SHAPE RASTERISATION
ALGORITHMS

SUBJECT

ALGORITHMS AND DATA STRUCTURES

BY

0xLeo (github.com/0xleo)

MAY 1, 2021

DRAFT X.YY
MISSING: . . .

Contents

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:

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

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*x`
- Addition of `m*c` 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 $0 \le m \le 1$. We'll later exploit the circular symmetry to derive the remaining 7 octants given the first. To reiterate, the main constraint imposed is

$$0 \le m \le 1, \quad m = \frac{y_2 - y_1}{x_2 - x_1} \tag{1.2}$$

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

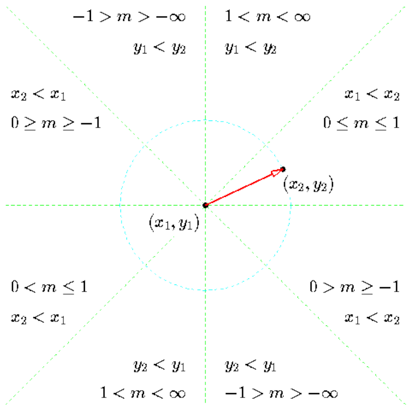


Fig. 1. The 8 octants and their slopes [1].

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).

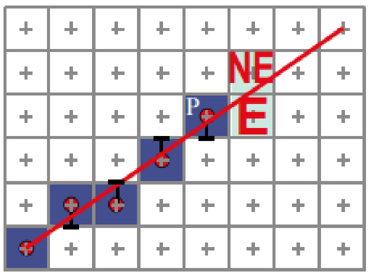


Fig. 2. At every update of pixel (x, y), we choose between the E and NE neighbour [2].

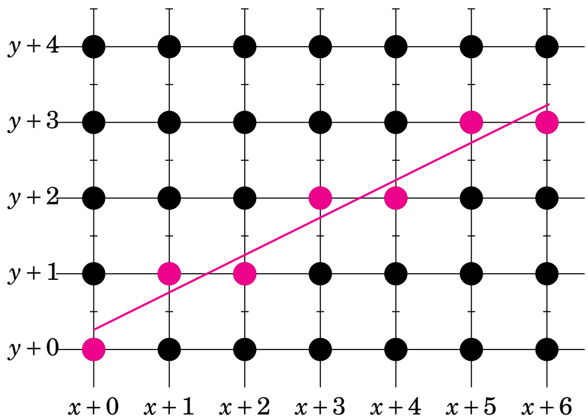


Fig. 3. The nodes represent pixel centres and the segments the midway y between neighbouring pixels. Nodes in magenta represent where the line will be drawn in the 2D discrete space.

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],

$$-0.5 \leq \epsilon < 0.5 \tag{1.3}$$

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).

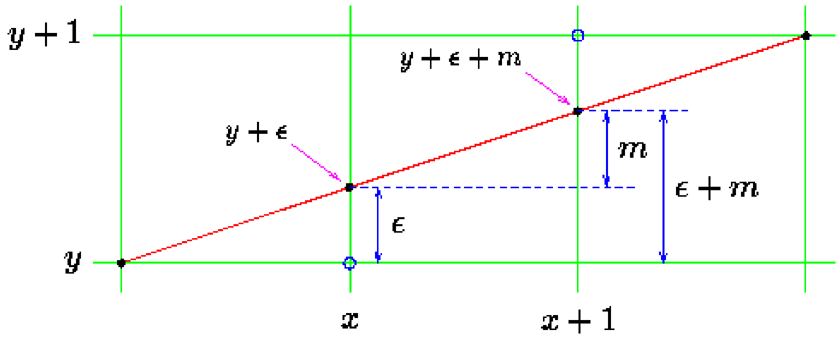


Fig. 4. The error at each pixel during the update [1].

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

$$y+\epsilon+m < y+0.5 \Rightarrow$$
$$\epsilon+m < 0.5$$

, 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 $x \to x+1$ the error gets updated too from ϵ to ϵ_{new}. We know that the error 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)

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

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

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

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.

procedure LINE-DRAWING-FP(x_1, y_1, x_2, y_2)
    $m \leftarrow \frac{y_2-y_1}{x_2-x_1}$
    $\epsilon \leftarrow 0$, $y \leftarrow y_1$ ▷ ϵ, y are all we keep track of.
    for $x = x_1, \ldots, x_2$ **do**
        DrawPixel(x, y)
        if $\epsilon + m < 0.5$ **then**
            $\epsilon \leftarrow \epsilon + m$ ▷ Move E; don't change y
        else
            $\epsilon \leftarrow \epsilon + m - 1$ ▷ Move NE
            $y \leftarrow y + 1$

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

Bresenham relies on integer operations.

$$\epsilon + m < 0.5 \tag{1}$$
$$\epsilon \leftarrow \epsilon + m \tag{2}$$
$$\epsilon \leftarrow \epsilon + m - 1 \tag{3}$$

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

$$2\underbrace{\epsilon\Delta x}_{\epsilon'} + 2\Delta y < \Delta x \tag{1'}$$

Eq. (2) and (3) become respectively

$$\underbrace{\epsilon\Delta x}_{\epsilon'} \leftarrow \underbrace{\epsilon\Delta x}_{\epsilon'} + \Delta y \tag{2'}$$

$$\underbrace{\epsilon\Delta x}_{\epsilon'} \leftarrow \underbrace{\epsilon\Delta x}_{\epsilon'} + \Delta y - \Delta x \tag{3'}$$

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

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

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

procedure BRESENHAM-1ST-OCTANT(x_1, y_1, x_2, y_2)
    $\Delta x \leftarrow x_2 - x_1$
    $\Delta y \leftarrow y_2 - y_1$
    $\epsilon' \leftarrow 0$, $y \leftarrow y_1$ ▷ ϵ', y are all we keep track of.
    if $0 \le \frac{\Delta y}{\Delta x} < 1$ **then**
        for $x = x_1, \ldots, x_2$ **do**
            DrawPixel(x, y)
            if $2(\epsilon' + \Delta y) < \Delta x$ **then**
                $\epsilon' \leftarrow \epsilon' + \Delta y$ ▷ Move E; don't change y
            else
                $\epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$ ▷ Move NE
                $y \leftarrow y + 1$

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 $1 \le m < \infty$, i.e. one that spans at the 2nd octant (Fig. 1). Note that a line $(l1): y = mx + b$ with slope $0 \le m < 1$ 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 $y_1 < y_2$.

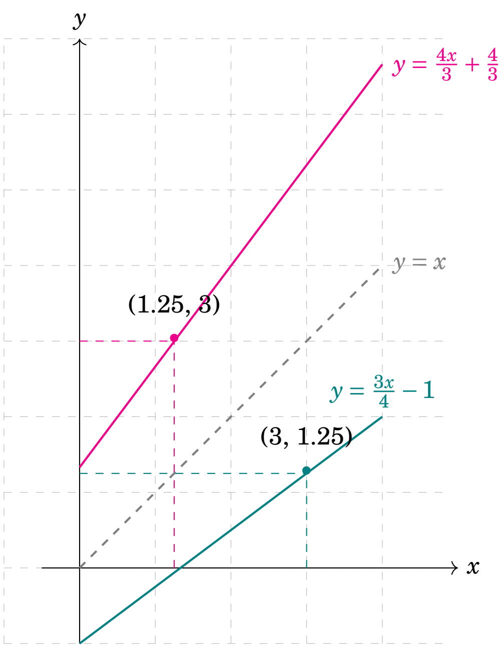


Fig. 5. Two lines in the first two octants symmetric about line $y = x$.

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

```
procedure BRESENHAM-2ND-OCTANT(x1, y1, x2, y2)
    Δx ← x2 − x1
    Δy ← y2 − y1
    ε′ ← 0, x ← x1                                  ▷ ε′, y are all we keep track of.
    if 1 ≤ Δy/Δx and y1 < y2 then
        for y = y1,...,y2 do
            DrawPixel(x, y)
            if 2(ε′ + Δx) < Δy then
                ε′ ← ε′ + Δx
            else
                ε′ ← ε′ + Δx − Δy
                x ← 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.

```
procedure BRESENHAM-2ND-QUADRANT(x1, y1, x2, y2)
    Δx ← x2 − x1
    Δy ← y2 − y1
    ε ← 0, y ← y1
    if Δy/Δx < −1 and y1 < y2 then       ▷ This is the 3rd octant (2nd octant mirror w.r.t y axis)
        for y = y1,...,y2 do
            DrawPixel(x, y)
            if 2(ε − Δx) < Δy then
                ε ← ε − Δx
            else
                ε ← ε − Δx − Δy
                x ← x − 1
    else if 0 ≤ Δy/Δx < −1 and x2 < x1 then     ▷ 4th octant (1st octant mirrored w.r.t y axis)
        for x = x1,...,x2 do
            DrawPixel(x, y)
            if 2(ε + Δy) < −Δx then
                ε ← ε + Δy
            else
                ε ← ε + Δy + Δx
                y ← 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.

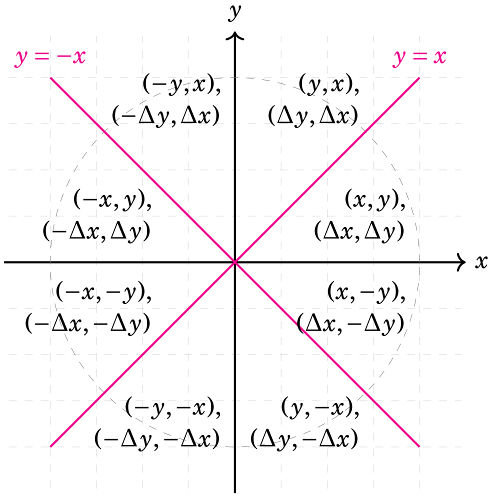


Fig. 6. The Bresenham signs for the full circle given the 1st octant.

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

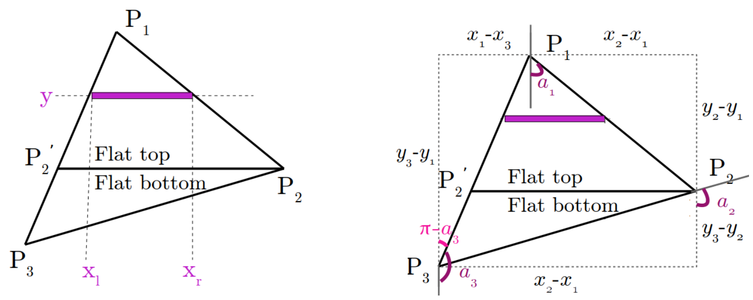
There are various algorithms to draw a solid triangle. Here, we discuss three of them:

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 $\Delta(P_2'P_2P_3)$.



(a) Triangle fill variables.

(b) Angles a_1, a_2, a_3 are utilised by the algorithm. highlighted.

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 it; by $m_{inv,13}$ is if $y \leq y_2$, else by $m_{inv,23}$. The i.s. $m_{inv,13}$ of $\overline{P_1P_1}$ 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 $\Delta(P_1P_2P_2\prime)$ first, followed by $\Delta(P_2\prime P_2P_3)$ is listed below.

Algorithm 5 Triangle fill by line sweep pseudocode.

```
procedure TriangleFillLineSweep(x1, y1, x2, y3, x3, y3)        ▷ Assuming y1 < y2 < y3
    m_inv,13 ← (x3−x1)/(y3−y1)
    m_inv,12 ← (x2−x1)/(y2−y1)
    m_inv,23 ← (x3−x2)/(y3−y2)
    x_l ← x1
    x_r ← x1
    for y = y1..y2 do                                         ▷ Flat top triangle
        for y = int(x_l)..int(x_r) do                         ▷ int denotes the casting from float to int
            drawPixel(x, y)
        x_l ← x_l + m_inv,13
        x_r ← x_r + m_inv,12
    for y = y2..y3 do                                         ▷ Flat bottom
        for y = int(x_l)..int(x_r) do
            drawPixel(x, y)
        x_l ← x_l + m_inv,13
        x_r ← 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* $\boldsymbol{a} = (a_x, a_y)$*, its perp vector* $\boldsymbol{a}^\perp$ *is defined as a vector with the same magnitude rotated by* 90° *ccw:*
>
> $$\boldsymbol{a}^\perp = (-a_y, a_x) \tag{2.1}$$

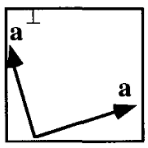
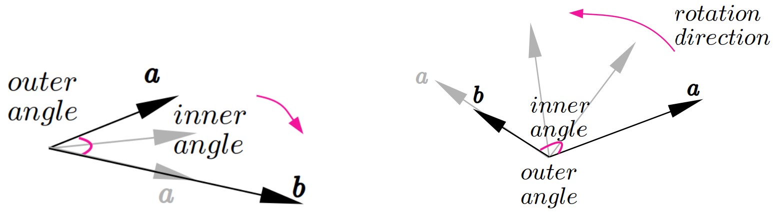


Fig. 8. Vector **a** and its "perp" (**a**) rotated by 90° ccw.

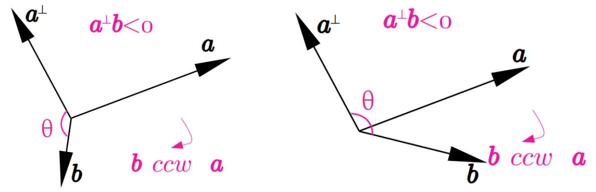
The perp vector comes in handy when we want to determine the relative orientation of two vectors, e.g. whether **a** is cw or ccw from **b**. But first, it's imortant to clarify what is meant by cw and ccw. By saying that **b** is cw of **a**, it is implied that to rotate **b** by the *inner* (smaller)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



(a) **b** cw from **a** **(b)** **b** ccw from **a**

Fig. 9. cw and ccw terminology

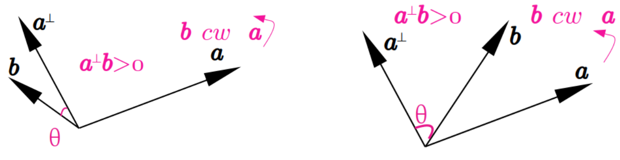
Back to the perp vector, how is it able to tell us the relative orientation between **a** and **b**? Remember that $\mathbf{b}^\perp$ is **b** rotated by 90° ccw. It turns out that if **b** is ccw from **a**, then $\mathbf{a}^\perp\mathbf{b} = \|\mathbf{a}\|\,\|\mathbf{b}\|\cos(\theta) < 0$, since $\theta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



(a) $\mathbf{a}^\perp\mathbf{b} < 0$ when b is ccw from **a** and the angle between **a**, **b** is obtuse. **(b)** $\mathbf{a}^\perp\mathbf{b} < 0$ when b is ccw from **a** and the angle between **a**, **b** is acute.

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

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



(a) $\mathbf{a}^\perp\mathbf{b} > 0$ when b is cw from **a** and the angle between **a**, **b** is obtuse. **(b)** $\mathbf{a}^\perp\mathbf{b} > 0$ when b is cw from **a** and the angle between **a**, **b** is acute.

Fig. 11. $\mathbf{a}^\perp\mathbf{b} < 0$ when **b** is ccw is **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* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *is defined as:*

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

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

To summarise, it has the following properties:

COROLLARY 2.1 (properties perp product).

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

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

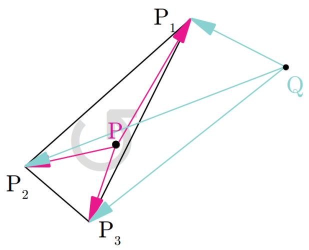
$$a^{\perp}b = 0 \quad \text{if} \quad b = \lambda a, \quad \lambda \in \mathbb{R} \tag{2.5}$$

2.2.2 Stating the interior test

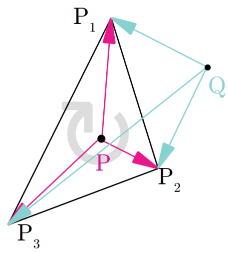
Using Cor. **??**, we can formulate whether a 2D point P is inside or outside a triangle $\Delta(P_1P_2P_3)$. We can use the perp dot product of vectors $\overrightarrow{PP_1}$, $\overrightarrow{PP_2}$, and $\overrightarrow{PP_3}$ to deduce whether P is inside or outside of the triangle, given that we 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.



(a) Vectors connecting P and Q to the vertices when P_1, P_2, P_3 are ccw.



(b) Vectors connecting P and Q to the vertices when P_1, P_2, P_3 are cw.

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):

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

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

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

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

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

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

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

, 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 $\Delta(P_1P_2P_3)$.

procedure PERDDOTPROD(x_1,y_1,x_2,y_2)
    return $x_1y_2 - y_1x2$

procedure ISINTERIOR($x_1,y_1,x_2,y_2,x_3,y_3,x,y$)
    $PP1_x \leftarrow x - x_1$ ▷ vector from $P(x,y)$ to verices $P_i(x_i,y_i)$
    $PP1_y \leftarrow y - y_1$
    $PP2_x \leftarrow x - x_2$
    $PP2_y \leftarrow y - y_2$
    $PP3_x \leftarrow x - x_3$
    $PP3_y \leftarrow y - y_3$
    cw $\leftarrow$ PerdDotProd($PP1_x,PP1_y,PP2_x,PP2_y$) < 0 AND
        PerdDotProd($PP2_x,PP2_y,PP3_x,PP3_y$) < 0 AND
        PerdDotProd($PP3_x,PP3_y,PP1_x,PP1_y$) < 0
    ccw$\leftarrow$ PerdDotProd($PP1_x,PP1_y,PP2_x,PP2_y$) > 0 AND
        PerdDotProd($PP2_x,PP2_y,PP3_x,PP3_y$) > 0 AND
        PerdDotProd($PP3_x,PP3_y,PP1_x,PP1_y$) > 0
    return cw 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 $y_1 \le y_2 \le y_3$. 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).

procedure TRIANGLEFILLINTERIORTEST(x_1,y_1,x_2,y_2,x_3,y_3) ▷ Assuming $y_1 \le y_2 \le y_3$
    xmin $\leftarrow$ min(x_1,x_2,x_3)
    xmax $\leftarrow$ max(x_1,x_2,x_3)
    for x$\leftarrow$ xmin . . xmax **do**
        for y$\leftarrow$ y_1. . $y3$ **do**
            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/0xLeo/gfx-v4/blob/master/src/gfx/gfx.c#L444.

3 Circle rasterisation (midpoint algorithm)

3.1 Derivation of the algorithm

One way to draw a circle centred at $(0,0)$ is by plotting the two y's given by the circle equation for each x.

$$x^2 + y^2 = r^2 \Rightarrow$$
$$y = \pm\sqrt{r^2 - x^2}$$

However, this method has two drawbacks; the square root is very expensive and the line is not guaranteed to be continuous, especially when the slope is high.

The goal is to derive an algorithm that plots a continuous circle using cheap operations. We can exploit the 8–way symmetry of a circle about its centre to derive the algorithm for one octant and then derive the rest, as illustrated in Fig. 6.

For the sake of convention, we will start with octant 2, i.e. starting from $x = 0,\ y = r$ and incrementing x until $y = x$, therefore for the slope m: $0 \le m \le 1$. Because we work in octant 2 (going clockwise), assuming we have just plotted pixel (x_p, y_p), we can either move to $E(x_p + 1, y_p)$ or $SE(x_p + 1, y_p - 1)$ (Fig. 13). Which of the two to select though? The idea is that if the circle curve passes above the midpoint $M(x_p + 1, y_p - \frac{1}{2})$ of the next two choices, then we select $E(x_p + 1, y_p)$. Otherwise, we select $SE(x_p + 1, y_p - 1)$.

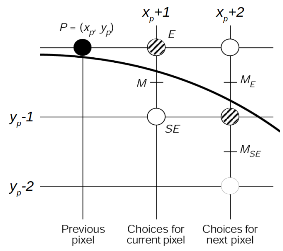


Fig. 13. The two possible choices for the next pixel when drawing a circle at octant 2; E and SE. For the above curve, we select E and SE as the next two pixels (striped).

We need a decision variable to determine which pixel to visit next (E or SE). The midpoint M between S and SE is at distance $(x_p + 1)^2 + (y_p - \frac{1}{2})^2$ from the origin. As shown in Fig. 13:

- If $r^2 \ge (x_p + 1)^2 + (y_p - \frac{1}{2})^2$, then we choose to visit the E pixel as it's closer to the line.
- Otherwise SE is closer and we visit this one.

Therefore we define the decision variable w.r.t. the midpoint $M(x_p + 1, y_p - \frac{1}{2})$ as:

$$D = F(M) = F\left(x_p + 1, y_p - \frac{1}{2}\right)$$
$$= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2$$

And we decide:

- If $D \ge 0$, the next pixel is SE.
- If $D < 0$, the next pixel is E.

It's important to note that at each iteration we compute the current D given the previous. Therefore we need to define how we accumulate it in case E or SE was chosen next.

- In case $E(x_p + 1, y_p - \frac{1}{2})$ was chosen, then the new midpoint (at $x_p + 1 + 1$) is $M(x_p + 2, y_p - \frac{1}{2})$. Then it can be proven (App. A.3) that the new value of the decision variable is:

  $$D_E = D + (2x_p + 3) \tag{3.1}$$

  Hence, D is incremented by $2x_p + 3$.

- In case $SE(x_p+1, y_p-1-\frac{1}{2})$ was chosen, then the new midpoint is also $M(x_p+2, y_p-\frac{1}{2})$ – same as before. Then it can be proven (App. A.3) that the decision variable D is incremented by $2x_p - 2y_p + 5$:

$$D_{SE} = D + (2x_p - 2y_p + 5) \tag{3.2}$$

All that's left to define is how to initialise D. Because we start at $(0, r)$ going clockwise, the first midpoint is at $M(1, r-\frac{1}{2})$. Therefore the initial decision variable is:

$$\begin{aligned} D_0 &= F(1, r-\frac{1}{2}) \\ &= 1 + (r-\frac{1}{2})^2 - r^2 \\ &= 1 + r^2 - r + \frac{1}{4} - r^2 \\ &= \frac{5}{4} - r \end{aligned}$$

To avoid dealing with floating points ($\frac{5}{4}$), notice that on each iteration we compare D to 0. D also gets updated with an integer quantity – either $2x_p + 3$ or $2x_p - 2y_p + 5$. Therefore $D + \frac{1}{4}$ is positive only when D is positive; it is safe to drop the $\frac{1}{4}$. Hence, we can set D_0 to $1 - r$ instead.

$$D_0 = 1 - r \tag{3.3}$$

Using the update rules in Eq. (3.1), Eq. (3.2), Eq. (3.3), we can draw a circular arc on the 2nd octant and mirror it on the rest to draw a full circle.

3.2 Pseudocode and implementation of the algorithm

Algorithm 8 Midpoint algorithm for circle drawing at a point (x_0, y_0) with radius r.

procedure MIDPOINTALGORITHM(x_0, y_0, r)
    $x \leftarrow 0$
    $y \leftarrow r$
    $D \leftarrow 1 - r$ ▷ Decision variable
    do
        putPixel($x + x_0,\ y + y_0$)   ▷ Octant 2; draw all octants to fill the circle
        putPixel($y + x_0,\ x + y_0$)   ▷ Octant 1
        putPixel($-x + x_0,\ y + y_0$)   ▷ Octant 3
        putPixel($-y + x_0,\ x + y_0$)   ▷ Octant 4
        putPixel($-y + x_0,\ -x + y_0$)   ▷ Octant 5
        putPixel($-x + x_0,\ -x + y_0$)   ▷ Octant 6
        putPixel($x + x_0,\ -y + y_0$)   ▷ Octant 7
        putPixel($y + x_0,\ -x + y_0$)   ▷ Octant 8
        if $D < 0$ **then**
            $D \leftarrow D + 2x + 3$
        else
            $y \leftarrow y - 1$
            $D \leftarrow D + 2x - 2y + 5$
        $x \leftarrow x + 1$
    while $x < y$

An implementation in C is found in my "gfx-v4" repo at https://github.com/0xLeo/gfx-v4/blob/master/src/gfx/gfx.c#L355.

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 9 Bresenham's full line drawing.

```
procedure FIND-OCTANT(x1, y1, x2, y2)                         ▷ See Fig. 1
    m ← (y2 − y1)/(x2 − x1)
    if x1 ≤ x2 and 0 ≤ m ≤ 1 then
        return 0                                               ▷ 1st
    else if y1 ≤ y2 and m > 1 then
        return 1                                               ▷ etc.
    else if y1 ≤ y2 and m < −1 then
        return 2
    else if x2 ≤ x1 and 0 ≥ m ≥ −1 then
        return 3
    else if x2 ≤ x1 and 0 < m ≤ 1 then
        return 4
    else if y2 ≤ y1 and m > 1 then
        return 5
    else if y2 ≤ y1 and m < −1 then
        return 6
    else if x1 ≤ x2 and −1 ≤ m ≤ 0 then
        return 7
    else                                                       ▷ x1 = x2, vertical line
        return 8

procedure BRESENHAM(x1, y1, x2, y2)
    Δx ← x2 − x1
    Δy ← y2 − y1
    ε ← 0
    oct ← Find-Octant(x1, y1, x2, y2)
    if oct = 0 then                                            ▷ 0 to 45 degrees with x axis
        y ← y1
        for x = x1..x2 do
            Draw-Pixel(x, y)
            ε ← ε + Δy
            if 2ε ≥ Δx then
                ε ← ε − Δx
                y ← y + 1
    else if oct = 1 then                                       ▷ 45 to 90
        x ← x1
        for y = y1..y2 do
            Draw-Pixel(x, y)
            ε ← ε + Δx
            if 2ε ≥ Δy then
                ε ← −Δy
                x ← x + 1
    else if oct = 2 then                                       ▷ 90 to 135
        x ← x1
        for y = y1..y2 do
            Draw-Pixel(x, y)
            ε ← ε − Δx
            if 2ε ≥ Δ then
                ε ← ε − Δy
                x ← x − 1
    else if oct = 3 then                                       ▷ 135 to 180
        y ← y1
        for x = x1..x2 do
            Draw-Pixel(x, y)
```

Algorithm 10 Bresenham's full line drawing – cont'ed

$\epsilon \leftarrow \epsilon + \Delta x$
if $2\epsilon \geq -\Delta x$ **then**
    $\epsilon \leftarrow \epsilon + \Delta x$
    $y \leftarrow y + 1$
else if $oct = 4$ **then** ▷ 180 to 215
    $y \leftarrow y_1$
    for $x = x_1..x_2$ **do**
        Draw-Pixel(x, y)
        $\epsilon \leftarrow \epsilon - \Delta y$
        if $2\epsilon \geq -\Delta x$ **then**
            $\epsilon \leftarrow \epsilon + \Delta x$
            $y \leftarrow y - 1$
else if $oct = 5$ **then** ▷ 215 to 270
    $x \leftarrow x_1$
    for $y = y_1..y_2$ **do**
        Draw-Pixel(x, y)
        $\epsilon \leftarrow \epsilon - \Delta x$
        if $2\epsilon \geq -\Delta y$ **then**
            $\epsilon \leftarrow \epsilon - \Delta y$
            $x \leftarrow x - 1$
else if $oct = 6$ **then** ▷ 270 to 315
    $x = x_1$
    for $y = y_1..y_2$ **do**
        Draw-Pixel(x, y)
        $\epsilon \leftarrow \epsilon + \Delta x$
        if $2\epsilon \geq -\Delta y$ **then**
            $\epsilon \leftarrow \epsilon + \Delta y$
            $x \leftarrow x + 1$
else if $oct = 7$ **then** ▷ 315 to 360
    $x \leftarrow x_1$
    for $y = y_1..y_2$ **do**
        Draw-Pixel(x, y)
        $\epsilon \leftarrow \epsilon + \Delta x$
        if $2\epsilon \geq -\Delta y$ **then**
            $\epsilon \leftarrow \epsilon + \Delta y$
            $x \leftarrow x + 1$
else if $oct = 8$ **then** ▷ Vertical line
    ▷ Draw a vertical at line at x_1 between $y1$, 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).

```c
/*
A simple graphics library for CSE 20211 by Douglas Thain

This work is licensed under a Creative Commons Attribution 4.0 International
    License.  https://creativecommons.org/licenses/by/4.0/

For complete documentation, see:
http://www.nd.edu/~dthain/courses/cse20211/fall2013/gfx
Version 3, 11/07/2012 - Now much faster at changing colors rapidly.
Version 2, 9/23/2011 - Fixes a bug that could result in jerky animation.
*/

#include <X11/Xlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gfx.h"

// <-- omitted -->
//
static unsigned int find_octant(int x1, int y1, int x2, int y2) {
    if (x1 == x2)
        return 8;
    float m = (float)(y2 - y1)/(x2 - x1);
    if ((x1 <= x2) && (0 <= m) && (m <= 1))
        return 0;
    else if ((y1 <= y2) && (m > 1))
        return 1;
    else if ((y1 <= y2) && (m < -1))
        return 2;
    else if ((x2 <= x1) && (0 >= m) && (m >= -1))
        return 3;
    else if ((x2 <= x1) && (0 < m) && (m <= 1))
        return 4;
    else if ((y2 <= y1) && (m > 1))
        return 5;
    else if ((y2 <= y1) && (m < -1))
        return 6;
    else if ((x1 <= x2) && (-1 <= m) && (m <= 0))
        return 7;
}


/* Draw a line from (x1,y1) to (x2,y2) using Bresenham's */
void gfx_line_bres(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    float m = (float)dy/dx;
    int err = 0;
    int y = y1, x = x1;
    unsigned int oct = find_octant(x1, y1, x2, y2);
    switch(oct){
        case 0: // 1st octant
            y = y1;
            for (x = x1; x < x2; x++) {
                gfx_point(x, y);
```

```
                err += dy;
                if (2*err >= dx){
                    err -= dx;
                    y++;
                }
            }
            break;
        case 1:
            x = x1;
            for (y = y1; y < y2; y++) {
                gfx_point(x, y);
                err += dx;
                if (2*err >= dy){
                    err -= dy;
                    x++;
                }
            }
            break;
        case 2:
            x = x1;
            for (y = y1; y < y2; y++) {
                gfx_point(x, y);
                err -= dx;
                if (2*err >= dy){
                    err -= dy;
                    x--;
                }
            }
            break;
        case 3:
            y = y1;
            for (x = x1; x > x2; x--) {
                gfx_point(x, y);
                err += dy;
                if (2*err >= -dx){
                    err += dx;
                    y++;
                }
            }
            break;
        case 4:
            y = y1;
            for (x = x1; x > x2; x--) {
                gfx_point(x, y);
                err -= dy;
                if (2*err >= -dx){
                    err += dx;
                    y--;
                }
            }
            break;
        case 5:
            x = x1;
            for (y = y1; y > y2; y--) {
                gfx_point(x, y);
                err -= dx;
                if (2*err >= -dy){
                    err -= dy;
                    x--;
                }
            }
            break;
        case 6:
            x = x1;
```

```
            for (y = y1; y > y2; y--) {
                gfx_point(x, y);
                err += dx;
                if (2*err >= -dy){
                    err += dy;
                    x++;
                }
            }
            break;
        case 7:
            y = y1;
            for (x = x1; x < x2; x++) {
                gfx_point(x, y);
                err -= dy;
                if (2*err >= dx){
                    err -= dx;
                    y--;
                }
            }
            break;
        case 8:
            if (y1 < y2){
                for (y = y1; y < y2; y++) {
                    gfx_point(x, y);
                }
            } else {
                for (y = y2; y < y1; y++) {
                    gfx_point(x, y);
                }
            }
            break;
    }
}
```

A.3 Midpoint algorithm – decision variable update

In case $E(x_p+2, y_p-\frac{1}{2})$ was chosen as the next point, then the new decision variable variable is:

$$\begin{aligned}
D_E &= F(x_p+2, y_p-\frac{1}{2}) \\
&= (x_p+2)^2 + (y_p-\frac{1}{2})^2 - r^2 \\
&= (x_p^2+4x_p+2) + (y_p-\frac{1}{2})^2 - r^2 \\
&= (x_p+2x_p+1) + (2x_p+3) + (y_p-\frac{1}{2})^2 - r^2 \\
&= (x_p+1)^2 + (y_p-\frac{1}{2})^2 - r^2 + (2x_p+3) \\
&= D + (2x_p+3)
\end{aligned}$$

In case $SE(x_p+2, y_p-1-\frac{1}{2})$ was chosen as the next point, then the new decision variable variable is:

$$\begin{aligned}
D_{SE} &= F(x_p+2, y_p-\frac{3}{2}) \\
&= (x_p+2)^2 + (y_p-\frac{3}{2})^2 \\
&= (x_p^2+2x_p+1) + (2x_p+3) + (y_p^2-y_p+\frac{1}{4}) + (-2y_p+\frac{8}{4}) - r^2 \\
&= (x_p+1)^2 + (y_p-\frac{1}{2})^2 - r^2 + (2x_p+3) + (-2y_p+2) \\
&= D + (2x_p-2y_p+5)
\end{aligned}$$