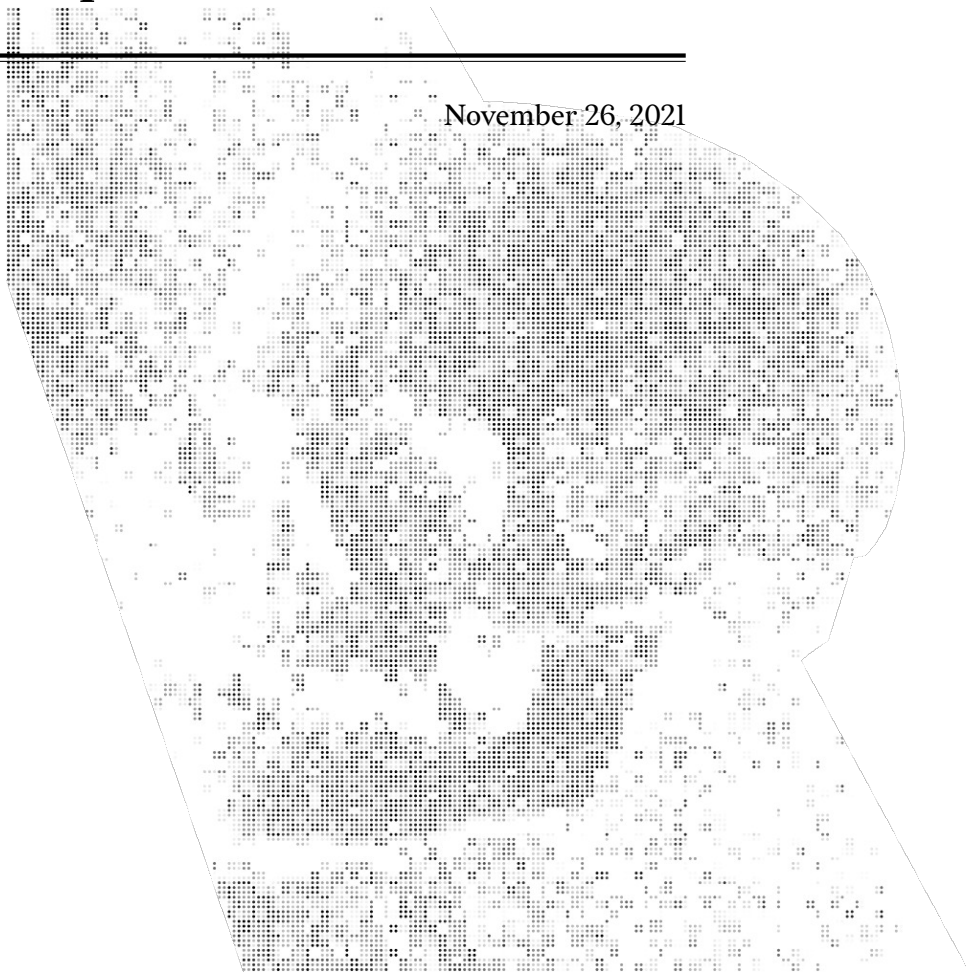

A Bitmapper's Companion

an introduction to basic **bitmap**
mathematics and algorithms with code
samples in **Rust**

epilys

November 26, 2021



Manos Pitsidianakis (epilys)
<https://nessuent.xyz>
<https://github.com/epilys>
epilys@nessuent.xyz

All non-screenshot figures were generated by hand in Inkscape unless otherwise stated.

The skull in the cover is a transformed bitmap of the skull in the 1533 oil painting by Hans Holbein the Younger, *The Ambassadors*, which features a floating distorted skull rendered in anamorphic perspective.

A Bitmapper's Companion, 2021

Special Topics ► Computer Graphics ► Programming
006.6'6-dc20

Copyright © 2021 by Emmanouil Pitsidianakis

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

The source code for this work is available under the GNU GENERAL PUBLIC LICENSE version 3 or later. You can view it, study it, modify it for your purposes as long as you respect the license if you choose to distribute your modifications.

The source code is available here

<https://github.com/epilys/bitmappers-companion>

Contents

I	Introduction	6
1	Data representation	7
2	Displaying pixels to your screen	10
3	Bits to byte pixels	13
4	Real pixels to byte pixels	14
5	Loading xbm files in Rust	16
II	Points and Lines	18
6	Distance between two points	19
7	Distance from a point to a line	20
8	Equations of a line	22
8.1	The common form	22
8.2	The parametric form	24
9	Angle between two lines	26
10	Intersection of two lines	28
11	Line through two points	30
12	Line equidistant from two points	32
13	Normal to a line through a point	34

<i>CONTENTS</i>	4
III Points, Lines and Circles	36
14 Equations of a Circle	39
15 Bounding Circle	41
IV Points, Line Segments and Arcs	43
16 Drawing a line segment from its two endpoints	44
17 Drawing Line segments With Width	47
18 Intersection of two line segments	51
18.1 <i>Fast</i> intersection of two line segments	51
V Curves other than circles	54
19 Parametric elliptical arcs	55
VI Points, Lines and Planes	57
20 Union, intersection and difference of polygons	58
21 Centroid of polygon	60
VII Vectors, matrices and transformations	62
22 Rotation of a bitmap	63
22.1 Fast 2D Rotation	68
23 90° Rotation of a bitmap by parallel recursive subdivision	70
24 Magnification/Scaling	72
24.1 Smoothing enlarged bitmaps	74
24.2 Stretching lines of bitmaps	76

<i>CONTENTS</i>	5
VIII Areas	78
25 Flood filling	81
IX Advanced	86
25.1 Faster Drawing a line segment from its two endpoints using Symmetry	87
26 Joining the ends of two wide line segments together	89
27 Composing monochrome bitmaps with separate alpha channel data	91
28 Orthogonal connection of two points	93
29 Join segments with round corners	95
30 Faster line clipping	97
31 Space-Filling Curves	99



Part I

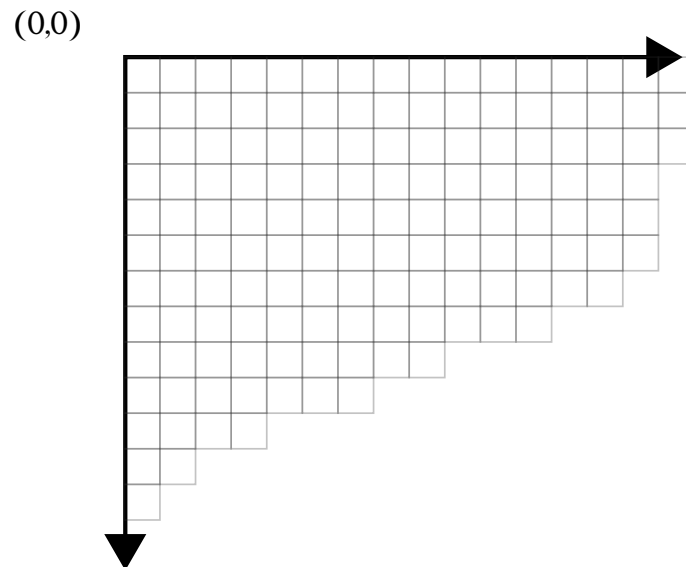
Introduction

Chapter 1

Data representation

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);
}
```

Chapter 2

Displaying pixels to your screen

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_companion::*;
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()
        }
    );
```

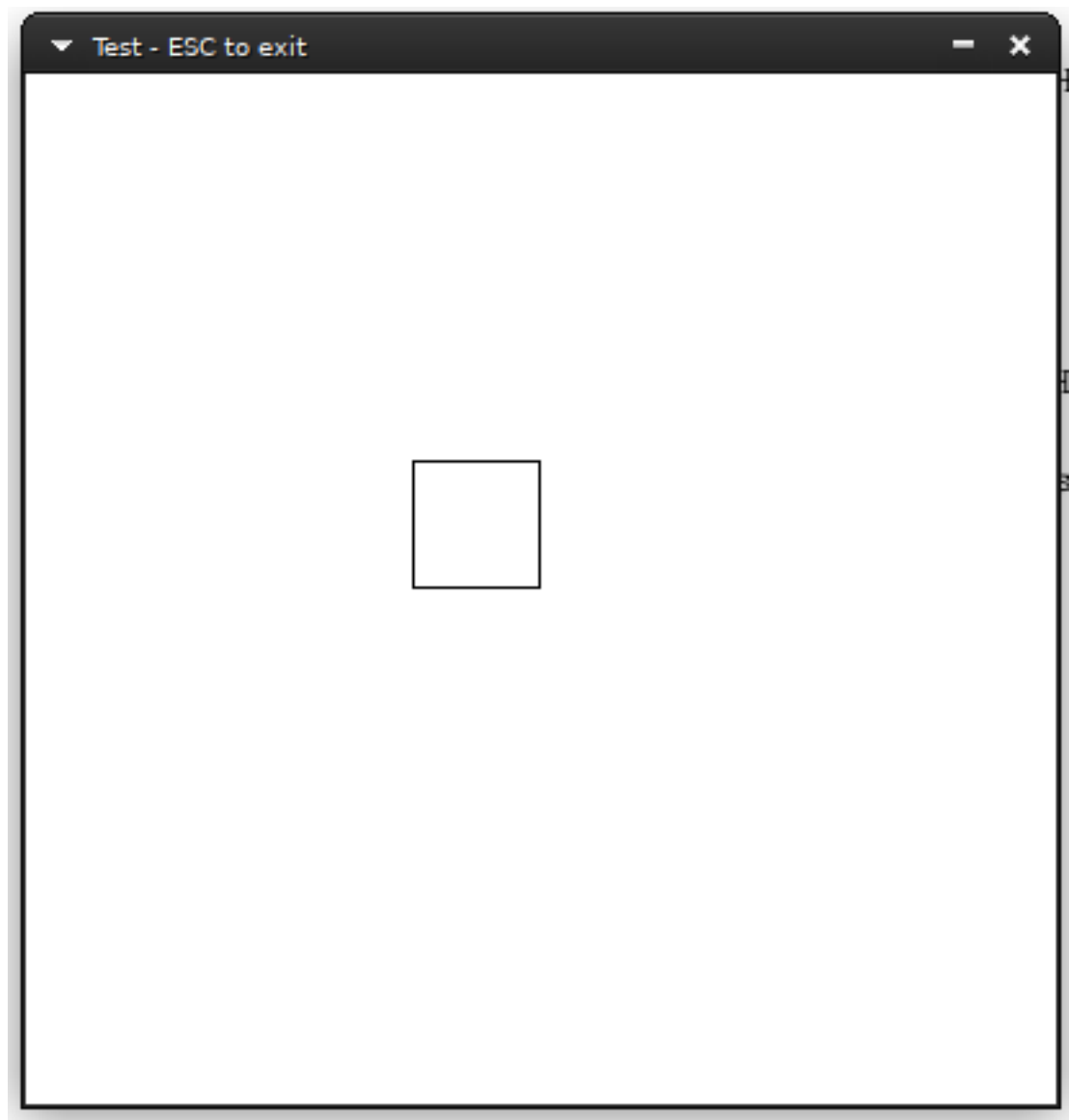
```
        },
    )
    .unwrap();

    // 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:



Chapter 3

Bits to byte pixels

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
}
```

Chapter 4

Real pixels to byte pixels





Chapter 5

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:



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 `]`.

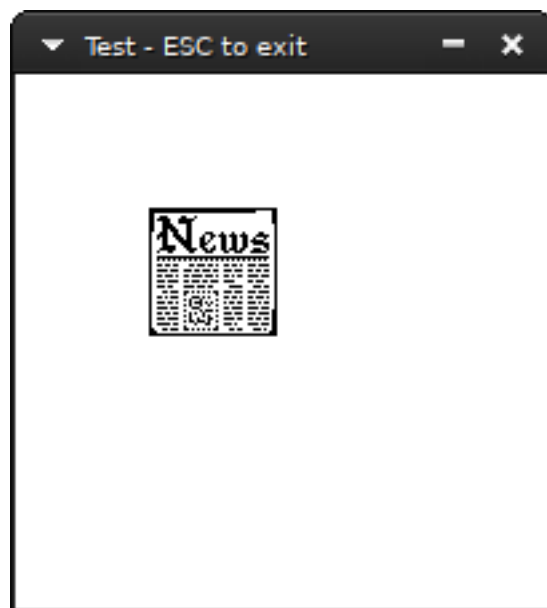
We can then include the new file in our source code:

```
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:

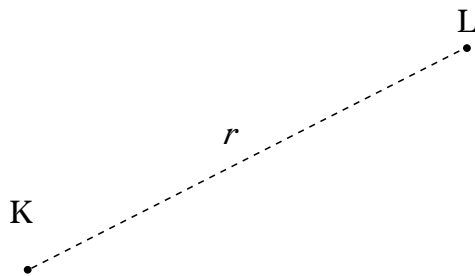


Part II

Points and Lines

Chapter 6

Distance between two points



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

$$r = \sqrt{(x_L - x_K)^2 + (y_L - y_K)^2} \quad (6.1)$$

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)  
}
```

Chapter 7

Distance from a point to a line



Chapter 8

Equations of a line

8.1 The common form

3



8.2 The parametric form



Chapter 9

Angle between two lines



Chapter 10

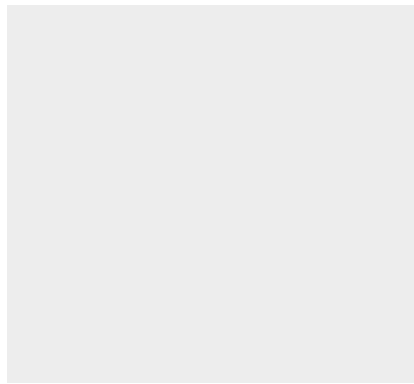
Intersection of two lines

6

[REDACTED]

Chapter 11

Line through two points



7

Figure 11.1:

8

It seems sufficient, given the coordinates of two points M, N , to calculate a, b and c to form a line equation:

$$ax + by + c = 0$$

If the two points are not the same, they necessarily form such a line. To get there, we start from expressing the line as parametric over t : at $t = 0$ it's at point M and at $t = 1$ it's at point N :

$$c = c_M + (c_N - c_M)t, t \in R, c \in \{x, y\}$$

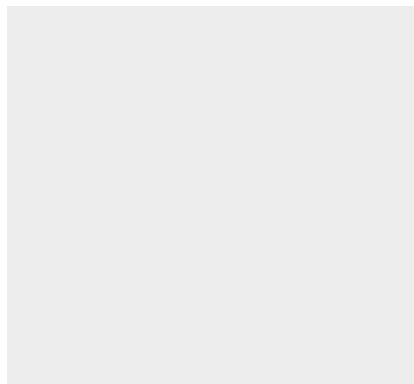
Substituting t in one of the equations we get:

$$(y_M - y_N)x + (x_N - x_M)y + (x_M y_N - x_N y_M) = 0$$

Which is what we were after. We finish by normalising what we found with $\frac{1}{\sqrt{a^2 + b^2}}$:

Chapter 12

Line equidistant from two points



9

Figure 12.1:

10

Let's name this line L . From the previous chapter we know how to get the line that's created by the two points M and N . If only we knew how to get a perpendicular line over the midpoint of a line segment!

Thankfully that midpoint also satisfies L 's equation, $ax + by + c$. The midpoint's coordinates are intuitively:

$$\left(\frac{x_M + x_N}{2}, \frac{y_M + y_N}{2}\right)$$

Putting them into the equation we can generate a triple of (a', b', c') and

then normalize it to get L .

Chapter 13

Normal to a line through a point

11



Part III

Points, Lines and Circles

12

[REDACTED]

[REDACTED]

Chapter 14

Equations of a Circle



Chapter 15

Bounding Circle

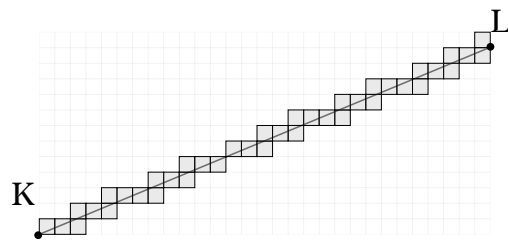
Part IV

Points, Line Segments and Arcs

Chapter 16

Drawing a line segment from its two endpoints

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.

```
pub fn plot_line_width(&mut self, (x1, y1): (i64, i64), (x2, y2): (i64, i64)) {  
    /* Bresenham's line algorithm */  
    let mut d;
```

```

let mut x: i64;
let mut y: i64;
let ax: i64;
let ay: i64;
let sx: i64;
let sy: i64;
let dx: i64;
let dy: i64;

dx = x2 - x1;
ax = (dx * 2).abs();
sx = if dx > 0 { 1 } else { -1 };

dy = y2 - y1;
ay = (dy * 2).abs();
sy = if dy > 0 { 1 } else { -1 };

x = x1;
y = y1;

let b = dx / dy;
let a = 1;
let double_d = (_wd * f64::sqrt((a * a + b * b) as f64)) as i64;
let delta = double_d / 2;

if ax > ay {
    d = ay - ax / 2;
    loop {
        self.plot(x, y);
        if x == x2 {
            return;
        }
        if d >= 0 {
            y = y + sy;

```

```

        d = d - ax;
    }
    x = x + sx;
    d = d + ay;
}
} else {
    d = ax - ay / 2;
    let delta = double_d / 3;
    loop {
        self.plot(x, y);
        if y == y2 {
            return;
        }
        if d >= 0 {
            x = x + sx;
            d = d - ay;
        }
        y = y + sy;
        d = d + ax;
    }
}
}

```

Chapter 17

Drawing Line segments With Width

```
pub fn plot_line_width(&mut self, (x1, y1): (i64, i64),
                      (x2, y2): (i64, i64), _wd: f64) {
    /* Bresenham's line algorithm */
    let mut d;
    let mut x: i64;
    let mut y: i64;
    let ax: i64;
    let ay: i64;
    let sx: i64;
    let sy: i64;
    let dx: i64;
    let dy: i64;

    dx = x2 - x1;
    ax = (dx * 2).abs();
    sx = if dx > 0 { 1 } else { -1 };

    dy = y2 - y1;
    ay = (dy * 2).abs();
    sy = if dy > 0 { 1 } else { -1 };

    x = x1;
```

```

y = y1;

let b = dx / dy;
let a = 1;
let double_d = (_wd * f64::sqrt((a * a + b * b) as f64)) as i64;
let delta = double_d / 2;

if ax > ay {
    d = ay - ax / 2;
    loop {
        self.plot(x, y);
        {
            let total = |_x| _x - (y * dx) / dy + (y1 * dx) / dy - x1;
            let mut _x = x;
            loop {
                let t = total(_x);
                if t < -1 * delta || t > delta {
                    break;
                }
                _x += 1;
                self.plot(_x, y);
            }
            let mut _x = x;
            loop {
                let t = total(_x);
                if t < -1 * delta || t > delta {
                    break;
                }
                _x -= 1;
                self.plot(_x, y);
            }
        }
    }
    if x == x2 {
        return;
    }
}

```



```

    }
    if d >= 0 {
        y = y + sy;
        d = d - ax;
    }
    x = x + sx;
    d = d + ay;
}
} else {
    d = ax - ay / 2;
    let delta = double_d / 3;
    loop {
        self.plot(x, y);
        {
            let total = |_x| _x - (y * dx) / dy + (y1 * dx) / dy - x1;
            let mut _x = x;
            loop {
                let t = total(_x);
                if t < -1 * delta || t > delta {
                    break;
                }
                _x += 1;
                self.plot(_x, y);
            }
            let mut _x = x;
            loop {
                let t = total(_x);
                if t < -1 * delta || t > delta {
                    break;
                }
                _x -= 1;
                self.plot(_x, y);
            }
        }
    }
}

```

```
        if y == y2 {
            return;
        }
        if d >= 0 {
            x = x + sx;
            d = d - ay;
        }
        y = y + sy;
        d = d + ax;
    }
}
```

Chapter 18

Intersection of two line segments

Let points **1** = (x_1, y_1) , **2** = (x_2, y_2) , **3** = (x_3, y_3) and **4** = (x_4, y_4) and **1,2**, **3,4** two line segments they form. We wish to find their intersection:

First, get the equation of line L_{12} and line L_{34} from chapter FIXME.

Substitute points **3** and **4** in equation L_{12} to compute $r_3 = L_{12}(\mathbf{3})$ and $r_4 = L_{12}(\mathbf{4})$ respectively.

If $r_3 \neq 0$, $r_4 \neq 0$ and $\text{sgn}(r_3) == \text{sign}(r_4)$ the line segments don't intersect, so stop.

In L_{34} substitute point **1** to compute r_1 , and do the same for point **2**.

If $r_1 \neq 0$, $r_2 \neq 0$ and $\text{sgn}(r_1) == \text{sign}(r_2)$ the line segments don't intersect, so stop.

At this point, L_{12} and L_{34} either intersect or are equivalent. Find their intersection point. (Refer to FIXME.)

18.1 *Fast* intersection of two line segments



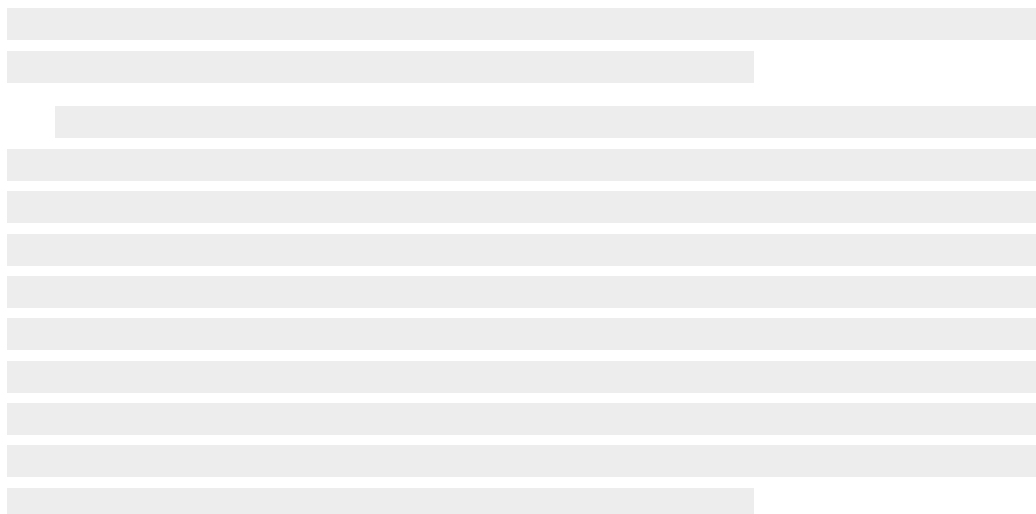
15

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]



Part V

Curves other than circles

Chapter 19

Parametric elliptical arcs



Part VI

Points, Lines and Planes

Chapter 20

Union, intersection and difference of polygons

[REDACTED]

[REDACTED]

[REDACTED]

Chapter 21

Centroid of polygon

18



Part VII

Vectors, matrices and transformations

Chapter 22

Rotation of a bitmap

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

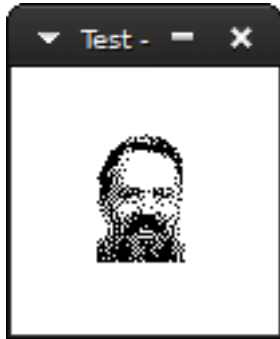
$$c = \cos\theta, s = \sin\theta, x_{p'} = x_p c - y_p s, y_{p'} = x_p s + y_p c.$$

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);
```

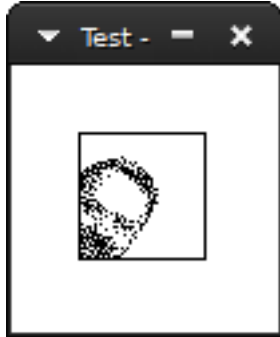
And then, loop for each byte in dmr's face and apply the rotation transformation.

```
let angle = 0.5;

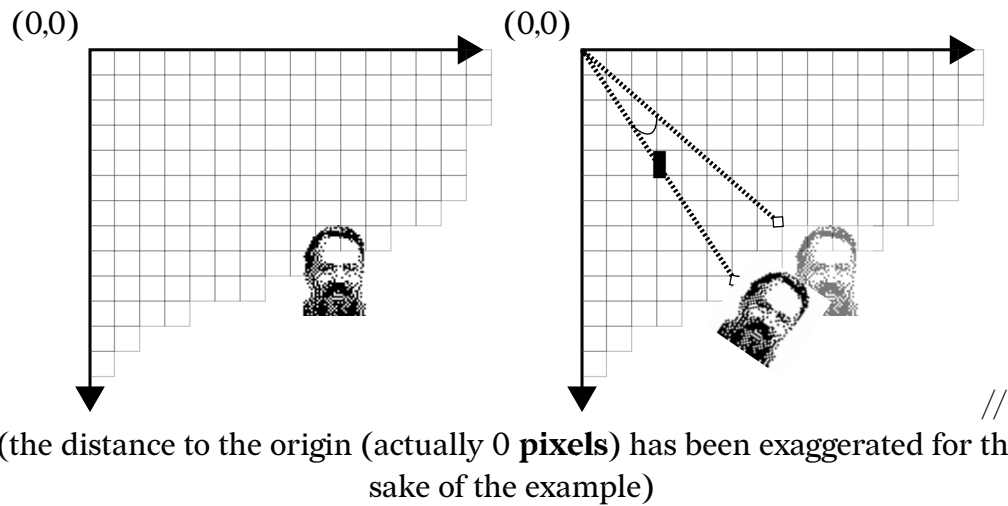
let c = f64::cos(angle);
let s = f64::sin(angle);

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:

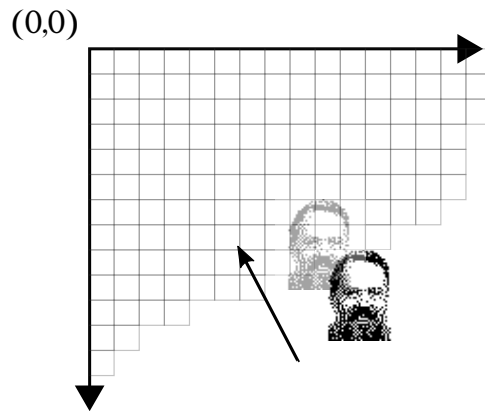
$$x_c = \frac{1}{n} \sum_{i=0}^n x_i$$

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

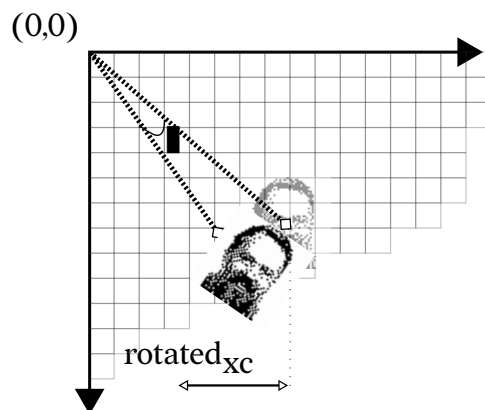
Since in this case we have a rectangle, the centroid has coordinates of half the width and half the height.

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

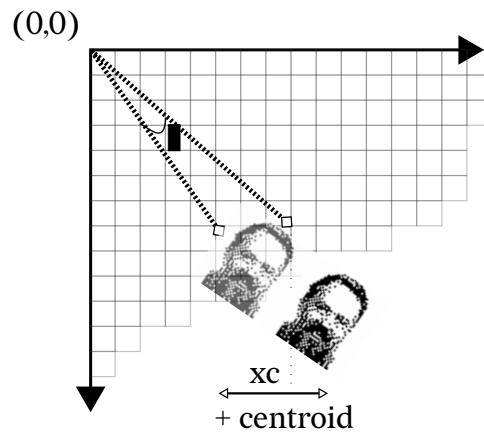
Here's it visually: First subtract the center point.



Then, rotate.



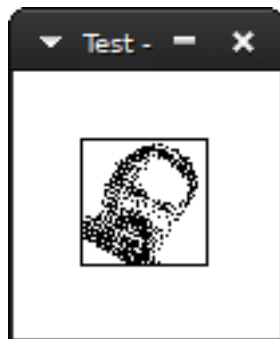
And subtract back to the original position.



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:



22.1 Fast 2D Rotation

19

[The following text is redacted with grey bars]

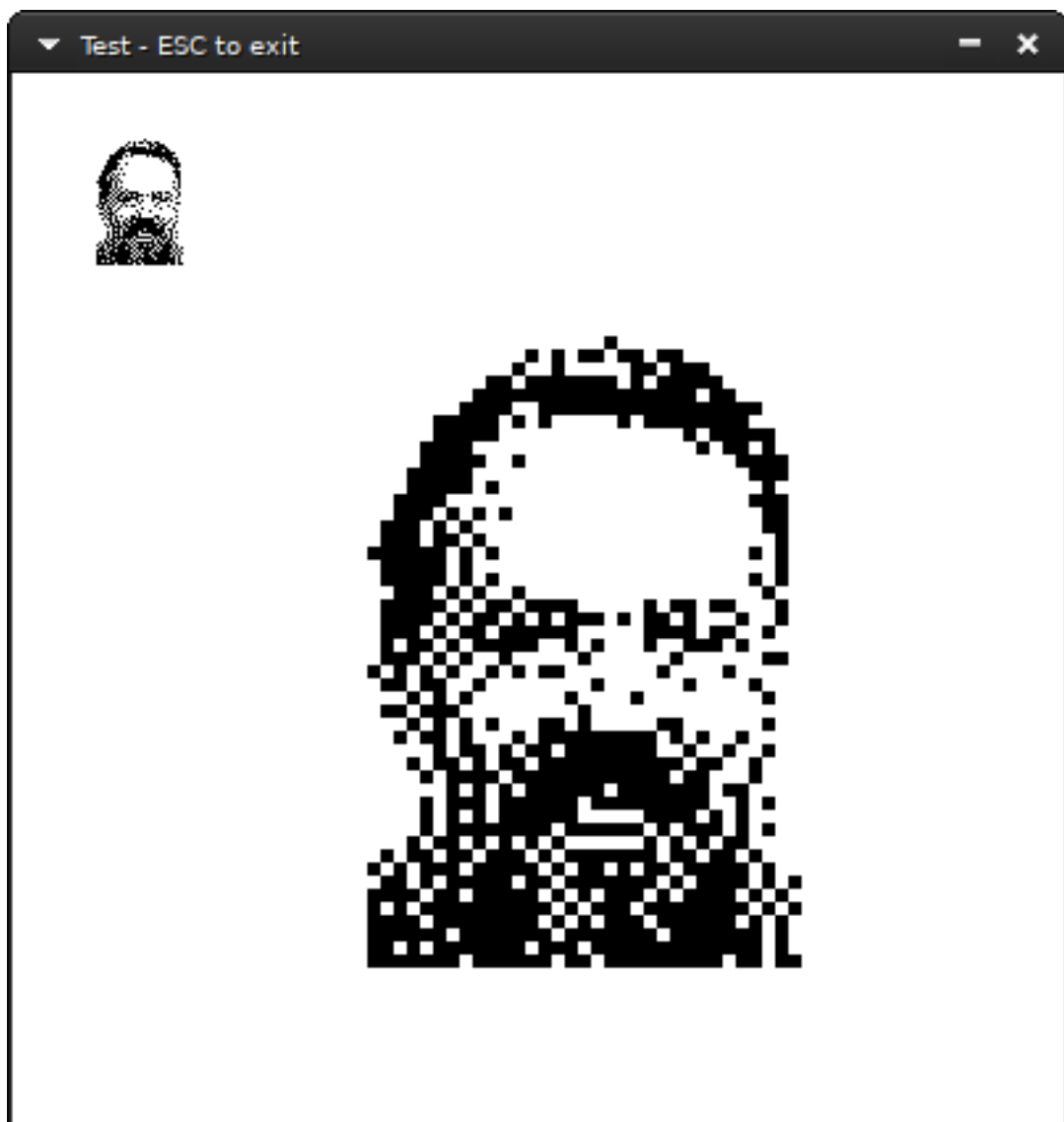
Chapter 23

90° Rotation of a bitmap by parallel recursive subdivision



Chapter 24

Magnification/Scaling



```
let mut original = Image::new(DMR_WIDTH, DMR_HEIGHT, 25, 25);
original.bytes = bits_to_bytes(DMR_BITS, DMR_WIDTH);
original.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);

let mut scaled = Image::new(DMR_WIDTH * 5, DMR_HEIGHT * 5, 100, 100);
let mut sx: i64; //source
let mut sy: i64; //source
let mut dx: i64; //destination
let mut dy: i64 = 0; //destination

let og_height = original.height as i64;
let og_width = original.width as i64;
let scaled_height = scaled.height as i64;
let scaled_width = scaled.width as i64;

while dy < scaled_height {
    sy = (dy * og_height) / scaled_height;
    dx = 0;
    while dx < scaled_width {
        sx = (dx * og_width) / scaled_width;
        if original.get(sx, sy) == Some(BLACK) {
            scaled.plot(dx, dy);
        }
        dx += 1;
    }
    dy += 1;
}
scaled.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);
```

24.1 Smoothing enlarged bitmaps

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[Redacted text block]

24.2 Stretching lines of bitmaps

[Redacted text block]

22

Part VIII

Areas

23

[REDACTED]

[REDACTED]

Chapter 25

Flood filling

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

Part IX

Advanced

25.1 Faster Drawing a line segment from its two endpoints using Symmetry

27



[REDACTED]

[REDACTED]

Chapter 26

Joining the ends of two wide line segments together



[REDACTED]

[REDACTED]

[REDACTED]

Chapter 27

Composing monochrome bitmaps with separate alpha channel data

CHAPTER 27. COMPOSING MONOCHROME BITMAPS WITH SEPARATE ALPHA CHANNEL

[Redacted text block 1]

[Redacted text block 2]

[Redacted text block 3]

Chapter 28

Orthogonal connection of two points

Chapter 29

Join segments with round corners



Chapter 30

Faster line clipping

Chapter 31

Space-Filling Curves



About this text

The text has been typeset in X_YL^AT_EX using the book class
and:

- *Redaction* for the main text.
- *Fira Sans* for referring to the programming language **Rust**.
- *Redaction20* for referring to the words **bitmap** and **pixels** as a concept.