
SHAPE RASTERISATION ALGORITHMS

SUBJECT

ALGORITHMS AND DATA STRUCTURES

BY

0xLeo (github.com/0xleo)

APRIL 29, 2021

DRAFT X.YY

MISSING: ...

Contents

1	Line rasterisation – Bresenham’s algorithm	2
1.1	Introduction	2
1.2	Bresenham’s assumptions	2
1.3	Deriving the algorithm	2
1.3.1	First attempt; a naive implementation	2
1.3.2	Bresenham’s line drawing algorithm idea	2
1.3.3	Bresenham’s line drawing at 1st octant; the derivation	3
1.3.4	Bresenham’s line drawing algorithm in octant 2	5
1.3.5	Bresenham’s line drawing algorithm in octants 3 and 4	6
1.4	Bresenham’s line drawing generalisation and implementation	7
1.5	Summary – pros and cons	7
2	Triangle fill algorithms	8
2.1	Line sweep triangle fill	8
2.2	Triangle fill by interior test	9
2.2.1	Mathematical background	9
2.2.2	Stating the interior test	11
2.2.3	The algorithm	12
3	Circle rasterisation	13
A	Appendices	15
A.1	Bresenham’s straight line drawing algorithm – pseudocode	15
A.2	Bresenham’s line drawing source code in C	17

1 Line rasterisation – Bresenham’s algorithm

1.1 Introduction

Bresenham’s line drawing algorithm was proposed in 1962. It takes as input two points and draws a line between in a discrete 2D grid. It decides either to draw or not to draw a pixel by traversing them in a certain way.

1.2 Bresenham’s assumptions

1. All pixels are sampled in a discrete 2D lattice.
2. The algorithm does not draw any colours – it simple decides whether to draw a pixel or not.
3. The line generated does not contain any holes. All line pixels must be 8-connected and for each column (x) there must be only one corresponding row (y).

To understand its advantages, we’ll try to derive the algorithm.

1.3 Deriving the algorithm

1.3.1 First attempt; a naive implementation

Given two points (x_1, y_1) , (x_2, y_2) , a naive first implementation is to iterate over all x ’s and find their y ’s as follows:

$$y = \text{round}(m \cdot x + b), \quad m = \frac{y_2 - y_1}{x_2 - x_1} \quad (1.1)$$

Translated to C:

```
#include <math.h>

/* Draw a line in a naive way assuming x1 < x2 */
void gfx_naive_line(int x1, int y1, int x2, int y2)
{
    // y = mx + b
    float m = (y2 - y1)/(x2 - x1);
    float b = y1 - m*x1;
    int x;
    int y;
    for (x = x1; x < x2; x++)
    {
        y = round(m*x + b);
        gfx_point(x,y);
    }
}
```

The drawback of this approach is that for each pixel it uses 2 floating point operations, plus rounding;

- Multiplication $m \cdot x$
- Addition of $m \cdot x$ with b

The cost of these operations adds up when 100s of pixels are drawn every time. Floating point operations are relatively expensive for CPUs and replacing them with integer arithmetic is a significant optimisation. Bresenham’s algorithm fully relies on integer operations.

1.3.2 Bresenham’s line drawing algorithm idea

For now, we’ll only be implementing the algorithm in the first octant (0° to 45° with x axis), i.e. assume that for the slope of the line $0 \leq m \leq 1$. We’ll later exploit the circular symmetry to derive the remaining 7 octants given the first. To reiterate, the main constraint imposed is

$$0 \leq m \leq 1, \quad m = \frac{y_2 - y_1}{x_2 - x_1} \quad (1.2)$$

, i.e. $x_2 - x_1 \geq y_2 - y_1$.

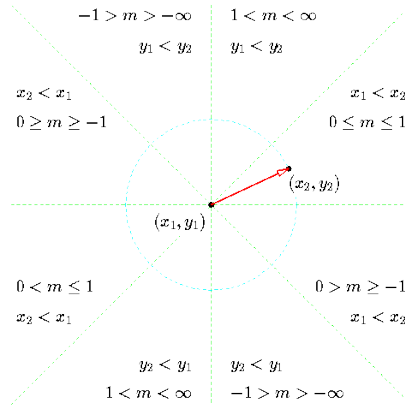


Fig. 1. The 8 octants and their slopes [1].

Let's say we have plotted a pixel (x, y) of the rasterised line. Because of the constraint in Eq. (1.2), the next pixel can be either East $(x+1, y)$ or North-East $(x+1, y+1)$. When the line is drawn in 2D, for each step from x to $x+1$, we have to find whether y or $y+1$ is closest to the y (floating) of the line. To do that, we increment y by the slope m (def'n of slope) and have to determine whether $y+m$ is above or below the midway $y+0.5$ between y , $y+1$ (Fig. 3).

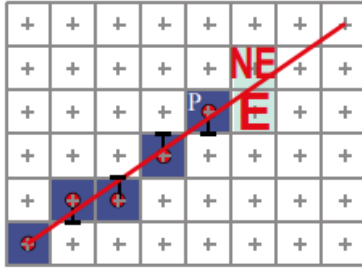


Fig. 2. At every update of pixel (x, y) , we choose between the E and NE neighbour [2].

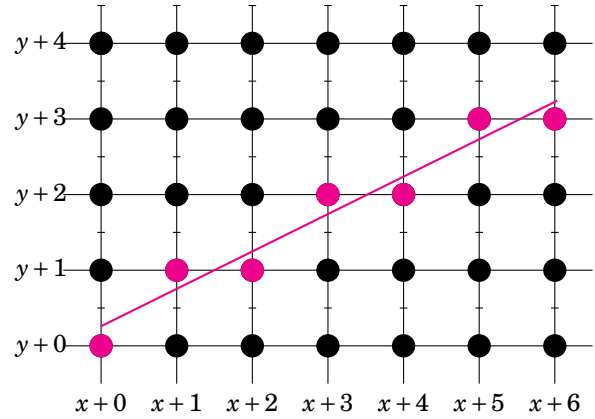


Fig. 3. The nodes represent pixel centres and the segments the midway y between neighbouring pixels. Nodes in magenta represent where the line will be drawn in the 2D discrete space.

1.3.3 Bresenham's line drawing at 1st octant; the derivation

Because we plot the original line in a discrete grid given a resolution, it will almost never cross a discrete point. Therefore it will always be at some error ϵ above or below the nearest discrete y . For the error [1],

$$-0.5 \leq \epsilon < 0.5 \quad (1.3)$$

The y_{actual} ordinate of the line is then given by $y_{actual} = y + \epsilon$. In moving from x to $x+1$ we increase the value of the true (mathematical) y -ordinate by an amount equal to the slope m (Fig. 4).

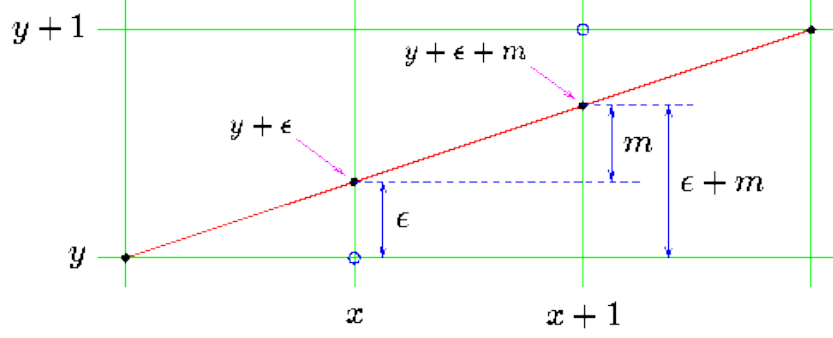


Fig. 4. The error at each pixel during the update [1].

From the plot in Fig. 4, it is clear after the transition $x \rightarrow x + 1$, if

$$\begin{aligned} y + \epsilon + m &< y + 0.5 \Rightarrow \\ \epsilon + m &< 0.5 \end{aligned}$$

, then we move East $(x + 1, y)$ to represent the line. Else we move North-East $(x + 1, y + 1)$. We make this decision to minimise the total error between what gets drawn on the display and the actual values.

However, after $x \rightarrow x + 1$ the error gets updated too from ϵ to ϵ_{new} . We know that the error is the distance of the mathematical line to the nearest y-ordinate of the grid, i.e. either y or $y + 1$. In case $(x + 1, y)$, the new error is given by [1] (Fig. 4)

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - y = \epsilon + m \quad (1.4)$$

Else, if $(x + 1, y + 1)$ was chosen

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - (y + 1) = \epsilon + m - 1 \quad (1.5)$$

Therefore a first implementation of the line drawing algorithm so far is below. Note that it still uses floating point which must be eliminated. Note also that for the algorithm to be consistent with the idea developed thus far, it is assumed that (x_1, y_1) is closer to the origin than (x_2, y_2) .

Algorithm 1 Line drawing with FP operations.

```

1: procedure LINE-DRAWING-FP( $x_1, y_1, x_2, y_2$ )
2:    $m \leftarrow \frac{y_2 - y_1}{x_2 - x_1}$ 
3:    $\epsilon \leftarrow 0, y \leftarrow y_1$  ▷  $\epsilon, y$  are all we keep track of.
4:   for  $x = x_1, \dots, x_2$  do
5:     DrawPixel( $x, y$ )
6:     if  $\epsilon + m < 0.5$  then
7:        $\epsilon \leftarrow \epsilon + m$  ▷ Move E; don't change  $y$ 
8:     else
9:        $\epsilon \leftarrow \epsilon + m - 1$  ▷ Move NE
10:     $y \leftarrow y + 1$ 

```

To optimise the algorithm, we must convert the following to integer operations

$$\epsilon + m < 0.5 \quad (1)$$

$$\epsilon \leftarrow \epsilon + m \quad (2)$$

$$\epsilon \leftarrow \epsilon + m - 1 \quad (3)$$

Plugging in $m = \Delta x / \Delta y = (y_2 - y_1) / (x_2 - x_1)$, Eq. (1) becomes

$$2 \underbrace{\epsilon \Delta x}_{\epsilon'} + 2 \Delta y < \Delta x \quad (1')$$

Bresenham
relies on
integer
operations.

Eq. (2) and (3) become respectively

$$\underbrace{\epsilon\Delta x}_{\epsilon'} \leftarrow \underbrace{\epsilon\Delta x}_{\epsilon'} + \Delta y \quad (2')$$

$$\underbrace{\epsilon\Delta x}_{\epsilon'} \leftarrow \underbrace{\epsilon\Delta x}_{\epsilon'} + \Delta y - \Delta x \quad (3')$$

The quantity $\epsilon\Delta x$ appears in all Eq. (1'), (2'), (3') therefore we let $\epsilon' := \epsilon\Delta x$. The algorithm we have arrived at is *Bresenham's for the 1st octant*. It is written in integer arithmetic as follows:

Algo stated assuming
 $0 \leq m \leq 1$
and $x_1 < x_2$.

Algorithm 2 Bresenham's line drawing – 1st octant.

```

1: procedure BRESENHAM-1ST-OCTANT( $x_1, y_1, x_2, y_2$ )
2:    $\Delta x \leftarrow x_2 - x_1$ 
3:    $\Delta y \leftarrow y_2 - y_1$ 
4:    $\epsilon' \leftarrow 0, y \leftarrow y_1$ 
5:   if  $0 \leq \frac{\Delta y}{\Delta x} < 1$  then
6:     for  $x = x_1, \dots, x_2$  do
7:       DrawPixel( $x, y$ )
8:       if  $2(\epsilon' + \Delta y) < \Delta x$  then
9:          $\epsilon' \leftarrow \epsilon' + \Delta y$ 
10:      else
11:         $\epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$ 
12:         $y \leftarrow y + 1$ 

```

$\triangleright \epsilon', y$ are all we keep track of.
 \triangleright Move E; don't change y
 \triangleright Move NE

This version is particularly efficient not only due to integer arithmetic but as multiplication by 2 can be implemented as left bit shifting. We can of course move the update $\epsilon' \leftarrow \epsilon' + \Delta y$ before the if-else block to end up with only one if for slightly more conciseness.

1.3.4 Bresenham's line drawing algorithm in octant 2

We now address the case of drawing a line with slope $1 \leq m < \infty$, i.e. one that spans at the 2nd octant (Fig. 1). Note that a line ($l1$): $y = mx + b$ with slope $0 \leq m < 1$ in the first octant is symmetric w.r.t to $y = x$ to the line ($l2$): $x = my + b \Leftrightarrow y = \frac{x}{m} - \frac{b}{m}$ (e.g. Fig. 5). If $(x_0, y_0) \in (l1)$ then $(y_0, x_0) \in (l2)$. Therefore to rasterise ($l2$) we can apply Alg. 2 on it modified by swapping x with y and Δx with Δy . Don't forget the ordinate condition for the 2nd octant, which is $y_1 < y_2$.

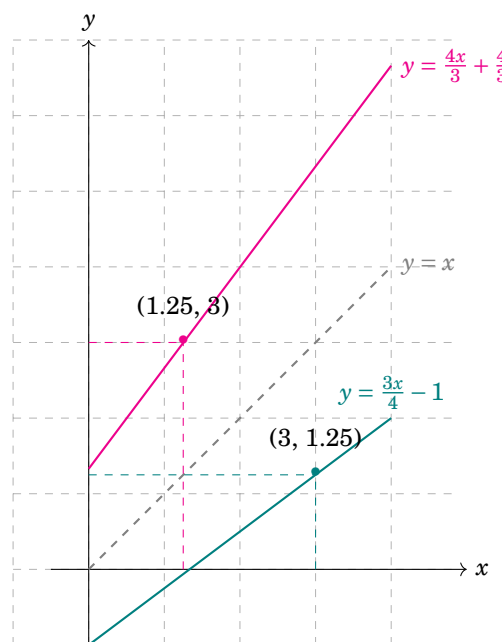


Fig. 5. Two lines in the first two octants symmetric about line $y = x$.

Algorithm 3 Bresenham's line drawing – 2nd octant.

```
1: procedure BRESENHAM-2ND-OCTANT( $x_1, y_1, x_2, y_2$ )
2:    $\Delta x \leftarrow x_2 - x_1$ 
3:    $\Delta y \leftarrow y_2 - y_1$ 
4:    $\epsilon' \leftarrow 0, x \leftarrow x_1$   $\triangleright \epsilon', y$  are all we keep track of.
5:   if  $1 \leq \frac{\Delta y}{\Delta x}$  and  $y_1 < y_2$  then
6:     for  $y = y_1, \dots, y_2$  do
7:       DrawPixel( $x, y$ )
8:       if  $2(\epsilon' + \Delta x) < \Delta y$  then
9:          $\epsilon' \leftarrow \epsilon' + \Delta x$ 
10:      else
11:         $\epsilon' \leftarrow \epsilon' + \Delta x - \Delta y$ 
12:         $x \leftarrow x + 1$ 
```

Octants 1 and 2 (quadrant 1) have been addressed. To complete the algorithm in the remaining 6 octants, observe that quadrant 2 (octants 3,4) is symmetric with quadrant 1 w.r.t the y axis. Quadrant 3 (octants 5, 6) is symmetric with 1 w.r.t x and y axes, and quadrant 4 is symmetric with 1 w.r.t the x axis.

1.3.5 Bresenham's line drawing algorithm in octants 3 and 4

Octants 3 and 4 are symmetric w.r.t the y axis to octants 2 and 1 respectively. Therefore to derive their line drawing we start with Alg. 3 and 2 respectively and substitute $(-x, y) \leftarrow (x, y)$, $\Delta x \leftarrow -\Delta x$. Therefore the line drawing for those octants is formulated as follows, renaming the error ϵ' to ϵ for simplicity.

Algorithm 4 Bresenham's line drawing – 2nd quadrant.

```
1: procedure BRESENHAM-2ND-QUADRANT( $x_1, y_1, x_2, y_2$ )
2:    $\Delta x \leftarrow x_2 - x_1$ 
3:    $\Delta y \leftarrow y_2 - y_1$ 
4:    $\epsilon \leftarrow 0, y \leftarrow y_1$ 
5:   if  $\frac{\Delta y}{\Delta x} < -1$  and  $y_1 < y_2$  then  $\triangleright$  This is the 3rd octant (2nd octant mirror w.r.t  $y$  axis)
6:     for  $y = y_1, \dots, y_2$  do
7:       DrawPixel( $x, y$ )
8:       if  $2(\epsilon - \Delta x) < \Delta y$  then
9:          $\epsilon \leftarrow \epsilon - \Delta x$ 
10:      else
11:         $\epsilon \leftarrow \epsilon - \Delta x - \Delta y$ 
12:         $x \leftarrow x - 1$ 
13:   else if  $0 \leq \frac{\Delta y}{\Delta x} < -1$  and  $x_2 < x_1$  then  $\triangleright$  4th octant (1st octant mirrored w.r.t  $y$  axis)
14:     for  $x = x_1, \dots, x_2$  do
15:       DrawPixel( $x, y$ )
16:       if  $2(\epsilon + \Delta y) < -\Delta x$  then
17:          $\epsilon \leftarrow \epsilon + \Delta y$ 
18:      else
19:         $\epsilon \leftarrow \epsilon + \Delta y + \Delta x$ 
20:         $y \leftarrow y + 1$ 
```

In the same way, given the algorithm for the first two octants, using the transform $(x, y) \leftarrow (-x, -y)$, $\Delta x \leftarrow -\Delta x$, $\Delta y \leftarrow -\Delta y$ we can derive octants 5 and 6. Finally, using $(x, y) \leftarrow (x, -y)$, $\Delta y \leftarrow -\Delta y$ we can derive octants 7 and 8. The x 's, y 's, Δx 's and Δy 's for each octant given the first are summarised in Fig. 6.

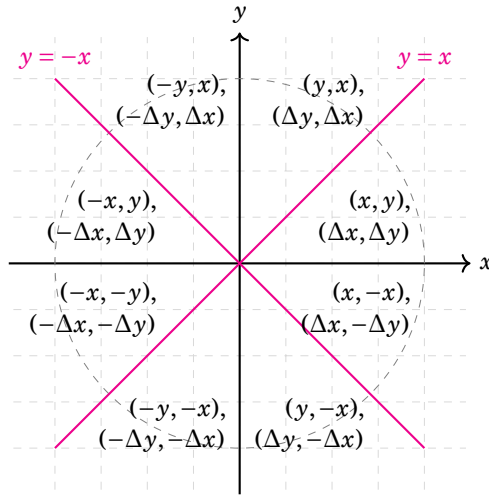


Fig. 6. Deriving Bresenham for the full circle given the 1st octant.

1.4 Bresenham's line drawing generalisation and implementation

The first step of the algorithm generalisation is to determine the octant of the line, which is done with the aid of the conditions in Fig. 1. Next, we start from the algorithm for quadrants 1 and 2 and transform the x 's, y 's, Δx 's and Δy 's given the symmetry. The pseudocode for the full algorithm is listed in ??.

To implement Bresenham and draw pixels in C, Prof D. Thain's "gfx" graphics library [3] was used. Method `gfx_line_bres` was added to implement the algorithm. To test it, each line was plotted against the library's `gfx_line` method and the lines overlapped for all 8 octants. The code for the full algorithm in C is found in A.2.

Finally, note that the algorithm runs in linear ($\mathcal{O}(n)$) time.

1.5 Summary – pros and cons

Bresenham's algorithm may be easy to implement and fast, but has a certain disadvantage. However it is still used by graphics cards and software libraries [4] thanks to its simplicity.

Pros:

- Simple to implement, can be efficiently implemented practically on any hardware!
- Fast – linear time.

Cons:

- Does not account for aliasing.

2 Triangle fill algorithms

There are various algorithms to draw a solid triangle. Here, we discuss three of them:

1. Line sweep (row-by-row fill).
2. Triangle interior test
3. Bresenham-based fill.

2.1 Line sweep triangle fill

Suppose we want to fill a triangle given points $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_3 = (x_3, y_3)$, $y_1 \geq y_2 \geq y_3$. Line sweep fill fills the interior points of the triangle row by row relying on the fact that the slope of each line P_1P_2 and P_2P_3 is constant.

At each iteration, it keeps track of three variables – the row number (y coordinate), the leftmost ordinate (column) of the line to draw (x_l), and the rightmost ordinate (column) of the line to draw (x_r). At each row sweep, we incrementally update x_l and x_r . Then, we can fill all pixels from x_l to x_r , forming a horizontal line (a.k.a. scanline). Finally, because the slope changes as we move from line P_1P_2 to P_2P_3 , we first draw the top flat triangle $\Delta(P_1P_2P'_2)$ and then the $\Delta(P'_2P_2P_3)$.

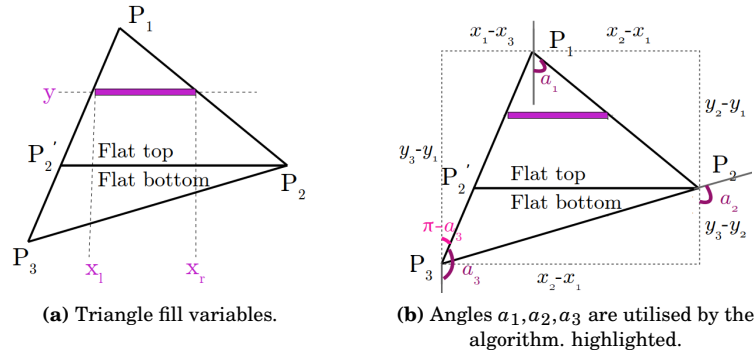


Fig. 7. Line sweep fill divides the original triangle in two flat ones. (a) The variables y, x_l, x_r considered at each iteration. (b) The angles utilised by the algorithm to update x_l, x_r

What's left to define is how x_l and x_r are updated at each iteration. Because x is a function of y (we iterated row-by-row), we increment the y of each line by the *inverse slope* (i.s.) of that line. The inverse slope has x as the adjacent side and y as the opposite, therefore $m_{inv} = \frac{\Delta x}{\Delta y}$. x_l is always incremented by the inverse slope of line $\overline{P_1P_3}$ and x_r is incremented by the i.s. of whatever line it belongs to; by $m_{inv,13}$ is if $y \leq y_2$, else by $m_{inv,23}$. The i.s. $m_{inv,13}$ of $\overline{P_1P_3}$ is the tangent of a_3 (Fig. 7):

$$m_{inv,13} = \tan a_3 = -\tan(\pi - a_3) = -\frac{x_1 - x_3}{y_3 - y_1} = \frac{x_3 - x_1}{y_3 - y_1}$$

$$\therefore m_{inv,13} = \frac{x_3 - x_1}{y_3 - y_1}$$

$$m_{inv,12} = \frac{x_2 - x_1}{y_2 - y_1}$$

$$m_{inv,23} = \frac{x_3 - x_2}{y_3 - y_2}$$

The formulas would end up the same if P_2 was left of P_3 . The final algorithm, which fills $\Delta(P_1P_2P'_2)$ first, followed by $\Delta(P'_2P_2P_3)$ is listed below.

Algorithm 5 Triangle fill by line sweep pseudocode.

```
1: procedure TRIANGLEFILLINESWEEP( $x_1, y_1, x_2, y_2, x_3, y_3$ ) ▷ Assuming  $y_1 < y_2 < y_3$ 
2:    $m_{inv,13} \leftarrow \frac{x_3 - x_1}{y_3 - y_1}$ 
3:    $m_{inv,12} \leftarrow \frac{x_2 - x_1}{y_2 - y_1}$ 
4:    $m_{inv,23} \leftarrow \frac{x_3 - x_2}{y_3 - y_2}$ 
5:    $x_l \leftarrow x_1$ 
6:    $x_r \leftarrow x_1$ 
7:   for  $y = y_1 \dots y_2$  do ▷ Flat top triangle
8:     for  $y = \text{int}(x_l) \dots \text{int}(x_r)$  do ▷ int denotes the casting from float to int
9:       drawPixel( $x, y$ )
10:     $x_l \leftarrow x_l + m_{inv,13}$ 
11:     $x_r \leftarrow x_r + m_{inv,12}$ 
12:   for  $y = y_2 \dots y_3$  do ▷ Flat bottom
13:     for  $y = \text{int}(x_l) \dots \text{int}(x_r)$  do
14:       drawPixel( $x, y$ )
15:     $x_l \leftarrow x_l + m_{inv,13}$ 
16:     $x_r \leftarrow x_r + m_{inv,23}$ 
```

This is super easy to implement and an implementation in C is found in my “gfx-v4” repository at <https://github.com/0xLeo/gfx-v4/blob/master/src/gfx/gfx.c#L395>.

2.2 Triangle fill by interior test

2.2.1 Mathematical background

Another method to fill the interior of a triangle is to find its bounding box, scan row-by-row all pixels in the box, and determine whether each pixel is in the interior of the triangle. The only tricky part about this algorithm is to derive an “interior test”.

Before delving in the algorithm or in its maths, we need the definition of perpendicular vector (a.k.a. perp) in 2D.

DEFINITION 2.1 (perp vector). Given a vector $\mathbf{a} = (a_x, a_y)$, its perp vector \mathbf{a}^\perp is defined as a vector with the same magnitude rotated by 90° ccw:

$$\mathbf{a}^\perp = (-a_y, a_x) \tag{2.1}$$

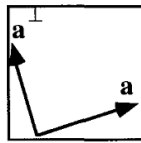


Fig. 8. Vector \mathbf{a} and its “perp” (\mathbf{a}) rotated by 90° ccw.

The perp vector comes in handy when we want to determine the relative orientation of two vectors, e.g. whether \mathbf{a} is cw or ccw from \mathbf{b} . But first, it’s important to clarify what is meant by cw and ccw. By saying that \mathbf{b} is cw of \mathbf{a} , it is implied that to rotate \mathbf{b} by the *inner* (smaller) angle until it’s aligned with \mathbf{a} , we move clockwise. Ccw rotation is defined in the same way. The figure below illustrates this.

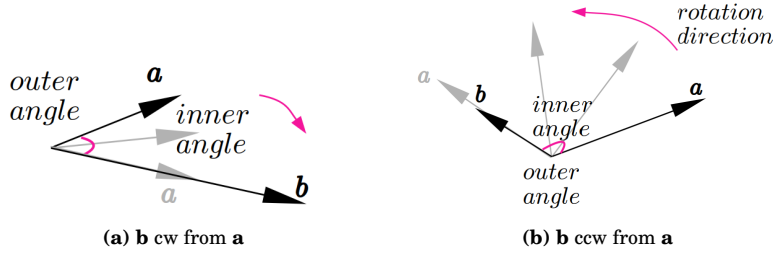


Fig. 9. cw and ccw terminology

Back to the perp vector, how is it able to tell us the relative orientation between **a** and **b**? Remember that \mathbf{b}^\perp is **b** rotated by 90° ccw. It turns out that if **b** is ccw from **a**, then $\mathbf{a}^\perp \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta) < 0$, since $\theta > 90^\circ$ (Fig. 10).

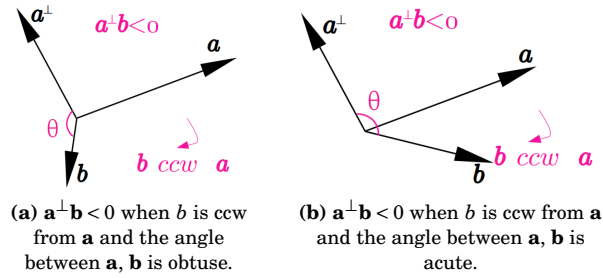


Fig. 10. $\mathbf{a}^\perp \cdot \mathbf{b} < 0$ when **b** is ccw from **a**.

Similarly, when **b** is cw from **a** then **a**, then $\mathbf{a}^\perp \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta) > 0$, since $\theta < 90^\circ$ (Fig. 10).

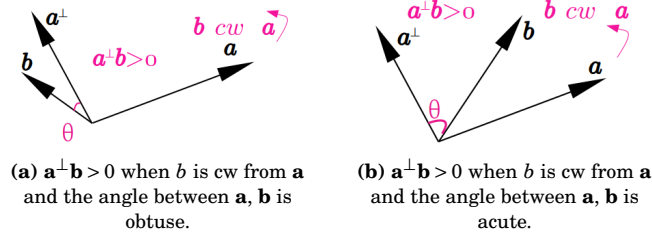


Fig. 11. $\mathbf{a}^\perp \cdot \mathbf{b} > 0$ when **b** is cw from **a**.

Therefore the so-called “perp dot product” tells us the relative orientation between **a** and **b**. To define it:

DEFINITION 2.2 (perp dot product). *The perp dot product between **a** and **b** is defined as:*

$$pdot(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\perp \cdot \mathbf{b} = a_x b_y - a_y b_x \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} \quad (2.2)$$

, where \mathbf{a}^\perp is **a** rotated by 90° , i.e. $\mathbf{a}^\perp = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{a}$.

To summarise, it has the following properties:

COROLLARY 2.1 (properties perp product).

$$\mathbf{a}^\perp \mathbf{b} > 0 \quad \text{if } \mathbf{b} \text{ ccw from } \mathbf{a} \quad (2.3)$$

$$\mathbf{a}^\perp \mathbf{b} < 0 \quad \text{if } \mathbf{b} \text{ cw from } \mathbf{a} \quad (2.4)$$

$$\mathbf{a}^\perp \mathbf{b} = 0 \quad \text{if } \mathbf{b} = \lambda \mathbf{a}, \quad \lambda \in \mathbb{R} \quad (2.5)$$

2.2.2 Stating the interior test

Using Cor. ??, we can formulate whether a 2D point P is inside or outside a triangle $\Delta(P_1P_2P_3)$. We can use the perp dot product of vectors $\overrightarrow{PP_1}$, $\overrightarrow{PP_2}$, and $\overrightarrow{PP_3}$ to deduce whether P is inside or outside of the triangle, given that we know the relative orientation of points P_1, P_2, P_3 , i.e. of vectors $\overrightarrow{OP_1}, \overrightarrow{OP_2}, \overrightarrow{OP_3}$, where O is the origin. To reiterate, We make two assumptions:

1. $y_1 \leq y_2 \leq y_3$
2. About the relative orientation of points P_1, P_2, P_3 ; they are either in cw or ccw.

The figures below show the vectors $\overrightarrow{PP_1}$, $\overrightarrow{PP_2}$, and $\overrightarrow{PP_3}$ for the ccw and cw cases.

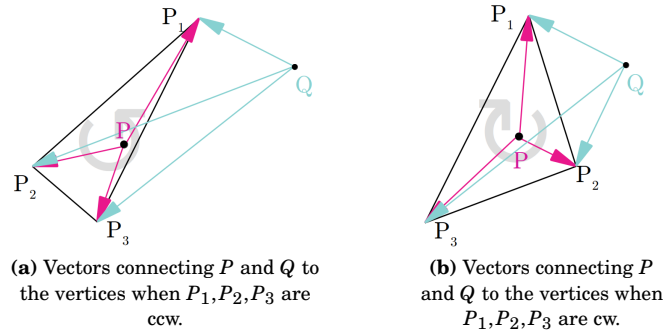


Fig. 12. For the interior test, we consider the vectors from a point to the vertices. The vectors for a typical interior point P and an exterior point Q are drawn.

From Eq. (2.4), Eq. (2.3) and Fig. 12 we can see that in case P_1, P_2, P_3 are ccw, then for an interior point (e.g. P):

$$\overrightarrow{PP_2} \text{ ccw from } \overrightarrow{PP_1} \Rightarrow \overrightarrow{PP_1}^\perp \overrightarrow{PP_2} > 0 \quad (1)$$

$$\overrightarrow{PP_3} \text{ ccw from } \overrightarrow{PP_2} \Rightarrow \overrightarrow{PP_2}^\perp \overrightarrow{PP_3} > 0 \quad (2)$$

$$\overrightarrow{PP_1} \text{ ccw from } \overrightarrow{PP_3} \Rightarrow \overrightarrow{PP_3}^\perp \overrightarrow{PP_1} > 0 \quad (3)$$

Similarly, from the same figure and equations for the cw case we obtain:

$$\overrightarrow{PP_2} \text{ cw from } \overrightarrow{PP_1} \Rightarrow \overrightarrow{PP_1}^\perp \overrightarrow{PP_2} < 0 \quad (4)$$

$$\overrightarrow{PP_3} \text{ cw from } \overrightarrow{PP_2} \Rightarrow \overrightarrow{PP_2}^\perp \overrightarrow{PP_3} < 0 \quad (5)$$

$$\overrightarrow{PP_1} \text{ cw from } \overrightarrow{PP_3} \Rightarrow \overrightarrow{PP_3}^\perp \overrightarrow{PP_1} < 0 \quad (6)$$

, where $\overrightarrow{PP_i} = \overrightarrow{OP_i} - \overrightarrow{OP}$. To summarise, an interior point satisfies either Eq. (1) and Eq. (2) and Eq. (3) or Eq. (4) and Eq. (5) and Eq. (6). Otherwise, it's exterior. We can therefore state the interior test in pseudocode as follows.

Algorithm 6 Testing whether a point $P(x, y)$ lies in the interior of a triangle $\Delta(P_1P_2P_3)$.

```

1: procedure PERDDOTPROD( $x_1, y_1, x_2, y_2$ )
2:   return  $x_1y_2 - y_1x_2$ 
3:
4: procedure ISINTERIOR( $x_1, y_1, x_2, y_2, x_3, y_3, x, y$ )
5:    $PP1_x \leftarrow x - x_1$  ▷ vector from  $P(x, y)$  to verices  $P_i(x_i, y_i)$ 
6:    $PP1_y \leftarrow y - y_1$ 
7:    $PP2_x \leftarrow x - x_2$ 
8:    $PP2_y \leftarrow y - y_2$ 
9:    $PP3_x \leftarrow x - x_3$ 
10:   $PP3_y \leftarrow y - y_3$ 
11:   $cw \leftarrow \text{PerdDotProd}(PP1_x, PP1_y, PP2_x, PP2_y) < 0 \text{ AND}$ 
     $\text{PerdDotProd}(PP2_x, PP2_y, PP3_x, PP3_y) < 0 \text{ AND}$ 
     $\text{PerdDotProd}(PP3_x, PP3_y, PP1_x, PP1_y) < 0$ 
12:   $ccw \leftarrow \text{PerdDotProd}(PP1_x, PP1_y, PP2_x, PP2_y) > 0 \text{ AND}$ 
     $\text{PerdDotProd}(PP2_x, PP2_y, PP3_x, PP3_y) > 0 \text{ AND}$ 
     $\text{PerdDotProd}(PP3_x, PP3_y, PP1_x, PP1_y) > 0$ 
13:  return  $cw \text{ OR } ccw$  ▷  $cw$  and  $ccw$  are boolean type

```

2.2.3 The algorithm

As usual, the aim of the algorithm is to fill a triangle given 3 points P_1, P_2, P_3 , assuming $y_1 \leq y_2 \leq y_3$. We can then iterate over every pixel of the bounding box of the triangle (see Fig. 7) and apply the interior test (Alg. 6) to each. Below is the basic pseudocode, without any cache misses/ hits considered.

Algorithm 7 Triangle fill by interior test (see Alg. 6 for interior test).

```

1: procedure TRIANGLEFILLINTERIORTEST( $x_1, y_1, x_2, y_2, x_3, y_3$ ) ▷ Assuming  $y_1 \leq y_2 \leq y_3$ 
2:    $xmin \leftarrow \min(x_1, x_2, x_3)$ 
3:    $xmax \leftarrow \max(x_1, x_2, x_3)$ 
4:   for  $x \leftarrow xmin$  to  $xmax$  do
5:     for  $y \leftarrow y_1$  to  $y_3$  do
6:       if IsInterior( $x, y$ ) then putPixel( $x, y$ )

```

This algorithm completely avoids floating point operations, unlike Alg. 5 (triangle fill by line sweep). In my “gfx-v4” repo, it is implemented in <https://github.com/OxLeo/gfx-v4/blob/master/src/gfx/gfx.c#L444>.

3 Circle rasterisation

References

- [1] *The bresenham line-drawing algorithm*. [Online]. Available: <https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html>.
- [2] M. Damian, *From vertices to fragments: Rasterization*. [Online]. Available: <http://www.csc.villanova.edu/~mdamian/Past/csc8470sp15/notes/Rasterization.pdf>.
- [3] D. Thain, *Gfx: A simple graphics library (v2)*. [Online]. Available: <https://www3.nd.edu/~dthain/courses/cse20211/fall2013/gfx/>.
- [4] P. Bhowmick, *Computer graphics selected lecture notes*, 2018. [Online]. Available: <https://cse.iitkgp.ac.in/~pb/pb-graphics-2018.pdf>.

A Appendices

A.1 Bresenham's straight line drawing algorithm – pseudocode

Algorithm 8 Bresenham's full line drawing.

```

1: procedure FIND-OCTANT( $x_1, y_1, x_2, y_2$ ) ▷ See Fig. 1
2:    $m \leftarrow \frac{y_2 - y_1}{x_2 - x_1}$ 
3:   if  $x_1 \leq x_2$  and  $0 \leq m \leq 1$  then
4:     return 0 ▷ 1st
5:   else if  $y_1 \leq y_2$  and  $m > 1$  then
6:     return 1 ▷ etc.
7:   else if  $y_1 \leq y_2$  and  $m < -1$  then
8:     return 2
9:   else if  $x_2 \leq x_1$  and  $0 \geq m \geq -1$  then
10:    return 3
11:  else if  $x_2 \leq x_1$  and  $0 < m \leq 1$  then
12:    return 4
13:  else if  $y_2 \leq y_1$  and  $m > 1$  then
14:    return 5
15:  else if  $y_2 \leq y_1$  and  $m < -1$  then
16:    return 6
17:  else if  $x_1 \leq x_2$  and  $-1 \leq m \leq 0$  then
18:    return 7
19:  else ▷  $x_1 = x_2$ , vertical line
20:    return 8
21:
22: procedure BRESENHAM( $x_1, y_1, x_2, y_2$ )
23:    $\Delta x \leftarrow x_2 - x_1$ 
24:    $\Delta y \leftarrow y_2 - y_1$ 
25:    $\epsilon \leftarrow 0$ 
26:    $oct \leftarrow \text{Find-Octant}(x_1, y_1, x_2, y_2)$ 
27:   if  $oct = 0$  then ▷ 0 to 45 degrees with x axis
28:      $y \leftarrow y_1$ 
29:     for  $x = x_1..x_2$  do
30:       Draw-Pixel( $x, y$ )
31:        $\epsilon \leftarrow \epsilon + \Delta y$ 
32:       if  $2\epsilon \geq \Delta x$  then
33:          $\epsilon \leftarrow \epsilon - \Delta x$ 
34:          $y \leftarrow y + 1$ 
35:   else if  $oct = 1$  then ▷ 45 to 90
36:      $x \leftarrow x_1$ 
37:     for  $y = y_1..y_2$  do
38:       Draw-Pixel( $x, y$ )
39:        $\epsilon \leftarrow \epsilon + \Delta x$ 
40:       if  $2\epsilon \geq \Delta y$  then
41:          $\epsilon \leftarrow \epsilon - \Delta y$ 
42:          $x \leftarrow x + 1$ 
43:   else if  $oct = 2$  then ▷ 90 to 135
44:      $x \leftarrow x_1$ 
45:     for  $y = y_1..y_2$  do
46:       Draw-Pixel( $x, y$ )
47:        $\epsilon \leftarrow \epsilon - \Delta x$ 
48:       if  $2\epsilon \geq \Delta$  then
49:          $\epsilon \leftarrow \epsilon - \Delta y$ 
50:          $x \leftarrow x - 1$ 
51:   else if  $oct = 3$  then ▷ 135 to 180
52:      $y \leftarrow y_1$ 
53:     for  $x = x_1..x_2$  do
54:       Draw-Pixel( $x, y$ )

```

Algorithm 9 Bresenham's full line drawing – cont'ed

```
1:   $\epsilon \leftarrow \epsilon + \Delta x$ 
2:  if  $2\epsilon \geq -\Delta x$  then
3:     $\epsilon \leftarrow \epsilon + \Delta x$ 
4:     $y \leftarrow y + 1$ 
5:  else if  $oct = 4$  then ▷ 180 to 215
6:     $y \leftarrow y_1$ 
7:    for  $x = x_1..x_2$  do
8:      Draw-Pixel( $x, y$ )
9:       $\epsilon \leftarrow \epsilon - \Delta y$ 
10:     if  $2\epsilon \geq -\Delta x$  then
11:        $\epsilon \leftarrow \epsilon + \Delta x$ 
12:        $y \leftarrow y - 1$ 
13:  else if  $oct = 5$  then ▷ 215 to 270
14:     $x \leftarrow x_1$ 
15:    for  $y = y_1..y_2$  do
16:      Draw-Pixel( $x, y$ )
17:       $\epsilon \leftarrow \epsilon - \Delta x$ 
18:      if  $2\epsilon \geq -\Delta y$  then
19:         $\epsilon \leftarrow \epsilon - \Delta y$ 
20:         $x \leftarrow x - 1$ 
21:  else if  $oct = 6$  then ▷ 270 to 315
22:     $x = x_1$ 
23:    for  $y = y_1..y_2$  do
24:      Draw-Pixel( $x, y$ )
25:       $\epsilon \leftarrow \epsilon + \Delta x$ 
26:      if  $2\epsilon \geq -\Delta y$  then
27:         $\epsilon \leftarrow \epsilon + \Delta y$ 
28:         $x \leftarrow x + 1$ 
29:  else if  $oct = 7$  then ▷ 315 to 360
30:     $x \leftarrow x_1$ 
31:    for  $y = y_1..y_2$  do
32:      Draw-Pixel( $x, y$ )
33:       $\epsilon \leftarrow \epsilon + \Delta x$ 
34:      if  $2\epsilon \geq -\Delta y$  then
35:         $\epsilon \leftarrow \epsilon + \Delta y$ 
36:         $x \leftarrow x + 1$ 
37:  else if  $oct = 8$  then ▷ Vertical line
38:    ▷ Draw a vertical at line at  $x_1$  between  $y_1, y_2$ 
```

A.2 Bresenham's line drawing source code in C

From <https://github.com/0xLeo/gfx-v4>.

Listing 1: Bresenham's code (src/bresenham.c).

```
1  /*
2  A simple graphics library for CSE 20211 by Douglas Thain
3
4  This work is licensed under a Creative Commons Attribution 4.0 International
   License.  https://creativecommons.org/licenses/by/4.0/
5
6  For complete documentation, see:
7  http://www.nd.edu/~dthain/courses/cse20211/fall2013/gfx
8  Version 3, 11/07/2012 - Now much faster at changing colors rapidly.
9  Version 2, 9/23/2011 - Fixes a bug that could result in jerky animation.
10 */
11
12 #include <X11/Xlib.h>
13 #include <unistd.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <math.h>
17
18 #include "gfx.h"
19
20 // <-- omitted -->
21 //
22 static unsigned int find_octant(int x1, int y1, int x2, int y2) {
23     if (x1 == x2)
24         return 8;
25     float m = (float)(y2 - y1)/(x2 - x1);
26     if ((x1 <= x2) && (0 <= m) && (m <= 1))
27         return 0;
28     else if ((y1 <= y2) && (m > 1))
29         return 1;
30     else if ((y1 <= y2) && (m < -1))
31         return 2;
32     else if ((x2 <= x1) && (0 >= m) && (m >= -1))
33         return 3;
34     else if ((x2 <= x1) && (0 < m) && (m <= 1))
35         return 4;
36     else if ((y2 <= y1) && (m > 1))
37         return 5;
38     else if ((y2 <= y1) && (m < -1))
39         return 6;
40     else if ((x1 <= x2) && (-1 <= m) && (m <= 0))
41         return 7;
42 }
43
44
45 /* Draw a line from (x1,y1) to (x2,y2) using Bresenham's */
46 void gfx_line_bres(int x1, int y1, int x2, int y2)
47 {
48     int dx = x2 - x1;
49     int dy = y2 - y1;
50     float m = (float)dy/dx;
51     int err = 0;
52     int y = y1, x = x1;
53     unsigned int oct = find_octant(x1, y1, x2, y2);
54     switch(oct){
55         case 0: // 1st octant
56             y = y1;
57             for (x = x1; x < x2; x++) {
58                 gfx_point(x, y);
```

```

59         err += dy;
60         if (2*err >= dx){
61             err -= dx;
62             y++;
63         }
64     }
65     break;
66 case 1:
67     x = x1;
68     for (y = y1; y < y2; y++) {
69         gfx_point(x, y);
70         err += dx;
71         if (2*err >= dy){
72             err -= dy;
73             x++;
74         }
75     }
76     break;
77 case 2:
78     x = x1;
79     for (y = y1; y < y2; y++) {
80         gfx_point(x, y);
81         err -= dx;
82         if (2*err >= dy){
83             err -= dy;
84             x--;
85         }
86     }
87     break;
88 case 3:
89     y = y1;
90     for (x = x1; x > x2; x--) {
91         gfx_point(x, y);
92         err += dy;
93         if (2*err >= -dx){
94             err += dx;
95             y++;
96         }
97     }
98     break;
99 case 4:
100    y = y1;
101    for (x = x1; x > x2; x--) {
102        gfx_point(x, y);
103        err -= dy;
104        if (2*err >= -dx){
105            err += dx;
106            y--;
107        }
108    }
109    break;
110 case 5:
111    x = x1;
112    for (y = y1; y > y2; y--) {
113        gfx_point(x, y);
114        err -= dx;
115        if (2*err >= -dy){
116            err -= dy;
117            x--;
118        }
119    }
120    break;
121 case 6:
122    x = x1;

```

```

123         for (y = y1; y > y2; y--) {
124             gfx_point(x, y);
125             err += dx;
126             if (2*err >= -dy){
127                 err += dy;
128                 x++;
129             }
130         }
131         break;
132     case 7:
133         y = y1;
134         for (x = x1; x < x2; x++) {
135             gfx_point(x, y);
136             err -= dy;
137             if (2*err >= dx){
138                 err -= dx;
139                 y--;
140             }
141         }
142         break;
143     case 8:
144         if (y1 < y2){
145             for (y = y1; y < y2; y++) {
146                 gfx_point(x, y);
147             }
148         } else {
149             for (y = y2; y < y1; y++) {
150                 gfx_point(x, y);
151             }
152         }
153         break;
154     }
155 }

```