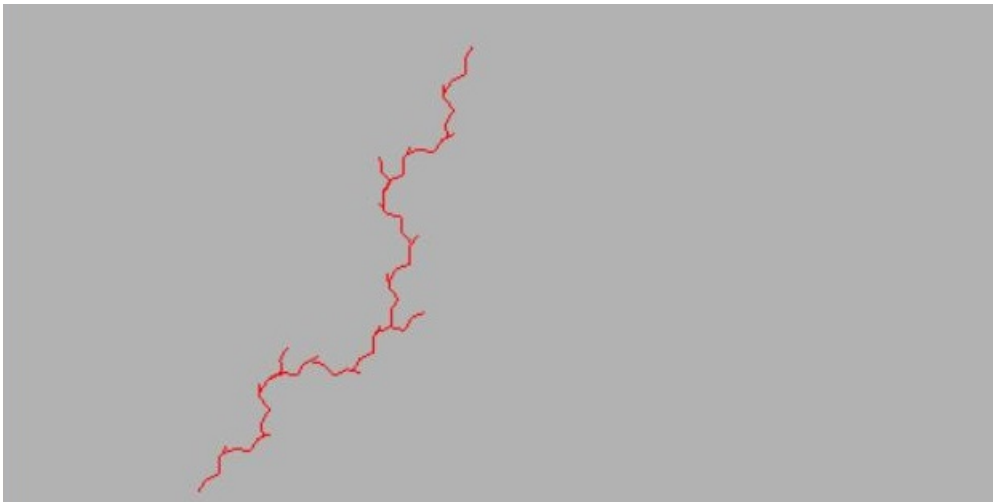


Figure 20: koch Curve Output

9.5 Discussion

From this experiment we have learnt about koch curve fractal

- The Koch curve is the limiting curve obtained by applying this construction an infinite number of times
- We have implemented this problem in opengl.

10 Cohen-Sutherland Algorithm

10.1 Objective

The main objective of this experiment is line clipping using Cohen-Sutherland Algorithm.

10.2 Introduction

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivially accepted or rejected. If the line cannot be trivially accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted. To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions.

Bit 1 : outside halfplane of top edge, above top edge [$Y > Y_{\max}$]

Bit 2 : outside halfplane of bottom edge, below bottom edge [$Y < Y_{\min}$]

Bit 3 : outside halfplane of right edge, to the right of right edge [$X > X_{\max}$]

Bit 4 : outside halfplane of left edge, to the left of left edge [$X < X_{\min}$]

10.3 Algorithm

To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

1. Given a line segment with endpoint and Compute the 4-bit codes for each endpoint. If both codes are 0000, (bitwise OR of the codes yields 0000) line lies completely inside the window: pass the endpoints to the draw routine. If both codes have a 1 in the same bit position (bitwise AND of the codes is not 0000), the line lies outside the window. It can be trivially rejected.

- 2.If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be clipped at the window edge before being passed to the drawing routine.

- 3.Examine one of the endpoints, say. Read 's 4-bit code in order: Left-to-Right, Bottom-to-Top.

- 4.When a set bit (1) is found, compute the intersection I of the corresponding window edge with the line from to. Replace with I and repeat the algorithm.

10.4 Source Code

```
#include <windows.h>
#include<GL/glut.h>
#include<math.h>
#include<stdio.h>
#include<iostream>
using namespace std;

void display();

float xmin=4;
float ymin=2;
float xmax=30;
float ymax=50;
float xd1,yd1,xd2,yd2;

void init(void)
{
```

```

    glClearColor(0.0,0,0,0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-300,300,-300,300);

}

int code(float x,float y)
{
    int c=0;
    if(y>ymax)c=8;//1000
    if(y<ymin)c=4;//0100
    if(x>xmax)c=c|2;//0010
    if(x<xmin)c=c|1;//0001
    return c;
}

void cohen_Line(float x1,float y1,float x2,float y2)
{
    int c1=code(x1,y1);
    int c2=code(x2,y2);
    float m=(y2-y1)/(x2-x1);
    while((c1|c2)>0)
    {
        if((c1 & c2)>0)
        {
            exit(0);
        }

        float xi=x1;float yi=y1;
        int c=c1;
        if(c==0)
        {
            c=c2;
            xi=x2;
            yi=y2;
        }
        float x,y;
        if((c & 8)>0)
        {
            y=ymax;
            x=xi+ 1.0/m*(ymax-yi);
        }
        else
        {
            if((c & 4)>0)
            {
                y=ymin;
                x=xi+1.0/m*(ymin-yi);
            }
            else
            {
                if((c & 2)>0)
                {
                    x=xmax;
                    y=yi+m*(xmax-xi);
                }
            }
        }
    }
}

```

```

        else
        if((c & 1)>0)
        {
            x=xmin;
            y=yi+m*(xmin-xi);
        }

        if(c==c1)
        {
            xd1=x;
            yd1=y;
            c1=code(xd1,yd1);
        }

        if(c==c2)
        {
            xd2=x;
            yd2=y;
            c2=code(xd2,yd2);
        }
    }

    display();
}

void mykey(unsigned char key,int x,int y)
{
    if(key=='c')
    {
        cohen_Line(xd1,yd1,xd2,yd2);
        glFlush();
    }
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,1.0,0.0);
    glBegin(GL_LINE_LOOP);
    glVertex2i(xmin,ymin);
    glVertex2i(xmin,ymax);
    glVertex2i(xmax,ymax);
    glVertex2i(xmax,ymin);
    glEnd();
    glColor3f(1.0,0.0,0.0);
    glBegin(GL_LINES);
    glVertex2i(xd1,yd1);
    glVertex2i(xd2,yd2);
    glEnd();
    glFlush();
}

int main(int argc,char** argv)

```

```

{
    printf("Enter line co-ordinates:");
    cin>>xd1>>y1>>xd2>>y2;

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(600,600);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Clipping");
    glutDisplayFunc(display);
    glutKeyboardFunc(mykey);

    init();
    glutMainLoop();
    return 0;
}

```

10.5 Input-Output

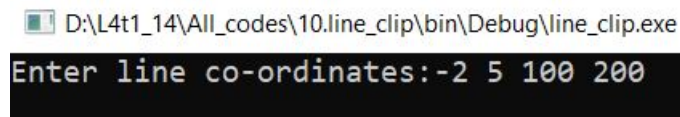


Figure 21: User Input

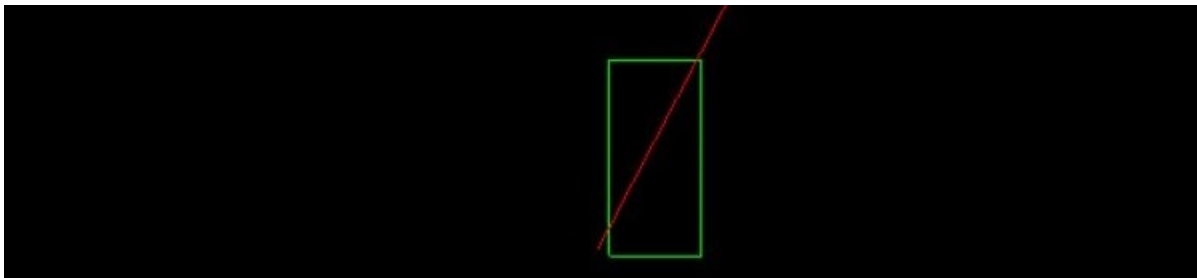


Figure 22: Line Clipping Output(without clipping)

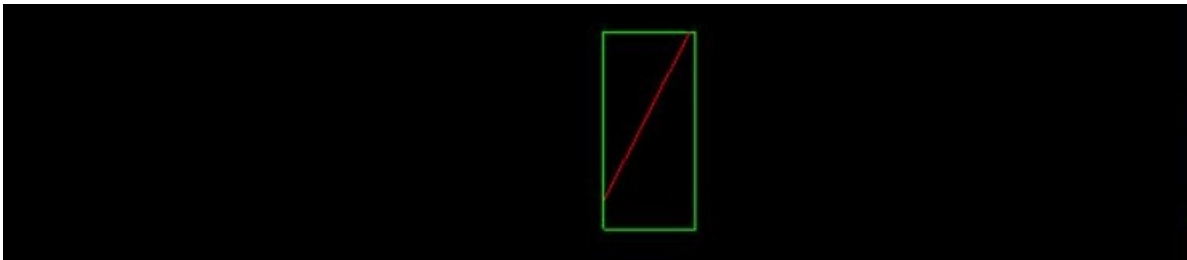


Figure 23: Line Clipping Output(After Clipping)

10.6 Discussion

From this experiment we knew about the Cohen-Sutherland line clipping algorithm and its implementation.

- We get to know how to clip the portion of a line that is outside a rectangle.

- We know about opengl and its implantation.

- This problem was implemented in opengl.

11 Sutherland-Hodgman Algorithm

11.1 Objective

The main objective of this experiment is to implement Sutherland Hodgman Algorithm.

11.2 Introduction

A polygon can be clipped by specifying the clipping window. Sutherland Hodgeman polygon clipping algorithm is used for polygon clipping. In this algorithm, all the vertices of the polygon are clipped against each edge of the clipping window.

First the polygon is clipped against the left edge of the polygon window to get new vertices of the polygon. These new vertices are used to clip the polygon against right edge, top edge, bottom edge, of the clipping window. While processing an edge of a polygon with clipping window, an intersection point is found if edge is not completely inside clipping window and the a partial edge from the intersection point to the outside edge is clipped.

11.3 Source Code

```
#include <windows.h>
#include <gl/glut.h>

struct Point
{
    float x,y;
} w[4],oVer[4];

int Nout;

void drawPoly(Point p[],int n)
{
    glBegin(GL_POLYGON);

    for(int i=0;i<n;i++)

        glVertex2f(p[i].x,p[i].y);

    glEnd();
}

bool insideVer(Point p)
{
    if((p.x>=w[0].x)&&(p.x<=w[2].x))

        if((p.y>=w[0].y)&&(p.y<=w[2].y))

            return true;

    return false;
}
```

```

void addVer(Point p)
{
    oVer[Nout]=p;

    Nout=Nout+1;
}

```

```

Point getInterSect(Point s,Point p,int edge){

    Point in;

    float m;

    if(w[edge].x==w[(edge+1)%4].x){ //Vertical Line

        m=(p.y-s.y)/(p.x-s.x);

        in.x=w[edge].x;

        in.y=in.x*m+s.y;
    }
    else{//Horizontal Line
        m=(p.y-s.y)/(p.x-s.x);
        in.y=w[edge].y;
        in.x=(in.y-s.y)/m;
    }
    return in;
}

```

```

void clipAndDraw(Point inVer[],int Nin){
    Point s,p,interSec;
    for(int i=0;i<4;i++)
    {
        Nout=0;
        s=inVer[Nin-1];
        for(int j=0;j<Nin;j++)
        {
            p=inVer[j];
            if(insideVer(p)==true){
                if(insideVer(s)==true){
                    addVer(p);
                }
                else{
                    interSec=getInterSect(s,p,i);
                    addVer(interSec);
                    addVer(p);
                }
            }
            else{

```

```

        if(insideVer(s)==true){
            interSec=getInterSect(s,p,i);
            addVer(interSec);
        }
    }
    s=p;
}
inVer=oVer;
Nin=Nout;
}
drawPoly(oVer,4);
}

```

```

void init(){

```

```

    glClearColor(0.0f,0.0f,0.0f,0.0f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,100.0,0.0,100.0,0.0,100.0);
    glClear(GL_COLOR_BUFFER_BIT);
    w[0].x =20,w[0].y=10;
    w[1].x =20,w[1].y=80;
    w[2].x =80,w[2].y=80;
    w[3].x =80,w[3].y=10;
}

```

```

void display(void){
    Point inVer[4];
    init();
    // As Window for Clipping
    glColor3f(0.0f,1.0f,1.0f);
    drawPoly(w,4);
    // As Rect
    glColor3f(0.5f,0.0f,1.0f);
    inVer[0].x =10,inVer[0].y=40;
    inVer[1].x =10,inVer[1].y=60;
    inVer[2].x =60,inVer[2].y=60;
    inVer[3].x =60,inVer[3].y=40;
    drawPoly(inVer,4);
    // As Rect
    glColor3f(1.0f,0.0f,0.0f);
    clipAndDraw(inVer,4);
    // Print
    glFlush();
}

```

```

int main(int argc,char *argv[]){
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(400,400);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Polygon Clipping!");
}

```

```
    glutDisplayFunc(display);  
    glutMainLoop();  
    return 0;  
}
```

11.4 Output

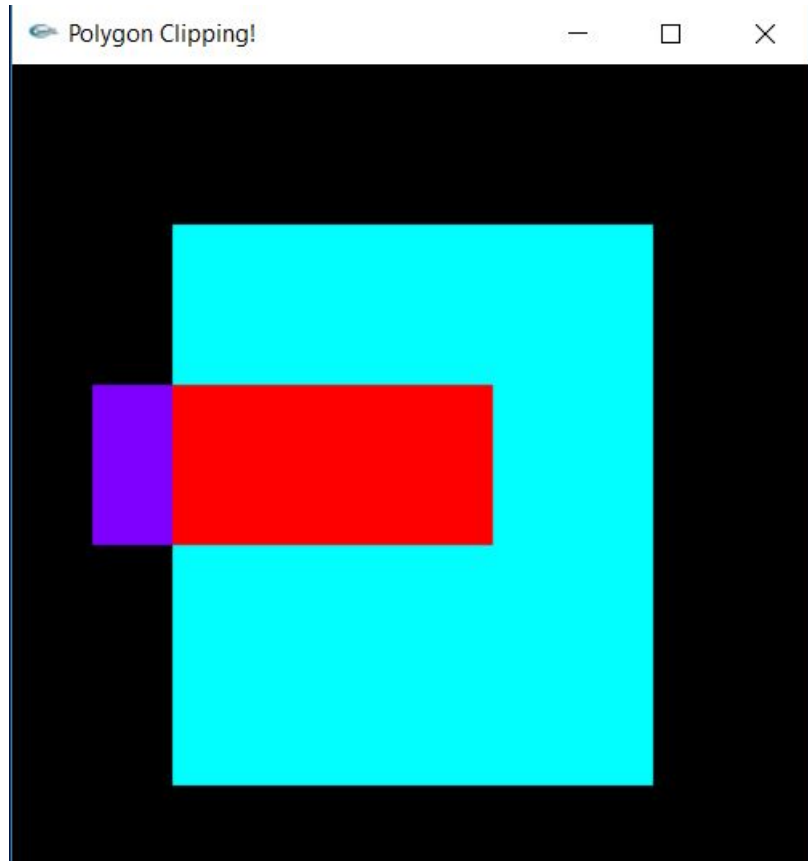


Figure 24: Polygon Clipping Output(After Clipping)

11.5 Discussion

- From this experiment, we have learnt about sutherland hodgement algorithm.
- This algorithm always produces a single output polygon, even if the clipped polygon is concave and arranged in such a way that multiple output polygons might reasonably be expected.
- We implemented the problem using opengl.