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

Course Code: CSE - 458
Course Title: Computer Graphics (Sessional)
Project Title: Simulation of Monopoly game.

Submitted to: Farzana Yasmin
Lecturer, Dept. of CSE, CUET.

Paresh Das ID:1404113[1] and MD Shahadat Hossain, ID: 1404120[2]

[1]Department of Computer Science & Engineering, CUET

Thursday 21st March, 2019

Contents

List of Figures

1 Objective

To develop a suitable graphics package to simulate the Monopoly game using OpenGL.

2 Introduction

Computer graphics is an art of drawing pictures, lines, charts, etc using computers with the help of programming. Computer graphics is made up of number of pixels. Pixel is the smallest graphical picture or unit represented on the computer screen. Basically there are two types of computer graphics namely.

2.1 Interactive Computer Graphics:

Interactive Computer Graphics involves a two way communication between computer and user. Here the observer is given some control over the image by providing him with an input device for example the video game controller of the ping pong game. This helps him to signal his request to the computer.

The computer on receiving signals from the input device can modify the displayed picture appropriately. To the user it appears that the picture is changing instantaneously in response to his commands. He can give a series of commands, each one generating a graphical response from the computer. In this way he maintains a conversation, or dialogue, with the computer.

Interactive computer graphics affects our lives in a number of indirect ways. For example, it helps to train the pilots of our airplanes. We can create a flight simulator which may help the pilots to get trained not in a real aircraft but on the grounds at the control of the flight simulator. The flight simulator is a mock up of an aircraft flight deck, containing all the usual controls and surrounded by screens on which we have the projected computer generated views of the terrain visible on take off and landing.

Flight simulators have many advantages over the real aircrafts for training purposes, including fuel savings, safety, and the ability to familiarize the trainee with a large number of the world's airports.

2.2 Non Interactive Computer Graphics:

In non interactive computer graphics otherwise known as passive computer graphics. it is the computer graphics in which user does not have any kind of control over the image. Image is merely the product of static stored program and will work according to the instructions given in the program linearly. The image is totally under the control of program instructions not under the user. Example: screen savers.

2.3 OpenGL

As a software interface for graphics hardware, OpenGL's main purpose is to render two- and three-dimensional objects into a frame buffer. These objects are described as sequences of vertices (which define geometric objects) or pixels (which define images). OpenGL performs several processing steps on this data to convert it to pixels to form the final desired image in the frame buffer.

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so you can write a single OpenGL program that works on both Win32 PCs and X11 workstations[**?**].

GLUT is designed for constructing small to medium sized OpenGL programs. While GLUT is well-suited to learning OpenGL and developing simple OpenGL applications, GLUT is not a full-featured toolkit so large applications requiring sophisticated user interfaces are better off using native window system toolkits like Motif. GLUT is simple, easy, and small.

3 Overview of The Project

Monopoly is a board game that is currently published by Hasbro. In the game, players roll two six-sided dice to move around the game board, buying and trading properties, and developing them with houses and hotels. Players collect rent from their opponents, with the goal being to drive them into bankruptcy. Money can also be gained or lost through Chance and Community Chest cards, and tax squares; players can end up in jail, which they cannot move from until they have met one of several conditions. The game has numerous house rules, and hundreds of different editions exist, as well as many spin-offs and related media. Monopoly has become a part of international popular culture, having been licensed locally in more than 103 countries and printed in more than 37 languages.
In this work we have to visualize the Monopoly game.

4 Requirements Specification

1. Operating System : Windows 10
2. Programming Language : C++
3. Toolkit : Glut Toolkit

5 Design

In this project, The body is segmented in different parts as follows:

1. Body of the board is implemented by a many polygons with many color.
2. Body of dices are implemented by polygon.
3. Body of coins are implemented by polygon.

6 OpenGL Functions

The project is implemented by using the below OpenGL functions:

6.1 Initialization Function

- **glutInitDisplayMode(unsigned int mode)** - Requests a display with the properties in mode. The value of mode is determined by the logical OR.
- **glutInitWindowPosition(int x, int y)** - Specifies the screen location for the upper-left corner of your window.
- **glutInitWindowSize(int width, int size)** - Specifies the size, in pixels, of your window.

6.2 Event Processing

- **void glutMainLoop()** - Cause the program to enter an event processing loop.

6.3 Window Management Function

Six routines perform tasks necessary to initialize a window.

- **glutInit(int *argc, char **argv)** - Initializes GLUT. The arguments from main are passed in by the application.
- **glutPostRedisplay()** - Marks the current window as needing to be redisplayed.
- **glutSwapBuffers()** - Performs a buffer swap on the layer in use for the current window.
- **int glutCreateWindow(char *string)** - Creates a window on display. The string title can be used to label the window.

6.4 Call Back Function

- **glutDisplayFunc(void (* func)(void))** - Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by glutDisplayFunc() is executed.
- **glutTimerFunc(unsigned int msecs, void(*func)(int value), value)** - Registers a timer callback to be triggered in a specfic number of miliseconds.

6.5 Transformation Function

- **void glTranslate(TYPEx, TYPEy, TYPEz)** - The glTranslated and glTranslatef functions multiply the current matrix by a translation matrix.

6.6 Interactive Function

- **void glPushMatrix() & void glPopMatrix** - Push to and pops from the matrix stack corresponding to the current matrix mode.
- **void glOrtho(GLdouble left, GLdouble right, GLdouble top, GLdouble bottom, GLdouble near, GLdouble far)** - Defines an Orthographic viewing volume with all parameters measured from the center of the projection plane.

6.7 Other OpenGL Functions

- **glMatrixMode(GL_Projection)** - Specifies which matrix is the current matrix.
- **glClearColor(r,g,b,alpha)** - specfies thered, green, blue and alpha values used by glClear to claear the color values.
- **gCollor3f(GLfloat red, GLfloat green, GLfloat blue)** - Specifies new red, green, blue values for the current color.
- **glVertex2f(GLfloat x, GLfloat y)** - Specifies x & y co-ordinates of a vertex.
- **glLoadIdentity()** - Repleces the current matrix with the identity matrix.
- **glFlush()** - Forces execution of GL commands in finite time.
- **glBegin() & glEnd()** - Delimit the vertices that define a primitve or a group of like primitives.

7 Implementation

Sample code for polygons:

```
glBegin(GL_POLYGON);
      glVertex2f (0+500, 0+50 );
      glVertex2f (50+500, 0+50);
      glVertex2f (50+500, 50+50);
      glVertex2f (0+500, 50+50);
glEnd();
glFlush();
```

Sample code for writing text:

```
char menu[8000];
    strcpy(menu,"Text/tSpaces/nNewline");
    int len,x=0,y=28;
    len = strlen(menu);
    glColor3f(0,0,0);
    glMatrixMode( GL_PROJECTION );
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D( -50, 600, -50, 600 );
    glMatrixMode( GL_MODELVIEW );
    glPushMatrix();
    glLoadIdentity();
    glRasterPos2i(x, y);
    for ( int i = 0; i < len; ++i )
       {
  if(menu[i]=='\n')
  {   x=0;
      y-=15;
  glRasterPos2i(x,y);
  }
else  if(menu[i]=='\t')
  {   x+=50;
  glRasterPos2i(x,y);
  }
  else
 glutBitmapCharacter(GLUT_BITMAP_9_BY_15, menu[i]);
}
    glPopMatrix();
    glMatrixMode( GL_PROJECTION );
    glPopMatrix();
    glMatrixMode( GL_MODELVIEW );

\
```

Sample code for showing dice:

```
void dice2(int y)
{
glPointSize(5);
glColor3f(1,1,1);
glBegin(GL_POLYGON);
glVertex2f(300,75);
```

```
glVertex2f(350,75);
glVertex2f(350,125);
glVertex2f(300,125);
glEnd();
glColor3f(0,0,0);
switch(y)
{
case 0:break;
case 1:
glBegin(GL_POINTS);
glVertex2f(300+25,100);
glEnd();
break;
    case 2:glBegin(GL_POINTS);
glVertex2f(300-10+25,100);
            glVertex2f(300+10+25,100);
glEnd();
         break;
 case 3:glBegin(GL_POINTS);
glVertex2f(300-10+25,100+10);
            glVertex2f(300+25,100);
glVertex2f(300+10+25,100-10);
glEnd();
         break;
case 4:glBegin(GL_POINTS);
glVertex2f(300+15,100+15);
            glVertex2f(300+35,100+15);
glVertex2f(300+15,100-15);
glVertex2f(300+35,100-15);
glEnd();
         break;
case 5:glBegin(GL_POINTS);
glVertex2f(300+15,100+15);
            glVertex2f(300+35,100+15);
glVertex2f(300+15,100-15);
glVertex2f(300+35,100-15);
glVertex2f(300+25,100);
glEnd();
        break;
case 6:glBegin(GL_POINTS);
glVertex2f(300+15,100+15);
            glVertex2f(300+35,100+15);
glVertex2f(300+15,100-15);
glVertex2f(300+35,100-15);
glVertex2f(300+15,100);
glVertex2f(300+35,100);
glEnd();
         break;

}
glFlush();
}
```

Sample Code for keyboard input:

```
void key(unsigned char key,int x,int y)
{

if(key=='q' || key=='Q')
call(0);
if(key=='1')
call(1);
if(key=='2')
call(2);
glutPostRedisplay();
}
```

Sample code for rolling dice:

```
void rolldice()
{
srand(time(0));
v2 = 1+(rand()% 6) ;
v1 =1+(rand()%6);
displaydice();
}
void call(int w)
{
    switch(w)
  {
    case 1:
         rolldice();
                  break;
            case 0:
                 exit(0);
     break;
            case 2:
         rolldice();
                 break;
  }
  glFlush();
}
```

Sample Code for initialization:

```
void init(void)
{
glClearColor(.1,0.1,0.1,0.0);
glOrtho(-50.0, 600.0, -50.0, 600.0, 0.0, 1000.0);
}
```

Sample code of main() function:

```
int main()
{
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize (900, 700);
glutInitWindowPosition (250, 0);
glutCreateWindow ("mist");
init();
glutKeyboardFunc(key);
```

```
glutDisplayFunc(displaydice);
    glutDisplayFunc(displayguti1);
 glutDisplayFunc(display);

glutMainLoop();
return 0;
}
```

8 Output Shows the board of monopoly game & and a stage of the game :

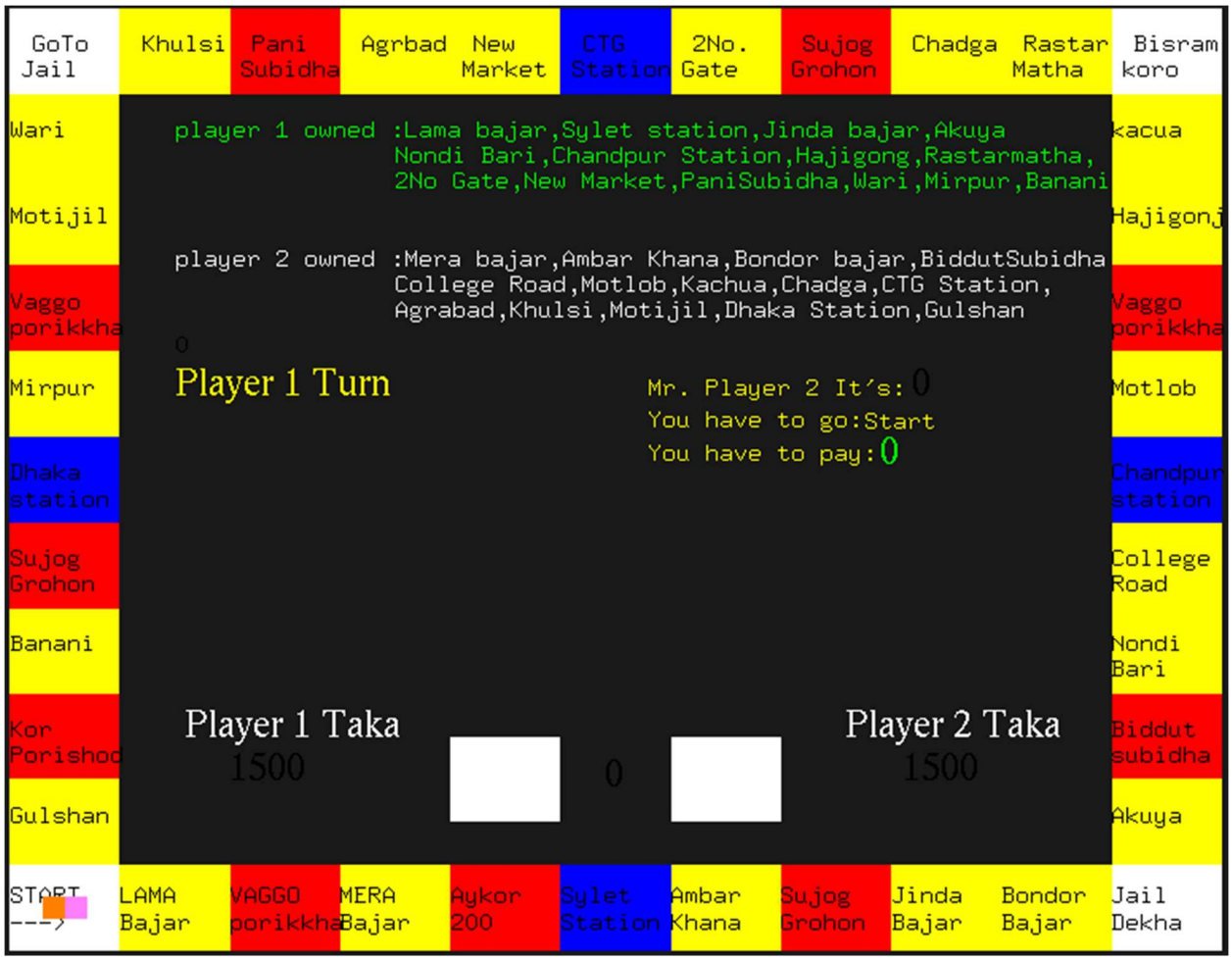

Figure 1: Initial state of the System

Figure 2: After passing some stage

9 Conclusion

1. The development of this project would improve the user's knowledge about computer graphics and OpenGL.
2. Keeping the factors of usability in mind we have develop this project to provide ease of use. This project will allow user to interact with it through the use of devices like keyboard or mouse.
3. This is a project mainly concern with learning computer graphics for drawing and movement of shape.
4. Some feature are not included in this project further improvement needed to this project

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

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

Submitted to: Farzana Yasmin
Lecturer, Dept. of CSE, CUET.

MD Shahadat Hossain, ID: 1404120[1]

[1]Department of Computer Science & Engineering, CUET

Wednesday 20th March, 2019

Contents

List of Figures

1 DDA Line Drawing Algorithm

1.1 Objective

To scan convert a line digital differential analyzer (DDA) using digital differential analyzer (DDA) line drawing algorithm.

1.2 Introduction

A line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. It is defined by its two endpoints and the line equation y = mx + b, where m is called the slope and b the y intercept of the line.

The digital differential analyzer (DDA) algorithm is an incremental scan-convertion method. Such as approach is characterized by performing calculations at each step using results from the preceding step. Suppose at step i we have calculated (Xi,Yi) to be a point on the line. Since the next point (Xi+1,Yi+1) should satisfy dy/dx = m where dy = Yi+1-Yi, and dx = Xi+1-Xi, we have

Yi+1 = Yi+ mdx
or
Xi+1 =Xi + dy/m

These formulas are used in the DDA algorithm as follows. When [m]¡= 1. we start with x = x1', and y = y1' and set dx=1. The y coordinate of each successive point on the line is calculated using Yi+1=Yi+m. When $|m| > 1$, we start with x = x1', and y = y1',and set dy = 1. The x coordinate of each successive points on the line is calculated using Xi+1 = Xi, + 1/m. This process continues until x reaches x2'; (for the $|m| < 1$ case) or y reaches y2' (for the $|m| > 1$ case) and all points found are scan-converted to pixel coordinates.

The DDA algorithm is faster than the direct use of the line equation since it calculates points on the line without any floating-point multiplication. However, a floating-point addition is still needed in determining each successive point.

1.3 Source Code

```
#include <windows.h>
#include <gl/glut.h>
#include <math.h>
#include <stdio.h>
#include<iostream>
using namespace std;
float x1,y12,x2,y13;
void drawLine(int x0,int y0,int x1,int y1){
    glPointSize(2);
    glBegin(GL_POINTS);
    glColor3f(0.0,0.0,0.0);
    double m,y,x;
     m=(double)(y1-y0)/(x1-x0);
    y=(double)y0;
     x=(double)x0;
     cout<<m;
    if(m<1)    {
        while(abs(x)<=abs(x1))        {
            glVertex2d(x,floor(y));
            printf("%f %f\n",x,floor(y));
```

```
            y=y+m;
            x++;
        }
    }
    else    {
        double m1=1/m;
        while(abs(y)<=abs(y1))        {
            glVertex2d(floor(x),y);
            printf("%f %f\n",x,floor(y));
            y++;
            x=x+m1;
        }
    }
    glEnd();
}
void init(void){
    glClearColor(0.0,8.0,8.0,8.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-100.0,100.0,-100.0,100,-100.0,100.0);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawLine(x1,y12,x2,y13);
    glutSwapBuffers();
}
int main(int argc, char** argv){
    scanf("%f%f%f%f",&x1,&y12,&x2,&y13);
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize(780,1300);
    glutInitWindowPosition(100,100);
    glutCreateWindow("DDA Line Drawing!");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

1.4 Sample I/O

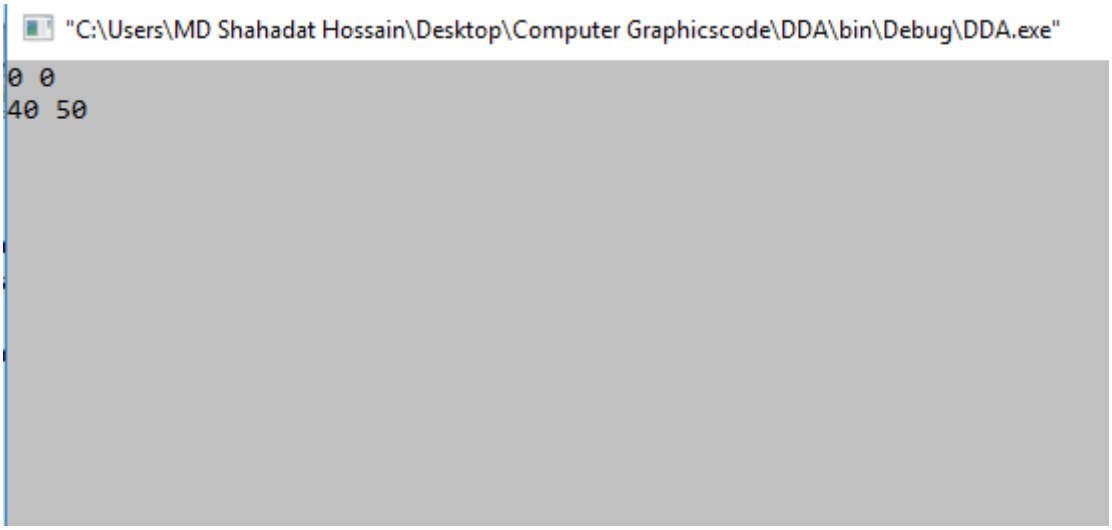

Figure 1: User Input.

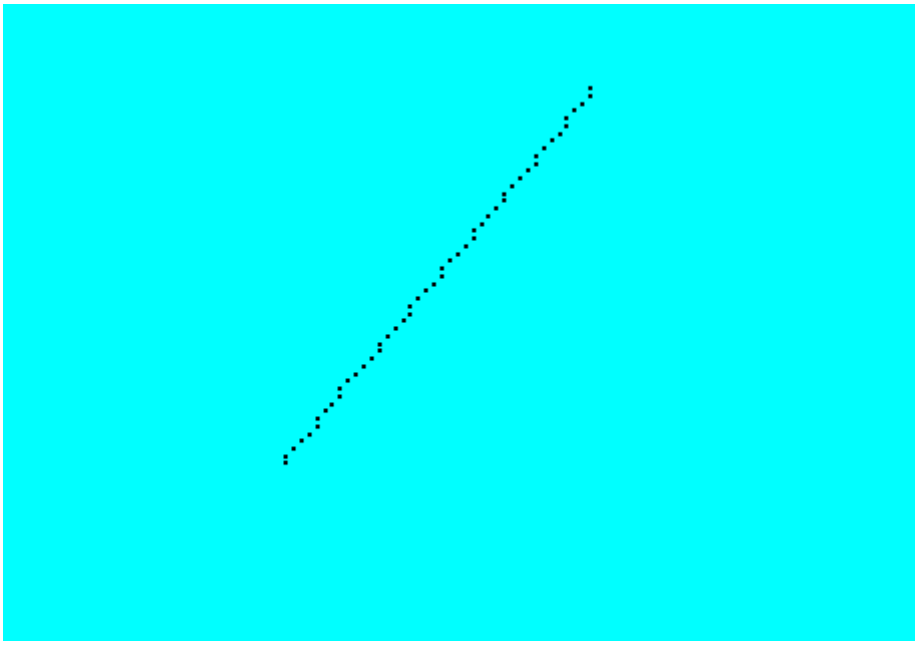

Figure 2: DDA Line Drawing Algorithm.

1.5 Conclusion

1. Familiarization with OpenGL and DDA algorithm has been done effectively.
2. DDA algorithm has been implemented using OpenGL.We have found a line so implementation was successful.
3. DDA algorithm is more time consuming as it uses floating point arithmetic with addition operation.

2 Bresenham Line drawing Algorithm

2.1 Objective

To implement a line using Bresenham's line drawing algorithm.

2.2 Introduction

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.
Bresenham's algorithm for scan-converting a line from $P1(x1', y1')toP2(x2', y2')withx1' < x2'$and $0 < m < 1$ can be stated as follows:

```
int x=x1' ,y=y1';
int dx = x2'-x1', dy = y2'-y1',dT = 2(dy |dx), dS = 2dy;
int d = 2dy - dx;
setPixel(x,y);
while (x < x2') {
x++;
if(d <0)
d=d+dS;
else {
y++;
d=d+dT;
}
setPizel(x,y);
}
```

Here we first initialize decision variable d and set pixel P,. During each iteration of the while loop, we increment x to the next horizontal position, then use the current value of d to select the bottom or top (increment y) pixel and update d, and at the end set the chosen pixel.

2.3 Source Code

```
#include<windows.h>
#include <gl/glut.h>
#include <stdio.h>
int x1, y1, x2, y2;
void myInit() {
glClear(GL_COLOR_BUFFER_BIT);
  glClearColor(0.0,8.0,8.0,8.0);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0, 100,0, 100);
}
void draw_line(int x1, int x2, int y1, int y2) {
     glPointSize(2);
    glBegin(GL_POINTS);
int dx, dy, i, e;
int incx, incy, inc1, inc2;
int x,y;dx = x2-x1;
dy = y2-y1;
if (dx < 0) dx = -dx;
if (dy < 0) dy = -dy;
incx = 1;
if (x2 < x1) incx = -1;
incy = 1;
```

```
if (y2 < y1) incy = -1;
x = x1; y = y1;
if (dx > dy) {
glVertex2i(x, y);
e = 2 * dy-dx;
inc1 = 2*(dy-dx);
inc2 = 2*dy;
for (i=0; i<dx; i++) {
if (e >= 0) {
y += incy;
e += inc1;
}
else
e += inc2;
x += incx;
glVertex2i(x, y);
}

} else {
glVertex2i(x, y);
e = 2*dx-dy;
inc1 = 2*(dx-dy);
inc2 = 2*dx;
for (i=0; i<dy; i++) {
if (e >= 0) {
x += incx;
e += inc1;
}
else
e += inc2;
y += incy;
glVertex2i(x, y);
}
}
glEnd();
}

void myDisplay() {
draw_line(x1, x2, y1, y2);
glFlush();
}

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

printf( "Enter (x1, y1, x2, y2)\n");
scanf("%d %d %d %d", &x1, &y1, &x2, &y2);

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(500, 500);
glutInitWindowPosition(0, 0);
glutCreateWindow("Bresenham's Line Drawing");
myInit();
glutDisplayFunc(myDisplay);
```

```
glutMainLoop();
}
```

2.4 Sample I/O

```
Enter (x1, y1, x2, y2)
0 0
40 50
```

Figure 3: User Input.

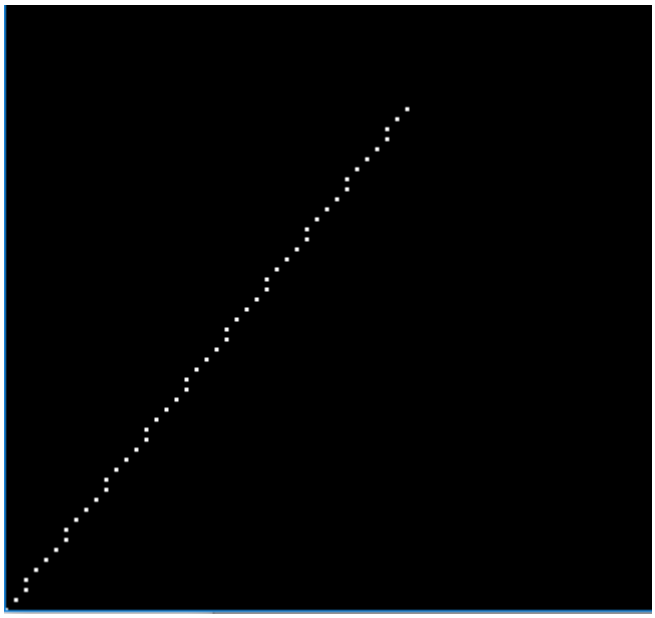

Figure 4: Bresenham Line Drawing Algorithm.

2.5 Conclusion

1. Familiarization with OpenGL has been done effectively.
2. Knowledge about Bresenhams line drawing algorithm has been acquired properly.
3. This algorithm has been implemented using several functions of OpenGL.
4. Bresenhams line algorithm is an fast incremental algorithm, as it uses only integer calculations.

3 Midpoint Circle Algorithm

3.1 Objective

To implement a circle using Midpoint Circle Algorithm.

3.2 Introduction

Circles have the property of being highly symmetrical. So for whole 360^0 of circle, it can be divided in 8 parts each octant of 45^0. Midpoint circle algorithm is based on this idea.

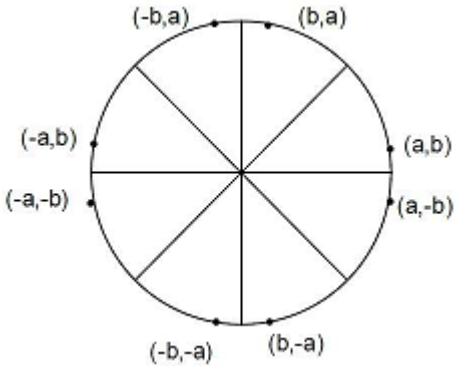

Figure 5: For a pixel(a,b), showing all possible pixels in 8 octants.

In midpoint circle algorithm, we use the above symmetry 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. For example, let us consider any given pixel (x, y), the next pixel to be plotted is either (x, y+1) or (x-1, y+1). This can be decided by following the steps below.
1. Find the mid-point p of the two possible pixels i.e (x-0.5, y+1).
2. If p lies inside or on the circle perimeter, we plot the pixel (x, y+1), otherwise if it's outside we plot the pixel (x-1, y+1).

The steps of this algorithm are as below-
Step 1 Input radius r and circle center (xc,yc) and obtain the first point on the circumference of the circle centered on the origin as - (x_0,y_0)= (0, r).
Step 2 Calculate the initial value of decision parameter as P_0=(5/4) - r.
Step 3 At each x_k position starting at k=0, perform the following test
If $P_k < 0$ then next point on circle (0,0) is $(x_(k+1), y_k)$ and

$$P_{k+1} = P_k + 2x_{k+1} + 1$$

Else

$$P_{k+1} = P_k + 2x_{k+1} + 1 - 2y_{K+1}$$

Where, $2x_{k+1} = 2x_{k+2}$
$2y_{k+1} = 2y_{k-2}$
Step 4 Determine the symmetry points in other seven octants.
Step 5 Move each calculate pixel position (X, Y) onto the circular path centered on (x_c, y_c) and plot the coordinate values.

$$x = x + x_c$$
$$y = y + y_c$$

Step 6 Repeat Step 3 through 5 until $x >= y$

3.3 Source Code

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

int center_x, center_y, r;

void plot(int x, int y)
{
glBegin(GL_POINTS);
glVertex2i(x+center_x, y+center_y);
glEnd();
}


void myInit (void)
{
glClearColor(1.0, 1.0, 1.0,1.0);
glColor3f(1.0f, 1.0f, 1.0f);
glPointSize(4.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-640, 640.0, -480.0, 480.0);
}


void midPointCircleAlgo()
{
int x = 0;
int y = r;
float decision = 5/4 - r;
plot(x, y);

while (y > x)
{
if (decision < 0)
{
x++;
decision += 2*x+3;
}
else
{
y--;
x++;
decision += 2*(x-y)+5;
}
plot(x, y);
plot(x, -y);
plot(-x, y);
plot(-x, -y);
plot(y, x);
```

```
plot(-y, x);
plot(y, -x);
plot(-y, -x);
}

}

void process(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (0.0, 0.0, 0.0);
glPointSize(1.0);

midPointCircleAlgo();

glFlush ();
}

main(int argc, char** argv)
{
cout << "Enter the coordinates of the center:\n\n" << endl;

cout << "(X,Y)   : ";
cin >> center_x>>center_y;
cout << "\nEnter radius : "; cin >> r;


glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (640, 480);
glutInitWindowPosition (100, 150);
glutCreateWindow ("Midpoint");
glutDisplayFunc(process);
myInit ();
glutMainLoop();

}
```

3.4 Sample I/O

```
Enter the coordinates of the center:


(X,Y)   : 0
0

Enter radius : 50

Process returned 0 (0x0)   execution time : 38.976 s
Press any key to continue.
```

Figure 6: User Input

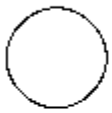

Figure 7: Midpoint Circle Drawing Output.

3.5 Conclusion

1. We find a circle that create according to the codes are generated in the opengl.
2. We observed the difference between midpoint and bresenham circle.
3. We know about opengl and its implantation.
4. Midpoint algorithm is much simple and requires only integer data.

4 Bresenham Circle Algorithm

4.1 Objective

To implement a circle using Bresenham's circle algorithm.

4.2 Introduction

Circles have the property of being highly summetrical. So for whole 360 degree of circle, it can be divided in 8 parts each octant of 45 degree. Bresenhams circle algorithm is based on this idea.Bresenham's circle algorithm is derived from the midpoint circle algorithm. Here, equation of the circle is,

$$x^2 + y^2 = r^2 \tag{1}$$

Bresenhams algorithm calculates the location of the pixels in the first octant of 45 degree. So for every pixel it calculates, we draw a pixel in each of the eight octants of the circle. The steps of this algorithm are as below-

```
Step 1 Get the
coordinates of the center of the circle and radius, and store them in x, y and R respectively. Set P=0
Q=R.
Step 2 Set decision parameter D = 3 2R.
Step 3 Repeat through step-8 while P Q..
Step 4 Call Draw Circle (X, Y, P, Q).
Step 5 If D < 0 then D = D + 4P + 6..
Step 6 Else Set R = R - 1, D = D + 4(P-Q) + 10..
Step 6 Call Draw Circle (X, Y, P, Q)...
```

4.3 Source Code

```
#include<windows.h>
#include <stdio.h>
#include <iostream>
#include <GL/glut.h>
using namespace std;
int center_x,center_y,radius;

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

void bresenham_circle(int r)
{
  int x=0,y=r;
  float pk=3-(2*r);
```

```
  plot_point(x,y);
  int k;
  while(x < y)
  {
    x = x + 1;
    if(pk < 0)
      pk = pk + 4*x+6;
    else
    {
      y = y - 1;
      pk = pk + 4*(x - y) + 10;
    }
    plot_point(x,y);
  }
  glFlush();
}

void process(void)
{

  glClear(GL_COLOR_BUFFER_BIT);

  bresenham_circle(radius);
}

void Init()
{
  glClearColor(1.0,1.0,1.0,0);

  glColor3f(0.0,0.0,0.0);

  gluOrtho2D(-640, 640 ,-480 , 480);
}

main(int argc, char **argv)
{
    cout<<"center x and y "<<endl;
    cin>>center_x>>center_y;
    cout<<"Enter Radius-";
    cin>>radius;
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowPosition(0,0);
  glutInitWindowSize(640,480);
  glutCreateWindow("bresenham_circle");
  Init();
  glutDisplayFunc(process);
  glutMainLoop();
}
```

4.4 Sample I/O

```
center x and y
0
0
Enter Radius- 100

Process returned 0 (0x0)   execution time : 43.680 s
Press any key to continue.
```

Figure 8: User Input.

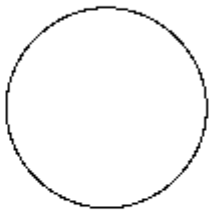

Figure 9: Bresenhame Circle Drawing Output.

4.5 Conclusion

1. Familiarization with OpenGL has been done effectively.
2. A clear concept on Bresenham's circle drawing algorithm has been acquired properly.
3. This algorithm been implemented using several functions of OpenGL.
4. Bresenham's algorithm is more efficient and accurate than DDA line drawing algorithm because it uses integer arithmetic along with addition, multiplication and subtraction only.

5 Midpoint Ellipse Algorithm

5.1 Objective

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 90^0.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 :

$$(x-h)^2/b^2 + (y-k)^2/a^2 = 1 \tag{2}$$

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

$$x = bcos\theta + h \tag{3}$$

$$y = asin\theta + h \tag{4}$$

Ellipse is defined as the locus of a point in a plane which moves in a plane in such a manner that the ratio of its distance from a fixed point called focus in the same plane to its distance from a fixed straight line called directrix is always constant, which should always be less than unity.

The midpoint ellipse drawing algorithm uses the four way symmetry of the ellipse to generate it. We divide each quadrant into two regions and the boundary of two regions is the point at which the curve has a slope of -1. We proceed by taking unit steps in the x direction to the point P(where curve has a slope of -1), then taking unit steps in the y direction and applying midpoint algorithm at every step.

Let us take the start position at (0,ry) and step along the ellipse path in clockwise order throughout the first quadrant. Ellipse function can be defined as: $fellipse(x,y) = ry^2x^2 + rx^2y^2 - rx^2ry^2$

According to this there are some properties which have been generated that are:
1. $fellipse(x,y) < 0$ which means (x,y) is inside the ellipse boundary.
2. $fellipse(x,y) > 0$ which means (x,y) is outside the ellipse boundary.
3. $fellipse(x,y) = 0$ which means (x,y) is on the ellipse boundary.

The steps of this algorithm are as below-
Step 1 Take as input the ellipse centre and radius and obtain the first point on an ellipse centered on the origin as a $(x, y_0) = (0, r_y)$
Step 2 Now calculate the initial decision parameter in region 1 as:

$$p_10 = r_y^2 + (1/4)r_x^2 - r_x^2r_y$$

Step 3 At each xk position in region 1 perform the following task. If $p_{1k} < 0$ then the next point along the ellipse centered on (0,0) is (x_{k+1}, y_k)

$$p1_{k+1} = p1_k + 2r_y^2x_{k+1} + r_y^2$$

Otherwise the next point along the circle is (x_{k+1}, y_{k-1})

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

Step 4 Now, again calculate the initial value in region 2 using the last point (x_0, y_0) calculated in a region 1 as:

$$p2_0 = r_y^2(x_0 + 1/2)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2$$

Step 5 At each yk position in region 2 starting at k = 0 perform the following task. If $p2_k < 0$ the next point along the ellipse centered on (0,0) is (x_k, y_{k-1})

$$p2_{k+1} = p2_{k-2} r_x^2 y_{k+1} + rx^2$$

Otherwise the next point along the circle will be (x_{k+1}, y_{k-1})

$$p2_{k+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$

Step 6 Now determine the symmetric points in another three quadrants.
Step 7 Plot the coordinate value as: $x = x + x_c$, $y = y + y_c$
Step 8 Repeat the steps for region 1 until $2r_y^2 x >= 2rx^2 y$

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 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 Sample I/O

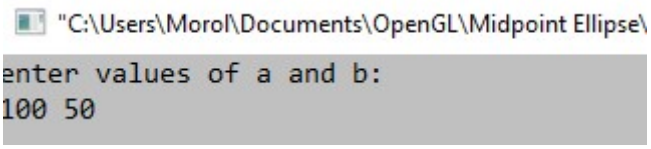

Figure 10: User Input.

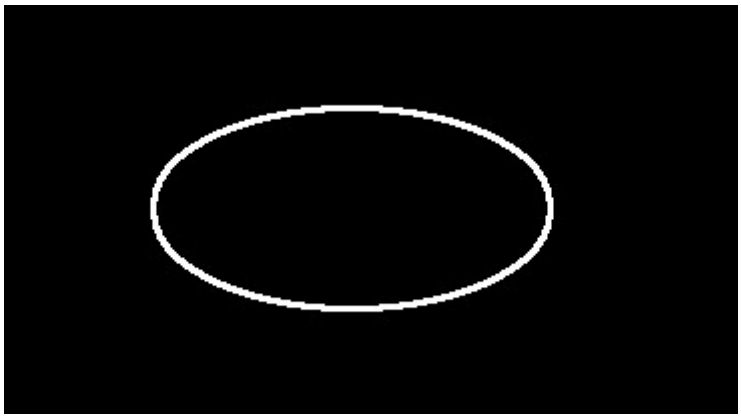

Figure 11: Output of Midpoint Ellipse Drawing Algorithm

5.5 Conclusion

1. A clear concept on midpoint ellipse drawing algorithm has been acquired properly.
2. Midpoint ellipse algorithm is the modified version of Bresenham's algorithm, that is, only addition operations are required in this program loop.
3. We find a ellipse that create according to the codes are generated in the opengl.
4. We know about opengl and its implantation.

6 Boundary Fill Algorithm

6.1 Objective

We have to fill an object using Boundary fill algorithm.

6.2 Introduction

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.

Boundary Fill Algorithm is recursive in nature. It takes an interior point(x, y), a fill color, and a boundary color as the input. The algorithm starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbours of (x, y). If a point is found to be of fill color or of boundary color, the function does not call its neighbours and returns. This process continues until all points up to the boundary color for the region have been tested.

The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.
4-connected pixels : After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above and below the current pixel.Areas filled by this method are called 4-connected.
8-connected pixels : More complex figures are filled using this approach. The pixels to be tested are the 8 neighboring pixels, the pixel on the right, left, above, below and the 4 diagonal pixels. Areas filled by this method are called 8-connected.

6.3 Source Code

```
#include<windows.h>
#include <iostream>
#include <math.h>
#include <time.h>
#include <GL/glut.h>
using namespace std;
void init(){
    glClearColor(1.0,1.0,1.0,0.0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0,640,0,480);
}

void bound_it(int x, int y, float* fillColor, float* bc){
    float color[3];
    glReadPixels(x,y,1.0,1.0,GL_RGB,GL_FLOAT,color);
    if((color[0]!=bc[0] || color[1]!=bc[1] || color[2]!=bc[2])&&(
     color[0]!=fillColor[0] || color[1]!=fillColor[1] || color[2]!=fillColor[2])){
        glColor3f(fillColor[0],fillColor[1],fillColor[2]);
        glBegin(GL_POINTS);
            glVertex2i(x,y);
        glEnd();
        glFlush();
          bound_it(x+1,y,fillColor,bc);
           bound_it(x-1,y,fillColor,bc);
            bound_it(x,y+1,fillColor,bc);
```

```
             bound_it(x,y-1,fillColor,bc);
            bound_it(x+1,y+1,fillColor,bc);
         bound_it(x+1,y-1,fillColor,bc);
         bound_it(x-1,y+1,fillColor,bc);
        bound_it(x-1,y-1,fillColor,bc);
    }
}

void mouse(int btn, int state, int x, int y){
    y = 480-y;

            float bCol[] = {1,0,0};
            float color[] = {0,0,1};
            bound_it(x,y,color,bCol);
}

void world(){
    glLineWidth(3);
    glPointSize(1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1,0,0);
    glBegin(GL_LINE_LOOP);
        glVertex2i(150/2,100/2);
        glVertex2i(450/2,100/2);
        glVertex2i(450/2,300/2);
        glVertex2i(150/2,300/2);
    glEnd();
    glFlush();

}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(200,200);
    glutCreateWindow("bfill");
    glutDisplayFunc(world);
    glutMouseFunc(mouse);
    init();
    glutMainLoop();
    return 0;
}
```

6.4 Sample I/O

Figure 12: Boundary Fill Output (After Filling)

6.5 Conclusion

1. 4 connected approach was used to fill the rectangle.
2. A rectangle was used to fill.
3. The boundary of the rectangle is green colored and the filling color is red.
4. For implementating this algorithm, we used opengl.
5. This algorithm is recursive in nature and time consuming.

7 Flood fill algorithm

7.1 Objective

We have to fill an 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.

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.

7.3 Source Code

```
#include<windows.h>
#include <math.h>
#include <gl/glut.h>
#include<bits/stdc++.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 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)
{
      y = 480-y;
Color newColor = {0, 0, 1};
Color oldColor = {1, 1, 1};
floodFill(x, y, oldColor, newColor);
}

void world(){
    glLineWidth(3);
    glPointSize(1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1,0,0);
    glBegin(GL_LINE_LOOP);
        glVertex2i(15,100);
        glVertex2i(45,100);
        glVertex2i(45,300);
        glVertex2i(15,300);
    glEnd();
    glFlush();
}
void display(void) {
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
 world();
glEnd();
glFlush();
}
int main(int argc, char** argv)
{
```

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(640, 480);
glutInitWindowPosition(200, 200);
glutCreateWindow("Open GL");
init();
glutDisplayFunc(display);
glutMouseFunc(onMouseClick);
glutMainLoop();
return 0;
}
```

7.4 Sample I/O

Figure 13: Flood Fill Output (After Filling)

7.5 Conclusion

1. 4 connected approach was used to fill the polygon.
2. A rectangle was used to fill.
3. For implementating this algorithm, we used opengl.
4. This algorithm is recursive in nature and time consuming.

8 C Curve Algorithm

8.1 Objective

To learn about C Curve Algorithm and implement it by finding C Curve in OpenGL platform.

8.2 Introduction

In mathematics, the Lévy C curve is a self-similar fractal that was first described and whose differentiability properties were analysed by Ernesto Cesàro in 1906 and Georg Faber in 1910, but now bears the name of French mathematician Paul Lévy, 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 +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 Sample I/O

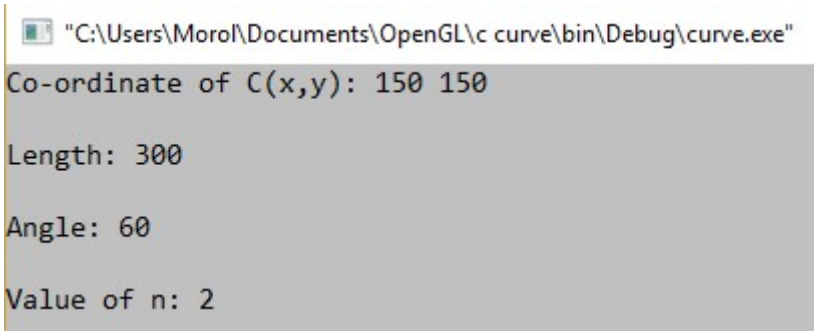

Figure 14: User Input

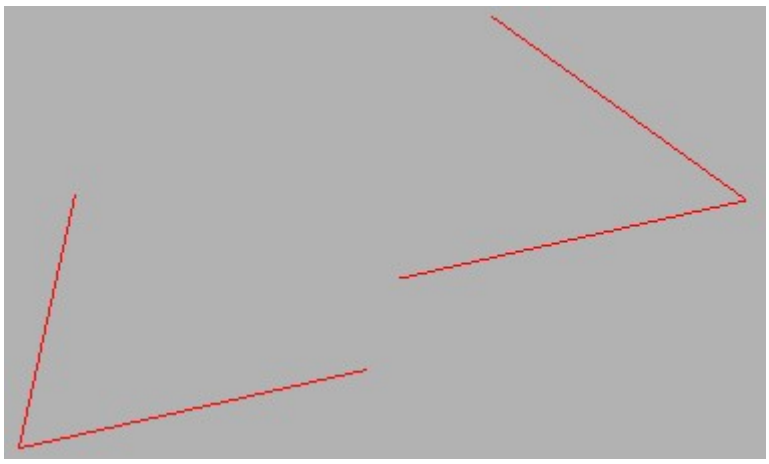

Figure 15: C Curve Output

8.5 Conclusion

1. A clear concept about C Curve algorithm has been acquired properly.
2. This algorithm been implemented using several functions of OpenGL.
3. Problem was faced while trying to implement this algorithm for large number of iteration.

9 Koch Curve Algorithm

9.1 Objective

To implement the Koch curve algorithm using OpenGL.

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 nth 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);
    gllColor3f (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 Sample I/O

```
Enter Co-ordinate of (x,y): 150 150

Enter Length: 300

Enter Degree of Angle: 60

Enter No. of Iteration: 2
```

Figure 16: User Input.

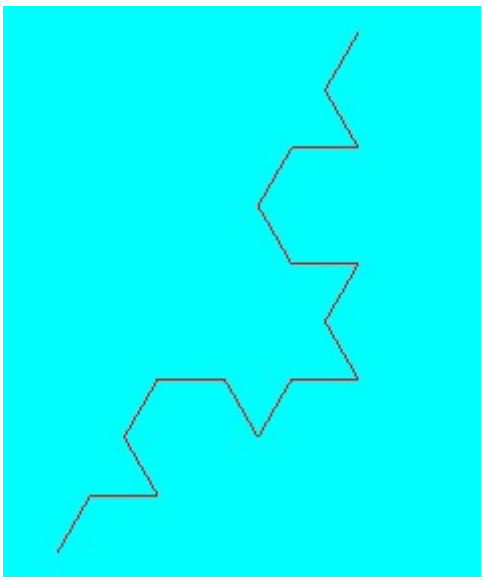

Figure 17: koch Curve Output.

9.5 Conclusion

1. The Koch curve is the limiting curve obtained by applying this construction an infinite number of times.
2. 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 trivally accepted or rejected. If the line cannot be trivally 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 > Ymax]
Bit 2 : outside halfplane of bottom edge, below bottom edge [Y < Ymin]
Bit 3 : outside halfplane of right edge, to the right of right edge [X > Xmax]
Bit 4 : outside halfplane of left edge, to the left of left edge [X < Xmin]

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.

10.3 Source Code

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

void display();
using namespace std;
float xmin=-100;
float ymin=-100;
float xmax=100;
float ymax=100;
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;
    if(y<ymin)c=4;
    if(x>xmax)c=c|2;
    if(x<xmin)c=c|1;
    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')
    {  cout<<"Hello";
        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>>yd1>>xd2>>yd2;
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(600,600);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Clipping");
    glutDisplayFunc(display);
   glutKeyboardFunc(mykey);
    init();
    glutMainLoop();
    return 0;
}
```

10.4 Sample I/O

```
Enter line co-ordinates:0
0
150
150
Hello
```

Figure 18: User Input

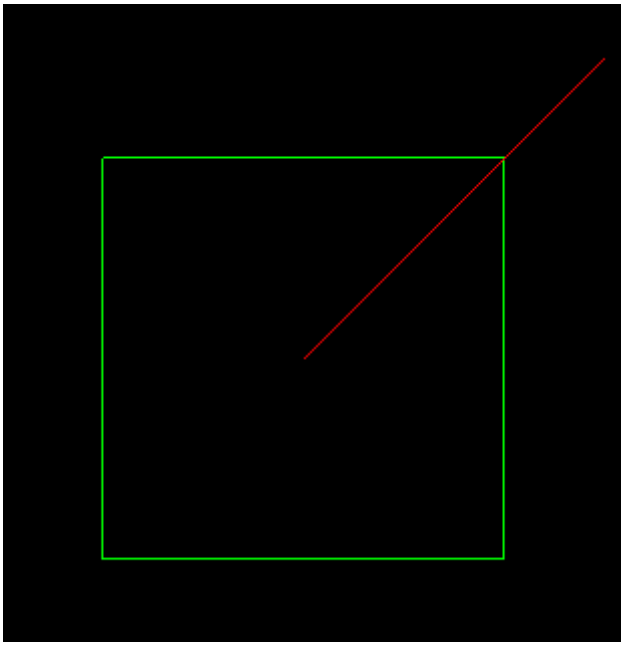

Figure 19: Line Clipping Output (without clipping)

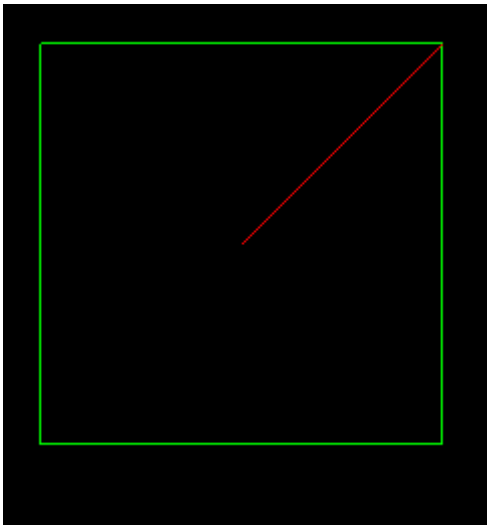

Figure 20: Line Clipping Output (after Clipping)

10.5 Conclusion

1. We get to know how to clip the portion of a line that is outside a rectangle.
2. We know about opengl and its implantation.
3. This problem was implemented in OpenGL.
4. We have implemented this algorithm by dividing the 2D space into 9 regions and used 4 bits to define these regions.

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 Sample I/O

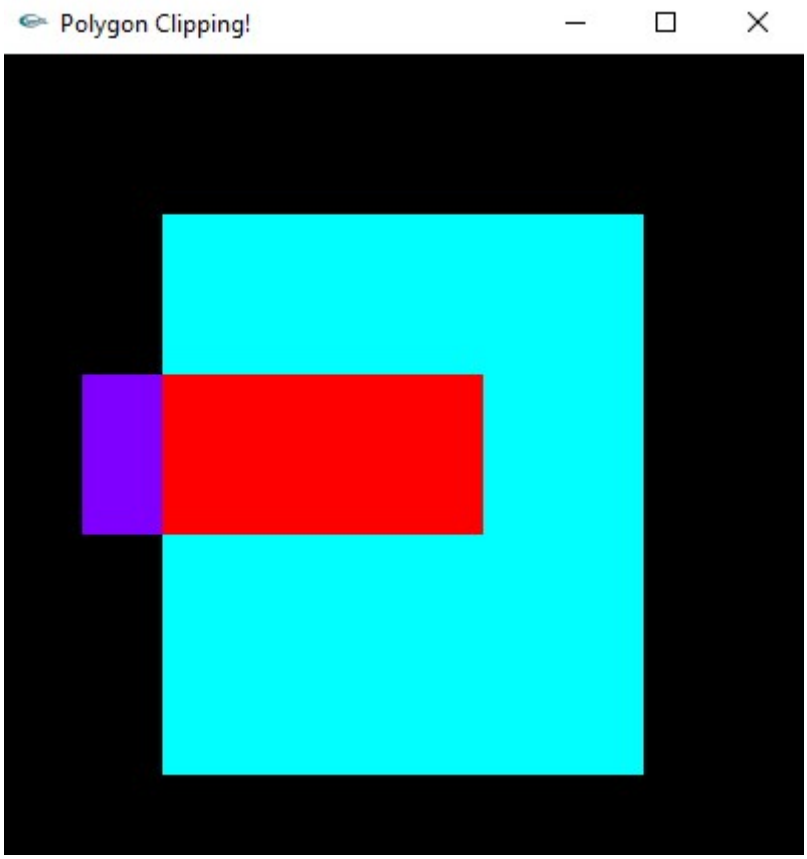

Figure 21: User Input

11.5 Conclusion

1. 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.
2. This algorithm been implemented using several functions of OpenGL.
3. A clear concept about Sutherland Hodgeman Polygon Clipping algorithm has been acquired properly.