Generative Art

Caleb Harmon

Ouachita Baptist University

SENIOR THESIS APPROVAL

This Honors thesis entitled

"Generative Art"

written by

Caleb Harmon

and submitted in partial fulfillment of
the requirements for completion of
the Carl Goodson Honors Program
meets the criteria for acceptance
and has been approved by the undersigned readers.

Dr. Jeff Matocha, thesis director

Professor Darin Buscher, second reader

Professor Ferris Williams, third reader

Dr. Barbara Pemberton, Honors Program director

April 16, 2021

Table of Contents

Background

There has been a lot of change throughout our lives due to the advancements throughout technology. One of the most important inventions is the computer. The computer has led to major changes in how the world works and how our culture works. Art has also been affected by the invention of the computer and the Internet. Digital software allows artists to enjoy freedoms that were not possible before the invention of the computer. Computers allow artists to attain precision and accuracy that would not be possible or would be very difficult using traditional methods. Continued development in technology and increasing availability of computers makes digital art accessible to an increasing number of people around the world.

Generative art is a genre in digital art with some characteristics that make it stand out from other types of digital art. Generative art applies many different concepts and ideas from mathematics and computer science to create artwork. Generative art is a system or set of rules that a computer uses to generate artwork. Many pieces of generative art can look very complex or chaotic, however, simple rules can create such a result. Generative art does not require a large amount of programming knowledge, but it can be helpful to have some understanding of programming or mathematics.

I believe that generative art can balance ideas from computer science, mathematics, and visual arts. I think that generative art brings a unique perspective and opportunity than other types of art. Generative art can and should share a lot of similarities with art and often draw inspiration from art history. Generative art can reflect many complex behaviors that occur in nature. Many generative systems mimic actions that occur in biology or other forms of science to try to replicate processes in nature.

Throughout my thesis we will present a variety of techniques and algorithms that can be used to create generative artwork. We will also explore the applications of the techniques and the creative process of creating a generative artwork.

Introduction

I present all code in Processing or p5.js. The developers of Processing call the programs *sketches* to keep an art mindset. Processing is a programming language with a development environment intended for teaching programming concepts through visual context. Processing derives from the programming language Java and uses Java libraries. P5.js is a JavaScript library and the p5.js website has an online web editor making creating sketches quick. Processing and p5.js are popular tools used by many generative artists. There are many other programming languages and tools that have other benefits that Processing and p5.js do not offer, but I choose Processing and p5.js because I am the most familiar with them and they are beginner friendly.

All of the code is displayed in color syntax for the appropriate language. We state the file name at the top of each sketch and include comments in the code for additional notes. We include an appendix for programs that correspond with the paper. Sketches that correspond with Processing are named with the file type "`.pde`" and p5.js sketches are named with the file type "`.js`". The code below presents an example of a template of a Processing sketch. The `Main.pde` file contains the functions defined by the Processing language, `setup()` and `draw()`. The `setup` function is used at the very start of run time to set variables and other settings for the sketch. The `size` function in setup sets the width and height of the window in pixels. The `draw` function is called continuously for each frame of the program until the window is closed or stopped. The `draw` function can be useful in repeated actions and making animations.

Main.pde

```
void setup(){
  size(600,600);
}

void draw(){
  background(255);
}
```

Techniques

In this section, I will be going over several techniques and processes used in generative art and discussing possible applications and variations. The following techniques are only a few among many different processes used in generative art. I have selected these particular processes to demonstrate the wide variety of visuals that can be created through generative art. I have selected techniques that are beginner friendly and other processes that are more advanced.

Generative art includes a wide variety of art and can be difficult to define. Much of generative art utilizes random numbers through computer's pseudo-random number generators. Pseudo-random number generators produce numbers algorithmically that seem random and can be used in statistics and simulations as random numbers. However, they are not truly random since they are generated with an algorithm.

Some generative art does not use a random number generator. Generative art can choose to use camera input, human interaction, or data from APIs or the web to make the artwork. The basics of generative art use a process that is typically random to generate a final product such as art, animation, or sound. Generative art may not even require a computer at all. An example of generative art could be flipping a coin 100 times to determine the color of each square in a 10 x 10 grid on a piece of paper. One of the benefits of using programming and technology for generative art is that the time is greatly reduced. Flipping 100 coins might be reasonable for a human to do, but flipping a million coins would take a significant amount of time for a human compared to the speed of a computer simulating a coin flip a million times.

Random Walkers

One of the simplest techniques used in generative art is a random walker. A random walker is a path that is made up of many points that are decided by a process using random number generators. A random walker usually consists of a current location and a function that can change the location. A vector represents the location of the walker. Random walkers can be used in a variety of different ways such as drawing lines or shapes or manipulating data. Many random walkers appear chaotic and jagged. However, there are many different ways that a random walker can be modified.

In Processing, there is a `setup` and `draw` function. The code in the `setup` function will be called exactly once at the beginning when the program is run. The code in the `draw` function will be called once per frame until the program is turned off manually or through the `exit` function. A simple random walker appears belows as:: `Main.pde` and `Walker.pde`. The `Main.pde` file will instantiate the random walker and call the methods in the walker. The code used is shown below.

```
Main.pde
Walker w;

void setup(){
  size(500,500);
  w = new Walker(250,250);
}

void draw(){
  w.step(); // Change the position through some process
  w.show(); // Visualize the walker
}
```

The `Main.pde` file requires a `Walker.pde` file below in order to run without any errors. The `Walker.pde` file implements the walker class that will provide the functionality of the random walker in the program. The `Walker` class shown below is a very basic example of a random walk. The walker contains two 2d vectors, one which will track the new position of the walk, `position,` and one to store the previous location, `prev`. The class also contains a variable, `len`, to store the length of a step. In the `step` method, the previous location is given the current position. Then a new vector is calculated from random numbers and remembered as the current position.

Walker.pde

```
class Walker{
  PVector position;
  PVector prev;
  float len = 5;

  Walker(float x, float y){
    position = new PVector(x,y);
    prev = position.copy();
  }

  void step(){
    prev = position.copy();
    position.add(new PVector(random(-1,1)*len,random(-1,1)*len));
  }

  void show(){
    line(position.x,position.y, prev.x, prev.y);
  }
}
```

The simplest way to create a random walker is to generate a random direction to move at every step. However, this method of random walker will create a very chaotic result. This can be useful when trying to achieve a chaotic result but many times it can be unpleasant and a smoother approach will be more suitable. In that case, another approach to creating a random walker would be to keep track of the previous angle. By keeping track of the previous angle, the new direction can be generated by changing the previous angle by a random number. The direction of the random walker can be stored as an angle in radians. At each step of the random walker, it calculates a new angle based on the previous angle and a random number and then moves a specified length in the direction of the new angle. The range of the random number chosen can be decreased or increased based on the amount of variation wanted.

```
Walker.pde
class Walker {
  PVector position;
  PVector prev;
  float stepSize = 1; // length in pixels of the step
  float aVariance = 1; // amount of change desired for each step
  float a; // the angle the walker is heading towards

  Walker(float x, float y) {
    position = new PVector(x, y);
    prev = position.copy();
    this.a = random(TAU);
  }

  void step() {
    prev = position.copy();
    a += radians(random(-aVariance, aVariance));
    position.add(stepSize*cos(a), stepSize*sin(a));
  }

  void show() {
    line(position.x, position.y, prev.x, prev.y);
  }
}
```

The following images in figure 1 are the results of the first thousand frames of each random walker. Each method produces a different result with different characteristics. Many variables can be adjusted in order to generate different results with the walkers. The step size can be increased or decreased depending on the size of the artwork or the length desired. Increasing the step size can produce paths with sharp corners at each step and decreasing the step size can create a path with more curves.

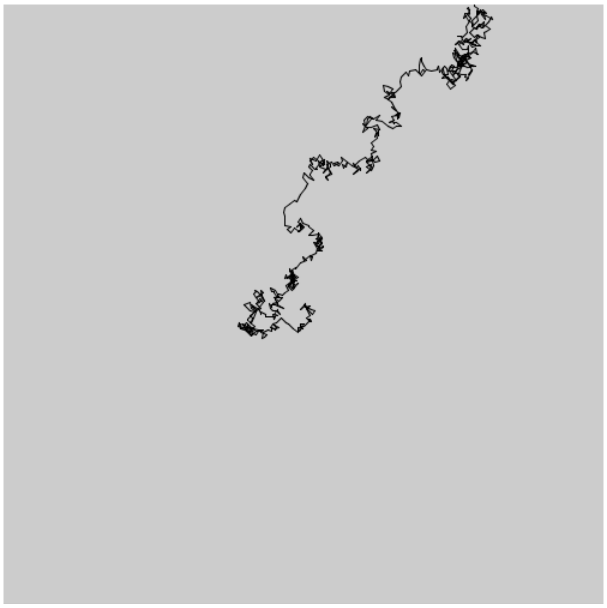

Figure 1a: Random walker that uses a random direction each step. It creates a very chaotic appearance.

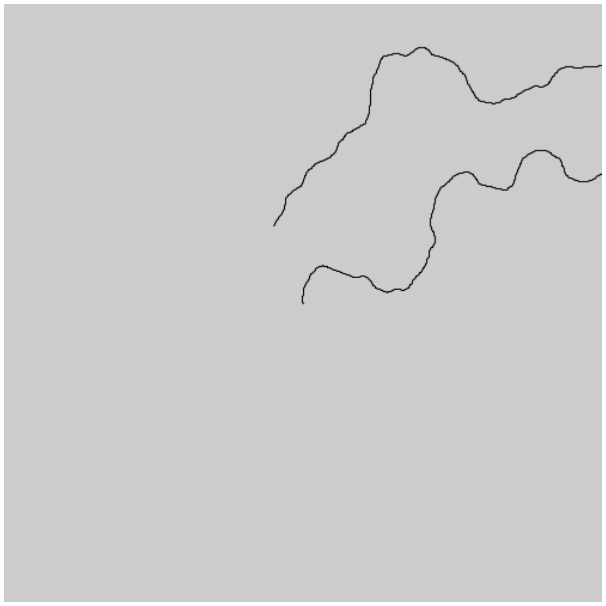

Figure 1b: Random walker that uses the previous angle to determine the new direction. It creates a smoother walk.

The logic and algorithmic part of the random walker is important in creating visually interesting results. However, there are many things that are not logically or algorithmically complex that can add to a generative work. Just like traditional art, it is important to use elements of art in effective ways with random walkers.

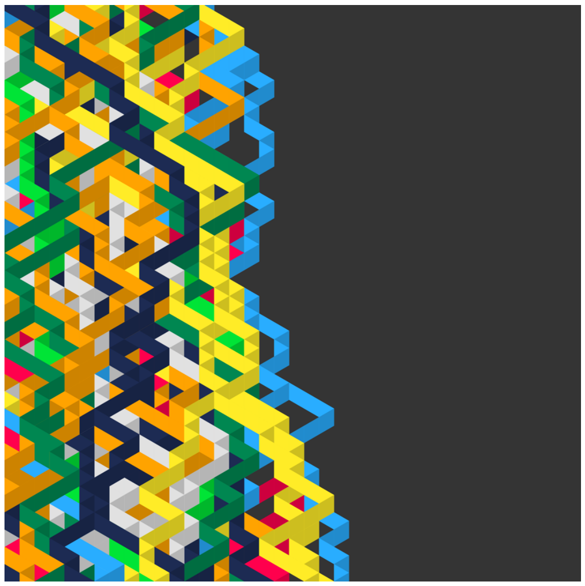

Figure 2: Artwork created using multiple random walkers.

In figure 2, a different variant of random walkers is used. At each step of the random walk, there are only two directions for the walker to choose. The walker can move 30° or 150° from the current location. Each step length is determined by a random integer. Each random walker is created at the top left side of the screen and will travel in the positive y direction until it is out of view. A walker travels randomly until it exits the bottom of the screen and then it is deleted, and a new walker is created. The step of the random walk is displayed as a polygon determined by the direction of the step. The direction also determines the fill color. If the direction is 30°, the fill will be the color determined at the beginning of the walk. Otherwise, if the direction is 150°, it uses a darker shade of the color determined at the beginning of the walk. This introduces some contrast in the random walk and create interesting features and patterns. Each color is randomly selected from a predetermined color palette.

An artist can create more control by creating color palettes to be used in a generative artwork. A generative artwork does not require colors to be randomly generated or picked. However, many generative artworks explore systems to generate colors. Entirely random colors will make colors that are not visually pleasing. By using color palettes, the artist can limit the colors and shades that are used and guarantee that matching colors will be used. Color is a very important aspect in any visual artwork, so utilizing color palettes and other methods of generating colors is important when creating a generative artwork.

Perlin Noise and Flow Fields

Perlin noise is a tool that is widely used throughout computer science and graphics. Random number generators typically create a chaotic appearance when used. So, in many cases there is a need for a technique to allow for randomness but without such a chaotic appearance. An example would be when generating random terrain, if using a random number generator, it would make the terrain very rough. But by using Perlin noise, it is possible to generate smooth terrain. Noise can be used to replace random numbers in many applications to give a more natural behavior or appearance.

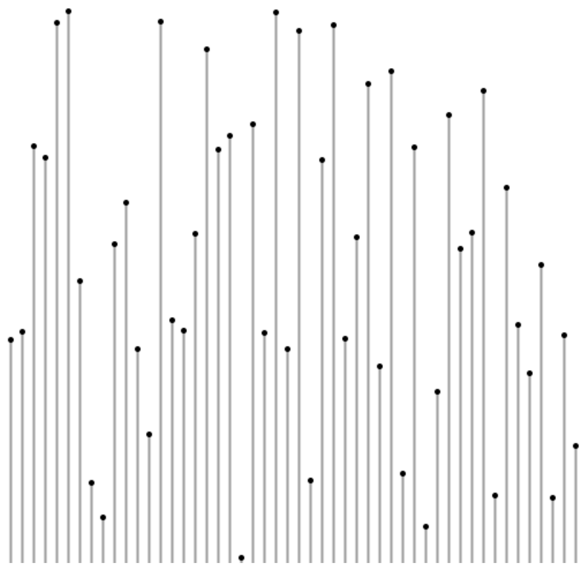
Figure 3a: Vector y values selected using a pseudo-random number algorithm.

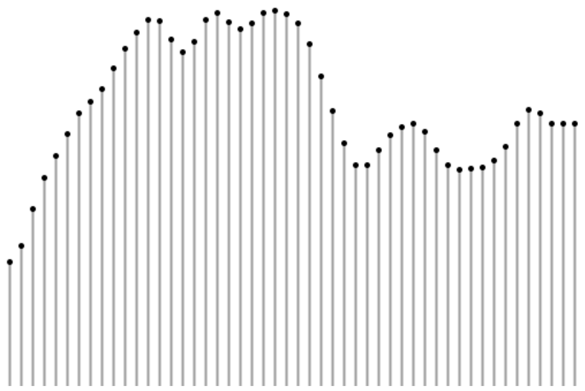
Figure 3b: Vector y values selected using a Perlin noise function.

In figure 3, y values of vectors that are evenly spaced on the x-axis are determined by a pseudo-random number algorithm and Perlin noise. The pseudo-random number algorithm as shown in figure 3a, creates a very chaotic image with no clear curve. However, the Perlin noise, shown in figure 3b, creates a clear curve with the points. The curve resembles a more natural pattern such as hills. This allows for the creation of natural textures and shapes that can be made using Perlin noise.

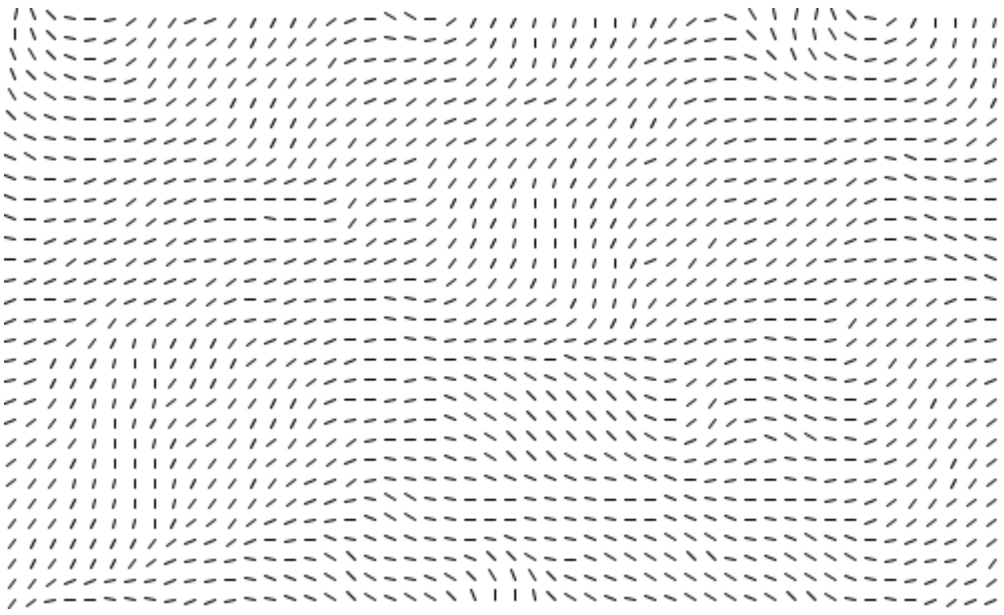

Figure 4: A vector field using Perlin noise to determine the angles.

The image above is created by the `FlowField.pde` program. The program creates a flow field from using the Perlin noise function, `noise()`, built into Processing and using the `map` function to get values between zero and two pi. The `map` function is a very useful function since the noise function provides numbers between 0 and 1. The `map` function takes the noise value in the 0 to 1 domain and can return the corresponding number in the new range. This can be helpful because in many applications since numbers often should be in a different range. Angles in mathematics are generally in the range from 0 to two pi and so using the noise function with numbers between 0 and 1 would limit the range of angles available. Therefore, by mapping the number to the range 0 and two pi, every angle can potentially be created.

```
FlowField.pde
void setup() {
  size(500, 300);
  background(255);
  float scale = 0.01;
  for (int i = 0; i < 50; i++) {
    for (int j = 0; j < 30; j++) {
      int x = 10*i+5;
      int y = 10*j+5;
      float a = map(noise(x*scale,y*scale),0,1,0,TAU);
      line(x,y,x+5*cos(a),y+5*sin(a));
    }
  }
  saveFrame("flowfield.png");
}
```

Cellular Automata

Cellular Automata use simple rules to create complex behaviors of a grid of cells. Two of the most common cellular automata rules are *Conway's Game of Life* and *one-dimensional cellular automata*. Both are capable of creating complex structures and interesting patterns. There are many examples of cellular automata being used to create art. One-dimensional cellular automata appear commonly in forms of generative art because it allows the viewer to see the entire structure in a two dimensional medium. Conway's Game of Life can still be used in different forms of generative art such as in three dimensional forms or animations.

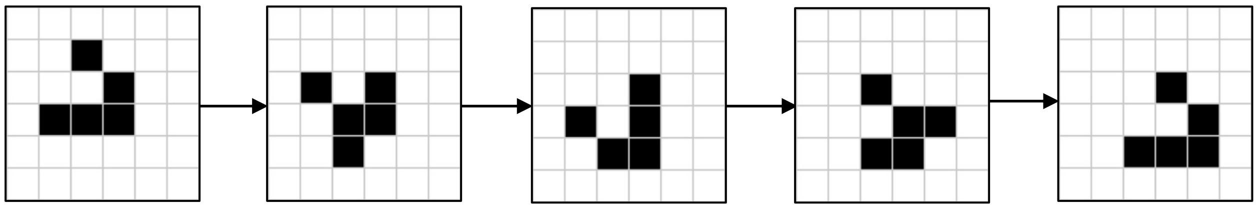

Figure 5: An example of a glider in Conway's Game of Life. Since the first state is identical to the last state it will repeat until interrupted.

Conway's Game of Life is played on a two-dimensional grid such as shown in figure 5. Each cell is assigned an initial configuration of being alive or dead. The initial configuration is shown in the leftmost image in figure 5. The cells that are alive are colored black and the dead cells are colored white. After the initial configuration is set, the game is played in generations. Each new generation is determined by the previous generation. Three rules determine if a cell is alive in the next generation: If the current cell is alive and has exactly 2 or 3 alive neighbors, it will stay alive in the next generation; If the current cell is dead and has exactly 3 alive neighbors, it will become alive in the next generation; Otherwise, the cell will be dead in the next generation. These simple rules create complex behaviors from the cells and the initial configuration. The initial configuration in figure 5 depicts a pattern known as a glider. This specific pattern will move itself one row down and one column across infinitely until other cells interfere with it. Conway's Game of Life can be used in generative art to create complex and interesting behaviors. Creating animations through the generations of cells is common in generative art.

One-dimensional cellular automata, also known as elementary cellular automata, are similar to Conway's Game of Life. Conway's Game of Life is a form of two-dimensional cellular automata because it uses a two-dimensional grid for the automata. Elementary cellular automata use a one-dimensional grid for each generation. This allows for time to be added as a second dimension when displaying the automata. In figure 6, the top row of each image is the initial configuration of the automata and each generation is shown in the row below. Adding time as a dimension to a two-dimensional cellular automaton such as Conway's Game of Life would require a three-dimensional image. In elementary cellular automata, the state of a cell in the next generation is determined by the current cell and the two cells next to it. Since there are 3 cells to consider for the next generation and 2 possible states, alive or dead, there are $2^3 = 8$ different combinations of states. All 8 different combinations are depicted in figure 6. Each rule determines the state of the cell for each possible combination. Since there are 8 different combinations of a group of 3 cells, there are $2^8 = 256$ different possible rules. Each rule is converted into binary in order to determine the 8 output states. 30 in binary is 00011110 and 73 in binary is equal to 01001001 and the binary representation corresponds with the output states show in figure 6. The images created by the cellular automata are dependent on initial random input. The initial random input can be controlled or fixed by providing a random seed to the program. A random seed is a starting point for a pseudo random number generator. With a seed, the computer program will run exactly the same every time.

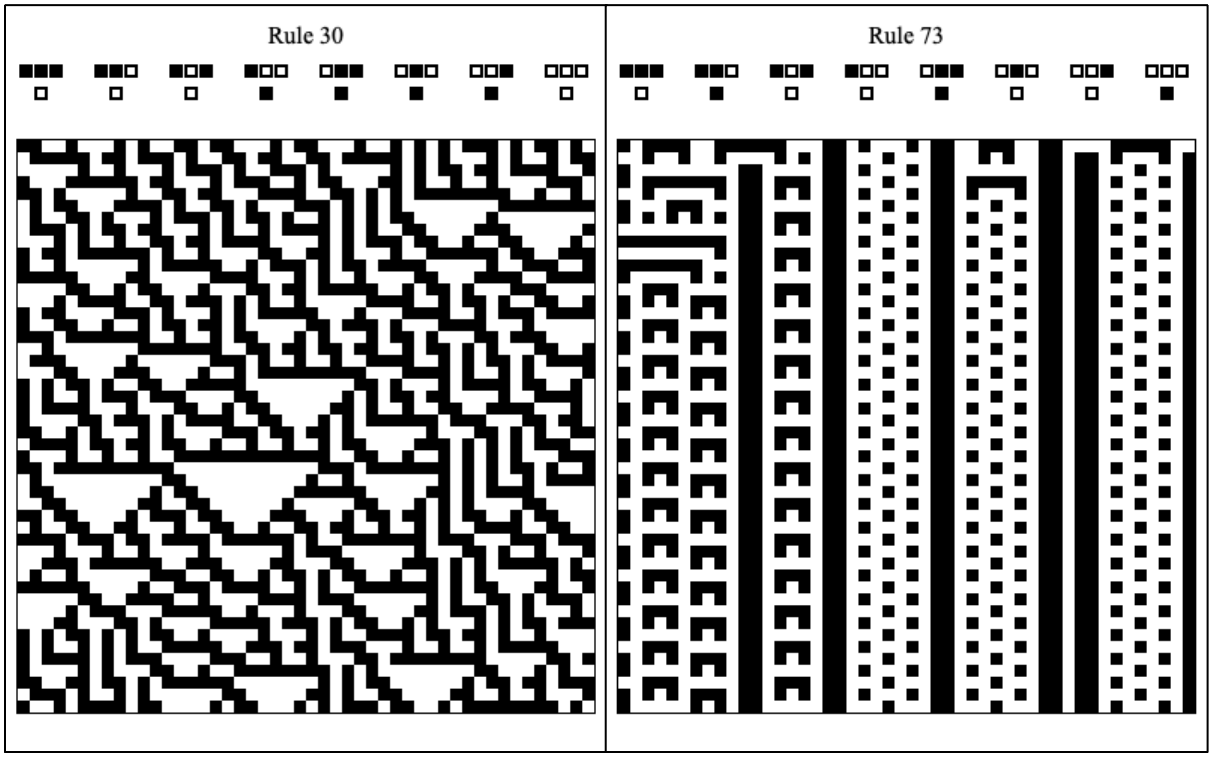

Figure 6: Rules 30 and 73 with sample images.

Elementary cellular automata can produce images that seem very ordered and have clear patterns such as the image produced by Rule 73. However, other rules such as Rule 30 can produce more chaotic and unpredictable structures. There are many ways to create different rules such as changing the grid to an isometric grid or changing the number of states from 2 to 3 or more. Cellular Automata can be useful in giving patterns that are similar in every iteration but unique.

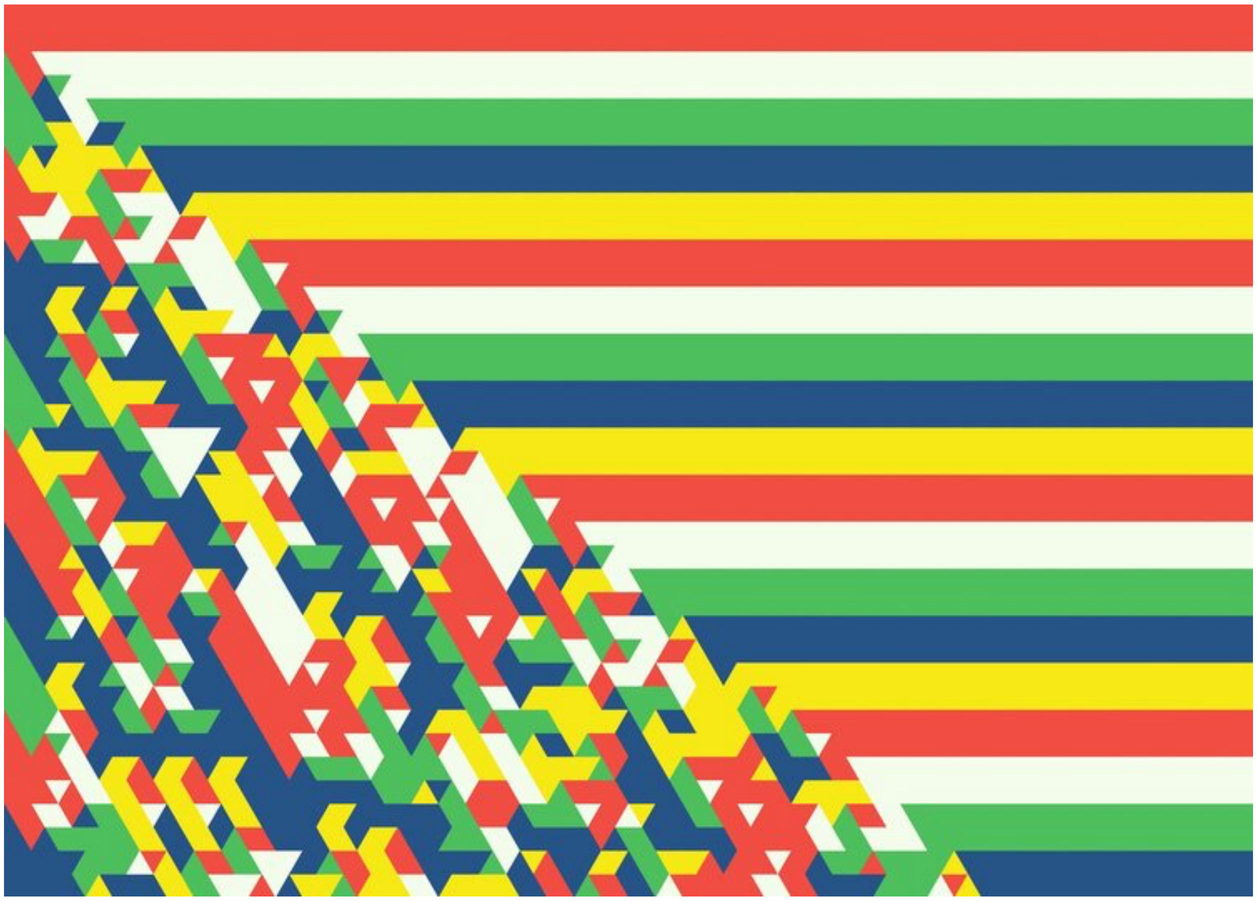

Figure 7: Kjetil Golid, *Complex 02*

In Kjetil Golid's *Complex 02*, One-dimensional automata are used to generate the patterns in the image. Instead of having cells be squares in a grid, the cellular automata determine the existence of a line in a hexagonal grid. After all of the lines are placed, another cellular automata is used to determine the fill colors of the shapes left on the grid.

Diffusion Limited Aggregation

Diffusion Limited Aggregation (DLA) is an algorithm that simulates growth and forms a structure known as a Brownian Tree. Brownian Trees are structures that simulate crystallization and form a tree-like fractal shape. DLA is one of many different computer algorithms that simulate growth. These algorithms can be used in art to get very organic and natural looking appearances. DLA and other growth algorithms can require millions of particles to simulate. In many programs it is necessary to upgrade the computer's hardware or optimize to produce the best results.

The DLA algorithm starts with a fixed particle or object that can be collided with. Then particles are added to a data structure. Each particle moves around as a random walk and continues until it collides with a fixed particle or object. Once a particle collides with a fixed particle or object, it becomes fixed. This causes growth from the fixed point or object. In figure 8, the algorithm starts with a fixed point in the middle of the screen. All of the collided particles are connected to the middle particle due to this. Random walkers are not very efficient, since the probability of a random walker hitting the first point is very low. The algorithm can be altered, such as in figure 8, so the particles are placed randomly around the center and move towards the center. However, the altered algorithm produces a different growth pattern than pure random walks but saves time.

In my DLA implementation using p5.js, I implemented a QuadTree to optimize the search for particles. A QuadTree contains a boundary box that will contain points that are inside of the box. The QuadTree contains a variable for the capacity of points. If the capacity is met or exceeded the QuadTree will subdivide into four regions: north-east, north-west, south-east, south-west. Then the points are split into the corresponding division. The QuadTree allows for a logarithmic search for points in a range.

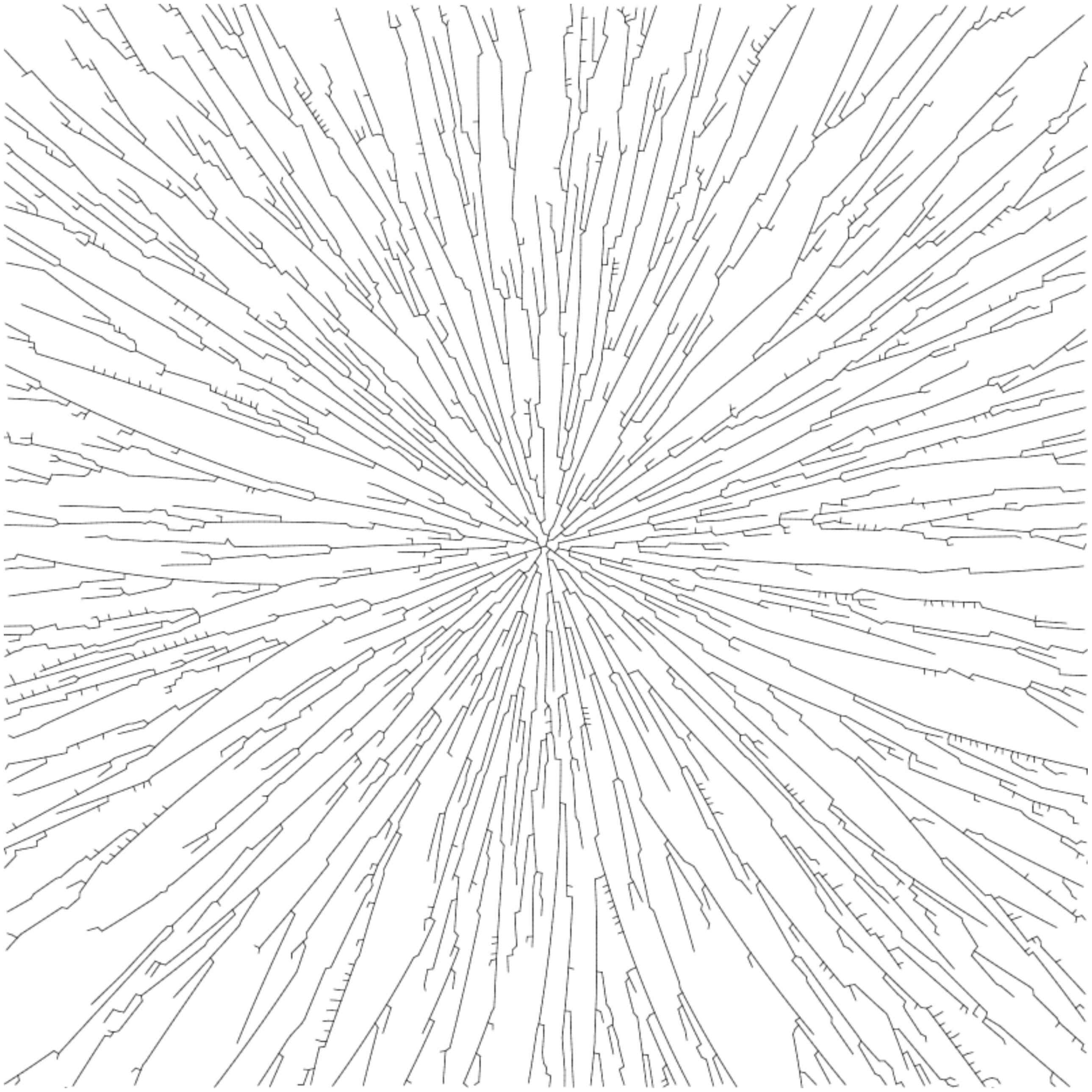

Figure 8: Diffusion Limited Aggregation using 20,000 particles.

Artwork

In this section, I will be exploring some artwork made by myself or other artists. I will be describing how each process works and the creative process behind each of the artwork. The code for much of the art will be in the appendix. The content will focus more on the artistic aspects of each of the pieces but will contain some technical details when appropriate.

Some of the artwork that will be analyzed will be accompanied by Processing code that was written by myself to produce similar outputs to the original piece of art. Many of these pieces were made many decades before Processing was made.

Many generative art pieces use mechanical plotters to produce the final work. Mechanical plotters are machines that can draw on a two-dimensional plane. Plotters use special file formats such as vector files to draw paths on the paper or other medium the artist is using. While it is possible to use printers to turn digital images into physical mediums, plotters can use pens and other tools in order to give the artwork a more traditionally drawn appearance.

A large part of generative art is constructing tools that can be used or modified in the creation of a piece of art. There are many times while making a new tool or process that it produces an unexpected result or can move in a different direction.

Schotter (Gravel)

Schotter (Gravel) by Georg Nees is one of the earliest examples of generative art. The artwork is composed of 24 rows of 12 squares each. The squares on the first row appear to be in a line with each edge touching the next square's edge. As the rows go down from the top of the piece the position of the squares is gradually altered by random numbers as well as the angle of the squares.

The rows of squares at the top of the page appear very ordered while the squares towards the bottom of the page appear more chaotic through the randomness that has been applied to them. However, the chaos creates more visually intriguing elements in the piece by creating more complex geometry through the intersections of multiple squares. While looking at the piece as a whole, there is a clear change from order and chaos from the top and bottom.

The piece creates an illusion of the squares breaking apart from each other through the gradual randomness in the angle and location that causes the squares to separate and intersect with each other. The piece is also made through very simple rules that are apparent to the viewer but creates a complex form and geometry at the end result.

The piece of art demonstrates that the amount of randomness can be controlled by the artist in order to achieve effects that are desired. The number of rows and columns are precisely chosen by the artist. In generative art, there is a balance between random and controlled variables by the artist. Controlled variables give the artist an idea of how the generated image will look at the end while not completely knowing the result. The amount of randomness and order in the program can determine the mood and tone of the resulting piece of art.

While recreating *Schotter (Gravel)* in Processing, it only takes a minimum amount of code to produce interesting results. The code is made through using two for loops to draw each row and column of the image.

```
Schotter.pde
void setup(){
  size(600,800);
  background(255); // Set background to white
  translate(50,50); // Move origin to (50,50) to create a margin
  rectMode(CENTER); // Draw rectangles according to center point
  noFill(); // Draw just the outlines of the squares

  for(int y = 0; y < 14; y++){
    float variance = map(y,0,14,0,1);
    for(int x = 0; x < 10; x++){
      pushMatrix();
      translate(x*50+25,y*50+25);
      rotate(random(-variance*PI/4.0,variance*PI/4.0));
      rect(random(-variance*15,variance*15),random(-
variance*15,variance*15),50,50);
      popMatrix();
    }
  }
  saveFrame("schotter.png");
}
```

All of the code is run in the setup function because one image file will be created and there will be nothing to animate. The program utilizes functionality of the Processing library such as the rect function to draw the squares and methods to move and rotate the objects in the sketch. In the figure below, the original piece by Georg Nees is shown next to the output from the processing sketch

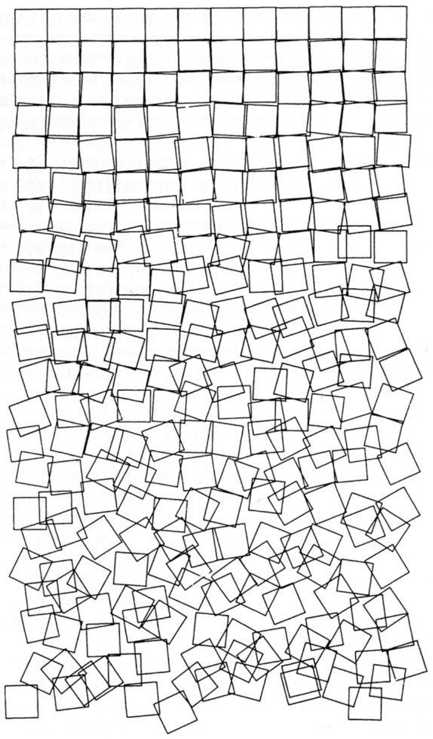

Figure 9a: Gravel by Georg Nees

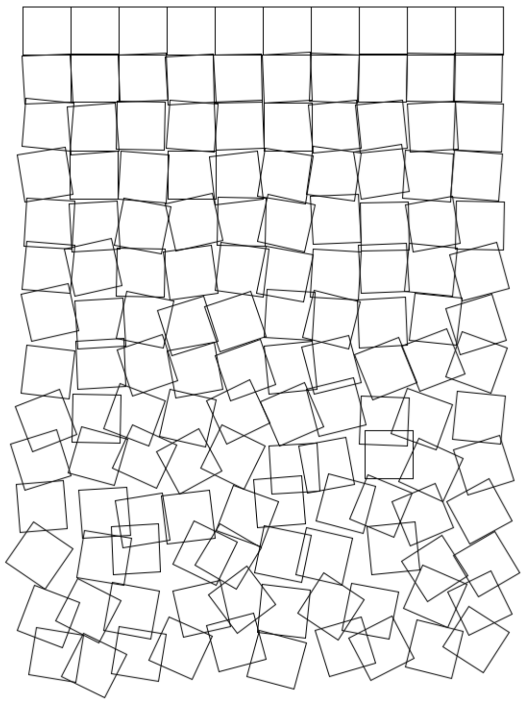

Figure 9b: Processing output

Generative Oranges

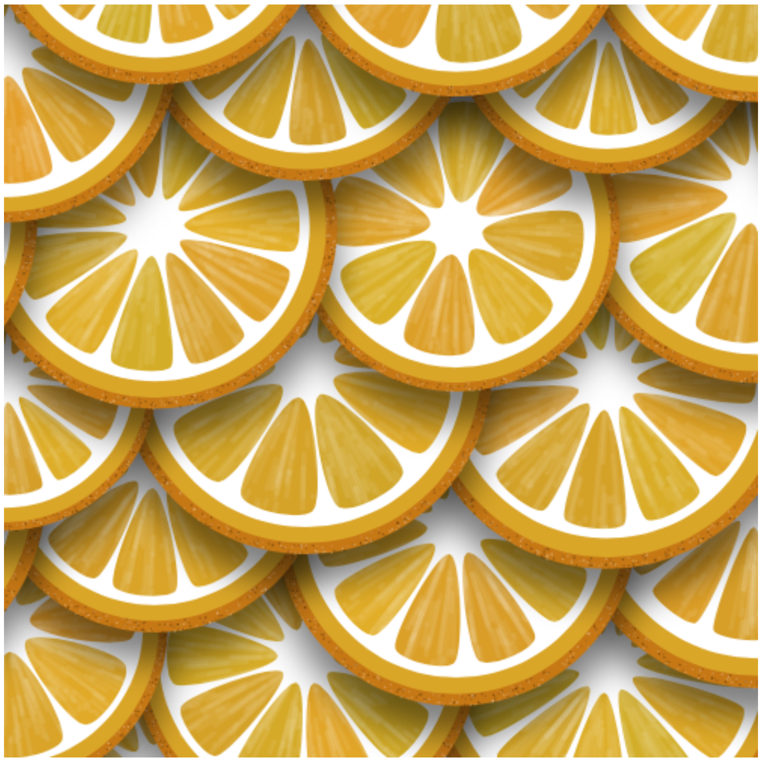

Figure 10: Oranges created using a generative system.

Generative Oranges is a program that I wrote to attempt to replicate the appearance of an orange slice through random generation. I had previously been experimenting with bézier curves and writing functions that would make shapes using the curves that could be used with random numbers. Bézier curves are curves that are defined by anchor points and control points and use mathematical functions to produce a curve. They are common in vector graphics to draw lines and curves in design. An image from the functions that I was experimenting can be seen to the right. It depicts a triangle pointing to the right and a triangle made with curves at the bottom facing left with iterations of the triangle between the top and bottom triangle. The function can take any shape and round the corners using the bézier curves. I used the function as one of the main features in the generative oranges. Each orange is randomly assigned a number of triangles which are then calculated using fractions of 2π. The bézier curves allow the triangles in the orange to look rounded and give the oranges a more organic and natural appearance than regular triangles would. Each orange also has three concentric circles with radii that are randomly modified to provide a slight variation. The largest of the three circles also contains randomly placed points that provide a slight texture to the outside of the orange. Finally, there are lines that come out of the center of the orange over the triangles in the center of each orange to provide additional texture.

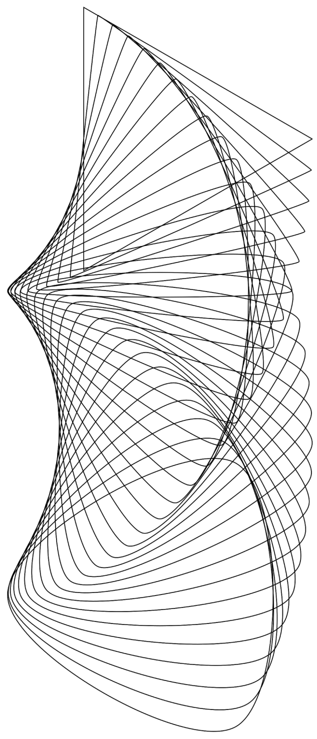
Figure 11: Experiment using bézier curves.

Each orange is created using the same function and the final image is made by composing the oranges together in the staggered layout. The oranges are created in a loop that goes from the bottom to top and right to left. The order is important because the oranges will appear in the final image in the reverse order than which they were created. That means that the oranges at the top left appear closer than the oranges on the bottom right. The size of each orange and center are randomly assigned creating uneven overlaps between each of the oranges on the screen. Each orange is also given a shadow to provide more depth to the image.

This piece features a common approach that many generative artists use in their work. A piece may contain many different elements which are made using the same process combined into one. Most of these works will have a grid with each object separate. Each orange in the artwork can stand alone as a single piece but are brought together in a grid and can see clear differences between each one.

Genetic Flowers

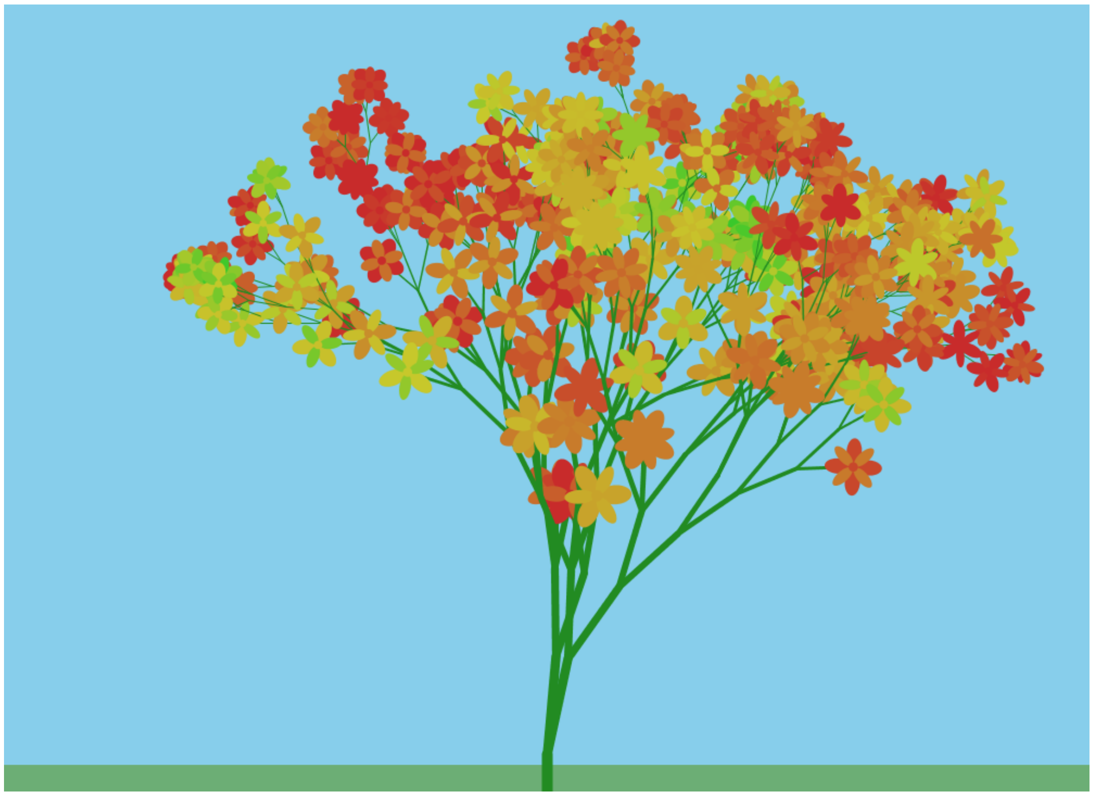

Figure 12: A result image from Genetic Flower program.

Many algorithms and concepts in computer science borrow ideas in other areas of science such as biology. A useful concept is genetic algorithms which has many practical applications such as artificial intelligence and optimization. Genetic algorithms can have useful applications to generative art as well.

A genetic algorithm consists of a population which contains genes. The population will evolve over time by creating children and removing objects in the population. Every new population is called a generation and the later generations evolve to solve a particular problem. There are three core parts of a genetic algorithm. The first part is heredity, which means that children objects receive genes from their parent object. In figure 12, each flower is given genes from the parent flower on the tree. The second part of a genetic algorithm is mutation or variation. Mutation changes the genes that are passed to the child, there are many different ways to mutate the genes but in the flowers the genes are just randomly mutated. The final part of the algorithm is selection. Selection is used to steer the generations by selecting genes to mutate that have a better chance at creating a solution. A fitness function is used to determine which objects are more fit than others. In generative art, a fitness function can be the viewer or artist's visual preference. By using genetic algorithms, an artist is able curate objects or images produced by a system.

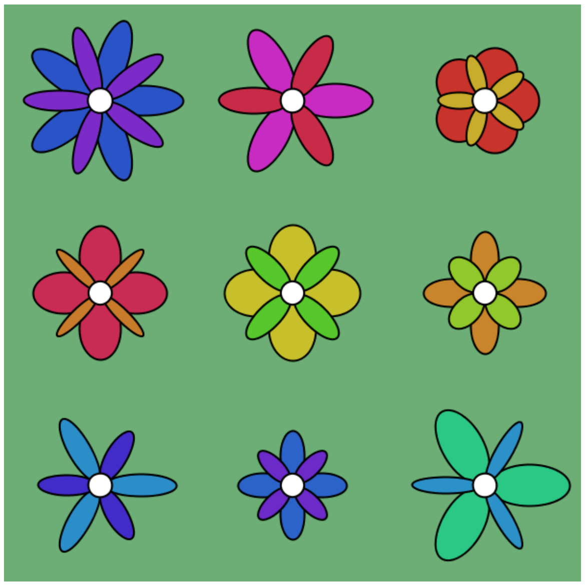
Figure 13: Grid of flowers using a generator.

While creating the genetic flowers, the original goal was to create a system to generate flowers. In figure 13, results of the flower generator are placed in a grid. The flower generator creates flowers out of ellipses. Each flower contains two sets of petals that alternate around the flower. Each set is assigned a petal length, width, and color. The saturation and brightness of the colors of each set of petals are the same while the hue of the colors is random. By keeping the saturation and brightness the same the colors will match each other better. After creating the flower generator, I modified the generator to use genes instead of random numbers. Then the flowers were place onto a random tree by using recursion. The genes are passed each segment of the tree and gets mutated each time. This creates a more natural visual to the end result and creates areas of the flower with unique properties than other areas.

Conclusion

Computers have changed the world in many ways, Art has been greatly affected due to technology and the increased use of software and digital art. Software has greatly enhanced artist’s abilities to express themselves and explore creativity. Generative art is a unique blend of Mathematics, Computer Science, Art, and Design. Automation allows for artists to create worlds and experiences that would have never been possible before. The continued improvement in technology has allowed generative art to become more accessible through tools such as Processing.

There are many different techniques that are used by generative artists and tools that can be used. There are many options of open-source software and libraries that can aid in generative art through using more advanced technical and mathematical concepts.

Bibliography

Bailey, Jason. “The Game of Life - Emergence in Generative Art.” *Artnome*, 14 July 2020, www.artnome.com/news/2020/7/12/the-game-of-life-emergence-in-generative-art.

Bailey, Jason. “Why Love Generative Art?” *Artnome*, 26 August 2018, https://www.artnome.com/news/2018/8/8/why-love-generative-art. Accessed 20 January 2021.

“Diffusion-Limited Aggregation.” Wikipedia, Wikimedia Foundation, 12 Mar. 2021, en.wikipedia.org/wiki/Diffusion-limited_aggregation#cite_note-1.

Gardner, Martin. “Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game ‘Life.’” *Scientific American*, Oct. 1970.

Golid, Kjetil. “Crosshatch Automata.” Medium, 18 Aug. 2020, kjetil-golid.medium.com/automaton-visuals-90bdd9f73286. Accessed 14 April 2021.

Hobbs, Tyler. *Flow Fields*. 3 Feb. 2020, tylerxhobbs.com/essays/2020/flow-fields.

Nees, Georg. *Schotter (Gravel)*. 1968, Germany.

Shiffman, Daniel. *The Nature of Code: Simulating Natural Systems with Processing*. 2012.

Weisstein, Eric W. "Cellular Automaton." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/CellularAutomaton.html

Appendix A: Elementary Cellular Automata

Sketch.js

```
function setup() {
  createCanvas(400, 500);
  let rule = 73
  textAlign(CENTER, CENTER);
  textFont("Times New Roman")
  textSize(16);
  background(255);
  graphicRule(rule)
  save("Rule"+rule+".png")
}

function displayGeneration(gen, row, l, w, n, stretch = true){
  noStroke();
  var start = w;
  push();
  for(let i = (stretch?row:0); i < Math.min(gen.length,n+row); i++){
    if(gen[i]){
      fill(0);
    }else{
      fill(255);
    }
    rect(l*(i-(stretch?row:0)),row*l,l,l);
  }
  pop();
}

function automata(arr, rule, stretch = true, wrap = true) {
  var ruleArr = getRuleArr(rule);
  var result;
  if (arr.length == 0) {
    result = [false];
  } else {
    result = new Array(arr.length);
    if (arr.length == 1) {
      result[0] = ruleArr[boolsToInt(getArr(-1,arr,wrap), arr[0],
getArr(1,arr,wrap))];
    } else {
      result[0] = ruleArr[boolsToInt(getArr(-1,arr,wrap), arr[0],
arr[1])];
      for (let i = 1; i < arr.length - 1; i++) {
        result[i] = ruleArr[boolsToInt(arr[i - 1], arr[i], arr[i +
1])];
      }
      result[arr.length - 1] = ruleArr[boolsToInt(arr[arr.length - 2],
arr[arr.length - 1], getArr(arr.length,arr,wrap))];
    }
    if (stretch) {
      result.unshift(ruleArr[boolsToInt(getArr(-2,arr,wrap), getArr(-
1,arr,wrap), arr[0])]);
      result.push(ruleArr[boolsToInt(arr[arr.length - 1],
getArr(arr.length,arr,wrap), getArr(arr.length+1,arr,wrap))]);
    }
  }
  return result;
}

function getArr(pos, arr, wrap){
  var result;
  if(pos<0){
    if(wrap){
      result = arr[arr.length+pos];
    }else{
      result = false;
    }
  }else if(pos>arr.length){
    if(wrap){
      result = arr[pos%arr.length];
    }else{
      result = false;
    }
  }else{
    result = arr[pos];
  }
  return result;
}

function boolsToInt(bool1, bool2, bool3) {
  var result = 0;
  if (bool1) {
    result += 1;
  }
  if (bool2) {
    result += 2;
  }
  if (bool3) {
    result += 4;
  }
  return result;
}

function graphicRule(rule){
  text("Rule "+rule,200,20);
  for(let i = 0; i < 8; i++){
    let arr = getCellStatesArr(7-i)
    for(let j = 2; j >= 0; j--){
      if(arr[j]){
        fill(0)
      }else{
        fill(255)
      }
      strokeWeight(2);
      stroke(0);
      rect((i-4)*50 + 200 + ((2-j)*10)+10,40,7,7);
    }
  }
  let arr2 = getRuleArr(rule);
  for(let i = 7; i >= 0; i--){
    if(arr2[7-i]){
      fill(0)
    }else{
      fill(255)
    }
    rect((i-4)*50 + 220,55,7,7);
  }
  let gen = []
  for(let i = 0; i < 48; i++){
    gen.push(random(1)>0.5);
  }
  push()
  translate(8,90);
  stroke(0)
  strokeWeight(2)
  rect(0,0,48*8,48*8)
  for(let i = 0; i < 48; i++){
    displayGeneration(gen,i,8,0,48);
    gen = automata(gen,rule,true,false)
  }
  pop()
}

function getCellStatesArr(n) {
  var result = [];
  for (let i = 0; i < 3; i++) {
    result.push(n % 2 == 1);
    n = floor(n / 2);
  }
  return result;
}

function getRuleArr(rule) {
  var result = [];
  for (let i = 0; i < 8; i++) {
    result.push(rule % 2 == 1);
    rule = floor(rule / 2);
  }
  return result;
}
```

Appendix B: Diffusion Limited Aggregation

sketch.js

```
var b;
var q;
var curr;
var d, n;
let num = 20000;
var h = 0;
var id = 0;

function setup() {
  d = new Date();
  n = d.getTime();
  createCanvas(800, 800);
  b = new AABB(400, 400, 600);
  q = new QuadTree(b);
  q.show()
  q.insert(new Particle(400, 400, 0, 0))
  let a = random(TAU)
  curr = createVector(400 + 600 * cos(a), 400 + 600 * sin(a));
  background(255);
  strokeWeight(2);
  stroke(0,0,255);
  point(400,400);
}

function draw(){
  for (let n = 0; n < 1000; n++) {
    var collided = false;
    while (!collided) {
      let ps = q.queryRange(new AABB(curr.x, curr.y, 3));
      var minDist = null;
      var minParticle = null;
      for (let i = 0; i < ps.length; i++) {
        if (minDist == null) {
          minDist = dist(curr.x, curr.y, ps[i].x, ps[i].y);
          minParticle = ps[i]
        } else if (minDist > dist(curr.x, curr.y, ps[i].x, ps[i].y)) {
          minDist = dist(curr.x, curr.y, ps[i].x, ps[i].y);
          minParticle = ps[i]
        }
      }
      var dir = createVector(400 - curr.x, 400 - curr.y);
      dir.normalize();
      if (minDist == null) {
        curr = curr.add(dir);
      } else if (minDist <= 2.01) {
        id = id + 1
        q.insert(new Particle(curr.x,curr.y,id,minParticle.id));
        strokeWeight(2);
        stroke(0,0,255);
        point(curr.x,curr.y);
        let a = random(TAU)
        curr = createVector(400 + 600 * cos(a), 400 + 600 * sin(a));
        collided = true;
      } else {
        curr = curr.add(dir.mult(minDist - 2))
      }
    }
  }
  num -= 1000;
  var d2 = new Date();
  console.log(num);
  if(num <= 0){
    console.log(q.getText())
    save(q.getText(),"dla.txt")
    noLoop();
  }
}
```

quadTree.js

```
//https://en.wikipedia.org/wiki/Quadtree

class QuadTree {
  constructor(boundary) {
    this.node_capacity = 5;
    this.boundary = boundary;
    this.points = [];
    this.northEast = null;
    this.northWest = null;
    this.southEast = null;
    this.southWest = null;
  }

  subdivide() {
    if (this.northEast == null) {
      let q = this.boundary.halfLength / 2.0;
      let NE = new AABB(this.boundary.x + q, this.boundary.y - q, q)
      let NW = new AABB(this.boundary.x - q, this.boundary.y - q, q)
      let SE = new AABB(this.boundary.x + q, this.boundary.y + q, q)
      let SW = new AABB(this.boundary.x - q, this.boundary.y + q, q)
      this.northEast = new QuadTree(NE)
      this.northWest = new QuadTree(NW)
      this.southEast = new QuadTree(SE)
      this.southWest = new QuadTree(SW)
    }
  }

  insert(p) {
    if (!this.boundary.contains(p)) {
      return false
    }

    if (this.points.length < this.node_capacity && this.northEast ==
null) {
      this.points.push(p);
      return true;
    }

    if (this.northEast == null) {
      this.subdivide();
      for (let i = 0; i < this.points.length; i++) {
        if (this.northEast.boundary.contains(this.points[i])) {
          this.northEast.points.push(this.points[i]);
        } else if (this.northWest.boundary.contains(this.points[i])) {
          this.northWest.points.push(this.points[i]);
        } else if (this.southEast.boundary.contains(this.points[i])) {
          this.southEast.points.push(this.points[i]);
        } else if (this.southWest.boundary.contains(this.points[i])) {
          this.southWest.points.push(this.points[i]);
        }
      }
      this.points = [];
    }

    if (this.northEast.insert(p)) {
      return true
    }
    if (this.northWest.insert(p)) {
      return true
    }
    if (this.southEast.insert(p)) {
      return true
    }
    if (this.southWest.insert(p)) {
      return true
    }
    return false
  }

  queryRange(range) {
    var result = []
    if (!this.boundary.contains(createVector(range.x, range.y)) &&
!range.contains(createVector(this.boundary.x,this.boundary.y)) &&
!this.boundary.intersect(range)) {
      return result
    }

    for (let i = 0; i < this.points.length; i++) {
      if (range.contains(this.points[i])) {
        result.push(this.points[i])
      }
    }
    if (this.northWest == null) {
      return result
    }

    result = result.concat(this.northWest.queryRange(range))
    result = result.concat(this.northEast.queryRange(range))
    result = result.concat(this.southWest.queryRange(range))
    result = result.concat(this.southEast.queryRange(range))
    return result
  }

  show() {
    if (this.northEast != null) {
      this.northEast.show();
      this.northWest.show();
      this.southEast.show();
      this.southWest.show();
    }
    stroke(0,50);
    strokeWeight(1);
    this.boundary.show();
  }

  showPoints() {
    if (this.northEast != null) {
      this.northEast.showPoints();
      this.northWest.showPoints();
      this.southEast.showPoints();
      this.southWest.showPoints();
    }
    stroke(0,0,255);
    strokeWeight(2);
    for(let i = 0; i < this.points.length; i++){
      point(this.points[i].x,this.points[i].y);
    }
  }

  getText(){
    var result = []
    for(let i = 0; i < this.points.length; i++){
      let p = this.points[i];
      let str = p.id + ',' + p.parent + ',' + p.x + ',' + p.y
      result.push(str)
    }
    if(this.northEast != null){
      result = result.concat(this.northEast.getText());
      result = result.concat(this.northWest.getText());
      result = result.concat(this.southEast.getText());
      result = result.concat(this.southWest.getText());
    }
    return result;
  }
}
```

AABB.js

```
//https://developer.mozilla.org/en-
US/docs/Games/Techniques/3D_collision_detection

class AABB{
  constructor(x,y,halfLength){
    this.x = x;
    this.y = y;
    this.halfLength = halfLength;
    this.minX = this.x - this.halfLength;
    this.maxX = this.x + this.halfLength;
    this.minY = this.y - this.halfLength;
    this.maxY = this.y + this.halfLength;
  }

  show(){
    noFill();
    rect(this.x-this.halfLength,this.y-
this.halfLength,2*this.halfLength,2*this.halfLength);
  }

  contains(point){
    return (point.x >= this.minX && point.x <= this.maxX) &&
           (point.y >= this.minY && point.y <= this.maxY);
  }

  intersect(other){
    return (this.minX <= other.maxX && this.maxX >= other.minX) &&
           (this.minY <= other.maxY && this.maxY >= other.minY);
  }
}
```

Particle.js

```
class Particle{
  constructor(x,y,id,parent){
    this.x = x;
    this.y = y;
    this.id = id;
    this.parent = parent;
  }
}
```

Appendix C: Generative Oranges

BezierShapes.js

```
function bezierShape(p,factor) {
  var a = [];
  var pos = [];
  var neg = [];

  a.push(angle3(p[0], p[1], p[p.length - 1]));
  for (let i = 1; i < p.length - 1; i++) {
    a.push(angle3(p[i], p[i - 1], p[i + 1]));
  }
  a.push(angle3(p[p.length - 1], p[p.length - 2], p[0]));
  for (let i = 0; i < p.length - 1; i++) {
    let a1 = (PI - a[i]) / 2.0;
    let a2 = angle2(p[i], p[i + 1]) - a1;
    pos.push(a2);
  }
  let n = a.length-1
  let a1 = (PI - a[n]) / 2.0;
  let a2 = angle2(p[n], p[0]) - a1;
  pos.push(a2);
  for(let i = 1; i < a.length; i++){
    let a1 = (PI - a[i]) / 2.0;
    let a2 = angle2(p[i], p[i - 1]) + a1;
    neg.push(a2);
  }
  let b1 = (PI - a[0]) / 2.0;
  let b2 = angle2(p[0], p[p.length-1]) + b1;
  neg.push(b2);
  beginShape();
  vertex(p[0].x,p[0].y);
  for(let i = 0; i < a.length; i++){
    bezierVertex(p[i].x + factor*cos(pos[i]),p[i].y + factor *
sin(pos[i]),p[(i+1)%a.length].x +
factor*cos(neg[i]),p[(i+1)%a.length].y +
factor*sin(neg[i]),p[(i+1)%a.length].x,p[(i+1)%a.length].y);
  }
  endShape();
}
```

```
function angle2(p1, p2) {
  return atan2((p2.y - p1.y), (p2.x - p1.x));
}

function angle3(p1, p2, p3) {
  //arccos((P12^2 + P13^2 - P23^2) / (2 * P12 * P13))
  let p12 = d(p1, p2);
  let p13 = d(p1, p3);
  let p23 = d(p2, p3);
  return acos((p12 * p12 + p13 * p13 - p23 * p23) / (2 * p12 * p13))
}

function d(p1, p2) {
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y -
p2.y))
}
```

Sketch.js

```
function setup() {
  createCanvas(500, 500);
  background("#f9f361");
  for(let i = 7; i >= -1; i--){
    for(let j = 7; j >= -1; j--){
      let w = random(200,250)
      if(i%2==0){
        orange(j*200,i*100+random(50),w);
      }else{
        orange(j*200+100,i*100+random(50),w);
      }
    }
  }
  //save("orangeicon.png");
}

function mousePressed(){
  background("#f9f361");
  for(let i = 7; i >= -1; i--){
    for(let j = 7; j >= -1; j--){
      let w = random(200,250)
      if(i%2==0){
        orange(j*200,i*100+random(50),w);
      }else{
        orange(j*200+100,i*100+random(50),w);
      }
    }
  }
}

function orange(x,y,w){
  push();
  translate(x,y);
  rotate(random(PI/3.0));
  scale(w/275.0);
  noStroke();
  drawingContext.shadowOffsetX = 0;
  drawingContext.shadowOffsetY = 20;
  drawingContext.shadowBlur = 20;
  drawingContext.shadowColor = 'rgba(0,0,0, 0.5)';
  fill("#c67815");
    let r2 = random(0,5);

  let r1 = r2+random(0,5);
  ellipse(0,0,275+r1,275+r1);
  drawingContext.shadowColor = 'rgba(0,0,0, 0)';
  fill("#daa628");
  ellipse(0,0,260+r2,260+r2);
  fill("#f6f1b5");
  fill(255);
  let r3 = random(0,5);
  ellipse(0,0,235+r3,235+r3);
  let c = color("#daa628");
  let num = floor(random(7,13));
    let l1 = 85+random(-3,3);
    let offset1 = 22.5;
    let angle_1 = TAU/(num*2 + random(1,3));

  for(let i = 0; i < num; i++){
    fill(clamp(red(c)+random(-10,10),0,255),clamp(green(c)+random(-
10,10),0,255),blue(c));
    push();
  rotate(i*TAU/num);
  var p = [];
  p.push(createVector(offset1, 0));
  p.push(createVector(offset1+l1*cos(-angle_1),l1*sin(-angle_1)));
      p.push(createVector(offset1+l1*cos(angle_1),l1*sin(angle_1)));
  bezierShape(p,map(num,7,13,15,5));
    pop();
  }
  for(let i = 0; i < random(400,500); i++){
    let a = random(TAU);
    let m = random((235 + r3)/2.0);
    let n = random(m,(235 + r3)/2.0);
    stroke(255,random(15,25));
    strokeWeight(4);
    line(m*cos(a),m*sin(a),n*cos(a),n*sin(a));
    noStroke();
  }
  for(let i = 0; i < random(500,900); i++){
    let a = random(TAU);
    let l = random(260+r2+1,275+r1-1)/2.0;
    if(random(1)>0.5){
      stroke(0,random(50,150));
    }else{
      stroke(255,random(50,150));
    }
    strokeWeight(1);
    point(l*cos(a),l*sin(a));
  }
  pop();
}

function clamp(n,min,max){
  if(n<min){
    return min
  }else if(n>max){
    return max
  }
  return n;
}
```

Appendix D: Genetic Flowers

Main.pde

```
import java.text.DecimalFormat;
import java.math.RoundingMode;
int count = 0;
int runNum;

void setup() {
  size(1000, 800);
  updateNumber();
  generate();
}

void draw() {
}

void mousePressed() {
  generate();
}

void generate() {
  colorMode(RGB);
  background(108, 174, 117);
  noStroke();
  fill(135, 206, 235);
  rect(0, 0, 1800, 700);
  float[] gene = new float[7];
  for (int j = 0; j < gene.length; j++) {
    gene[j] = random(0, 1);
  }
  stem(500, 790, -PI/2.0, 100, 30, 0.1, gene);
  fill(0);
  textSize(16);
  text("DNA: "+geneString(gene), 10, 750);
  textAlign(LEFT);
}

void keyPressed(){
  if(key == 's' || key == 'S'){
    println("SAVE");
    saveFrame("photos/"+runNum + '-' + count + ".png");
    count++;
  }
```

```
}

String geneString(float[] arr){
  DecimalFormat df = new DecimalFormat("##.##");
  df.setRoundingMode(RoundingMode.DOWN);
  StringBuilder result = new StringBuilder();
  result.append('[');
  for(int i = 0; i < arr.length; i++){
    result.append(df.format(arr[i]));
    if(i!=arr.length-1){
      result.append(", ");
    }
  }
  result.append(']');
  return result.toString();
}

void stem(float x, float y, float a, float l, int d, float c, float[]
gene) {
  colorMode(RGB);
  stroke(34, 139, 34);
  strokeWeight(max(map(l, 100, 40, 10, 1), 1));
  float x2 = x + l*cos(a);
  float y2 = y + l*sin(a);
  line(x, y, x2, y2);
  if (d>0 && (random(1)>c || l > 80)) {
    stem(x2, y2, a+map(randomGaussian(),-1,1,-0.4,0), l*.9, d-1, c *
1.2, mutate(gene));
    stem(x2, y2, a+map(randomGaussian(),-1,1,0,.4), l*.9, d-1, c *
1.2, mutate(gene));
    if (random(1)>0.7) {
      stem(x2, y2, a+random(-PI/6, PI/6), l*.9, 0, c * 1.4,
mutate(gene));
    }
  } else {
    pushMatrix();
    translate(x2, y2);
    rotate(random(TAU));
    scale(map(l, 100, 40, .25, .15));
    flower(gene);
    popMatrix();
  }
}

float[] mutate(float[] gene) {
  float[] result = new float[gene.length];
  for (int i = 0; i < gene.length; i++) {
    float g = gene[i];
    g += random(-.05, .05);
    if (g>1) {
      g=1;
    } else if (g<0) {
      g=0;
    }
    result[i]=g;
  }
  return result;
}

void flower(float[] gene) {
  colorMode(HSB);
  stroke(0);
  strokeWeight(4);
  noStroke();
  color c = color(map(gene[0], 0, 1, 0, 255), 200, 200);
  color c2 = color((hue(c) + map(gene[6], 0, 1, 0, 127.5))%255, 200,
200);
  int petals = floor(map(gene[1], 0, 1, 3, 7))*2;
  float petalLength = map(gene[2], 0, 1, 100, 170);
  float petalLength2 = petalLength - map(gene[3], 0, 1, 10, 30);
  float petalWidth = map(gene[4], 0, 1, 40, 100);
  float petalWidth2 = petalWidth - map(gene[5], 0, 1, 10, 30);
  for (int i = 0; i < petals; i++) {
    float l, w;
    if (i<petals/2) {
      fill(c);
      l = petalLength;
      w = petalWidth;
    } else {
      fill(c2);
      l = petalLength2;
      w = petalWidth2;
    }
    if (i==petals/2) {
      rotate(TAU/petals);
    }
    rotate(TAU/petals*2);
    ellipseMode(CORNER);
    ellipse(10, -w/2.0, l, w);
  }
  fill(c);
  ellipseMode(CENTER);
  ellipse(0, 0, 50, 50);
}

void updateNumber() {
  String[] arr = loadStrings("run.txt");
  runNum = Integer.parseInt(arr[0]) + 1;
  PrintWriter output = createWriter("Run.txt");
  output.println(runNum);
  output.close();
}
```