
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	8	intro
Points And Lines	18	lines
Points and Line Segments	34	segments
Points, Lines and Circles	41	circles
Curves other than circles	49	curves
Points, Lines and Shapes	57	shapes
Vectors, matrices and transformations	67	trans- forma- tions
Addendum	82	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.

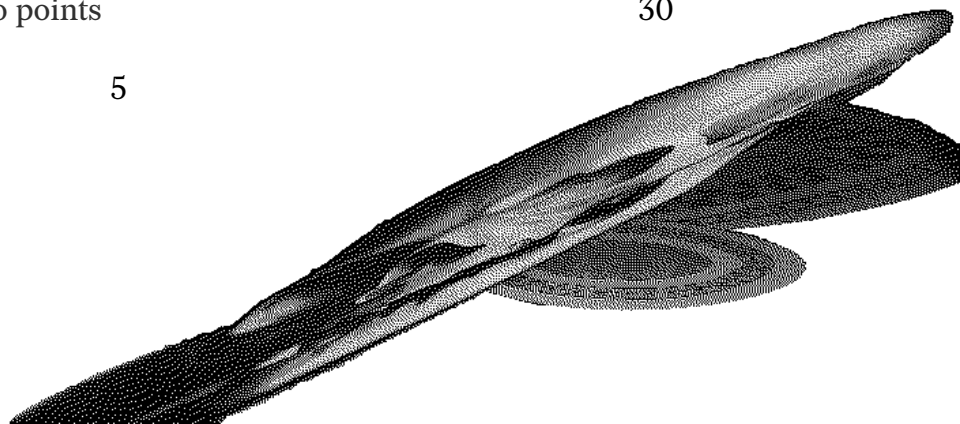
The source code for this work is available under the GNU GENERAL PUBLIC LICENSE version 3 or later. You can view it, study it, modify it for your purposes as long as you respect the license if you choose to distribute your modifications.

The source code is available here

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

Contents

I	Introduction	9
1	Data representation	10
2	Displaying pixels to your screen	12
3	Bits to byte pixels	14
4	Loading graphics files in Rust	15
5	Including xbm files in Rust	16
II	Points And Lines	19
6	Distance between two points	20
7	Moving a point to a distance at an angle	21
8	Equations of a line	22
8.1	Line through a point $P = (x_p, y_p)$ and a slope m	22
8.2	Line through two points	23
9	Distance from a point to a line	24
9.1	Using the implicit equation form	24
9.2	Using an L defined by two points P_1, P_2	25
9.3	Using an L defined by a point P_l and angle $\hat{\theta}$	25
9.4	Find perpendicular to line that passes through given point	25
10	Angle between two lines	26
11	Intersection of two lines	28
12	Line equidistant from two points	30



13	Normal to a line through a point	32
14	Angle Sectioning	33
14.1	Bisection	33
14.2	Trisection	33
III	Points And Line Segments	35
15	Drawing a line segment from its two endpoints	36
16	Drawing line segments with width	38
17	Intersection of two line segments	40
17.1	<i>Fast</i> intersection of two line segments	40
IV	Points, Lines and Circles	42
18	Equations of a circle	44
19	Bounding circle	45
V	Curves other than circles	50
20	Parametric elliptical arcs	51
21	Squircle	52
22	Bézier curves	53
22.1	Quadratic Bézier curves	54
22.1.1	Drawing the quadratic	54
22.2	Cubic Bézier curves	58
22.3	Weighted Béziers	58
VI	Points, Lines and Shapes	59
23	Rectangles and parallelograms	60
23.1	From a center point	60
23.2	From a corner point	60
24	Triangles	61

24.1	Making a triangle from a point and given angles	61
25	Union, intersection and difference of polygons	62
26	Centroid of polygon	63
27	Polygon clipping	64
28	Triangle filling	65
29	Flood filling	67
VII	Vectors, matrices and transformations	68
30	Rotation of a bitmap	69
30.1	Fast 2D Rotation	73
31	90° Rotation of a bitmap by parallel recursive subdivision	74
32	Magnification/Scaling	75
32.1	Smoothing enlarged bitmaps	76
32.2	Stretching lines of bitmaps	76
33	Mirroring	78
34	Shearing	79
34.1	The relationship between shearing factor and angle	81
35	Projections	82
VIII	Addendum	83
35.1	Faster Drawing a line segment from its two endpoints using Symmetry	84
36	Joining the ends of two wide line segments together	85
37	Composing monochrome bitmaps with separate alpha channel data	86
38	Orthogonal connection of two points	87
39	Join segments with round corners	88
40	Faster line clipping	92
41	Tilings	93
41.1	Hexagon Tiling	93
42	Space-filling Curves	94

42.1	Hilbert curve	95
42.2	Sierpiński curve	97
42.3	Peano curve	97
42.4	Z-order curve	98
42.5	Flowsnake curve	101
43	Dithering	103
43.1	Floyd–Steinberg	104
43.2	Atkinson dithering	106
44	Marching squares	108
	Index	109



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).

src/lib.rs:



This code file is a PDF attachment

```
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 magick_open(path: &str, x_offset: usize, y_offset: usize) -> Result<Self,
↳ Box<dyn Error>>;
    pub fn from_xbm(path: &str, x_offset: usize, y_offset: usize) -> Result<Self, Box<dyn
↳ Error>>;
    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);
}
```

intro

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();

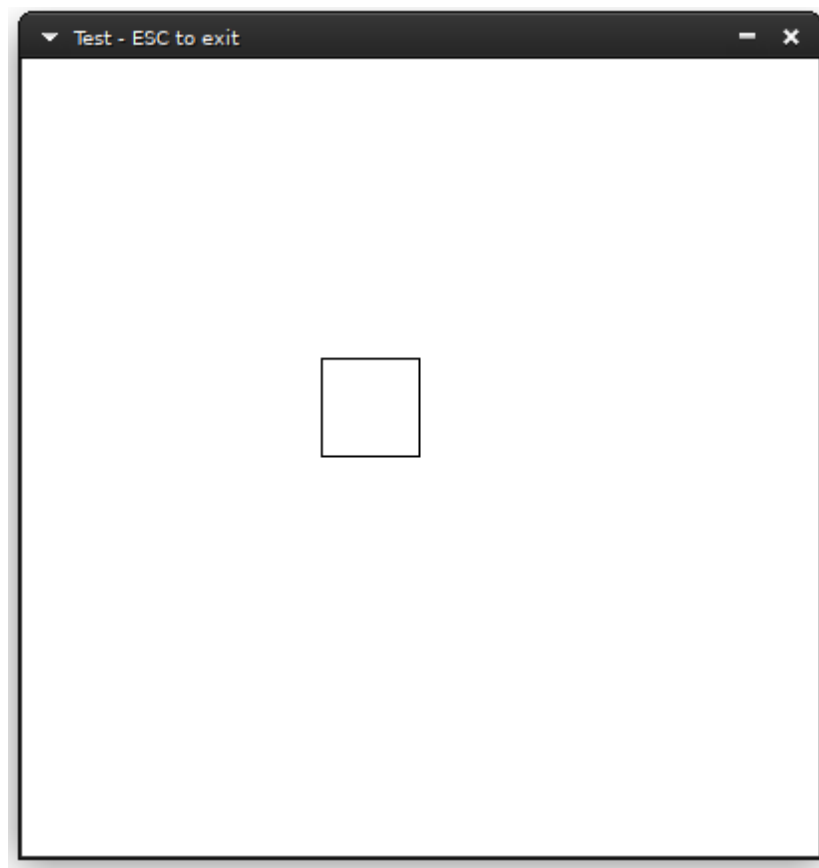
    // Limit to max ~60 fps update rate
    window.limit_update_rate(Some(std::time::Duration::from_micros(16600)));

    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  
}
```

Chapter 4

Loading graphics files in Rust

The book's library includes a method to load xbm files on runtime (see *Including xbm files in Rust* for including them in your binary at compile time). If your system has ImageMagick installed and the commands `identify` and `magick` are in your `PATH` environment variable, you can use the `Image::magick_open` method:

```
impl Image {  
    ...  
    pub fn magick_open(path: &str, x_offset: usize, y_offset: usize) -> Result<Self,  
↳   Box<dyn Error>>;  
    ...  
}
```

It simply converts the image file you pass to it to raw bytes using the invocation `magick convert path RGB:-` which prints raw RGB content to `stdout`.

If you have another way to load pictures such as your own code or a picture format library crate, all you have to do is convert the pixel information to an `Image` whose definition we repeat here:

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

Chapter 5

Including 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 `]`.

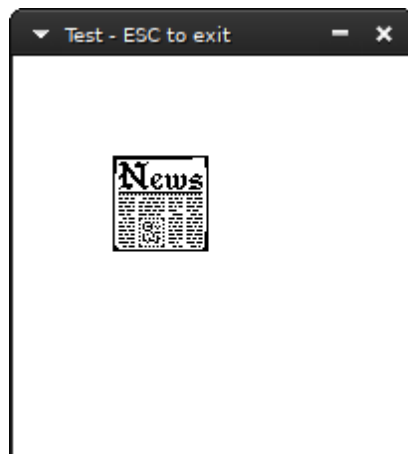
We can then include the new file in our source code:


```
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:



intro

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 xbmtools -- file.xbm > file.xbm.rs
```

```
use regex;  
use regex::Regex;  
use std::fs::File;  
use std::io::prelude::*;  
  
fn main() {  
    let args = std::env::args().skip(1).collect::<Vec<String>>();  
    if args.len() != 1 {  
        println!("one argument expected, the xbm file path to convert.");  
        return;  
    }  
    let mut file = match File::open(&args[0]) {  
        Err(err) => panic!("couldn't open {}: {}", args[0], err),  
        Ok(file) => file,  
    };  
  
    let mut s = String::new();  
    if let Err(err) = file.read_to_string(&mut s) {  
        panic!("couldn't read {}: {}", args[0], err);  
    }  
  
    let re = Regex::new(  
        r"(?imax)  
        ^\s*\x23\s*define\s+(?P<i>.+?)_width\s+(?P<w>\d\d*)$  
    )".unwrap().unwrap();  
}
```

src/bin/xbmtors.rs:



This code file is a PDF attachment

```

\s*
~\s*\w23\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());
}

```

Part II

Points And Lines

Chapter 6

Distance between two points

lines



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

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

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

Moving a point to a distance at an angle

lines

Moving a point $P = (x, y)$ at distance d at an angle of r radians is solved with simple trigonometry:

$$P' = (x + d \times \cos r, y + d \times \sin r)$$

Why? The problem is equivalent to calculating the point of a circle with P as the center, d the radius at angle r and as we will later* see this is how the points of a circle are calculated.

**Equations of a circle* page 44

Chapter 8

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 :

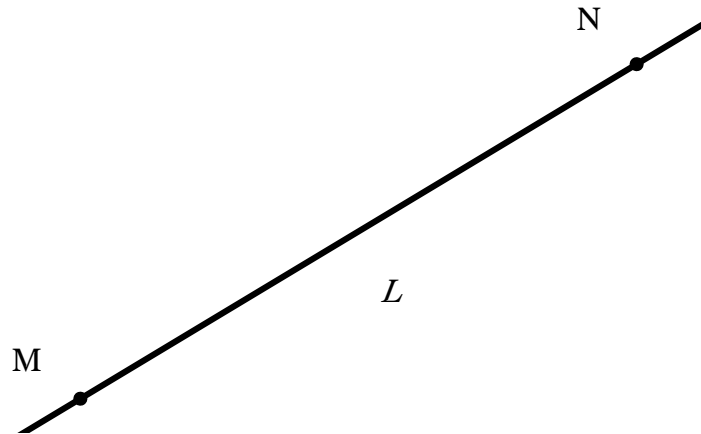
$$y = ax + b$$

The *parametric* form...

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

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

8.2 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 :

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

$$c = c_M, t \in R, c \in \{x, y\}$$

Substituting t in one of the equations we get:

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

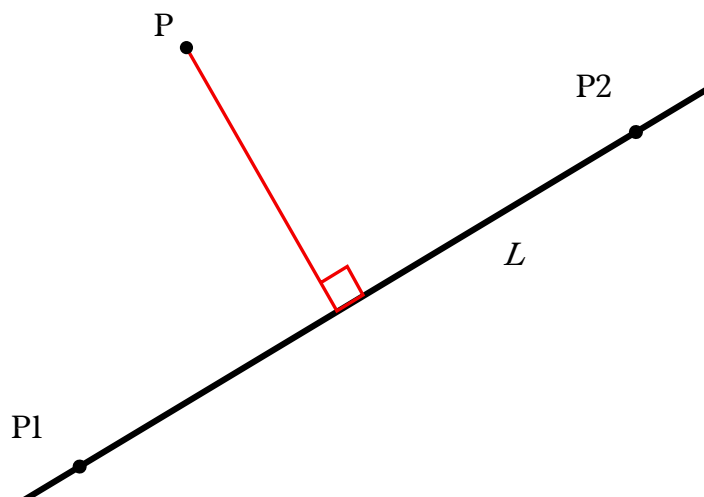
Which is what we were after. We finish by normalising what we found with $\frac{1}{\sqrt{a^2 + b^2}}$:

Chapter 9

Distance from a point to a line

lines

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



9.1 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 $ax + by = c$.

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

9.2 Using an L defined by two points P_1, P_2

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

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

9.3 Using an L defined by a point P_l and angle $\hat{\theta}$

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

9.4 Find perpendicular to line that passes through given point

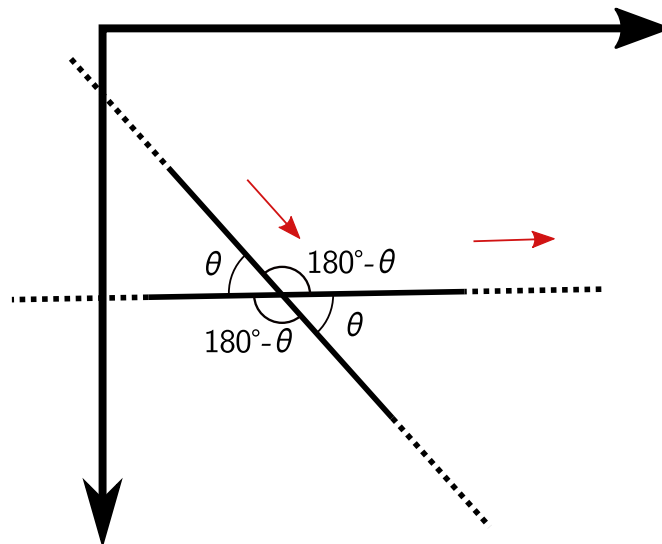
Now, we wish to find the equation of the line that passes through P and is perpendicular to L . Let's call it L_\perp . L in implicit form is $ax + by + c = 0$. The perpendicular will be:

$$L_\perp : bx - ay + (aP_y - bP_x) = 0$$

Chapter 10

Angle between two lines

lines



By angle we mean the angle formed by the two directions of the lines; and direction vectors start from the origin (in the figure, they are the **red arrows**). So if we want any of the other three angles, we already know them from basic geometry as shown in the figure above.

If you prefer using the implicit equation, bring the two lines L_1 and L_2 to that form $(a_1x + b_1y + c = 0$ and $a_2x + b_2y + c_2 = 0)$ and you can directly find $\hat{\theta}$ with the formula:

$$\hat{\theta} = \arccos \frac{a_1a_2 + b_1b_2}{\sqrt{(a_1^2 + b_1^2)(a_2^2 + b_2^2)}}$$

For the following parametric equations of L_1, L_2 :

$$L_1 = (\{x = x_1 + f_1 t\}, \{y = y_1 + g_1 t\})$$

$$L_2 = (\{x = x_2 + f_2 s\}, \{y = y_2 + g_2 s\})$$

the formula is:

$$\hat{\theta} = \arccos \frac{f_1 f_2 + g_1 g_2}{\sqrt{(f_1^2 + g_1^2)(f_2^2 + g_2^2)}}$$

The code:

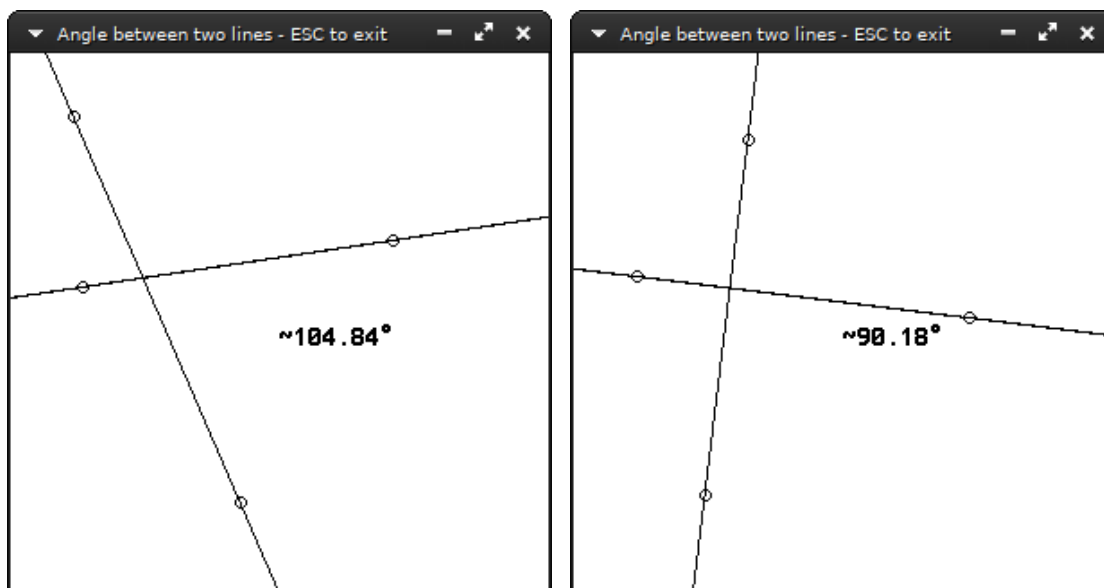
```
fn find_angle((a1, b1, c1): (i64, i64, i64), (a2, b2, c2): (i64, i64, i64)) -> f64 {
    let nom = (a1 * a2 + b1 * b2) as f64;
    let denom = ((a1 * a1 + b1 * b1) * (a2 * a2 + b2 * b2)) as f64;
    f64::acos(nom / f64::sqrt(denom))
}
```

src/bin/anglebe



lines

This code file is a PDF attachment

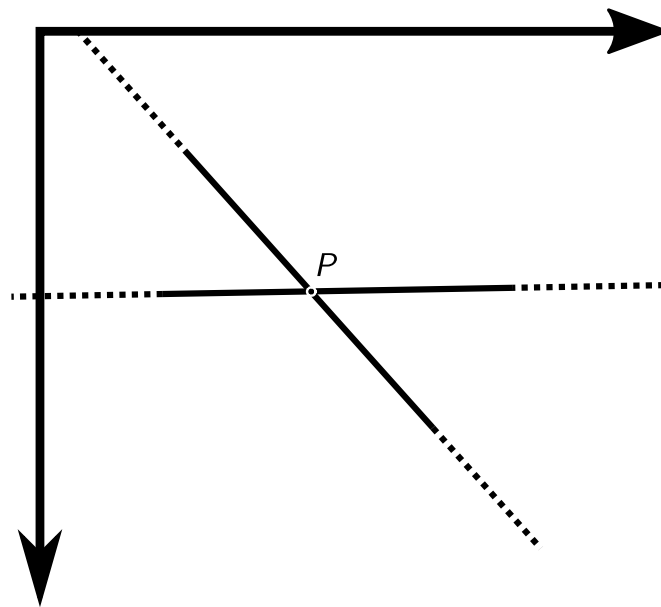


The `src/bin/anglebetweenlines.rs` example has two interactive lines and computes their angle with 64bit floating point accuracy.

Chapter 11

Intersection of two lines

lines



If the lines L_1, L_2 are in implicit form ($a_1x + b_1y + c = 0$ and $a_2x + b_2y + c_2 = 0$), the result comes after checking if the lines are parallel (in which case there's no single point of intersection):

$$a_1b_2 - a_2b_1 \neq 0$$

If they are not parallel, P is:

$$P = \left(\frac{b_1c_2 - b_2c_1}{a_1b_2 - a_2b_1}, \frac{a_2c_1 - a_1c_2}{a_1b_2 - a_2b_1} \right)$$

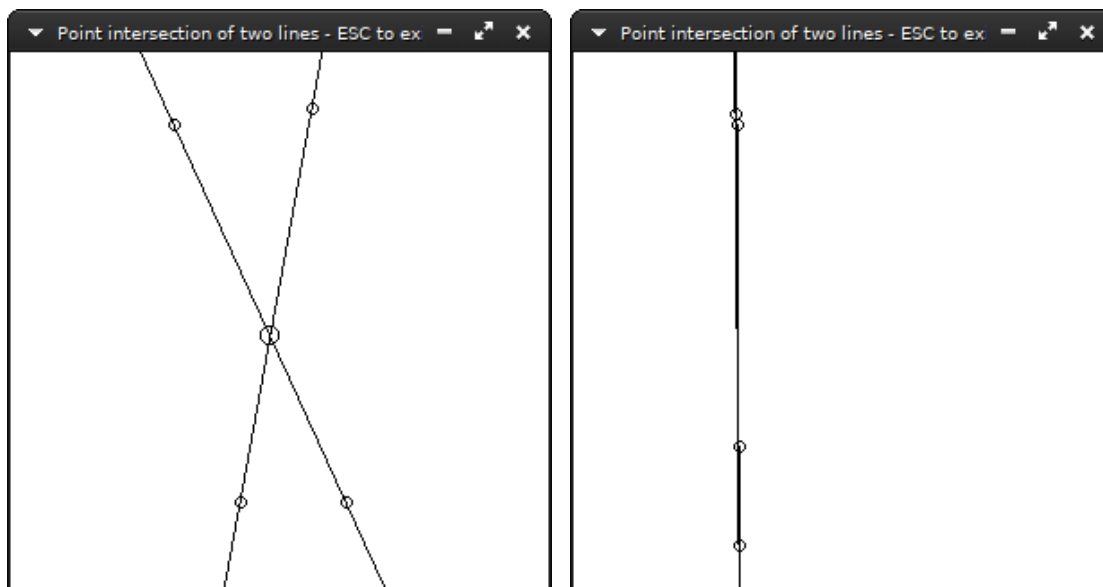
The code:

```
fn find_intersection((a1, b1, c1): (i64, i64, i64), (a2, b2, c2): (i64, i64, i64)) ->
  ↪ Option<Point> {
    let denom = a1 * b2 - a2 * b1;
    if denom == 0 {
      return None;
    }
    Some(((b1 * c2 - b2 * c1) / denom, (a2 * c1 - a1 * c2) / denom))
  }
```

src/bin/lineintersection.rs:



This code file is a PDF attachment



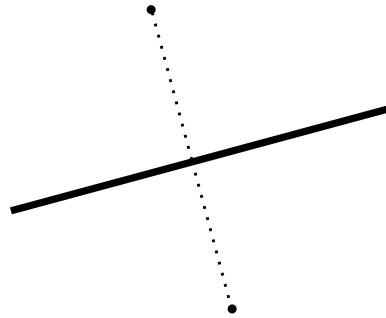
The src/bin/lineintersection.rs example has two interactive lines and computes their point of intersection.

lines

Chapter 12

Line equidistant from two points

lines



Let's name this line L . From previous chapter* we know how to get the line L that's created by the two points M and N :

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

We need the perpendicular line over the midpoint of L .[†] The midpoint also satisfies L 's equation. The midpoint's coordinates are intuitively:

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

The perpendicular's L_{\perp} equation is

$$L_{EQ} = L_{\perp} : yx - ay + (aP_{mid_y} - bP_{mid_x}) = 0$$

*See *Line through two points*, page 23

[†]See *Find perpendicular to line that passes through given point*, page 25

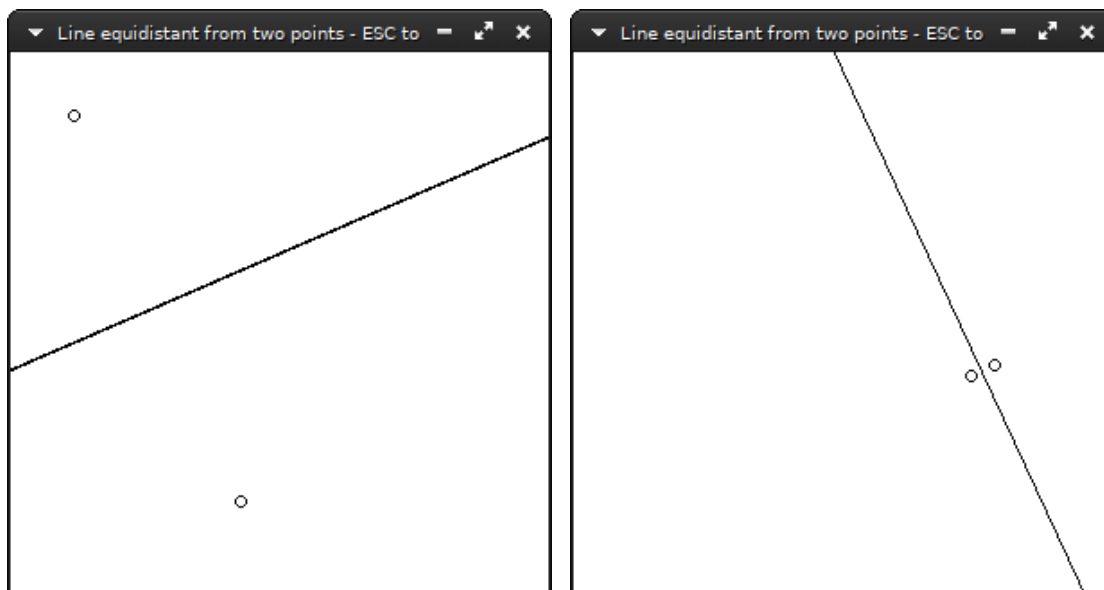
The code:

```
fn find_equidistant(point_a: Point, point_b: Point) -> (i64, i64, i64) {
    let (xa, ya) = point_a;
    let (xb, yb) = point_b;
    let midpoint = ((xa + xb) / 2, (ya + yb) / 2);
    let al = ya - yb;
    let bl = xb - xa;
    // If we had subpixel accuracy, we could do:
    //assert_eq!(al*midpoint.0+bl*midpoint.1, -cl);
    let a = bl;
    let b = -1 * al;
    let c = (al * midpoint.1) - (bl * midpoint.0);
    (a, b, c)
}
```

src/bin/equidistant.rs:



This code file is a PDF attachment



The `src/bin/equidistant.rs` example has two interactive points and computes their L_{EQ} .

lines

Chapter 13

Normal to a line through a point

lines

Add Normal to a line through a point

A series of horizontal gray bars, likely representing a list or table. The bars are arranged in a single column and are of varying lengths, suggesting a list of items or a table with multiple rows. The bars are light gray and have a thin black border.

Chapter 14

Angle Sectioning

lines

14.1 Bisection



14.2 Trisection

If the title startled you, be assured it's not a joke. It's totally possible to trisect an angle... with a ruler. The adage that angle trisection is impossible refers to using only a compass and unmarked straightedge.





lines

Part III

Points And Line Segments

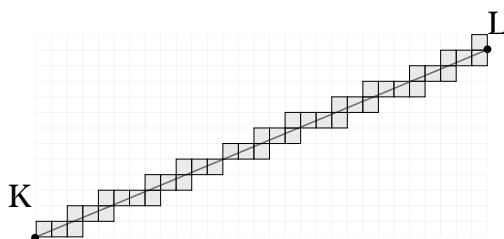
segments

Chapter 15

Drawing a line segment from its two endpoints

segments

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 };  
  
    if ax > ay {  
        d = 1 - ax;  
        x = x1;  
        y = y1;  
        while x != x2 {  
            if d < 0 {  
                d += ay;  
                x = x + sx;  
            } else {  
                d += ax;  
                y = y + sy;  
            }  
            self.plot(x, y);  
        }  
    } else {  
        d = 1 - ay;  
        x = x1;  
        y = y1;  
        while y != y2 {  
            if d < 0 {  
                d += ax;  
                x = x + sx;  
            } else {  
                d += ay;  
                y = y + sy;  
            }  
            self.plot(x, y);  
        }  
    }  
}
```

```

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;
  }
}
}

```

segments

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

Chapter 16

Drawing line segments with width

segments

```
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;
}
}
}
}

```

segments

Chapter 17

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*

17.1 Fast intersection of two line segments





segments

Part IV

Points, Lines and Circles

circles

[Redacted text block]

Chapter 18

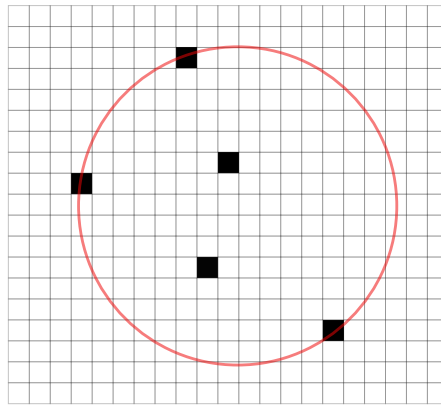
Equations of a circle

Add Equations of a circle

circles

Chapter 19

Bounding circle



src/bin/boundingcircle.rs:



This code file is a PDF attachment

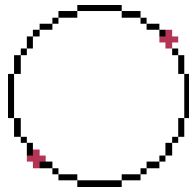
circles

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.



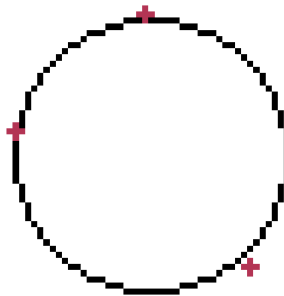
We can use the following methodology to find the bounding circle: start from two points and the circle they make up, and for each of the rest of the points check if the circle includes them. If not, make a bounding circle that includes every point up to the current one. To do this, we need some primitive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

```

circles

```

        } 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);
  }
}

```

circles

Part V

Curves other than circles

curves

Chapter 20

Parametric elliptical arcs

Add *Parametric elliptical arcs*



curves

Chapter 21

Squircle

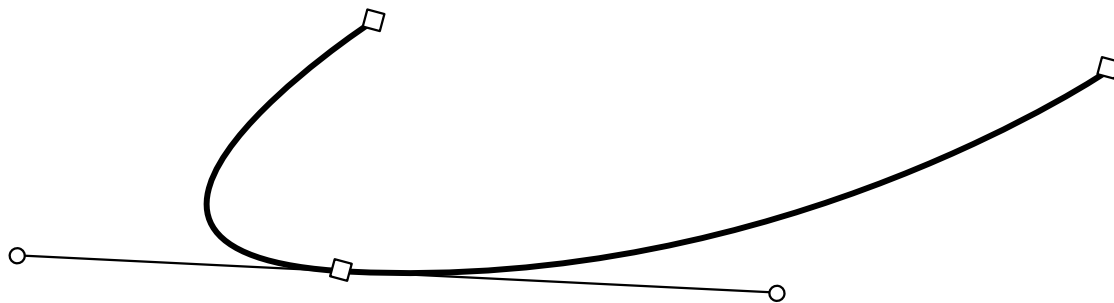
Add *Squircle*



curves

Chapter 22

Bézier curves



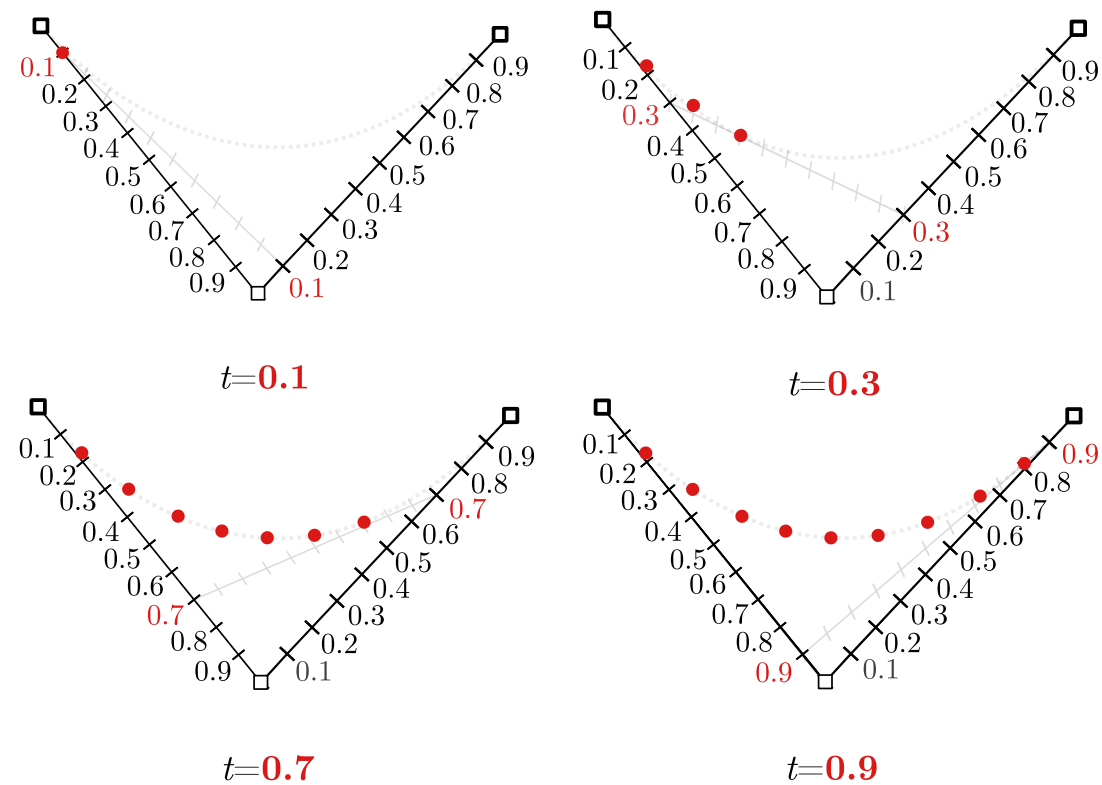
Two cubic *Bézier* curves joined together as displayed in graphics software.

curves

22.1 Quadratic Bézier curves

22.1.1 Drawing the quadratic

To actually draw a curve, i.e. with points P_1, P_2, P_3 we will use *de Casteljau's algorithm*. The gist behind the algorithm is that the length of the curve is visited at specific percentages (e.g. 0%, 0.2%, 0.4% ... 99.8%, 100%), meaning we will have that many steps, and for each such percentage t we calculate a line starting at the t -nth point of P_1P_2 and ending at the t -nth point of P_2P_3 . The t -eth point of that line also belongs to the curve, so we plot it.



Computing curve points for values of $t \in [0, 1]$ with de Casteljau's algorithm

src/bin/bezier.rs:

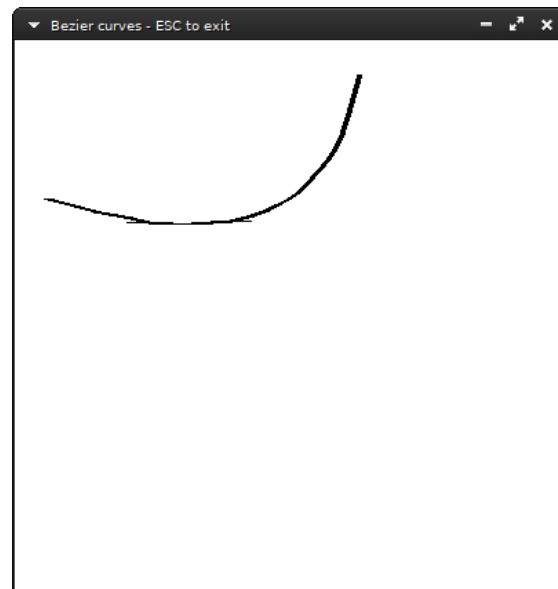


This code file is a PDF
attachment

Let's draw the curve $P_1 = (25, 115), P_2 = (225, 180), P_3 = (250, 25)$

```
|
```

The result:



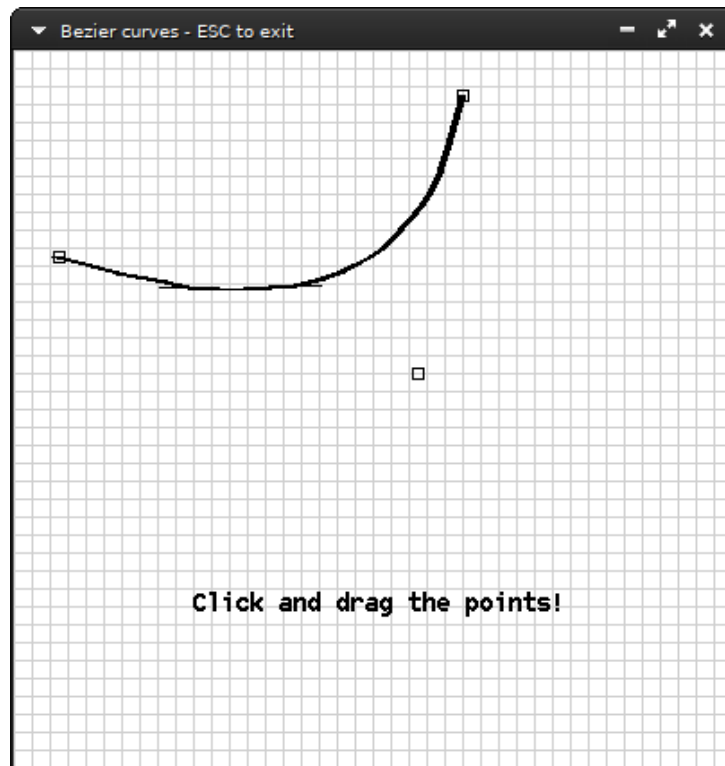
The `minifb` library allows to track user input, so we detect user clicks and the mouse's position; thus we can interactively modify a curve with some modifications in the code:

```
|
```

```

|
|
|
|
|
|
|
|
|
|
|
|
```

curves



Interactively modifying a curve with the `bezier.rs` tool.

curves

`src/bin/bezierglyph.rs`:

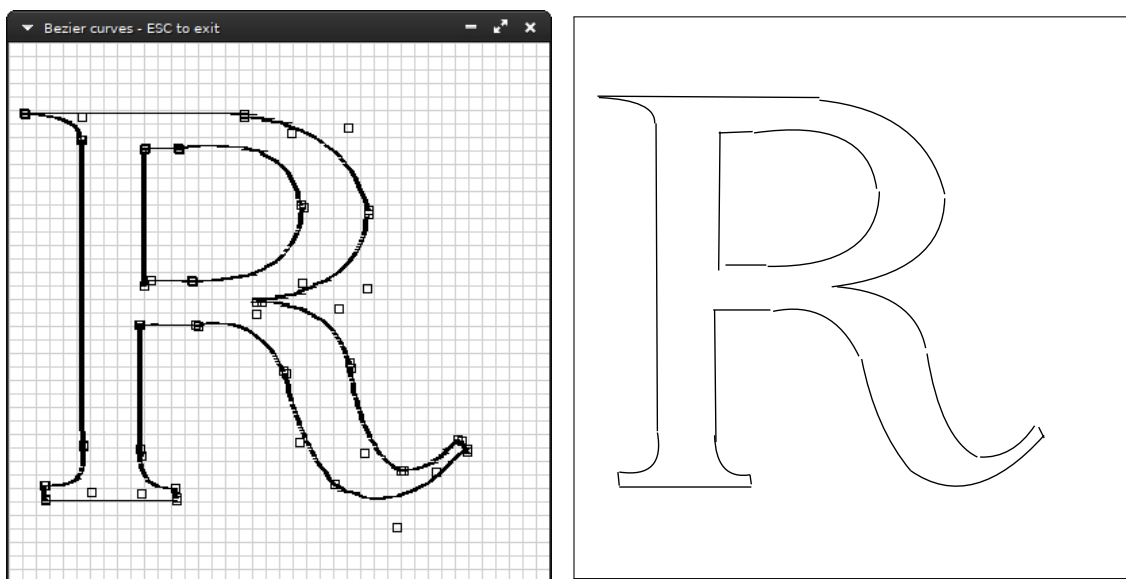


This code file is a PDF
attachment

We can go one step further and insult type designers* and use the tool to make a font glyph.

Of course, it requires effort to match the beginning and end of each curve that makes up the glyph. That's why font designing tools have *point snapping* to ensure curve continuation. But for a quick font designer app prototype, it's good enough.

*who use cubic Béziers or other fancier curves (*splines*)



Left: A font glyph drawn with the interactive `bezieryglyph.rs` tool. *Right:* the same glyph exported to SVG.

22.2 Cubic Bézier curves



22.3 Weighted Béziers



Part VI

Points, Lines and Shapes

shapes

Chapter 23

Rectangles and parallelograms



23.1 From a center point



23.2 From a corner point

Chapter 24

Triangles



24.1 Making a triangle from a point and given angles



shapes

Chapter 25

Union, intersection and difference of polygons

Add Union, intersection and difference of polygons



shapes

Chapter 26

Centroid of polygon

Add Centroid of polygon



shapes

Chapter 27

Polygon clipping



shapes

Chapter 28

Triangle filling

Add *Triangle filling* explanation

The book's library methods include a `fill_triangle` method:

This code is included in the distributed library file in the *Data representation* chapter.

```
pub fn fill_triangle(&mut self, q1: Point, q2: Point, q3: Point) {
    let make_equation =
        |p1: Point, p2: Point, p3: Point, a: &mut i64, b: &mut i64, c: &mut i64| {
            *a = p2.1 - p1.1;
            *b = p1.0 - p2.0;
            *c = p1.0 * p2.1 - p1.1 * p2.0;

            if *a * p3.0 + *b * p3.1 + *c < 0 {
                *a = -*a;
                *b = -*b;
                *c = -*c;
            }
        };

    let mut x_min = q1.0;
    let mut y_min = q1.1;
    let mut x_max = q1.0;
    let mut y_max = q1.1;
    let mut a = [0_i64; 3];
    let mut b = [0_i64; 3];
    let mut c = [0_i64; 3];

    // find bounding box
    for q in [q1, q2, q3] {
        x_min = std::cmp::min(x_min, q.0);
        x_max = std::cmp::max(x_max, q.0);
        y_min = std::cmp::min(y_min, q.1);
        y_max = std::cmp::max(y_max, q.1);
    }
    make_equation(q1, q2, q3, &mut a[0], &mut b[0], &mut c[0]);
    make_equation(q1, q3, q2, &mut a[1], &mut b[1], &mut c[1]);
    make_equation(q2, q3, q1, &mut a[2], &mut b[2], &mut c[2]);

    let mut d0 = a[0] * x_min + b[0] * y_min + c[0];
    let mut d1 = a[1] * x_min + b[1] * y_min + c[1];
    let mut d2 = a[2] * x_min + b[2] * y_min + c[2];

    for y in y_min..=y_max {
        let mut f0 = d0;
        let mut f1 = d1;
        let mut f2 = d2;

        d0 += b[0];
        d1 += b[1];
        d2 += b[2];

        for x in x_min..=x_max {
```

shapes

```
        if f0 >= 0 && f1 >= 0 && f2 >= 0 {  
            self.plot(x, y);  
        }  
        f0 += a[0];  
        f1 += a[1];  
        f2 += a[2];  
    }  
}
```

shapes

Chapter 29

Flood filling

Add Flood filling



shapes

Part VII

Vectors, matrices and transformations

Chapter 30

Rotation of a bitmap

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

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

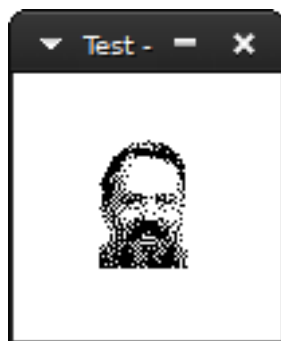
Let's load an xface. We will use `bits_to_bytes` (See *Bits to byte pixels*, page 14).

```
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



trans-
forma-
tions

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);
```

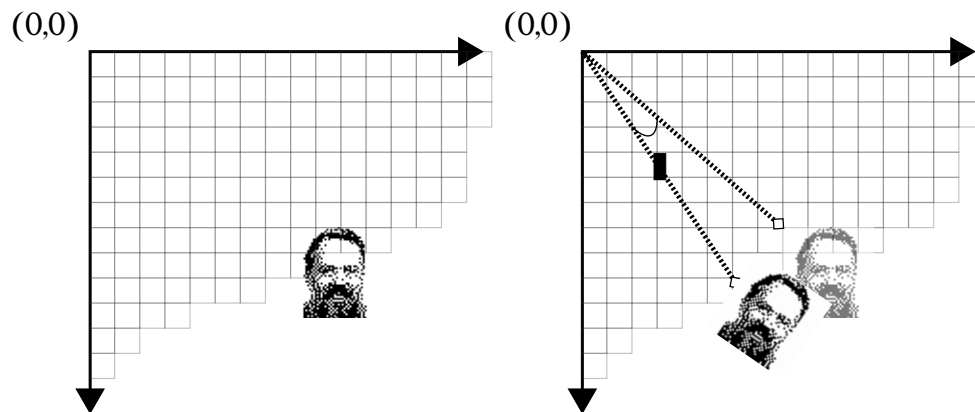
And then, loop for each byte in dmr's face and apply the rotation transformation.

```
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:



(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:

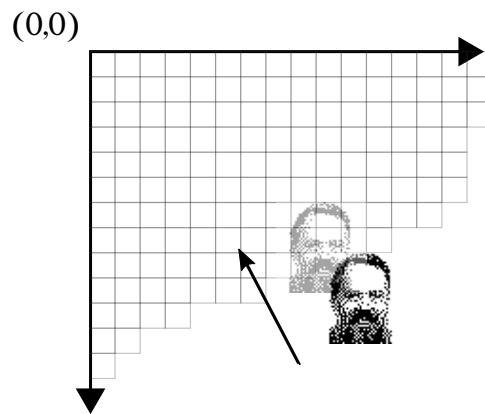
$$x_c = \frac{1}{n} \sum_{i=0}^n x_i$$

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

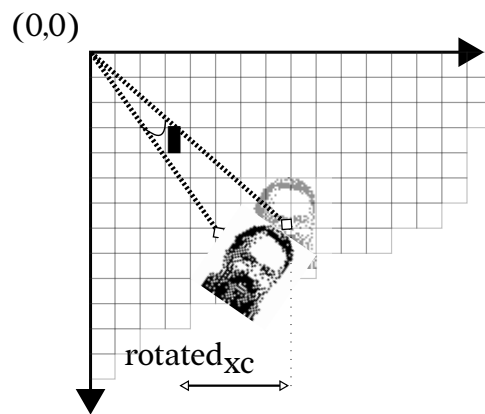
Since in this case we have a rectangle, the centroid has coordinates of half the width and half the height.

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

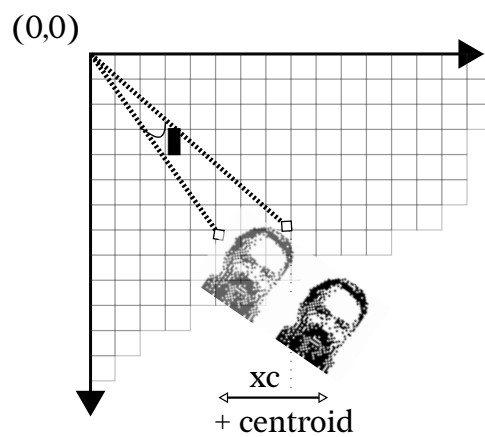
Here's it visually: First subtract the center point.



Then, rotate.



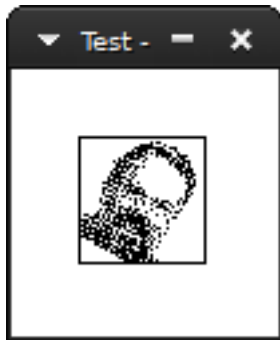
And subtract back to the original position.



trans-
forma-
tions

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:

30.1 Fast 2D Rotation

Add Fast 2D Rotation



trans-
forma-
tions

Chapter 31

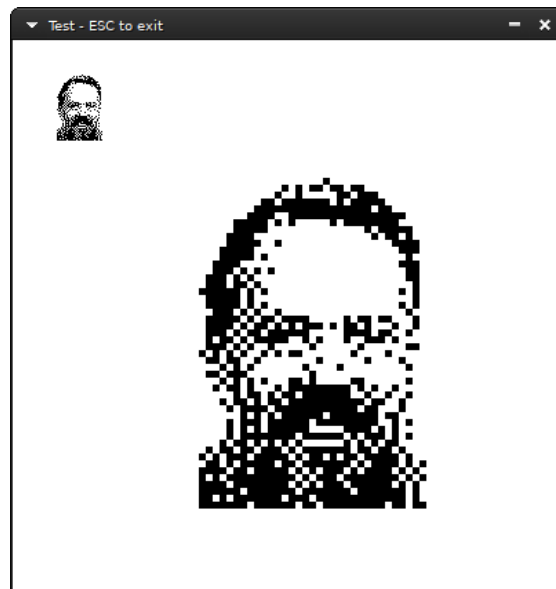
90° Rotation of a bitmap by parallel recursive subdivision

Add 90° Rotation of a bitmap by parallel recursive subdivision



Chapter 32

Magnification/Scaling



We want to magnify a bitmap without any smoothing. We define an Image scaled to the dimensions we want, and loop for every pixel in the scaled Image. Then, for each pixel, calculate its source in the original bitmap: if the coordinates in the scaled bitmap are (x, y) then the source coordinates (sx, sy) are:

$$sx = \frac{x * original.width}{scaled.width}$$
$$sy = \frac{y * original.height}{scaled.height}$$

So, if (sx, sy) are painted, then (x, y) must be painted as well.

src/bin/scale.rs:



This code file is a PDF attachment

```
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);
```

32.1 Smoothing enlarged bitmaps

Add Smoothing enlarged bitmaps



32.2 Stretching lines of bitmaps

Add Stretching lines of bitmaps





Chapter 33

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 34

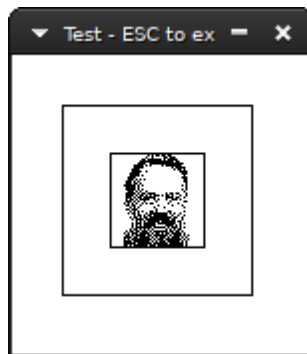
Shearing

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



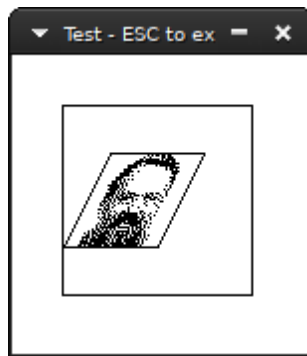
With l being equal to the desired tilt away from the y axis, the transformation is described by the following matrix:

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

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)  
}
```

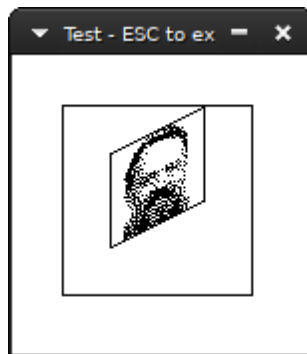
trans-
forma-
tions



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();
```

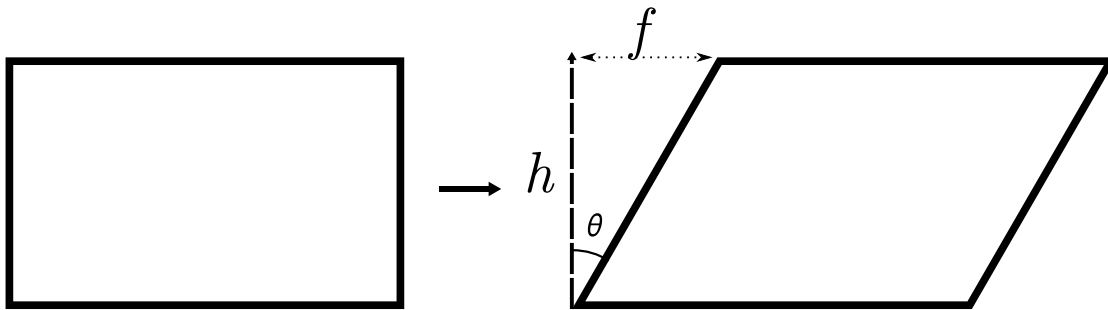


```

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();

```

34.1 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 35

Projections

Add Projections

Part VIII

Addendum

adden-
dum

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

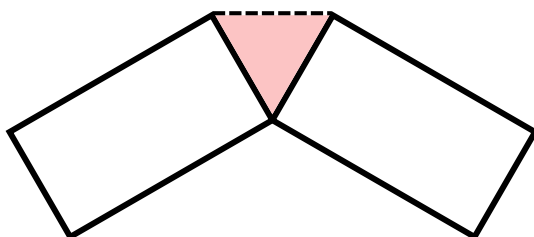
Add Faster Drawing a line segment from its two endpoints using Symmetry



Chapter 36

Joining the ends of two wide line segments together

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



A series of horizontal gray lines, likely representing a list or a set of data, arranged in a rectangular block.

Chapter 37

Composing monochrome bitmaps with separate alpha channel data

Add *Composing monochrome bitmaps with separate alpha channel data*



Chapter 38

Orthogonal connection of two points

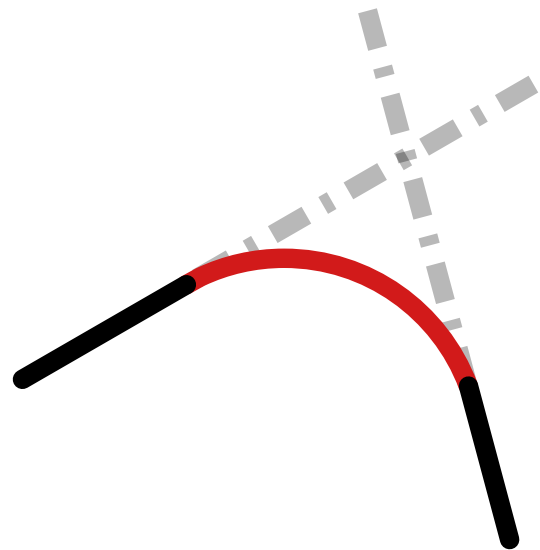
Add *Orthogonal connection of two points*

A series of horizontal gray bars, likely representing a list or table. The bars are arranged in a single column and are of varying lengths, suggesting a list of items or a table with multiple rows. The bars are light gray and have a thin black border.

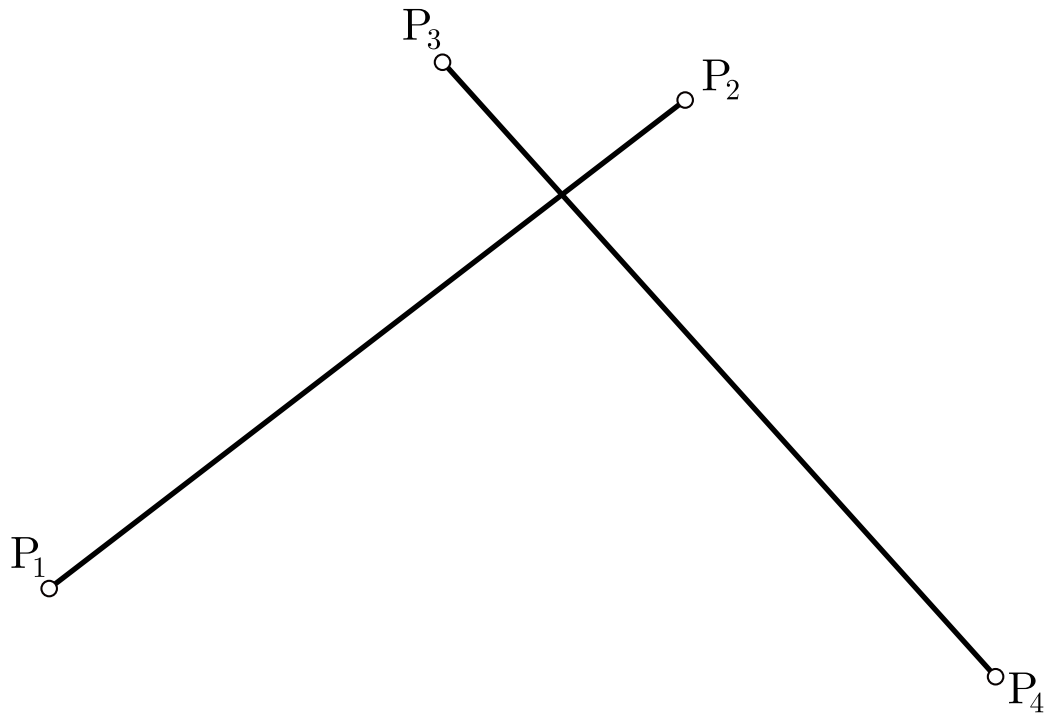
Chapter 39

Join segments with round corners

Round corners are everywhere around us. It is useful to know at least one method of construction. This specific method constructs a circle that has a common point with each given line segment, and calculates the arc that when added to the line segments they are smoothly joined. The excess length, since those common points will be before the end of the line segments, must be erased. Therefore, it's best to begin with just the points of the two segments before starting to draw anything.



Since the segments intercept, the round corner will end up beneath the intersection. We wish to find a circle that has a common point with each segment and the arc made up from those points and the circle is the round corner we are after.



We are given 4 points, P_1, P_2 and P_3, P_4 that make up segments S_1 and S_2 . Begin by finding the midpoints m_1 and m_2 of segments S_1 and S_2 . These will be:

$$m_1 = \frac{P_1 + P_2}{2}$$

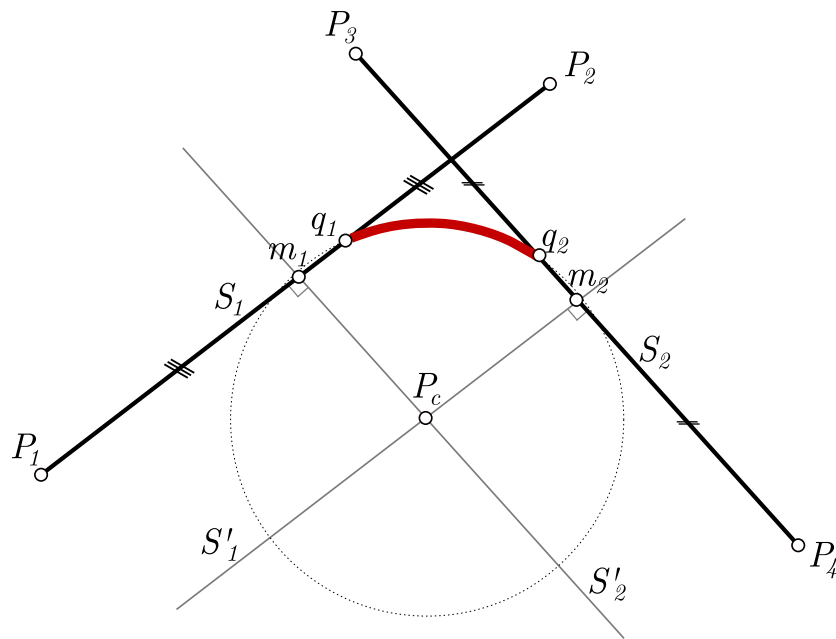
$$m_2 = \frac{P_3 + P_4}{2}$$

Then, find the signed distances (i.e. don't use the absolute value of distance) d_1 of m_1 from S_2 and d_2 of m_2 from S_1 .

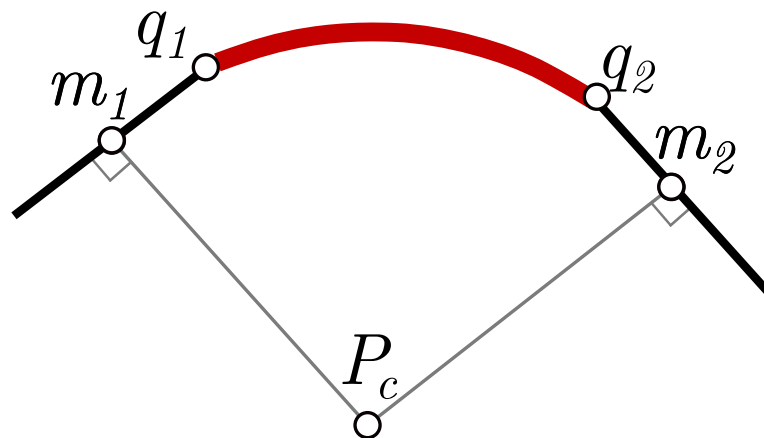
Construct parallel lines l_1 to S_1 that is d_1 pixels away. Repeat with l_2 for S_2 and d_2 .

Their intersection is the circle's center, P_c .

The intersection of l_1, l_2 with the two segments are the points where we should clip or extend the segments: q_1 and q_2 .



The starting angle is found by calculating the angle of q_1P_c with the x -axis with the `atan2` math library procedure.



addendum

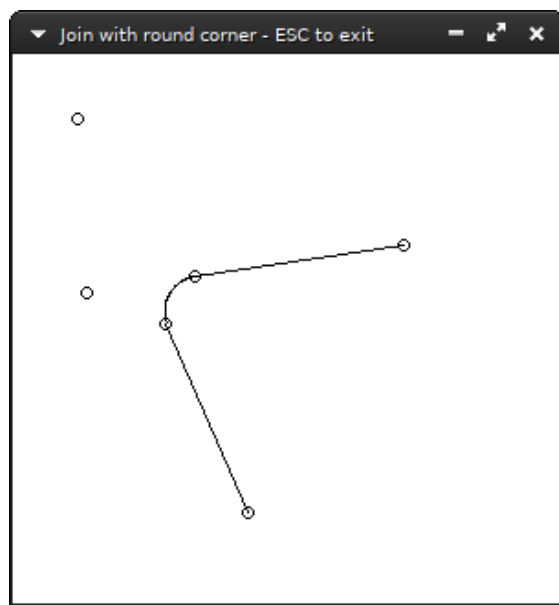
The *subtended* angle* of the arc from the center P_c is found by calculating the dot product of q_1P_c and q_2P_c :

The code:

src/bin/roundcorner.rs:



This code file is a PDF attachment



The src/bin/roundcorner.rs example has two interactive lines and computes the joining fillet.

addendum

*the *subtended* angle of an arc \widehat{AC} to a point P is the angle between PA and PC :



Chapter 40

Faster line clipping

Add *Faster line clipping*



Chapter 41

Tilings

Add Tilings

41.1 Hexagon Tiling

Chapter 42

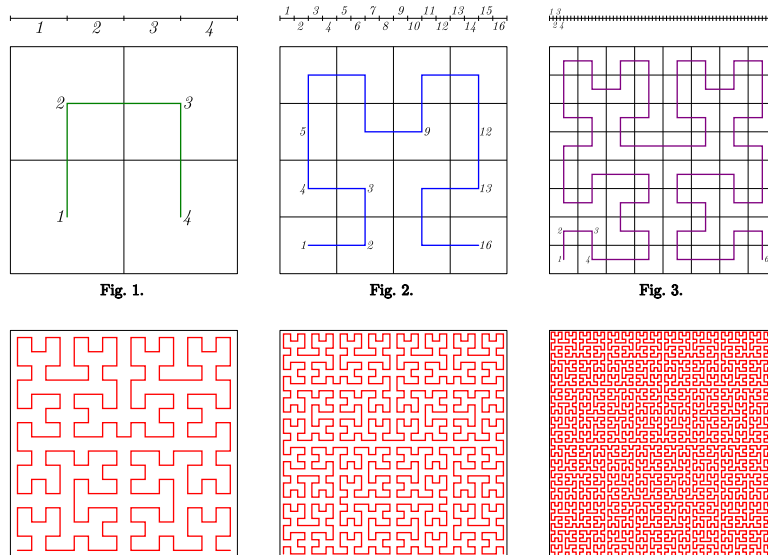
Space-filling Curves

Add Space-filling Curves



42.1 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.*

```
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

*Griffiths, 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

```

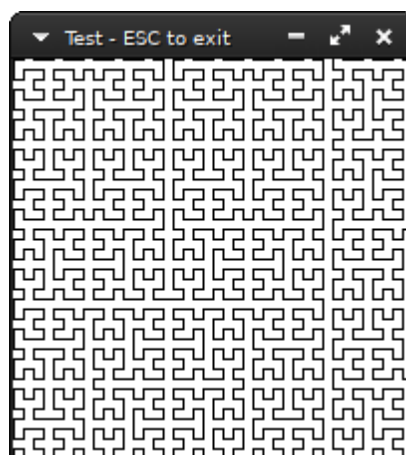
        // NE
        y -= STEP_SIZE;
        x += STEP_SIZE;
    }
    0 => {
        //E
        x += STEP_SIZE;
    }
    7 => {
        //SE
        x += STEP_SIZE;
        y += STEP_SIZE;
    }
    6 => {
        //S
        y += STEP_SIZE;
    }
    5 => {
        //SW
        y += STEP_SIZE;
        x -= 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)
}

```

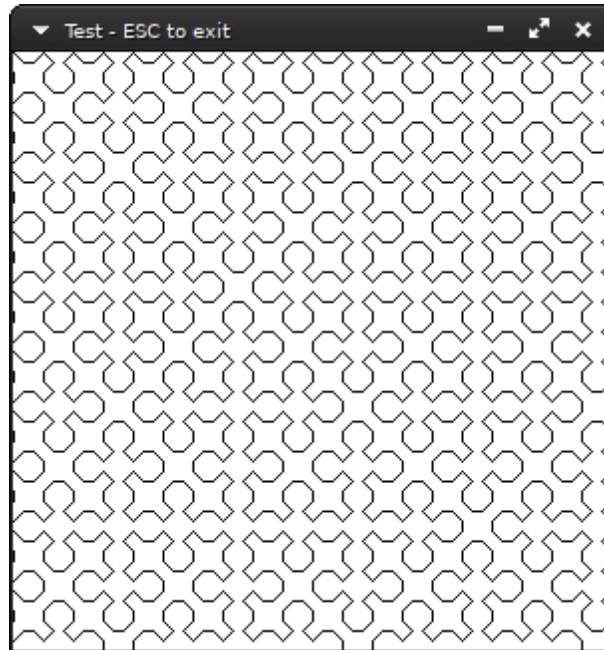
```

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

```



42.2 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);
```

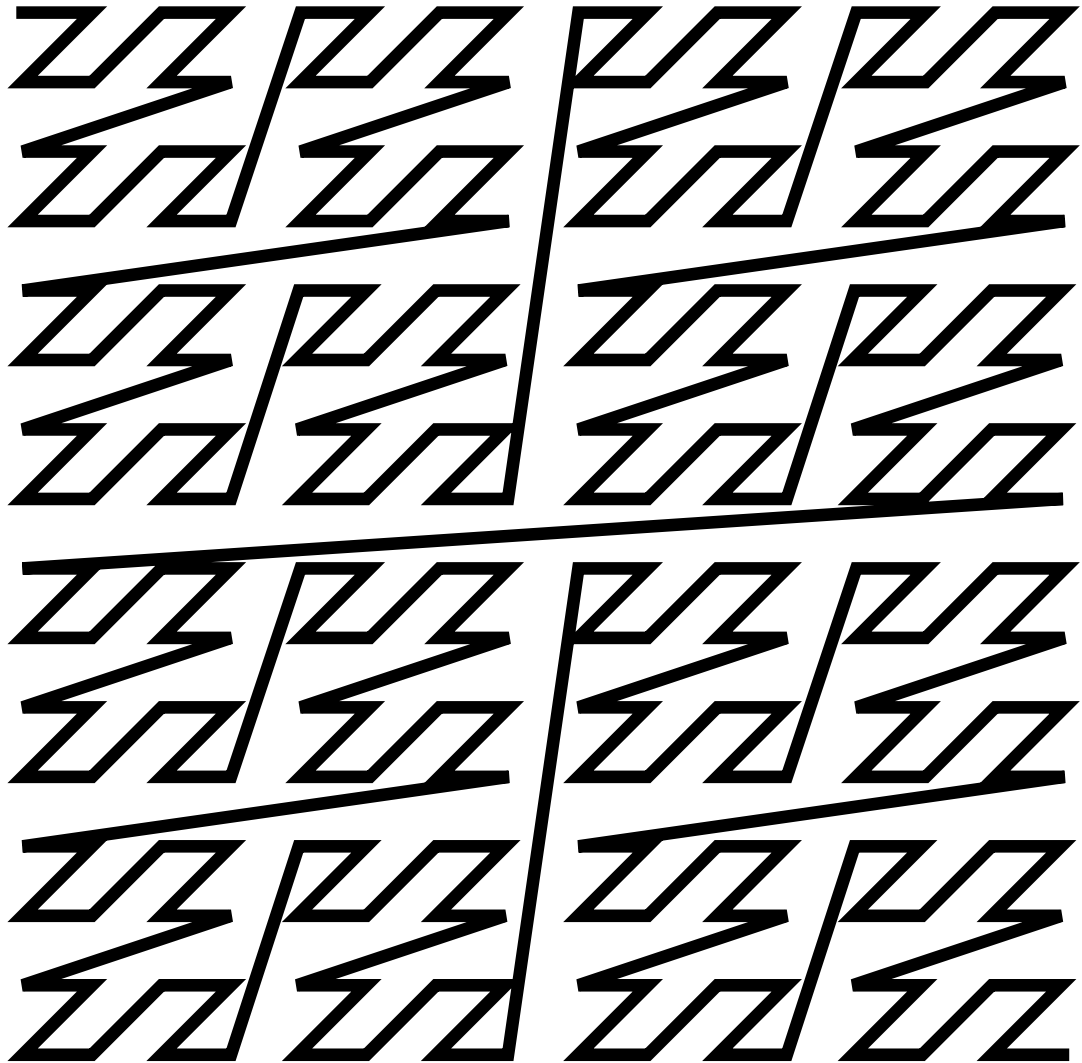
You can draw a Sierpinshi curve of order n by calling `curve(&mut image, 0, n+1, 0, 0)`.

42.3 Peano curve

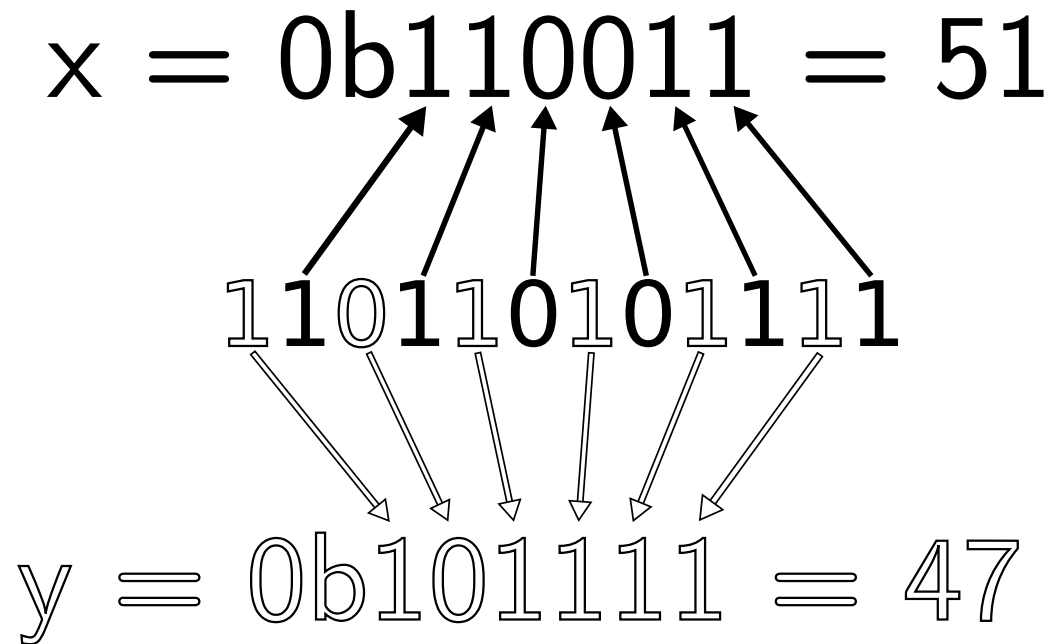
Add Peano curve

addendum

42.4 Z-order curve



Drawing the Z-order curve is really simple: first, have a counter variable that starts from zero and is incremented by one at each step. Then, you extract the (x,y) coordinates the new step represents from its binary representation. The bits for the x coordinate are located at the odd bits, and for y at the even bits. I.e. the values are interleaved as bits in the value of the step:



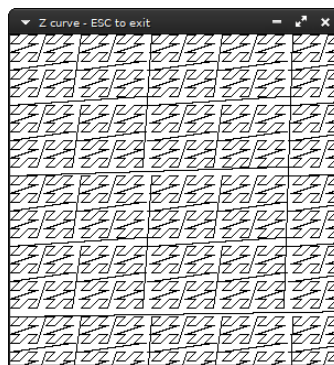
Knowing this, implementing the drawing process will consist of computing the next step, drawing a line segment from the current step and the next, set the current step as the next and continue;

```
fn zcurve(img: &mut Image, x_offset: i64, y_offset: i64) {
    const STEP_SIZE: i64 = 8;
    let mut sx: i64 = 0;
    let mut sy: i64 = 0;
    let mut b: u64 = 0;
    let mut prev_pos = (sx + x_offset, sy + y_offset);
    loop {
        let next = b + 1;
        sx = 0;
        if (next & 1) as i64 > 0 {
            sx += STEP_SIZE;
        }
        if next & 0b100 > 0 {
            sx += 2 * STEP_SIZE;
        }
        if next & 0b10_000 > 0 {
            sx += 4 * STEP_SIZE;
        }
        if next & 0b1_000_000 > 0 {
            sx += 8 * STEP_SIZE;
        }
        if next & 0b100_000_000 > 0 {
            sx += 16 * STEP_SIZE;
        }
        if next & 0b10_000_000_000 > 0 {
            sx += 32 * STEP_SIZE;
        }
        if next & 0b1_000_000_000_000 > 0 {
            sx += 64 * STEP_SIZE;
        }
        if next & 0b100_000_000_000_000 > 0 {
            sx += 128 * STEP_SIZE;
        }
    }
}
```

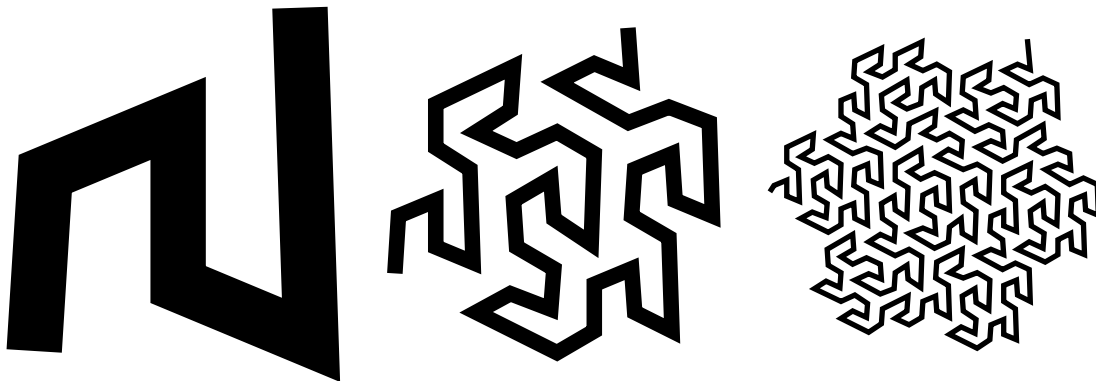
```

    }
    if next & 0b10_000_000_000_000_000 > 0 {
        sx += 256 * STEP_SIZE;
    }
    if next & 0b1_000_000_000_000_000_000 > 0 {
        sx += 512 * STEP_SIZE;
    }
    sy = 0;
    if (next & 0b10) as i64 > 0 {
        sy += STEP_SIZE;
    }
    if next & 0b1_000 > 0 {
        sy += 2 * STEP_SIZE;
    }
    if next & 0b100_000 > 0 {
        sy += 4 * STEP_SIZE;
    }
    if next & 0b10_000_000 > 0 {
        sy += 8 * STEP_SIZE;
    }
    if next & 0b1_000_000_000 > 0 {
        sy += 16 * STEP_SIZE;
    }
    if next & 0b100_000_000_000 > 0 {
        sy += 32 * STEP_SIZE;
    }
    if next & 0b10_000_000_000_000 > 0 {
        sy += 64 * STEP_SIZE;
    }
    if next & 0b1_000_000_000_000_000 > 0 {
        sy += 128 * STEP_SIZE;
    }
    if next & 0b100_000_000_000_000_000 > 0 {
        sy += 256 * STEP_SIZE;
    }
    if next & 0b10_000_000_000_000_000_000 > 0 {
        sy += 512 * STEP_SIZE;
    }
    img.plot_line_width(prev_pos, (sx + x_offset, sy + y_offset), 1.0);
    if next == 0b111_111_111_111_111_111_111 {
        break;
    }
    if sx as usize > img.width && sy as usize > img.height {
        break;
    }
    prev_pos = (sx + x_offset, sy + y_offset);
    b = next;
}
}

```



42.5 Flowsnake curve



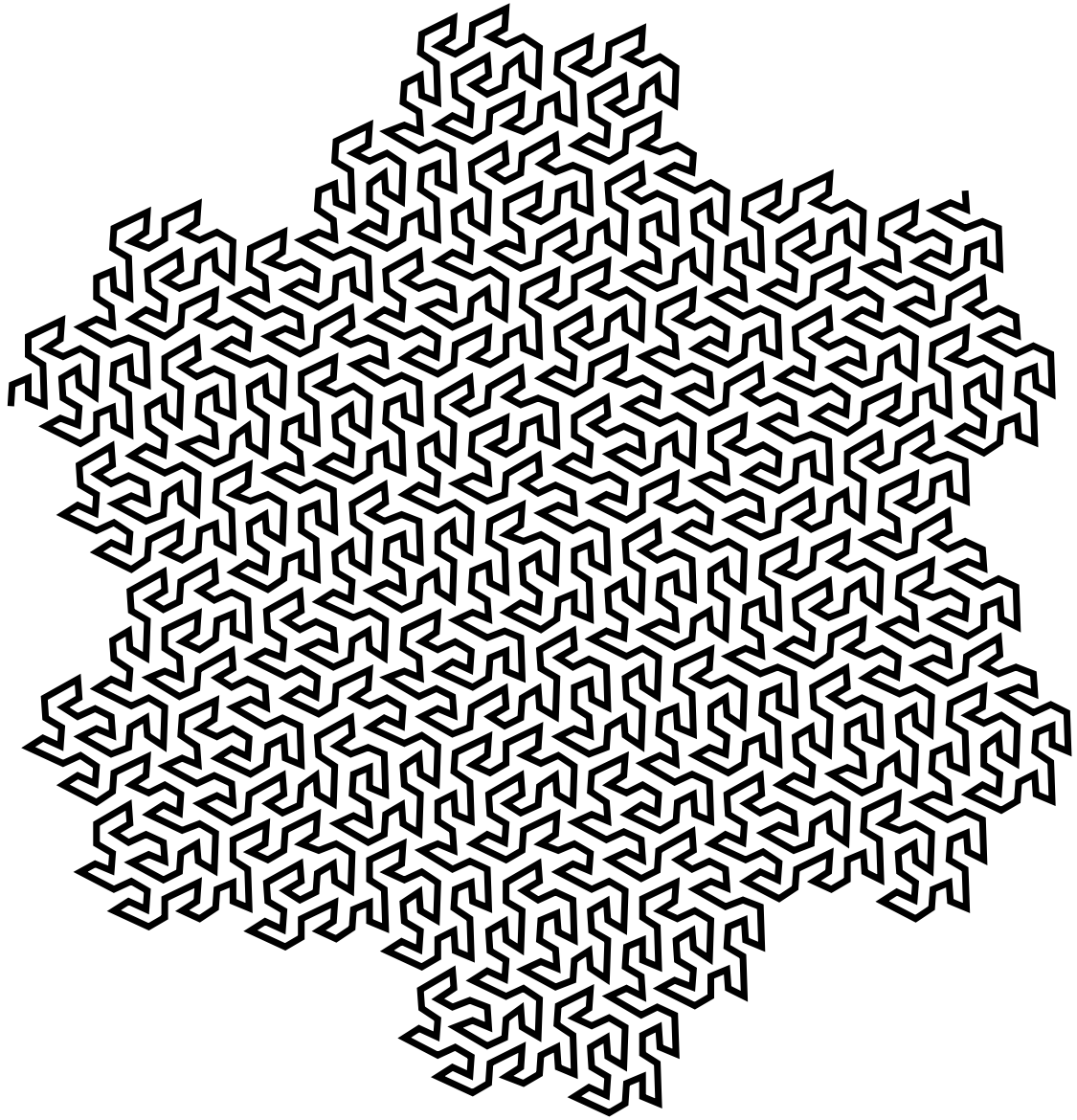
The first three orders of the Gosper curve.

As a fractal curve, the *flowsnake curve* or *Gosper curve* is defined by a set of recursive rules for drawing it. There are four kind of rules and two of them define rulesets (i.e. they are non-terminal steps).

$$A \mapsto A-B--B+A++AA+B-$$

$$B \mapsto +A-BB--B-A++A+B$$



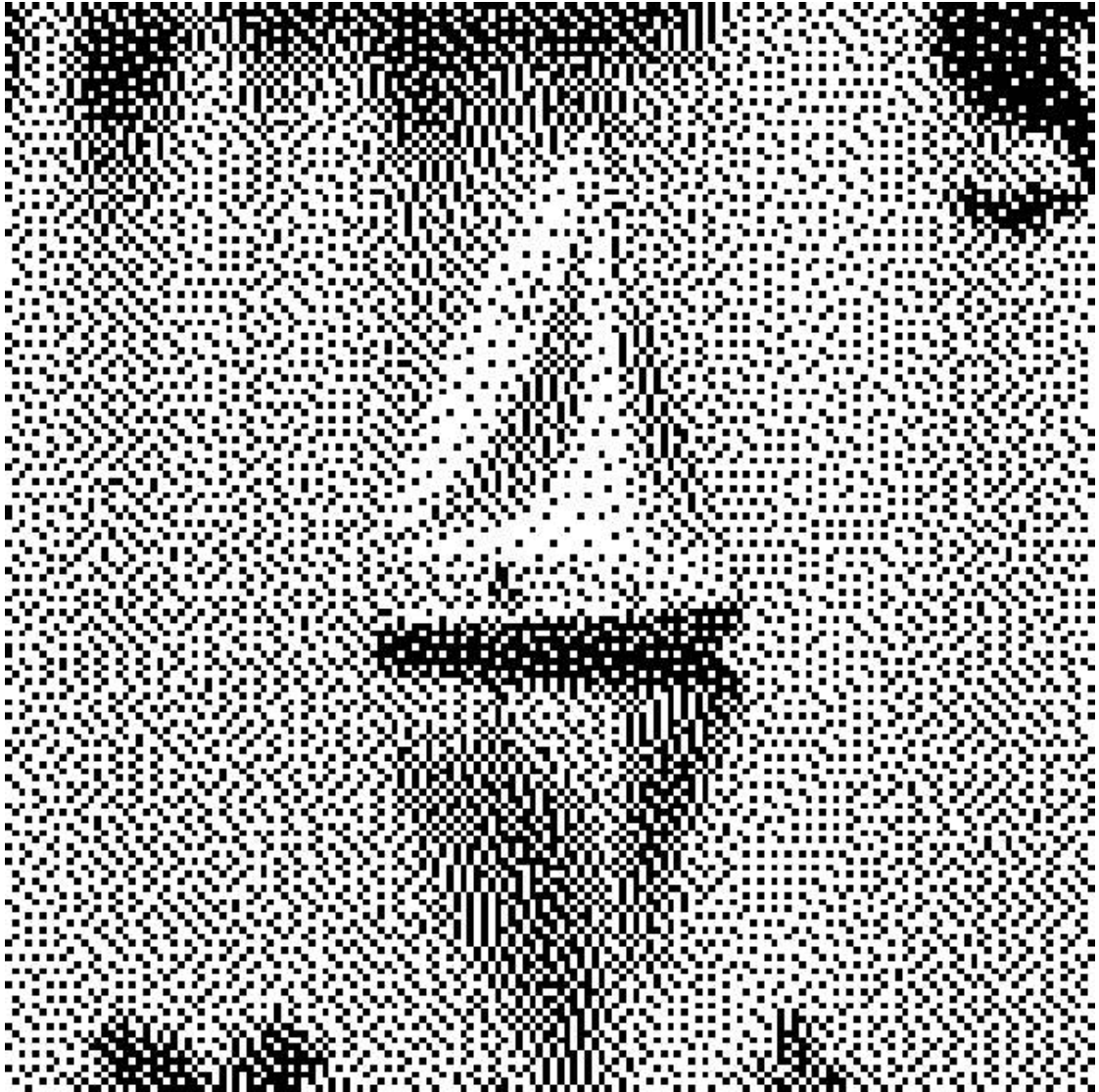


The fourth order Gosper curve consists of a minimum of 2057 distinct line segments (but our algorithm draws 36015)

Chapter 43

Dithering

43.1 Floyd-Steinberg



detail of a standard test image, [*Sailboat on lake*](#), with Floyd-Steinberg dithering


```

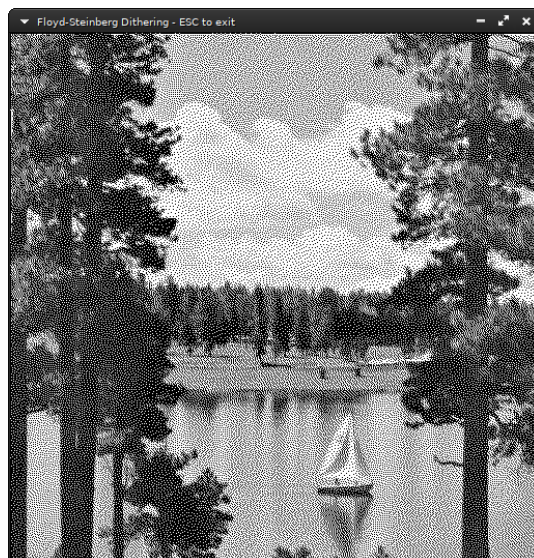
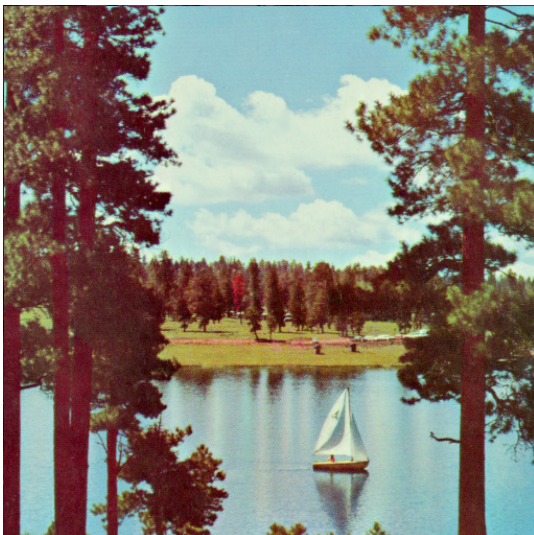
fn floyd(image: &mut Image) {
    let w = image.width;
    let m = [(0, 7), (w - 2, 3), (w - 1, 5), (w, 1)];
    let mut e = vec![0.0; w + 1];
    let bytes = image
        .bytes
        .iter()
        .map(|&byte| {
            let (r, g, b) = from_u32_rgb(byte);
            let g: f64 = (0.299 * (r as f64)) + (0.587_f64 * (g as f64)) + (0.114 * (b as
↪ f64));
            let pix = g / 255.0 + {
                e.push(0.);
                e.remove(0)
            };
            let col = if pix > 0.5 { 1. } else { 0. };
            let err = (pix - col) / 16.;
            for (x, y) in m.iter() {
                e[*x] += err * (*y as f64);
            }
            if col.floor() as u32 == 1 {
                WHITE
            } else {
                BLACK
            }
        })
        .collect::<Vec<u32>>();
    image.bytes = bytes;
}

```

src/bin/floyd_dither.rs:

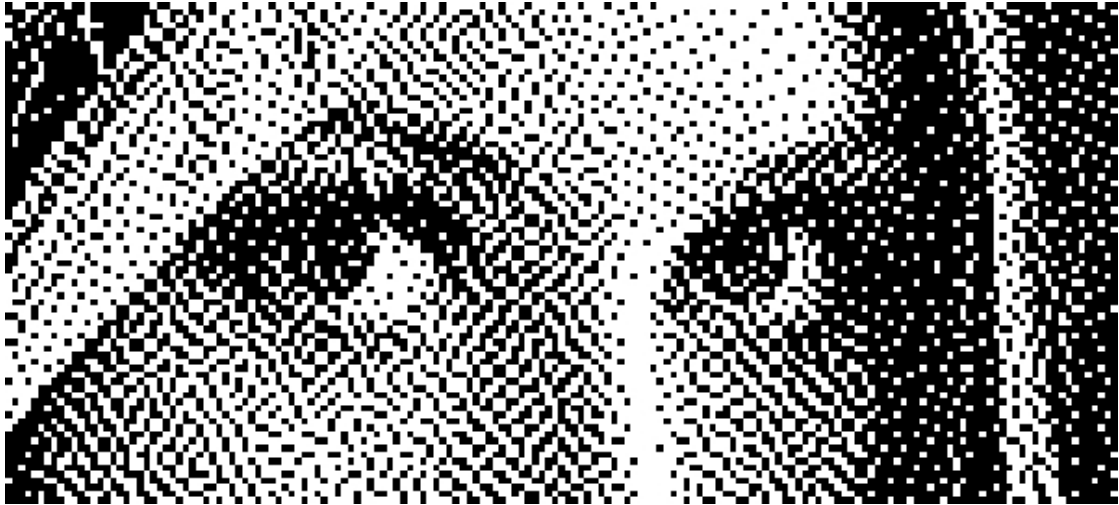


This code file is a PDF attachment



addendum

43.2 Atkinson dithering



detail of a standard test image, *Lenna*, with Atkinson dithering



The following code implements Atkinson dithering:*

```
fn atkinson(image: &mut Image) {
    let w= image.width;
    let mut e = vec![0.0;2*w];
    let m = [0, 1, w-2, w-1, w, 2*w-1];
    for byte in image.bytes.iter_mut() {
        let (r,g,b) = from_u32_rgb(*byte);
        let g:f64 = ((0.299*(r as f64)) ) + ((0.587_f64*(g as f64)) ) + ((0.114*(b as
↪ f64)) );
        let pix = g/255.0 + { e.push(0.); e.remove(0)};
        let col = if pix > 0.5 { 1. } else { 0. };
        let err = (pix-col)/8.;
        for m in m.iter() {
            e[*m] += err;
        }
        *byte = if (col.floor() as u32 == 1) {
            WHITE
        }
```

*Algorithm taken from <https://beyondloom.com/blog/dither.html>

src/bin/atkinsondither.rs:



This code file is a PDF
attachment

adden-
dum

```
    } else {  
      BLACK  
    };  
  }  
}
```



Chapter 44

Marching squares



Index

- alpha channel, 86
- angle
 - between two lines, 26
 - bisectioning, 33
 - trisectioning, 33
- area filling, *see* flood filling
- Atkinson dithering, 106
- bucket filling, *see* flood filling
- centroid
 - polygon, 63
 - rectangle, 71
- circle
 - bounding, 45
 - equations, 44
 - out of three points, 46
 - out of two points, 46
- contour, *see* marching squares
- curves
 - Bézier, 53
 - cubic, 58
 - quadratic, 54
 - weighted, 58
 - elliptical, 51
 - Flowsnake curve, 101
 - Hilbert curve, 95
 - Peano curve, 97
 - space-filling, 94
- de Casteljau's algorithm, 54
- distance
 - between two points, 20
 - moving a point, 21
 - point from a line, 24
- dithering, 103
 - Atkinson, 106
 - Floyd-Steinberg, 104
- equidistant line, 30
- flood filling, 67
 - triangle filling, 65
- Flowsnake curve, 101
- Floyd-Steinberg dithering, 104
- Gosper curve, *see* Flowsnake curve
- Hilbert curve, 95
- line
 - equations, 22
 - equidistant, 30
 - intersection, 28
 - through point and slope, 22
 - through two points, 23
- magnification, 75
- marching squares, 108
- midpoint, 30, 89
- Peano curve, 97
- perpendicular, 25
- polygon
 - boolean operations, 62
 - centroid, 63
 - clipping, 64

rotation, 69

scaling, 75

shearing, 79

skewing, *see* shearing

smoothing, 76

stretching, 76

triangle, 61

filling, 65

from point and angles, 61

About this text

The text has been typeset in $\text{X}_{\text{F}}\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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>	24
Add <i>Normal to a line through a point</i>	32
Add some explanation behind the algorithm in <i>Drawing a line segment from its two endpoints</i>	37
Add code sample in <i>Intersection of two line segments</i>	40
Add <i>Equations of a circle</i>	44
Add <i>Parametric elliptical arcs</i>	51
Add <i>Squircle</i>	52
Add <i>Union, intersection and difference of polygons</i>	62
Add <i>Centroid of polygon</i>	63
Add <i>Triangle filling</i> explanation	65
Add <i>Flood filling</i>	67
Add <i>Fast 2D Rotation</i>	73
Add <i>90° Rotation of a bitmap by parallel recursive subdivision</i>	74
Add <i>Smoothing enlarged bitmaps</i>	76
Add <i>Stretching lines of bitmaps</i>	76
Add screenshots and figure and code in <i>Mirroring</i>	78
Add <i>Projections</i>	82
Add <i>Faster Drawing a line segment from its two endpoints using Symmetry</i>	84
Add <i>Joining the ends of two wide line segments together</i>	85
Add <i>Composing monochrome bitmaps with separate alpha channel data</i>	86
Add <i>Orthogonal connection of two points</i>	87
Add <i>Faster line clipping</i>	92

Add <i>Tilings</i>	93
Add <i>Space-filling Curves</i>	94
Add <i>Hilbert curve</i> explanation	95
Add <i>Peano curve</i>	97