A Bitmapper's Geometry

An introduction to basic bitmap mathematics and algorithms

epilys

November 18, 2021

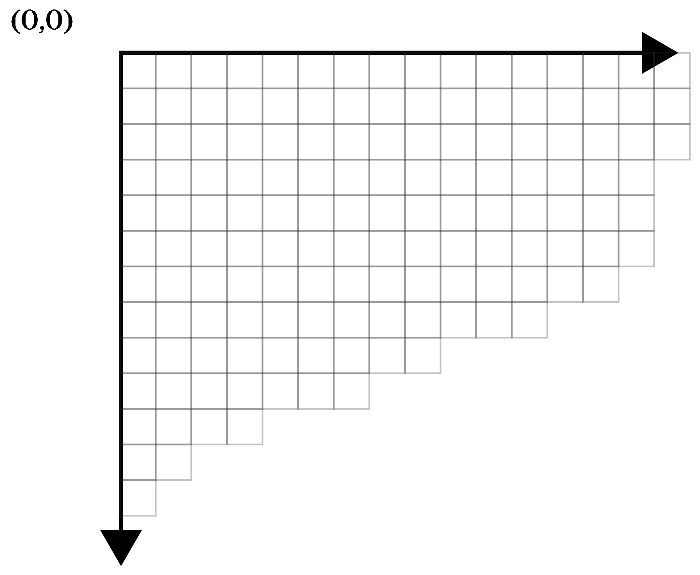
Contents

Chapter 1

Introduction

The data structures we're going to use is *Point* and *Image*. *Image* represents a bitmap, although we will use full RGB colors for our points therefore the size of a pixel in memory will be u8 instead of 1 bit.

We will work on the cartesian grid representing the framebuffer that will show us the pixels. The *origin* of this grid (i.e. the center) is at $(0, 0)$.



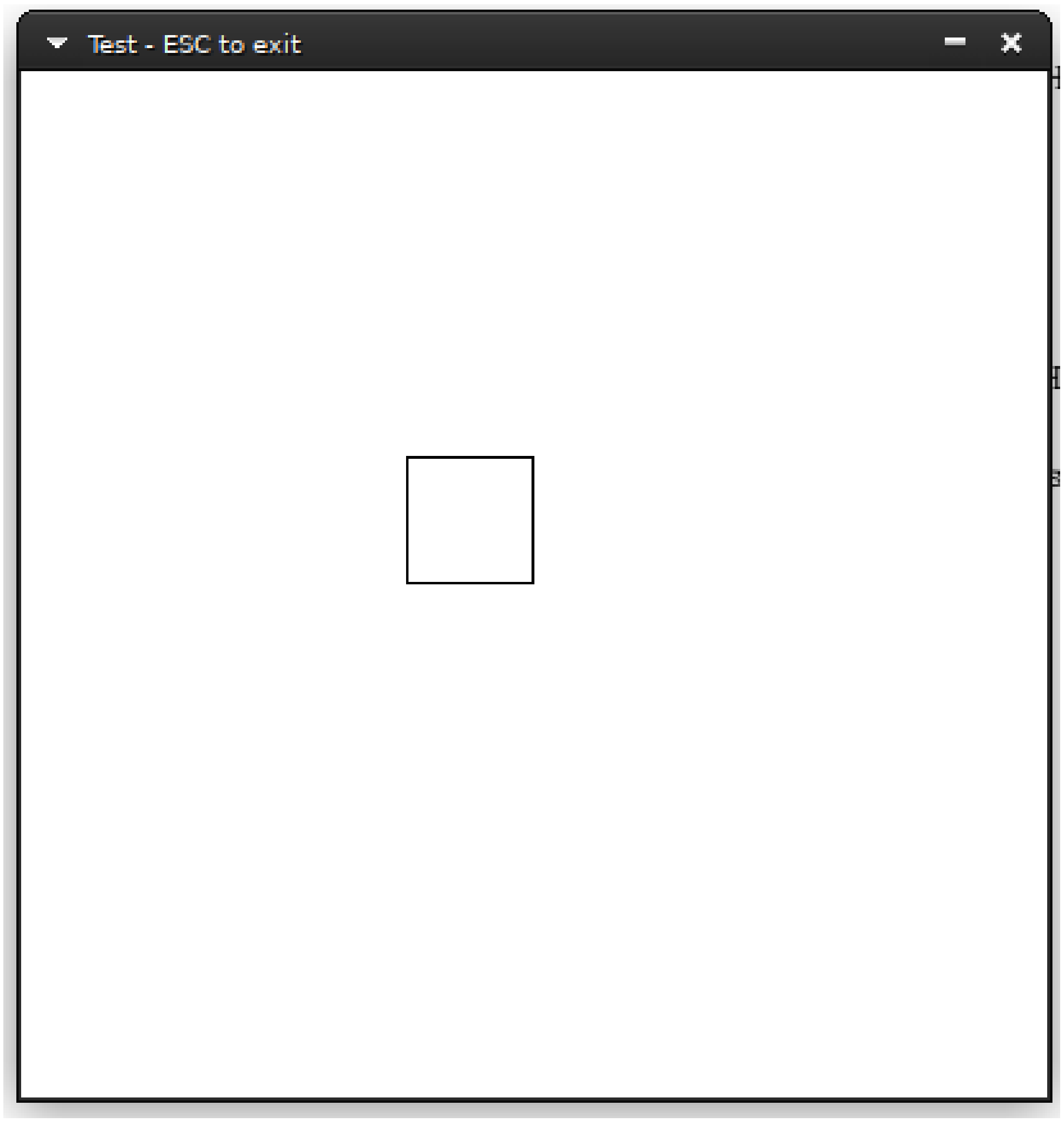
We will represent points as pairs of signed integers. When actually drawing them though, negative values and values outside the window's geometry will be ignored (clipped).

```
pub type Point = (i64, i64);

pub const fn from_u8_rgb(r: u8, g: u8, b: u8) -> u32 {
    let (r, g, b) = (r as u32, g as u32, b as u32);
    (r << 16) | (g << 8) | b
}
pub const AZURE_BLUE: u32 = from_u8_rgb(0, 127, 255);
pub const RED: u32 = from_u8_rgb(157, 37, 10);
pub const WHITE: u32 = from_u8_rgb(255, 255, 255);
pub const BLACK: u32 = 0;

pub struct Image {
    pub bytes: Vec<u32>,
    pub width: usize,
    pub height: usize,
    pub x_offset: usize,
    pub y_offset: usize,
}

impl Image {
    pub fn new(width: usize,
        height: usize,
        x_offset: usize,
        y_offset: usize) -> Self;
    pub fn draw(&self,
        buffer: &mut Vec<u32>,
        fg: u32,
        bg: Option<u32>,
        window_width: usize);
    pub fn draw_outline(&mut self);
    pub fn clear(&mut self);
    pub fn plot(&mut self, x: i64, y: i64);
    pub fn get(&mut self, x: i64, y: i64) -> u32;
    pub fn plot_ellipse(
        &mut self,
        (xm, ym): (i64, i64),
        (a, b): (i64, i64),
        quadrants: [bool; 4],
        _wd: f64,
    );
    pub fn plot_line_width(&mut self,
            point_a: Point,
            point_b: Point,
            wd: f64);
    pub fn flood_fill(&mut self, mut x: i64, y: i64);
}
```

A way to display an *Image* is to use the `minifb` crate which allows you to create a window and draw pixels directly on it. Here's how you could set it up:

```
use bitmappers_geometry::*;
use minifb::{Key, Window, WindowOptions};

const WINDOW_WIDTH: usize = 400;
const WINDOW_HEIGHT: usize = 400;

fn main() {
    let mut buffer: Vec<u32> = vec![WHITE; WINDOW_WIDTH * WINDOW_HEIGHT];
    let mut window = Window::new(
        "Test - ESC to exit",
        WINDOW_WIDTH,
        WINDOW_HEIGHT,
        WindowOptions {
            title: true,
            //borderless: true,
            //resize: false,
            //transparency: true,
            ..WindowOptions::default()
        },
    )
    .unwrap();

    // Limit to max ~60 fps update rate
    window.limit_update_rate(Some(std::time::Duration::from_micros(16600)));

    let mut image = Image::new(50, 50, 150, 150);
    image.draw_outline();
    image.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);

    while window.is_open()
         && !window.is_key_down(Key::Escape)
         && !window.is_key_down(Key::Q) {
        window
            .update_with_buffer(&buffer, WINDOW_WIDTH, WINDOW_HEIGHT)
            .unwrap();
        let millis = std::time::Duration::from_millis(100);
        std::thread::sleep(millis);
    }
}
```

Running this will show you something like this:



1.1 Loading `xbm` files in Rust

`xbm` files are C source code files that contain the pixel information for an image as macro definitions for the dimensions and a static `char` array for the pixels, with each bit column representing a pixel. If the width dimension doesn't have 8 as a factor, the remaining bit columns are left blank/ignored.

They used to be a popular way to share user avatars in the old internet and are also good material for us to work with, since they are small and numerous. The following is such an image:

First, let's define a way to convert bit information to a byte vector:

```
pub fn bits_to_bytes(bits: &[u8], width: usize) -> Vec<u32> {
    let mut ret = Vec::with_capacity(bits.len() * 8);
    let mut current_row_count = 0;
    for byte in bits {
        for n in 0..8 {
            if byte.rotate_right(n) & 0x01 > 0 {
                ret.push(BLACK);
            } else {
                ret.push(WHITE);
            }
            current_row_count += 1;
            if current_row_count == width {
                current_row_count = 0;
                break;
            }
        }
    }
    ret
}
```

Then, we can convert the xbm file from C to Rust with the following transformations:

```
#define news_width 48
#define news_height 48
static char news_bits[] = {
```

to

```
const NEWS_WIDTH: usize = 48;
const NEWS_HEIGHT: usize = 48;
const NEWS_BITS: &[u8] = &[
```

And replace the closing } with].

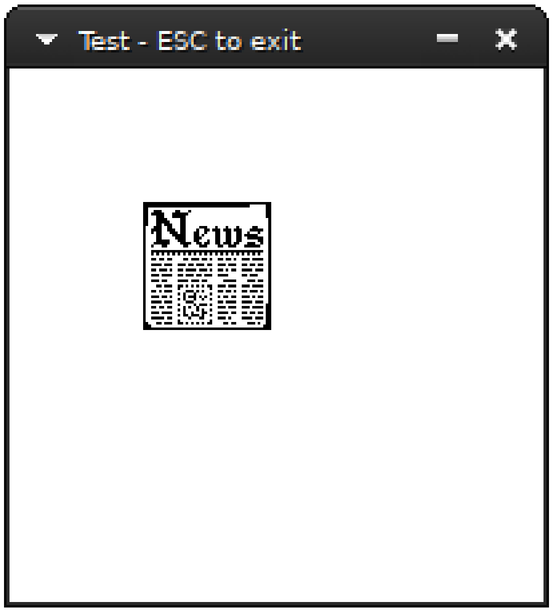
We can then include the new file in our source code:

```
include!("news.xbm.rs");
```

load the image:

```
let mut image = Image::new(NEWS_WIDTH, NEWS_HEIGHT, 25, 25);
image.bytes = bits_to_bytes(NEWS_BITS, NEWS_WIDTH);
```

and finally run it:

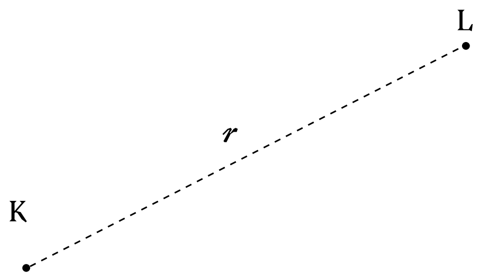


Part I

Points and Lines

Chapter 2

Distance between two points



Given two points, K and L, an elementary application of Pythagoras' Theorem gives the distance between them as

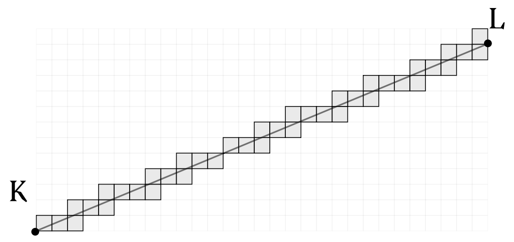
$$r = \sqrt{(x_L - x_K)^2 + (y_L - y_K)^2} \tag{2.1}$$

which is simply coded:

```
pub fn distance_between_two_points(p_k: Point, p_l: Point) -> f64 {
    let (x_k, y_k) = p_k;
    let (x_l, y_l) = p_l;
    let xlk = x_l - x_k;
    let ylk = y_l - y_k;
    f64::sqrt((xlk*xlk + ylk*ylk) as f64)
}
```

2.1 Drawing a line segment from its two end-points

For any line segment with any slope, pixels must be matched with the infinite amount of points contained in the segment. As shown in the following figure, a segment *touches* some pixels; we could fill them using an algorithm and get a bitmap of the line segment.



The algorithm presented here was first derived by Bresenham. In the *Image* implementation, it is used in the `plot_line_width` method.

Chapter 3

Equations of a line

Chapter 4

The parametric form

Chapter 5

Angle between two lines

Chapter 6

Intersection of two lines

Chapter 7

Line through two points

Chapter 8

Line equidistant from two points

Chapter 9

Normal to a line through a point

Part II

Points Lines and Circles

Chapter 10

Equations of a Circle

Part III

Points line segments and Arcs

Part IV

Curves other than circles

Part V

Points, lines and planes

Part VI

Vectors, matrices and transformations

Chapter 11

Rotation of a bitmap

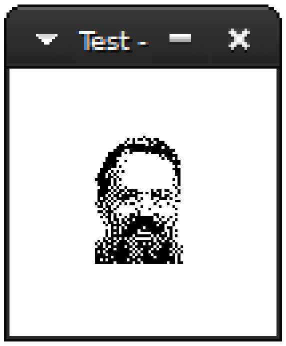
$$p' = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix}$$

$$c = cos\theta, s = sin\theta, x_{p'} = x_p c - y_p s, y_{p'} = x_p s + y_p c.$$

Let's load an `xface`. We will use `bits_to_bytes` (See Introduction).

```
include!("dmr.rs");

const WINDOW_WIDTH: usize = 100;
const WINDOW_HEIGHT: usize = 100;

let mut image = Image::new(DMR_WIDTH, DMR_HEIGHT, 25, 25);
image.bytes = bits_to_bytes(DMR_BITS, DMR_WIDTH);
```



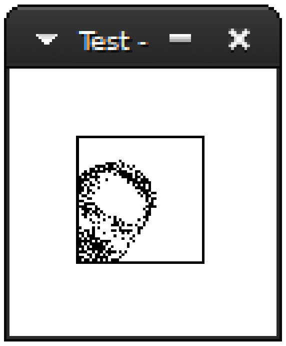
This is the `xface` of `dmr`. Instead of displaying the bitmap, this time we will rotate it 0.5 radians. Setup our image first:

```
let mut image = Image::new(DMR_WIDTH, DMR_HEIGHT, 25, 25);
image.draw_outline();
let dmr = bits_to_bytes(DMR_BITS, DMR_WIDTH);
```

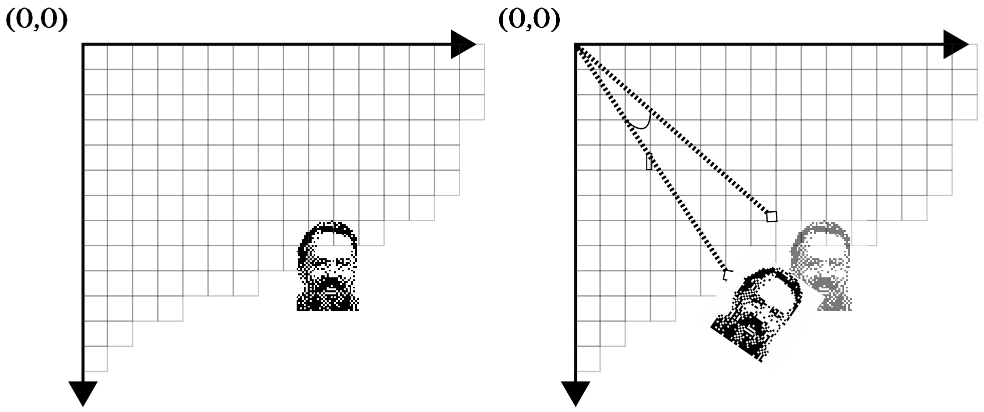
And then, loop for each byte in `dmr`'s face and apply the rotation transformation.

```
let angle = 0.5;

let c = f64::cos(angle);
let s = f64::sin(angle);

for y in 0..DMR_HEIGHT {
    for x in 0..DMR_WIDTH {
        if dmr[y * DMR_WIDTH + x] == BLACK {
            let x = x as f64;
            let y = y as f64;
            let xr = x * c - y * s;
            let yr = x * s + y * c;
            image.plot(xr as i64, yr as i64);
        }
    }
}
```

The result:



We didn't mention in the beginning that the rotation has to be relative to a *point* and the given transformation is relative to the *origin*, in this case the upper left corner $(0, 0)$. So `dmr` was rotated relative to the origin:



Usually, we want to rotate something relative to itself. The right point to choose is the *centroid* of the object.

If we have a list of n points, the centroid is calculated as:

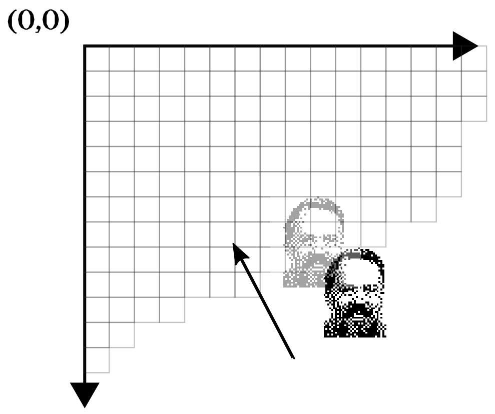
$$x_c = \frac{1}{n} \sum_{i=0}^{n} x_i$$

$$y_c = \frac{1}{n} \sum_{i=0}^{n} y_i$$

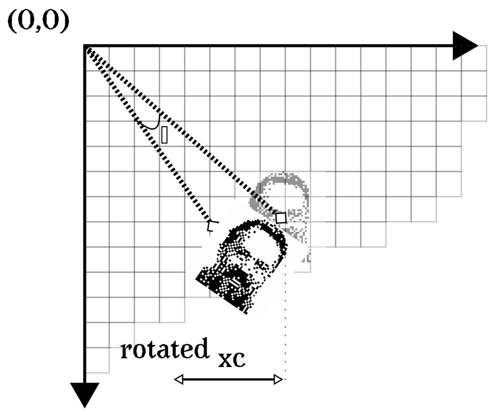
Since in this case we have a rectangle, the centroid has coordinates of half the width and half the height.

By subtracting the centroid from each point before we apply the transformation and then adding it back after we get what we want:

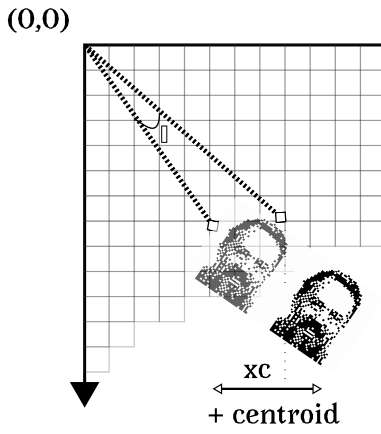
Here's it visually: First subtract the center point.



Then, rotate.

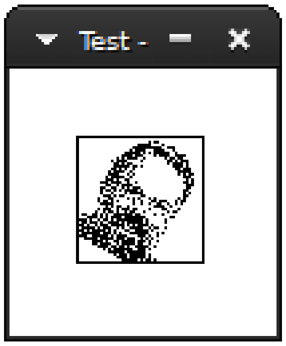


And subtract back to the original position.



In code:

```
let center_point = ((DMR_WIDTH/2) as i64, (DMR_HEIGHT/2) as i64);
for y in 0..DMR_HEIGHT {
    for x in 0..DMR_WIDTH {
        if dmr[y * DMR_WIDTH + x] == BLACK {
            let x = (x as i64 -center_point.0) as f64;
            let y = (y as i64 -center_point.1) as f64;
            let xr = x * c - y * s;
            let yr = x * s + y * c;
            image.plot(xr as i64+center_point.0,
                       yr as i64 + center_point.1);
        }
    }
}
```

The result: 

Chapter 12

Rotation of a bitmap by parallel re-cusive subdivision

Chapter 13

Magnification

Part VII

Flood filling

Part VIII

Areas

Part IX

Volumes

1

[redacted]

[redacted]