
A Bitmapper's Companion

epilys

2021

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



| | | |
|---------------------------------------|----|------------------------------------|
| Table Of Contents | 4 | toc |
| Introduction | 7 | intro |
| Points And Lines | 19 | lines |
| Points and Line Segments | 36 | segments |
| Points, Lines and Circles | 45 | circles |
| Curves other than circles | 57 | curves |
| Points, Lines and Shapes | 62 | shapes |
| Vectors, matrices and transformations | 72 | trans- forma- tions |
| Addendum | 96 | adden- dum |



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 | 9 |
| 1 | Data representation | 11 |
| 2 | Displaying pixels to your screen | 13 |
| 3 | Bits to byte pixels | 15 |
| 4 | Loading xbm files in Rust | 17 |
| | | |
| II | Points And Lines | 21 |
| 5 | Distance between two points | 23 |
| 6 | Equations of a line | 25 |
| 6.1 | Line through a point $P = (x_p, y_p)$ and a slope m | 25 |
| 6.2 | Line through two points | 26 |
| 7 | Distance from a point to a line | 27 |
| 7.1 | Using the implicit equation form | 27 |
| 7.2 | Using an L defined by two points P_1, P_2 | 28 |
| 7.3 | Using an L defined by a point P_l and angle θ | 28 |
| 8 | Angle between two lines | 29 |
| 9 | Intersection of two lines | 31 |
| 10 | Line equidistant from two points | 33 |
| 11 | Normal to a line through a point | 35 |



| | | |
|------------|--|-----------|
| III | Points And Line Segments | 37 |
| 12 | Drawing a line segment from its two endpoints | 39 |
| 13 | Drawing line segments with width | 41 |
| 14 | Intersection of two line segments | 43 |
| 14.1 | <i>Fast</i> intersection of two line segments | 43 |
| | | |
| IV | Points, Lines and Circles | 47 |
| 15 | Equations of a circle | 51 |
| 16 | Bounding circle | 53 |
| | | |
| V | Curves other than circles | 59 |
| 17 | Parametric elliptical arcs | 61 |
| | | |
| VI | Points, Lines and Shapes | 63 |
| 18 | Union, intersection and difference of polygons | 65 |
| 19 | Centroid of polygon | 67 |
| 20 | Polygon clipping | 69 |
| 21 | Flood filling | 71 |
| | | |
| VII | Vectors, matrices and transformations | 73 |
| 22 | Rotation of a bitmap | 75 |
| 22.1 | Fast 2D Rotation | 79 |
| 23 | 90° Rotation of a bitmap by parallel recursive subdivision | 81 |
| 24 | Magnification/Scaling | 83 |
| 24.1 | Smoothing enlarged bitmaps | 85 |
| 24.2 | Stretching lines of bitmaps | 86 |
| 25 | Mirroring | 89 |
| 26 | Shearing | 91 |
| 26.1 | The relationship between shearing factor and angle | 93 |

| | |
|--|-----------|
| <i>CONTENTS</i> | 7 |
| 27 Projections | 95 |
| VIII Addendum | 97 |
| 27.1 Faster Drawing a line segment from its two endpoints using Symmetry | 99 |
| 28 Joining the ends of two wide line segments together | 101 |
| 29 Composing monochrome bitmaps with separate alpha channel data | 103 |
| 30 Orthogonal connection of two points | 105 |
| 31 Join segments with round corners | 107 |
| 32 Faster line clipping | 109 |
| 33 Space-filling Curves | 111 |
| 33.1 Hilbert curve | 113 |
| 33.2 Sierpiński curve | 115 |
| 33.3 Peano curve | 115 |
| 33.4 Z-order curve | 117 |
| 33.5 flowsnake curve | 117 |
| 34 Dithering | 119 |
| 35 Marching squares | 121 |
| Index | 123 |



intro

Part I

Introduction

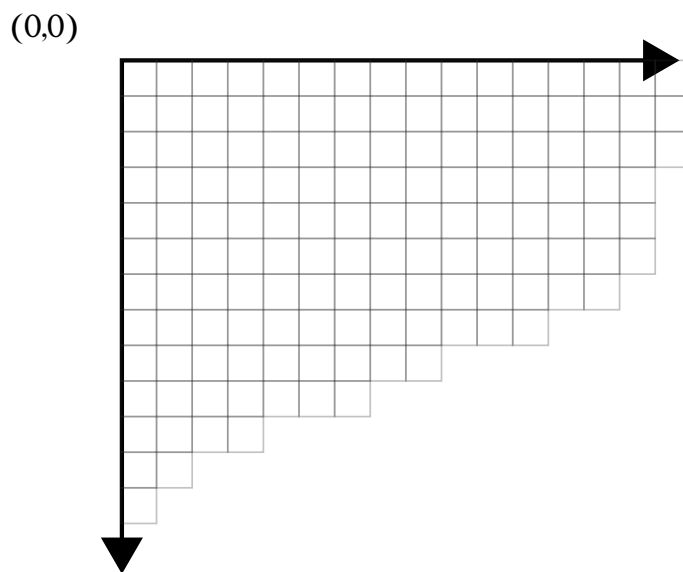
intro

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 l 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

src/lib.rs: ignored (clipped).



This code file is a PDF
attachment

intro

```
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:

`src/bin/introduction.rs`



This code file is a PDF attachment

```
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();

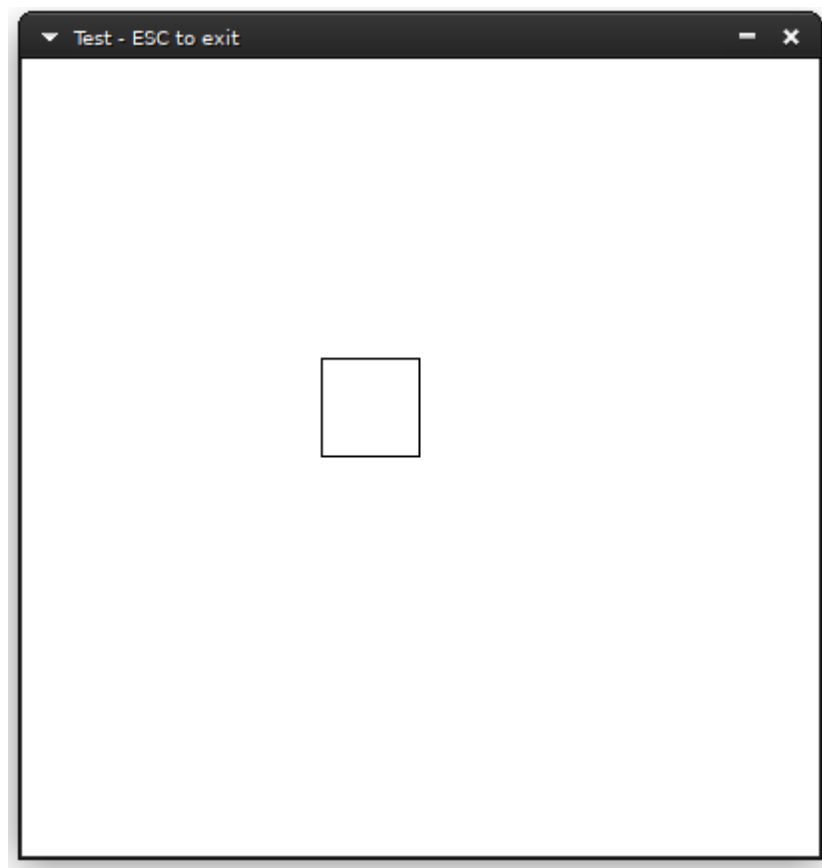
    // Limit to max ~60 fps update rate
    window.limit_update_rate(Some(std::time::Duration::from_micros(16600)));

    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:

intro



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  
}
```

intro

Chapter 4

Loading xbm files in Rust

*The end of this chapter includes a short **Rust** program to automatically convert **xbm** files to equivalent **Rust** code.*

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 `]`.

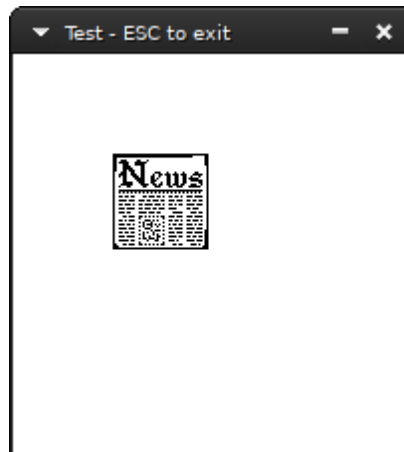
We can then include the new file in our source code:

```
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:



The following short program uses the `regex` crate to match on these simple rules and print the equivalent code in `stdout`. You can use it like so:

```
cargo run --bin xbmtrs -- file.xbm > file.xbm.rs
```

```
src/bin/xbmtors.rs:
```



This code file is a PDF attachment

```
use regex;
use regex::Regex;
use std::fs::File;
use std::io::prelude::*;

fn main() {
    let args = std::env::args().skip(1).collect::
```

```

    \|s*\x23\s*define\s+.\+_height\s+(?P<h>\d\d*)$
    \|s*\s*static(\s+unsigned){0,1}\s+char\s+.\+_bits..\s*=\s*\{(?P<b>[\^}]+)\};
",
)
.unwrap();
let caps = re
    .captures(&s)
    .expect("Could not convert file, regex doesn't match :(");
let ident = caps.name("i").unwrap().as_str().to_uppercase();
let out = re.replace_all(&s, format!("const {i}_WIDTH: usize = $w;\nconst {i}_HEIGHT:
↪  usize = $h;\nconst {i}_BITS: &[u8] = &[$b];", i = &ident));
println!("{}", out.trim());
}

```

lines

Part II

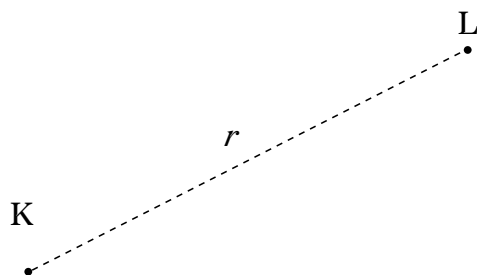
Points And Lines

lines

Chapter 5

Distance between two points

lines



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

$$r = \sqrt{(x_L - x_K)^2 + (y_L - y_K)^2} \quad (5.1)$$

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)  
}
```

lines

Chapter 6

Equations of a line

lines

There are several ways to describe a line mathematically. We'll list the convenient ones for drawing pixels.

The equation that describes every possible line on a two dimensional grid is the *implicit* form $ax + by = c$, $(a, b) \neq (0, 0)$. We can generate equivalent equations by adding the equation to itself, i.e. $ax + by = c \equiv 2ax + 2by = 2c \equiv a'x + b'y = c'$, $a' = 2a, b' = 2b, c' = 2c$ as many times as we want. To "minimize" the constants a, b, c we want to satisfy the relationship $a^2 + b^2 = 1$, and thus can convert the equivalent equations into one representative equation by multiplying the two sides with $\frac{1}{\sqrt{a^2 + b^2}}$; this is called the normalized equation.

The *slope intercept form* describes any line that intercepts the y axis at $b \in \mathbb{R}$ with a specific slope a :

$$y = ax + b$$

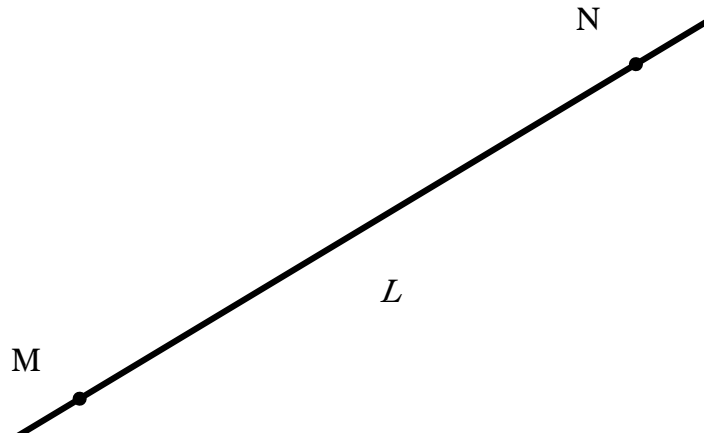
The *parametric* form...

6.1 Line through a point $P = (x_p, y_p)$ and a slope m

$$y - y_p = m(x - x_p)$$

6.2 Line through two points

lines



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 :

$$x = x_M + (x_N - x_M)t, t \in \mathbb{R}, x \in \{x, y\}$$

$$y = y_M + (y_N - y_M)t, t \in \mathbb{R}, y \in \{x, y\}$$

Substituting t in one of the equations we get:

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

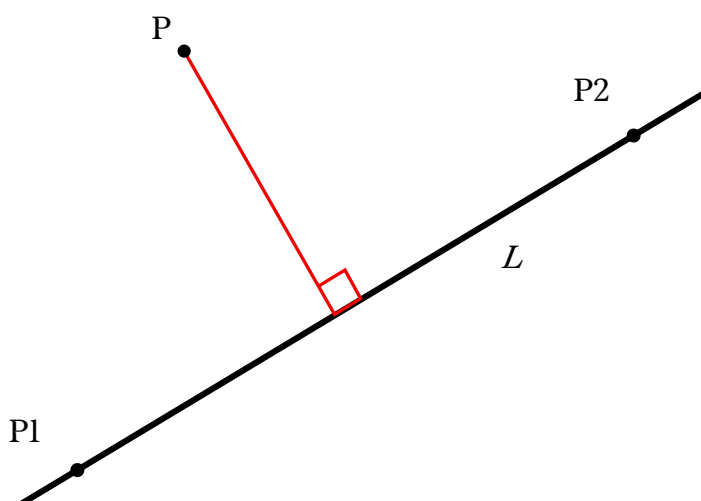
Which is what we were after. We finish by normalising what we found with $\frac{1}{\sqrt{a^2 + b^2}}$:

Chapter 7

Distance from a point to a line

lines

Add code samples in *Distance from a point to a line*



7.1 Using the implicit equation form

Let's find the distance from a given point P and a given line L . Let d be the distance between them. Bring L to the implicit form $ax + by = c$.

$$d = \frac{|ax_p + by_p + c|}{\sqrt{a^2 + b^2}}$$

7.2 Using an L defined by two points P_1, P_2

With $P = (x_0, y_0)$, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.

$$d = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

7.3 Using an L defined by a point P_l and angle θ

$$d = |\cos(\theta)(P_{ly} - y_p) - \sin(\theta)(P_{lx} - P_x)|$$

Chapter 8

Angle between two lines

lines

Add *Angle between two lines*

1

lines

Chapter 9

Intersection of two lines

lines

Add *Intersection of two lines*

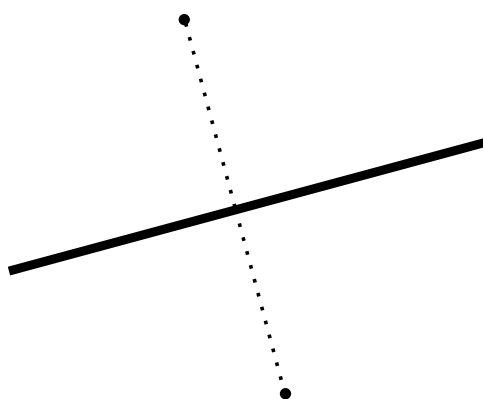
2

lines

Chapter 10

Line equidistant from two points

lines



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 .

lines

Chapter 11

Normal to a line through a point

lines

Add *Normal to a line through a point*

3

lines

Part III

Points And Line Segments

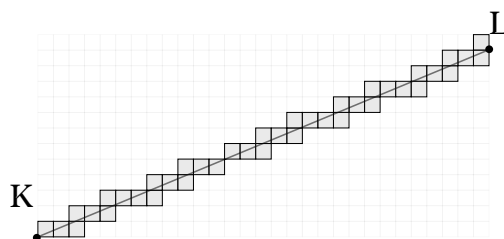
segments

segments

Chapter 12

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 };  
}
```

segments

```
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;
  }
}
```

Add some explanation behind the algorithm in *Drawing a line segment from its two endpoints*

Chapter 13

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);  
                }  
            }  
        }  
    }  
}
```

segments

```

        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 14

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 *Equations of a line*.

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 *Intersection of two lines*.)

Add code sample in *Intersection of two line segments*

segments

14.1 Fast intersection of two line segments

4



Part IV

Points, Lines and Circles

circles

5

[Redacted text block 1]

[Redacted text block 2]

[Redacted text block 3]

[Redacted text block 4]

[Redacted text block]

Chapter 15

Equations of a circle

Add Equations of a circle

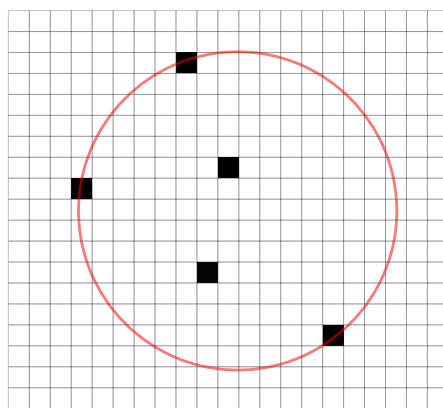
6

circles

[Redacted text block containing multiple paragraphs of placeholder content]

Chapter 16

Bounding circle



src/bin/boundingcircle.rs:



This code file is a PDF attachment

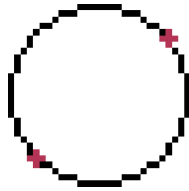
circles

A bounding circle is a circle that includes all the points in a given set. Usually we're interested in one of the smallest ones possible.



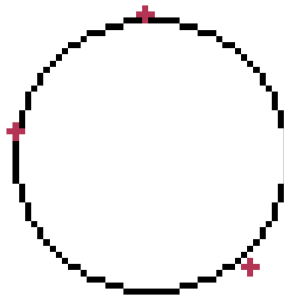
We can use the following methodology to find the bounding circle: start from two points and the circle they make up, and for each of the rest of the points check if the circle includes them. If not, make a bounding circle that includes every point up to the current one. To do this, we need some primitive operations.

We will need a way to construct a circle out of two points:



```
let p1 = points[0];
let p2 = points[1];
//The circle is determined by two points, P and Q. The center of the circle
↪ is
//at (P + Q)/2.0 and the radius is |(P - Q)/2.0|
let d_2 = (
  ((p1.0 + p2.0) / 2), (p1.1 + p2.1) / 2),
  (distance_between_two_points(p1, p2) / 2.0),
);
```

And a way to make a circle out of three points:



```
fn min_circle_w_3_points(q1: Point, q2: Point, q3: Point) -> Circle {
  let (ax, ay) = (q1.0 as f64, q1.1 as f64);
  let (bx, by) = (q2.0 as f64, q2.1 as f64);
  let (cx, cy) = (q3.0 as f64, q3.1 as f64);

  let mut d = 2. * (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by));
  if d == 0.0 {
    d = std::cmp::max(
      std::cmp::max(
        distance_between_two_points(q1, q2) as i64,
        distance_between_two_points(q2, q3) as i64,
      ),
      distance_between_two_points(q1, q3) as i64,
    ) as f64
  }
  / 2.;
}
let ux = ((ax * ax + ay * ay) * (by - cy)
  + (bx * bx + by * by) * (cy - ay)
  + (cx * cx + cy * cy) * (ay - by))
  / d;
let uy = ((ax * ax + ay * ay) * (cx - bx)
```

```

    + (bx * bx + by * by) * (ax - cx)
    + (cx * cx + cy * cy) * (bx - ax))
    / d;
let mut center = (ux as i64, uy as i64);
if center.0 < 0 {
    center.0 = 0;
}
if center.1 < 0 {
    center.1 = 0;
}
let d = distance_between_two_points(center, q1);
(center, d)
}

```

The algorithm:

```

use bitmappers_companion::*;
use minifb::{Key, Window, WindowOptions};
use rand::seq::SliceRandom;
use rand::thread_rng;
use std::f64::consts::{FRAC_PI_2, PI};

include!("../me.xbm.rs");

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

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)
}

fn image_to_points(image: &Image) -> Vec<Point> {
    let mut ret = Vec::with_capacity(image.bytes.len());
    for y in 0..(image.height as i64) {
        for x in 0..(image.width as i64) {
            if image.get(x, y) == Some(BLACK) {
                ret.push((x, y));
            }
        }
    }
    ret
}

type Circle = (Point, f64);

fn bc(image: &Image) -> Circle {
    let mut points = image_to_points(image);
    points.shuffle(&mut thread_rng());
    min_circle(&points)
}

fn min_circle(points: &[Point]) -> Circle {
    let mut points = points.to_vec();
    points.shuffle(&mut thread_rng());

    let p1 = points[0];
    let p2 = points[1];
    //The circle is determined by two points, P and Q. The center of the
    ↪ circle is
    //at (P + Q)/2.0 and the radius is |(P - Q)/2.0|
    let d_2 = (
        ((p1.0 + p2.0) / 2), (p1.1 + p2.1) / 2),
        (distance_between_two_points(p1, p2) / 2.0),
    );
    let mut d_prev = d_2;

    for i in 2..points.len() {
        let p_i = points[i];
        if distance_between_two_points(p_i, d_prev.0) <= (d_prev.1) {
            // then d_i = d_(i-1)

```

```

    } else {
        let new = min_circle_w_point(&points[..i], p_i);
        if distance_between_two_points(p_i, new.0) <= (new.1) {
            d_prev = new;
        }
    }
}
d_prev
}

fn min_circle_w_point(points: &[Point], q: Point) -> Circle {
    let mut points = points.to_vec();
    points.shuffle(&mut thread_rng());
    let p1 = points[0];
    //The circle is determined by two points, P_1 and Q. The center of the
    ↪ circle is
    //at (P_1 + Q)/2.0 and the radius is |(P_1 - Q)/2.0|
    let d_1 = (
        ((p1.0 + q.0) / 2), (p1.1 + q.1) / 2),
        (distance_between_two_points(p1, q) / 2.0),
    );
    let mut d_prev = d_1;
    for j in 1..points.len() {
        let p_j = points[j];
        if distance_between_two_points(p_j, d_prev.0) <= (d_prev.1) {
            //d_prev = d_prev;
        } else {
            let new = min_circle_w_points(&points[..j], p_j, q);
            if distance_between_two_points(p_j, new.0) <= (new.1) {
                d_prev = new;
            }
        }
    }
    d_prev
}

fn min_circle_w_points(points: &[Point], q1: Point, q2: Point) -> Circle {
    let mut points = points.to_vec();
    let d_0 = (
        ((q1.0 + q2.0) / 2), (q1.1 + q2.1) / 2),
        (distance_between_two_points(q1, q2) / 2.0),
    );
    let mut d_prev = d_0;
    for k in 0..points.len() {
        let p_k = points[k];
        if distance_between_two_points(p_k, d_prev.0) <= (d_prev.1) {
        } else {
            let new = min_circle_w_3_points(q1, q2, p_k);
            if distance_between_two_points(p_k, new.0) <= (new.1) {
                d_prev = new;
            }
        }
    }
    d_prev
}

fn min_circle_w_3_points(q1: Point, q2: Point, q3: Point) -> Circle {
    let (ax, ay) = (q1.0 as f64, q1.1 as f64);
    let (bx, by) = (q2.0 as f64, q2.1 as f64);
    let (cx, cy) = (q3.0 as f64, q3.1 as f64);
    let mut d = 2. * (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by));
    if d == 0.0 {
        d = std::cmp::max(
            std::cmp::max(
                distance_between_two_points(q1, q2) as i64,
                distance_between_two_points(q2, q3) as i64,
            ),
            distance_between_two_points(q1, q3) as i64,
        ) as f64
        / 2.;
    }
}

```



```

let ux = ((ax * ax + ay * ay) * (by - cy)
  + (bx * bx + by * by) * (cy - ay)
  + (cx * cx + cy * cy) * (ay - by))
  / d;
let uy = ((ax * ax + ay * ay) * (cx - bx)
  + (bx * bx + by * by) * (ax - cx)
  + (cx * cx + cy * cy) * (bx - ax))
  / d;
let mut center = (ux as i64, uy as i64);
if center.0 < 0 {
  center.0 = 0;
}
if center.1 < 0 {
  center.1 = 0;
}
let d = distance_between_two_points(center, q1);
(center, d)
}

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: true,
      //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 full = Image::new(WINDOW_WIDTH, WINDOW_HEIGHT, 0, 0);
  let mut image = Image::new(ME_WIDTH, ME_HEIGHT, 45, 45);
  image.bytes = bits_to_bytes(ME_BITS, ME_WIDTH);
  let (center, r) = bc(&image);
  image.draw_outline();

  full.plot_circle((center.0 + 45, center.1 + 45), r as i64, 0.);
  while window.is_open() && !window.is_key_down(Key::Escape) &&
  ↪ !window.is_key_down(Key::Q) {
    image.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);
    full.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);

    window
      .update_with_buffer(&buffer, WINDOW_WIDTH, WINDOW_HEIGHT)
      .unwrap();

    let millis = std::time::Duration::from_millis(100);
    std::thread::sleep(millis);
  }
}

```


Part V

Curves other than circles

curves

Chapter 17

Parametric elliptical arcs

Add *Parametric elliptical arcs*

7

curves

[Redacted text block]

[Redacted text block]

[Redacted text block]

Part VI

Points, Lines and Shapes

shapes

Chapter 18

Union, intersection and difference of polygons

Add Union, intersection and difference of polygons

8

shapes



Chapter 19

Centroid of polygon

Add Centroid of polygon

9

shapes

1. The first part of the document discusses the importance of maintaining accurate records of all transactions and activities. It emphasizes the need for transparency and accountability in financial reporting.

2. The second part of the document outlines the various methods and techniques used to collect and analyze data. It includes a detailed description of the experimental procedures and the statistical analysis performed.

3. The third part of the document presents the results of the study and discusses the implications of the findings. It compares the results with previous research and provides a conclusion based on the evidence.

Chapter 20

Polygon clipping

Chapter 21

Flood filling

Add Flood filling

10

shapes

1. The first part of the document discusses the importance of maintaining accurate records of all transactions and activities. It emphasizes the need for transparency and accountability in financial reporting.

2. The second part of the document outlines the various methods and techniques used to collect and analyze data. It includes a detailed description of the experimental procedures and the statistical analysis performed.

3. The third part of the document presents the results of the study and discusses the implications of the findings. It compares the results with previous research and provides a comprehensive analysis of the data.

Part VII

Vectors, matrices and transformations

trans-
forma-
tions

Chapter 22

Rotation of a bitmap

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

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

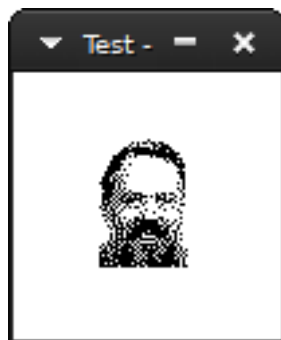
Let's load an xface. We will use `bits_to_bytes` (See Introduction).

```
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);
```

src/bin/rotation.rs:



This code file is a PDF attachment



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

trans-
forma-
tions

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

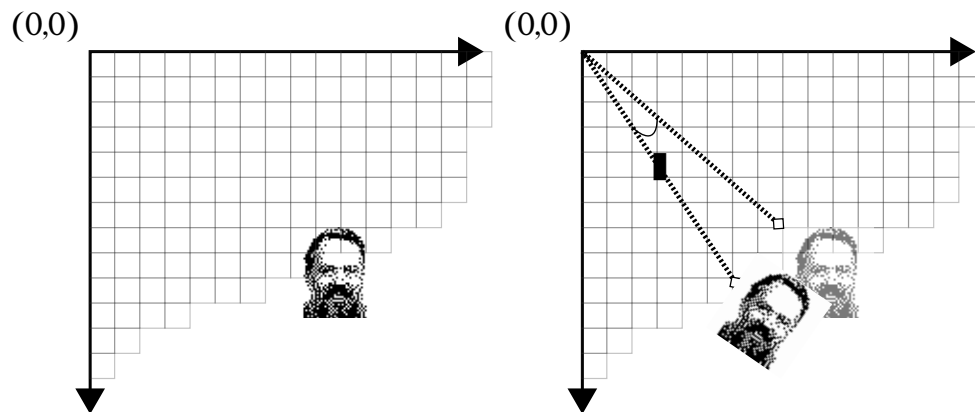
And then, loop for each byte in dmr's face and apply the rotation transformation.

```
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:



(the distance to the origin (actually 0 pixels) has been exaggerated for the sake of the example)

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:

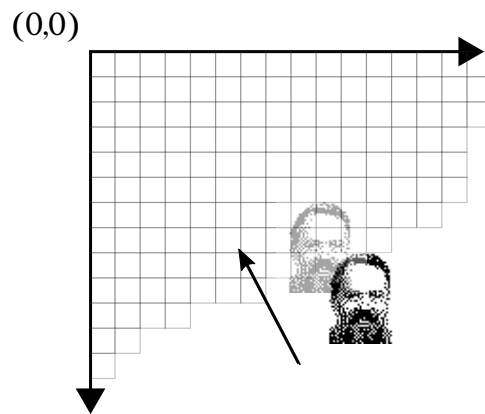
$$x_c = \frac{1}{n} \sum_{i=0}^n x_i$$

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

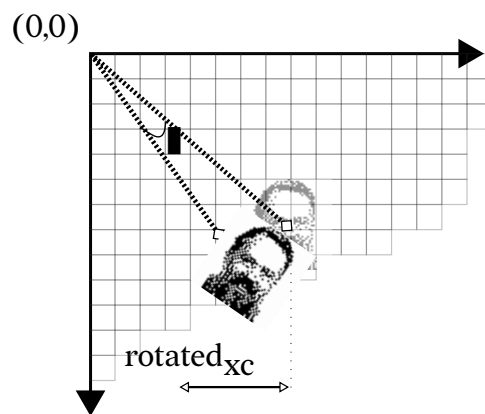
Since in this case we have a rectangle, the centroid has coordinates of half the width and half the height.

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

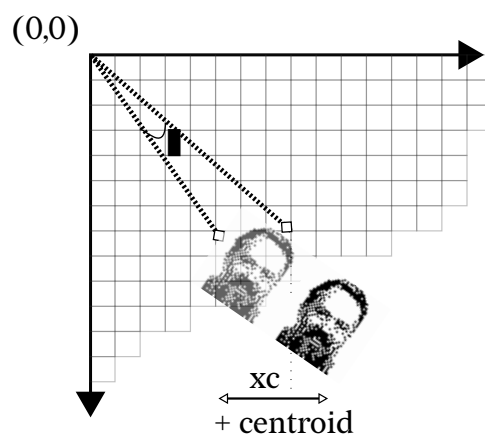
Here's it visually: First subtract the center point.



Then, rotate.

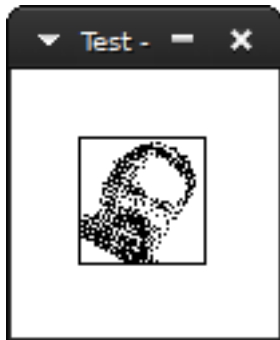


And subtract back to the original position.



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

Add Fast 2D Rotation

11

trans-
forma-
tions

[Redacted text block]

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 23

90° Rotation of a bitmap by parallel recursive subdivision

Add 90° Rotation of a bitmap by parallel recursive subdivision

12

trans-
forma-
tions

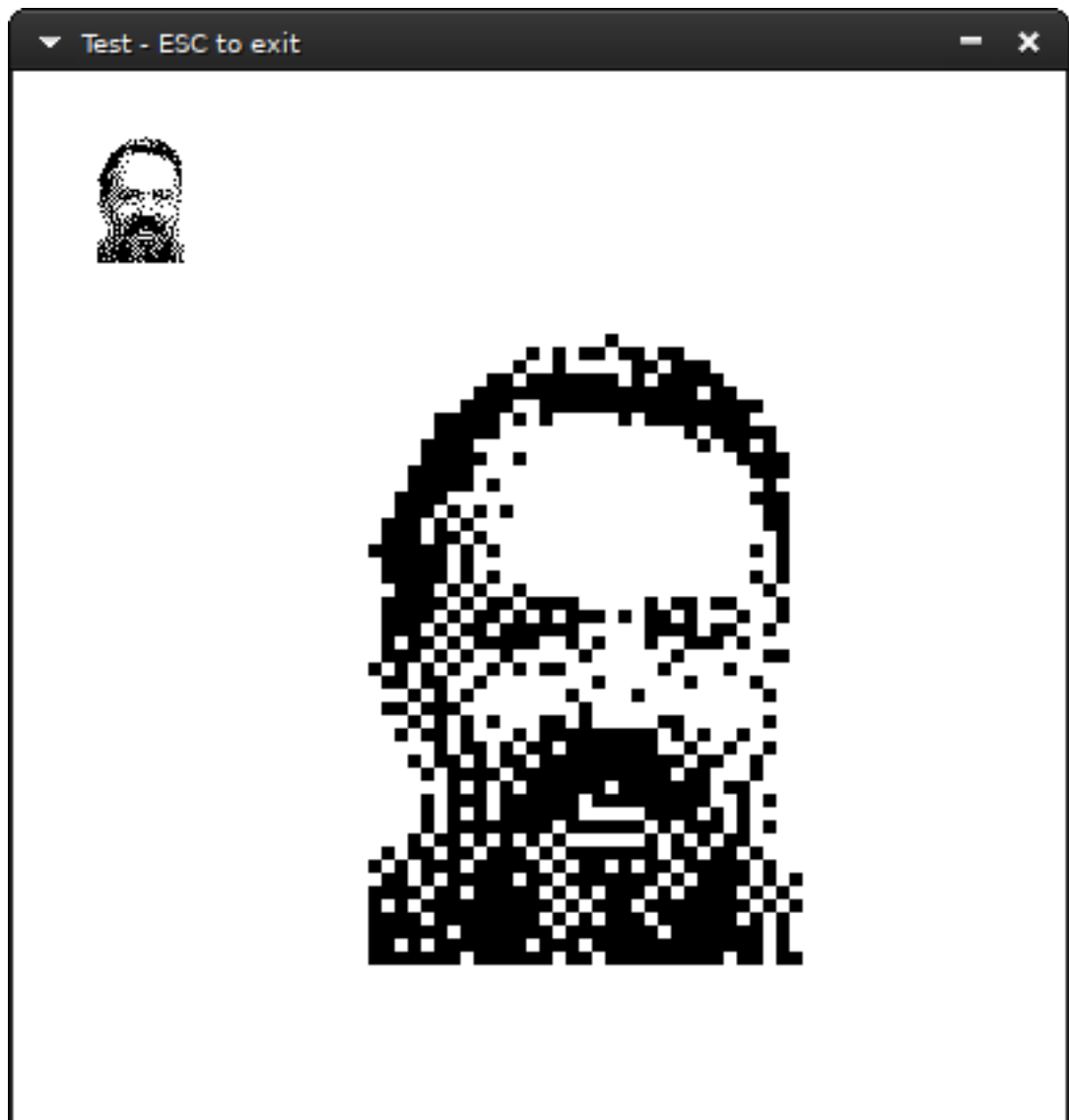
[Redacted text block 1]

[Redacted text block 2]

[Redacted text block 3]

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);
```

src/bin/scale.rs:



This code file is a PDF attachment

24.1 Smoothing enlarged bitmaps

Add *Smoothing enlarged bitmaps*

13

trans-
forma-
tions



24.2 Stretching lines of bitmaps

Add *Stretching lines of bitmaps*



[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]



Chapter 25

Mirroring

Add screenshots and figure and code in *Mirroring*

Mirroring to an axis is the transformation of one coordinate to its equidistant value across the axis:

To mirror a pixel across the x axis, simply multiply its coordinates with the following matrix:

$$M_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

This results in the y coordinate's sign being flipped.

For y -mirroring, the transformation follows the same logic:

$$M_y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Chapter 26

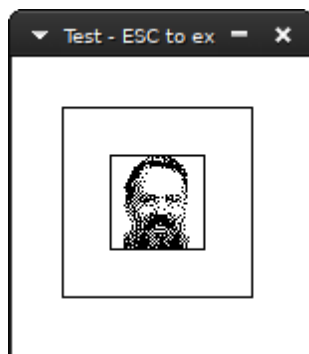
Shearing

Simple shearing is the transformation of one dimension by a distance proportional to the other dimension, In x -shearing (or horizontal shearing) only the x coordinate is affected, and likewise in y -shearing only y as well.

src/bin/shearing.rs:



This code file is a PDF attachment



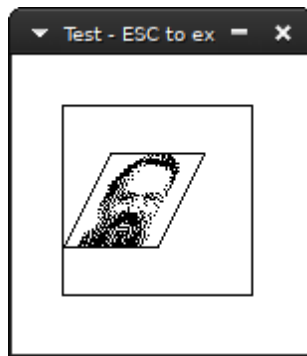
With l being equal to the desired tilt away from the y axis, the transformation is described by the following matrix:

$$S_x = \begin{bmatrix} 1 & l \\ 0 & 1 \end{bmatrix}$$

Which is as simple as this function:

```
fn shear_x((x_p, y_p): (i64, i64), l: f64) -> (i64, i64) {
    (x_p + (l * (y_p as f64)) as i64, y_p)
}
```

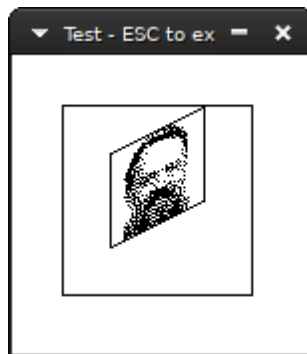
trans-
forma-
tions



For y -shearing, we have the following:

$$S_y = \begin{bmatrix} 1 & 0 \\ l & 1 \end{bmatrix}$$

```
fn shear_y((x_p, y_p): (i64, i64), l: f64) -> (i64, i64) {
    (x_p, (l*(x_p as f64)) as i64 + y_p)
}
```



A full example:

```
include!("../dmr.xbm.rs");
const WINDOW_WIDTH: usize = 200;
const WINDOW_HEIGHT: usize = 200;

fn shear_x((x_p, y_p): (i64, i64), l: f64) -> (i64, i64) {
    (x_p+(l*(y_p as f64)) as i64, y_p)
}
fn shear_y((x_p, y_p): (i64, i64), l: f64) -> (i64, i64) {
    (x_p, (l*(x_p as f64)) as i64 + y_p)
}

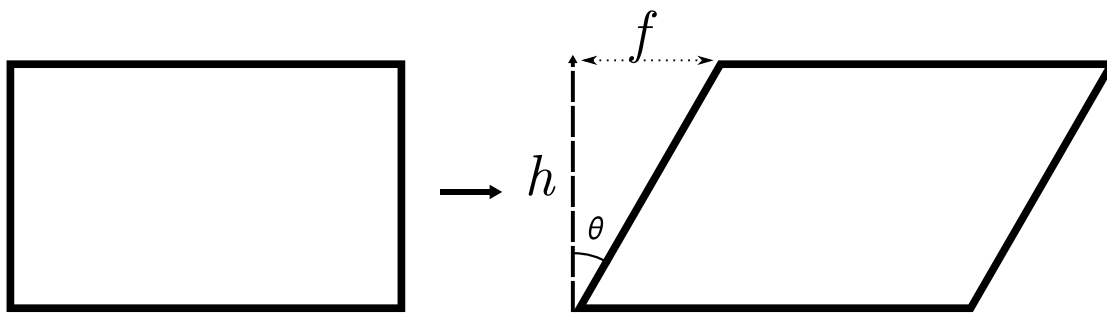
let mut image = Image::new(DMR_WIDTH, DMR_HEIGHT, 25, 25);
image.bytes = bits_to_bytes(DMR_BITS, DMR_WIDTH);
image.draw_outline();
```

```

let l = -0.5;
let mut sheared = Image::new(DMR_WIDTH*2, DMR_HEIGHT*2, 25, 25);
for x in 0..DMR_WIDTH {
  for y in 0..DMR_HEIGHT {
    if image.bytes[y * DMR_WIDTH + x] == BLACK {
      let p = shear_x((x as i64 ,y as i64 ), l);
      sheared.plot(p.0+(DMR_WIDTH/2) as i64, p.1+(DMR_HEIGHT/2) as i64);
    }
  }
}
sheared.draw_outline();

```

26.1 The relationship between shearing factor and angle



Shearing is a delta movement in one dimension, thus the point before moving and the point after form an angle with the x axis. To move a point $(x, 0)$ by 30° forward we will have the new point $(x + f, 0)$ where f is the shear factor. These two points and (x, h) where h is the height of the bitmap form a triangle, thus the following are true:

$$\cot\theta = \frac{h}{f}$$

Therefore to find your factor for any angle θ replace its cotangent in the following formula:

$$f = \frac{h}{\cot\theta}$$

For example to shear by -30° (meaning the bitmap will move to the right, since rotations are always clockwise) we need $\cot(-30deg) = -\sqrt{3}$ and $f = -\frac{h}{\sqrt{3}}$.

Chapter 27

Projections

Add Projections

15

trans-
forma-
tions

[Redacted text block]

[Redacted text block]

[Redacted text block]

Part VIII

Addendum

adden-
dum

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

Add *Faster Drawing a line segment from its two endpoints using Symmetry*

16

[REDACTED]

[REDACTED]

Chapter 28

Joining the ends of two wide line segments together

Add *Joining the ends of two wide line segments together*

17

addendum

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 29

Composing monochrome bitmaps with separate alpha channel data

Add Composing monochrome bitmaps with separate alpha channel data

18

addendum

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 30

Orthogonal connection of two points

Add *Orthogonal connection of two points*

19

addendum

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 31

Join segments with round corners

Add Join segments with round corners

20

addendum

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 32

Faster line clipping

Add *Faster line clipping*

21

addendum

[Redacted text block]

[Redacted text block]

[Redacted text block]

Chapter 33

Space-filling Curves

Add Space-filling Curves

22

addendum

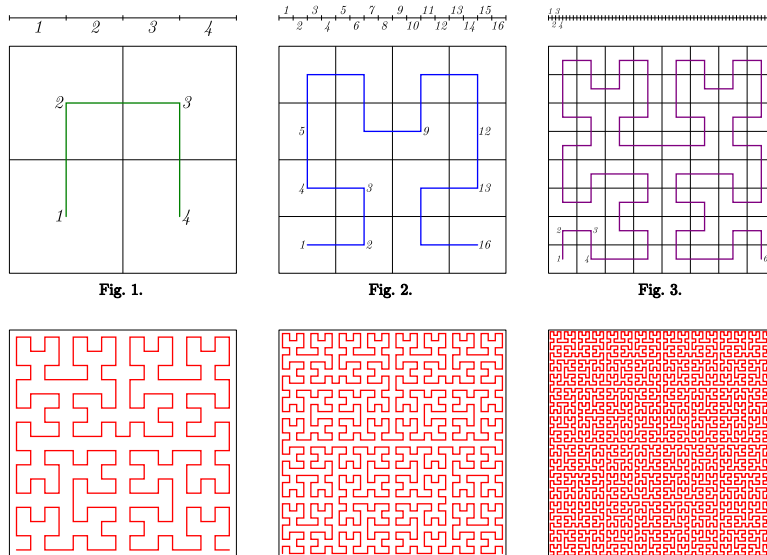
[Redacted text block]

[Redacted text block]

[Redacted text block]

33.1 Hilbert curve

Add Hilbert curve explanation



The first six iterations of the Hilbert curve by [Braindrain0000](#)

Here's a simple algorithm for drawing a Hilbert curve.¹

```
const HILBERT: &[&[usize]] = &[
    &[22, 10, 16, 38],
    &[10, 22, 24, 48],
    &[44, 36, 30, 18],
    &[36, 44, 42, 28],
];

fn curve(img: &mut Image, k: usize, order: i64, mut x: i64, mut y: i64) -> (i64, i64) {
    const STEP_SIZE: i64 = 5;
    let mut row: usize;
    let mut direction: usize;
    if order > 0 {
        for j in 0..4 {
            let step = HILBERT[k][j];
            row = (step / 10) - 1;
            let (xn, yn) = curve(img, row, order - 1, x, y);
            x = xn;
            y = yn;
            direction = step % 10;
            let prev = (x, y);
            match direction {
                8 => {
                    // null op
                }
                2 => {
                    // N
                    y -= STEP_SIZE;
                }
                1 => {

```

src/bin/hilbert.rs:



This code file is a PDF attachment

addendum

¹Griffiths, J. G. (1985). *Table-driven algorithms for generating space-filling curves*. Computer-Aided Design, 17(1), 37–41. doi:10.1016/0010-4485(85)90009-0

```

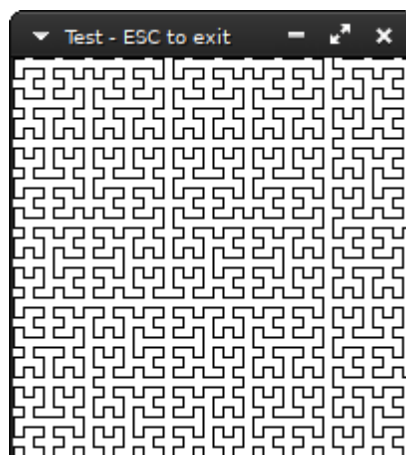
        // NE
        y -= STEP_SIZE;
        x += STEP_SIZE;
    }
    0 => {
        //E
        x += STEP_SIZE;
    }
    7 => {
        //SE
        x += STEP_SIZE;
        y += STEP_SIZE;
    }
    6 => {
        //S
        y += STEP_SIZE;
    }
    5 => {
        //SW
        y += STEP_SIZE;
        x -= STEP_SIZE;
    }
    4 => {
        //W
        x -= STEP_SIZE;
    }
    3 => {
        //NW
        y -= STEP_SIZE;
        x -= STEP_SIZE;
    }
    other => unreachable!("{}", other),
}
img.plot_line_width(prev, (x, y), 0.);
}
}
(x, y)
}

```

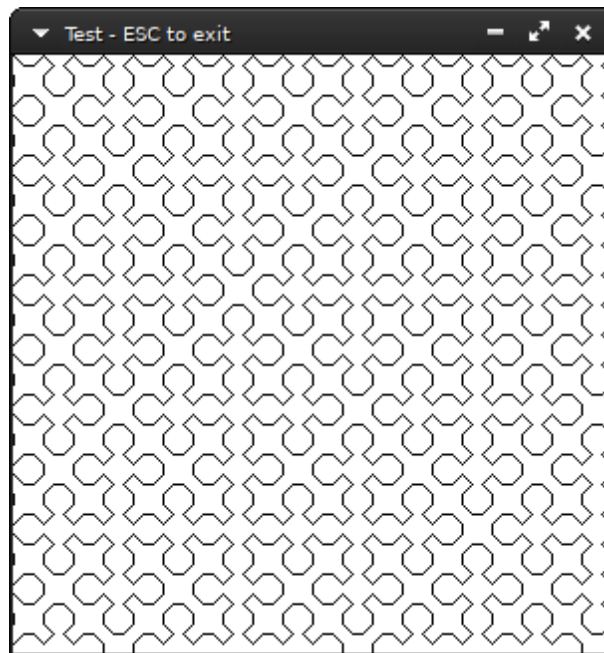
```

let mut image = Image::new(WINDOW_WIDTH, WINDOW_WIDTH, 0, 0);
curve(&mut image, 0, 7, 0, WINDOW_WIDTH as i64);

```



33.2 Sierpiński curve



Switching the table from the Hilbert implementation to this:

```
const SIERP: &[[usize]] = &[
    &[17, 25, 33, 41],
    &[17, 20, 41, 18],
    &[25, 36, 17, 28],
    &[33, 44, 25, 38],
    &[41, 12, 33, 48],
];
```

And switching two lines from the function to

```
- let step = HILBERT[k][j];
- row = (step / 10) - 1;
+ let step = SIERP[k][j];
+ row = (step / 10);
```

You can draw a Sierpinshi curve of order n by calling `curve(&mut image, 0, n+1, 0, 0)`.

33.3 Peano curve

Add Peano curve

[Redacted text block 1]

[Redacted text block 2]

[Redacted text block 3]

[Redacted text block 4]

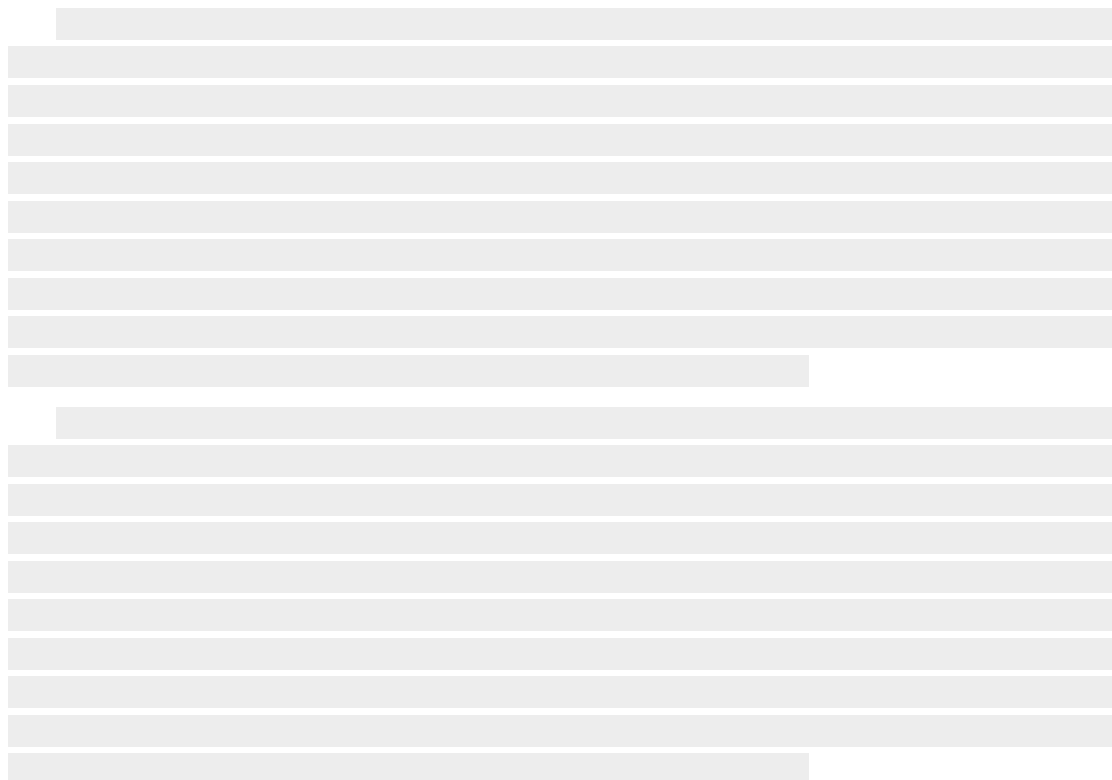


33.4 Z-order curve

Add Z-order curve

33.5 flowsnake curve

Add flowsnake curve



24

[Redacted text block 1]

[Redacted text block 2]

[Redacted text block 3]

Chapter 34

Dithering

Chapter 35

Marching squares

Index

centroid, 67, 77

circle out of three points, 54

circle out of two points, 54

midpoint, 33

shearing, 91

skewing, *see* shearing

About this text

The text has been typeset in \LaTeX 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.

Todo list

| | |
|---|-----|
| Add code samples in <i>Distance from a point to a line</i> | 27 |
| Add <i>Angle between two lines</i> | 29 |
| Add <i>Intersection of two lines</i> | 31 |
| Add <i>Normal to a line through a point</i> | 35 |
| Add some explanation behind the algorithm in <i>Drawing a line segment from its two endpoints</i> | 40 |
| Add code sample in <i>Intersection of two line segments</i> | 43 |
| Add <i>Equations of a circle</i> | 51 |
| Add <i>Parametric elliptical arcs</i> | 61 |
| Add <i>Union, intersection and difference of polygons</i> | 65 |
| Add <i>Centroid of polygon</i> | 67 |
| Add <i>Flood filling</i> | 71 |
| Add <i>Fast 2D Rotation</i> | 79 |
| Add <i>90° Rotation of a bitmap by parallel recursive subdivision</i> | 81 |
| Add <i>Smoothing enlarged bitmaps</i> | 85 |
| Add <i>Stretching lines of bitmaps</i> | 86 |
| Add screenshots and figure and code in <i>Mirroring</i> | 89 |
| Add <i>Projections</i> | 95 |
| Add <i>Faster Drawing a line segment from its two endpoints using Symmetry</i> | 99 |
| Add <i>Joining the ends of two wide line segments together</i> | 101 |
| Add <i>Composing monochrome bitmaps with separate alpha channel data</i> | 103 |
| Add <i>Orthogonal connection of two points</i> | 105 |
| Add <i>Join segments with round corners</i> | 107 |

| | |
|--------------------------------------|-----|
| Add <i>Faster line clipping</i> | 109 |
| Add <i>Space-filling Curves</i> | 111 |
| Add <i>Hilbert curve</i> explanation | 113 |
| Add <i>Peano curve</i> | 115 |
| Add <i>Z-order curve</i> | 117 |
| Add <i>flowsnake curve</i> | 117 |