A Symphony of Fractals: Window to the Infinite

PH435 - Microprocessors Course Project

Devansh Satra, Archit Swamy

November 10, 2023

1 Fractals: Introduction

Fractals are complex geometric shapes or patterns that are infinitely self-similar. This means that as you zoom in on a fractal, you will continue to see the same structure repeat itself endlessly, i.e., the fractal is constructed of itself infinitely.

In sections 2 and 3, we delve into some of the math behind fractals, namely the Mandelbrot and Julia sets, and we run through the code written on the Arduino UNO which displays the above on a 3.5" LCD screen with a resolution of 480x320 pixels.

Section 4 details the part of the project where one chooses a particular julia set from the mandelbrot set using a resistive touch pad.

2 Holomorphic Dynamics

2.1 The Mandelbrot Set

The Mandelbrot Set is a two-dimensional set which is defined by the following mathematical definition:

$$M = \{c \in \mathcal{C} : |f_c(0)|,\ |f_c(f_c(0))|,\ |f_c(f_c(f_c(0)))|,\ \ldots\} \text{ is bounded} \tag{1}$$

$$f_c(z) = z^2 + c \tag{2}$$

The region interior to the boundary is the set of all points for which the sequence $f_c(z)$ remains bounded. What we observe is that the boundary of this set is a fractal curve. An interesting property of this boundary is that a point c belongs to the mandelbrot set if and only if $|z_n| < 2 \ \forall \ n \geq 0$, where z_n is given by $z_{n+1} = f_c(z)$. We note that we obtain the mandelbrot set by varying $c \in \mathcal{C}$. Here's the arduino implementation:

```
void generateMandelbrot() {
  for (int y = 0; y < TFT_HEIGHT; y++) {
    for (int x = 0; x < TFT_WIDTH; x++) {
      double zx = 0;
      double zy = 0;
      double cx = (x - TFT_WIDTH / 2.0) / zoom + offsetX;
      double cy = (y - TFT_HEIGHT / 2.0) / zoom + offsetY;

      int iteration = 0;
      while (zx * zx + zy * zy < 4 && iteration < maxIterations) {
        double tmp = zx * zx - zy * zy + cx;
        zy = 2 * zx * zy + cy;
        zx = tmp;
        iteration++;
      }

      int color[3] = {cmap_data[iteration]};
      tft.drawPixel(x, y, tft.color565(color[0], color[1], color[2]));
    }
  }
  flag = true;
}
```

(z_x, z_y) and (c_x, c_y) represent complex numbers z, c. The while loop simply performs the iterative sequence defined by $f_c(\cdot)$.

$$z^2 = (z_x^2 - z_y^2) + i(2z_xz_y) \tag{3}$$

As we are performing the computation on an Arduino UNO, computation has been kept minimal due to the lack of resources on this piece of hardware. Thus in constructing the fractal boundary for the mandelbrot set, for all c, we only iterate the sequence $maxIterations = 12$ times.

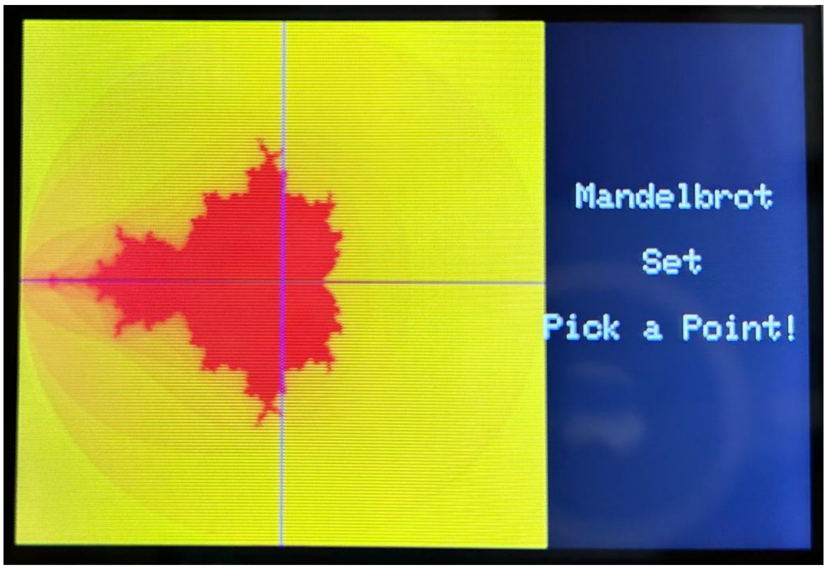

Fig. 2: Mandelbrot set generated with maxIterations = 12

As noted before, if $z_n > 2$ for any $n < 12$, that corresponding c lies outside the mandelbrot set. Thus, we expect that for a higher value of *maxIterations*, the resulting set will be displayed more accurately, with greater resolution. Thus, we would expect the contours to be closer together. Accordingly, we apply a colourmap to the mandelbrot set so that we can distinctly observe the points that are inside and outside the set. Points which take a lesser number of iterations to diverge are coloured blacker, and points which converge are coloured redder.

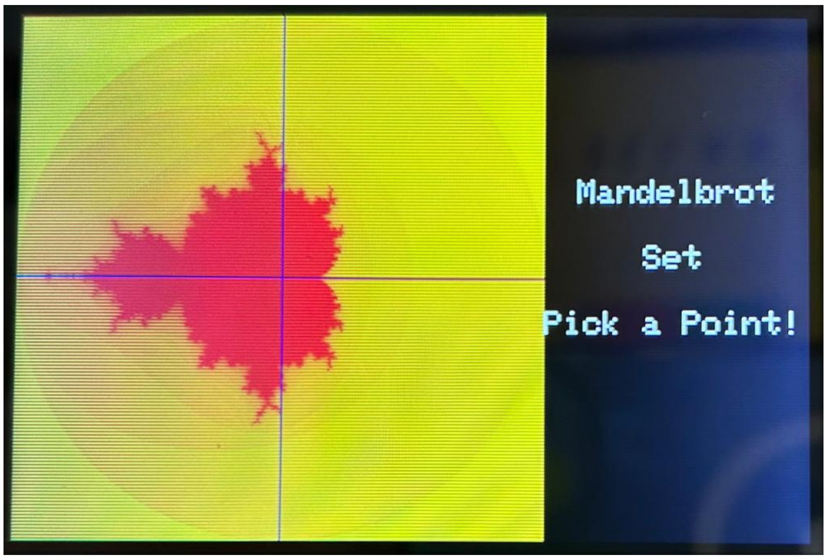

Fig. 3: Mandelbrot set generated with maxIterations = 24

In the above image, using $maxIterations = 24$, we see that we are able to achieve further resolution.

2.2 Julia Sets

Julia Sets are another class of fractals closely related to the Mandelbrot Set, but they are defined by varying the constant c while keeping the function $f_c(z)$ fixed.

Julia sets are defined in the following manner.

For the same $f_c(z)$ defined above, fix some $R > 0$ large enough that $R^2 - R \geq |c|$. Then the filled Julia set for this system is the subset of the complex plane given by

$$K(f_c) = \{z \in \mathcal{C} : \forall n \in \mathcal{N}, |f_c^n(z)| \leq R\},$$

where $f_c^n(z)$ is the n^{th} iterate of $f_c(z)$. The Julia set $J(f_c)$ of this function is the boundary of $K(f_c)$. Thus for $R = 2$, the Julia set $J(f_c)$ is the boundary of the mandelbrot set.

Below is the arduino implementation for generating a julia set for a given z-coordinate in the complex plane:

```
void drawJuliaSet(float x1, float y1) {
  for (int x = 0; x < TFT_WIDTH; x=x+2) {
    for (int y = 0; y < TFT_HEIGHT; y=y+2) {
      float zx = x * (xmax - xmin) / (TFT_WIDTH - 1) + xmin;
      float zy = y * (ymax - ymin) / (TFT_HEIGHT - 1) + ymin;
      float cx = x1;
      float cy = y1;

      int i;
      for (i = 0; i < max_iterations; i++) {
        float x2 = zx * zx;
        float y2 = zy * zy;

        if (x2 + y2 > 4.0) break;

        float xtemp = x2 - y2 + cx;
        zy = 2.0 * zx * zy + cy;
        zx = xtemp;
      }

      int color[3] = {cmap_data[i]};
      tft.drawPixel(x, y, tft.color565(color[0], color[1], color[2]));
    }
  }
}
```

The lowest level (innermost) for loop performs the same function here as the while loop in the generateMandelbrot() function, of performing the iteration $f_c(\cdot)$.

Shown below are julia sets generated by the arduino for varying values of c.

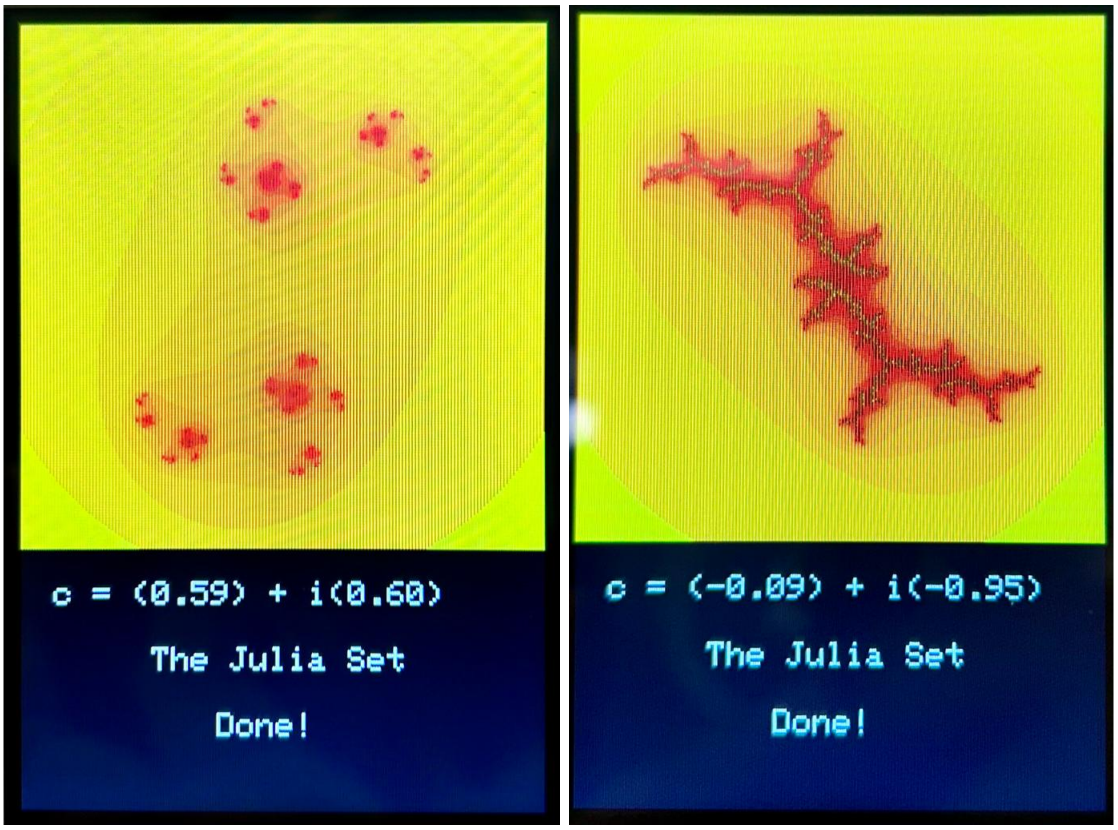

Fig. 4: Julia Sets

3 Lorenz System

The Lorenz attractor is a chaotic system that exhibits sensitive dependence on initial conditions. It was discovered by Edward Lorenz in 1963 as a simplified model of atmospheric convection.
The Lorenz equations are a system of three coupled nonlinear differential equations:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Where σ, ρ, and β are system parameters. For certain values of these parameters, the Lorenz system exhibits chaotic behavior. Small differences in initial conditions lead to vastly different outcomes over time. To simulate the Lorenz attractor, we numerically integrate the equations over discrete time steps. The Arduino code implements a simple Euler integration with time step dt:

$$x_{n+1} = x_n + dt \cdot \sigma(y_n - x_n)$$
$$y_{n+1} = y_n + dt \cdot (x_n(\rho - z_n) - y_n)$$
$$z_{n+1} = z_n + dt \cdot (x_n y_n - \beta z_n)$$

Where x_n, y_n, z_n are the variable values at the current time step.

```
#include <MCUFRIEND_kbv.h>

MCUFRIEND_kbv tft;

const int screenWidth = 480; // Width of your TFT screen
const int screenHeight = 320; // Height of your TFT screen

#define BLACK 0x0000
#define WHITE 0xFFFF

const int numPoints = 10000; // Number of points to plot
const float dt = 0.001; // Time step
const float sigma = 20.0; // Lorenz system parameter (scaled down)
const float rho = 50.0; // Lorenz system parameter
const float beta = 8.0 / 3.0; // Lorenz system parameter

float x1[3] = {0.1, 0.1, 0.1}; // Initial conditions for three points
float y1[3] = {0.1, 0.1, 0.3};
float z1[3] = {0.1, 0.1, 0.2};

int px[3];
int pz[3];

int prev_px[3] = {0, 0, 0};
int prev_pz[3] = {0, 0, 0};

int last_x[3][100];
int last_z[3][100];
int last_index[3] = {0, 0, 0};
bool flag[3] = {false, false, false};

void setup() {
  tft.reset();
  uint16_t identifier = tft.readID();
  tft.begin(identifier);
  tft.setRotation(1); // Set screen rotation if needed
  tft.fillScreen(TFT_BLACK);
}

void loop() {
  for (int i = 0; i < 3; ++i) {
    float dx = sigma * (y1[i] - x1[i]);
    float dy = x1[i] * (rho - z1[i]) - y1[i];
    float dz = x1[i] * y1[i] - beta * z1[i];

    x1[i] += dx * dt;
    y1[i] += dy * dt;
    z1[i] += dz * dt;

    px[i] = map(z1[i], 0, 100, 0, screenWidth);
    pz[i] = map(x1[i], 30, -30, 0, screenHeight);

    // Map z1 value to a color gradient
    int colorR = map(y1[i], -50, 50, 0, 51000);
    int colorG = map(y1[i], -50, 50, 51000, 0);
    int colorB = map(y1[i], -50, 50, 10000, 30000);
    uint16_t pixelColor = tft.color565(colorR, colorG, colorB);

    tft.drawLine(prev_px[i], prev_pz[i], px[i], pz[i], pixelColor);

    prev_px[i] = px[i];
    prev_pz[i] = pz[i];

    // Erase the oldest line drawn after i == 99
    if (flag[i]) {
      tft.drawLine(last_x[i][last_index[i]], last_z[i][last_index[i]], prev_px[i], prev_pz[i], BLACK);
    }

    // Store the last 100 points
    last_x[i][last_index[i]] = px[i];
    last_z[i][last_index[i]] = pz[i];
    last_index[i] = (last_index[i] + 1) % 100;
    if (last_index[i] == 99) {
      flag[i] = true;
    }
  }
}
```

The code simulates three different initial condition sets and integrates them forward in time. The x and z values are mapped to screen coordinates to plot the attractor. Color gradients are also added based on the y value. Old points are erased using background color to create a trailing effect.

This allows visualizing the characteristic butterfly shape of the Lorenz attractor and demonstrates sensitivity to initial conditions as the three traces visibly diverge. The attractor shape and chaotic behavior emerge from the simple nonlinear equations.

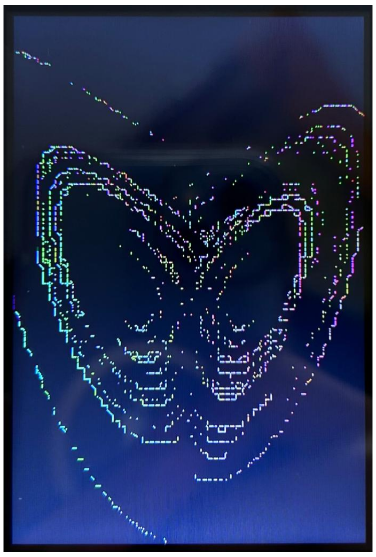
Fig. 5: Lorenz Attractor

4 Chaos Game and Sierpinski Triangle

The chaos game is a simple algorithm that can generate fractal patterns like the Sierpinski triangle. It involves randomly picking points inside a shape and plotting points between the selected point and a randomly selected vertex at each iteration at a predefined ratio.

The Sierpinski triangle is constructed by repeatedly dividing triangles into smaller congruent triangles. On a binary level, it exhibits self-similarity - smaller parts resemble the whole.

Mathematically, we define three vertex points of an equilateral triangle:

$$
\begin{aligned}
p_1 &= (x_1, y_1)\\
p_2 &= (x_2, y_2)\\
p_3 &= (x_3, y_3)
\end{aligned}
$$

We pick a random starting point inside the triangle and iterate:

$$p_{\text{new}} = \frac{1}{2}(p_{\text{old}} + p_{\text{random}})$$

Where p_{random} is randomly chosen from p_1, p_2, p_3.

This process is repeated, plotting each new point. The Sierpinski triangle emerges as the fractal pattern after sufficient points.

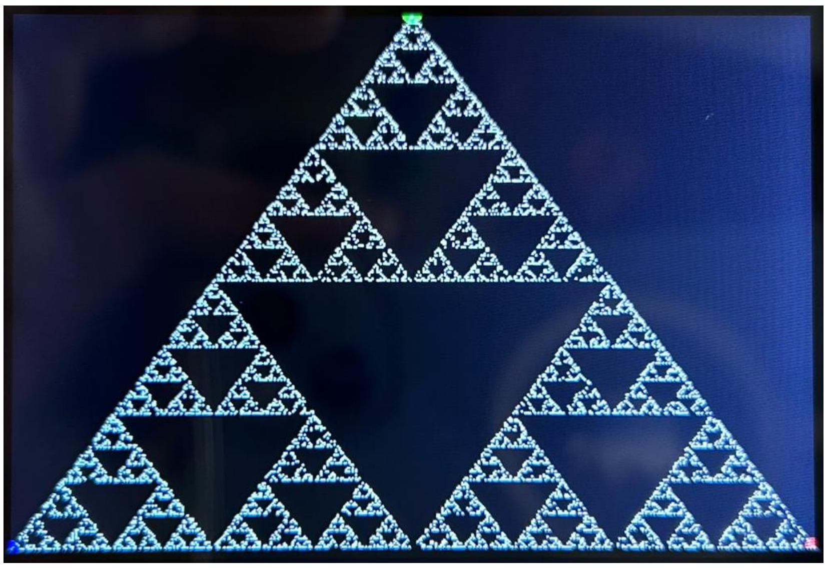

Fig. 6: The Sierpinski Triangle

The Arduino code implements this algorithm, using 3 predefined vertex points. It plots each new midpoint in white, with a small delay between points. This animates the fractal taking shape.

```
#include <Adafruit_GFX.h>
#include <MCUFRIEND_kbv.h>

MCUFRIEND_kbv tft;

// Define colors
#define BLACK 0x0000
#define RED   0xF800
#define GREEN 0x07E0
#define BLUE  0x001F
#define WHITE 0xFFFF

// Define the vertices of the triangle
int16_t vertexX[3] = {0, 240, 480};
int16_t vertexY[3] = {0, 320, 0};

// Initialize the current point as a random point inside the triangle
int16_t currentX = random(160, 241);
int16_t currentY = random(20, 221);

void setup() {
  tft.begin(0x9481); // Initialize the LCD
  tft.setRotation(1); // Set the display rotation (0 for portrait)
  tft.fillScreen(BLACK); // Fill the screen with black

  // Draw the vertices of the triangle
  tft.fillCircle(vertexX[0], vertexY[0], 5, RED);
  tft.fillCircle(vertexX[1], vertexY[1], 5, GREEN);
  tft.fillCircle(vertexX[2], vertexY[2], 5, BLUE);
}

```

```
void loop() {
  // Choose a random vertex
  int16_t randomVertex = random(3);

  // Calculate the midpoint between the current point and the chosen vertex
  currentX = (currentX + vertexX[randomVertex]) / 2;
  currentY = (currentY + vertexY[randomVertex]) / 2;

  // Draw the new point
  tft.drawPixel(currentX, currentY, WHITE);

  // You can adjust the delay to control the animation speed
  delay(10); // Delay between points (adjust as needed)
}
```

The delay controls the animation speed. More points result in a more detailed Sierpinski triangle structure emerging on the display.

5 Hardware Description

5.1 Making it Interactive

Owing to the fact that we wrote code for a Lorenz Attractor, the Chaos Game, and the Mandelbrot and Julia sets, we wanted to display all of them. For this purpose, we made a menu with options for all of these simulations. This was done using the LCD's touchscreen capability. Below we have displayed the Main Menu, Windows to the Infinite:

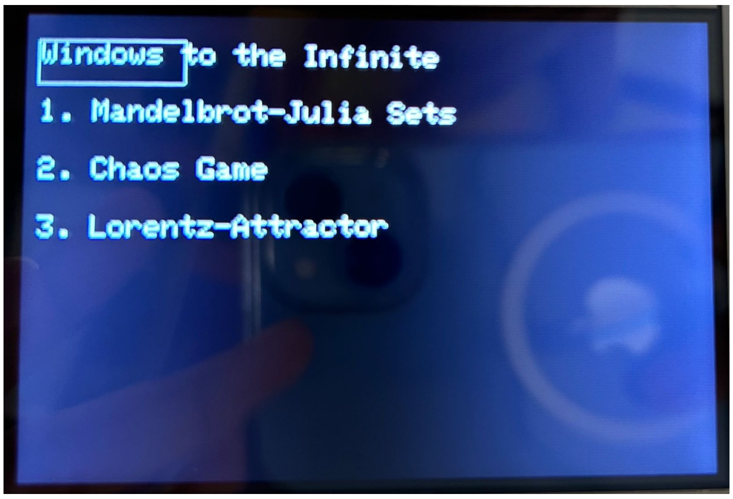

Fig. 7: The Main Menu

For each simulation, having run through a number of iterations, the simulation stops and waits until it registers another click, after which it returns to the main menu so that another simulation may be selected. Here's the code for its implementation:

```
void loop() {
  if(chose == true) {
    showMenu();
  }

  while(x_ == x_old && y_ == y_old) {
    Point p = ts.getPoint();
    if (p.z > ts.pressureThreshhold && p.z < 1500) {
      x_ = p.x;
      y_ = p.y;
    }
  }

  x_old = x_;
  y_old = y_;

  choice = getChoice(y_);
  if(choice == 1) {
    mandelbrot();
    chose = true;
  }
  else if(choice == 2) {
    chaos();
    chose = true;
  }
  else if(choice == 3) {
    lorenz();
    chose = true;
  }
  else {
    chose = false;
  }
}
```

The above code puts the main menu inside arduino's standard loop() function and waits for appropriate input. The getChoice() function gets the title which was pressed on the screen using the (x, y) coordinates, and the mandelbrot(), chaos() and lorenz() functions are functionally exactly the same as the code mentioned in their respective sections.

The showMenu() code has been displayed below.

```
void showMenu() {
  tft.reset();
  uint16_t ID = tft.readID();
  tft.begin(ID);
  tft.setRotation(3);
  tft.setTextSize(2);
  tft.fillScreen(TFT_BLACK);

  tft.setCursor(10, 20);
  tft.drawFastHLine(8, 18, 100, 0xFFFF);
  tft.drawFastVLine(8, 18, 30, 0xFFFF);
  tft.drawFastHLine(8, 48, 100, 0xFFFF);
  tft.drawFastVLine(108, 18, 30, 0xFFFF);
  tft.print("Windows to the Infinite");

  tft.setCursor(10, 60);
  tft.print("1. Mandelbrot-Julia Sets");

  tft.setCursor(10, 100);
  tft.print("2. Chaos Game");

  tft.setCursor(10, 140);
  tft.print("3. Lorentz-Attractor");
}
```

5.2 Working Principle of the TFT LCD Display

In this project, we use the touchpad feature of the LCD to allow a user to interact with the arduino and choose a value of c for which to generate the Julia set. In fact, the julia sets shown above were generated in the same manner.

The LCD has a resistive touchpad feature i.e, using what is essentially a potentiometer, one can get the (x, y) information of where the screen was touched, given that it has been deformed sufficiently. Here's how it works:

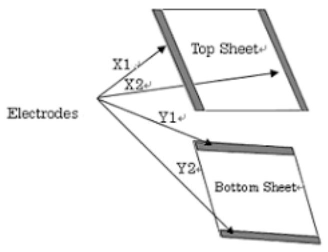

Fig. 6: Working of a Resistive Touchscreen

In the functioning of a four-wire touchscreen, a consistent and one-way voltage gradient is applied to the initial sheet. When both sheets are pressed in contact, the second sheet calculates the voltage as distance relative to the first sheet, thereby determining the X coordinate. Once this contact point is determined, the voltage gradient is then applied to the second sheet to determine the Y coordinate. This distance calculation is performed by measuring the voltage along the electrodes in the transverse direction i.e, the x-coordinate is interpreted from a voltage reading on the electrodes on the bottom sheet (depicted). Sicne the film is resistive, by taking a voltage reading, one can understand at what distance from the negative or positive electrode contact was made, thus acting like a potentiometer. These processes happen very swiftly, ensuring the precise recording of the touch location as soon as contact is established.

The full code can be found here.