PARTA

|  |
| --- |
| 1. Draw the following pattern using any Line drawing algorithms |

## DDA Algorithm

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

**Step 1** − Get the input of two end points (*X*0,*Y*0)

and (*X*1,*Y*1)

.

**Step 2** − Calculate the difference between two end points.

dx = X1 - X0

dy = Y1 - Y0

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

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

Steps = absolute(dx);

else

Steps = absolute(dy);

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

Xincrement = dx / (float) steps;

Yincrement = dy / (float) steps;

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

for(int v=0; v < Steps; v++)

{

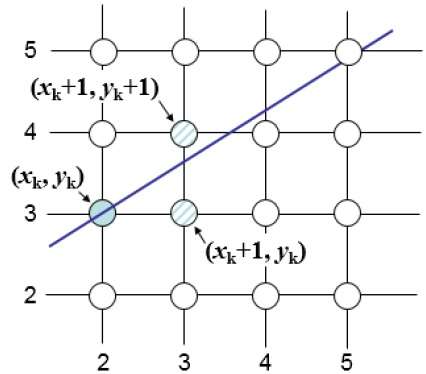
x = x + Xincrement;

y = y + Yincrement;

putpixel(Round(x), Round(y));

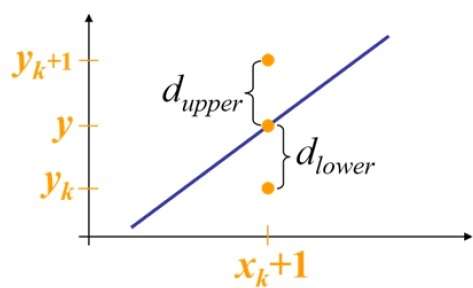
}

## Bresenham’s Line Generation



At sample position *X**k*+1,the vertical separations from the mathematical line are labelled as *d**u**p**p**e**r* and *d**l**o**w**e**r*

.



**Step 1** − Input the two end-points of line, storing the left end-point in (*x*0,*y*0)

.

**Step 2** − Plot the point (*x*0,*y*0)

.

**Step 3** − Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as −

*p*0=2*d**y*−*d**x*

**Step 4** − At each *X**k*

along the line, starting at k = 0, perform the following test −

If *p**k*

< 0, the next point to plot is (*x**k*+1,*y**k*)

and

*p**k*+1=*p**k*+2*d**y*

Otherwise,

(*x**k*,*y**k*+1)

*p**k*+1=*p**k*+2*d**y*−2*d**x*

**Step 5** − Repeat step 4 (dx – 1) times.

For m > 1, find out whether you need to increment x while incrementing y each time.

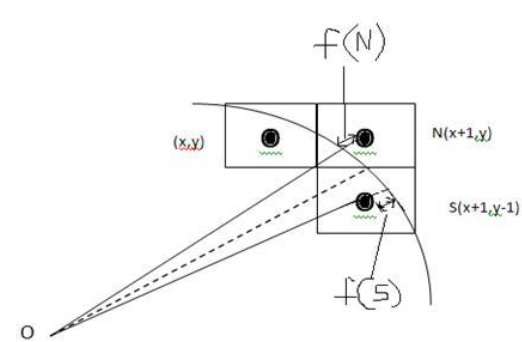
After solving, the equation for decision parameter *P**k*

will be very similar, just the x and y in the equation gets interchanged.

|  |
| --- |
| 2. Draw inscribed and Circumscribed circles in the triangle as shown as an example below.(Use any Circle drawing and Line drawing algorithms) |

We cannot display a continuous arc on the raster display. Instead, we have to choose the nearest pixel position to complete the arc.

From the following illustration, you can see that we have put the pixel at (X, Y) location and now need to decide where to put the next pixel − at N (X+1, Y) or at S (X+1, Y-1).



This can be decided by the decision parameter **d**.

* If d <= 0, then N(X+1, Y) is to be chosen as next pixel.
* If d > 0, then S(X+1, Y-1) is to be chosen as the next pixel.

### Algorithm

**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 and Q=R.

**Step 2** − Set decision parameter D = 3 – 2R.

**Step 3** − Repeat through step-8 while X < Y.

**Step 4** − Call Draw Circle (X, Y, P, Q).

**Step 5** − Increment the value of P.

**Step 6** − If D < 0 then D = D + 4x + 6.

**Step 7** − Else Set Y = Y + 1, D = D + 4(X-Y) + 10.

**Step 8** − Call Draw Circle (X, Y, P, Q).

|  |
| --- |
| 3. Draw the polygons by using the mouse. Choose colors by clicking on the designed color pane. Use window port to draw. (Use DDA algorithm for line drawing) |

Detecting Mouse Click  
GLUT provides a function glutMouseFunc which is responsible for detecting mouse click generated by a program. Its syntax is   
void glutMouseFunc(void (\*func)(int button, int state, int x, int y));  
Where,

|  |
| --- |
| func –> Function that handles the mouse event i.e. when mouse click is detected, func is called |
| func must have three parameters, as we see in above syntax The first parameter is button, its value is one of the following |
| * GLUT\_LEFT\_BUTTON – when left mouse button click is detected * GLUT\_RIGHT\_BUTTON  - when right mouse button click is detected * GLUT\_MIDDLE\_BUTTON – when middle mouse button click is detected |
| The second parameter is state, its value is one of the following |
| * GLUT\_DOWN – When mouse button is pressed * GLUT\_UP – When mouse button is released |
| When callback is generated with GLUT\_DOWN GLUT environment assumes that GLUT\_UP comes afterward even if mouse is not inside the window. |
| Other two parameters x and y are coordinates of the point where mouse pointer is moved. |

Detecting Mouse Motion

GLUT provides two types of function as according to types of motion detection either Active Motion or Passive Motion. Active Motion refers to the Motion of mouse when there is Click and Passive motion refers to motion of a mouse without a click. The syntax of two functions are

|  |
| --- |
| glutMotionFunc(void (\*func)(int x, int y)); //for Active motion |
| glutMousePassiveMotionFunc(void (\*func)(int x, int y)) // for Passive motion |
| Where, |
| func –> Function that responsible for handling respective motion |
| x, y –> coordinates of the mouse relative to upper left corner of w |

|  |
| --- |
| 4. Draw a 8X8 chessboard rotated 45˚ with the horizontal axis. Use Bresenham algorithm to draw all the lines. Use seed fill algorithm to fill black squares of the rotated chessboard. |

Flood fill, also called seed fill, is an algorithm that determines the area connected to a given

node in a multi-dimensional array. It is used in the "bucket" fill tool of paint programs to

determine which parts of a bitmap to fill with color, and in puzzle games such as

Minesweeper, PuyoPuyo, Lumines, Samegame and Magical Drop for determining which

pieces are cleared.

A method exists that uses essentially no memory for four-connected regions by

pretending to be a painter trying to paint the region without painting themselves into a corner.

This is also a method for solving mazes. The four pixels making the primary boundary are

examined to see what action should be taken. The painter could find themselves in one of

several conditions:

1. All four boundary pixels are filled.

2. Three of the boundary pixels are filled.

3. Two of the boundary pixels are filled.

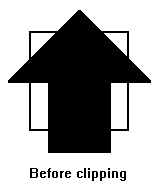
4. One boundary pixel is filled.

5. Zero boundary pixels are filled.

PART B

|  |
| --- |
| 1. Implement Cohen Sutherland Hodgman algorithm to clip any given polygon. Provide the vertices of the polygon to be clipped and pattern of clipping interactively. |

The Sutherland - Hodgman algorithm performs a clipping of a polygon against each window edge in turn. It accepts an ordered sequence of verices v1, v2, v3, ..., vn and puts out a set of vertices defining the clipped polygon.

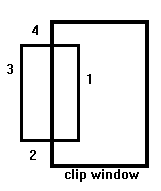
This figure represents a polygon (the large, solid, upward pointing arrow) before clipping has occurred.

The following figures show how this algorithm works at each edge, clipping the polygon.

1. Clipping against the left side of the clip window.
2. Clipping against the top side of the clip window.
3. Clipping against the right side of the clip window.
4. Clipping against the bottom side of the clip window.

## Four Types of Edges

As the algorithm goes around the edges of the window, clipping the polygon, it encounters four types of edges. All four edge types are illustrated by the polygon in the following figure. For each edge type, zero, one, or two vertices are added to the output list of vertices that define the clipped polygon.



The four types of edges are:

1. Edges that are totally inside the clip window. - add the second inside vertex point
2. Edges that are leaving the clip window. - add the intersection point as a vertex
3. Edges that are entirely outside the clip window. - add nothing to the vertex output list
4. Edges that are entering the clip window. - save the intersection and inside points as vertices

|  |
| --- |
| 2. Implement translation, sheer, rotation and scaling transformations on equilateral triangle and rhombus. |

### Rotation in 2D

point (X,Y) is to be rotated about the origin by angle theta to location (X',Y')

X' = X \* cos(theta) - Y \* sin(theta)  
Y' = X \* sin(theta) + Y \*cos(theta)

note that this does involve sin and cos which are much more costly than addition or multiplication

or P' = R \* P where

\_ \_

P' = | X' |

| Y' |

- -

\_ \_

R = | cos(theta) -sin(theta) |

| sin(theta) cos(theta) |

- -

\_ \_

P = | X |

| Y |

- -

rotation is performed **about the origin (0,0)** not about the center of the line/polygon/whatever

### Derivation of the 2D Rotation Equations

Where does this matrix come from?

(X,Y) is located r away from (0,0) at a CCW angle of phi from the X axis.  
(X',Y') is located r away from (0,0) at a CCW angle of theta+phi from the X axis.

Since rotation is about the origin, (X',Y') must be the same distance from the origin as (X,Y).

from trigonometry we have:

X = r \* cos(phi)  
Y = r \* sin(phi)

and

X' = r \* cos(theta+phi)  
Y' = r \* sin(theta+phi)

Now making use of the following trigonometric identities:

cos(a+b) = cos(a) \* cos(b) - sin(a) \* sin(b)  
sin(a+b) = sin(a) \* cos(b) + cos(a) \* sin(b)

and substituting in for the above equations for X' and Y', we get:

X' = r \* cos(theta) \* cos(phi) - r \* sin(theta) \* sin(phi)  
Y' = r \* sin(theta) \* cos(phi) + r \* cos(theta) \* sin(phi)

Then we substitute in X and Y from their definitions above, and the final result simplifies to:

X' = X \* cos(theta) - Y \* sin(theta)  
Y' = X \* sin(theta) + Y \* cos(theta)

### Rotation of 2D Homogenous Coordinates

P' = R \* P where

\_ \_

P' = | X' |

| Y' |

| 1 |

- -

\_ \_

R = | cos(theta) -sin(theta) 0 | = R(theta)

| sin(theta) cos(theta) 0 |

| 0 0 1 |

- -

\_ \_

P = | X |

| Y |

| 1 |

- -

|  |
| --- |
| 3. Implement Cube rotation about vertical axis passing through its centroid. |

We use homogeneous coordinates from the beginning. This means that the general transformation matrix is a 4x4 matrix, and that the general vector form is a column vector with four rows.

P2=M·P1

## Translation

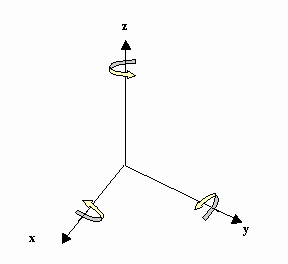
|  |  |
| --- | --- |
|  | A translation in space is described by tx, ty and tz. It is easy to see that this matrix realizes the equations:  x2=x1+tx  y2=y1+ty  z2=z1+tz |

## Scaling

|  |  |
| --- | --- |
|  | Scaling in space is described by sx, sy and sz. We see that this matrix realizes the following equations:  x2=x1·sx  y2=y1·sy  z2=z1·sz |

## Rotation

Rotation is a bit more complicated. We define three different basic rotations, one around every axis.



|  |  |  |
| --- | --- | --- |
| Around the Z-axis | Around the X-axis | Around the Y-axis |

## Geometry

The identity matrix

|  |  |
| --- | --- |
|  | Again we can interpret it as:   * Scale with 1 along all axes * Rotate with 0, around all axes * Translate with 0 in all axe directions |

Mirroring

|  |  |
| --- | --- |
|  | We can mirror the different planes by using scaling factor -1 on the axis that is placed normally on the plane. Notice the matrix to the left. It mirrors around the xy-plane, and changes the coordinates from a right hand system to a left hand system. |

Projection

|  |  |
| --- | --- |
|  | So far we have only concentrated on moving points around in plane and space. Later we will worry about how we can depict space figures on a plane. It can be useful to notice that this can be done with a matrix operation. We can project a point orthogonal down into one of the main planes by using a matrix that scale the axis normally onto the plane with 0. The matrix to the left is a parallel projection down into the xy-plane. |

|  |
| --- |
| 4. Generate fractal patterns by using Koch curves |

## The Koch Snowflake

From the Koch Curve, comes the Koch Snowflake. Instead of one line, the snowflake begins with an equilateral triangle. The steps in creating the Koch Curve are then repeatedly applied to each side of the equilateral triangle, creating a "snowflake" shape.

## The Math Behind It

NUMBER OF SIDES (n)

For each iteration, one side of the figure from the previous stage becomes four sides in the following stage. Since we begin with three sides, the formula for the number of sides in the Koch Snowflake is

n = 3\*4a

in the ath iteration.

For iterations 0, 1, 2 and 3, the number of sides are 3, 12, 48 and 192, respectively.

LENGTH OF A SIDE (length)

In every iteration, the length of a side is 1/3 the length of a side from the preceding stage. If we begin with an equilateral triangle with side length x, then the length of a side in interation a is

length = x\*3-a

For iterations 0 to 3, length = a, a/3, a/9 and a/27.

PERIMETER (p)

Since all the sides in every iteration of the Koch Snowflake is the same the perimeter is simply the number of sides multiplied by the length of a side

p = n\*length

p = (3\*4a)\*(x\*3-a)

for the ath iteration.

Again, for the first 4 iterations (0 to 3) the perimeter is 3a, 4a, 16a/3, and 64a/9.

As we can see, the perimeter increases by 4/3 times each iteration so we can rewrite the formula as

p = (3a)\*(4/3)a

So as a-->infinity the perimeter continues to increase with no bound.

Also, as a-->infinity the snowflake is made up of sharp corners with no smooth lines connecting them. Therefore, while the perimeter of the snowflake, which is an infinite series, is continuous because there are no breaks in the perimeter, it is not differentiable since there are no smooth lines.