JADAVPUR UNIVERSITY

COMPUTER GRAPHICS LAB REPORT

NAME : ADNAN KHURSHID
ROLL NO : 002010501025
CLASS : BCSE III

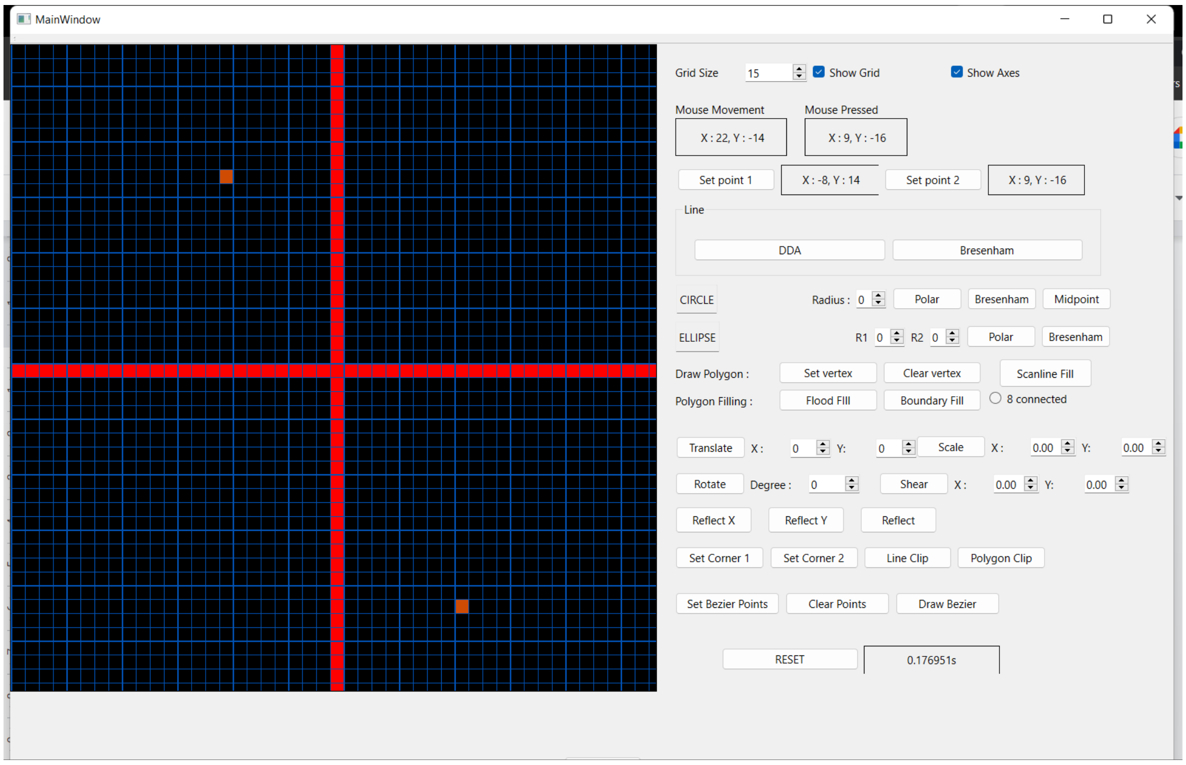
Assignment 2 : Implement a line drawing algorithm to draw lines between two end points in the raster grid using, a) DDA b) Bresenham's line drawing algorithm. Show execution times for each algorithm in ms

a) DDA Algorithm

CODE :

```
void MainWindow::on_dda_button_clicked(int r=32,int g=252,int b=3 )
{

high_resolution_clock::time_point t1 = high_resolution_clock::now();

   int gridsize=ui->grid_spin_box->value();
   double x1 = p1.x()/gridsize;
   double y1 = p1.y()/gridsize;
   double x2 = p2.x()/gridsize;
   double y2 = p2.y()/gridsize;
   double xinc, yinc, step;

   double slope = fabs(y2-y1)/fabs(x2-x1);
   if(slope  <= 1.00) {
       xinc = 1;
       yinc = slope;
       step = fabs(x2 - x1);
   } else {
       xinc = 1/slope;
       yinc = 1;
       step = fabs(y2 - y1);
   }
   if(x1 > x2) xinc *= -1;
   if(y1 > y2) yinc *= -1;
   double x = x1*gridsize + gridsize/2;
   double y = y1*gridsize + gridsize/2;
   for(int i=0;i<=step;i++) {
       point(x,y,r,g,b);
       x += xinc * gridsize;
       y += yinc * gridsize;
       delay(50);
   }

   high_resolution_clock::time_point t2 = high_resolution_clock::now();

    duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

     ui->dda_time->setText (QLocale ().toString (time_span.count()) + "s");}
```

OUTPUT:



MainWindow

Grid Size 15 [✓ Show Grid] [✓ Show Axes]

Mouse Movement: X : 22, Y : -14
Mouse Pressed: X : 9, Y : -16

Set point 1: X : -8, Y : 14
Set point 2: X : 9, Y : -16

Line: DDA, Bresenham

CIRCLE Radius : 0 Polar Bresenham Midpoint

ELLIPSE R1 0 R2 0 Polar Bresenham

Draw Polygon : Set vertex, Clear vertex, Scanline Fill

Polygon Filling : Flood Fill, Boundary Fill, 8 connected

Translate X : 0 Y: 0 Scale X : 0.00 Y: 0.00

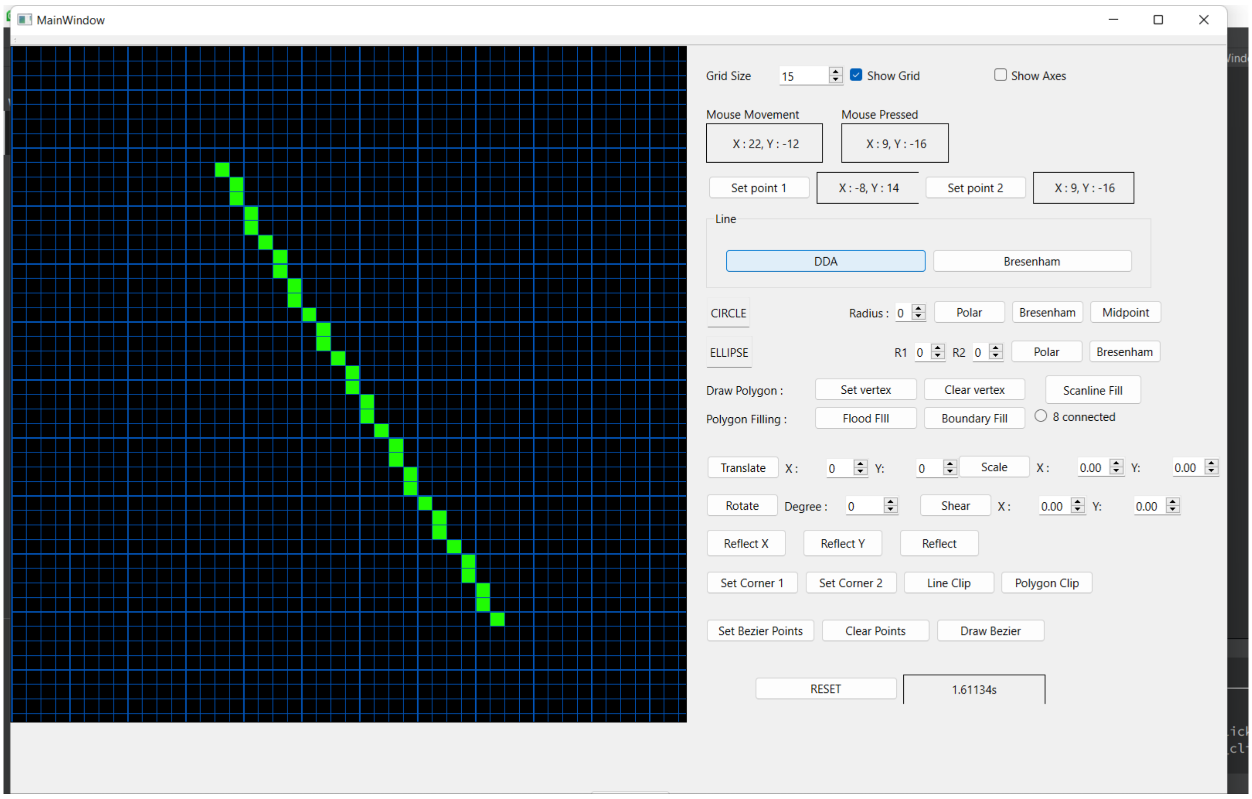
Rotate Degree : 0 Shear X : 0.00 Y: 0.00

Reflect X, Reflect Y, Reflect

Set Corner 1, Set Corner 2, Line Clip, Polygon Clip

Set Bezier Points, Clear Points, Draw Bezier

RESET 0.176951s



MainWindow

Grid Size 15 [✓ Show Grid] Show Axes

Mouse Movement: X : 22, Y : -12
Mouse Pressed: X : 9, Y : -16

Set point 1: X : -8, Y : 14
Set point 2: X : 9, Y : -16

Line: DDA, Bresenham

CIRCLE Radius : 0 Polar Bresenham Midpoint

ELLIPSE R1 0 R2 0 Polar Bresenham

Draw Polygon : Set vertex, Clear vertex, Scanline Fill

Polygon Filling : Flood Fill, Boundary Fill, 8 connected

Translate X : 0 Y: 0 Scale X : 0.00 Y: 0.00

Rotate Degree : 0 Shear X : 0.00 Y: 0.00

Reflect X, Reflect Y, Reflect

Set Corner 1, Set Corner 2, Line Clip, Polygon Clip

Set Bezier Points, Clear Points, Draw Bezier

RESET 1.61134s

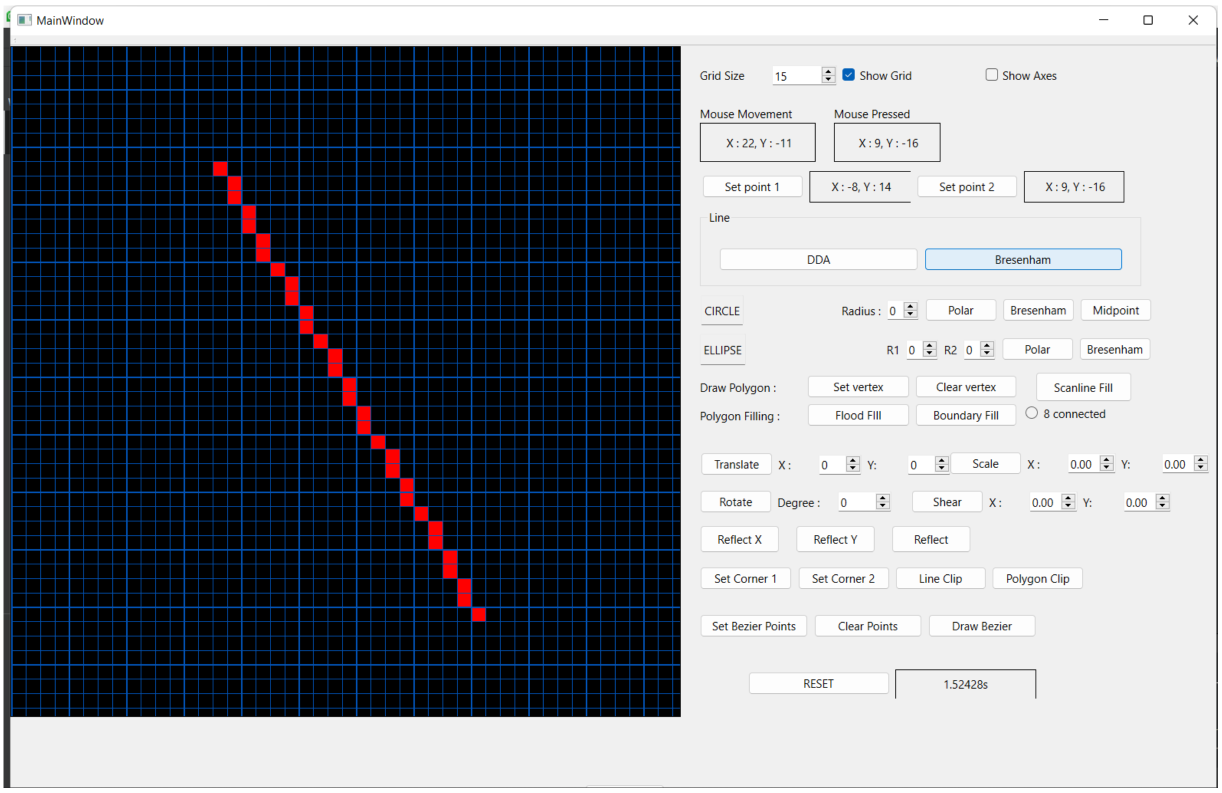
Time taken : 1.61134 s

b) Bresenham Line Drawing Algorithm

CODE :

```
void MainWindow::on_bresline_button_clicked()
{
   high_resolution_clock::time_point t1 = high_resolution_clock::now();
    int gridsize = ui->grid_spin_box->value();
    int x1 = p1.x() / gridsize;
    int y1 = p1.y() / gridsize;
    int x2 = p2.x() / gridsize;
    int y2 = p2.y() / gridsize;
    int dx = fabs(x2 - x1);
    int dy = fabs(y2 - y1);

    int xinc = (x1 > x2 ? -1 : 1);
    int yinc = (y1 > y2 ? -1 : 1);
    bool flag = 1;
    int x = x1 * gridsize + gridsize / 2;
    int y = y1 * gridsize + gridsize / 2;
    if (dy > dx)
    { // if slope > 1, then swap
        swap(dx, dy);
        swap(x, y);
        swap(xinc, yinc);
        flag = 0;
    }
    int decision = 2 * dy - dx;
    int step = dx;
    for (int i = 0; i <= step; i++)
    {
        if (flag)
            point(x, y);
        else
            point(x, y);
        if (decision < 0)
        {
            x += xinc * gridsize;
            decision += 2 * dy;
        }
        else
        {
            x += xinc * gridsize;
            y += yinc * gridsize;
            decision += 2 * dy - 2 * dx;
        }
        delay(10);
    }
  high_resolution_clock::time_point t2 = high_resolution_clock::now();

 duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

   ui->dda_time->setText(QLocale().toString(time_span.count()) + "s");
}
```

OUTPUT :



Grid Size 15 ☑ Show Grid ☐ Show Axes

Mouse Movement X : 22, Y : -11

Mouse Pressed X : 9, Y : -16

Set point 1 X : -8, Y : 14

Set point 2 X : 9, Y : -16

Radius : 0

R1 0 R2 0

Translate X : 0 Y: 0

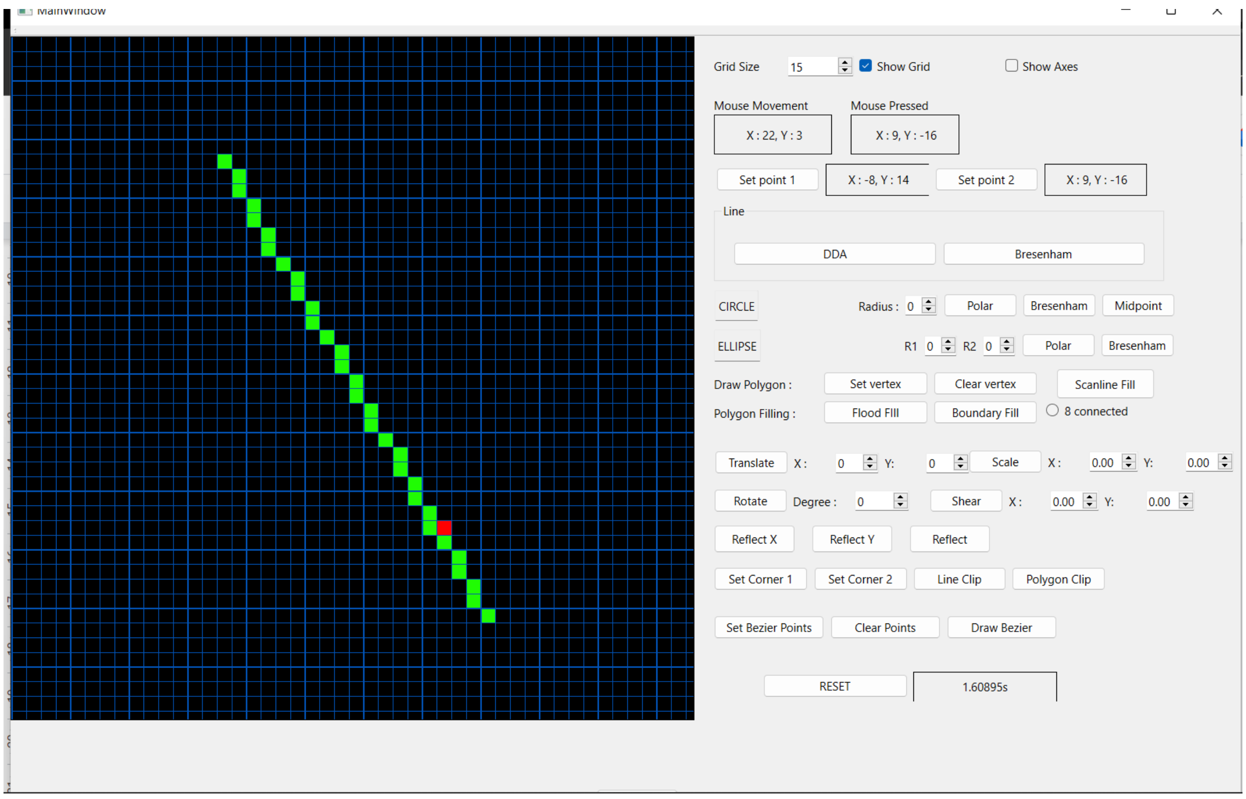
Scale X : 0.00 Y: 0.00

Rotate Degree : 0

Shear X : 0.00 Y: 0.00

1.52428s

Time taken : 1.52428 s



Grid Size 15 ☑ Show Grid ☐ Show Axes

Mouse Movement X : 22, Y : 3

Mouse Pressed X : 9, Y : -16

Set point 1 X : -8, Y : 14

Set point 2 X : 9, Y : -16

Radius : 0

R1 0 R2 0

Translate X : 0 Y: 0

Scale X : 0.00 Y: 0.00

Rotate Degree : 0

Shear X : 0.00 Y: 0.00

1.60895s

DDA (in green) and Bresenham algorithm (in red)

Assignment 6 : Implement the seed-fill algorithms: a) Boundary fill, b) Flood fill.

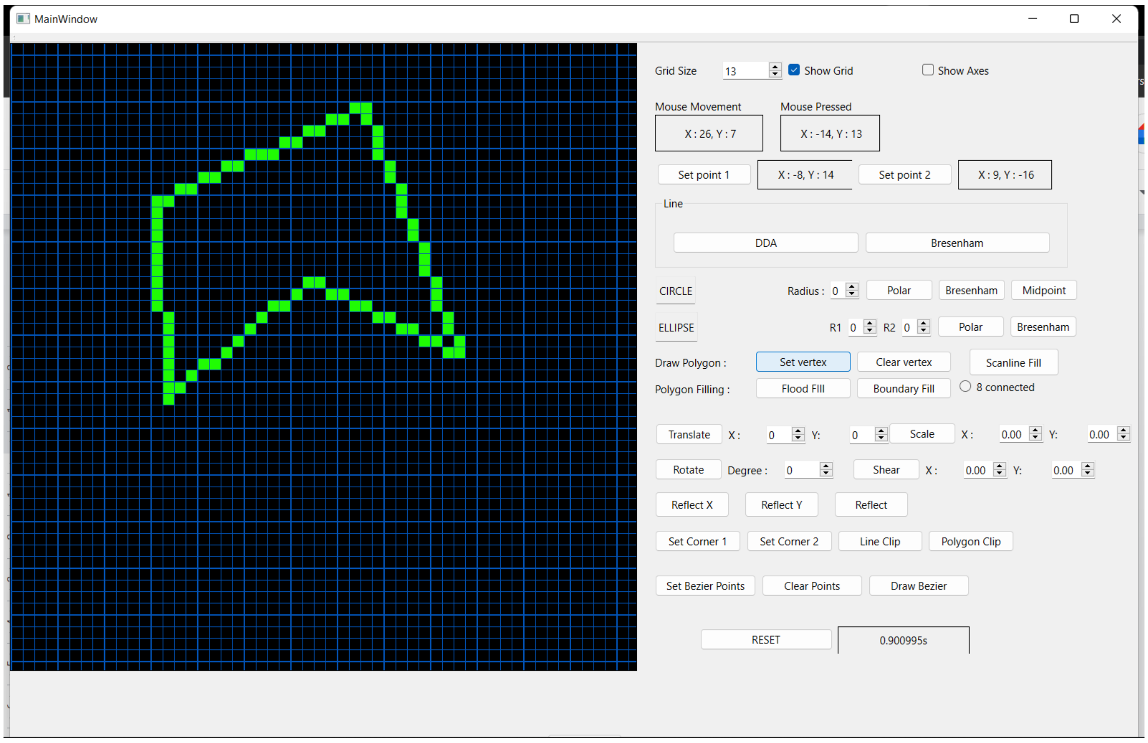
a) Boundary Fill

CODE :

```
// Boundary fill algorithm
void MainWindow::boundaryfillutil(int x, int y, QRgb edgecolour, int r, int g, int b)
{
    int gridsize = ui->grid_spin_box->value();

    if (x < 0 || y < 0 || x >= img.width() || y >= img.height())
        return;

    if (img.pixel(x, y) == edgecolour || img.pixel(x, y) == qRgb(r, g, b))
        return;

    point(x, y, r, g, b);
    delay(5);
    boundaryfillutil(x, y + gridsize, qRgb(32, 252, 3), r, g, b);
    boundaryfillutil(x + gridsize, y, qRgb(32, 252, 3), r, g, b);
    boundaryfillutil(x, y - gridsize, qRgb(32, 252, 3), r, g, b);
    boundaryfillutil(x - gridsize, y, qRgb(32, 252, 3), r, g, b);

    if (ui->connected8_radio->isChecked())
    {
        boundaryfillutil(x + gridsize, y + gridsize, qRgb(32, 252, 3), r, g, b);
        boundaryfillutil(x - gridsize, y - gridsize, qRgb(32, 252, 3), r, g, b);
        boundaryfillutil(x + gridsize, y - gridsize, qRgb(32, 252, 3), r, g, b);
        boundaryfillutil(x - gridsize, y + gridsize, qRgb(32, 252, 3), r, g, b);
    }
}

void MainWindow::on_boundary_fill_clicked()
{
    int gridsize = ui->grid_spin_box->value();
    p1.setX(ui->frame->x);
    p1.setY(ui->frame->y);
    int x = p1.x() / gridsize;
    int y = p1.y() / gridsize;
    x = x * gridsize + gridsize / 2;
    y = y * gridsize + gridsize / 2;
    point(x, y, 0, 0, 0);
    if (ui->connected8_radio->isChecked())
    {
        boundaryfillutil(x, y, qRgb(32, 252, 3), 255, 255, 255);
    }
    else
    {
        boundaryfillutil(x, y, qRgb(32, 252, 3), 255, 255, 0);
    }
}
```

OUTPUT :

4-Connected Boundary Fill :



Grid Size 13 [✓ Show Grid] Show Axes

Mouse Movement X : 26, Y : 7

Mouse Pressed X : -14, Y : 13

Set point 1 X : -8, Y : 14 Set point 2 X : 9, Y : -16

Radius : 0

R1 0 R2 0

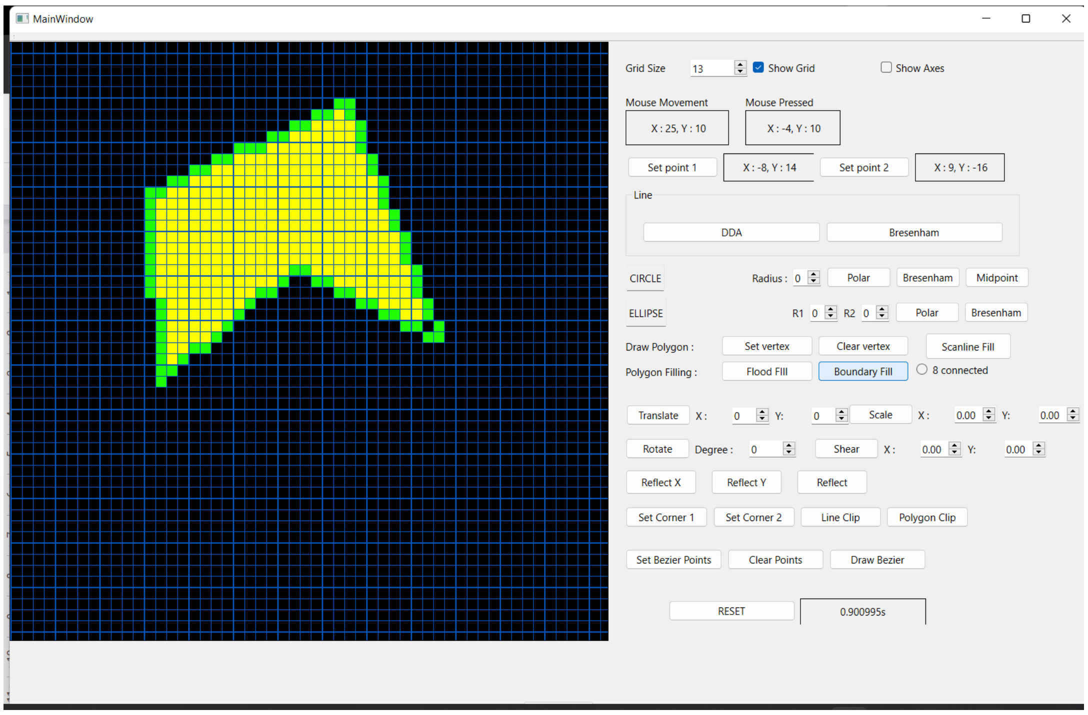
Draw Polygon :

Polygon Filling : ○ 8 connected

X : 0 Y: 0 X : 0.00 Y: 0.00

Degree : 0 X : 0.00 Y: 0.00

0.900995s



Grid Size 13 [✓ Show Grid] Show Axes

Mouse Movement X : 25, Y : 10

Mouse Pressed X : -4, Y : 10

Set point 1 X : -8, Y : 14 Set point 2 X : 9, Y : -16

Radius : 0

R1 0 R2 0

Draw Polygon :

Polygon Filling : ○ 8 connected

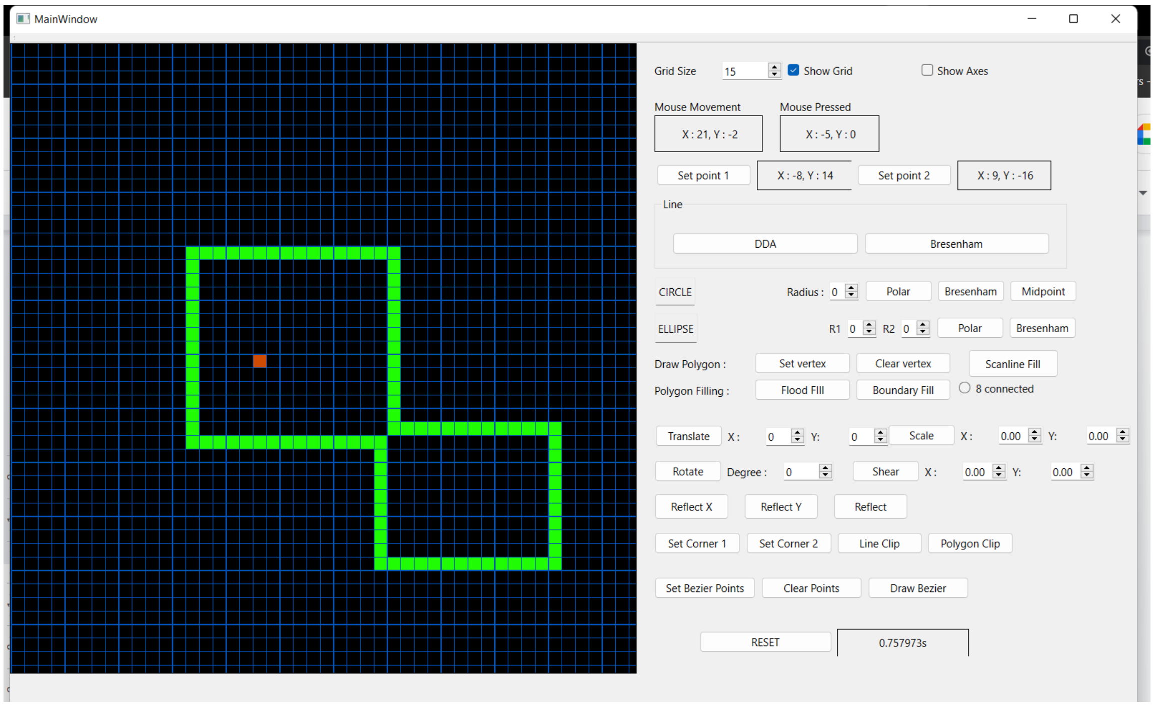
X : 0 Y: 0 X : 0.00 Y: 0.00

Degree : 0 X : 0.00 Y: 0.00

0.900995s

As it can be seen, one pixel is left unfilled because the 4-connected algorithm could not reach that pixel.

8-Connected Boundary Fill :



Grid Size 15 [✓ Show Grid] [ ] Show Axes

Mouse Movement X : 21, Y : -2
Mouse Pressed X : -5, Y : 0

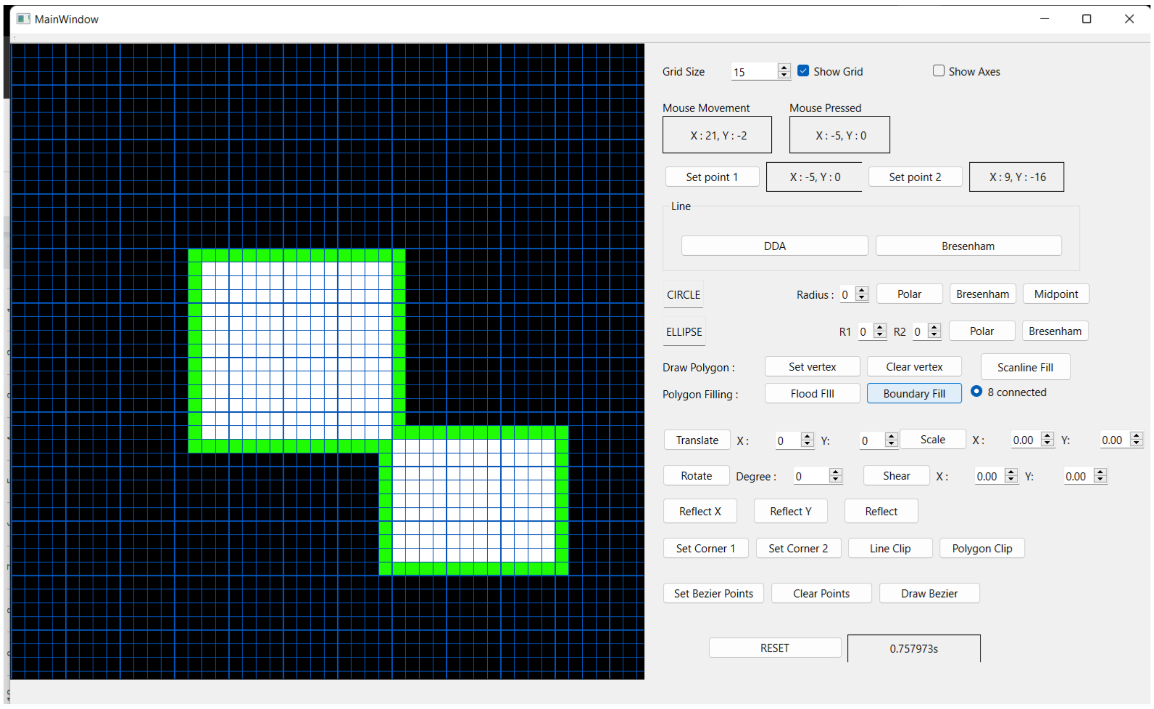
Set point 1 X : -8, Y : 14
Set point 2 X : 9, Y : -16

Radius : 0
R1 0 R2 0

Draw Polygon :
Polygon Filling :
[ ] 8 connected

Translate X : 0 Y: 0
Scale X : 0.00 Y: 0.00
Rotate Degree : 0
Shear X : 0.00 Y: 0.00

0.757973s



Grid Size 15 [✓ Show Grid] [ ] Show Axes

Mouse Movement X : 21, Y : -2
Mouse Pressed X : -5, Y : 0

Set point 1 X : -5, Y : 0
Set point 2 X : 9, Y : -16

Radius : 0
R1 0 R2 0

Draw Polygon :
Polygon Filling :
[✓ 8 connected]

Translate X : 0 Y: 0
Scale X : 0.00 Y: 0.00
Rotate Degree : 0
Shear X : 0.00 Y: 0.00

0.757973s

The 8-connected Boundary fill algorithm can rectify the problem previously faced in 4-connected Algorithm

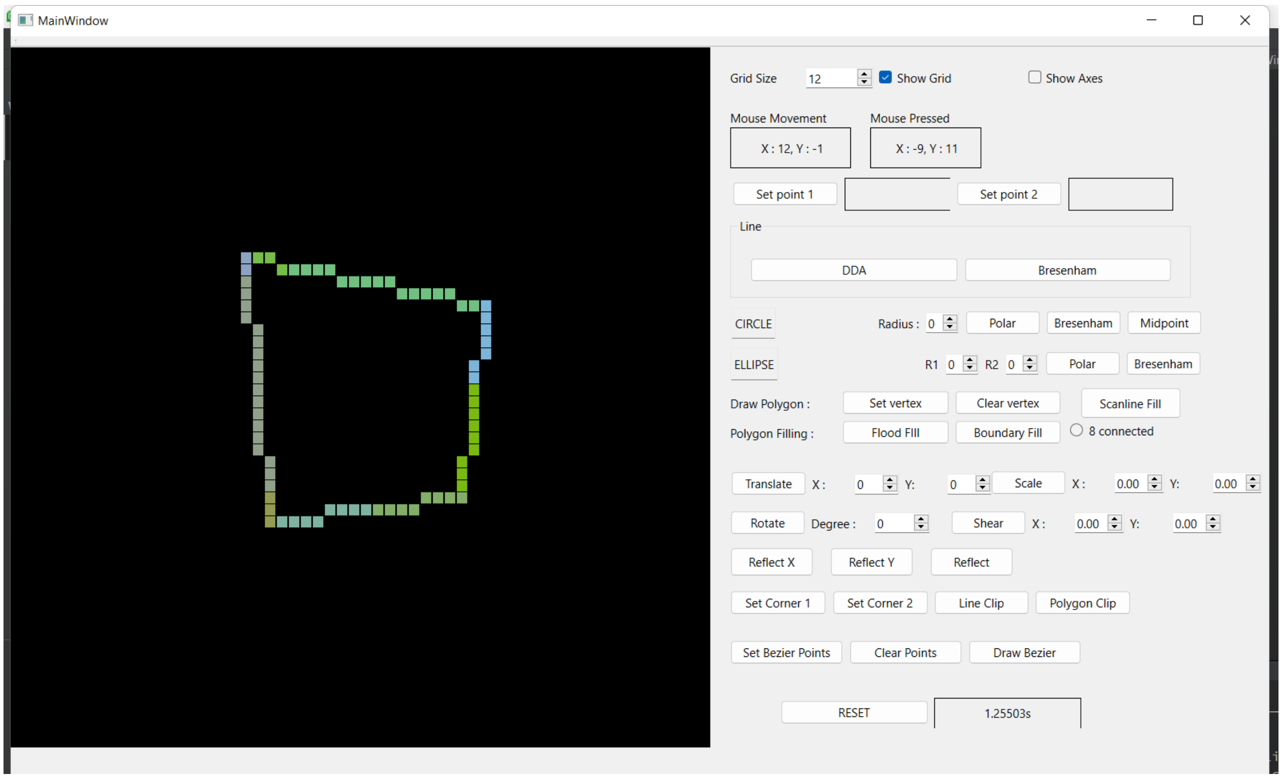
b) Flood Fill :

CODE :

```
// Flood fill algorithm
void MainWindow::floodfillutil(int x, int y, int r, int g, int b)
{
    if (x < 0 || y < 0 || x >= img.width() || y >= img.height())
        return;
    if (img.pixel(x, y) == qRgb(r, g, b) || img.pixel(x, y) == qRgb(0, 0, 0))
    {
        int gridsize = ui->grid_spin_box->value();

        if (ui->connected8_radio->isChecked())
        {
            point(x, y, 255, 255, 0);
        }
        else
        {
            point(x, y, 255, 255, 255);
        }
        delay(5);
        floodfillutil(x - gridsize, y, r, g, b);
        floodfillutil(x + gridsize, y, r, g, b);
        floodfillutil(x, y - gridsize, r, g, b);
        floodfillutil(x, y + gridsize, r, g, b);

        if (ui->connected8_radio->isChecked())
        {
            floodfillutil(x + gridsize, y + gridsize, r, g, b);
            floodfillutil(x - gridsize, y - gridsize, r, g, b);
            floodfillutil(x + gridsize, y - gridsize, r, g, b);
            floodfillutil(x - gridsize, y + gridsize, r, g, b);
        }
    }
    else
        return;
}

void MainWindow::on_floodfill_clicked()
{
    int gridsize = ui->grid_spin_box->value();
    p1.setX(ui->frame->x);
    p1.setY(ui->frame->y);
    int x = p1.x() / gridsize;
    int y = p1.y() / gridsize;
    x = x * gridsize + gridsize / 2;
    y = y * gridsize + gridsize / 2;
    if (ui->connected8_radio->isChecked())
    {
        point(x, y, 255, 255, 255);
    }
    else
    {
        point(x, y, 0, 0, 0);
    }
    if  (ui->connected8_radio->isChecked())
    {
        floodfillutil(x, y, 255, 255, 255);
    }
    else
    {
        floodfillutil(x, y, 0, 0, 0);
    }
}
```

OUTPUT :

Flood Fill 4 - connected



Grid Size 12 [✓ Show Grid]

Mouse Movement X : 12, Y : -1

Mouse Pressed X : -9, Y : 11

Radius : 0

R1 0 R2 0

Draw Polygon :

Polygon Filling : ○ 8 connected

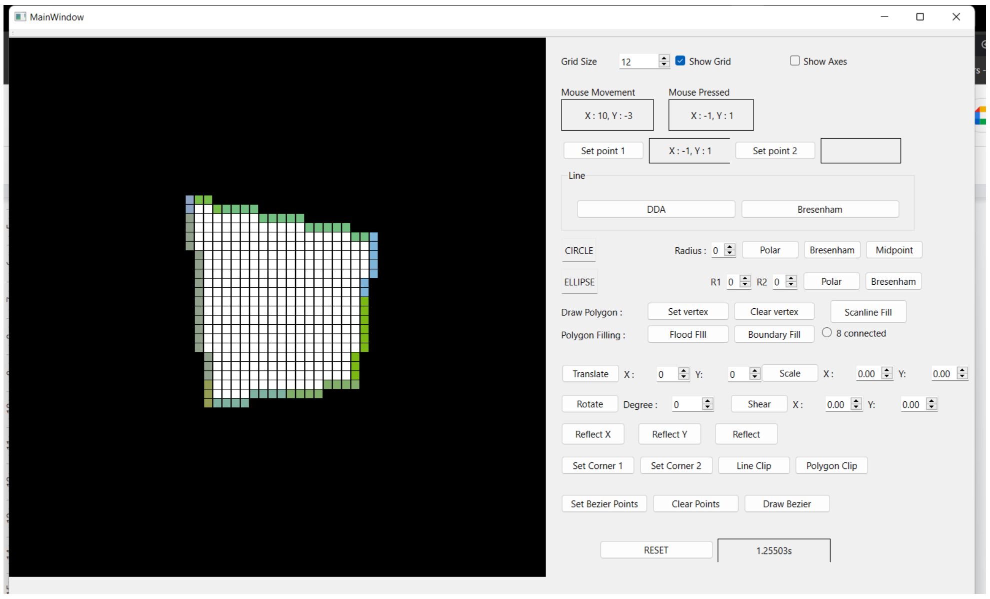
Translate X : 0 Y: 0

Scale X : 0.00 Y: 0.00

Rotate Degree : 0

Shear X : 0.00 Y: 0.00

1.25503s



Grid Size 12 [✓ Show Grid] Show Axes

Mouse Movement X : 10, Y : -3 Mouse Pressed X : -1, Y : 1

X : -1, Y : 1

Radius : 0

R1 0 R2 0

Draw Polygon :

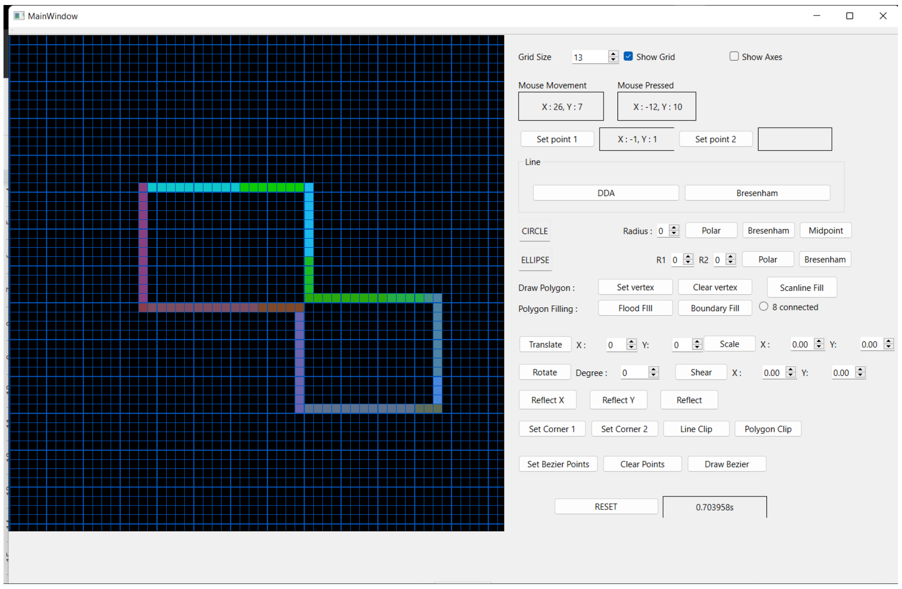
Polygon Filling : 8 connected

X : 0 Y: 0 X : 0.00 Y: 0.00

Degree : 0 X : 0.00 Y: 0.00

1.25503s

Flood Fill 8 - connected



Grid Size 13 [✓ Show Grid] Show Axes

Mouse Movement X : 26, Y : 7 Mouse Pressed X : -12, Y : 10

X : -1, Y : 1

Radius : 0

R1 0 R2 0

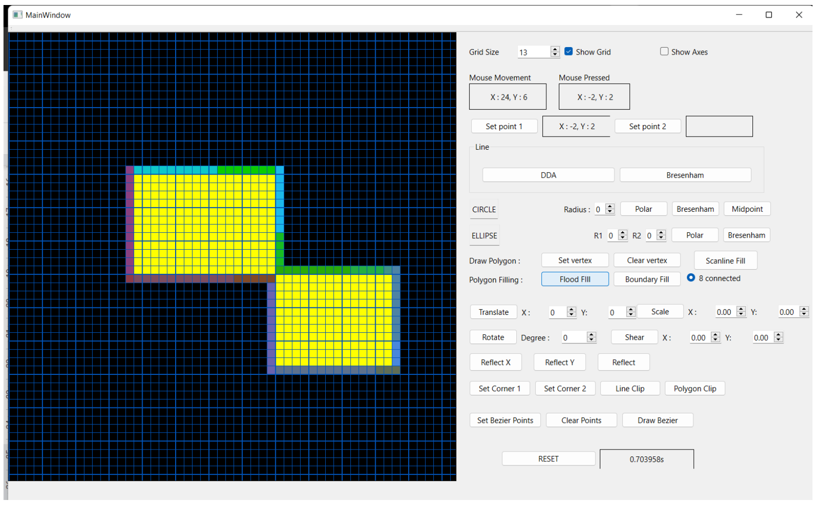
Draw Polygon :

Polygon Filling : 8 connected

X : 0 Y: 0 X : 0.00 Y: 0.00

Degree : 0 X : 0.00 Y: 0.00

0.703958s



Grid Size 13 [✓ Show Grid] Show Axes

Mouse Movement X : 24, Y : 6 Mouse Pressed X : -2, Y : 2

X : -2, Y : 2

Radius : 0

R1 0 R2 0

Draw Polygon :

Polygon Filling : [✓ 8 connected]

X : 0 Y: 0 X : 0.00 Y: 0.00

Degree : 0 X : 0.00 Y: 0.00

0.703958s

Assignment 9 : Implement Line Clipping with respect to a rectangular clip window, using a) Cohen-Sutherland Algorithm

CODE :

Helper Functions :

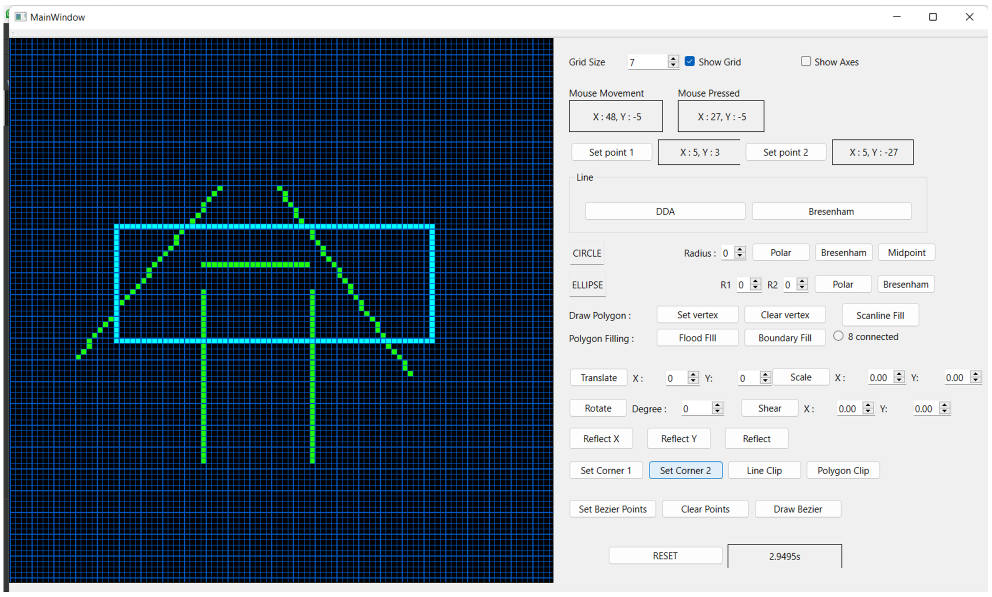
```
int clipper_points[4][2];
void MainWindow::on_setcorner1_clicked()
{
    int gridsize = ui->grid_spin_box->value();
    cp1.setX((ui->frame->x / gridsize) * gridsize + gridsize / 2);
    cp1.setY((ui->frame->y / gridsize) * gridsize + gridsize / 2);
}

void MainWindow::on_setcorner2_clicked()
{
    int gridsize = ui->grid_spin_box->value();
    cp2.setX((ui->frame->x / gridsize) * gridsize + gridsize / 2);
    cp2.setY((ui->frame->y / gridsize) * gridsize + gridsize / 2);

    clipper_points[0][0] = cp1.x();
    clipper_points[0][1] = cp1.y();
    clipper_points[1][0] = cp1.x();
    clipper_points[1][1] = cp2.y();
    clipper_points[2][0] = cp2.x();
    clipper_points[2][1] = cp2.y();
    clipper_points[3][0] = cp2.x();
    clipper_points[3][1] = cp1.y();

    draw_Window(0, 255, 255);
}

void MainWindow::draw_Window(int r = 0, int g = 255, int b = 255)
{
    p1.setX(clipper_points[0][0]);
    p1.setY(clipper_points[0][1]);
    p2.setX(clipper_points[1][0]);
    p2.setY(clipper_points[1][1]);
    on_dda_button_clicked(r, g, b);

    p1.setX(clipper_points[1][0]);
    p1.setY(clipper_points[1][1]);
    p2.setX(clipper_points[2][0]);
    p2.setY(clipper_points[2][1]);
    on_dda_button_clicked(r, g, b);

    p1.setX(clipper_points[2][0]);
    p1.setY(clipper_points[2][1]);
    p2.setX(clipper_points[3][0]);
    p2.setY(clipper_points[3][1]);
    on_dda_button_clicked(r, g, b);

    p1.setX(clipper_points[3][0]);
    p1.setY(clipper_points[3][1]);
    p2.setX(clipper_points[0][0]);
    p2.setY(clipper_points[0][1]);
    on_dda_button_clicked(r, g, b);

    p1 = temp1;
    p2 = temp2;
}

// Defining region codes
const int INSIDE = 0; // 0000
const int LEFT = 1;   // 0001
const int RIGHT = 2;  // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8;    // 1000

// Function to compute region code for a point(x, y)
int MainWindow::computeCode(int xa, int ya)
{
    int x_min = cp1.x(), x_max = cp2.x(), y_min = cp1.y(), y_max = cp2.y();

    // initialised as being inside
    int code = INSIDE;
    if (xa < x_min) // to the left of rectangle
        code |= LEFT;
    else if (xa > x_max) // to the right of rectangle
        code |= RIGHT;
    if (ya < y_min) // below the rectangle
        code |= BOTTOM;
    else if (ya > y_max) // above the rectangle
        code |= TOP;

    return code;
}

```

```
// Clipping a line from P1 = (x2, y2) to P2 = (x2, y2)
void MainWindow::cohenSutherlandClip(int x1, int y1,int x2, int y2)
{
                                                                                        int
x_min=clipper_points[0][0],x_max=clipper_points[2][0],y_min=clipper_points[0][1],y_max=c
lipper_points[2][1];
    // Compute region codes for P1, P2
    int code1 = computeCode(x1, y1);
    int code2 = computeCode(x2, y2);

    // Initialize line as outside the rectangular window
    bool isInside= false;

    while (true)
    {

        if ((code1 == 0) && (code2 == 0))
        {
            // If both endpoints lie within rectangle
            isInside = true;
            break;
        }
        else if (code1 & code2)
        {
            // If both endpoints are outside rectangle,
            // in same region
            break;
        }
        else
        {
            // Some segment of line lies within the
            // rectangle
            int code_out;
            int x, y;

            // At least one endpoint is outside the
            // rectangle, pick it.
            if (code1 != 0)
                code_out = code1;
            else
                code_out = code2;

            // Find intersection point;
            // using formulas y = y1 + slope * (x - x1),
            // x = x1 + (1 / slope) * (y - y1)
            if (code_out & TOP)
            {
                // point is above the clip rectangle
                  x = x1 + (int)((double)(x2 - x1) *(double)(y_max - y1) /(double)(y2 -
y1));

                y = y_max;
            }
            else if (code_out & BOTTOM)
            {
                // point is below the rectangle
                 x = x1 + (int)((double)(x2 - x1) * (double)(y_min - y1) / (double)(y2 -
y1));

                y = y_min;
            }
            else if (code_out & RIGHT)
            {
                // point is to the right of rectangle
                 y = y1 + (int)((double)(y2 - y1) * (double)(x_max - x1) / (double)(x2 -
x1));

                x = x_max;
            }
            else if (code_out & LEFT)
            {
                // point is to the left of rectangle
                 y = y1 + (int)((double)(y2 - y1) * (double)(x_min - x1) / (double)(x2 -
x1));

                x = x_min;
            }
            // Now intersection point x,y is found
            // We replace point outside rectangle
            // by intersection point
            if (code_out == code1)
            {
                x1 = x;
                y1 = y;
                code1 = computeCode(x1, y1);
            }
            else
            {
                x2 = x;
                y2 = y;
                code2 = computeCode(x2, y2);
            }
        }
    }
    if (isInside)
    {
        //If accepted
        //Just reset and draw the boundary and the line
        //Reset the screen and draw the grid

        p1.setX(x1);
        p1.setY(y1);

        p2.setX(x2);
        p2.setY(y2);

        on_dda_button_clicked(255,255,255);
        draw_Window();
    }
    else
    {
        //If not accepted
        //Just reset and draw the boundary
        //Reset the screen and draw the grid
        draw_Window();
    }

}
```

```

void MainWindow::on_lineclipping_clicked()
{

    for (int i = 0; i < (int)line_endpts.size(); i += 2)
    {
        int x1 = line_endpts[i].first;
        int y1 = line_endpts[i].second;
        int x2 = line_endpts[i + 1].first;
        int y2 = line_endpts[i + 1].second;
        p1.setX(x1);
        p1.setY(y1);
        p2.setX(x2);
        p2.setY(y2);
        on_dda_button_clicked(0, 0, 0);
        cohenSutherlandClip(x1, y1, x2, y2);
    }

    line_endpts.clear();
}
```

OUTPUT :

Rectangular window for clipping :



MainWindow

Grid Size 7 [✓ Show Grid] Show Axes

Mouse Movement X : 48, Y : -5

Mouse Pressed X : 27, Y : -5

Set point 1 X : 5, Y : 3

Set point 2 X : 5, Y : -27

Radius : 0

R1 0 R2 0

8 connected

Translate X : 0 Y : 0

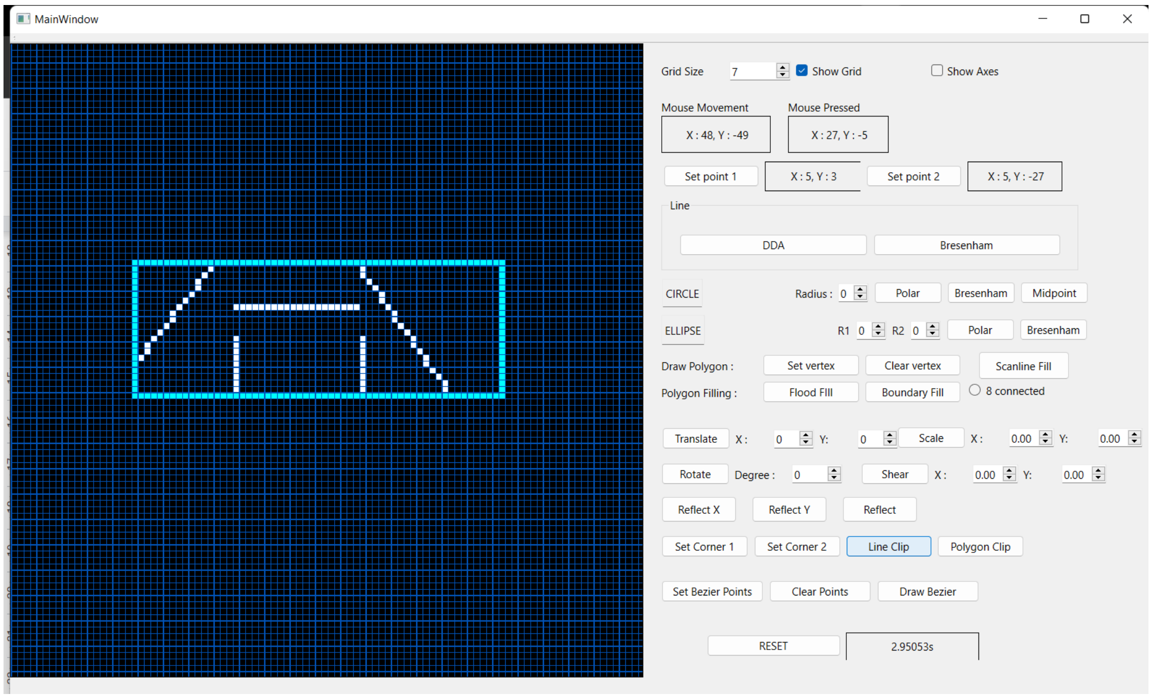
Scale X : 0.00 Y : 0.00

Rotate Degree : 0

Shear X : 0.00 Y : 0.00

2.9495s

After Clipping :



MainWindow

Grid Size 7 [✓ Show Grid] Show Axes

Mouse Movement X : 48, Y : -49

Mouse Pressed X : 27, Y : -5

Set point 1 X : 5, Y : 3

Set point 2 X : 5, Y : -27

Radius : 0

R1 0 R2 0

8 connected

Translate X : 0 Y : 0

Scale X : 0.00 Y : 0.00

Rotate Degree : 0

Shear X : 0.00 Y : 0.00

2.95053s