Computer Graphic Algorithms

Introduction

Computer graphics is a discipline of computer science which focuses on the creation and manipulation of images using a computer. It has a wide range of applications such as animation creation, 3D simulations, information display and user interfaces. To draw images on the screen computers, utilize application of core fundamentals from several disciplines including physics, mathematics and computer algorithms. This document will discuss several algorithms used in computer graphics, particularly line generation algorithms and polygon filling algorithms with a focus on the flood filling algorithm.

Polygon Filling Algorithms

A polygon is a closed figure represented by a collection of more than 2 line segments connected end to end (Krat Keytal). It has a boundary which is made of line segments called edges. Vertices are the endpoints of the edges. In computer graphics, a polygon is an ordered list of the vertices. Examples of polygons are illustrated in the figures below.

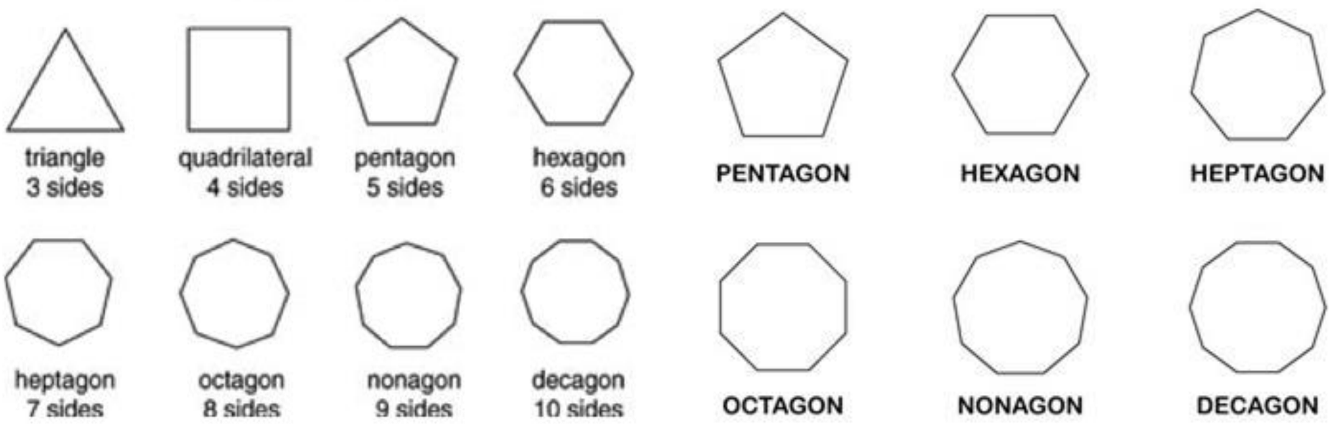

Figure 1 Regular Polygons

To add color to a polygon there is a need to fill the polygon with the desired color. Polygon filling is a process whereby every pixel within the region of the polygon is colored. To achieve this, pixels falling on the borders of the polygon and pixels inside the polygon must be determined. There are many techniques used to achieve this including: Scan line algorithm, Flood filling algorithm and the Boundary fill algorithm. Although Our interest is on the flood filling algorithm, the next section starts with a high level review of the 2 other algorithm and finishes with an in-depth review on the flood filling algorithm.

Scan Line Algorithm

This algorithm is also referred to as an image space algorithm that processes one line at a time rather than a pixel at a time. It maintains a record of polygon edge list and active edges. The edge list also known as the edge table keeps coordinates of 2 endpoints. This algorithm operates by intersecting the scanline with the edges of the polygon to form intersections and these are the active edges (AEL). The algorithm then colors the polygon by filling it with color between pairs of the intersections. It achieves this in 4 steps as outline below:

1. Determine the Ymin and Ymax of the given polygon
2. Intersect the scanline with each edge of the polygon from Ymin to Ymax and name each intersection point as depicted in the illustration
3. Sort the intersection points in increasing order of the x coordinate. For instance, in the illustration a proper order is p0, p1, p2 and p3.
4. Fill all pairs of coordinates inside the polygon and ignore alternate pairs.
5. Repeat steps 3-5 for each line.

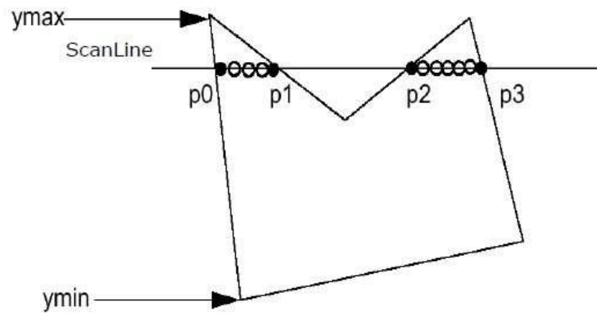

Figure 2 Scanline operation

It is an algorithm used for solid color filling in polygons.

Boundary Fill Algorithm

This algorithm, as its name suggests, fills a color in a closed area by starting at a point inside a region and paint the interior outward towards the boundary. It determines and selects a point inside the polygon. This point is known as the seed point.

For it to work several conditions must be met. The boundary color and the color we want to fill must be different otherwise it will fail. The boundary must have a single color.

In boundary fill algorithm you start from a point inside the region and fill the interior outward towards the boundary pixel by pixel. The algorithm checks the default color of the pixel before filling. If the color of the pixel is not a boundary color, it is filled with the fill color and the algorithm proceeds to another pixel, repeating the process until a boundary color is filled.

This algorithm achieves this functionality in 5 steps as outlined below:

1. Initialize the value of the seed point **seedx**, **seedy** and the 2 colors: default color (**dcolor**) and fill color(**fcolor**). Where seedx and seedy are coordinate positions of an interior pixel where the algorithm will start.
2. Define the boundary values of the polygon. The boundary pixel color or boundary color.

3. Check if the current pixel is of the default-color and if yes then repeat Step 4 and step 5 till the boundary pixels are reached.

```
If getpixel (x, y) = dcol then repeat step 4 and 5
```

4. Change the default color with the fill color at the current pixel

```
setPixel (seedx, seedy, fcol)
```

5. Recursively repeat steps 4 and 5 for the 4 neighboring pixels

```
FloodFill (seedx – 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy-1, fcol, dcol)
FloodFill (seedx – 1, seedy + 1, fcol, dcol)
```

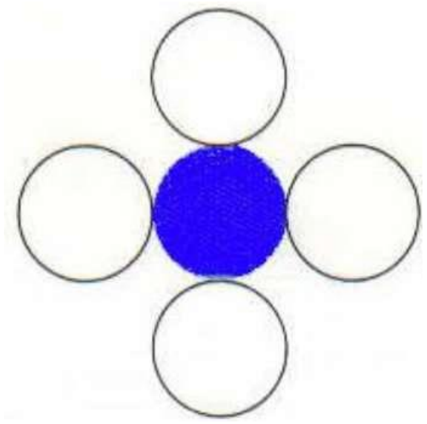

Figure 3 4-connected flood-fill implementation

A problem with this algorithm arises in the 4-connected implementation where by it fails to fill all the pixels in narrow spaces where there may be diagonally connected neighboring pixels. The solution is to use an 8-connected implementation which considers neighboring pixels that are diagonally connected. These implementations are discussed in the flood filling algorithm as they are common to both this boundary fill algorithm and flood filling algorithm.

Flood Filling Algorithm

In this section we will now take an in-depth review of the flood filling algorithm

Background

The flood fill algorithm is a modified form of the boundary fill algorithm. Is also known as **seed-fill** algorithm. The algorithm works by starting at a **seed(node)** point, a point known to be inside the polygon, setting the color of the neighboring pixels to the replacement color if they match a specific color known as the target color. It is used by most games such as Minesweeper, bucket-fill tools such as the bucket tool in Microsoft Windows Paint Program and in solving mazes. In the paint programs it is used to fill connected, similarly-colored areas with a different color.

It is a very useful algorithm in scenarios where there is no single color boundary for the polygon, that is where the boundary has multiple colors. In flood filling, instead of filling color till you encounter a specific boundary color you just fill the pixels with default color. In its simplest form the flood fill algorithm replaces the interior color of a polygon until no more pixels of the interior original color exists. Then the algorithm is said to be complete.

There are 2 methods of implementation for this algorithm:

- 4-connected pixels method and
- 8-connected pixels method

In 4-connected pixels method when performing the fill, 4 pixels adjacent to the current pixel are checked. These 4 pixels are namely north (top), south (bottom), west(left) and east(right). With respect to the current pixel coordinate of (x, y) these correspond to these pixel coordinates:

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

In 8-connected pixels when performing the fill, unlike in 4-connected pixels method, adjacent diagonal pixel positions are also checked thus N, S, W, E, NW, NE, SW, SE. Illustrative with respect to the current coordinate (x, y), the following pixel positions would be checked as well:

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

The flood-fill algorithm may be structured in so many ways but underneath they all use a stack or queue.

Algorithm Steps

Depending on the implementation method and data structure used steps may have a slight variance. In its simplest form, the flood-fill is implemented using a stack and a 4-connected pixels method.

Flood-fill (node, target-color, replacement-color):

1. If target-color is equal to replacement-color, return.

2. If the color of node is not equal to target-color, return.
3. Set the color of node to replacement-color.
4. Perform fill.

   Perform Flood-fill (one step to the south of node, target-color, replacement-color)

   Perform Flood-fill (one step to the north of node, target-color, replacement-color)

   Perform Flood-fill (one step to the west of node, target-color, replacement-color)

   Perform Flood-fill (one step to the east of node, target-color, replacement-color)
5. Return.

Other implementations that use the queue data structure and loops for memory optimizations are available.

A possible implementation of the flood-fill algorithm in OpenGL using the GLUT library is attached in appendix**, flood-fill-square.cpp.**

Strengths

The flood filling algorithm has several advantages over other algorithms. Outlined below are some of these strengths:

- **Simplicity**: Flood fill is a simple algorithm with no requirement to have a mathematical description of the region.
- **Reliable**: Unlike boundary fill algorithm it is more reliable and performant in cases where there no single color boundary for the polygon.
- **No leakages**: Unlike in boundary fill algorithm where leakages may occur, there are no leakages in flood-fill.
- **Adaptability**: There are many variants of the flood-fill algorithm that can be used depending on requirements.

Weaknesses

outlined below are some of the notable weaknesses associated with the flood-fill algorithm

- **Process inefficient**: It uses a getPixel() system call making it less efficient as system calls introduce latency
- **Memory inefficient:** Not efficient for large polygons as it requires large stack memory and hence introduces stack operation latency.
- It requires a seed point.
- Similar to boundary fill, flood filling has Limitations in narrow environments where it may miss regions of polygons because of narrow areas. This is a severe problem in 4-connected pixels.

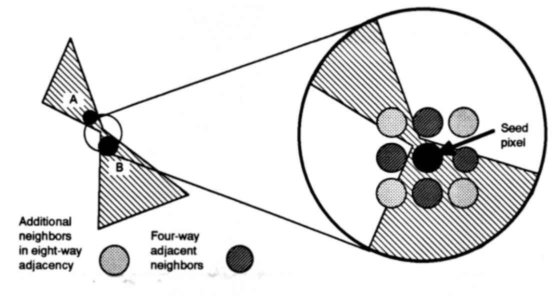
A
B
Seed pixel
Additional neighbors in eight-way adjacency
Four-way adjacent neighbors

Figure 4 narrow spaces

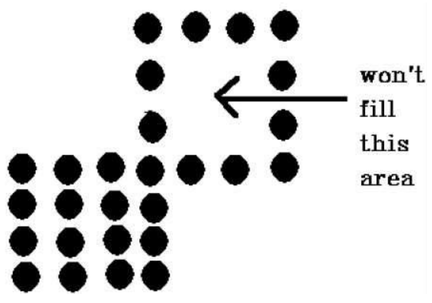

Figure 5 region missed by a 4-connected implementation

Solution of flood fill algorithm

The solution to the limitation of flood fill algorithm in narrow environments is to use an 8-connected implementation. This implementation ensures that all connected pixels are properly filled and not missed out.

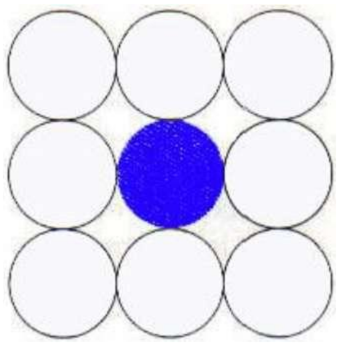

Figure 6 8-connected pixels implementation

Line Generation Algorithm

A line is one of the basic elements in computer graphics which connects 2 points. As such to draw a line it is a requirement that 2 points are available. In Computer graphics there are several algorithms for drawing lines. These include:

- Digital differential analyzer (DDA) Algorithm
- Bresenhams line algorithm and
- Mid-point algorithm

Discussed below is a review of these algorithms.

Digital differential analyzer algorithm

The digital differential analyzer algorithm is an algorithm for simple line generation. In a 2-D plane you can get a line by connecting 2 points (X_0, Y_0) and (X_1, Y_1). In a computer graphics screen the case is different as you cannot directly connect 2 coordinate points. There is a need to calculate intermediate coordinate points and put a pixel on each coordinate with the desired color. For instance, to generate a line between (1,1) and (4,4) we need the intermediate points (2,2) and (3,3). The DDA algorithm is a basic algorithm that helps compute the intermediate points for line generation.

Steps

1. Get the input of 2 end points (X_0, Y_0) and (X_1, Y_1)
2. Calculate the difference between the 2 end points

   dx = X_0-X_1

   dy = Y_0-Y_1
3. Depending on the result from step 2 – identification of the number of steps to put the pixel is done.

```
IF abs(dx) > abs(dy) then
        steps = abs(dx) // you need more steps in the x coordinate
ELSE
        steps = abs(dy)  // you need more steps in the y coordinate
```

4. Calculate the increment in x and y coordinates

   Xincrement = dx / (float) steps

   Yincremenet = dy / (float) steps
5. Put the pixel by successively incrementing x and y coordinates accordingly and draw the line. You iterate the number of steps.

```
FOR iter =0 to steps DO
        x = x + Xincrement
        y  = y + Yincrement
putpixel(Round(x), Round(y), color)
        ENDFOR
```

A possible implementation of the DDA line generation algorithm in OpenGL using the GLUT library is attached in appendix**, dda.cpp.**

Strengths

- **Simple**: It is the simplest algorithm for generating lines
- **Quick**: It is a faster method of calculating pixel positions than the direct use of the gradient equation (y = mx+b). It eliminates the multiplication in the equation by using raster characteristics to determine appropriate increments to be applied in the x and y directions to find pixel positions along that path.

Weaknesses

- It still Suffers inefficiencies since floating point arithmetic operations are time consuming in the round functions.
- End point accuracy is very poor since the algorithm is orient dependent.

Bresenhams line algorithm

This is an incremental scan conversion algorithm. Naively a mathematical line between 2 points can be drawn by computing the gradient and moving along the x-axis plotting points by computing the formula mx+c. The value has to be rounded up for it to work. This process is slow and this is where Bresenhams algorithm comes into play. It avoids floating point multiplication and addition in the computation of **mx+c** and rounding of **mx+c** in each step.

The algorithm works by moving across the X axis in unit intervals and at each step selecting the appropriate Y coordinate between 2 coordinates (X_k+ 1, Y_k) and (X_k+ 1, Y_k+ 1). For instance in the illustration below, from coordinate position (2,3) you need to chose between points (3,3) and (3,4).

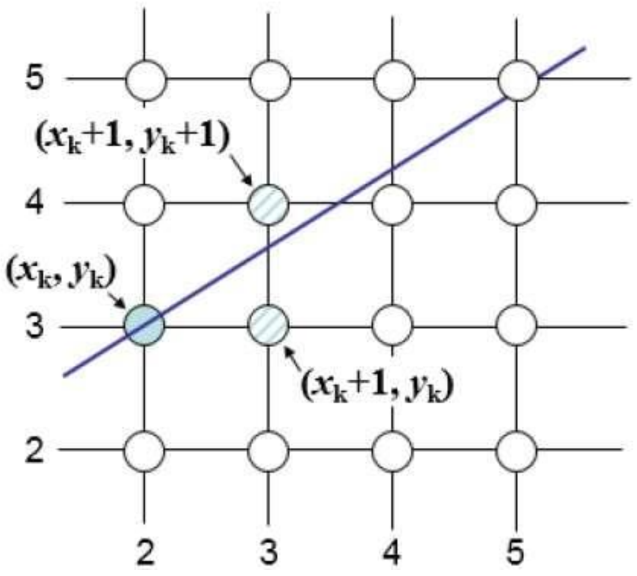

Figure 7 Bresenham's LGA illustration

The preferred point is the point closer to the original line and in this case this is (3,4). The mathematical line is separated into 2 vertical sections d_{upper} and d_{lower}. Taking a sample point of X_k+1 illustratively this is

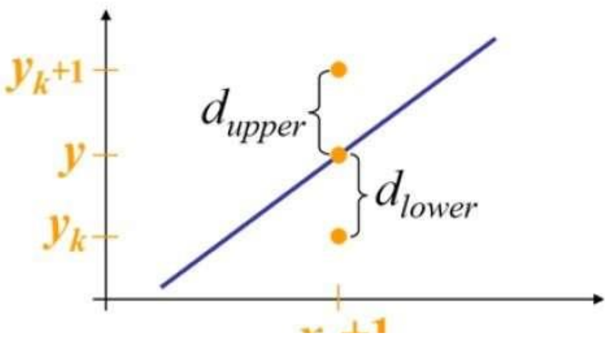

Figure 8 vertical sections

The y coordinate at the sample point given by the expression $Y= m(K_x+1)$ is negative so d_{upper} and d_{lower} are given by the following mathematical formulas.

$\mathbf{d_{upper}= Y_{k+1}-Y = m(K_x+1)+b-Y_k}$

$\mathbf{d_{lower} = Y-Y_k = m(K_x+1)+b-Y_k}$

The 2 values are used to decide which pixel is closer to the mathematical line by calculating the difference between the 2 pixel positions. The difference is given by the formula:

$\mathbf{d_{lower}-d_{upper} = 2m(K_x+1)-2Y_k+2b-1}$

Considering that **m** is dy/dx and making a substitution it evaluates to

$$\begin{aligned} dx(d_{lower} - d_{upper}) &= dx(2\frac{dy}{dx}(x_k + 1) - 2y_k + 2b - 1) \\ &= 2dy.\, x_k - 2dx.\, y_k + 2dy + 2dx(2b - 1) \\ &= 2dy.\, x_k - 2dx.\, y_k + C \end{aligned}$$

So for each step along the line, the decision parameter P_k is given by the difference between d_{lower} and d_{upper} whose evaluation is $2dy.x_k-2dx.y_k+C$. The first decision parameter is given by the denotation

$P_0= 2dy-dx$

The mathematical sign of P_k.

The decision parameter is used to keep track and control the slope error from previous increments to y.

Steps

Considering all the concepts from the above discussion on Bresenhman's line algorithm, for $m < 1$, these are the general steps of the algorithm

1. Input the 2 end points, keeping the left endpoint in.
2. Plot the left endpoint as X_0, Y_0
3. Calculate the constant values of dx, dy, 2dy and (2dy-2dx) to obtain the value of the first decision parameter P_0.
4. At each step along the line given by X_k, starting at $k = 0$, determine the point to plot by performing the following test

```
IF Pk < 0 THEN
    point = $(X_k+1,Y_k)$
    $P_{k+1} = P_k+2dy$
ELSE
    point = ($X_k$, $Y_k$+1)
    $P_{k+1}$ = $P_k$+2dy-2dx
ENDIF
```

5. Repeat the previous step (4), dx-1 times

For m > 1, y is incremented each time whilst the decision parameter is used to determine whether x should be incremented or not. A significant difference is that the x and y get interchanged in the equation.

A possible implementation of the Bresenhman's line generation algorithm in OpenGL using the GLUT library is attached in appendix**, bresenham.cpp.**

Strength

- **Speed**: It is a fast incremental algorithm that uses only integer calculations such as addition, subtraction and bit shifting.
- **Efficient**: It is a highly efficient algorithm as compared to DDA and it produces a mathematically correct line using integers operations only.

Weaknesses

- **Limitation**: It can only be used for simple or basic line drawings. Its does not perform well in complex lines and suffers aliasing problems.
- **Poor performance on modern CPU due to Loop branch mis-predictions**: Bresenhams line algorithm uses conditional branching in the loop which often results in mis-predictions in the CPU. This makes it less superior to DDA on modern computers since it uses fixed points with fewer instructions in the loop body.

Mid-point Algorithm

The mid-point algorithm is a modified form of the Bresenhams line algorithm by Pitteyway and Van Aken. It assumes that the slope of the line is greater than 0 but less than 1 denoted by $0 \leq k \leq 1$. Supposing that one approximate point P = (x, y) is already determined, for the next point we have to make a choice between 2 points namely E and N denoted by (x+1, y) and (x+1, y+1) respectively. Thus

E = (x+1, y) and NE = (x+1, y+1)

The ideal choice is the point which is closer to **k(x+1) +d,** E or N. To make the selection intersection points, denoted by Q are identified. The point closest to the intersection Q is selected as the next point and the algorithm restarts.

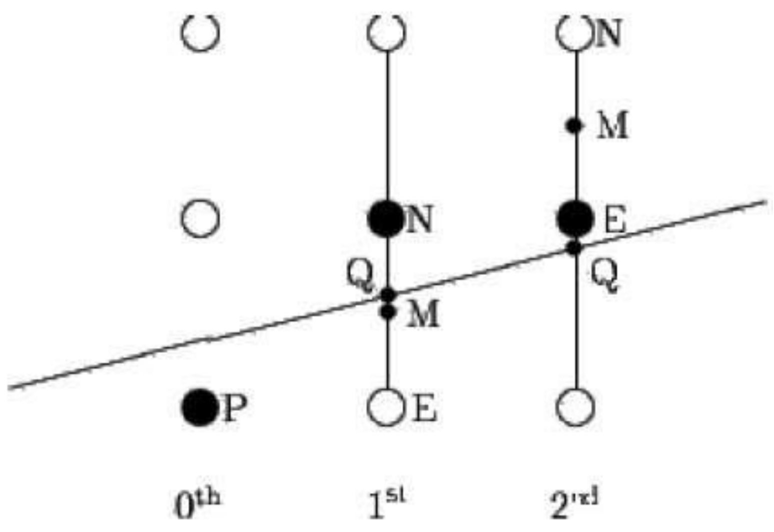

Figure 9 mid-point choices illustration

Steps

1. Calculate the mid-point m using x+1,y+0.5.
2. If the intersection point Q of the line with the vertical line connecting E and N are below m , then use E as the next point else use N as the next point.

An implicit equation is used to check this condition. The equation is:

F(x, y) = mx+b-y

Iteratively perform this

```
FOR positive m at any given x
IF y is on the line THEN
        F(x, y) = 0
ELSE IF y is above the line THEN
        F(x, y) < 0
ELSE IF y is below the line THEN
        F(x, y) > 0
```

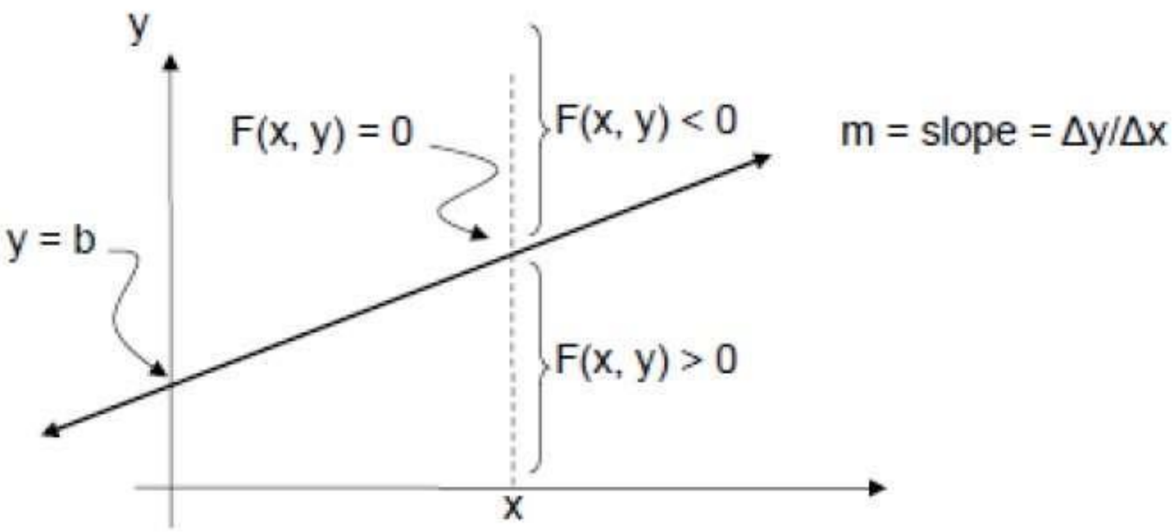

Figure 10 mid-point vertical seperation

A possible implementation of the mid-point line generation algorithm in OpenGL using the GLUT library is attached in appendix**, midpoint.cpp.**

Strengths

- **Error Elimination**: This algorithm does not use division and multiplication operations which potentially introduces errors (truncation errors).
- **Simple**: It is a simple algorithm that uses integer data only and simple mathematical operations
- **High Accuracy**: unlike basic incremental algorithms, which have challenges when $x_0 > x_1$ or when the gradient is greater than 1, midpoint is highly reliable.
- **Smoothness**: Results are of highly quality than other line drawing algorithms

Weaknesses

- **Limitation**: it is not an ideal algorithm for complex graphics and images
- No **significant** improvement is made by this algorithm as compared to the others. In addition, the accuracy still needs improvement.

Appendix

code sources attached in next page producing these outputs:

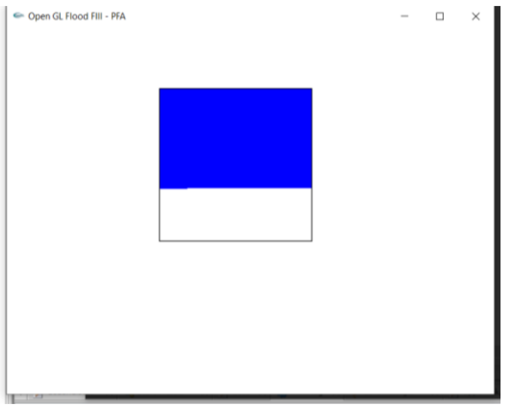

Figure 11 flood-fill output

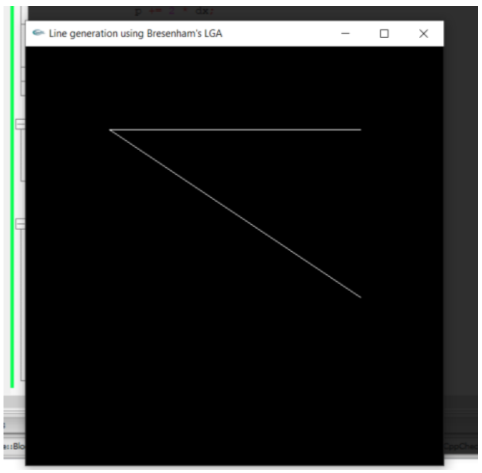

Figure 13 Bresenham's LGA output

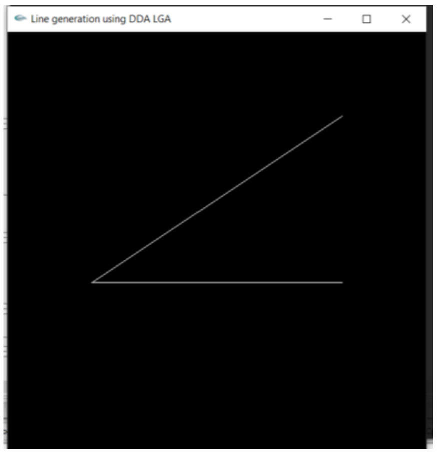

Figure 12 DDA LGA output

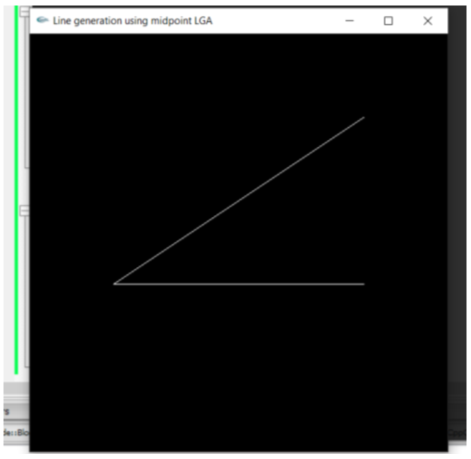

Figure 14 midpoint LGA output