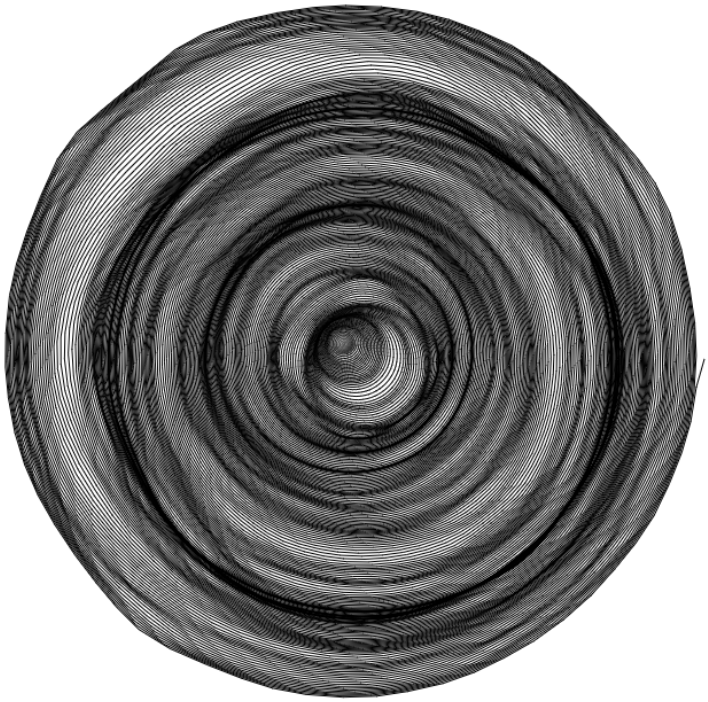
How I Made It

Andrew Bryant

2020-10-18T13:23:19Z



Introduction

In this ebook I explain how I made some of my best sketches.

The primary audience for this ebook is me.

I'm scratching my own itch, putting something out into the world that I wish existed.

I see sketches and wonder how they were made. How did they do that? Why did they do that? I study the code of others, but often feel stuck. Not knowing why things worked as they did.

My goal is to show how my code works. So someone can read this and get how the sketches I make are constructed. Hopefully then the reader can build upon them and make them their own faster.

I assume you have basic knowledge of coding and p5.js. Because of this assumption, I gloss over some elementary-level details (e.g., how to draw a line). I focus on the big picture; how I combined those basic elements to create a sketch that looks good (at least to me!).

I try not to repeat myself to make the book more concise. I explain a concept the first time it comes up, and leave it unexplained in later occurrences.

You can find the code on my repo. The title of each chapter will correspond to the name of the folder with the code. Because of the randomness in the sketch code, the image you get when running it may be different from its image in the book.

Miscellaneous

- Link to the repository of the book's code: https://github.com/apbryant/creativecoding/tree/master/ebook
- The createCols function used throughout the book is from https://www.openprocessing.org/sketch/943564/ and available under the Creative Commons Attribution-ShareAlike 3.0 Unported license.
- Book released under the Creative Commons Attribution-ShareAlike 3.0 Unported license.
- Link to the Creative Commons Attribution-ShareAlike 3.0 Unported license: https://creativecommons.org/licenses/by-sa/3.0/legalcode

200901b

What the code does

Draw short lines through the middle of the sketch. Create a smooth curve by varying the place the lines are drawn. The length and direction of the line varies a bit. noise() is used to create the variation. The wavy effect comes from

the rotation of the canvas. The canvas is rotated according to the sine of x in radians.

About noise()

noise() is a Perlin noise generator. Perlin noise creates pseudorandom numbers that vary naturally. They are used to create graphics that look real.

Picking colors

coolors.co is a site where you can create and share color palettes. You can find the colors in each palette's url. The colors are in hex format at the end of the url. createCols() is a widely-used function. The function returns the colors from the coolor.co url as an array. I shuffle the array to randomize the color order. Colors, when done well, make a sketch come alive. They're hard to combine into a palette, though. You could spend a lifetime mastering this skill. Thanks to coolors, color selection has been outsourced. I can get ready-made palettes and pick the one I like.

Code

```
function setup() {
  let colors=shuffle(createCols("https://coolors.co/264653-2a9d8f-e9c46a-f4a261-e76f51"));
  createCanvas(w=600,w)
  background(colors.pop());
  noFill()
  strokeWeight(2)

  // Line segments are drawn in these for loops
  for(i=w*0.4;i<w*0.6;i++){
    t=0
    stroke(random(colors))
    oldx=0
    oldy=i

    for(x=0;x<w;x++){
      push()
      // Perlin noise used here.
      // Value returned by noise() mapped to larger range
      // This makes it easier to see variation.
      y=oldy+map(noise(t),0,1,-3,3)
      rotate(sin(radians(x)))
      line(oldx,oldy,x,y)
      // Store x and y in oldx and oldy to connect lines
      oldx=x
      oldy=y
```

```
      t+=0.01
      pop();
    }
  }
}

function createCols(_url)
{
  let slash_index = _url.lastIndexOf('/');
  let pallate_str = _url.slice(slash_index + 1);
  let arr = pallate_str.split('-');
  for (let i = 0; i < arr.length; i++) {
    arr[i] = '#' + arr[i];
  }
  return arr;
}
```

Image

200919l

What the code does

w*w points are being drawn. The cosine and sine of a is added to x and y respectively to create the point coordinates.

Why points? Points are one of the simplest things you can make in creative coding. It's just a dot, after all! But by drawing many dots, and putting them in just the right places, beautiful patterns can be created.

Understanding the pattern

noise(xˆy) always between zero and one the radian will be between zero and about 0.017. Multiplied by 45 the result will be between zero and about 0.79.

cos(a) will vary between 1 and about 0.53. sin(a) is between 0 and about 0.85.

The nature of Perlin noise means that you'll get recurring patterns. Different numbers on the "noise scale" will generate the same output, so cos(a) and sin(a) will be the same for numerous x and y values.

The constant multiplier (here 45) zooms in/out the pattern.

setup=_=>

setup=_=>{...} is equivalent to function setup() {...}

_ is a placeholder variable. => is shorthand for creating a function. What's happening is the variable setup is being assigned to a function.

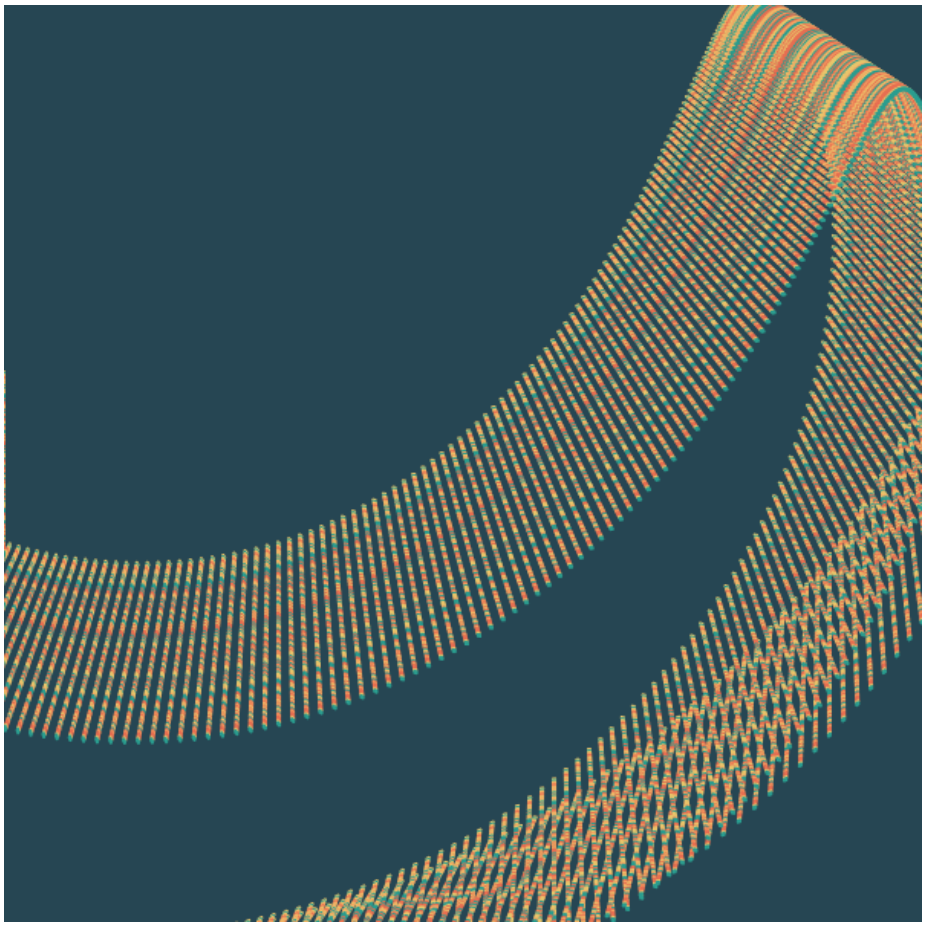


Figure 1: 200901b

Why use it? It's a bit shorter and I think it looks cooler.

Code

```
setup=_=>{
  createCanvas(w=600,w)
  background(0)
  stroke(255)

  for(x=0;x<w;x++){
    for(y=0;y<w;y++){
      a=radians(noise(x^y))*45
      point(x+cos(a),y+sin(a))
    }
  }
}
```

Image

200923c

What the code does

A spiral made of line segments is being drawn. After drawing each line segment the radius r is decremented and the angle at which the line is drawn is incremented. The combination decreasing the radius and incrementing the angle creates the spiral. I made this because I thought it would be interesting to make a complex circular shape that's made of one connected line.

I use beginShape(), two vertices, and endShape() to draw a line between the end of the previous segment and (x,y).

Just using line(oldx,oldy,x,y) would work as well.

How Perlin noise is used

This sketch is a great example of Perlin noise. The x and y coordinates I draw from vary a bit from being a perfect circle according to Perlin noise.

This gives the spiral the natural looking waves, like ripples in a pond.

I pass three variables into noise to make sure the noise value that's returned is unique for each line segment. Because the value returned is between zero and one, I use map() to map the value to a larger range. This is so that the variation due to noise can be seen more easily.

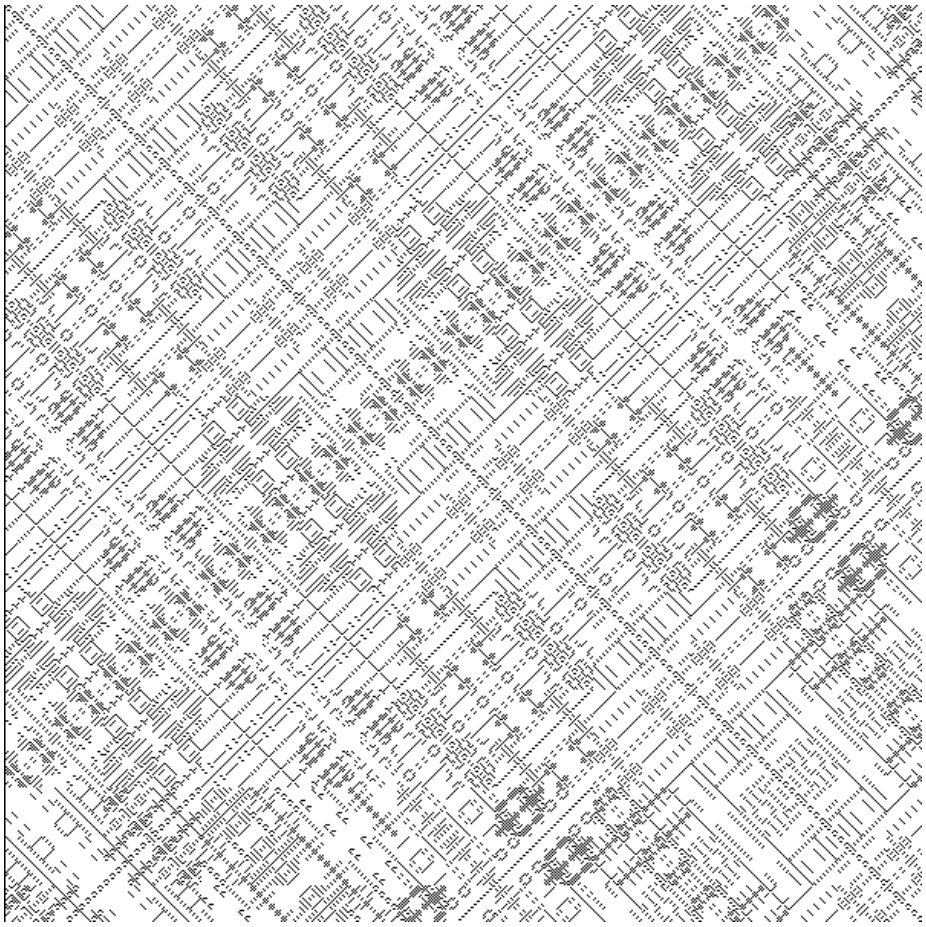


Figure 2: 200919l

Code

```
setup=_=>{
    createCanvas(w=800,w)
    background(255)
    stroke(0)
    noFill()

    a=0
    r=301
    ainc=0.1
    rdec=0.02
    oldx=w/2+cos(a)*r
    oldy=w/2+sin(a)*r
    a+=ainc
    r-=rdec

    while(r>0){
        beginShape()
        scl=0.004
        m=30
        nx=map(noise(r*scl,a*scl,oldx*scl),0,1,-m,m)
        ny=map(noise(r*scl,a*scl,oldy*scl),0,1,-m,m)
        x=(w/2+nx)+cos(a)*r
        y=(w/2+ny)+sin(a)*r
        vertex(oldx,oldy)
        vertex(x,y)
        endShape()

        a+=ainc
        r-=rdec
        oldx=x
        oldy=y
    }
}
```

Image

201004a

What the code does

This sketch is of hexagonal random walks. A random walk is a path that consists of random steps over a space (in this case the canvas).

I added a twist by making it so the steps would create the outlines of hexagons.

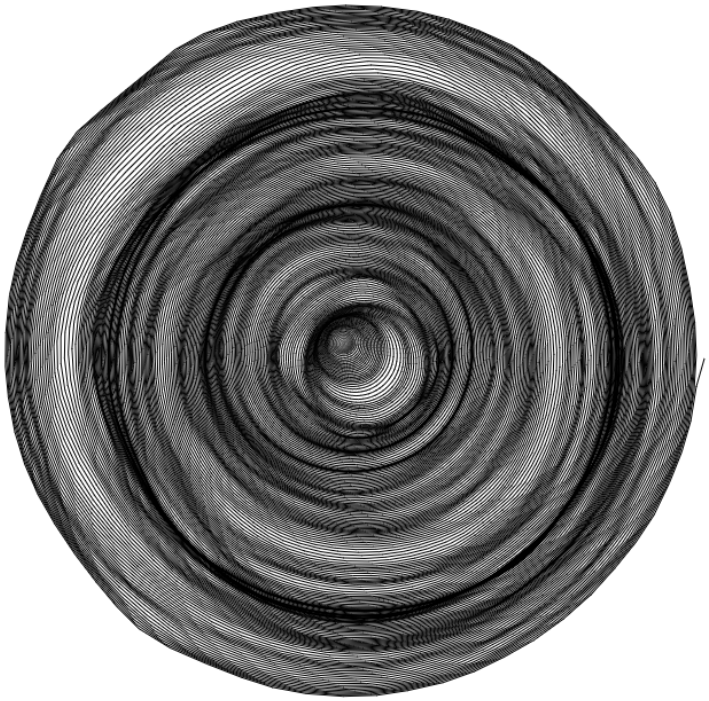


Figure 3: 200923c

I draw ten random walks using the j for loop.

This is done by building the shape from smaller line segments, as was described in other sketches.

To get the "random" part of the random walk, an integer between 0 and 10 is selected. I calculate the modulo (i.e., the remainder) of two for the randomly picked integer. It will either be zero or one. There's a 50% chance the modulo is zero and a 50% chance it's one.

According to result I decide whether to rotate the canvas clockwise or counter-clockwise.

A few words about hexagons

Hexagons appear all over nature. Some examples of where they come up:

- Honeycombs
- Snowflakes
- Eyes of insects
- Turtle shells

The reason is that to divide up an area into an equal size, a hexagon leaves no wasted space and minimizes the material needed to create the divisions.

Code

```
draw=_=>{
  createCanvas(w=800,w)
  // The sketch will re-draw itself once every second.
    frameRate(1)
    colors=shuffle(createCols("https://coolors.co/227c9d-17c3b2-ffcb77-fef9ef-fe6d73"))
    background(colors.pop())
    stroke(255)
    strokeWeight(2)

    for(j=0;j<10;j++){
        oldx=w/2
        oldy=w/2
        stroke(colors[int(random(colors.length))])

        for(i=0;i<500;i++){
            translate(oldx,oldy)

            if(int(random(10))%2){
        // TAU/6 = 60 degrees
                rotate(TAU/6)
            } else {
```

```
                rotate(-TAU/6)
            }

        // The step size is the same.
        // Twenty pixels along the x axis and twenty pixels along the y axis
            x=20
            y=20
            line(0,0,x,y)
            oldx=x
            oldy=y
        }
    }
}

function createCols(_url)
{
  let slash_index = _url.lastIndexOf('/');
  let pallate_str = _url.slice(slash_index + 1);
  let arr = pallate_str.split('-');
  for (let i = 0; i < arr.length; i++) {
    arr[i] = '#' + arr[i];
  }
  return arr;
}
```

Image

Sierpiński Triangle

What the code does

This sketch is of triangles inside of triangles. This is called a Sierpiński triangle. How it's done is I find the midpoint of each side of a triangle. Using the midpoints I divide the triangle into three smaller ones. For each of those new, smaller triangles, I repeat the process.

The Sierpiński triangle is named after Wacław Sierpiński, a Polish mathmetician. However, the pattern had already been discovered and used long before him.

Recursion

Recursion is when the solution of a problem depends on a the solution of a smaller version of the problem.

In programming terms, recursion allows a function to call itself. It allows for complex algorithms to be created with a small amount of code. It's one of the most important, and beautiful, ideas in programming.

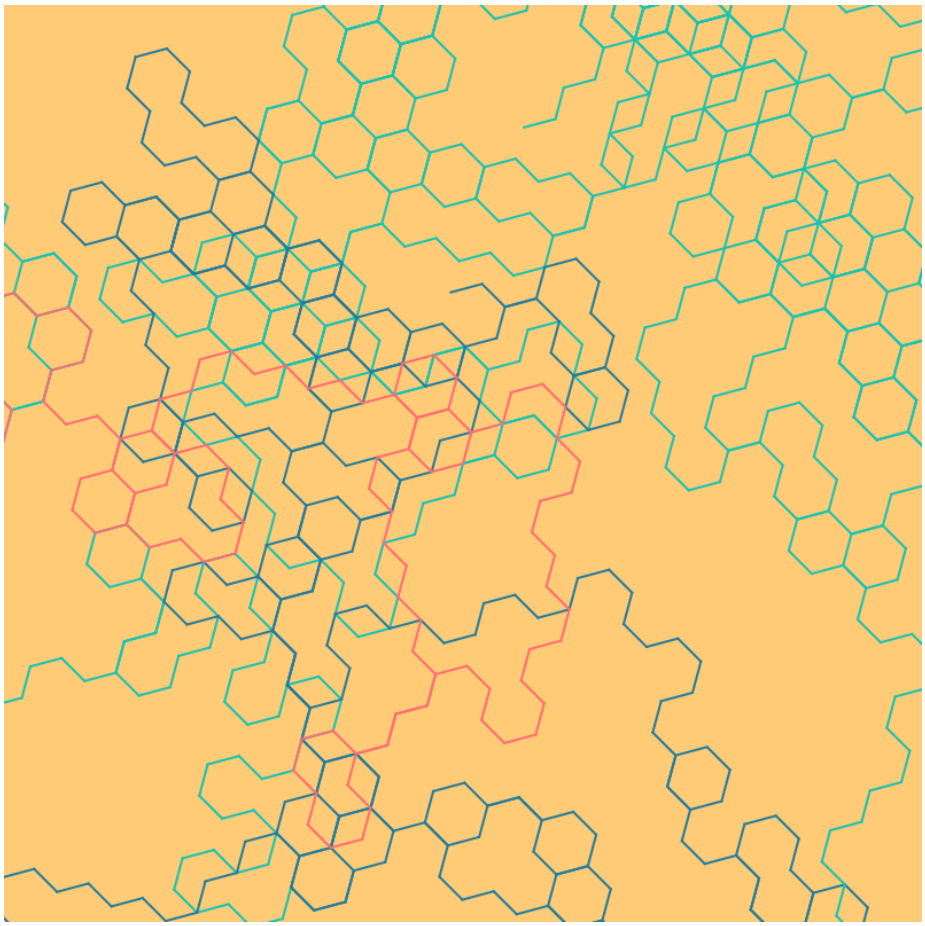


Figure 4: 201004a

It's how we get patterns that repeat themselves at smaller scales. In this case, it's how we can draw three smaller triangles within each triangle.

Code

```
Triangle = function(_x1, _y1, _x2, _y2, _x3, _y3) {
  this.x1 = _x1;
  this.y1 = _y1;
  this.x2 = _x2;
  this.y2 = _y2;
  this.x3 = _x3;
  this.y3 = _y3;

  this.show = function() {
    fill(255);
    noStroke();
    triangle(this.x1, this.y1, this.x2, this.y2, this.x3, this.y3);
  }

  this.recurse = function(maxDepth, currentDepth){
    if(currentDepth < maxDepth){
      // Calculate midpoints to create smaller triangles
      let mpx1 = this.x1 + ((this.x2 - this.x1) / 2);
      let mpy1 = this.y1  - ((this.y1 - this.y2) / 2);
      let mpx2 = this.x2;
      let mpy2 = this.y1;
      let mpx3 = this.x2 + ((this.x3 - this.x2) / 2);
      let mpy3 = this.y3  - ((this.y3 - this.y2) / 2);;

      fill(0);
      triangle(mpx1, mpy1, mpx2, mpy2, mpx3, mpy3);

      // Create new triangle
      let bottomLeft = new Triangle(this.x1, this.y1, mpx1, mpy1, mpx2, mpy2);
      bottomLeft.show();
      /*To get the recursion to stop at different depths, I only continue if the random numb
      // Recurse
      if(random(10)<8)bottomLeft.recurse(maxDepth, currentDepth + 1);

      // Repeat for two remaining small triangles
      let top = new Triangle(mpx1, mpy1, this.x2, this.y2, mpx3, mpy3);
      top.show();
      if(random(10)<8)top.recurse(maxDepth, currentDepth + 1);

      let bottomRight = new Triangle(mpx2, mpy2, mpx3, mpy3, this.x3, this.y3);
      bottomRight.show();
```

```
      if(random(10)<8)bottomRight.recurse(maxDepth, currentDepth + 1);
    }
  }
}

function setup() {
  createCanvas(1000, 1000);
  background(0);
  fill(255);
}

function draw() {
  frameRate(1)
  background(0)
    translate(100, 100);
  let t = new Triangle(0, 800, 400, 0, 800, 800);
  t.show();
  let v = int(random(10))
  t.recurse(v, 0);
}
```

Image

Conclusion

A theme in the sketches I make, and in those of many other creative coders, is patterns. I like combining simple shapes and seeing what they turn into.

There may be variations in noise, time, color, but the essence of many sketches is patterns.

Humans are in love with patterns. Our brains are designed to recognize them. It's not surprising then that we seek them out when we code.

When you see patterns in nature, they aren't identical. Each iteration of the pattern is slightly different. When you look at the tiles on a surface they are very similar, but each one is unique. Each one's color will be just a bit different. Each one will be ever so slightly out of alignment. Some may be worn down more than others. Some will have cracks.

With noise and randomness you can create patterns that look real. That's one of the greatest features of creative coding. The understanding you get from coding in turn helps you understand nature better. Everything is the same, but everything is different. Everything has its place, and everything comes together as one.

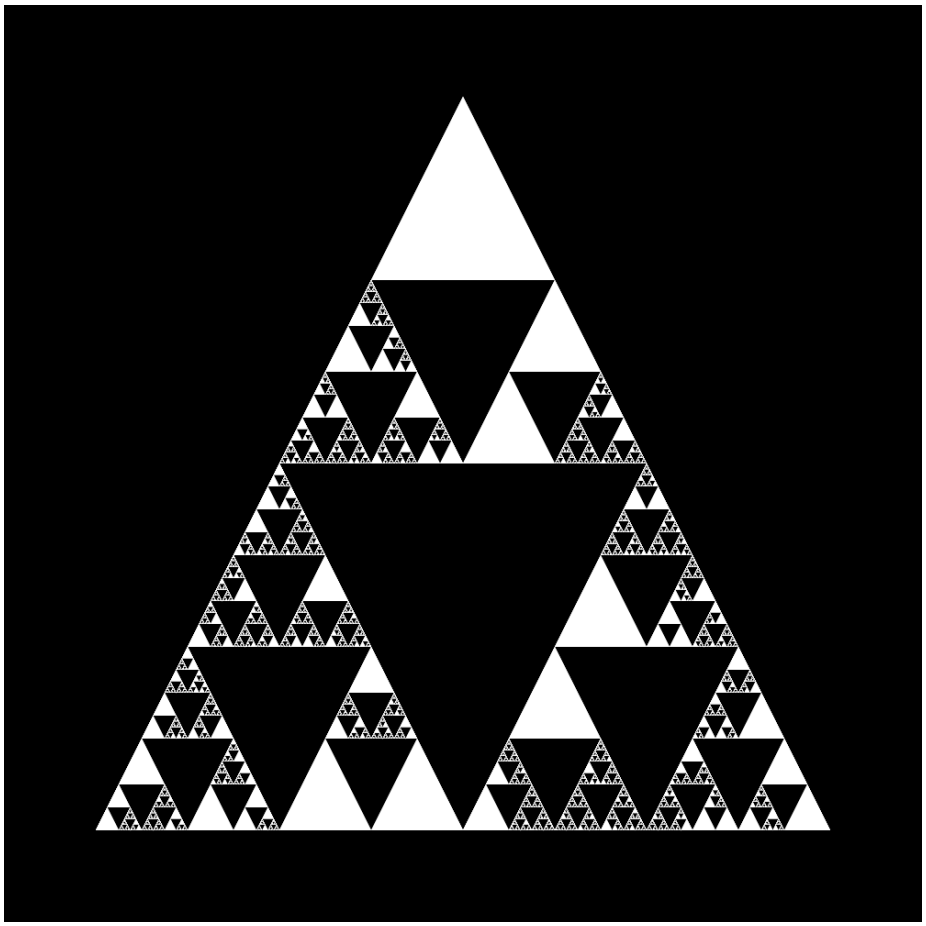


Figure 5: sierpinski_triangle