ASSIGNMENT 2: IMPLEMENT DDA & BRESENHAM LINE DRAWING ALGORITHM.

DDA and Bresenham line drawing algorithms are very well-known algorithms to draw a line between two points in a raster graphical grid. As we know, raster graphics deals with pixels, so these algorithms give us a way to visualize any random line in a pixelized display. It is obvious that those lines will not be perfect, because they are drawn by filling some intermediate pixels. One thing to note in this context is that decrement in pixel size will significantly improve the visual quality of the line, but on the other hand, the execution time will increase. Now, let us discuss these two algorithms one by one.

DDA LINE DRAWING ALGORITHM:

DDA stands for **Digital Differential Analyzer**. It is an incremental method of scan conversion of lines. In this method, the calculation is performed at each step but by using the results of previous steps.

Consider we have a two-dimensional raster grid. We have taken two points(pixels) from this grid and want to draw a line between them using DDA algorithm. We will fill some intermediate pixels to draw the line. We know that the general line equation is –

$y = mx + b$

Suppose, Two points are – (x1,y1), (x2,y2),

At step i, the pixels is (x_i, y_i)

The equation at that step i gives us,

$$y_i = mx_i + b$$

Next value will be

$$y_{i+1} = mx_{i+1} + b$$

$$m = \frac{\Delta y}{\Delta x}$$

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

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

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

$$\Delta y = m\Delta x$$

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

$$\Delta x = \Delta y/m$$

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

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

Case1: When |M|<1 then

x= x1, y=y1 set $\Delta x=1$

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

Until x = x2

Case2: When |M|<1 then (assume that y12)

x= x1,y=y1 set $\Delta y=1$

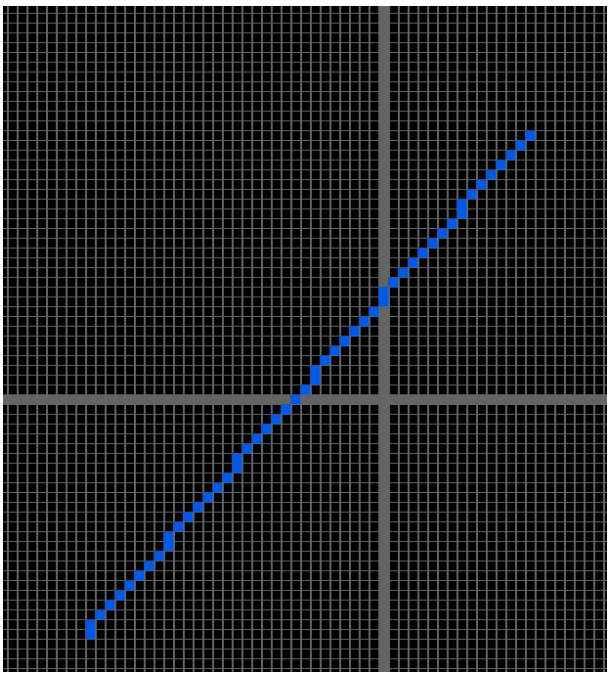
$x_{i+1} = x_i + 1/m$, y=y+1

Until y = y2

CPP CODE IN QT(GIVEN ONLY DDA FUNCTION):

```
void MainWindow::on_pushButton_2_clicked()
{
    int sss = ui->spinBox->value();
    float xi1 = ((p1.x()/sss))+0.5/sss;
    float yi1 = ((p1.y()/sss))+0.5/sss;

    float xi2 = ((p2.x()/sss))+0.5/sss;
    float yi2 = ((p2.y()/sss))+0.5/sss;

    int dx = xi1-xi2;
    int dy= yi1-yi2;

      int step;

      if (abs(dx) > abs(dy))
          step = abs(dx);
      else
          step = abs(dy);

        //calculate x-increment and y-increment for each step
      float x_incr = (float)dx / step;
      float y_incr = (float)dy / step;

        //take the initial points as x and y
      float x = xi2;
      float y = yi2;

      auto start = high_resolution_clock::now();

      for (int i = 0; i <= step; i ++) {
          fillPix(x,y, prim_col);
          x += x_incr;
          y += y_incr;
      }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    ui->label_6->setText(QString::number(duration.count()));

}
```

QT OUTPUT (DDA LINE):



Advantage:

1. It is a faster method than a method of using direct use of line equations.
2. This method does not use the multiplication theorem.
3. It allows us to detect the change in the value of x and y, so plotting the same point twice is not possible.
4. It is an easy method because each step involves just two additions.

Disadvantage:

1. It involves floating point additions rounding off is done. Accumulations of round-off error cause accumulation of error.
2. Rounding-off operations and floating point operations consume a lot of time.
3. It is more suitable for generating lines using the software. But it is less suited for hardware implementation.

BRESENHAM LINE DRAWING ALGORITHM:

This algorithm was developed by Bresenham. It only uses incremental integer calculations to decide intermediate pixel filling. As a result of avoiding floating point calculation, there is an improvement in execution speed compared to DDA. The concepts of this algorithm are even extended to draw other geometric figures. In Bresenham algorithm, the general case is when the line's slope is 0<m<1. The method for this general case can be extended to m>1 by just exchanging the x and y in the algorithm. This algorithm may be generalized to any arbitrary sloped line, considering the symmetry between the various octants.

For |m|<1, the number of y increments is less than the number of x increments. So, in each step, we will increase the x value. But using the decision parameter we will decide whether to increase the value of y or not. So, for each known pixel, there are two possibilities for the next pixel – either right or top-right. In the decision parameter, it is checked which pixel among those two is closer to the actual line.

The algorithm for |m|<1 dictates –

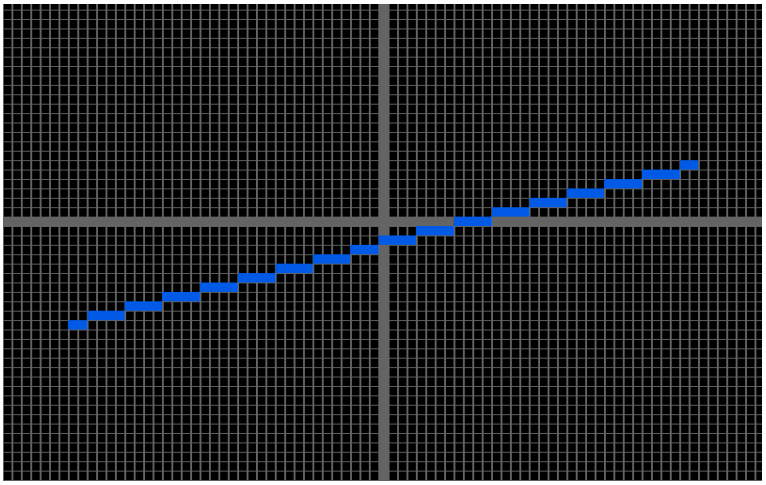
1. Input the two endpoints and store the left endpoint(x0,y0).
2. Plot (x0,y0) into raster grid.
3. Calculate constants such as – Δx, Δy, $2\Delta y$, $2\Delta y$-$2\Delta x$ and the starting value of the decision parameter is – $p = 2\Delta y - \Delta x$.
4. For each x_k along the line, starting at k=0, test if $p_k<0$, then next point will be – (x_k+1,yk) and $P_{k+1} = p_k + 2\Delta y$,
   Otherwise, next point is (x_k+1,y_k+1) and $p_{k+1} = p_k + 2\Delta y - 2\Delta x$.
5. Now, fill the chosen point.
6. Repeat STEP 4 & 5, Δx times.

CPP CODE (ONLY BRESENHAM LINE DRAWING FUNCTION):

```
void MainWindow::on_pushButton_3_clicked()
{
    int sss = ui->spinBox->value();
    int xa = p1.x()/sss, xb = p2.x()/sss, ya = (p1.y())/sss, yb = (p2.y())/sss;
    int dx = abs(xa-xb), dy = abs(ya-yb);
    bool yy = false;
    if(dy>dx){
        yy = true;
    }
    int p = 2*dy-dx;
    int twoDy = 2*dy, twoDyDx = 2*(dy-dx);
    int x, y, xEnd, ydir=1,yEnd;

    if(!(ya>yb != xa<xb)){
```

```
        ydir=-1;
    }

    if(xa>xb){
        x = xb;
        y = yb;
        xEnd = xa;
    }else{
        x = xa;
        y = ya;
        xEnd = xb;
    }

    fillPix(x,y,prim_col);
    auto start = high_resolution_clock::now();
    if(!yy){
        while(x<xEnd){
            x++;
            if(p<0){
                p += twoDy;
            }else{
                y+=ydir;
                p += twoDyDx;
            }
            fillPix(x,y,prim_col);
        }
    }else{
        if(ya>yb){
            x = xb;
            y = yb;
            yEnd = ya;
        }else{
            x = xa;
            y = ya;
            yEnd = yb;
        }

        p = 2*dx-dy;
        int twoDx = 2*dx;
        twoDyDx *= -1;
        while(y<yEnd){
            y++;
            if(p<0){
                p += twoDx;
            }else{
                x+=ydir;
                p += twoDyDx;
            }
            fillPix(x,y,prim_col);
        }
    }

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    ui->label_6->setText(QString::number(duration.count()));
}
```

QT OUTPUT (BRESENHAM LINE):



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

These two are two well-known filling algorithm for any closed 2d shape. These mostly are similar to graph traversal algorithms. Interactive software like MS Paint uses similar but more improved version of this algorithms to fill a shape. Filling algorithms may be 4-connected or 8-connected.

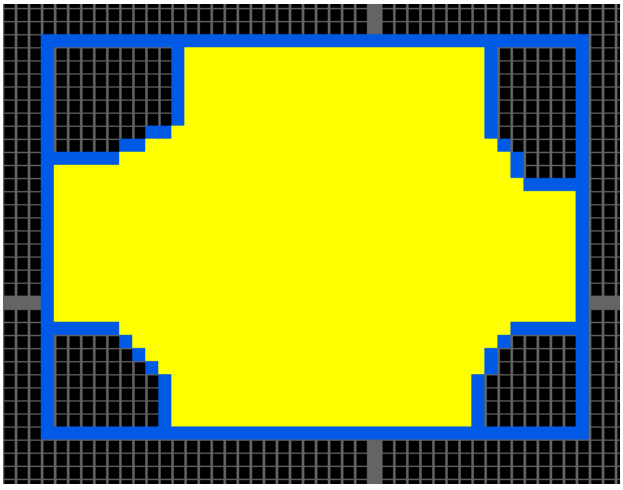
BOUNDARY FILL

CPP CODE (BOUNDARY FILL FUNCTION):

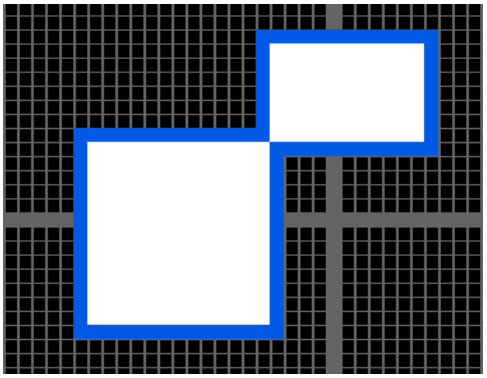
```
void MainWindow::recurTillBound(int x, int y){ // VIS IS A FUNCTION TO CHECK IF
                                                // THAT POINT ALREADY VISITED OR NOT
    if(vis[x][y]) return;
    vis[x][y] = true;

    if(is8) // is8 is just shows 8-connected filling
        fillPix(x,y,QColor::fromRgb(255,255,255));
    else
        fillPix(x,y,QColor::fromRgb(255,255,0));

    delay();
    recurTillBound(x+1,y);
    recurTillBound(x,y-1);
    if(is8){
      recurTillBound(x+1,y+1);
      recurTillBound(x+1,y-1);
      recurTillBound(x-1,y+1);
      recurTillBound(x-1,y-1);
    }

    recurTillBound(x-1,y);
    recurTillBound(x,y+1);
}


void MainWindow::on_boundaryF_clicked()
{
    int sss = ui->spinBox->value();
    int xc = circCenter.x()/sss, yc = circCenter.y()/sss;
    vis[xc][yc] = false;
    recurTillBound(xc,yc);
}
```

QT OUTPUT(BOUNDARY FILL):

4-CONNECTED→



8-CONNECTED→



FLOOD FILL

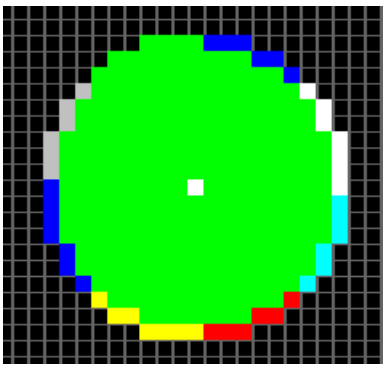
CPP CODE (FLOOD FILL FUNCTION):

```
void MainWindow::floodUtils(int x, int y, QColor prev, QColor newc){
     int sss = ui->spinBox->value();
     QColor col = img.pixelColor(sss*((float)x+0.5), sss*((float)y+0.5));

     if(prev!=col || newc==col){
        return;
     }

     fillPix(x,y,newc);
     delay();


     floodUtils(x+1,y,prev,newc);
     floodUtils(x-1,y,prev,newc);
     if(is8){
         floodUtils(x+1,y+1,prev,newc);
         floodUtils(x+1,y-1,prev,newc);
         floodUtils(x-1,y+1,prev,newc);
         floodUtils(x-1,y-1,prev,newc);
     }
     floodUtils(x,y+1,prev,newc);
     floodUtils(x,y-1,prev,newc);
}
```

```
void MainWindow::on_floodFill_clicked()
{
    int sss = ui->spinBox->value();
    int xc = circCenter.x()/sss, yc = circCenter.y()/sss;

    fillPix(xc,yc,prevc);

    if(prevc!=Qt::green){
        floodUtils(xc,yc,prevc,Qt::green);
    }else{
        floodUtils(xc,yc,prevc,Qt::red);
    }

}
```

QT OUTPUT(FLOOD FILL):

We know that flood fill works irrespective of boundary color. Here is an example output



Assignment 7 : Draw a closed polygon and implement different transformation functions (with respect to origin) on it. a) translation, b) rotation, c) scaling, d) shear, e) reflection with respect to x/y axes. Extend the algorithm to apply the transformations successively on the same object using homogeneous coordinates and matrix multiplication.

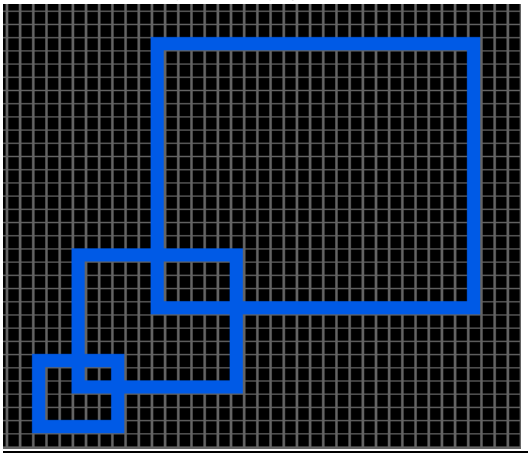
CPP CODE(ALL TRANSFORMATIONS):

```
struct point {
    float x;
    float y;
};

point translate(point p, int tx, int ty){
    point ans;
    ans.x = p.x + tx;
    ans.y = p.y + ty;
    return ans;
}

point rotate(point p, int angle_deg){
    float angle_radian = 3.14/180 * angle_deg;
```

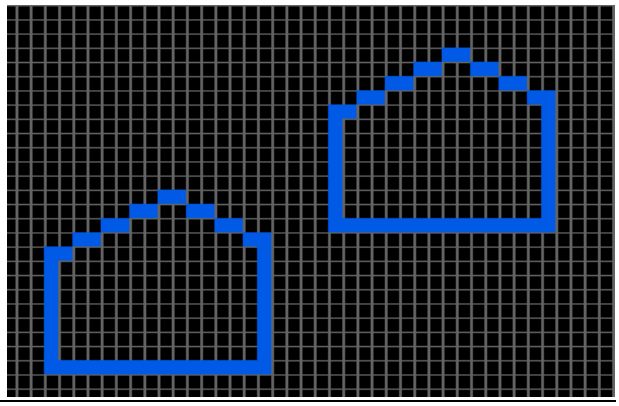
```
    point ans;
    ans.x = p.x * cos(angle_radian) - p.y * sin(angle_radian);
    ans.y = p.x * sin(angle_radian) + p.y * cos(angle_radian);
    return ans;
}

point scale(point p, float sx, float sy){
    point ans;
    ans.x = sx * p.x;
    ans.y = sy * p.y;
    return ans;
}

point shear(point p, float shx, float shy){
    point ans;
    ans.x = p.x+shx * p.y;
    ans.y = p.y+shy * p.x;
    return ans;
}

void MainWindow::on_scaleUI_clicked()
{

    vector<struct point> finalpts;
    int sss = ui->spinBox->value();

    for(QPoint pp : points){
        struct point p;
        p.x = floor(pp.x()/sss)*sss-350;
        p.y = floor(-pp.y()/sss)*sss+350;
        finalpts.push_back(scale(p,ui->sx->value(),ui->sy->value()));
    }

    points.clear();
    for(struct point p: finalpts){
        points.push_back(QPoint(p.x+350,-p.y+350));
    }

    on_drawPoly_clicked();
}


void MainWindow::on_rotateUI_clicked()
{

    vector<struct point> finalpts;
int sss = ui->spinBox->value();

    for(QPoint pp : points){
        struct point p;
        p.x = floor(pp.x()/sss)*sss-350;
        p.y = floor(-pp.y()/sss)*sss+350;
        finalpts.push_back(rotate(p,ui->theta->value()));
    }

    points.clear();
    for(struct point p: finalpts){
        points.push_back(QPoint(p.x+350,-p.y+350));
    }

    on_drawPoly_clicked();
}


void MainWindow::on_translateUI_clicked()
{
    vector<struct point> finalpts;
    int sss = ui->spinBox->value();
```

```
    for(QPoint pp : points){
        struct point p;
        p.x = floor(pp.x()/sss)*sss-350;
        p.y = floor(-pp.y()/sss)*sss+350;
        finalpts.push_back(translate(p,ui->tx->value()*sss,ui->ty->value()*sss));
    }

    points.clear();
    for(struct point p: finalpts){
        points.push_back(QPoint(p.x+350,-p.y+350));
    }

    on_drawPoly_clicked();
}


void MainWindow::on_reflectUI_clicked()
{
    float sx = ui->sx->value();
    float sy = ui->sy->value();

    ui->sx->setValue(1.0);
    ui->sy->setValue(-1.0);

    on_scaleUI_clicked();

    ui->sx->setValue(sx);
    ui->sy->setValue(sy);
}


void MainWindow::on_reflectYUI_clicked()
{
    float sx = ui->sx->value();
    float sy = ui->sy->value();

    ui->sx->setValue(-1.0);
    ui->sy->setValue(1.0);

    on_scaleUI_clicked();

    ui->sx->setValue(sx);
    ui->sy->setValue(sy);
}


void MainWindow::on_shearUI_clicked()
{
    vector<struct point> finalpts;
    int sss = ui->spinBox->value();

    for(QPoint pp : points){
        struct point p;
        p.x = floor(pp.x()/sss)*sss-350;
        p.y = floor(-pp.y()/sss)*sss+350;
        finalpts.push_back(shear(p,ui->shearx->value(),ui->sheary->value()));
    }

    points.clear();
    for(struct point p: finalpts){
        points.push_back(QPoint(p.x+350,-p.y+350));
    }

    on_drawPoly_clicked();
}
```

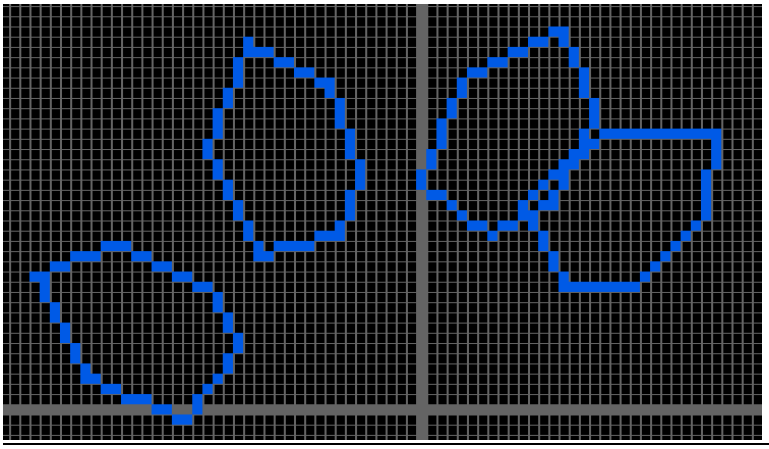
QT OUTPUT(SCALE);



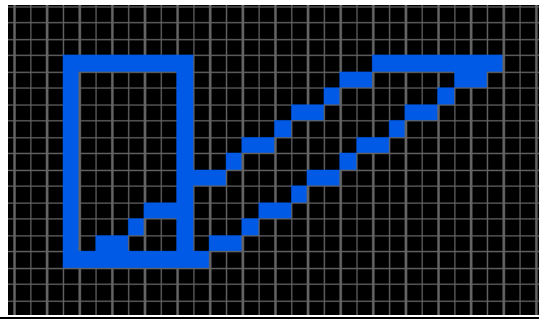
QT OUTPUT(TRANSLATE ALONG X & Y):



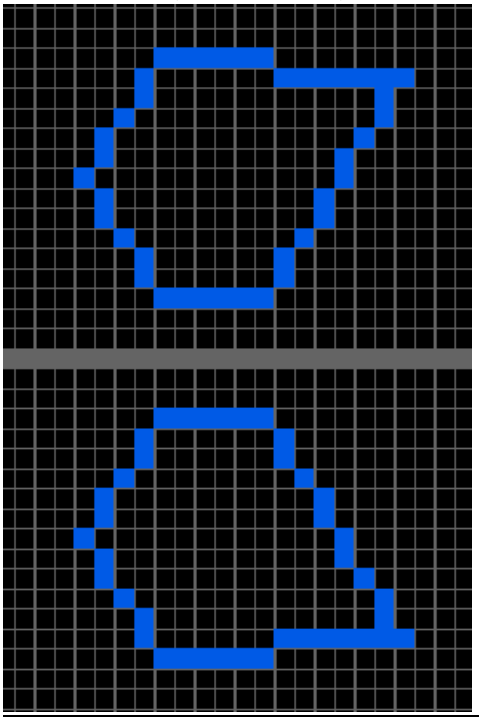
QT OUTPUT(ROTATE):



QT OUTPUT(SHEAR):



QT OUTPUT(REFLECT x)



QT OUTPUT(REFLECT Y)