JADAVPUR UNIVERSITY

Name : Dibyendu Mukhopadhyay
Class : BCSE III
Roll Number : 001710501077
Subject : Computer Graphics Lab

1. DDA

Algorithm

```
1.  Start Algorithm
2.  Declare x1,y1,x2,y2,dx,dy,x,y , where x1 and y1 are source , and x2 and y2 are
    destinations
3.  Enter the value of x1,y1,x2,y2 (i.e, two endpoints)
4.  Calculate dx = x2-x1
5.  Calculate dy= y2-y1
6.  If fabs(dx)>fabs(dy)
        a.  Then,step =fabs(dy)
        b.  Else  step =fabs(dx)
7.  Xinc = dx/ step
8.  Yinc = dy/ step
9.  X =x1
10.  Y =y1
11.  putpixel(x,y,color)
12. X = x+ Xinc
13. Y = y +Yinc
14. putpixel(x,y,color)
15. Repeat step 12 until step
16. End Algorithm
```

Explanation

For generating any line segment, we need two endpoints and for calculating them by using a basic algorithm called DDA line generating algorithm.

Well, DDA stands for Digital Differential Analyzer, which is one of the incremental method of scan conversion of line. In this method calculation is performed at each step but by using the previous steps .

Suppose at step i, the pixels is (x_i,y_i)

The line of equation for step i

$y_i=mx_{i+b}$......................equation 1

Next value will be

$y_{i+1}=m_{xi+1}+b$.................equation 2

$m =\Delta y/\Delta x$

$y_{i+1}-y_i=\Delta y$.......................equation 3

$y_{i+1}-x_i=\Delta x$......................equation 4

$y_{i+1}=y_i+\Delta y$

$\Delta y=m\Delta x$

$y_{i+1}=y_i+m\Delta x$

$\Delta x=\Delta y/m$

$x_{i+1}=x_i+\Delta x$

$x_{i+1}=x_i+\Delta y/m$

Case1: When $|m|<1$ then (assume that $x_1$2)

$x= x_1, y=y_1$ set $\Delta x=1$

$y_{i+1}=y_{1+m}$, $x=x+1$

Until $x = x_2$

Case2: When $|m|<1$ then (assume that $y_1$2)

$x= x_1, y=y_1$ set $\Delta y=1$

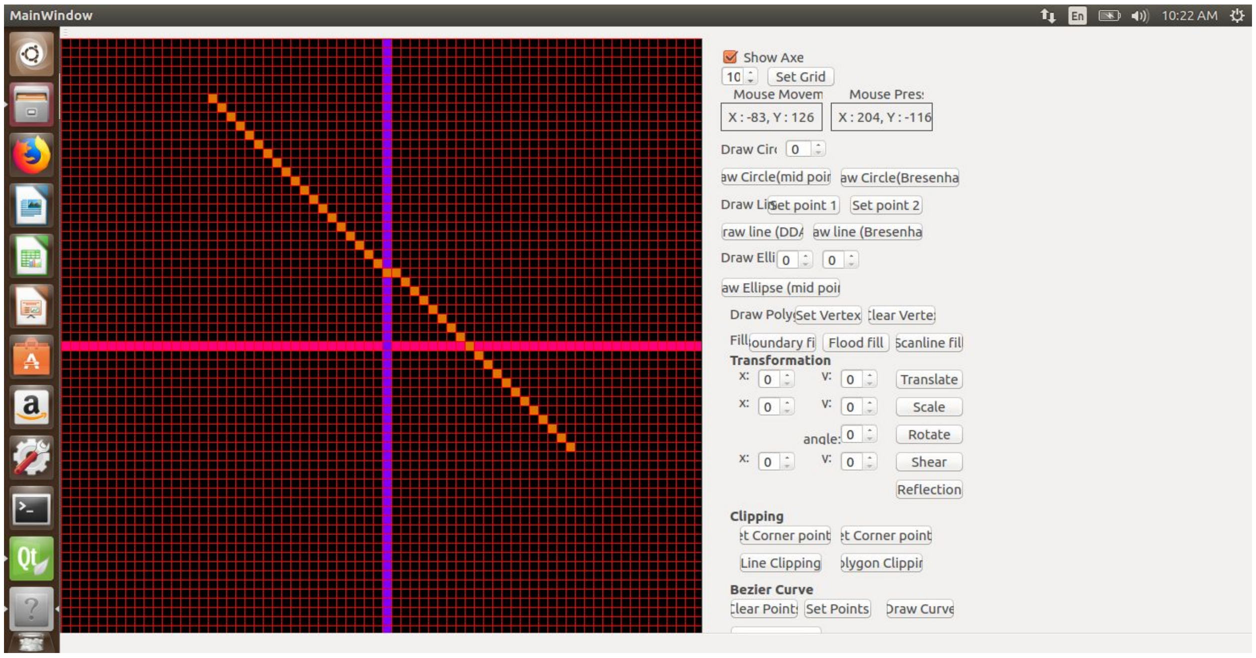
$x_{i+1}=1/m$, $y=y+1$

Until $y \rightarrow y_2$

Code Implementation

```
void MainWindow::on_DDAButton_clicked()
{
        int sourcex=p1.x()/gridsize;
        int sourcey=p1.y()/gridsize;
        int destx=p2.x()/gridsize;
        int desty=p2.y()/gridsize;
        double dx=destx-sourcex;
        double dy=desty-sourcey;

        double steps=fabs(dx)>fabs(dy)?fabs(dx):fabs(dy);

        double nextx=dx/steps;
        double nexty=dy/steps;
        double x=sourcex*gridsize+gridsize/2;
        double y=sourcey*gridsize+gridsize/2;

        for(int i=0;i<=steps;i++)
        {
        point((int)x,(int)y,220,120,0);
        x+=nextx*gridsize;
        y+=nexty*gridsize;
        }
        ui->frame->setPixmap(QPixmap::fromImage(img));

}
```

Output :



X : -83, Y : 126

X : 204, Y : -116

2. <u>Floodfill</u>

Algorithm

1. Start Algorithm
2. If getpixel(x,y) = old_color
3. putpixel (x, y, fill_color)

```
4. fill (x+1, y, fill_color, old_color)
5. fill (x-1, y, fill_color, old_color)
6. fill (x, y+1, fill_color, old_color)
7. fill (x, y-1, fill_color, old_color)
8. End Algorithm
```

Explanation :

In this method, a point / seed which is inside the region is pressed . This point is called a seed point. Then four connected approaches approaches is used to fill with desired color.

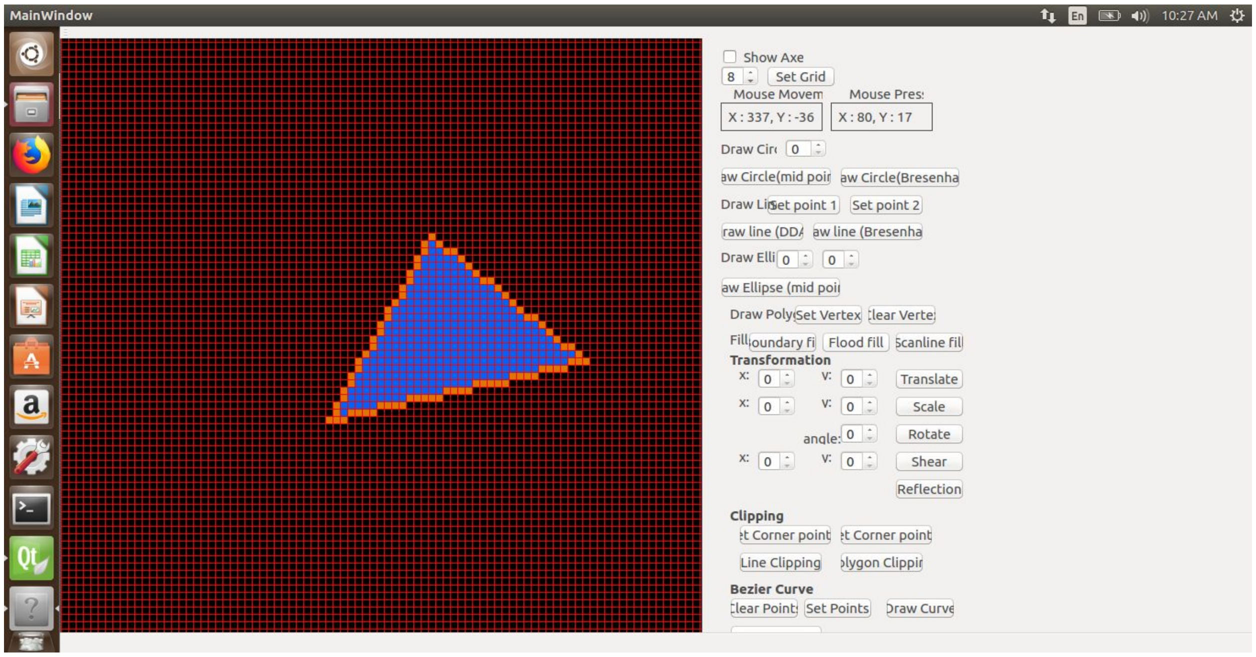
This method is more suitable for filling multiple colors boundary. When boundary is of many colors and interior is to be filled with one color, we use this algorithm.

In this algorithm, we start from a specified interior point (x, y) and reassign all pixel values which are currently set to a given interior color with the desired color. Using a 4-connected approaches, we then step through pixel positions until all interior points have been repainted .

Code Implementation

```
void MainWindow::floodfill(int fx,int fy,QRgb old_color,int r,int g,int b)
{
        if(img.pixel(fx,fy)==old_color)
        {
        point(fx,fy,r,g,b);
        floodfill(fx+gridsize,fy,old_color,r,g,b);
        floodfill(fx-gridsize,fy,old_color,r,g,b);
        floodfill(fx,fy+gridsize,old_color,r,g,b);
        floodfill(fx,fy-gridsize,old_color,r,g,b);
        }
}

void MainWindow::on_floodfillButton_clicked()
{
        int fx=ui->frame->x,fy=ui->frame->y;
        point(fx,fy,qRed(old_color),qGreen(old_color),qBlue(old_color));
        ui->frame->setPixmap(QPixmap::fromImage(img));
        fx=(fx/gridsize)*gridsize+gridsize/2;
        fy=(fy/gridsize)*gridsize+gridsize/2;
        floodfill(fx,fy,old_color,0,100,250);
}
```

Output :



3. <u>BEZIER CURVE</u>

Algorithm

1. Start Algorithm
2. Put control points(upto 4)
3. Use and apply mathematical formula :
   - x(u) = $(1-u)^3 * x_0 + 3 * u^1 * (1-u)^2 * x_1 + 3 * (1-u)^2 * u^2 * x_2 + u^3 * x_3$
   - y(u) = $(1-u)^3 * y_0 + 3 * u^1 * (1-u)^2 * y_1 + 3 * (1-u)^2 * u^2 * y_2 + u^3 * y_3$
4. putpixel(x(u),y(u), color)
5. End Algorithm

Explanation

A Bezier curve can be described by using a mathematical formula.

Given the coordinates of control points P_i: the first control point has coordinates $P_1 = (x_1, y_1)$, the second: $P_2 = (x_2, y_2)$, and so on, the curve coordinates are described by the equation that depends on the parameter u from the segment [0,1].

- The formula for a 2-points curve:
  $P = (1-u)P_1 + u*P_2$
- For 3 control points:
  $P = (1-u)^2P_1 + 2(1-u)*u*P_2 + u^2P_3$
- For 4 control points:
  $P = (1-u)^3P_1 + 3*(1-u)^2*u*P_2 + 3*(1-u)*u^2P_3 + u^3P_4$

These are vector equations. In other words, we can put `x` and `y` instead of `P` to get corresponding coordinates.

For instance, the 3-point curve is formed by points `(x,y)` calculated as:

- $x = (1-u)^2x_1 + 2(1-u)ux_2 + u^2x_3$
- $y = (1-u)^2y_1 + 2(1-u)uy_2 + u^2y_3$

Instead of x_1, y_1, x_2, y_2, x_3, y_3 we should put coordinates of upto 3 control points, and then as `u` moves from `0` to `1`, for each value of `u` we'll have `(x,y)` of the curve.

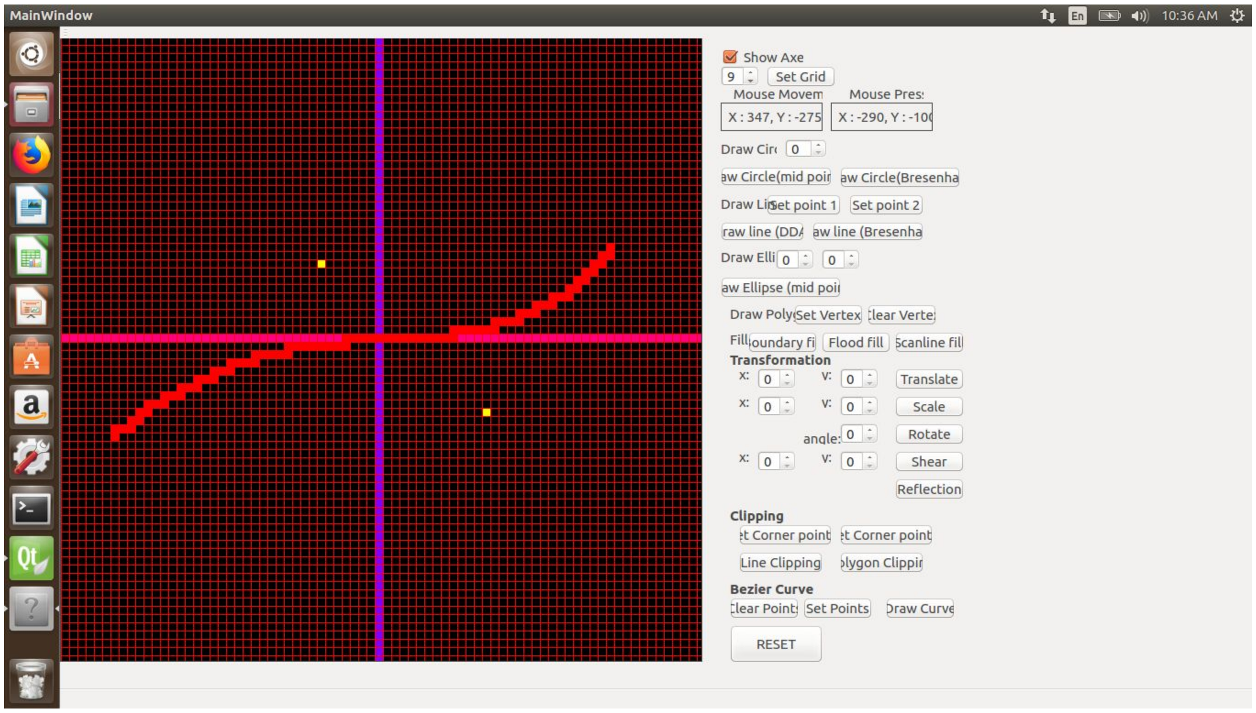
Now as `u` runs from `0` to `1`, the set of values `(x,y)` for each `u` forms the curve for such control points.

Code Implementation

```
std::vector<std::pair<int,int> > BezList;

void MainWindow::on_clearBezButton_clicked()
{
        BezList.clear();
}

void MainWindow::on_setBezButton_clicked()
{
        int k=gridsize;
        int x=((ui->frame->x)/k)*k+k/2;
        int y=((ui->frame->y)/k)*k+k/2;
        BezList.push_back(std::make_pair(x,y));

        int i=BezList.size();

        if(BezList.size()>1)
        {
```

```
        storeEdgeInTable(BezList[i-2].first, BezList[i-2].second, BezList[i-1].first,
BezList[i-1].second);//storage of edges in edge table.

        p1.setX(BezList[BezList.size()-1].first);
        p2.setX(BezList[BezList.size()-2].first);

        p1.setY(BezList[BezList.size()-1].second);
        p2.setY(BezList[BezList.size()-2].second);

        draw_DDA_line(0,255,0);

        }
}

void MainWindow::bezierCurve()
{
        double xu = 0.0 , yu = 0.0 , u = 0.0 ;
        int i = 0 ;
        for(u = 0.0 ; u <= 1.0 ; u += 0.0001)
        {
        xu =
pow(1-u,3)*BezList[0].first+3*u*pow(1-u,2)*BezList[1].first+3*pow(u,2)*(1-u)*BezList[2].first+pow
(u,3)*BezList[3].first;
        yu =
pow(1-u,3)*BezList[0].second+3*u*pow(1-u,2)*BezList[1].second+3*pow(u,2)*(1-u)*BezList[2].s
econd+pow(u,3)*BezList[3].second;
        point((int)xu , (int)yu,255,0,0) ;
        }
}

void MainWindow::on_drawBezierButton_clicked()
{
        bezierCurve();
}
```

Output :



Show Axe ✓

Set Grid: 9

Mouse Movem: X : 347, Y : -275

Mouse Pres: X : -290, Y : -10(

Draw Circ: 0

Draw Elli: 0 0

x: 0 y: 0 Translate

x: 0 y: 0 Scale

angle: 0 Rotate

x: 0 y: 0 Shear