A Bitmapper's Companion

epilys

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

.

Map 3

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.

Contents

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 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 $(0, 0)$.

(0,0)
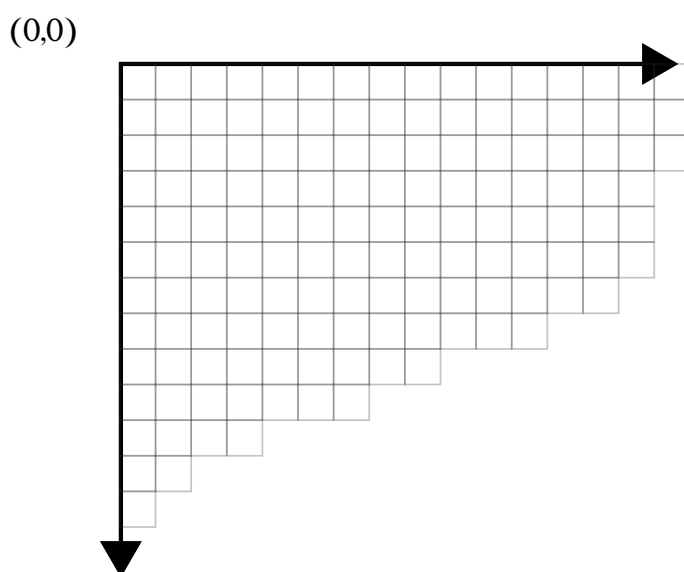
We will represent points as pairs of signed integers. When actually drawing them though, negative values and values outside the window's geometry will be ignored (clipped).

src/lib.rs:

This code file is a PDF attachment

```rust
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

```rust
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:

13
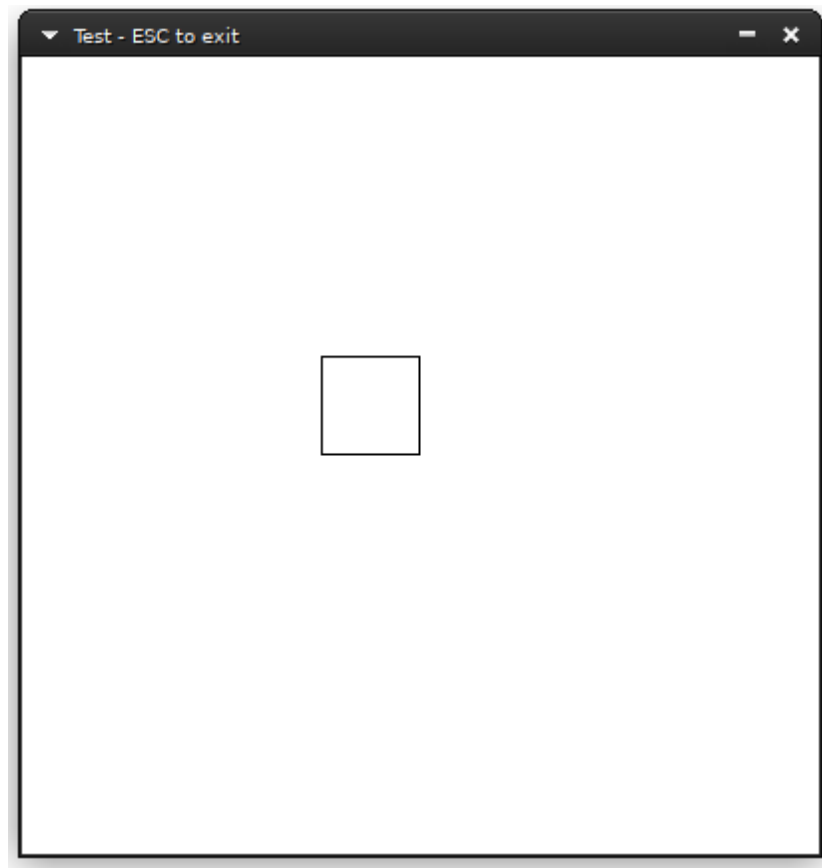
Chapter 3

Bits to byte pixels

Let's define a way to convert bit information to a byte vector:

```rust
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
}
```

15

intro

Chapter 4

Real pixels to byte pixels

Chapter 5

Loading `xbm` files in **Rust**

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

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

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

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

to

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

And replace the closing `}` with `]`.

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

19

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

load the image:

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

and finally run it:
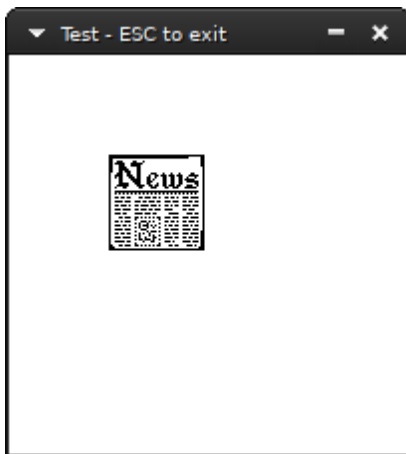
Part II

Points And Lines

lines

Chapter 6

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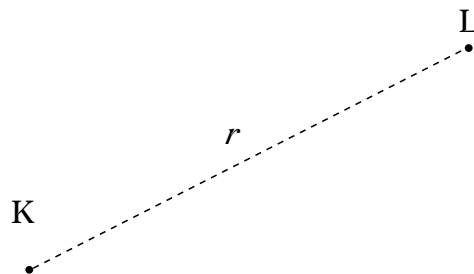
Given two points, K and L, an elementary application of Pythagoras' Theorem gives the distance between them as

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

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)
}
```

lines

Chapter 7

Equations of a line

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...

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

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

7.2 Line through two points

N

L

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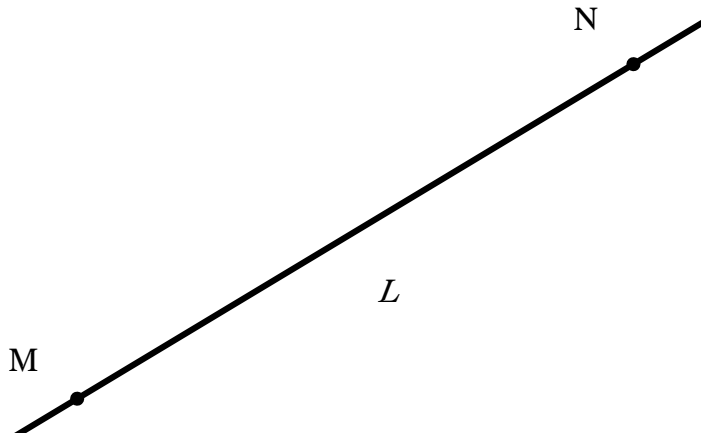
It seems sufficient, given the coordinates of two points M, N, to calculate a, b and c to form a line equation:

$$ax + by + c = 0$$

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 8

Distance from a point to a line
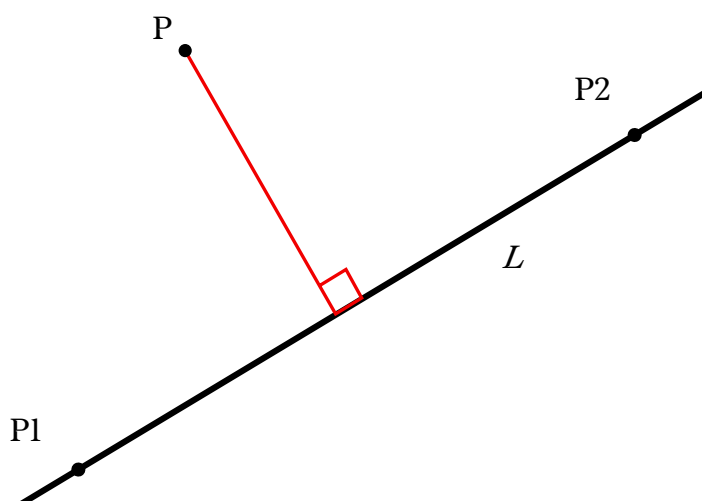
8.1   Using the implicit equation form

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 $ax + by = c$.

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

8.2 Using an L defined by two points P_1, P_2

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

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

8.3 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 9

Angle between two lines

2

Chapter 10

Intersection of two lines

lines

Chapter 11

Line equidistant from two points

add figure
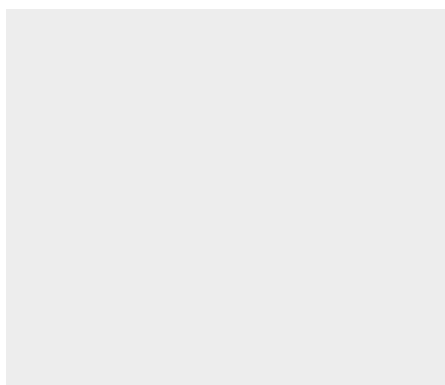
Figure 11.1:

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:

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

Putting them into the equation we can generate a triple of (a', b', c') and then normalize it to get L.

lines

Chapter 12

Normal to a line through a point

lines

Part III

Points And Line Segments

segments

Chapter 13

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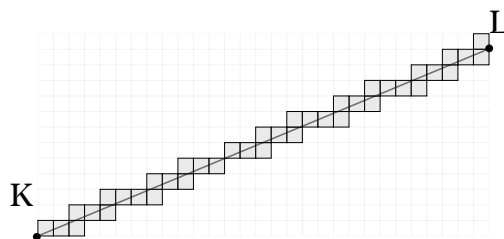
The algorithm presented here was first derived by Bresenham. In the *Image* implementation, it is used in the `plot_line_width` method.

```rust
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();
```

39

```
    sx = if dx > 0 { 1 } else { -1 };

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

    x = x1;
    y = y1;

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

    if ax > ay {
        d = ay - ax / 2;
        loop {
            self.plot(x, y);
            if x == x2 {
                return;
            }
            if d >= 0 {
                y = y + sy;
                d = d - ax;
            }
            x = x + sx;
            d = d + ay;
        }
    } else {
        d = ax - ay / 2;
        let delta = double_d / 3;
        loop {
            self.plot(x, y);
            if y == y2 {
                return;
            }
            if d >= 0 {
                x = x + sx;
                d = d - ay;
            }
            y = y + sy;
            d = d + ax;
        }
    }
}
```

add some explanation behind the algorithm

Chapter 14

Drawing line segments with width

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

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

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

    x = x1;
    y = y1;

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

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

```
            if x == x2 {
                return;
            }
            if d >= 0 {
                y = y + sy;
                d = d - ax;
            }
            x = x + sx;
            d = d + ay;
        }
    } else {
        d = ax - ay / 2;
        let delta = double_d / 3;
        loop {
            self.plot(x, y);
            {
                let total = |_x| _x - (y * dx) / dy + (y1 * dx) / dy - x1;
                let mut _x = x;
                loop {
                    let t = total(_x);
                    if t < -1 * delta || t > delta {
                        break;
                    }
                    _x += 1;
                    self.plot(_x, y);
                }
                let mut _x = x;
                loop {
                    let t = total(_x);
                    if t < -1 * delta || t > delta {
                        break;
                    }
                    _x -= 1;
                    self.plot(_x, y);
                }
            }
            if y == y2 {
                return;
            }
            if d >= 0 {
                x = x + sx;
                d = d - ay;
            }
            y = y + sy;
            d = d + ax;
        }
    }
}
```

Chapter 15

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 $r_3 = L_{12}(3)$ and $r_4 = L_{12}(4)$ respectively.

If $r_3 \neq 0$, $r_4 \neq 0$ and $sgn(r_3) == sign(r_4)$ 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 $sgn(r_1) == sign(r_2)$ 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

15.1 *Fast* intersection of two line segments

7

43

segments

circles

Part IV

Points, Lines and Circles

circles

circles

Chapter 16

Equations of a circle

circles

Chapter 17

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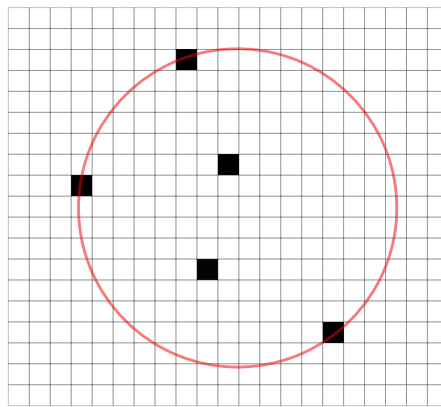
A bounding circle is a circle that includes all the points in a given set. Usually we're interested in one of the smallest ones possible.


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:
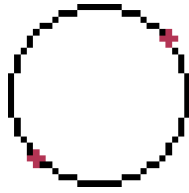

```
let p1 = points[0];
let p2 = points[1];
//The  circle  is  determined  by  two  points,  P  and  Q.  The  center  of the  circle
↪  is
//at  (P  +  Q)/2.0  and  the  radius  is  |(P  -  Q)/2.0|
let d_2 = (
(((p1.0 + p2.0) / 2), (p1.1 + p2.1) / 2),
(distance_between_two_points(p1, p2) / 2.0),
);
```

And a way to make a circle out of three points:
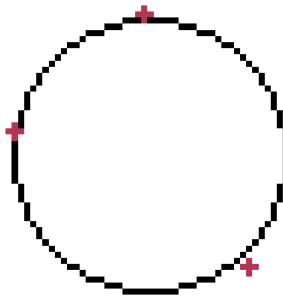
```
fn min_circle_w_3_points(q1: Point, q2: Point, q3: Point) -> Circle {
    let (ax, ay) = (q1.0 as f64, q1.1 as f64);
    let (bx, by) = (q2.0 as f64, q2.1 as f64);
    let (cx, cy) = (q3.0 as f64, q3.1 as f64);

    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)
```

```rust
        } 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.;
    }
```

```rust
    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

curves

Part V

Curves other than circles

curves

Chapter 18

Parametric elliptical arcs

curves

Part VI

Points, Lines and Shapes

shapes

shapes

Chapter 19

Union, intersection and difference of polygons

shapes

shapes

Chapter 20

Centroid of polygon

shapes

Chapter 21

Flood filling

shapes

shapes

shapes

15

shapes

shapes

Part VII

Vectors, matrices and transformations

trans-
forma-
tions

Chapter 22

Rotation of a bitmap

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

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

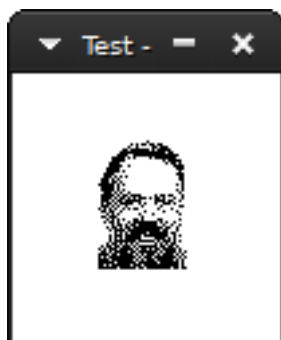$$c = cos\theta, s = sin\theta, x_{p'} = x_p c - y_p s, y_{p'} = x_p s + y_p c.$$

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);
```


trans-
forma-
tions

This is the `xface` of `dmr`. Instead of displaying the `bitmap`, this time we will

77

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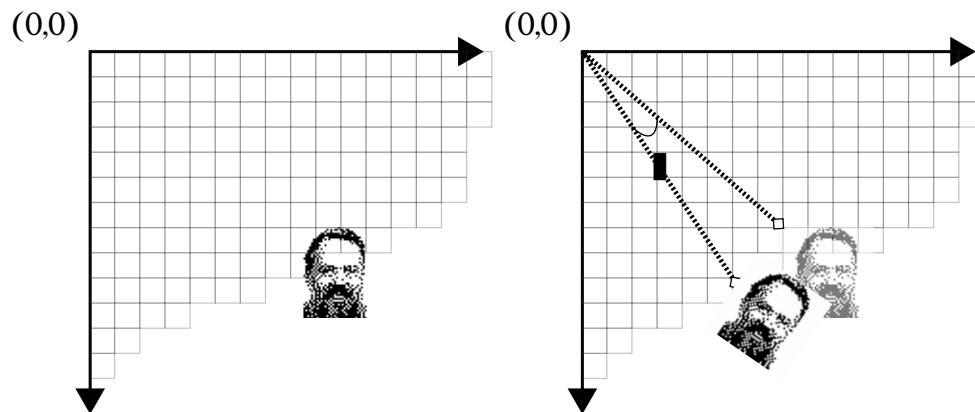
(the distance to the origin (actually 0 pixels) has been exaggerated for the sake of the example)

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:

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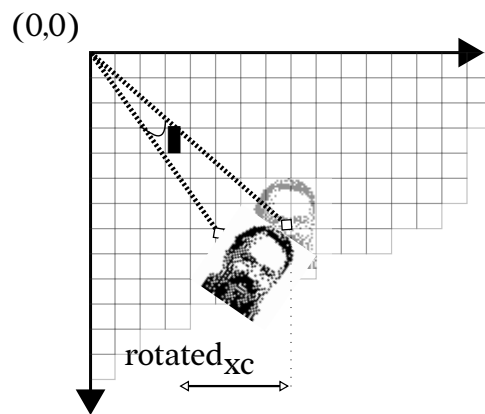
$$y_c = \frac{1}{n} \sum_{i=0}^{n} y_i$$

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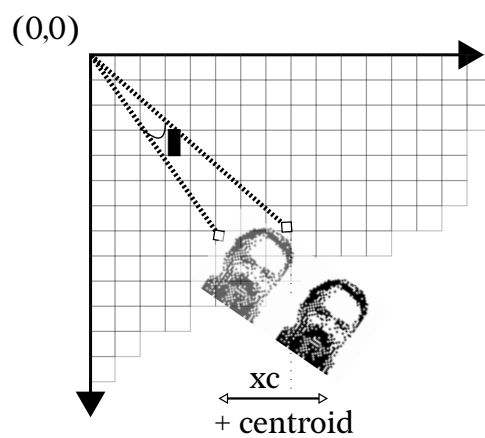
Here's it visually: First subtract the center point.

trans-
forma-
tions

(0,0)

Then, rotate.

(0,0)

rotated$_{xc}$
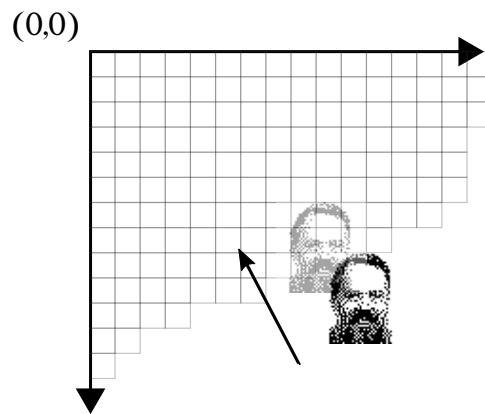
And subtract back to the original position.

(0,0)

xc

+ centroid

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);
        }
    }
}
```
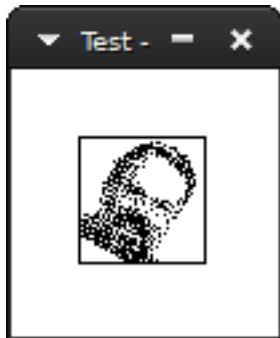

The result:

22.1   Fast 2D Rotation

16

trans-
forma-
tions

transformations

Chapter 23

90° Rotation of a bitmap by parallel recursive subdivision

transforma-
tions

83

Chapter 24

Magnification/Scaling

src/bin/scale.rs:

This code file is a PDF attachment
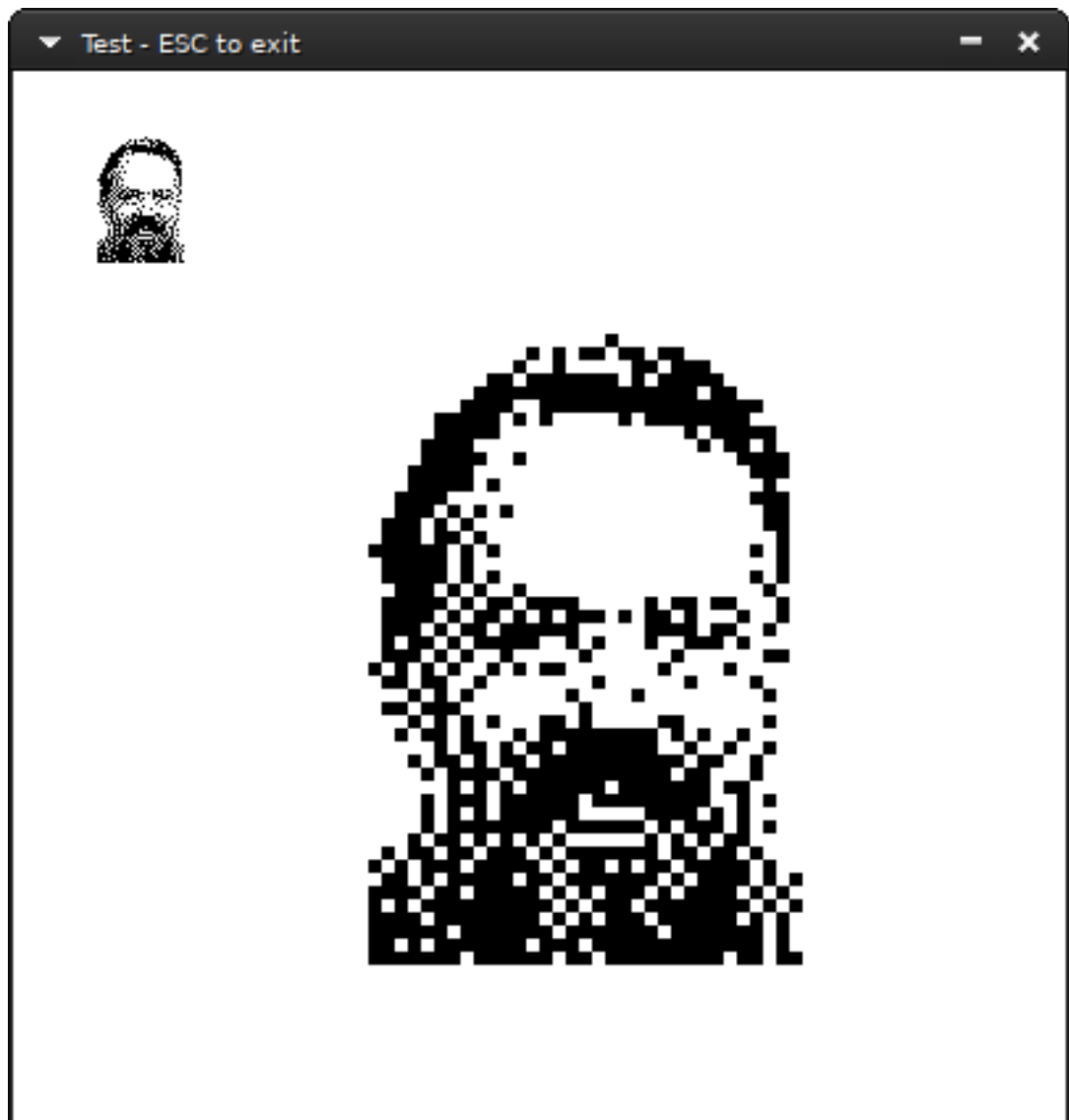
```rust
let mut original = Image::new(DMR_WIDTH, DMR_HEIGHT, 25, 25);
original.bytes = bits_to_bytes(DMR_BITS, DMR_WIDTH);
original.draw(&mut buffer, BLACK, None, WINDOW_WIDTH);

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

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

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

24.1 Smoothing enlarged bitmaps

18

trans-
forma-
tions

24.2   Stretching lines of bitmaps

19

trans-
forma-
tions

Chapter 25

Mirroring

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

To mirror a pixelacross 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:
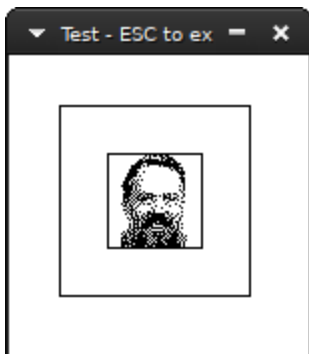
$$M_y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

trans-
forma-
tions

Chapter 26

Shearing

src/bin/shearing.rs:

This code file is a PDF attachment

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.
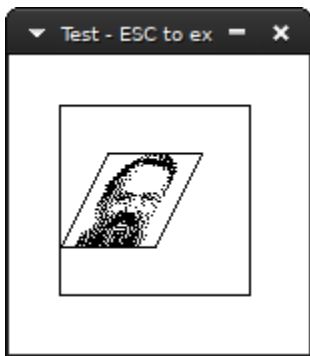

With l being equal to the desired tilt away from the y axis, the transformation is described by the following matrix:

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

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)
}
```
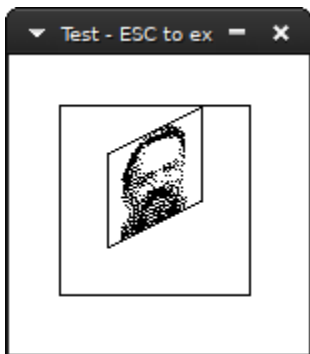
trans-forma-tions

93

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();
```

trans-
forma-
tions

**trans-
forma-
tions**

Chapter 27

Projections

**trans-
forma-
tions**

Part VIII

Addendum

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

addendum

Chapter 28

Joining the ends of two wide line segments together

Chapter 29

Composing monochrome bitmaps with separate alpha channel data

Chapter 30

Orthogonal connection of two points

adden-
dum

Chapter 31

Join segments with round corners

Chapter 32

Faster line clipping

Chapter 33

Space–filling Curves

adden-
dum

33.1 Hilbert curves

addendum

33.2 Peano curves

33.3   Z-order curves

Index

About this text

The text has been typeset in X͟ͅƎLATEX 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