Title:

Implementing DDA line drawing algorithm using GLUT

Objective:

i) Implementing the Digital Differential Analyzer algorithm.
ii) Understanding it's mechanism.

Theory:

DDA is a hardware/software used for interpolation of variables over an interval between a start and end point. This algorithm is used to draw a line. We calculate the slope m in this algorithm. There are two cases:-

Case 1: $m<1$

$x = x_1$
$y = y_1$
Set $\Delta x = 1$
$y_{i+1} = y_i + m$
$x = x+1$
until $x = x_2$

Case 2: $m>1$

$x = x_1$
$y = y_1$
Set $\Delta y = 1$
$x_{i+1} = x + \frac{1}{m}$
$y = y+1$
until $y = y_2$

Here, (x_1, y_1) & (x_2, y_2) are two end points.

Algorithm:

1. Get two end points (x_1, y_1) & (x_2, y_2) where $x_1 \neq x_2$ & $y_1 \neq y_2$
2. Calculate $dx = x_2 - x_1$
   $dy = y_2 - y_1$

3. Calculate $m = \frac{dy}{dx}$

Set $x = x_1$, $y = y_1$

4. Case (i)
if $(m < 1)$
$x = x_1 + 1$
$y = y + m$
until $x = x_2$

Case (ii)
if $(m > 1)$
$x = x + \frac{1}{m}$
$y = y + 1$
until $y = y_2$

5. Put pixel by rounding x & y.

6. End.

Used Resources:

1. Vs Code

Used programming language & dependencies:

1. C++
2. GLUT OpenGL utility toolkit.

Procedure:

1. A C++ file was opened
2. The algorithm was implemented using Glut.
3. The code was compiled and run.
4. Result was observed.

Conclusion:

DDA is a very simple algorithm. I faced no difficulty while implementing it. But the program sometimes halted during runtime.

Title:

Implementing Bresenham's line drawing algorithm.

Objective:

i) To implement Bresenham's algorithm using OpenGL
ii) To understand the mechanism of the algorithm.

Theory:

Bresenham's algorithm is an incremental conversion algorithm across the X-axis in unit interval where each step chooses between two different y-coordinates.

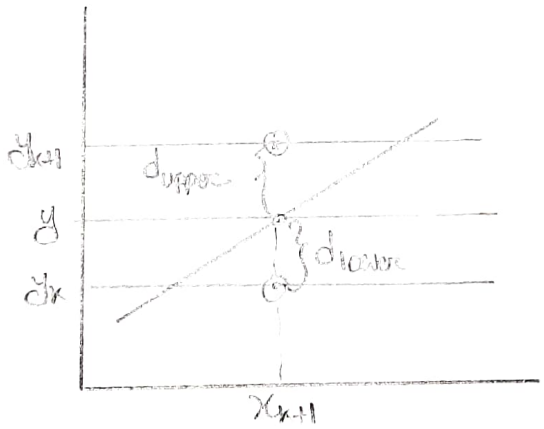


fig. X_{k+1} point and choice of y point.

In the figure, at sample position X_{k+1} the vertical seperations from the mathematical line are labelled as dupper and dlower.

Here,

$$y = m(x_{k+1}) + b$$

$$d_{upper} = y_{k+1} - y$$
$$= y_{k+1} - m(x_{k+1}) - b$$

$$d_{lower} = y - y_{k+1}$$
$$= m(x_k+1)+b-y_k.$$

Therefore,

$$d_{lower} - d_{upper} = 2m(x_k+1)+2b-2y_k-1$$

Three cases can arise depending on the m value.

Case 1: $0<m<1$

If $P_k<0$,

next point $=(x_k+1, y_k)$

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

else,

next point $=(x_k+1, y_k+1)$

$P_{k+1} = P_k + 2dy - 2dx$

Case 2: $-1<m<0$

Counterpart the end points (x_1', y_1') and (x_2', y_2') to $(x_1', -y_1')$ and $(x_2', -y_2')$, then calculate as case 1. Plot the next point converting y value multiplying by -1.

Case 3: $m>1$

The calculation simply swaps the position of x & y.

Used Resources:

1. Vs Code.

Used programming language & dependencies:

1. C++
2. GLUT OpenGL utility toolkit.

Procedure:

1. A C++ file was opened in VS code.
2. The algorithm was implemented carefully.
3. The code was compiled and run.
4. Results were observed.

Conclusion:

The Bresenham line drawing algorithm is a relatively simple algorithm. I faced no difficulty while implementing it. At first the line was not very visible. But after changing the point size to 4 using glPointSize() function, the line was clearly visible.

Title:

Implementing Midpoint circle algorithm

Objective:

1. To understand the algorithm.
2. To implement the algorithm using GLUT

Theory:

In midpoint circle algorithm, we use 8-way symmetry to calculate the perimeter points of the circle in the first octant and plot them along with their mirror points in the other octants. This greatly reduces the computational cost.

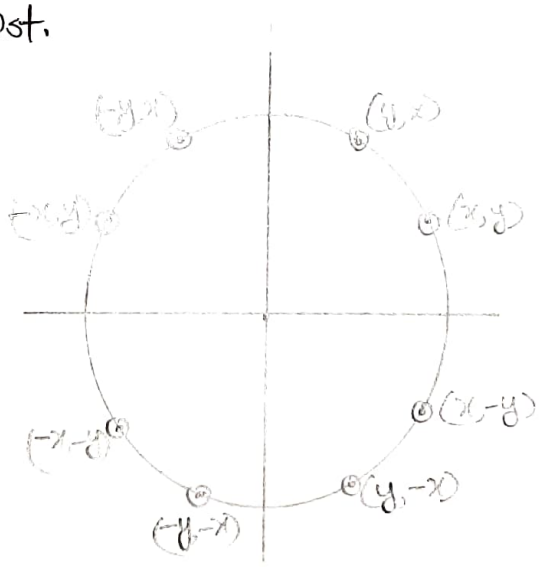


fig. All possible pixels in 8-way symmetry

Lets consider any given point (x,y). The next pixel to be plotted is either (x+1, y+1) or (x+1, y-1). This can be achieved by finding the midpoint P of the two possible pixels. If P lies inside the circle perimeter, we plot (x+1, y). Else we plot (x+1, y-1) pixel.

Algorithm:

1. Input radius r and center (x_c, y_c) then obtain the first point of the circle centered on the origin as $(x_0, y_0) \rightarrow (0, r)$
2. Calculate initial value of decision parameter,

$$P_0 = \frac{5}{4} - r$$

3. If $P_k < 0$,

Next point $= (x_{k+1}, y_k)$

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

Else

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

4. Determine the symmetry points of other 7 octants
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the value.
6. Repeat step 3 and 5 until $x >= y$
7. End.

Used Resources:

1. VS Code

Used programming language & dependencies:

1. C++
2. GLUT OpenGL Utility toolkit.

Procedure:

1. A C++ file was opened in VS code
2. The midpoint circle algorithm was implemented.
3. The code was compiled and run.
4. Result was observed.

Conclusion:

At first the circle was not showing. Upon debugging, I found a bug in the gluOrtho2D() function parameter. After giving the correct parameter, the circle was showing but it was missing a portion in 3rd quadrant. Upon more debugging, I found I ~~did~~ wrote a same line twice, without changing the symmetry point. After fixing it, the circle was drawn perfectly.

Title:

Implementing Recursively defined drawings, C curve, Koch curve and Sierpinski Gasket algorithm.

Objective:

1. To implement to algorithms using OpenGL
2. To understand the principles of the algorithm.

Theory:

C-curve algorithm:

In mathematics, the Lagc curve is a self similare factore that ceas first described. The construction of a C-curve starts with a straight line. An isosceles triangle with angles of 45°, 90° and 45° is built using this line as its ~~to~~ hypotenuse. The original line is then replaced by the two other side of the triangle. At the second stage, the two new line each form the bone for another right signed isosceles triangle and are replaced by the other two 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 triangle built on it. After n stages, the curve consists of 2n line segments each of which is smaller than original line by a factore or 2n/2.

Koch Curve algorithm:

The koch curve is a mathematical curve and one of the easiest fractal curves to have been described. The koch snowflake can be constructed by starting with an equilateral triangle, then recursively altering each line segment as follows:-

(i) Divide the line segment into 3 segments of equal length.

(ii) Draw an equilateral triangle that has the middle segment from step (i) as its base and points onwards.

iii) Remove the line segment which is the base of the triangle from step (ii).

After one iteration of the process, the resulting shape is the outline of a hexagram. The length of the curve between any two points is 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 equibilateral triangle form a simple closed curve that forms the boundary of the koch snowflake.

Sierpinski Gasket Algorithm:

The S-gasket is defined as follows:-

It takes a solid equilateral triangle, divides it into four congruent equilateral ~~trian~~ triangle then removes

the middle triangle. Then it does the same for the each of the remaining triangles and so on. It is a recursive algorithm. The pseudocode is given below:-

```
S-gasket (x1, y1, x2, y2, x3, y3, n) {
float x12, y12, x13, y13, x23, y23;
    if (n>0)
      {
        x12 = (x1+x2)/2
        x13 = (x1+x3)/2
        x23 = (x2+x3)/2
        y12 = (y1+y2)/2
        y13 = (y1+y3)/2
        y23 = (y2+y3)/2
        S-gasket (x1, y1, x12, y12, x13, y13, n-1)
        S-gasket (x12, y12, x2, y2, x23, y23, n-1)
        S-gasket (x13, y13, x23, y23, x3, y3, n-1)
      }
    else
      triangle(x1, y1, x2, y2, x3, y3)
}
```

Used Resources:

1. VS Code

Used Programming languages and dependencies:

1. C++
2. GLUT OpenGL utility toolkit.

Algorithms

Procedures:

1. Three seperate C++ files were opened in VScode
2. All three algorithms were implemented in their corresponding C++ files.
3. The codes were compiled and run
4. Result was observed.

Conclusion:

The C-curve and S-gasket algorithm was comparatively easier to implement. But I ran into some problems with the Koch curve. The line segments were being displayed broken. After a lot of debugging I still could not manage to find the issue and the lab-time was over. Later I found the issue and fixed it.

Title:

Implementing Boundary fill Algorithm

Objectives:

i) To implement Boundary fill algorithm using OpenGL

ii) To understand the mechanism of the algorithm.

Theory:

Boundary fill algorithm starts at pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary. It takes an interior point (x, y), a fill color and a boundary color as input. The algorithm starts by checking the color of (x, y). If it's color is not equal to fill color, and the boundary color, Then it is painted with fill color. It can be implemented by 4-connected / 8-cond' pixels.

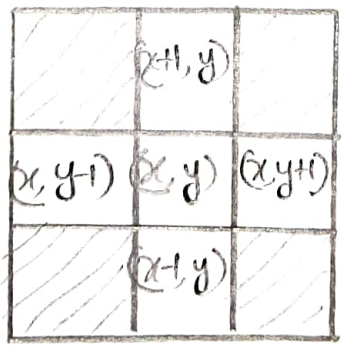


fig. 4 connected pixels.

In 4 connected approach, 4 connected pixels are used as shown in the figure.

We are putting the pixels to the above, below, right and left of the current pixel and the process continues until the boundary color is found with different color.

Algorithm:

1. Initialize seed point, fill color & boundary color
2. Define boundary values of polygon
3. Check if the current seed point is of default color. Then repeat step 4 and 5 until the boundary pixel is reached.
4. ~~The~~ Change the default color with fill-color at the seed point.
5. Recursively follow the procedure for the 4 neighbor points.
6. Exit.

Used Resources:

1. VS Code

Used programming language & dependencies:

1. C++
2. GLUT OpenGL utility toolkit.

Procedure:

1. A new file was opened in VS code.
2. The boundary fill algorithm was implemented
3. The code was compiled and run.
4. Result was observed.

Conclusion:

The boundary fill algorithm is a very straightforward method. The recursive function was written carefully. But the boundary was being filled so fast I could not see how the shape (A circle) was being filled. So I used a 10 millisecond delay function upon each recursion to visualize properly.

Title:

Implementing flood fill algorithm

Objective:

1. To implement the flood fill algorithm using OpenGL
2. To understand the principle of the algorithm.

Theory:

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 multidimensional array. This algorithm looks for all nodes in the array that are connected to the start node by a path of the target color and changes them to replacement color.

Algorithm:

Flood Fill (node, target color, replacement color)

1. If target color = replacement color, return
2. If node color != target color, return
3. Set node color = replacement color.
4. Perform flood fill according to 8-connected neighbors

(x-1, y+1)	(x, y+1)	(x-1, y+1)
(x-1, y)	(x, y)	(x+1, y)
(x-1, y-1)	(x, y-1)	(x+1, y+1)

5. Exit.

Used Resources:

1. VS Code

Used programming language & dependencies:

1. C++
2. GLUT OpenGL utility toolkit.

Procedure:

1. A C++ file was opened in VS code
2. The algorithm was implemented
3. The code was compiled and run
4. Result was observed.

Conclusion:

The algorithm was easy and straightforward to implement. I ran into no problem while compiling the code. The result was satisfactory. I used a circle as a shape to implement this algorithm.

Title:

Implementing geometric transformation operations (Translation, Rotation & Scaling)

Objectives:

i) To implement translation operation using OpenGL
ii) To implement rotation operation using OpenGL
iii) To implement scaling operation using OpenGL

Theory:

Geometric transformation is the process of the simulated spatial manipulation of objects in space. It is necessary because of several objects, each of which independently defined in its own coordinate system, need to be properly positioned into a common scene in a master coordinate system.

2D Translation:

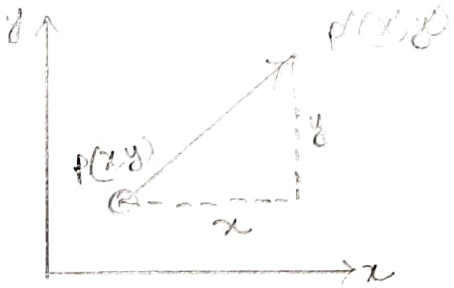
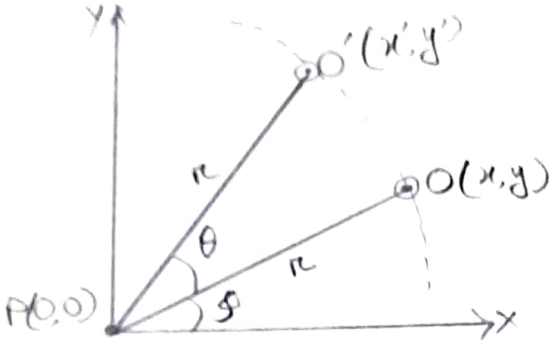


fig. Translation

To translate a 2D position, we add translation distances t_x and t_y to the original coordinates (x, y) to obtain the new coordinate position,

$$x' = x + t_x$$
$$y' = y + t_y$$

2D Rotations



2D rotation of an object is obtained by representing the object along circular path in xy plane. Parameters for 2D rotation are:-

i) Rotation angle θ

ii) Pivot point (x,y)

If the pivot point is origin (0,0), then new coordinates are:

$$x' = r\cos(\theta+\varphi)$$
$$= r\cos\varphi\cos\theta - r\sin\varphi\sin\theta$$
$$= x\cos\theta - y\sin\theta \quad [\because r\cos\varphi = x,\ r\sin\varphi = y]$$

$$y' = r\sin(\varphi+\theta)$$
$$= r\cos\varphi\sin\theta + r\sin\varphi\cos\theta$$
$$= x\sin\theta + y\cos\theta$$

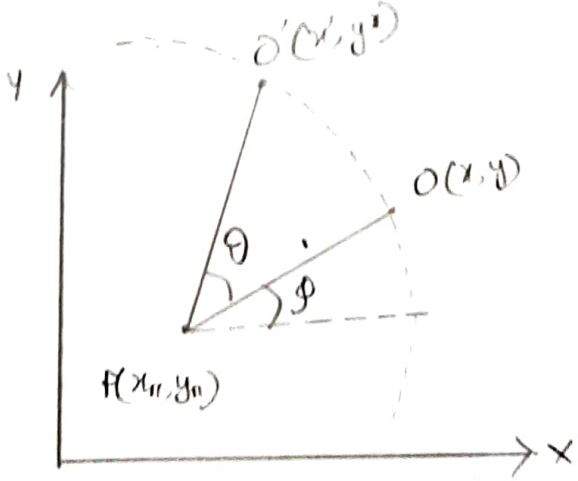


Fig. 2D rotation with arbitrary pivot point.

To rotate a point about an arbitrary pivot, the new coordinates will be:

$$x' = (x - x_r)\cos\theta - (y - y_r)\sin\theta + x_r$$
$$y' = (x - x_r)\sin\theta + (y - y_r)\cos\theta + y_r$$

2D Scaling:

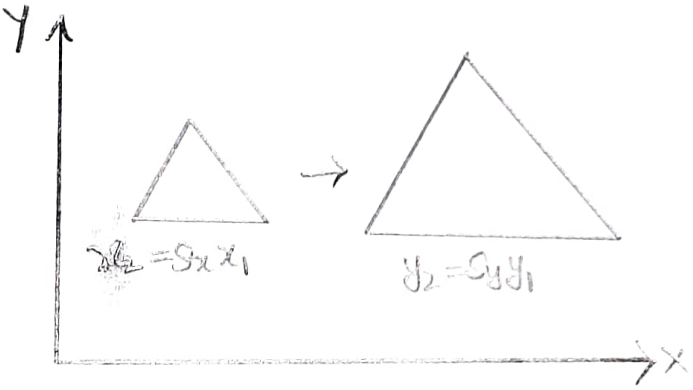


Fig. 2D scaling.

A simple 2D scaling operation is performed by multiplying object position (x, y) by scaling factors S_x and S_y. to produce the transformed coordinates (x', y'). The scaling will be.

$$x' = x \cdot S_x$$
$$y' = y \cdot S_y$$

Any positive values can be assigned to the scaling factors, where values less than 1 reduce the size of object and values greater than 1 enlarges the object.

Used Resources:

1. VS Code

Used programming language and dependencies:

1. C++
2. GLUT @ OpenGL utility toolkit.

Procedure:

1. A C++ file was opened in VS code.
2. The code for translation, rotation and scaling was implemented using the mathematical equations as well as the built in function.
3. The script was compiled and run.
4. Result was observed.

Conclusion:

I did not face any difficulty when implementing the codes for geometric transformation using the equations. But when I tried to implement it using the built in function, I was always getting a compile time error. It turned out the built in functions for translation, rotation and scaling work for 3D transformation as well, hence there is a parameter for a 3rd coordinate. After reviewing the glut API documentation, I put 0 in that parameter and the code worked perfectly.