SHAPE RASTERISATION
ALGORITHMS

Contents

By

0xLeo (github.com/0xleo)

July 18, 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:

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

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 $0 \leq m \leq 1$. We'll later exploit the circular symmetry to derive the remaining 7 octants given the first. To reiterate, the main constraint imposed is

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

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

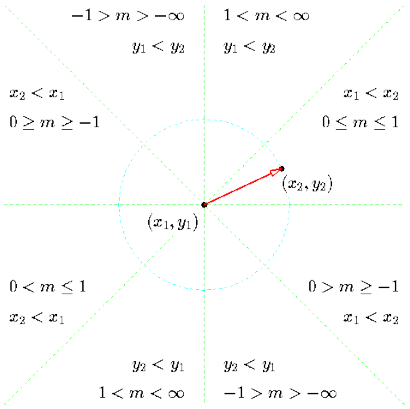


Fig. 1. The 8 octants and their slopes [1].

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).

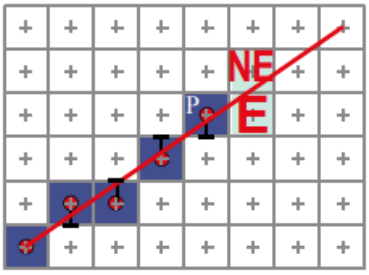


Fig. 2. At every update of pixel (x, y), we choose between the E and NE neighbour [2].

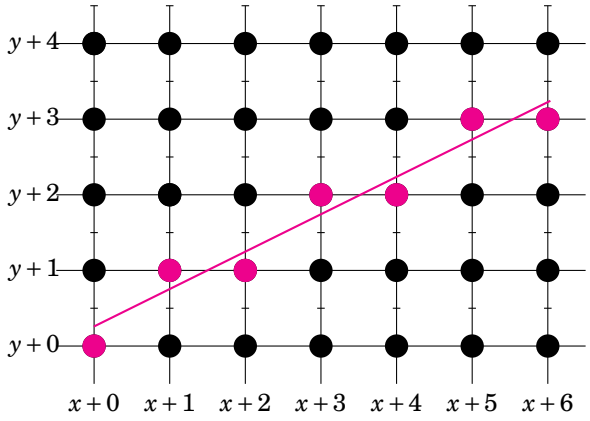


Fig. 3. The nodes represent pixel centres and the segments the midway y between neighbouring pixels. Nodes in magenta represent where the line will be drawn in the 2D discrete space.

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],

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

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).

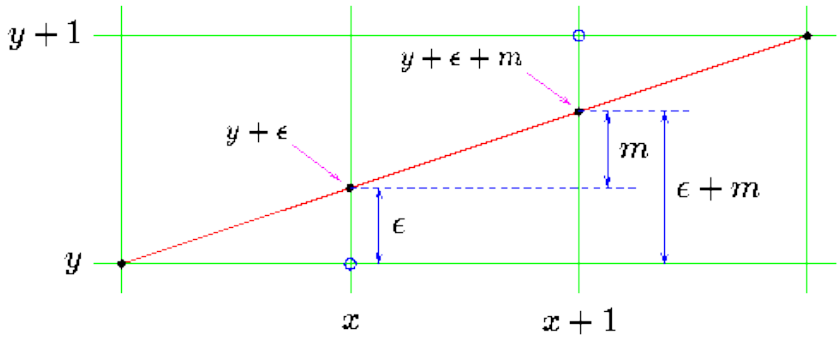


Fig. 4. The error at each pixel during the update [1].

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.

```
1:  procedure LINE-DRAWING-FP(x1, y1, x2, y2)
2:      m ← (y2−y1)/(x2−x1)
3:      ε ← 0, y ← y1                          ▷ ε, y are all we keep track of.
4:      for x = x1,...,x2 do
5:          DrawPixel(x, y)
6:          if ε + m < 0.5 then
7:              ε ← ε + m                      ▷ Move E; don't change y
8:          else
9:              ε ← ε + m − 1                  ▷ Move NE
10:             y ← 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.

```
1:  procedure BRESENHAM-1ST-OCTANT(x1, y1, x2, y2)
2:      Δx ← x2 − x1
3:      Δy ← y2 − y1
4:      ε' ← 0, y ← y1                          ▷ ε', y are all we keep track of.
5:      if 0 ≤ Δy/Δx < 1 then
6:          for x = x1, ..., x2 do
7:              DrawPixel(x, y)
8:              if 2(ε' + Δy) < Δx then
9:                  ε' ← ε' + Δy                ▷ Move E; don't change y
10:             else
11:                 ε' ← ε' + Δy − Δx           ▷ Move NE
12:                 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 $y_1 < y_2$.

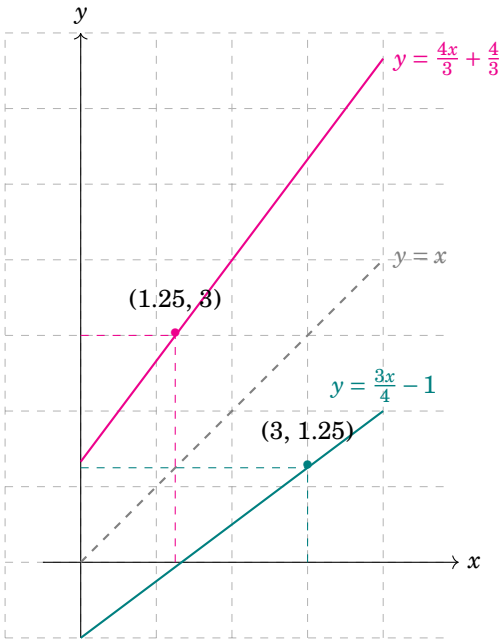


Fig. 5. Two lines in the first two octants symmetric about line $y = x$.

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

procedure BRESENHAM-2ND-OCTANT(x_1, y_1, x_2, y_2)
    $\Delta x \leftarrow x_2 - x_1$
    $\Delta y \leftarrow y_2 - y_1$
    $\epsilon' \leftarrow 0$, $x \leftarrow x_1$ ▷ ϵ', y are all we keep track of.
    if $1 \leq \frac{\Delta y}{\Delta x}$ and $y_1 < y_2$ **then**
        for $y = y_1, \ldots, y_2$ **do**
            DrawPixel(x, y)
            if $2(\epsilon' + \Delta x) < \Delta y$ **then**
                $\epsilon' \leftarrow \epsilon' + \Delta x$
            else
                $\epsilon' \leftarrow \epsilon' + \Delta x - \Delta y$
                $x \leftarrow x + 1$

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(x1, y1, x2, y2)
2:      Δx ← x2 − x1
3:      Δy ← y2 − y1
4:      ϵ ← 0, y ← y1
5:      if Δy/Δx < −1 and y1 < y2 then          ▷ This is the 3rd octant (2nd octant mirror w.r.t y axis)
6:          for y = y1, ..., y2 do
7:              DrawPixel(x, y)
8:              if 2(ϵ − Δx) < Δy then
9:                  ϵ ← ϵ − Δx
10:             else
11:                 ϵ ← ϵ − Δx − Δy
12:                 x ← x − 1
13:     else if 0 ≤ Δy/Δx < −1 and x2 < x1 then   ▷ 4th octant (1st octant mirrored w.r.t y axis)
14:         for x = x1, ..., x2 do
15:             DrawPixel(x, y)
16:             if 2(ϵ + Δy) < −Δx then
17:                 ϵ ← ϵ + Δy
18:             else
19:                 ϵ ← ϵ + Δy + Δx
20:                 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.

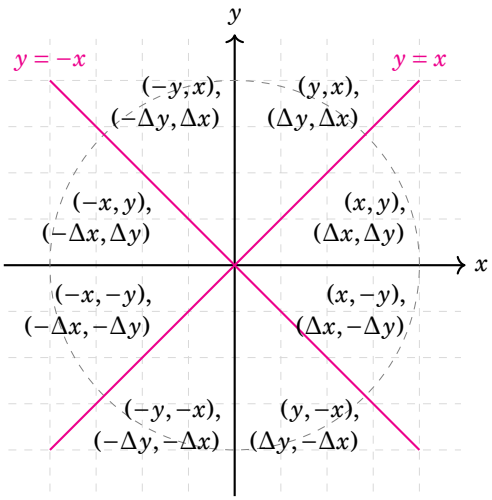


Fig. 6. The Bresenham signs for the full circle given the 1st octant.

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 – i.e. linear time.

Cons:

- Does not account for aliasing.

2 Bézier and De Chasteljau curves from a programmer's perspective

2.1 Applications

Bézier (1910–1999) and De Chasteljau (1930–) were/are two mathematicians who established an intuitive and efficient way of drawing smooth and constrained curves given only a few control points. They focused on cubic curves as cubic curves are not only smoothly joined but also have no "wiggles"'. Several decades ago, this property made them applicable to car design, as Bézier and De Chasteljau used to independently from each other work for Renault and Citröen respectively. Currently, however, their applications have been extended not only to software (fonts, scalable graphics, graphical design applications such as Inkscape etc.), but also to ship, aircraft design, etc.

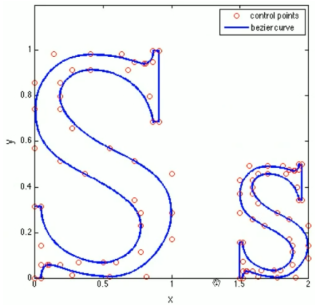


Fig. 7. To render the S character at any size, only a few control points are needed (dots) instead of the entire curve.

We will denote Bézier-De Chasteljau curves as BCD curves. BDC curves are parametric curves (instead of x and y, they are defined given some scalar parameter t) and are based on a certain type of polynomials called Bernstein polynomials, which have some neat properties.

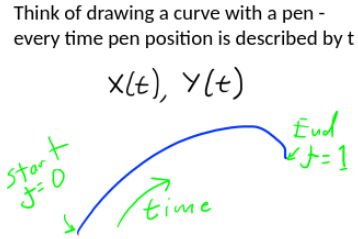


Fig. 8. When we draw a curve with a pen, instead of describing the position as (x, y), we can describe it w.r.t t: $t \mapsto x(t), y(t)$.

2.2 Bernstein polynomials

2.2.1 The Bernstein polynomial subspace

As already known, 3rd degree polynomials are defined as:

$$P(t) = 1 \cdot a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

The functions $\{1, t, t^2, t^3\}$ form a monomial basis (a.k.a. Hermite basis). Therefore the dimension of the space they are a basis of is 4 (since we have 4 basis vectors). Bernstein 3rd degree polynomials are defined as follows:

$$P(t) = (1-t)^3 b_0 + 3(1-t)^2 t b_1 + 3(1-t) t^2 b_2 + t^3 b_3 \tag{2.1}$$

Just for reference, the general definition of Bernstein polynomials is provided.

DEFINITION 2.1 (Bernstein polynomial)**.** *The Bernstein polynomial basis functions of degree n are*

defined as:

$$B_{in}(t) = \binom{n}{i}(1-t)^{n-i}t^i \tag{2.2}$$

, where i is the index of basis function.

For instance, $B_{03} = \binom{3}{0}(1-t)^{3-0}t^0 = (1-t)^3$, $B_{13} = \binom{3}{1}(1-t)^{3-1}t^1 = 3(1-t)^2t$. Without loss of generality, we will see later from De Chasteljau algorithm that it suffices to restrict $t \in (0,1)$. Therefore the basis functions of 3rd degree Bernstein polynomials are $\{(1-t)^3, 3(1-t)^2t, 3(1-t)t^3, t^3\}$. They are a basis because they are linearly independent. Indeed, if

$$c_0(1-t)^3 + c_1 3(1-t)^2t + c_2 3(1-t)t^2 + c_3t^3 = 0$$

then

$$c_0\left(\frac{1-t}{t}\right)^3 + 3c_1\left(\frac{1-t}{t}\right)^2 + 3c_2\left(\frac{1-t}{t}\right) + c_3 = 0$$

Because $\frac{1-t}{t} > 0 \quad \forall\, t \in (0,1)$, we conclude that $c_0 = c_1 = c_2 = c_3 = 0$.

2.2.2 Change of basis from Hermite to Bernstein

As a reminder, a cubic Hermite polynomial is defined as:

$$P(t) = a_0 + ta_1 + t^2a_2 + t^3a_3$$

Therefore it is spanned by the "monomial" basis $\{1, t, t^2, t^3\}$. A question that comes naturally at this point is how the Bernstein is transformed to a Bernstein one, which is $\{(1-t)^3, 3(1-t)^3t, 3(1-t)t^2, t^3\}$, or more generally; how a Hermite polynomial is transformed into a Bernstein one, which is:

$$\begin{aligned} P(t) &= b_0(1-t)^3 + 3b_1(1-t)^2t + 3b_2(1-t)t^2 + b_3t^3 \\ &= b_0(1-3t+3t^2-t^3) + b_1(3t-6t+3t^3) + b_2(3t^2-3t^3) + b_3t^3 \\ &= \begin{bmatrix} 1-3t+3t^2-t^3 & 3t-6t+3t^3 & 3t^2-3t^3 & t^3 \end{bmatrix} \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{\mathbf{B}} \end{aligned} \tag{1}$$

Having expanded the Bernstein basis vectors, it's easy to map from Hermite to Bernstein with a matrix multiplication:

$$\begin{bmatrix} 1-3t+3t^2-t^3 \\ 3t-6t+3t^3 \\ 3t^2-3t^3 \\ t^3 \end{bmatrix}^\top = \underbrace{\begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{C}} \underbrace{\begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}}_{\mathbf{H}} \tag{2}$$

From Eq. (1), (2), we obtain that $P(t) = \mathbf{H}^\top\mathbf{C}^\top\mathbf{B}$.

COROLLARY 2.1 (change of basis from Hermite to Bernstein)**.** *The cubic Bernstein polynomial $P(t) = b_0(1-t)^3 + 3b_1(1-t)^2t + 3b_2(1-t)t^2 + b_3t^3$ can be written w.r.t. the standard monomial basis $\{1, t, t^2t^3\}$ as follows:*

$$P(t) = \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}^\top \begin{bmatrix} 1 & -3 & 3-1 & \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}^\top \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.3}$$

2.2.3 Bernstein polynomials as blending functions

The Bernstein polynomials (not just the cubic, but any degree) can also be viewed as a weighted average of their coefficients. Why a weighted average? Because the basis functions sum to 1.

COROLLARY 2.2 (Bernstein basis functions sum to 1). *The basis functions* $\{B_{03}, B_{13}, B_{23}, B_{33}\} = \{(1-t)^3, 3(1-t)^3 t, 3(1-t)t^2, t^3\}$ *of the cubic Bernstein polynomial* $P(t) = b_0(1-t)^3 + b_1 3(1-t)^2 t + b_2 3(1-t)t^2 + b_3 t^3$ *sum to 1*[a]

[a] This holds for any degree n, however we are only interested in the cubic case.

Proof.

$$\begin{aligned} 1 &= ((1-t)+t)^3 \\ &= (1-t)^3 + 3(1-t)^3 t + 3(1-t)t^2 + t^3 \end{aligned}$$

□

Hence it is said that $B_{i3}(t),\ i = 0,1,2,3$ "blend together" the polynomial coefficients $b_i,\ i = 0,1,2,3$. Apart from the fact that they sum to 1, the basis functions $B_{i3}(t)$ are strictly positive for $t \in (0,1)$.

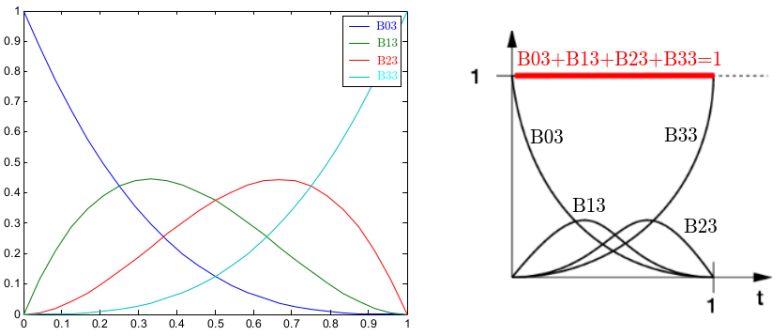


Fig. 9. The 4 basis functions of the cubic Bernstein polynomial.

2.3 The Bézier curve

2.3.1 Definition

Since Bernstein polynomials have been introduced, it is time to define the cubic Bézier curve $\mathscr{C}(t)$:

$$\mathscr{C}(t) = (1-t)^3 \mathbf{b}_0 + 3(1-t)^2 t \mathbf{b}_1 + 3(1-t)t^3 \mathbf{b}_2 + t^3 \mathbf{b}_3, \quad t \in [0,1] \tag{2.4}$$

$\mathbf{b}_i = (x_i, y_i)$ in this case are called control points because they control the shape of $\mathscr{C}(t)$ as we will see soon. Eq. (2.4) can be also expanded in each dimension as:

$$\begin{aligned} x(t) &= (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^3 x_2 + t^3 x_3 \\ y(t) &= (1-t)^3 y_0 + 3(1-t)^2 t y_1 + 3(1-t)t^3 y_2 + t^3 y_3 \end{aligned}$$

2.3.2 Why are $\mathbf{b}_i$ control points?

Plugging in Eq. (2.4) $t = 0$ and $t = 1$ respectively:

$$\begin{aligned} \mathscr{C}(0) &= \mathbf{b}_0\ , \\ \mathscr{C}(1) &= \mathbf{b}_3 \end{aligned}$$

Therefore $\mathbf{b}_0 = \mathscr{C}(0)$ and $\mathbf{b}_3 = \mathscr{C}(1)$ control the beginning and end of the curve. If we derive $\mathscr{C}(t)$, we furthermore compute:

$$\mathscr{C}\prime(t) = 3(\mathbf{b}_1 - \mathbf{b}_0)(1-t)^2 + 6(\mathbf{b}_2 - \mathbf{b}_1)(1-t)t + 3(\mathbf{b}_3 - \mathbf{b}_2)t^2$$

Plugging in the derivative $t = 0$ and then $t = 1$:

$$\begin{aligned} \mathscr{C}\prime(0) &= 3(\mathbf{b}_1 - \mathbf{b}_0)\ , \\ \mathscr{C}\prime(1) &= 3(\mathbf{b}_3 - \mathbf{b}_2) \end{aligned}$$

Therefore the tangent of $\mathscr{C}(t)$ at the beginning ($t=0$) points from $\mathbf{b}_0$ to $\mathbf{b}_1$ and the tangent at the end points from $\mathbf{b}_2$ to $\mathbf{b}_3$. This is how $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ control the curve. In general, a Bézier curve of degree n is controlled by $n+1$ control points.

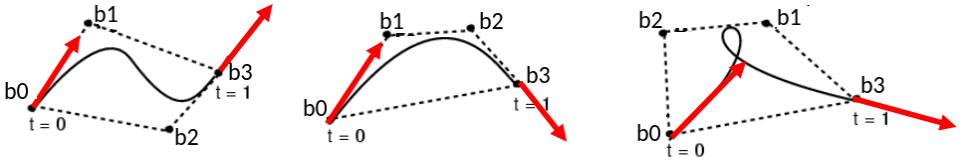


Fig. 10. How the control points affect the shape of the Bézier curve.

It can also be proven that the tangent property holds for Bézier curves of any degree.

2.3.3 Convex hull property

The convex hull property states that the Bézier curve is always contained in the polygon formed by the 4 control points. Before it's proven, some definitions needs to be made.

DEFINITION 2.2 (convex set). *A set $C \subset \mathbb{R}^n$ is convex iff for any two points $\boldsymbol{x}_1, \boldsymbol{x}_2$ in C, the line segment joining them [a] also lies in C, i.e.*

$$\theta \boldsymbol{x}_1 + (1-\theta)\boldsymbol{x}_2 \in C \quad \forall \theta \in [0,1], \quad \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in C \tag{2.5}$$

[a] see parametric straight line equation in A.3

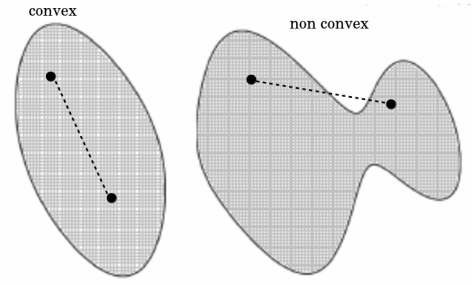


Fig. 11. Example of a convex (left) and a non convex set (right).

DEFINITION 2.3 (convex hull). *Given a set of n points $\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$, the convex hull of them is defined as:*

$$CH(p_1, \ldots, p_n) = \left\{ \boldsymbol{p}_1 p_1 + \ldots + \boldsymbol{p}_n p_n \mid p_1, \ldots, p_n \in [0,1] \wedge \sum_{i=1}^{n} p_i = 1 \right\} \tag{2.6}$$

The convex hull is a set of points is alternatively defined as the smallest convex set containing all points. Visually speaking the convex hull can be pictured by stretching an elastic band so that it is as small as possible and still contains all the control points. For example here is the convex hull for a set of eight points:

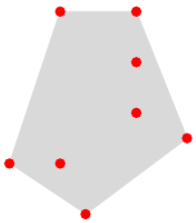


Fig. 12. A convex hull can be visualised as the area inside a rubber band fit around the points.

But, given the set $\{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$why are all combinations $\{p_1\mathbf{p}_1 + \ldots + p_n\mathbf{p}_n \mid p_1, \ldots, p_n \in [0,1] \, and \, \sum_{i=1}^{n} p_i = 1\}$

contained within the convex hull of the latter set? The lemma below proves why.

> **LEMMA 2.1.** *If C is a convex set and $\boldsymbol{p}_1,\ldots,\boldsymbol{p}_n \in C$, then any convex combination (c.c.) $p_1\boldsymbol{p}_1+\ldots+p_n\boldsymbol{p}_n$ (i.e. $p_i \geq 0$ and $p_1+\ldots+p_n=1$) is also contained in C.*

Proof. By induction. For $n=1$ the lemma is trivial. For $n=2$ ($p_1+p_2=1$) then the original set is simply the straight line between p_1 and p_2. The convex combination of p_1 and p_2 is also $p_1p_1+(p_2-p_1)p_2$, which is the equation of the line segment between p_1 and p_2, therefore the c.c. is also within the original set. Assume the statement is true for an integer $n-1$, $n\geq 3$, i.e. the c.c. of $\{p_1,\ldots,p_{n-1}\}$, which is any vector $\mathbf{y}$ such that:

$$\mathbf{y}=p_1\mathbf{p}_1+\ldots+p_n\mathbf{p}_{n-1}, \quad \forall\, p_i : p_i\geq 0,\ p_1+\ldots+p_{n-1}=1$$

$$\therefore \mathbf{y}=\frac{p_1}{p_1+\ldots+p_{n-1}}\mathbf{p}_1+\ldots+\frac{p_{n-1}}{p_1+\ldots+p_{n-1}}\mathbf{p}_{n-1}$$

, is in C. For the next integer n, we want to prove that $\mathbf{z}=p_1\mathbf{p}_1+\ldots+p_n\mathbf{p}_n, \quad \forall\, p_i : p_i\geq 0,\ p_1+\ldots+p_n=1$ is also in C. However

$$\mathbf{z}=p_1\mathbf{p}_1+\ldots+p_{n-1}\mathbf{p}_{n-1}+p_n\mathbf{p}_n=(p_1+\ldots+p_{n-1})\mathbf{y}+p_n\mathbf{p}_n$$

Therefore $\mathbf{z}$ lies in the segment between $\mathbf{y}$ and $\mathbf{p}_n$. We know that both $\mathbf{y}$ and $\mathbf{p}_n$ are in the convex set C and because C is convex all points along the latter segment (described by vector $\mathbf{z}$) also belong in C. □

Returning to the Bézier curve, these lemmas allows us to a establish a nice property about its shape. Recall that a Bézier curve is given by Eq. (2.4):

$$C(t)=(1-t)^3\mathbf{b}_0+3(1-t)^2t\mathbf{b}_1+3(1-t)t^3\mathbf{b}_2+t^3\mathbf{b}_3$$

It's been proven that the weights $(1-t)^3, 3(1-t)^2t, 3(1-t)t^3, t^3$ are non-negative and sum to 1. Therefore we can establish the following nice property.

> **LEMMA 2.2** (Bézier curves and convex hull)**.** *All points of the cubic Bezier curve*
>
> $$C(t)=(1-t)^3\boldsymbol{b}_0+3(1-t)^2t\boldsymbol{b}_1+3(1-t)t^3\boldsymbol{b}_2+t^3\boldsymbol{b}_3$$
>
> *lie within the convex hull defined by the four control points $\boldsymbol{b}_0,\boldsymbol{b}_1,\boldsymbol{b}_2,\boldsymbol{b}_3$.*

This property is helpful as we know that the shape of the Bezier curve is spatially constrained in the polygon defined by $\mathbf{b}_0,\mathbf{b}_1,\mathbf{b}_2,\mathbf{b}_3$. Other curves, such as B-splines do not have that property, which makes them difficult to control or predict.

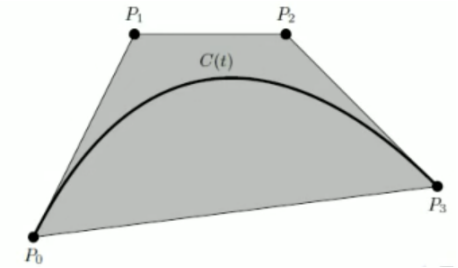


Fig. 13. A Bézier curve is always contained within of the convex hull of its control points.

The convex hull lemma holds for Bézier curves of any degree, however its generalisation will not be proven.

2.3.4 Variation diminishing property

Variation diminishing property limits how "wiggly" Bézier curves can be. Without any proof, it states that:

> **LEMMA 2.3** (variation diminishing of Bézier curves)**.** *The number of intersection points of any straight line with a Bezier curve is at most the number of intersection points of the same straight line with the*

control polygon of the curve.

In non technical words, it states that Bezier curves are smoother than the polygon formed by their control points. This property gives us another hint in advance regarding the shape of the Bezier curve to be drawn given some control points.

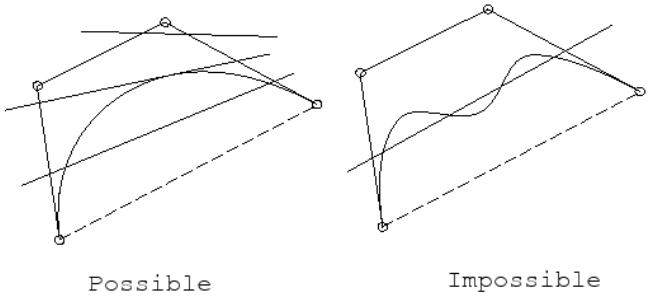


Fig. 14. Left: The straight line crosses the Bézier curve up to 2 times and the polygon up to 2 times. Right: the straight line crosses the Bézier curve 3 times but cross the polygon 2 times.

2.4 De Chasteljau'salgorithm to draw Bézier curves

Bézier curves have some properties, but one needs to know how to draw them in practice. One could compute the Bernstein coefficients from Eq. (2.2) for each t but this gets computationally expensive, especially for large degrees n. De Chasteljau independently developed a recursive algorithm to compute each point of a Bezier curve without relying directly on Eq. (2.2). We will attempt to derive from scratch a De Chasteljau's method to draw cubic Bézier curves.

The goal is to plot a cubic Bézier curve given 4 control points $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$. Suppose we want to plot a point $\mathbf{b}_0^3(t)$ (this is not a control point, the weird superscript and subscript will become obvious later) in the curve for some fixed $t \in [0,1]$. Therefore $\mathbf{b}_0^3(t)$ is written as:

$$\begin{aligned}\mathbf{b}_0^3(t) &= (1-t)^3\mathbf{b}_0 + 3(1-t)^2 t\mathbf{b}_1 + 3(1-t)t^2\mathbf{b}_2 + t^3\mathbf{b}_3 \\ &= (1-t)\underbrace{[(1-t)^2\mathbf{b}_0 + 2(1-t)t\mathbf{b}_1 + t^2\mathbf{b}_2]}_{\mathbf{b}_0^2} + t\underbrace{[(1-t)^2\mathbf{b}_1 + 2(1-t)t\mathbf{b}_2 + t^2\mathbf{b}_3]}_{\mathbf{b}_1^2} \qquad (1)\end{aligned}$$

Eq. (1) expresses the point to draw $\mathbf{b}_3^0(t)$ as a linear interpolation of two points $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$. These points lie on two quadratic Bézier (coefficients sum to one and non-negative) curves. Therefore $\mathbf{b}_3^0(t)$ is determined by two new control points $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$. If we look at $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$, then each point can be rewritten as:

$$\begin{aligned}\mathbf{b}_0^2(t) &= (1-t)^2\mathbf{b}_0 + 2(1-t)t\mathbf{b}_1 + t^2\mathbf{b}_2 \\ &= (1-t)[(1-t)\mathbf{b}_0 + t\mathbf{b}_1] + t[(1-t)\mathbf{b}_1 + t\mathbf{b}_2] \\ &\overset{\mathbf{b}_i = \mathbf{b}_i^0}{=} (1-t)\underbrace{[(1-t)\mathbf{b}_0^0 + t\mathbf{b}_1^0]}_{\mathbf{b}_0^1} + t\underbrace{[(1-t)\mathbf{b}_1^0 + t\mathbf{b}_2^0]}_{\mathbf{b}_1^1}, \qquad (2)\end{aligned}$$

$$\mathbf{b}_1^2(t) = (1-t)\underbrace{[(1-t)\mathbf{b}_1^0 + t\mathbf{b}_2^0]}_{\mathbf{b}_1^1} + t\underbrace{[(1-t)\mathbf{b}_2^0 + t\mathbf{b}_3^0]}_{\mathbf{b}_2^1} \qquad (3)$$

Eq. (2) and (3) state that each of $\mathbf{b}_0^2(t)$ and $\mathbf{b}_1^2(t)$ is determined as a linear interpolation of $\mathbf{b}_0^1$, $\mathbf{b}_1^1$ and $\mathbf{b}_1^1$, $\mathbf{b}_2^1$ respectively. Finally, moving on to $\mathbf{b}_0^1, \mathbf{b}_1^1$ and $\mathbf{b}_2^1$, they are linear interpolations of the original control points. Renaming for consistency $\mathbf{b}_i$ to $\mathbf{b}_i^0$ respectively, where $i = 0,1,2,3$, then $\mathbf{b}_i^1$ is a linear interpolation of $\mathbf{b}_i^0$ and $\mathbf{b}_{i+1}^0$, $i = 0,1,2$.

To summarise so far, Eq. (1), (2), (3) can be generalised as follows:

$$\mathbf{b}_i^j(t) = (1-t)\mathbf{b}_i^{j-1}(t) + t\mathbf{b}_{i+1}^{j-1}(t), \quad i,j \leq 3 \qquad (2.7)$$

If Eq. (2.7) is unfolded, then it shows how $\mathbf{b}_0^3$ can be computed in a pyramid-like recursive way.

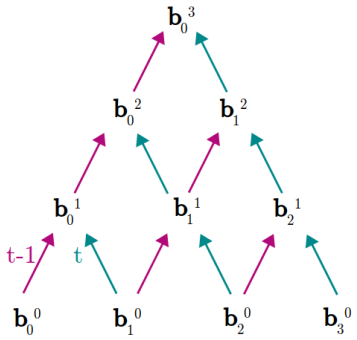


Fig. 15. Computing $\mathbf{b}_0^3$ recursively from the control points. Arrow crossing denotes addition and each arrow denotes a multiplication.

Therefore a pseudocode to implement De Chasteljau (with uniform sampling for t from 0 to 1) is the following.

Algorithm 5 De Chasteljau curve drawing with uniform sampling – works with any number of control points, therefore curve of any degree.

```
1:  procedure ComputePoint(points, t)    ▷ Computes the topmost point of the DeChasteljau pyramid
2:      if len(points) = 0 then
3:          DrawPixel(points)                                  ▷ Only one point – we're at the top
4:      else
5:          higherLevelPoints ← {}
6:          for i= 0,...,len(points)-1 do
7:              higherLevelPoint ← (1 − t)points[i] + t · points[i+1]
8:              higherLevelPoints.append(higherLevelPoint)
            ComputePoint(higherLevelPoints)
9:
10: procedure DeChasteljau(b_0, b_1, b_2, b_3, dt)    ▷ b_i are the control points, dt the sampling width
11:     t ← 0
12:     while t ≤ 1 do
13:         ComputePoint(b_0, b_1, b_2, b_3, t)
14:         t ← t + dt
```

Although uniform is relatively simple to implement, it was the drawback that if the portion of the curve being plotted is too flat (straight line), it's likely that we end up plotting redundant points. If the curve portion is too curvy, we may not plot enough points to capture its curvature. There are variations of the basic De Chasteljau's algorithm which subdivide the portion to draw based on the curvature, however they will not be discussed here.

To get better intuition, it's also helpful to understand how De Chasteljau's can be recreated geometrically. Again, beginning with the control points $\mathbf{b}_0^0, \mathbf{b}_1^0, \mathbf{b}_2^0, \mathbf{b}_3^0$ and for some $t \in [0,1]$, the steps are the following.

1. For each line segment $\mathbf{b}_0^0\mathbf{b}_1^0$, $\mathbf{b}_1^0\mathbf{b}_2^0$, $\mathbf{b}_2^0\mathbf{b}_3^0$, subdivide the segment starting from $\mathbf{b}_{i+1}^0$ to $\mathbf{b}_i^0$ at a ratio t and $1-t$. Therefore each new point $\mathbf{b}_i^1$ is located $t\times$ the length of the segment away from $\mathbf{b}_i^0$.
2. Now we have located $\mathbf{b}_0^1, \mathbf{b}_1^1, \mathbf{b}_2^2$. Once again, $\mathbf{b}_0^2$ is located $t\times$ the length of segment $\mathbf{b}_0^1\mathbf{b}_1^1$ away from $\mathbf{b}_0^1$ and $\mathbf{b}_1^2$ is located $t\times$ the length of segment $\mathbf{b}_1^1\mathbf{b}_2^1$ away from $\mathbf{b}_1^1$.
3. We have determined $\mathbf{b}_0^2$ and $\mathbf{b}_1^2$ so there's only the final point $\mathbf{b}_0^3$ left to determine. Again, subdivide segment $\mathbf{b}_0^2\mathbf{b}_1^2$ and starting from $\mathbf{b}_0^2$ move t times its length away. That's where $\mathbf{b}_0^3$ is located.
4. Repeat for all necessary values of t.

This is visualised by the figure below, where every colour denotes each step.

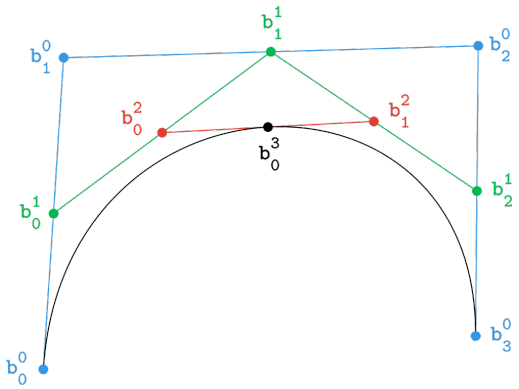


Fig. 16. Geometrically obtaining $\mathbf{b}_0^3$ starting from the control points $\mathbf{b}_i^0$ for $t = 0.5$.

EXAMPLE 2.1. *Given the control points* $\boldsymbol{b}_0 = (-1,0), \boldsymbol{b}_1 = (0,1), \boldsymbol{b}_2 = (0,-1), \boldsymbol{b}_3 = (1,0)$*, use De Chasteljau, to find the coordinates of* $X(1/4)$*.*

SOLUTION 2.1. *We were given* $t = 1/4$*.*

$$\boldsymbol{b}_0^1 = (1-t)\boldsymbol{b}_0 + t\boldsymbol{b}_1 = (-3/4, 1/4),\ \boldsymbol{b}_1^1 = (1-t)\boldsymbol{b}_1 + t\boldsymbol{b}_2 = (0, 2/4),\ \boldsymbol{b}_2^1 = (1-t)\boldsymbol{b}_2 + t\boldsymbol{b}_3 = (1/4, -3/4)$$

$$\boldsymbol{b}_0^2 = (1-t)\boldsymbol{b}_0^1 + t\boldsymbol{b}_1^1 = 3/4(-3/4, 1/4) + 1/4(0, 2/4) = (-9/16, 5/16)$$
$$\boldsymbol{b}_1^2 = (1-t)\boldsymbol{b}_1^1 + t\boldsymbol{b}_2^1 = 3/4(0, 2/4) + 1/4(1/4, -3/4) = (1/16, 3/16)$$

$$\boldsymbol{b}_0^3 = (1-t)\boldsymbol{b}_0^2 + t\boldsymbol{b}_1^2 = 3/4(-9/16, 5/16) + 1/4(1/16, 3/16) = (-26/64, 18/64)$$

De Chasteljau has been implemented in C in my `gfx-v4` repo at https://github.com/0xLeo/gfx-v4.

3 Triangle fill algorithms

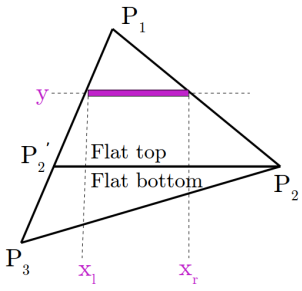
There are various algorithms to draw a solid triangle. Here, we discuss three of them:

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

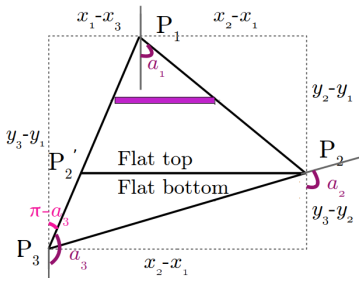
3.1 Line sweep triangle fill

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. 17. 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. 17):

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

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

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

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

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 6 Triangle fill by line sweep pseudocode.

procedure TRIANGLEFILLLINESWEEP(x_1, y_1, x_2, y_3, x_3, y_3) ▷ Assuming $y_1 < y_2 < y_3$
  $m_{inv,13} \leftarrow \frac{x3-x_1}{y_3-y_1}$
  $m_{inv,12} \leftarrow \frac{x2-x_1}{y_2-y_1}$
  $m_{inv,23} \leftarrow \frac{x3-x_2}{y_3-y_2}$
  $x_l \leftarrow x_1$
  $x_r \leftarrow x_1$
  for $y = y_1 . . y_2$ **do** ▷ Flat top triangle
    for $y = \text{int}(x_l) . . \text{int}(x_r)$ **do** ▷ int denotes the casting from float to int
      drawPixel(x, y)
    $x_l \leftarrow x_l + m_{inv,13}$
    $x_r \leftarrow x_r + m_{inv,12}$
  for $y = y_2 . . y_3$ **do** ▷ Flat bottom
    for $y = \text{int}(x_l) . . \text{int}(x_r)$ **do**
      drawPixel(x, y)
    $x_l \leftarrow x_l + m_{inv,13}$
    $x_r \leftarrow x_r + m_{inv,23}$

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.

3.2 Triangle fill by interior test

3.2.1 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 3.1** (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:*
>
> $$\boldsymbol{a}^\perp = (-a_y, a_x) \tag{3.1}$$

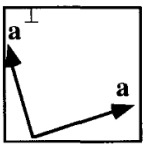
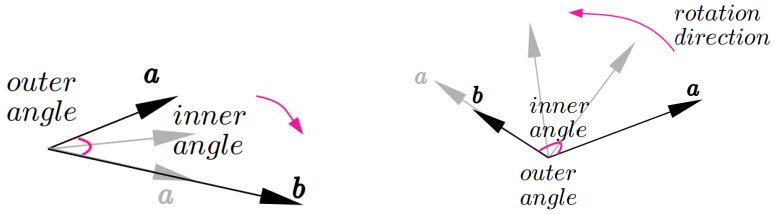


Fig. 18. Vector **a** and its "perp" (**a**) rotated by 90° ccw.

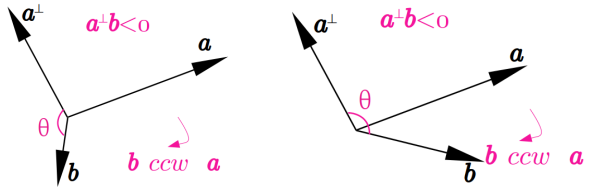
The perp vector comes in handy when we want to determine the relative orientation of two vectors, e.g. whether **a** is cw or ccw from **b**. But first, it's imortant to clarify what is meant by cw and ccw. By saying that **b** is cw of **a**, it is implied that to rotate **b** by the *inner* (smaller) 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. 19. 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. 20).

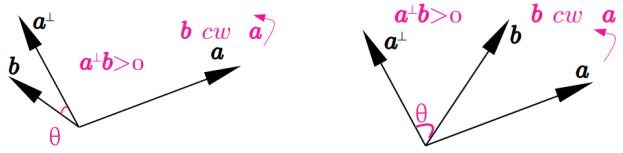


(a) $\mathbf{a}^\perp\mathbf{b} < 0$ when b is ccw from **a** and the angle between **a**, **b** is obtuse.

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

Fig. 20. $\mathbf{a}^\perp\mathbf{b} < 0$ 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. 20).



(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. 21. $\mathbf{a}^\perp\mathbf{b} < 0$ 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 3.2 (perp dot product). *The perp dot product between* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *is defined as:*

$$pdot(\boldsymbol{a},\boldsymbol{b}) = \boldsymbol{a}^\perp\boldsymbol{b} = a_x b_y - a_y b_x \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} \tag{3.2}$$

, 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 3.1 (properties perp product).

$$\boldsymbol{a}^{\perp}\boldsymbol{b} > 0 \quad \text{if} \quad \boldsymbol{b} \quad \text{ccw} \quad \text{from} \quad \boldsymbol{a} \tag{3.3}$$

$$\boldsymbol{a}^{\perp}\boldsymbol{b} < 0 \quad \text{if} \quad \boldsymbol{b} \quad \text{cw} \quad \text{from} \quad \boldsymbol{a} \tag{3.4}$$

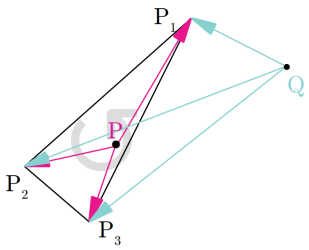
$$\boldsymbol{a}^{\perp}\boldsymbol{b} = 0 \quad \text{if} \quad \boldsymbol{b} = \lambda\boldsymbol{a}, \quad \lambda \in \mathbb{R} \tag{3.5}$$

3.2.2 Stating the interior test

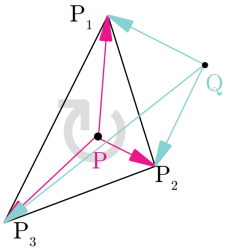
Using Cor. **??**, we can formulate whether a 2D point P is inside or outside a triangle $\Delta(P_1P_2P_3)$. We can use the perp dot product of vectors $\overrightarrow{PP_1}$, $\overrightarrow{PP_2}$, and $\overrightarrow{PP_3}$ to deduce whether P is inside or outside of the triangle, given that we 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. 22. 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. (3.4), Eq. (3.3) and Fig. 22 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 7 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

3.2.3 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. 17) and apply the interior test (Alg. 7) to each. Below is the basic pseudocode, without any cache misses/ hits considered.

Algorithm 8 Triangle fill by interior test (see Alg. 7 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. 6 (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.

4 Circle rasterisation (midpoint algorithm)

4.1 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. 23). 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 $SE(x_p + 1, y_p - 1)$.

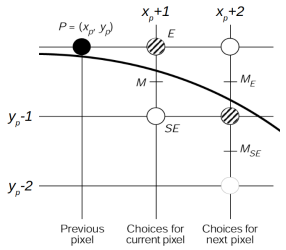


Fig. 23. The two possible choices for the next pixel when drawing a circle at octant 2; E and SE. For the above curve, we select E and SE as the next two pixels (striped).

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. 23:

- 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.4) that the new value of the decision variable is:

$$D_E = D + (2x_p + 3) \tag{4.1}$$

  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.4) that the decision variable D is incremented by $2x_p-2y_p+5$:

$$D_{SE} = D + (2x_p - 2y_p + 5) \tag{4.2}$$

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.

$$D_0 = 1 - r \tag{4.3}$$

Using the update rules in Eq. (4.1), Eq. (4.2), Eq. (4.3), we can draw a circular arc on the 2nd octant and mirror it on the rest to draw a full circle.

4.2 Pseudocode and implementation of the algorithm

Algorithm 9 Midpoint algorithm for circle drawing at a point (x_0, y_0) with radius r.

```
1:  procedure MIDPOINTALGORITHM(x0, y0, r)
2:      x ← 0
3:      y ← r
4:      D ← 1 − r                                   ▷ Decision variable
5:      do
6:          putPixel(x + x0, y + y0)                ▷ Octant 2; draw all octants to fill the circle
7:          putPixel(y + x0, x + y0)                ▷ Octant 1
8:          putPixel(−x + x0, y + y0)               ▷ Octant 3
9:          putPixel(−y + x0, x + y0)               ▷ Octant 4
10:         putPixel(−y + x0, −x + y0)              ▷ Octant 5
11:         putPixel(−x + x0, −x + y0)              ▷ Octant 6
12:         putPixel(x + x0, −y + y0)               ▷ Octant 7
13:         putPixel(y + x0, −x + y0)               ▷ Octant 8
14:         if D < 0 then
15:             D ← D + 2x + 3
16:         else
17:             y ← y − 1
18:             D ← D + 2x − 2y + 5
19:         x ← x + 1
20:     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 10 Bresenham's full line drawing.

```
1:  procedure FIND-OCTANT(x1, y1, x2, y2)                          ▷ See Fig. 1
2:      m ← (y2 − y1) / (x2 − x1)
3:      if x1 ≤ x2 and 0 ≤ m ≤ 1 then
4:          return 0                                               ▷ 1st
5:      else if y1 ≤ y2 and m > 1 then
6:          return 1                                               ▷ etc.
7:      else if y1 ≤ y2 and m < −1 then
8:          return 2
9:      else if x2 ≤ x1 and 0 ≥ m ≥ −1 then
10:         return 3
11:     else if x2 ≤ x1 and 0 < m ≤ 1 then
12:         return 4
13:     else if y2 ≤ y1 and m > 1 then
14:         return 5
15:     else if y2 ≤ y1 and m < −1 then
16:         return 6
17:     else if x1 ≤ x2 and −1 ≤ m ≤ 0 then
18:         return 7
19:     else                                                       ▷ x1 = x2, vertical line
20:         return 8
21:
22: procedure BRESENHAM(x1, y1, x2, y2)
23:     Δx ← x2 − x1
24:     Δy ← y2 − y1
25:     ϵ ← 0
26:     oct ← Find-Octant(x1, y1, x2, y2)
27:     if oct = 0 then                                            ▷ 0 to 45 degrees with x axis
28:         y ← y1
29:         for x = x1..x2 do
30:             Draw-Pixel(x, y)
31:             ϵ ← ϵ + Δy
32:             if 2ϵ ≥ Δx then
33:                 ϵ ← ϵ − Δx
34:                 y ← y + 1
35:     else if oct = 1 then                                       ▷ 45 to 90
36:         x ← x1
37:         for y = y1..y2 do
38:             Draw-Pixel(x, y)
39:             ϵ ← ϵ + Δx
40:             if 2ϵ ≥ Δy then
41:                 ϵ ← −Δy
42:                 x ← x + 1
43:     else if oct = 2 then                                       ▷ 90 to 135
44:         x ← x1
45:         for y = y1..y2 do
46:             Draw-Pixel(x, y)
47:             ϵ ← ϵ − Δx
48:             if  2ϵ ≥ Δ then
49:                 ϵ ← ϵ − Δy
50:                 x ← x − 1
51:     else if oct = 3 then                                       ▷ 135 to 180
52:         y ← y1
53:         for x = x1..x2 do
54:             Draw-Pixel(x, y)
```

Algorithm 11 Bresenham's full line drawing – cont'ed

```
1:  ϵ ← ϵ + Δx
2:  if 2ϵ ≥ −Δx then
3:      ϵ ← ϵ + Δx
4:      y ← y + 1
5:  else if oct = 4 then                                   ▷ 180 to 215
6:      y ← y1
7:      for x = x1..x2 do
8:          Draw-Pixel(x, y)
9:          ϵ ← ϵ − Δy
10:         if 2ϵ ≥ −Δx then
11:             ϵ ← ϵ + Δx
12:             y ← y − 1
13: else if oct = 5 then                                   ▷ 215 to 270
14:     x ← x1
15:     for y = y1..y2 do
16:         Draw-Pixel(x, y)
17:         ϵ ← ϵ − Δx
18:         if 2ϵ ≥ −Δy then
19:             ϵ ← ϵ − Δy
20:             x ← x − 1
21: else if oct = 6 then                                   ▷ 270 to 315
22:     x = x1
23:     for y = y1..y2 do
24:         Draw-Pixel(x, y)
25:         ϵ ← ϵ + Δx
26:         if 2ϵ ≥ −Δy then
27:             ϵ ← ϵ + Δy
28:             x ← x + 1
29: else if oct = 7 then                                   ▷ 315 to 360
30:     x ← x1
31:     for y = y1..y2 do
32:         Draw-Pixel(x, y)
33:         ϵ ← ϵ + Δx
34:         if 2ϵ ≥ −Δy then
35:             ϵ ← ϵ + Δy
36:             x ← x + 1
37: else if oct = 8 then                                   ▷ Vertical line
38:                                  ▷ Draw a vertical at line at x1 between y1, y2
```

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).

```
/*
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 Parametric form of straight line

LEMMA A.1. *Given two points P_0, P_1 and a parameter $t \in \mathbb{R}$, a straight line can be parametrised as:*

$$P(t) = (1-t)P_0 + tP_1 \tag{A.1}$$

, where $P = (x, y)$, $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$.

Unwrapping this equation in each dimension, the points $(x(t), y(t))$ of line P_0P_1 are given by

$$x(t) = (1-t)x_0 + tx_1$$

$$y(t) = (1-t)y_0 + ty_1$$

If we restrict $t \in [0,1]$, then we obtain the line segment bar P_0P_1, i.e. as t changes from 0 to 1 we move between P_0 and P_1. For $t = 0$, P coincides with P_0, and for $t = 1$ with P_1.

Proof. As the figure below shows, any point between P_0 and P_1 can be reached from P_0 by adding $t(\mathbf{p}_1 - \mathbf{p}_0)$ to it, where t is a normalised scalar indicating where p falls along line $\mathbf{p}_0\mathbf{p}_1$ ($t = 0$ at $\mathbf{p}_0$, $t = 1$ at $\mathbf{p}_1$). Therefore:

$$\mathbf{p} = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1$$

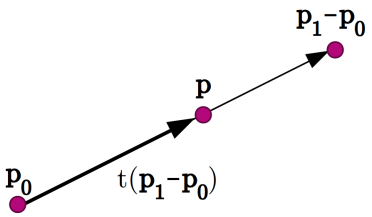


Fig. 24. How a point $\mathbf{p}$ along a straight segment defined by $\mathbf{p}_1, \mathbf{p}_2$ can be parametrised.

□

A.4 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:

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