SHAPE RASTERISATION
ALGORITHMS

Subject

Algorithms and Data Structures

By

0xLeo (github.com/0xleo)

September 27, 2020

DRAFT X.YY
Missing: . . .

Contents

1 Bresenhma's line drawing 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 Assumptions

- All pixels are sampled in a discrete 2D lattice.
- The algorithm does not draw any colours – it simple decides whether to draw a pixel or not.
- The line generated does not contain any holes. All line pixels must be 8-connected and each column (x) must correspond to a 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, which is expensive);

- Multiplication `m*x`
- Addition of `m*c` with `b`
- For now, we'll only be implementing the algorithm in the first octant (0 deg to 45 deg with x axis), i.e. assume that for the slope of the line $0 \le m \le 1$.

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 often a significant optimisation. Bresenham's algorithm fully relies on integer operations.

1.3.2 Bresenham's line drawing algorithm idea

Consider drawing a line on the first octant (1) (0 deg to 45 deg)of the 2D discrete lattice. The remaining 7 octants will be addressed later. Therefore 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. $x_2 - x_1 \ge y_2 - y_1$.

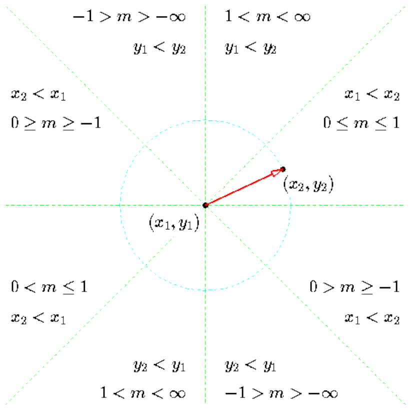

Fig. 1. The 8 octants and their slopes [1].

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).

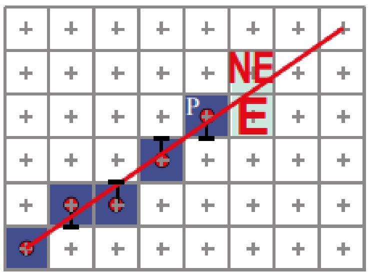

Fig. 2. At every update of pixel (x, y), we choose between the E and NE neighbour [2].

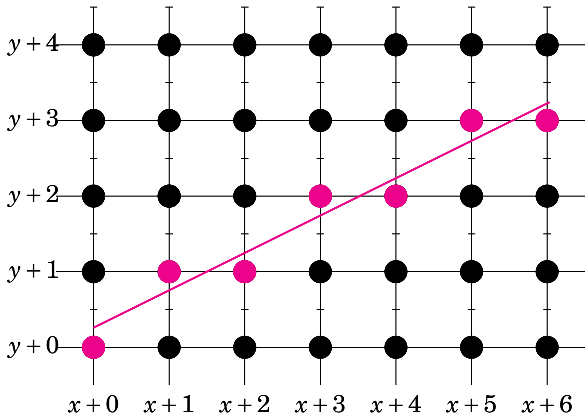

Fig. 3. The nodes represent pixel centres and the segments the midway y between neighbouring pixels. Nodes in magenta represent where the line will be drawn in the 2D discrete space.

2

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).

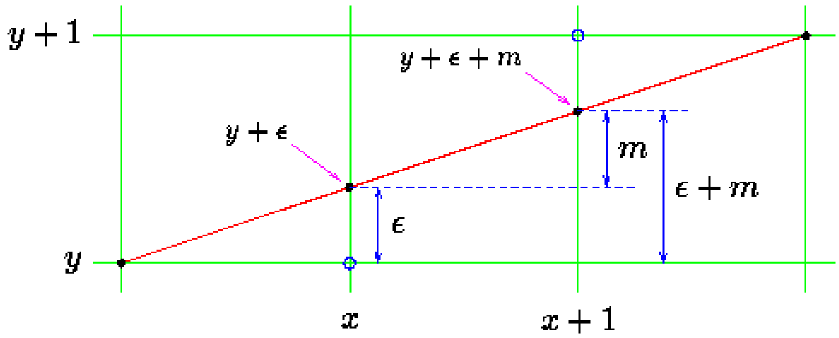

Fig. 4. The error at each pixel during the update [1].

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

$$\epsilon + m < 0.5 \tag{1}$$

$$\epsilon \leftarrow \epsilon + m \tag{2}$$

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

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

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

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

.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(x1, y1, x2, y2)
    Δx ← x2 − x1
    Δy ← y2 − y1
    ε′ ← 0, y ← y1                                  ▷ ε′, y are all we keep track of.
    if 0 ≤ Δy/Δx < 1 then
        for x = x1,...,x2 do
            DrawPixel(x, y)
            if 2(ε′ + Δy) < Δx then
                ε′ ← ε′ + Δy                        ▷ Move E; don't change y
            else
                ε′ ← ε′ + Δy − Δx                   ▷ Move NE
                y ← 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 $y_1 < y_2$.

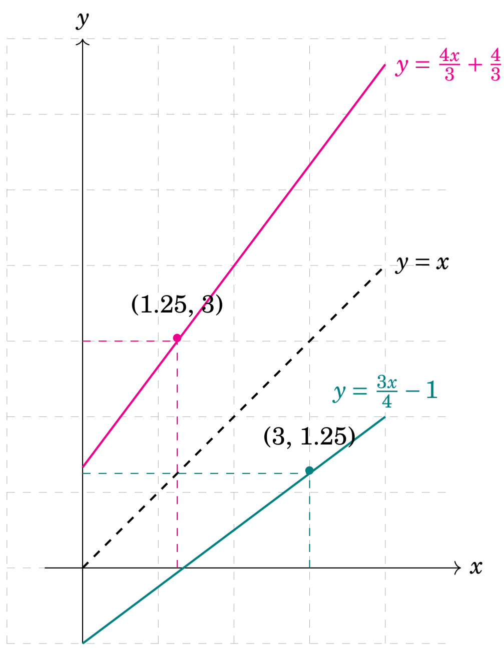

Fig. 5. Two lines in the first two octants symmetric about line $y = x$.

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.

1.4 Bresenham's line drawing generalised

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 it given the symmetry if necessary. The algorithm in its full glory is listed below.

Algorithm 5 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)
            ε ← ε + Δx
            if 2ε ≥ −Δx then
                ε ← ε + Δx
                y ← y + 1
```

Algorithm 6 Bresenham's full line drawing – cont'ed

if ... **then** ▷ Continuing from previous page
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
=0

It is obvious that the algorithm runs in $\mathcal{O}(n)$ time and uses exclusively integer operations during the iteration.

1.5 Implementation in C

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 corresponding repository is at `https://github.com/0xLeo/gfx-v4`. The code in C is found in A.1.

1.6 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.

Cos:

- Does not account for aliasing.

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 line drawing implementation in C

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;
    }
}
```