

---

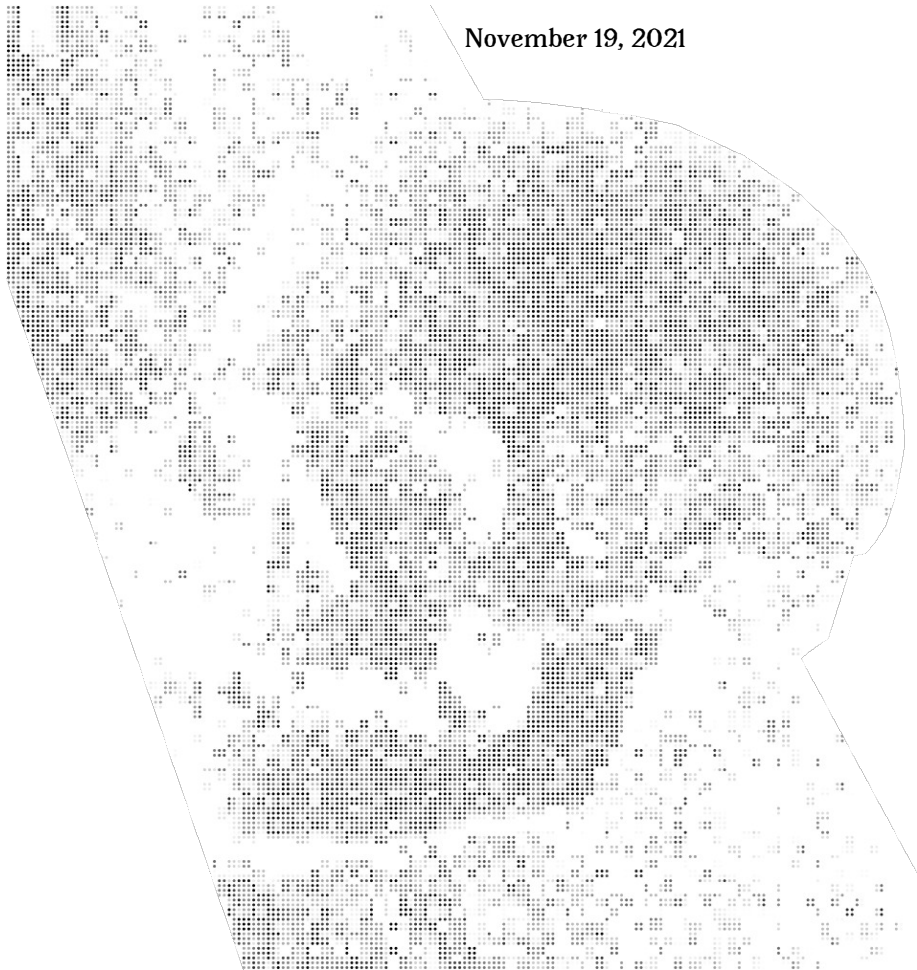
# A Bitmapper's Geometry

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

---

epilys

November 19, 2021



Manos Pitsidianakis (epilys)

<https://nessuent.xyz>

<https://github.com/epilys>

[epilys@nessuent.xyz](mailto: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 Geometry*, 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-geometry>

# Contents

<b>I</b>	<b>Introduction</b>	<b>6</b>
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 <b>Rust</b>	16
<b>II</b>	<b>Points and Lines</b>	<b>18</b>
6	Distance between two points	19
7	Distance from a point to a line	20
8	Equations of a line	22
9	The parametric form	24
10	Angle between two lines	26
11	Intersection of two lines	28
12	Line through two points	30
13	Line equidistant from two points	32
14	Normal to a line through a point	34
15	Bounding Circle	36

<i>CONTENTS</i>	4
<b>III Points, Lines and Circles</b>	<b>38</b>
16 Equations of a Circle	41
<b>IV Points, Line Segments and Arcs</b>	<b>43</b>
17 Drawing a line segment from its two endpoints	44
17.1 Faster Drawing a line segment from its two endpoints using Symmetry	45
18 Intersection of two line segments	47
18.1 <i>Fast</i> intersection of two line segments	49
19 Printing Line segments With Width	51
20 Joining the ends of two wide line segments together	53
<b>V Curves other than circles</b>	<b>55</b>
21 Parametric elliptical arcs	56
<b>VI Points, Lines and Planes</b>	<b>58</b>
22 Union, intersection and difference of polygons	59
23 Centroid of polygon	61
24 Space-Filling Curves	63
<b>VII Vectors, matrices and transformations</b>	<b>65</b>
25 Rotation of a bitmap	66
25.1 Fast 2D Rotation	71
26 90deg Rotation of a bitmap by parallel recursive subdivision	73
27 Magnification	75
27.1 Smoothing enlarged bitmaps	77

<i>CONTENTS</i>	5
27.2 Stretching lines of bitmaps	78
<b>VIII Flood filling</b>	<b>81</b>
<b>IX Areas</b>	<b>87</b>
<b>X Volumes</b>	<b>90</b>
<b>XI Advanced</b>	<b>93</b>
28 Composing monochrome bitmaps with separate alpha channel data	94
29 Orthogonal connection of two points	96
30 Join segments with round corners	98
31 Faster line clipping	100



# **Part I**

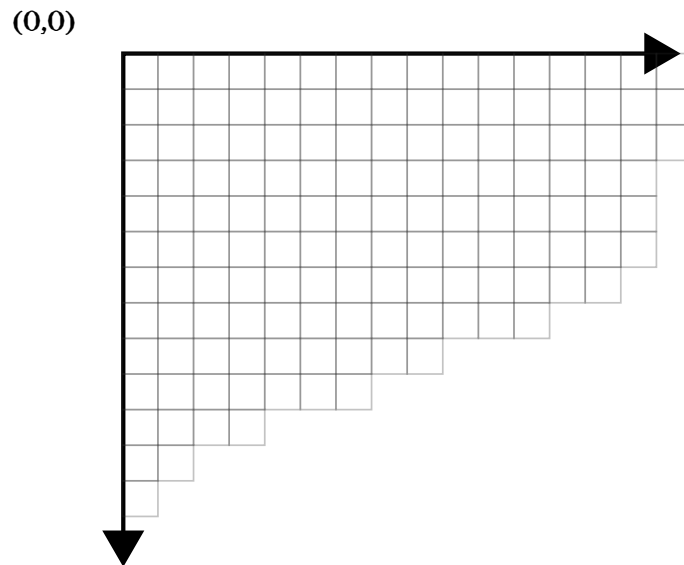
## **Introduction**

## 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:

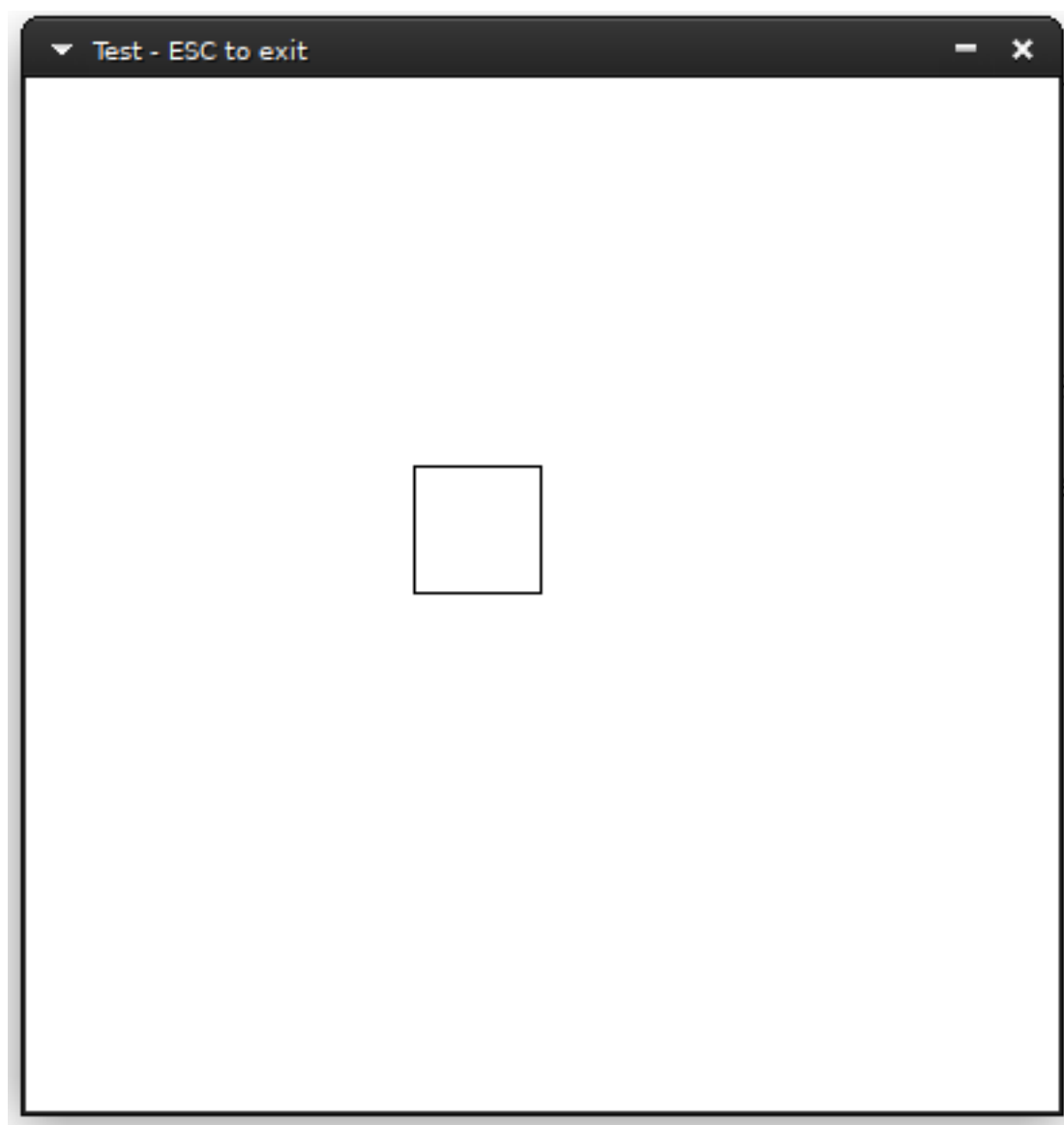
```
use bitmappers_geometry::*;
use minifb::{Key, Window, WindowOptions};

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:



## 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 `]`.

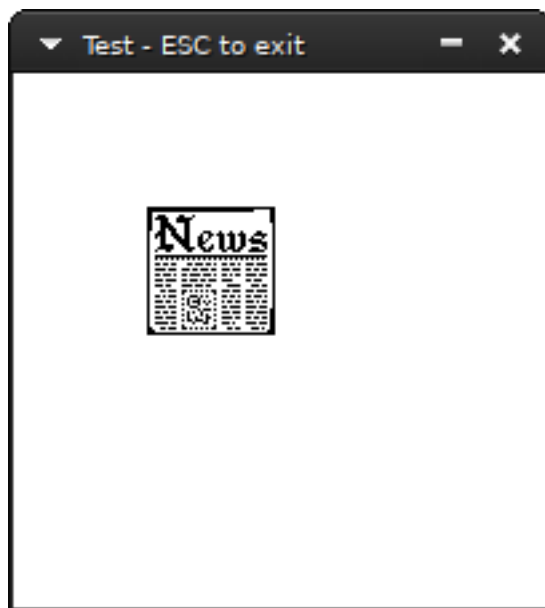
We can then include the new file in our source code:

```
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:

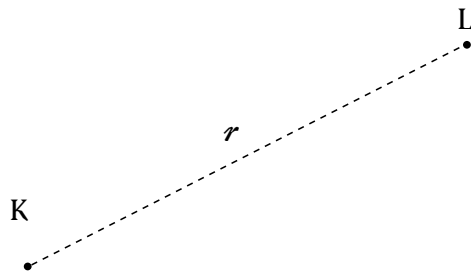


## **Part II**

### **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

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 8

## Equations of a line

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 9

## The parametric form

4



[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 10

## Angle between two lines

[REDACTED]

## Chapter 11

## Intersection of two lines

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 12

## Line through two points

7

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 13

### Line equidistant from two points







## Chapter 14

### Normal to a line through a point

[Redacted text block]

9

[Redacted text block]

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 15

## Bounding Circle

[REDACTED]

[REDACTED]

[REDACTED]

## **Part III**

# **Points, Lines and Circles**

[Redacted text block containing multiple paragraphs of obscured content]

[REDACTED]

[REDACTED]



## Chapter 16

## Equations of a Circle

[REDACTED]

[REDACTED]

[REDACTED]

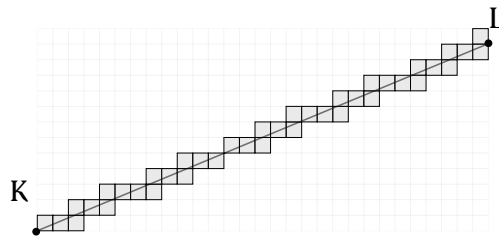
## **Part IV**

### **Points, Line Segments and Arcs**

## Chapter 17

# 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.

## 17.1 Faster Drawing a line segment from its two endpoints using Symmetry

13



## Chapter 18

## Intersection of two line segments

[REDACTED]

[REDACTED]

[REDACTED]



## 18.1 *Fast* intersection of two line segments

15

[Redacted content]



## Chapter 19

## Printing Line segments With Width



## Chapter 20

# Joining the ends of two wide line segments together



[REDACTED]

[REDACTED]

[REDACTED]

## **Part V**

### **Curves other than circles**

## Chapter 21

## Parametric elliptical arcs





## **Part VI**

# **Points, Lines and Planes**

## Chapter 22

# Union, intersection and difference of polygons

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 23

## Centroid of polygon

20

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 24

# Space-Filling Curves





## **Part VII**

# **Vectors, matrices and transformations**

## Chapter 25

### Rotation of a bitmap

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

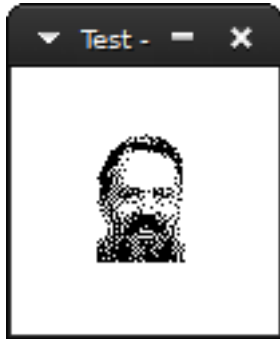
$$c = \cos\theta, s = \sin\theta, x_{p'} = x_p c - y_p s, y_{p'} = x_p s + y_p c.$$

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);
```

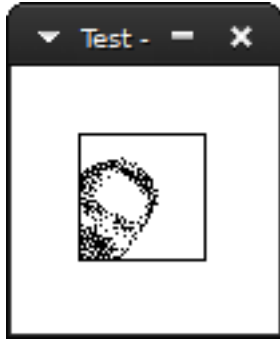
And then, loop for each byte in dmr's face and apply the rotation transformation.

```
let angle = 0.5;

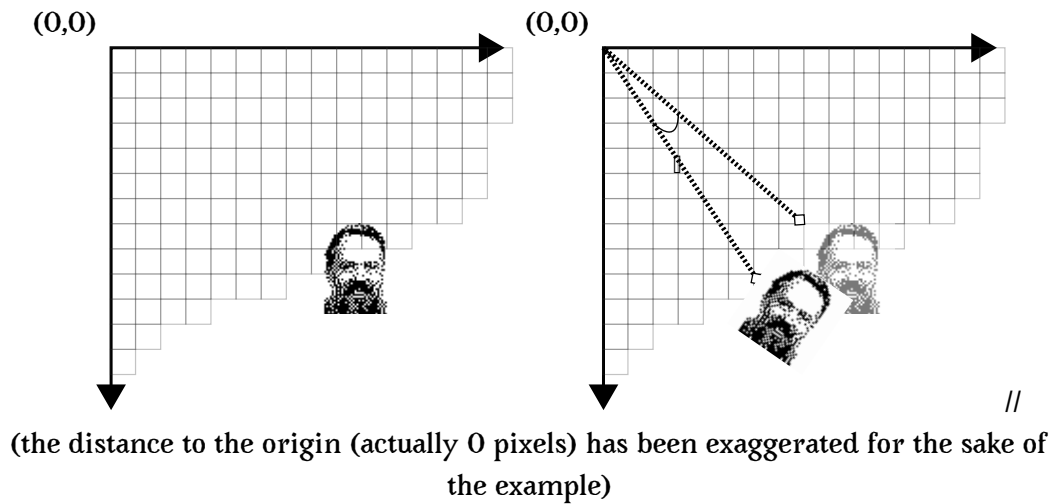
let c = f64::cos(angle);
let s = f64::sin(angle);

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:

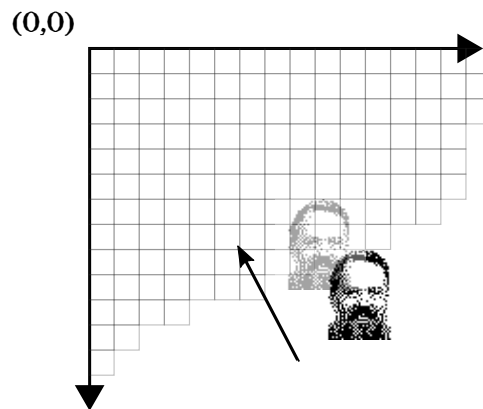
$$x_c = \frac{1}{n} \sum_{i=0}^n x_i$$

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

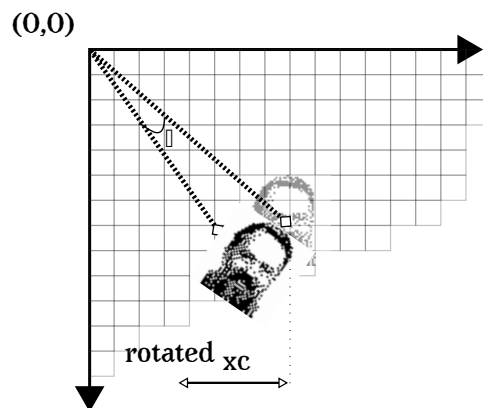
Since in this case we have a rectangle, the centroid has coordinates of half the width and half the height.

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

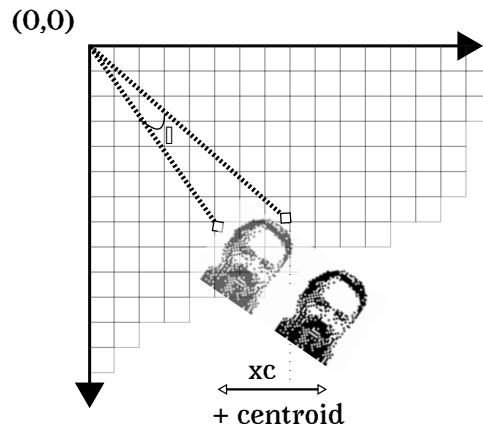
Here's it visually: First subtract the center point.



Then, rotate.



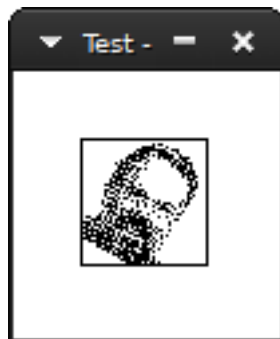
And subtract back to the original position.



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:



## 25.1 Fast 2D Rotation

22

[The following text is redacted with grey bars]





## Chapter 26

## 90deg Rotation of a bitmap by parallel recursive subdivision



## Chapter 27

## Magnification

[REDACTED]

[REDACTED]

[REDACTED]

## 25

[Redacted text block]

## 27.2 Stretching lines of bitmaps

[Redacted text block]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]





## **Part VIII**

### **Flood filling**

27

[Redacted text block containing multiple paragraphs of obscured content]

[Redacted text block]

[Redacted text block]

[Redacted text block]

[Redacted text block]



[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]



## **Part IX**

### **Areas**

[Redacted text block containing multiple paragraphs of obscured content]



[REDACTED]

[REDACTED]

# **Part X**

## **Volumes**

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

## **Part XI**

### **Advanced**

## Chapter 28

## Composing monochrome bitmaps with separate alpha channel data

[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 29

## Orthogonal connection of two points



[REDACTED]

[REDACTED]

[REDACTED]

## Chapter 30

## Join segments with round corners



## Chapter 31

## Faster line clipping



## About this text

The text has been typeset in X<sub>Y</sub>L<sup>A</sup>T<sub>E</sub>X using the book class and:

- **Avara** for the main text.
- *Fira Sans* for referring to the programming language **Rust**.
- Terminal Grotesque for referring to the word `bitmap` as a concept.