Mohamad Abdullah Mawed

Procedural Content Generation Techniques and Approaches

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
February 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Mohamad Abdullah Mawed

Procedural Content Generation Techniques and Approaches

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
February 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Table of Contents

List of figures

List of tables

Abbreviation list

Abbreviation	
PCG	Procedural Content Generation
PRNGs	Pseudo Random Number Generators
MSM	Middle Square Method
LCG	Linear Congruential Generator
MT	Mersenne Twister
DLA	Diffusion Limited Aggregation
CA	Cellular Automata
SCA	Space Colonization Algorithm
L-System	Lindenmayer System
BSP	Binary Space Partition
RB	Recursive Backtracker
RDA	Reaction Diffusion Algorithm

This is for you my beloved Mom Khawla and Dad Abdullah who have raised me to be the person I am today. I owe you everything!

To my lovely wife, Hanan, for your unconditional love, and your endless patience with all the time I took to writing this thesis.
You are every hope and every dream I have ever had.

To my little angles, Nawar, Taim and Luna. You bring unlimited happiness into my life just by being you. Wherever you go, whatever you do….. I will always be cheering for you.

To my gorgeous siblings, Amani, Baraa and Majd.

I love you all!

Preface

Some people underestimate the power of nostalgia, and they believe that nostalgia is a sign of weakness. For my part, I completely disagree. Usually nostalgia affects my decisions to a large extent. Choosing my master thesis topic was no exception. When I had contact with my supervisor for the first-time last summer, a large collection of different projects was available. Choosing " Procedural Content Generation" was an easy matter. So now you may be wondering why? Again, because of the nostalgia. Let me explain that in more detail.
The story began for ten years ago, in august 2009, when I started my first master thesis in my beloved country, Syria. The dissertation was mainly about genetic algorithms, and how it can be used to control the output of procedural algorithms. That project was a big success in my career, and I can say that with full confidence. I got A-grade, and that moment is unforgettable.
Due to exceptional circumstances, I had to leave my gorgeous country. Fate decreed in 2013 that I have to move to a faraway country called Norway.
Six years later, in 2019, I submit my second master thesis that has a title similar to some extent to the previous one. I believe that procedural generation represents the lucky charm for me. Honestly, I am hopeful that my previous experience with master thesis occurs on more time. That is, getting A-grade eventually.

If that happens this time, my belief in nostalgia will turn to a faith!

Abstract

Nowadays, using PCG gains more and more importance, and PCG plays an increasingly important role in various areas. Particularly in game design and development where PCG have made a qualitative leap forward, especially in recent years.

In fact, PCG is a very broad discipline. Investigating of this matter is so interesting, and satisfaction is guaranteed.

This thesis mainly reviews the primary techniques, approaches and algorithms that are usually used in PCG. Most of the studied approaches and techniques are considered state-of-the-art. This thesis goes beyond the classical way of demonstrating how PCG techniques works unlike any traditional research about this subject. The primary focus is on the practical side rather than purely theoretical matters.

To experience how PCG works in action, over 20 different techniques will be demonstrated, implemented in code line by line and tested. After running each algorithm, the results will be discussed and a comparison with other techniques will be done if that makes sense. At the end of this thesis, the strengths and the weaknesses of each technique will be highlighted. At the end of this thesis, an infinite terrain will be implemented as a practical part of PCG.

Without further ado, let´s get started

Acknowledgment

In fact, I would like to express my gratitude and appreciation to my thesis supervisor prof. Sverre Hendseth for his great guidance, motivation, patience and valuable recommendations throughout my thesis. His office was always open whenever I encountered whatsoever problem. Honestly, this thesis would not have been like this without his unlimited advices, encouragement and support.

Chapter 1

Introduction

1. Motivation

Procedural Generation is a method of creating data algorithmically with limited or no human contribution, and it's commonly used by developers to create content very quickly. It is a powerful tool in the first place. However, it becomes a killer tool that completely ruins your design if it is not used wisely. One thing you should always keep in mind before using procedural generation that it is not easily controllable. That is, the procedural content can be difficult to be manages, and you can easily lose control.

2. Scheduling is a critical duty

Building a successful project is a challenging process. To be successful, you have to plan duly. Otherwise, you will most properly run over budget, and you project eventually fails. Using a procedural content generation in any projects is a risky choice. Therefore, you have to know exactly when to introduce such technique into your project. Usually, the decision to use PCG or not is taken early in the project`s lifecycle. In most cases, this crucial decision occurs during the design phase **[1]**. Undoubtedly, time is vital in any project. That being said, the next important task after choosing PCG is to carefully estimate how much time you need to build the content of your project using PCG. In many projects, this time is underestimated. Be careful and do not do that.

To be safe, you should always make your estimation flexible as far as possible. That flexibility is a necessity because:

- The one constant in software development is the change **[2].** The direction of your project could be drastically change at any moment. That implies different undesired consequences. For example, the code you have written becomes invalid under the new circumstances. Therefore, it should be drafted and undrafted many times.
- Even in a very stable project where the changes are minimal, some adjustments and tweaking after releasing the project is inevitable.

In short, using PCG requires tight development from the very beginning of any project.

3. Implementation and coding

Some notes on the fly about the code and the figures:

- Demonstrating how an algorithm works step by step is good. Writing code additionally to implement the algorithm is even better. Therefore, I have decided to write the code for every single algorithm in my thesis. In my humble opinion, this will mainly help to clarify how the different algorithms behave in action.
- More than 20 algorithms will be demonstrated, implemented and tested throughout this thesis. The majority of those algorithms have implemented from scratch and coded line by line. In some algorithms, however, I got stuck at some point. I tried every possible way I can to go out of this impasse. When I failed, I asked for help on the internet. Even though the final code I wrote is completely unique, I still have to credit every single help I got from others **[3] [4]**.
- Programming is my favorite hobby, and I fell in love with it more than ten years ago. My programming experience is mainly in both Java and JavaScript. I have started using Java professionally since 2008, and JavaScript since 2012. However, I was very hesitant about which one to use in this thesis to implement the various algorithms. Eventually, I opted JavaScript for two main reasons:
  1- Its simplicity over Java. That is, the amount of code needed in Java to implement any of the algorithms is at least doubled.
  2- The available libraries for drawing and animation functionality are superior in JavaScript nowadays.
- All the code in this thesis works perfectly. It has been tested many times, and it is absolutely error-free. In order to run this code properly in your own machine, you have to do one of the following things:
  1- Use HTML file in order to run the code in your browser.
  2- Use JavaScript open source server environment like Node.js to run the code using command prompt.
- Last but not least, there are over 70 different figures in this thesis. All of them, without any exception, are a direct result of running the code that I wrote to implement each algorithm. No copy-paste figures at all from any resource. On my part, I do not like off-the-shelf stuff. I bet you agree with me!

One last remark: Keep in mind before reading this thesis that there is **no clear distinction between the theory part and the results part** as usual in any technical report. There are results and discussion in every single chapter starting from chapter 2 until chapter 10.

Chapter 2

Theory

1. The uselessness / usefulness of PCG

1.1 PCG Disadvantages

PCG is simply a two-edge sword. It can be very useful in many cases, but it can be a bad choice sometimes. It is absolutely not a panacea, and in particular situations, avoiding it is considered to be the right choice. If you decide to integrate it into your project, then you should realize all the hazards PCG might bring into your project.

1.1.1 Quality assurance tests

Quality assurance (QA) is a crucial factor in any project. It is, however, the main reason to avoid using PCG in dome projects as well. Using PCG makes the behavior of your project unpredictable somehow. As a sequence, your project is not guaranteed to work exactly as designed all the time. As mentioned earlier, the generated content is not fully controllable. Moreover, it is impractical to test each part of the generated content. That is infeasible.
Some risky situations are unavoidable with PCG. Your generator may hide some tiny bugs that never appear before the project gets released. To handle such situations, you need testers. But it is also possible to introduce extra constraints in order to augment the controllability of the generator **[1]**. However, the more constraints you add to your generator, the less variety the generator will bring to your project. The chance to get repetitive content increases drastically with over-constrained levels.

1.1.2 More Demands on Hardware

At the end of the day, procedural generation is nothing more than an algorithm running on your computer to generate a desired content somehow or other. The more complicated your generator is, the more advanced hardware your computer should have to run those algorithms properly. Likewise, the more advanced hardware installed in your machine, the more money you have to pay. It is actually a sort of trade-off. You have always to ask yourself the following question: Does it worth to spend extra money in order to enhance the performance of your generator?

1.1.3 Unpredictable Time

At the first glance PCG seems like a huge time-saver when it is compared to a handcrafted content. But sometimes this is completely deceptive.
There is no guarantee that the estimated time to build the generator is going to be accurate. Practically if the amount of existing code to count on when you design the generator is not enough, it will take much more time than what you expect to have your generator done **[1].**
Another important factor that affects the time hugely is the amount of tweaking and improvement your generator needs after completion it. This process may occur over and over again depending on the changes in the requirements of your project. That being said, the estimated time in a handcrafted process is much more predictable.
Therefore, do not rush into making a decision, and reconsider how much PCG you will involve in your project. Such decision is very important specifically if both time and money are critical for your project. Any careless decision will eventually put you face to face with the hard fact; the failure of your project.

1.1.4 Change is not easy

Usually you choose an algorithm or a technique to be the core of your generator. If you are not satisfied with the generated content after completion the generator, then changing the behavior of your generator is an intricate task. Doing that demand a change in the used algorithms, and that is impractical if you build a huge generator.

1.1.5 Random output

Occasionally, things do work out the way you planned them. It would be unacceptable by any means that you invest many resources to build a well-functioning generator, but eventually the output of the generator is just random contents. What a frustration that would be !
Randomness in this context means that the generated content is boring and full of repetitive content that does not make a good experience.

1.2. PCG Advantages

After reading all the above drawbacks of using PCG, you may wonder why you should use PCG after all? Do not rush to judgment on that. Good news is coming. In fact, there are several obvious beneficial reasons to use PCG, and to prefer it over the traditional, old-style, handcrafted content. Moreover, there exist some fun reasons to utilize PCG in your project.

1.2.1 A Great Time-saver (most of the time)

This is probably the most tempting reason to use PCG. Time is crucial, valuable and vital in real-time projects. With PCG you can create enormous amounts of content far faster than a pure manual-created content. But that not always the case. Although this merit is most often very useful, time-saving in particular situations is merely a myth. Particularly, if your generator needs a lot of tweaking after completion. Consequently, you are wasting time instead of saving it.

1.2.2 Provide a Unique Experience Each Time

A great feature of PCG is its ability to give each user a completely unique experience. That is due to the unlimited space of varied content it can produce. A hand-crafted content will never be able to provide this infinite variety in the generated content.

1.2.3 Reusability of Existing Code

Once you build a generator, you can whenever it makes sense reuse it across completely different applications. If you decide to reuse an existing code, then all you need is some tweaking to the existing variables and attributes to get a content that suits properly another project. Thanks to the modular design of the generators, you can easily swap modules between different applications **[1]**.

When you implement big projects sequentially, code reusability is absolutely precious.

1.2.4 Mirror our Humanity in Different Ways

As a human being, we all believe in the legendary quote: “nothing comes from nothing”. PCG is no exception at all. Although a procedural content is produced algorithmically using computers, it depends originally on a human- designed content, even a tiny one. What the generator eventually creates is nothing more than a reflection of the creator creativity. In fact, the generated content is somehow or other a mirror that tells us about ourselves.

1.2.5 Enforce a Predefined Set of Rules

When you intend to build a generator, you can determine a set of rules that must be enforced across your generator. These rules guarantee connectivity between elements through the generated content. If your code is clean, bug-free and well-structured, then the chance is high that your predefined rules get applied properly in your generator. The main purpose of these rules is to let you direct the output of your generator the way you want.

1.2.6 Generate an Infinite Content

In fact, one of the most reasons to make PCG impressive is its ability to create an infinite output somehow. If you ask the most skillful designer to create the best content he can come up with, at some point there is certainly a limit to what he can do. PCG can, however, generate a content that is simply beyond the scope of our imagination.

1.2.7 Produce an Unpredictable and Enjoyable Output

This feature is interesting because even the most-experienced designers will not be able to anticipate what exactly the generator will produce each time it runs. This might seem a drawback, particularly from a QA perspective. But for you, as the creator of the generator, it gives you a lot of excitement when you experiment your generator and enjoy the unpredictable content it generates.

1.2.8 Provide Content that Feel Real to us

One of the most prominent weaknesses of hand-crafted content is its lack of diversity. Usually such kind of content ends up with a lot of repetition one way or another. PCG, on the other hand, generates content that is far more varied and diverse. Procedural techniques are able to produce all kinds of lively objects such as weather systems, civilizations, floods, populations, and much more. What makes such contents great is that they appear natural to us. That is due to the fact many procedural techniques and approaches use patterns that are to a great extent analogous to the existing patterns in reality **[1].** Later on, a lot of these techniques and approaches like random walks, L-Systems, cellular automata and others will be discussed in detail in this thesis.

1.2.9 Simplify the Concept of Infinity

Infinity is a weird word. We hear it a lot, but perceive it a bit. It is an abstract concept that is difficult to be grasped by humans. As mentioned above, procedural generators have the capability to come up with infinite content. Dealing with such content is actually a kind of pleasure. A good example would be the procedurally generated games which have an infinite space such as Minecraft **[5]**, and No Man´s Sky **[6].** The enjoyment you feel when you play these games comes mainly from exploring infinite worlds. PCG gives you a unique opportunity to experience and look closely at an infinite content.

Chapter 3

Random Numbers

1. Pseudorandom Number Generators (PRNGs)

Generating random numbers is actually pretty hard. The whole concept of how computers work is that they take an input, and apply some algorithm to it and provide the output of that process. There is no algorithm that will produce a truly random response because if we use the same input and the same algorithm again, we will eventually get the same output. The best we can do is to generate what we call pseudorandom numbers. This is where the connection between the input and the output is sufficiently unpredictable that it seems random **[7]**. However, at some level of inspection, that randomness will break down and a pattern will eventually appear. But there are some pretty solid algorithms for pseudorandom number generators, aka PRNGs. PRNGs are deterministic algorithms **[8]**. This is to say that you start with a seed value, which is used to calculate a new random number. This new random number becomes a seed for the next number. The new random number becomes again a new seed for the next number and so on (figure 1). Every time you start with a specific seed, you will get the same sequence of random numbers. Using different starting seed gives you different random sequence **[7]**. In other words, the pseudorandom number generator will always remember its output for each specific seed value. That is, whenever you provide the generator with a specific seed value, it will always generate exactly the same sequence of random values and with the same exact order. To exemplify this concept, let us say that you choose the initial seed to be 8, and the generator gives you the random sequence 23,45,2,99,156 when you ask about the first five values from this sequence. Whenever you provide the generator again with the value 8 as starting seed for the generator, it will return the same previous sequence 23,45,2,99,156 and exactly in the same order. This also holds for all the remaining numbers in the sequence when the initial seed is 8. Once you change the initial seed value, the generator returns a different series of random numbers. This can be useful in applications where you want an unpredictable sequence, but you want to be able to repeat it whenever you need it again.

There are two important characteristics that should be taken into account when you analyze any PRNG **[9]**:

1- <u>Period:</u> which represents how many random numbers the generator can generate before it starts repeating itself. Remember that because each number is generated from a seed, which is the previous random number, once you hit a repeat you will get stuck in a loop repeating the same series of numbers over and over.

2- <u>Distribution:</u> which indicates how random are the generated numbers. If the number the generator generates tend to be more high than low, low than high, clump around certain values, avoid other values or form some kind of pattern, then the usefulness of the generators is minimized.

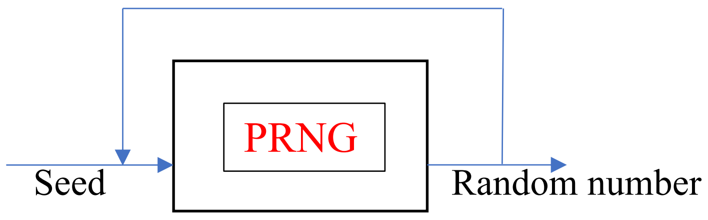


Figure 1 – How PRNGs work

A final concern is how easy to predict the sequence. This is extremely important for security applications that use PRNGs. There are many PRNGs algorithms available. In this section, I will start by demonstrating two simple PRNGS. The first one is *Middle-Square method[MSM]* and the second one is *Linear Congruential generator[LCG]*. It is worth to mention beforehand that these two algorithms are not the best out there, but because they are so simple, they are good to use for demonstration purposes. After that, another PRNG called *Mersenne Twister[MT]* will be demonstrated as well.

1.1 Middle Square Method[MSM]

This algorithm is the simplest PRNGs, but practically it is inefficient. That it due to its very short period (we will experiment and prove that shortly). This algorithm was invented by John Van Neumann in 1949 **[10]**, and according to him it can be described by the following steps:

1- A seed value consists of a certain number of digits is used as an input (let us say the seed consists of only 4-digits).

2- The seed value gets squared.

3- At most the result will contain 8-digits. If it is less than that, the left digits are substituted by 0s to make sure that the result always includes 8-digits.

4- The middle four digits represent the new seed value.

5- Check the termination condition, and go to step 6 if it holds. Otherwise go to step 2.

6- Terminate the algorithm.

Implementation

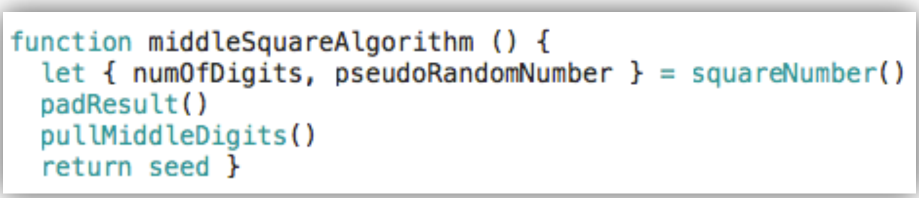
In practice, you do not need to write all this code (hard-coding) in order to use this algorithm. What you can do, instead, is to use a library in JavaScript that contains a predefined function performs all this stuff in just a single line. That way is much more efficient in real world scenario. However, I have intentionally written this code here just for demonstration purposes. My aim is actually twofold :

1- Examine how this method works if we ask it to generate a set of random numbers (10 numbers for instance). For the sake of simplicity, assume that the seed consists of only 4-digits. We will run the algorithm using different seed values.

2- Demonstrate how the period changes when the seed varies. In other words, how many numbers the generator can generate before it starts generating duplicates. Using 4-digit number, we know that at most it is going to be 9999. Let us test this method for different initial seed values, and check the period for each value.

Code

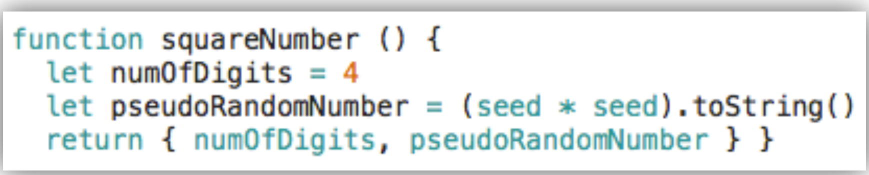
The main function that generates a pseudorandom number is:



```
function middleSquareAlgorithm () {
  let { numOfDigits, pseudoRandomNumber } = squareNumber()
  padResult()
  pullMiddleDigits()
  return seed }
```

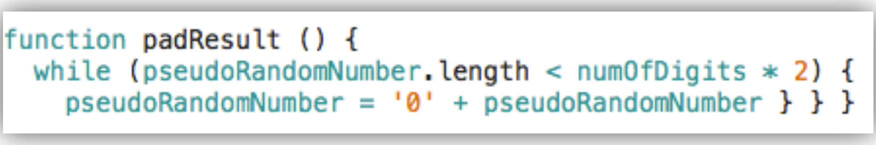
This function calls three different function to complete its task (steps 2 & 3 & 4).

Step 2



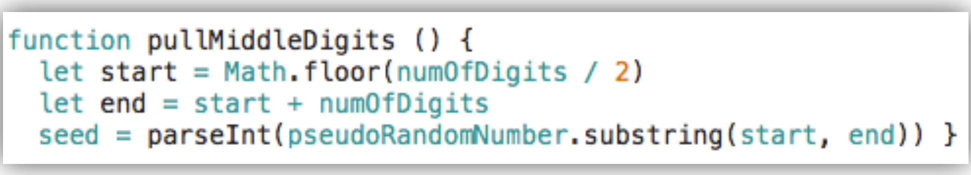
```
function squareNumber () {
  let numOfDigits = 4
  let pseudoRandomNumber = (seed * seed).toString()
  return { numOfDigits, pseudoRandomNumber } }
```

Step 3



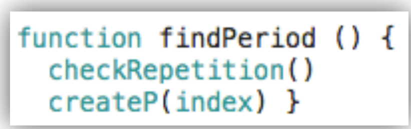
```
function padResult () {
  while (pseudoRandomNumber.length < numOfDigits * 2) {
    pseudoRandomNumber = '0' + pseudoRandomNumber } } }
```

Step 4



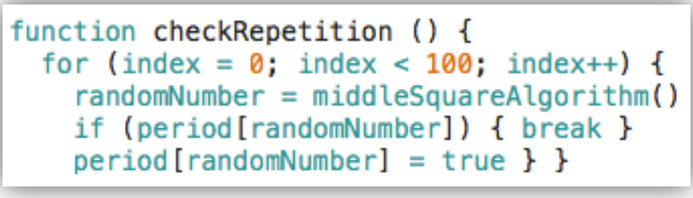
```
function pullMiddleDigits () {
  let start = Math.floor(numOfDigits / 2)
  let end = start + numOfDigits
  seed = parseInt(pseudoRandomNumber.substring(start, end)) }
```

To calculate the period, another function should be used:



```
function findPeriod () {
  checkRepetition()
  createP(index) }
```

This function calls another function:

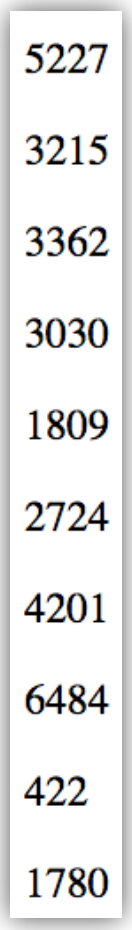


```
function checkRepetition () {
  for (index = 0; index < 100; index++) {
    randomNumber = middleSquareAlgorithm()
    if (period[randomNumber]) { break }
    period[randomNumber] = true } }
```

Let us test this algorithm using the following seed values (table 1):

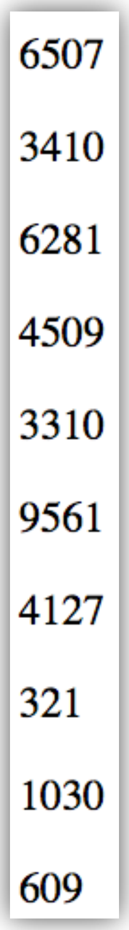
Initial seed	Generated sequence	Period
1234	Figure 2.1	56
2766	Figure 2.2	8
9834	Figure 2.3	15
2369	Figure 2.4	58
7777	Figure 2.5	43
7865	Figure 2.6	9
4321	Figure 2.7	60
1111	Figure 2.8	43

Table 1 Different seed values for testing MSM



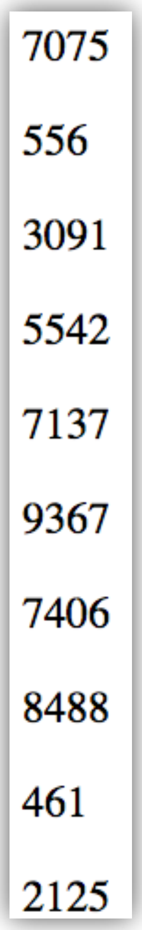
```
5227
3215
3362
3030
1809
2724
4201
6484
422
1780
```

Figure 2.1 MSM-1-



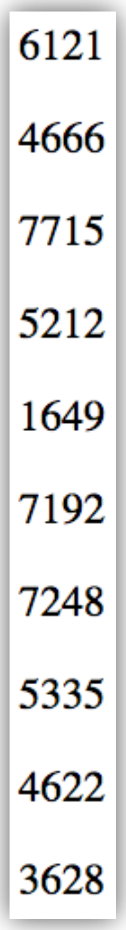
```
6507
3410
6281
4509
3310
9561
4127
321
1030
609
```

Figure 2.2 MSM-2-



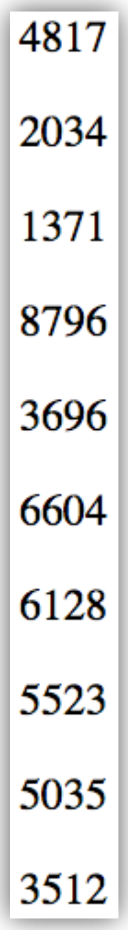
```
7075
556
3091
5542
7137
9367
7406
8488
461
2125
```

Figure 2.3 MSM-3-



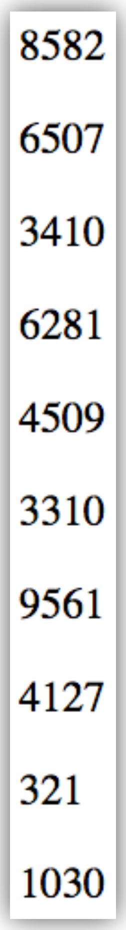
```
6121
4666
7715
5212
1649
7192
7248
5335
4622
3628
```

Figure 2.4 MSM-4-



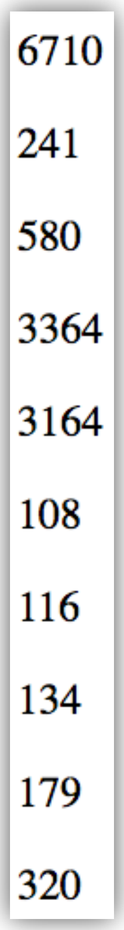
```
4817
2034
1371
8796
3696
6604
6128
5523
5035
3512
```

Figure 2.5 MSM-5-



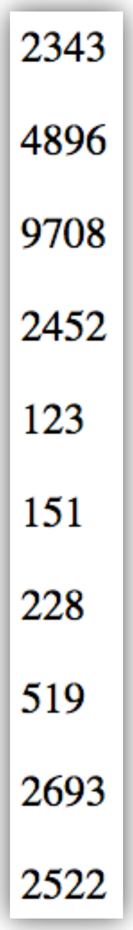
```
8582
6507
3410
6281
4509
3310
9561
4127
321
1030
```

Figure 2.6 MSM-6-



```
6710
241
580
3364
3164
108
116
134
179
320
```

Figure 2.7 MSM-7-



```
2343
4896
9708
2452
123
151
228
519
2693
2522
```

Figure 2.8 MSM-8-

Discussion

Looking back at the periods the generator generated (table 1), it is very obvious that the generator was unable to create varied pseudorandom values for a long time. That is, it repeated the sequence after short number of iterations. At best, it could generate 60 different seeds (in our test). That considers to be a huge drawback of using this method, and therefore we can say that this PRNG is inefficient if our system needs a lot of different seed values. To wrap up this discussion, this method is simple and fast, and this what makes it tempting. But remember its unacceptable deficiency, and the flaw it can cause in your system.

Think twice before making your decision! Remorse will get you nowhere!

Let us have a look at another, more efficient, PRNG called Linear Congruential Generator.

1.2. Linear Congruential Generator(LCG)

This algorithm has been used as the default PRNG in some languages and systems **[11]**. For my part, I have used it in Java (java.util.Random). It is very simple to implement, and it is pretty fast. Its speed and simplicity make it far superior to the middle-squares method. But do not think that it is an optimal algorithm. In fact, it is far from perfect. It is completely unsuitable for any type of security uses **[12]**.

In addition to a seed value, this algorithm needs three other values in order to work properly. A multiplier, an increment and a modulus. These variables are usually called a, c, m. The algorithm works as follows:

- Multiplies the current seed by the multiplier a.
- Add the increment c to the previous multiplication.
- Mod the result by the modulator m.

That being said, the formula of this generator can be written as proposed in **[13]**:

Seed = (a* seed + c) % m

You may wonder about the values that should be used for those three parameters (a, c, m). These values should be selected carefully. There are certain criteria that should be followed in order to make good choices here. Poor choices turn the algorithm to be a useless PRNG, and good choices, on the other hand, turns it to be efficient **[13]**.

Implementation

As mentioned above, there are pre-tested values for those parameters that guarantee good results. Let us try two sets of those values and see what we are going to get for each one. Each set of values affects the generated sequence the generator generates. You can certainly play around with all those values, and see how the generator behaves for different sets of values.

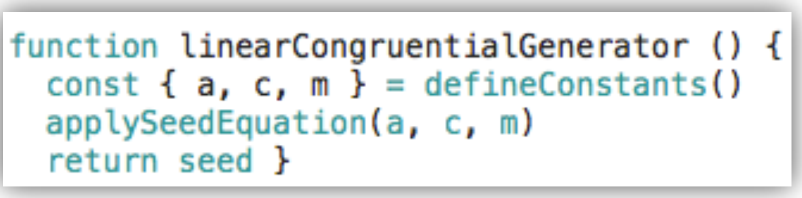
The following values will be tested:

Source	Modulus (m)	Multiplier (a)	Increment (c)
Numerical Recipes **[14]**	2^32	1664525	1013904223
Java.util.Random	2^48	25214903917	11

Table 2 Two sets of well-known values for LCG parameters

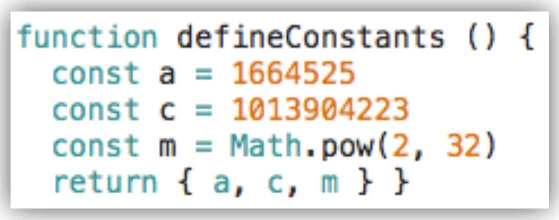
Code

The main function to generate the seed value is:



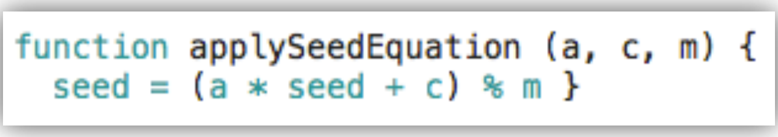
```
function linearCongruentialGenerator () {
  const { a, c, m } = defineConstants()
  applySeedEquation(a, c, m)
  return seed }
```

This function calls another two functions. The first one is just to define the constant that are listed in table 2 :



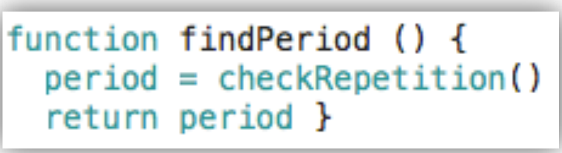
```
function defineConstants () {
  const a = 1664525
  const c = 1013904223
  const m = Math.pow(2, 32)
  return { a, c, m } }
```

The other function applies the equation mentioned above in order to calculate the seed value:



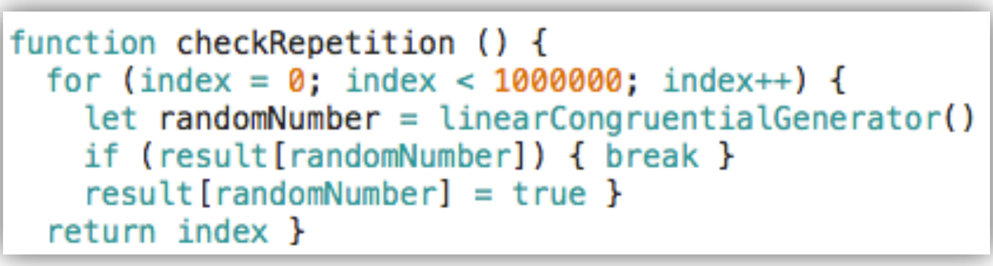
```
function applySeedEquation (a, c, m) {
  seed = (a * seed + c) % m }
```

To calculate the period, this function is used:



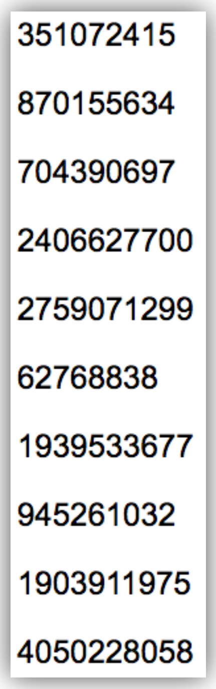
```
function findPeriod () {
  period = checkRepetition()
  return period }
```

This function calls another function:



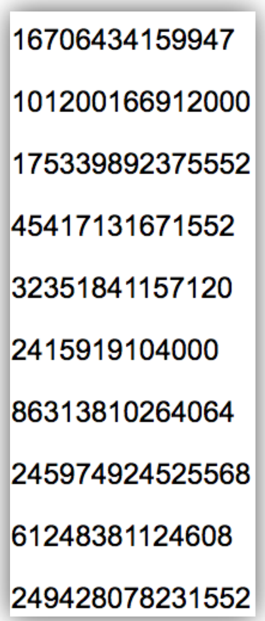
```
function checkRepetition () {
  for (index = 0; index < 1000000; index++) {
    let randomNumber = linearCongruentialGenerator()
    if (result[randomNumber]) { break }
    result[randomNumber] = true }
  return index }
```

Running this algorithm, it generates the following sequence of numbers:



```
351072415
870155634
704390697
2406627700
2759071299
62768838
1939533677
945261032
1903911975
4050228058
```

Figure 3.1 Generated sequence using Numerical Recipes values



```
16706434159947
101200166912000
175339892375552
45417131671552
32351841157120
2415919104000
86313810264064
245974924525568
61248381124608
249428078231552
```

Figure 3.2 Generated sequence using Java.util.Random values

Discussion

It is worth to mention that I ran this algorithm for one million iteration without getting any duplicates. That is far better than the middle-square method where the duplicates started to appear even before 100 iteration. In fact, this is a huge improvement and nobody can deny that. But wait! We still have a long way to go.

1.3. Mersenne Twister[MT]

This algorithm was found in 1997 by both Makoto Matsumoto and Takuji Nishimura **[15]**. It is one of the most popular high-quality PRNGs. It is the default generator in many languages. Even if it is not the default in your favorite programming language, you can easily find libraries that implement it. To test the quality of PRNGs, there are well-known tests called the diehard tests **[16]**. This algorithm passes all of them. It is far superior to the previously discussed PRNGs, but remember the one constant trade-off in any algorithm. The more quality the algorithm is, the more complex it becomes as well. Accordingly, this algorithm uses more memory and is slower than the previous PRNGs.

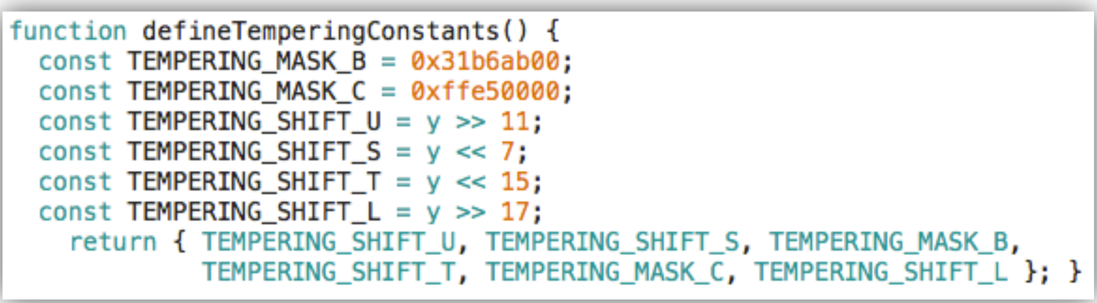
The way this algorithm works is somewhat complicated, and it contains complex operations and equations. Therefore, I have avoided writing those steps here. However, I have implemented every single step in my code (see implementation underneath). If you are curious to know how this algorithm works behind the scene, please have a look at **[15]**.

Implementation

My implementation is completely built on the steps (five steps), and constants values that are presented at **[15]**. More precisely, my code is an implementation to what is called MT 11213B as entitled in **[15].**

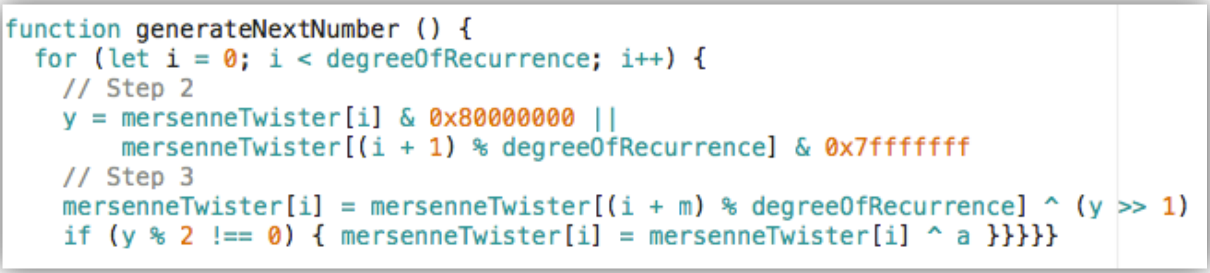
Code

First of all, we define 6-constants that will be used later on in this algorithm. The name of these constants seems somewhat weird. They are called *tempering parameters* **[17]**.



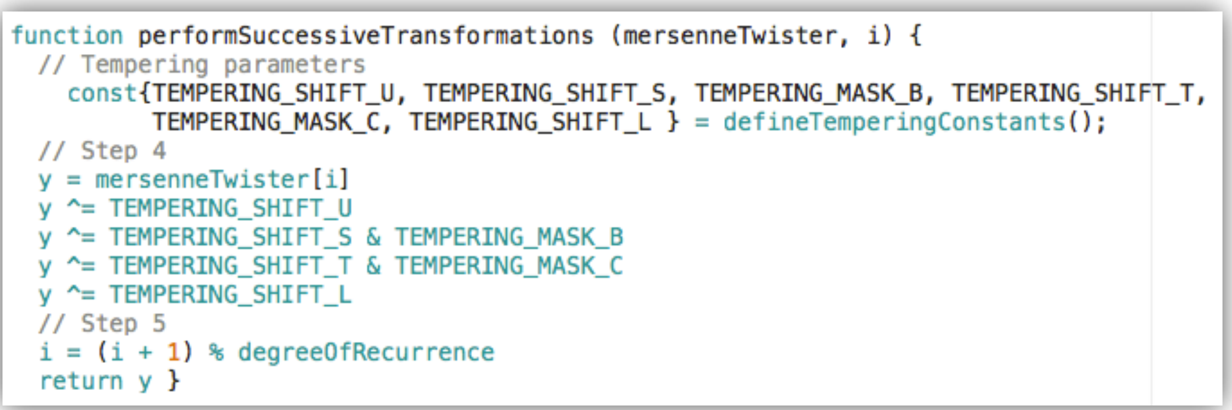
```
function defineTemperingConstants() {
  const TEMPERING_MASK_B = 0x31b6ab00;
  const TEMPERING_MASK_C = 0xffe50000;
  const TEMPERING_SHIFT_U = y >> 11;
  const TEMPERING_SHIFT_S = y << 7;
  const TEMPERING_SHIFT_T = y << 15;
  const TEMPERING_SHIFT_L = y >> 17;
    return { TEMPERING_SHIFT_U, TEMPERING_SHIFT_S, TEMPERING_MASK_B,
             TEMPERING_SHIFT_T, TEMPERING_MASK_C, TEMPERING_SHIFT_L }; }
```

Steps 2 & 3 are implemented using the following function:



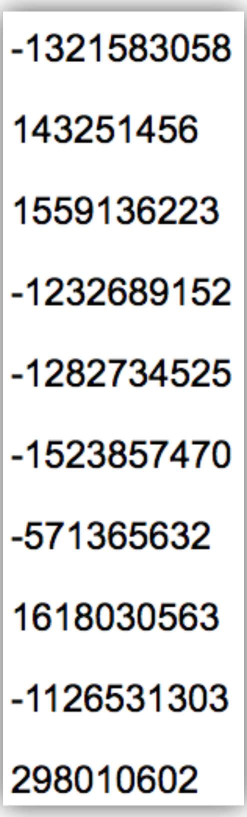
```
function generateNextNumber () {
  for (let i = 0; i < degreeOfRecurrence; i++) {
    // Step 2
    y = mersenneTwister[i] & 0x80000000 ||
        mersenneTwister[(i + 1) % degreeOfRecurrence] & 0x7fffffff
    // Step 3
    mersenneTwister[i] = mersenneTwister[(i + m) % degreeOfRecurrence] ^ (y >> 1)
    if (y % 2 !== 0) { mersenneTwister[i] = mersenneTwister[i] ^ a }}}}}
```

And this function is for steps 4 & 5:



```
function performSuccessiveTransformations (mersenneTwister, i) {
  // Tempering parameters
    const{TEMPERING_SHIFT_U, TEMPERING_SHIFT_S, TEMPERING_MASK_B, TEMPERING_SHIFT_T,
          TEMPERING_MASK_C, TEMPERING_SHIFT_L } = defineTemperingConstants();
  // Step 4
  y = mersenneTwister[i]
  y ^= TEMPERING_SHIFT_U
  y ^= TEMPERING_SHIFT_S & TEMPERING_MASK_B
  y ^= TEMPERING_SHIFT_T & TEMPERING_MASK_C
  y ^= TEMPERING_SHIFT_L
  // Step 5
  i = (i + 1) % degreeOfRecurrence
  return y }
```

The complete code can be found in the appendix.
Running the generator will produce the following sequence:



```
-1321583058
143251456
1559136223
-1232689152
-1282734525
-1523857470
-571365632
1618030563
-1126531303
298010602
```

Figure 4 Generated sequence using Mersenne Twister 11213B

Discussion

Testing this algorithm showed that it has extremely large period even better than LCG. But it is much more complicated. From the above experiments, it is obvious that MT generated both positive and negative numbers, while both MSM and LCG generated only positive numbers.

2. Normal Probability Distribution

It was discovered by Carl Friedrich Gauss in 1809 **[18]**. In many cases, generating a normal distribution of numbers instead of linear one is very useful. This way, we end up with something resembles a bell. Therefore, it is commonly called bell curve. This curve can be useful to generate many types of content. It approximates the variety of properties for many things in the real-world. The majority of these properties lie around the middle values, while the highest and lowest values occur rarely. There are two main terms that describes the normal distribution. The first one is the mean, which represents the value at the peak of the curve. The second term is the standard deviation which indicates how "wide" the curve is **[1]**.

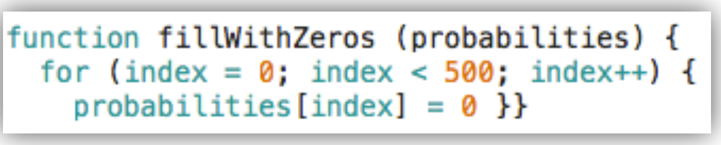
<u>Implementation</u>

To demonstrate this concept, I have fulfilled four different experiments (table 3). At the first one, the normal distribution has been drawn for the average of two random numbers. To improve the resulted curve a little bit, another random number has been added. The resulted curve was not satisfactory. Therefore, two more experiments were done. The first one using 10 numbers, and lastly with 100 numbers. A much better curve has eventually been obtained

<u>Code</u>

The way I implement this algorithm can be described by the following steps:

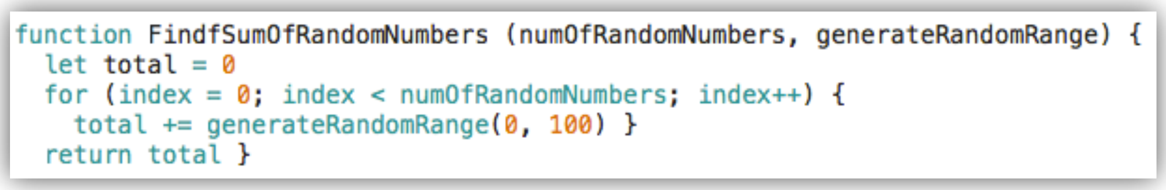
1. Create an array and fill it initially with zeros.



```
function fillWithZeros (probabilities) {
  for (index = 0; index < 500; index++) {
    probabilities[index] = 0 }}
```

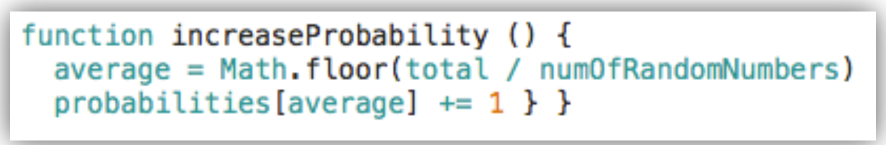
Each placeholder in the array represents the probability of the corresponding index.

2. Generate a set of random numbers between 0 and 100, and add them together.



```
function FindfSumOfRandomNumbers (numOfRandomNumbers, generateRandomRange) {
  let total = 0
  for (index = 0; index < numOfRandomNumbers; index++) {
    total += generateRandomRange(0, 100) }
  return total }
```

3. Calculate the average of the generated numbers. Then increase by one the corresponding placeholder in the array.

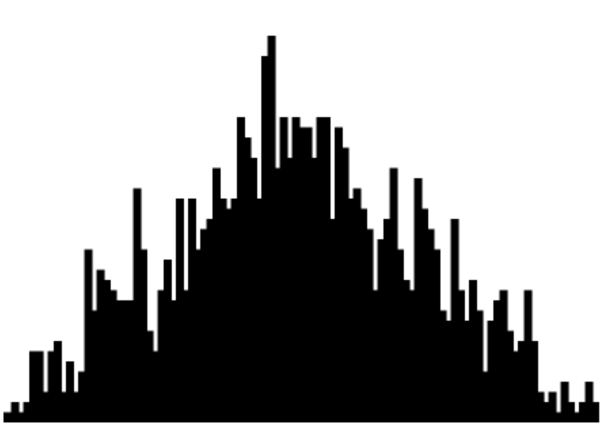


```
function increaseProbability () {
  average = Math.floor(total / numOfRandomNumbers)
  probabilities[average] += 1 } }
```

The algorithm was tested using the following different values (table 3).

Number of random numbers to be averaged	Result
2	Figure 5.1
3	Figure 5.2
10	Figure 5.3
100	Figure 5.4

Table 3 Tested values for creating different Bell curves



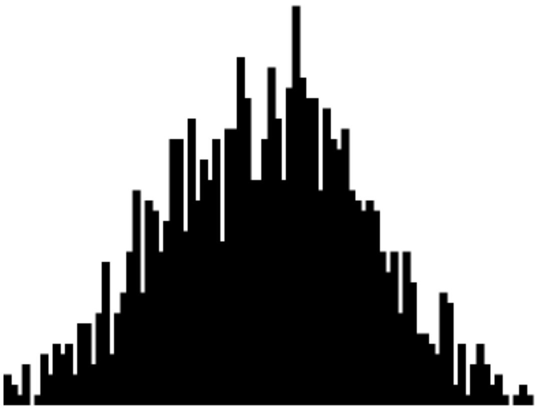
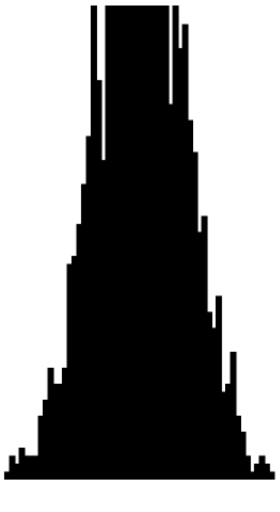


Figure 5.1 Bell curve for 2-random numbers **Figure 5.2** Bell curve for 3-random numbers



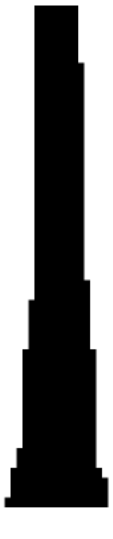


Figure 5.3 Bell curve for10-random numbers **Figure 5.4** Bell curve for100-random numbers

<u>Discussion</u>

It is very obvious from the previous experiments that for only two random numbers (figure 5.1), the curve is somewhat chaotic. To get a curve similar to bell curve, we have to average more random numbers. Hopefully you can see that we got a better bell curve shape as we increased the random numbers. In a nutshell, the more random numbers you use, the more like-bell curve shape you get.

3. Weighted Random

The concept of weighted random looks familiar and simple. It introduced by Fisher in 1935 and Rao 1965 **[19]**. It is very beneficial in a plenty of various applications. A weighted random function is a way to define several random outcomes, and then choose one of those randomly in a way where some outcomes are more likely to occur than others. This can be very useful in a game, for instance, where at a certain point of the game a monster will appear. Let us say that your game has four different types of monsters. Those monsters are troll, zombie, vampire and ghost. Your purpose is to keep the process random, but mostly you want trolls with the occasional zombie or vampire and every once in a long while a rare ghost will appear. To do that, you have to use a weighted random function where you prioritize some monsters over others. In this example, you might give the trolls 50%, both the zombie and the vampire 22.5% and the ghost only 5%. Weighted random can be used whenever you have some cases, and you want to choose among them in a controllable manner, but without losing the sense of randomness. In other words, you give higher priority to what you think more important, and lower priority to what you think less important.

Implementation

There are unlimited number of applications where this concept can be applied. For the sake of simplicity, and to keep my code as short as possible I have chosen an application called *coin flipper* and implemented it to demonstrate how this concept can be used in practice. The coin flipper system has only two cases. The idea of this application is very simple. You have a coin, and you want to flip it a specific number of times (let us say 100 times).

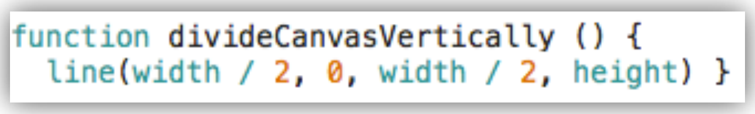
Our space (the canvas) will be divided into two separate regions. The right side represent number of heads, while the left side represents the number of tails.

Experiment 1

Let us assume firstly that both cases (heads and tails in our example) are equally important. This means that both of them have the same priority. In other words, they have the same chance to be chosen. Since the process is still random, and not deterministic, you will not necessarily get 50 heads and 50 tails if you flip the coin 100 times. What you will actually get is roughly equal amount of coins on each side.

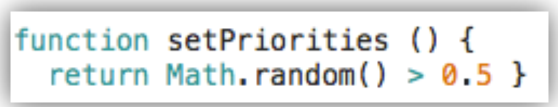
<u>Code</u>

The code is straightforward. First of all, the space (canvas in our implementation) is divided vertically into two equal parts in order to display the number of tails on one side, and the number of heads on the other side.



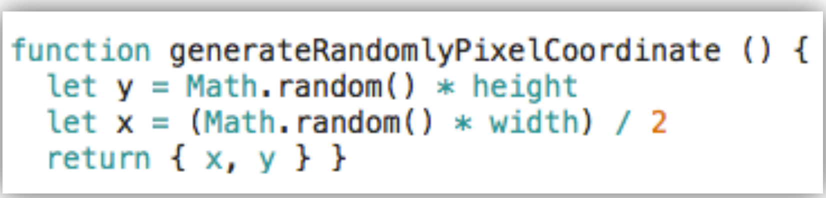
```
function divideCanvasVertically () {
  line(width / 2, 0, width / 2, height) }
```

After that, the priority for either heads or tails should be determined.



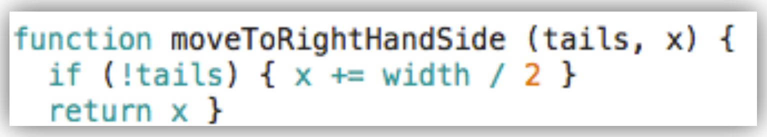
```
function setPriorities () {
  return Math.random() > 0.5 }
```

This way we set the priority for both heads and tails to be roughly 0.5.
Then, we generate randomly the coordinate (x , y) of a pixel (point) inside the left-hand side of the canvas where the tails will be drawn.



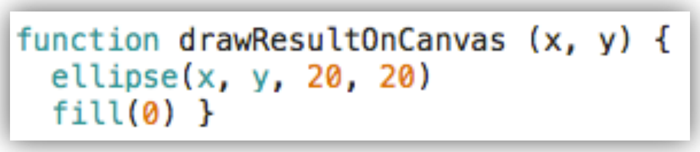
```
function generateRandomlyPixelCoordinate () {
  let y = Math.random() * height
  let x = (Math.random() * width) / 2
  return { x, y } }
```

We check then to see if the current result is a tail, we draw a circle inside the left-hand side. Otherwise, the circle is drawn inside the right-hand side.
So, whenever it is not a tail, we have to move to the right side to draw a head.



```
function moveToRightHandSide (tails, x) {
  if (!tails) { x += width / 2 }
  return x }
```

Lastly, no matter whether we are on the left-side or on the right-side, at the end of the day, a circle must be drawn.



```
function drawResultOnCanvas (x, y) {
  ellipse(x, y, 20, 20)
  fill(0) }
```

Running this code, we get the following distribution:

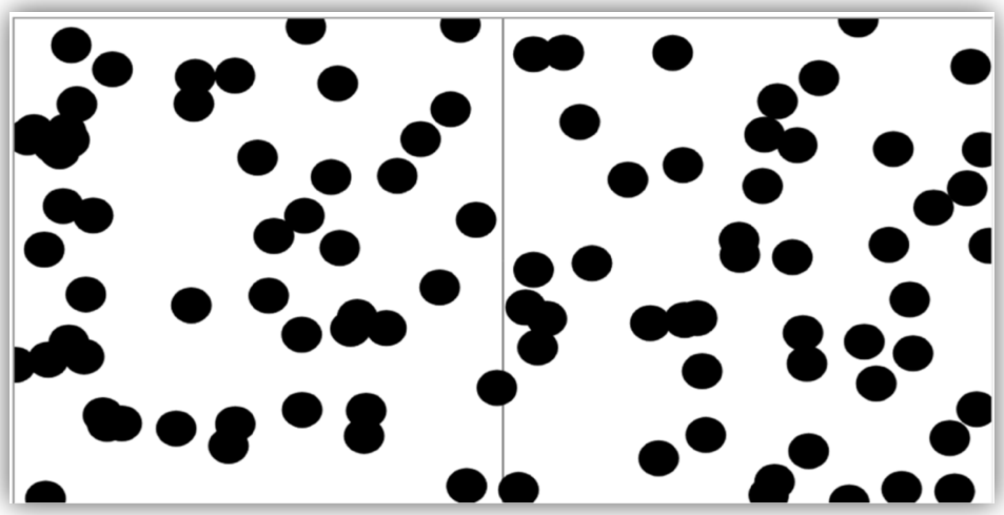


Figure 6.1 Weighted random for two equally-important cases

Experiment 2

Let us assume that the tails are much more important than the heads. Therefore, you prioritize it by setting its chance of appearing to 80%, and only 20% to the heads. Running the code with the new values will give the following result:

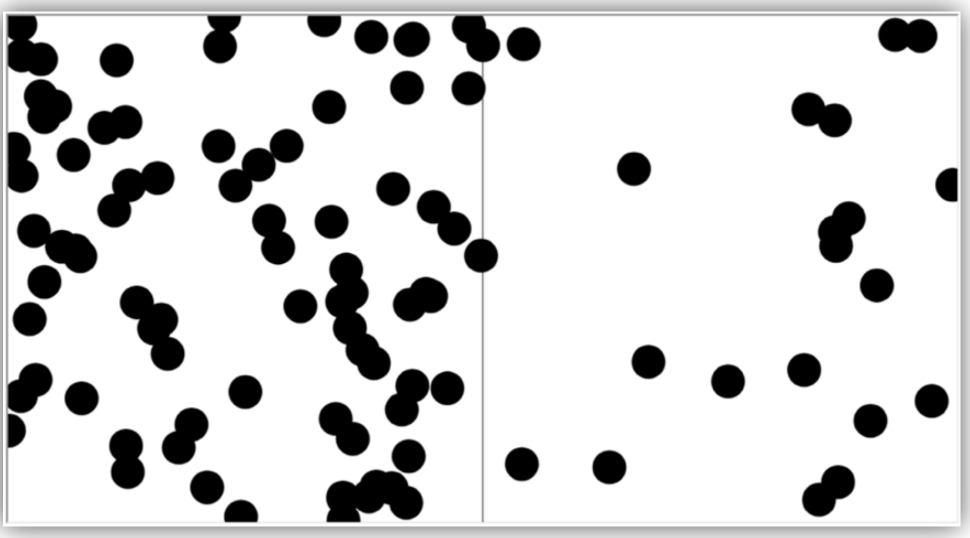


Figure 6.2 Weighted random for two cases with different importance

Discussion

In the previous examples, we experimented only two cases (heads/tails), but in fact this principle holds no matter how many cases exist in your application. The idea is still completely valid, and you can easily expand its uses to meet the requirements of your system. How utilitarian weighted random is!

Chapter 4

Perlin Noise

Thanks to Ken Perlin for his tremendous contribution when he discovered Perlin noise in 1983 **[20]**. At its core, Perlin noise produces smooth sequence of pseudo-random numbers. Smoothness, in this context, means that the random number you might pick at any point in time is related to the random number you might pick later on or a moment before. On the other hand, in pure random algorithms, those numbers are not related at all. That is the core difference. Random noise is just a set of random values that has no relationship to each other. Perlin noise, however, also provides randomness, but the values are close to each other and have a relationship, which can be seen as a smooth transition between the points. Therefore, we mentioned above that Perlin noise provide a smooth randomness. When generating textures and terrains from scratch, you normally want to apply some kind of randomness. However, using a completely random numbers to generate a texture makes it looks chaotic and extremely random. Instead, we use a noise function such as Perlin noise function where the values are not totally random, but they have some relationship to each other. That allows changes to occur gradually. This kind of noise is referred to usually as pseudo random noise. Perlin noise has become a common tool that is used to generate heightmaps in two/three dimensional spaces, although it can be used for an arbitrary number of dimensions. Typically, textures consist of several layers of noise. Each layer has its own frequency/amplitude values. Usually, the amplitude is high and the frequency is low at the first layer. Additional layers have lower amplitude, but higher frequency. These layers have two different impacts. A small effect on the overall structural height due to its lower amplitude. An important effect by adding interesting details to the existing valleys and peaks **[1]**

Note:

Perlin noise function will be used in chapter 10 to build an elegant infinite terrain.

Implementation:

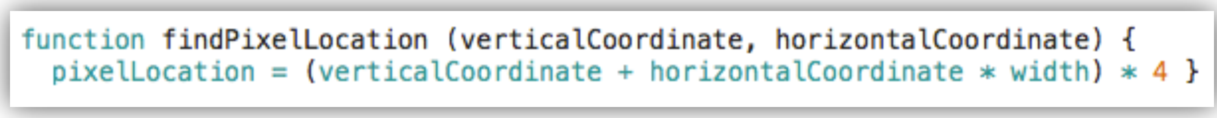
My purpose is to test Perlin noise in two-dimensional space. What I am going to do here is to color every single pixel inside the canvas with a grayscale value according to the corresponding Perlin noise value at that particular pixel.

Code

In JavaScript, there exist a function called noise() that returns a value between 0.0 and 1.0 represent the Perlin noise value at given coordinates.

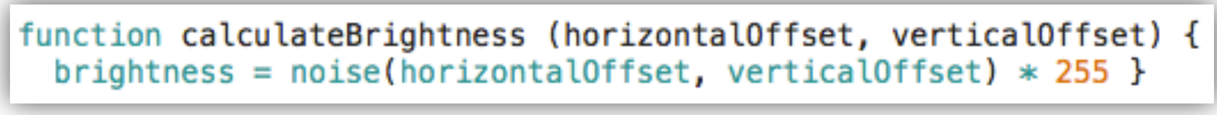
The idea is simple. For any pixel inside the canvas, do the following three steps:

- Find its location.



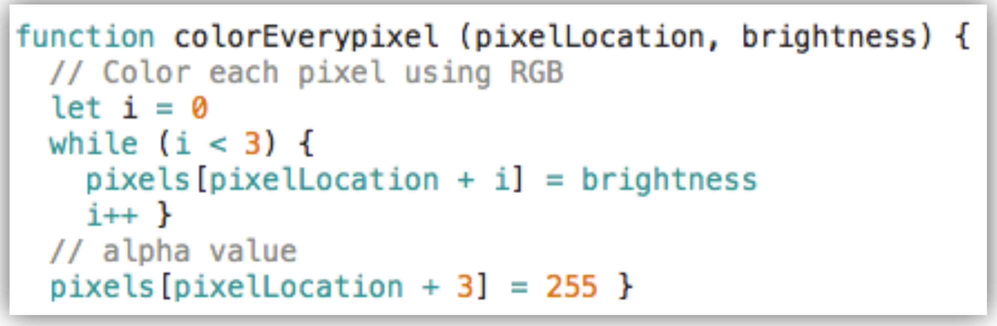
```
function findPixelLocation (verticalCoordinate, horizontalCoordinate) {
  pixelLocation = (verticalCoordinate + horizontalCoordinate * width) * 4 }
```

- Calculate its brightness using the built-in noise() function.



```
function calculateBrightness (horizontalOffset, verticalOffset) {
  brightness = noise(horizontalOffset, verticalOffset) * 255 }
```

- Color it according to the value from the previous step.



```
function colorEverypixel (pixelLocation, brightness) {
  // Color each pixel using RGB
  let i = 0
  while (i < 3) {
    pixels[pixelLocation + i] = brightness
    i++ }
  // alpha value
  pixels[pixelLocation + 3] = 255 }
```

Let us experiment Perlin noise for different increment values.

Increment	Result
5	Figure 7.1
0.5	Figure 7.2
0.05	Figure 7.3
0.005	Figure 7.4

Table 4 Different increment values for Perlin noise

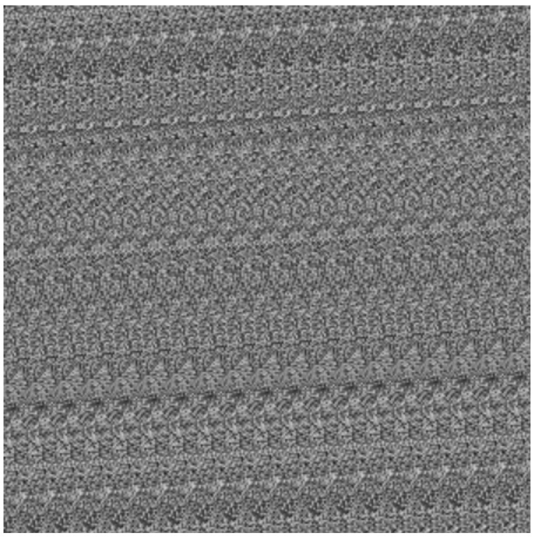


Figure 7.1 Perlin noise -1-

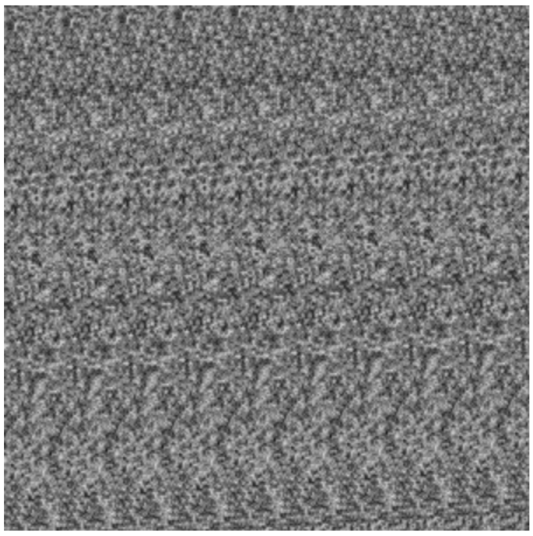


Figure 7.2 Perlin noise -2-

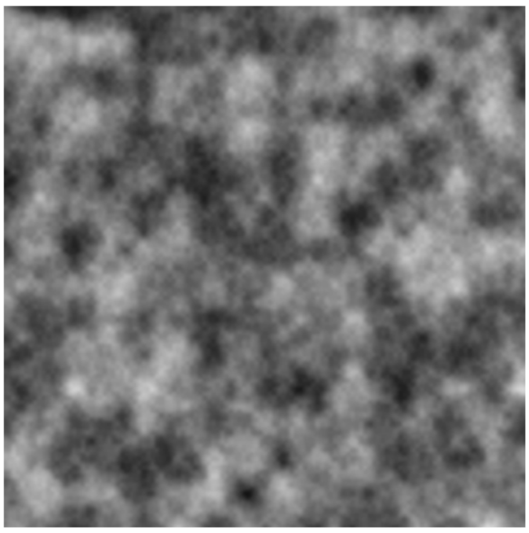


Figure 7.3 Perlin noise -3-

Figure 7.4 Perlin noise -4-

Discussion

By looking at the previous figures, we can easily infer that:

- As long as the increment decrease, the changes from one pixel to another decrease as well. Therefore, we end up with an almost gray image (figure 7.4).
- Looking closer at (figure 7.4), the image appears cloudy because pixels are similar to the surrounding pixels. It is very much like blurring an image. Consequently, the fewer changes there are, the more blurring effect there is.

Chapter 5

Filling space

1. Random Walks

This term was introduced for the first time in 1905 by Karl Pearson **[21].** Random walks are ubiquitous in applications of mathematics. They are used to modeling plenty of real-world phenomena in which the movement of an object can be described in a sequence of steps. The direction of each step is chosen randomly **[22]**, and is determined probabilistically. To simplify this concept, and demonstrate it in an interesting manner, let us assume that there is a drunk person stumbling around. His movement is a kind of walk haphazardly, and does not really have a predictable pattern of which way he will go. Random walks essentially emulate exactly this behavior. Let us say that the drunk person is walking on a two-dimensional grid. Then, he can move within certain number of directions (figure 8).

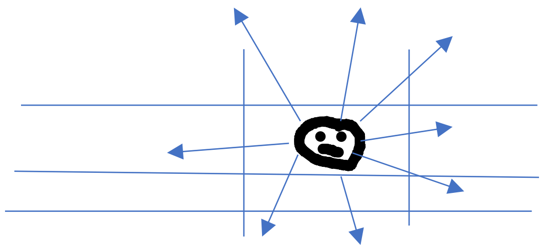


Figure 8 A drunk person moves in 2D space

Our task is to track how the drunk person moves over a long period of time. He would probably cross his own path sometimes. He might stumble back and forth. If we track everywhere that he would go, we most probably come up eventually with a pattern similar to a map. That is essentially what we are going to get when we perform a random walk. We just imagine that there is an object who just walks around the map. The object randomly picks a direction, and keep doing that over and over a specific number of times according to our application. This technique, as all other techniques, has advantages and disadvantages.

Its **advantages**:

- It is fast and simple to implement.
- It is a powerful tool if our purpose is just to create maps haphazardly.

Its main **drawback**:

- It is not the best choice if we want to generate maps that follow specific patterns. In such cases, we need more advanced techniques like cellular automata which will be discussed in detail later on in this chapter.

Random walks are not restricted to movement in two-dimensional space. It can be applied in any dimension. For example, If the random walks occur in two-dimensional integer grid, it starts somewhere on the grid. Then the next movement has equal probability to take place in one of the four available directions; up, down, left and right. In general, if the random walks occur in a d dimensions space, there will be 2*d directions to move on. Since any movement is equally likely to be in any direction, there is a 1/2d chance for the movement to occur in a specific direction.

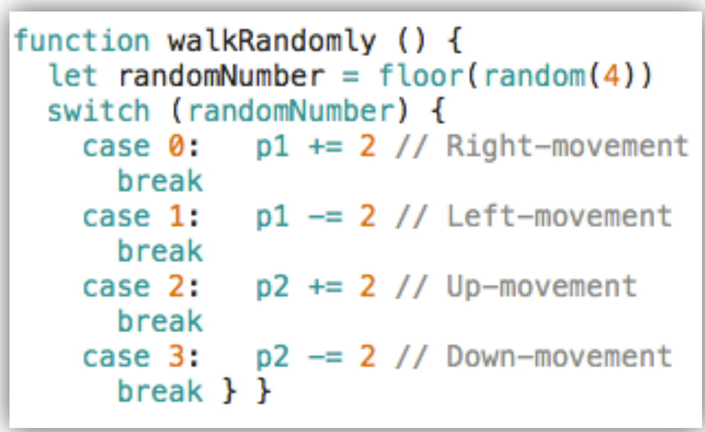
Implementation:

Let us implement a random walk in 2D space. We will start with a random point that has (x , y) coordinates, and at each step there are four different possibilities for the next movement. Each time one direction will be randomly chosen:

- Moving to the left →
- Moving to the right ←
- Moving up ↑
- Moving down ↓

Code

the code is very straightforward. Firstly, a random number between 0-3 is generated. Secondly, a direction is chosen depending on the previously generated value.



```
function walkRandomly () {
  let randomNumber = floor(random(4))
  switch (randomNumber) {
    case 0:   p1 += 2 // Right-movement
      break
    case 1:   p1 -= 2 // Left-movement
      break
    case 2:   p2 += 2 // Up-movement
      break
    case 3:   p2 -= 2 // Down-movement
      break } }
```

The step length can be chosen according to your specific need. In the above implementation, the step length is 2. You may have to modify the step length many times in order to end up with the best possible value that suits your application.

Running this code for roughly two-minutes generates the following map.

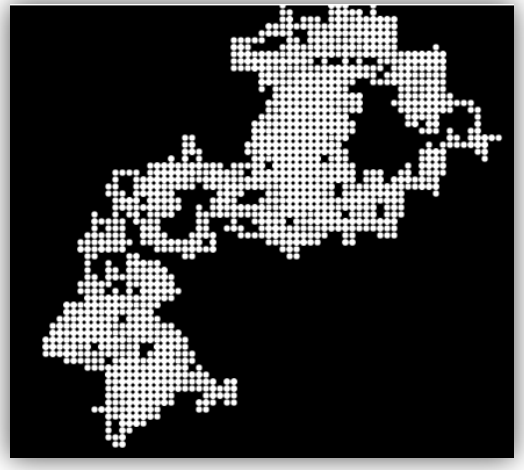


Figure 9.1 Random walk in 2D

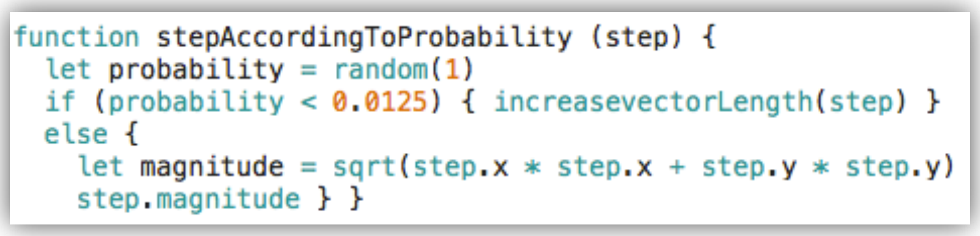
<u>Suggested-improvement (Modified version)</u>:

We all know that a vector has a length and a direction. Instead of having separate values for both x and y, we will use a vector that can point to any direction in the space, and has a variant length. This way, the random walks can be at happened at any given direction, and the length of the vector (step size) could change between different movements. This modification enhances the classical random walks, and make makes this version of random walks much more flexible. The reason is that the movement is no longer limited to occur in only one of the four obligatory directions. Previously, the probability of moving in any direction was exactly the same for all directions (25% for each direction in 2D space). But now, the probabilities will vary and the random walks do not have to be equal for each move. Some steps can be longer than others. Therefore, the probability of the step size will vary as well.

<u>Code</u>

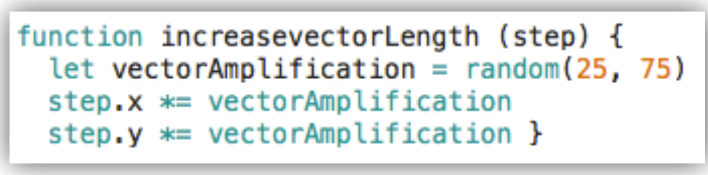
The main difference compared to the previous code is that we will use vectors instead of points. Otherwise, the core idea is still exactly the same. That is, every movement occurs according to a probability.

We generate a random number between 0 and 1. If its value is less than 0.0125, then increase the step value. Otherwise, move normally without any step amplification.



```
function stepAccordingToProbability (step) {
  let probability = random(1)
  if (probability < 0.0125) { increasevectorLength(step) }
  else {
    let magnitude = sqrt(step.x * step.x + step.y * step.y)
    step.magnitude } }
```

This function calls another function:



```
function increasevectorLength (step) {
  let vectorAmplification = random(25, 75)
  step.x *= vectorAmplification
  step.y *= vectorAmplification }
```

Running the improved version generates the following pattern

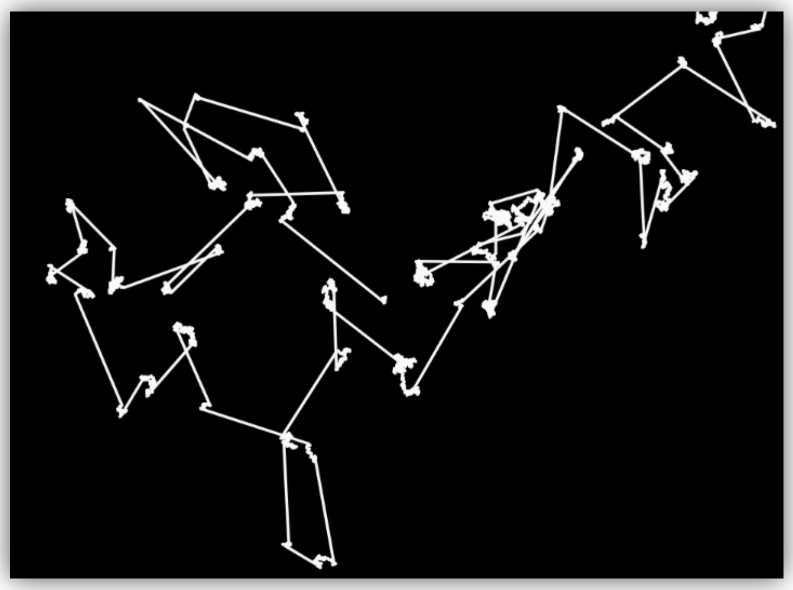


Figure 9.2 Improved version of random walk in 2D using vectors

Running it again, the random walker will follow a completely different path.

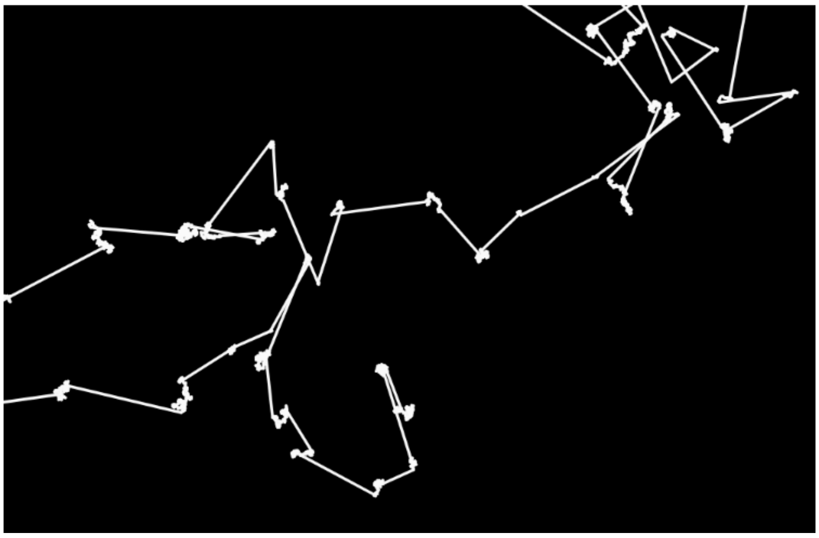


Figure 9.3 Improved version of random walk in 2D using vectors

2. Diffusion Limited Aggregation (DLA)

This algorithm proposed by both T.A Witten Jr. and L.M. Sander in 1981 **[23]**. The idea of this algorithm is quite simple. It is strongly related to the concept of random walks that we discussed earlier. At its core, it represents an object that grows gradually over time. It starts by only one particle, and then becomes larger and larger due to the process of diffusion and aggregation of other particles in the space. How this algorithm works can be described by the following steps **[24]**:

1. Locate a particle (point) somewhere in a 2D space.
2. Locate a set of random walkers (other points) randomly in the space.
3. Fire a random walker in the space.
4. Continuously, the random walker moves randomly until it hits the previously located particle /particles.
5. Attach the particles together.
6. Check the termination condition
    - 6.1. If it holds, Go to step 7.
    - 6.2. Otherwise, Go to step 3.
7. Terminate the execution.

This process repeats over and over again as long as there are random walkers waiting to be fired in the space. The path each random walker follows to reach the particle represents a part of the final generated content. By combining all those paths, we get the pattern this algorithm creates **[24]**.

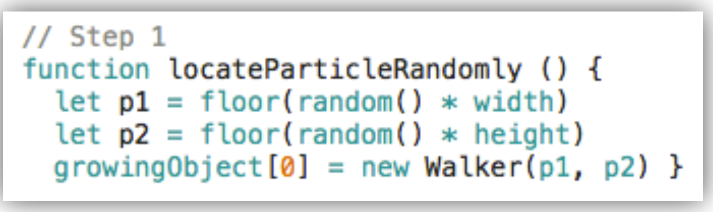
<u>Implementation</u>

This algorithm will be tested in a 2D space. The code is fairly large, and therefore I will only mention to the most important parts of it. However, the whole code can be found in the appendix.

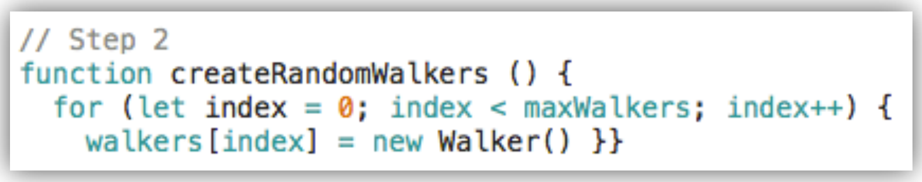
There are some important parameters that affects the performance of the algorithm:

- How many walkers in the system?
- The radius of each particle.
- The collision criterion.

Code

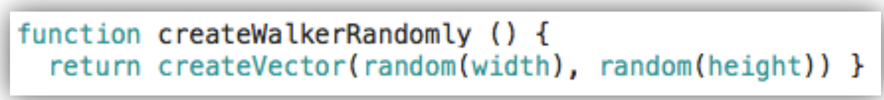


```
// Step 1
function locateParticleRandomly () {
  let p1 = floor(random() * width)
  let p2 = floor(random() * height)
  growingObject[0] = new Walker(p1, p2) }
```

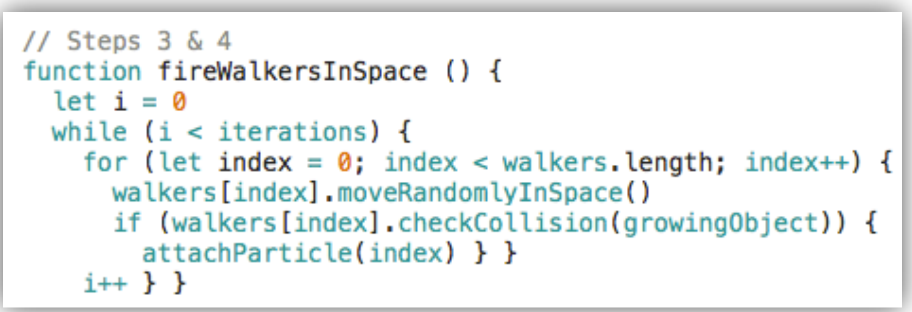


```
// Step 2
function createRandomWalkers () {
  for (let index = 0; index < maxWalkers; index++) {
    walkers[index] = new Walker() }}
```

Each walker inside this array is created randomly:

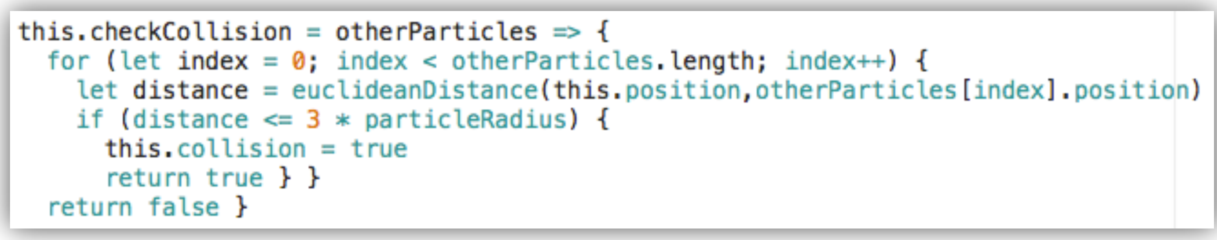


```
function createWalkerRandomly () {
  return createVector(random(width), random(height)) }
```

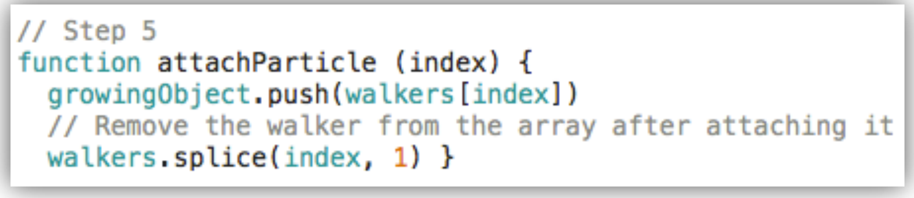


```
// Steps 3 & 4
function fireWalkersInSpace () {
  let i = 0
  while (i < iterations) {
    for (let index = 0; index < walkers.length; index++) {
      walkers[index].moveRandomlyInSpace()
      if (walkers[index].checkCollision(growingObject)) {
        attachParticle(index) } }
    i++ } }
```

This function calls another function that checks if a collision occurs between the current walker and the previously located particles.



```
this.checkCollision = otherParticles => {
  for (let index = 0; index < otherParticles.length; index++) {
    let distance = euclideanDistance(this.position,otherParticles[index].position)
    if (distance <= 3 * particleRadius) {
      this.collision = true
      return true } }
  return false }
```



```
// Step 5
function attachParticle (index) {
  growingObject.push(walkers[index])
  // Remove the walker from the array after attaching it
  walkers.splice(index, 1) }
```

Experiment 1

The collision criterion holds when the distance between the moving walker and any other particle in the space equals three times the radius of any particle.

The algorithms took roughly 6 minutes to generate the following pattern (figure 10.1).

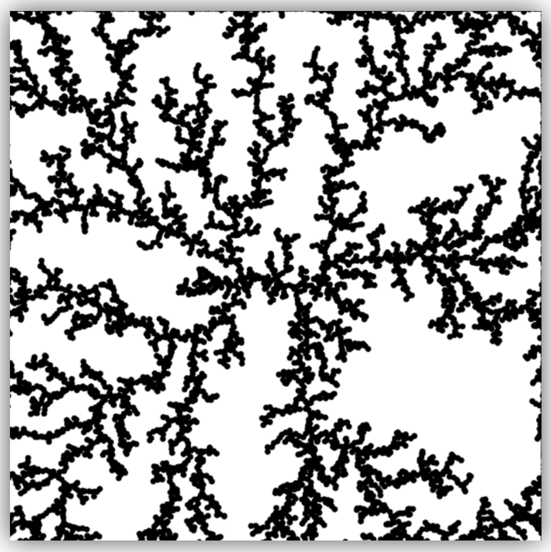


Figure 10.1 Generated content using DLA -1-

Experiment 2

The collision criterion holds when the distance between the moving walker and any other particle in the space equals twice the radius of any particle.

The algorithms took over 15 minutes to generate the pattern (figure 10.2).

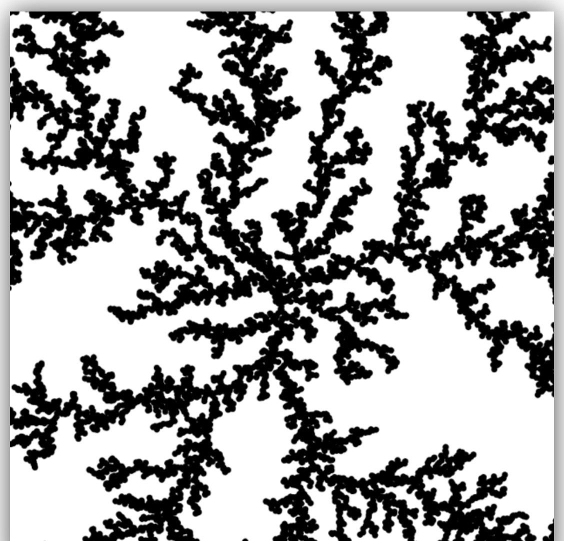


Figure 10.2 Generated content using DLA -2-

Discussion

The experiments showed that:

- Each time the algorithm run, a completely different content will be generated. The reason for that is due to the random, unpredictable behavior for each random walker. A random walker follows a different path each time, and therefore this algorithm generates a different pattern.
- The values of the parameters mentioned above affected the performance of the algorithm hugely. A simple modification in any parameter will dramatically increase / decrease the performance. For example, a little change in the termination criterion as mentioned in the experiments above decreased the performance in the second algorithm by roughly 3-times. As a result, changing the value of any parameter should be done very carefully.

3. Cellular Automata (CA)

This concept was originally discovered by Stanislaw Ulam and Jon von Neumann in the 1940s **[25]**. CA are algorithmic models that use computation to iterate on very simple rules to try to simulate complex phenomena **[26]**. CA can simulate a variety of real-world systems including biological and chemical ones.

There are two important principles related to CA as proposed in **[26]**.

1- Spatial structure: a typical example is a 2D integer grid structure. Every cell represents a location where an agent representing an individual is settled. You can think of this in a very abstract and simplified model of how people live in a city for example. A cell represents the place of a house or an apartment. Each cell has neighbors on each side.

2- Local interaction: means that individuals can only interact with others in a close distance around them. The way you model the neighborhood can be used to express how local or global the interaction your model is.

To model what happens in interaction in a cellular automaton, there are two basic principles **[27]**:

1- Agents in the cells have a state. The state represents, for example, their opinions, their ethnicities, etc...

2- States can change. Agents can change their states depending on their current state and the state of their neighbors as well.

Time is discrete in cellular automata models **[28]**. That is, time moves on in steps. We distinguish two important types of state change dynamics.

- Influence dynamics: the agents do not change their locations, but they change the state. For example, you may change your opinion depending on the opinion of the majority of your neighbors.
- Migration dynamics: the agent changes their locations. For example, the agent may move into another place in the worlds depending on the current state of the neighborhoods (if the agent has conflicts with the neighbors for instance).

Implementation

Due to its fame in the world of cellular automata, the *Game of live* has been chosen to be implemented in this section. This cellular automaton was invented in 1970 by John Horton Conway **[29]**. The Game of life exists in two dimensions. There are a number of generations in this game, and to move from one generation to another, you should run some computations on the grid's cells. Each state could be either 0 or 1. At each generation we evaluate the state

of the cells one by one and get a new state based on the previous state. In this cellular automaton, the neighborhoods for each cell in the grid are the eight cells surrounding it. This means that for each cell, we evaluate all its neighbors to determine whether it should stay as a zero or it should turn into new state; becomes one.
The reason it is called the Game of the life is because its rules for the movement from one generation to another resembles some type of biological process. If we think about population, a cell that is surrounded by dead neighbors cannot stay alive. On the other hand, a cell that is surrounded by neighbors that are alive, can come to live or stay alive. Finally, a cell that is surrounded by too many neighbors cannot stay alive due to overpopulation.
The ruleset for this cellular automaton can be described by Conways genetic laws **[29]**:

- Reproduction phase: a dead cell (its value 0) becomes alive if and only if it has three live neighbors.
- Survival phase: A live cell survives and moves to the next generation if and only if it has either two or three neighbors.
- Death phase: A live cell (its value 1) dies whether it has less than two or greater than three live neighbors.

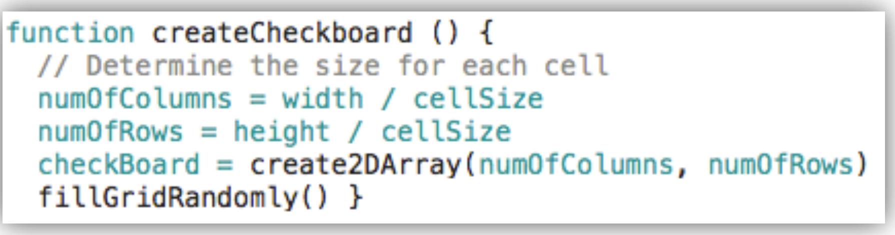
Keep in mind that all phases take place concurrently at each generation. The way the game of life works can be described as follows:

1. Create a checkerboard (2D grid).
2. Fill the checkerboard with random values (combination of zeros or ones).
   - Zero represents a dead cell.
   - One represents a live cell.
3. For each cell:
   - Count the number of its live neighbors.
   - Determine its new state in the next generation using Conway´s genetic laws.
4. At each generation, color the survival cells with white to distinguish them from other cell.

Code

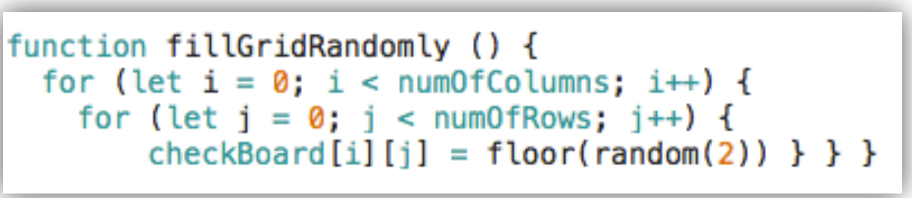
The following code is a step by step implementation of the steps above.

Step 1:



```
function createCheckboard () {
  // Determine the size for each cell
  numOfColumns = width / cellSize
  numOfRows = height / cellSize
  checkBoard = create2DArray(numOfColumns, numOfRows)
  fillGridRandomly() }
```

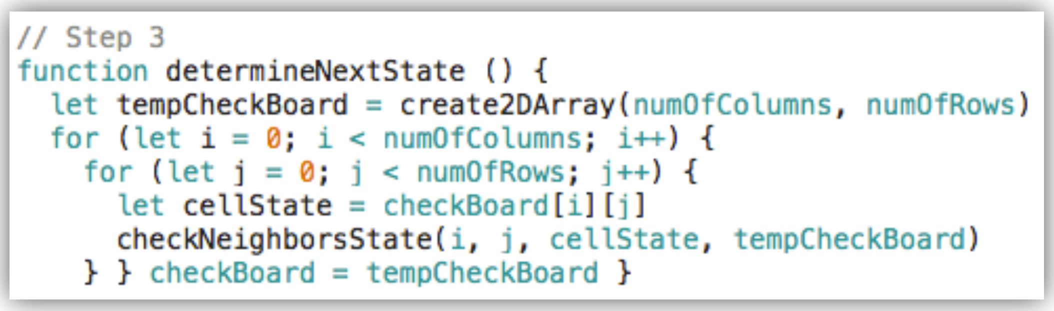
Step 2:



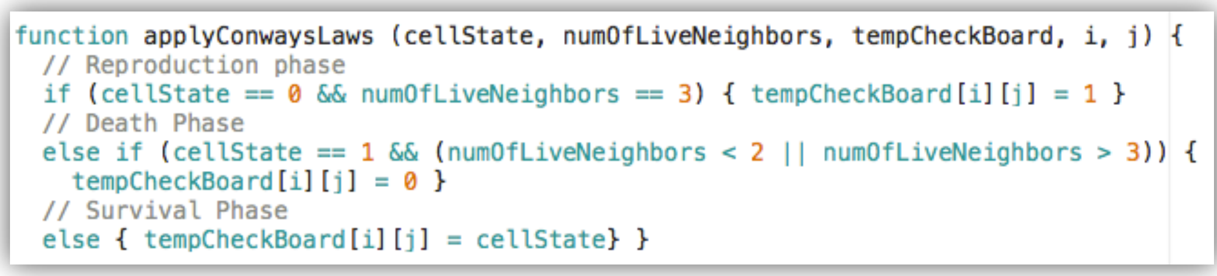
```
function fillGridRandomly () {
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
        checkBoard[i][j] = floor(random(2)) } } }
```

The number of neighbors for each cell depends on its location at the checkerboard. For example, a cell at the corner has only three neighbors, while the cell in the center has eight neighbors. Therefore, we should find the number of neighbors for each cell before applying Conways genetic rules.

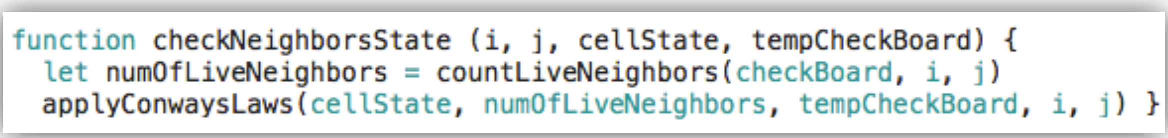
Step 3:



```
// Step 3
function determineNextState () {
  let tempCheckBoard = create2DArray(numOfColumns, numOfRows)
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
      let cellState = checkBoard[i][j]
      checkNeighborsState(i, j, cellState, tempCheckBoard)
    } } checkBoard = tempCheckBoard }
```

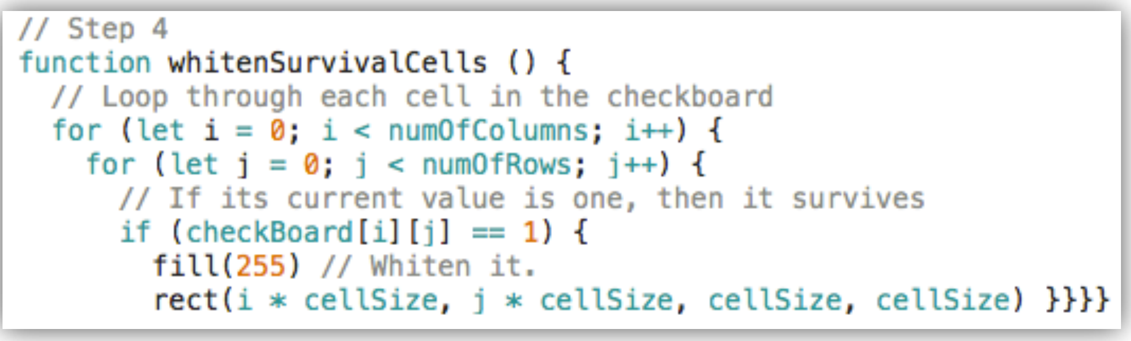


```
function applyConwaysLaws (cellState, numOfLiveNeighbors, tempCheckBoard, i, j) {
  // Reproduction phase
  if (cellState == 0 && numOfLiveNeighbors == 3) { tempCheckBoard[i][j] = 1 }
  // Death Phase
  else if (cellState == 1 && (numOfLiveNeighbors < 2 || numOfLiveNeighbors > 3)) {
    tempCheckBoard[i][j] = 0 }
  // Survival Phase
  else { tempCheckBoard[i][j] = cellState} }
```



```
function checkNeighborsState (i, j, cellState, tempCheckBoard) {
  let numOfLiveNeighbors = countLiveNeighbors(checkBoard, i, j)
  applyConwaysLaws(cellState, numOfLiveNeighbors, tempCheckBoard, i, j) }
```

Step 4:



```
// Step 4
function whitenSurvivalCells () {
  // Loop through each cell in the checkboard
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
      // If its current value is one, then it survives
      if (checkBoard[i][j] == 1) {
        fill(255) // Whiten it.
        rect(i * cellSize, j * cellSize, cellSize, cellSize) }}}}
```

Running this code generates the following pattern:

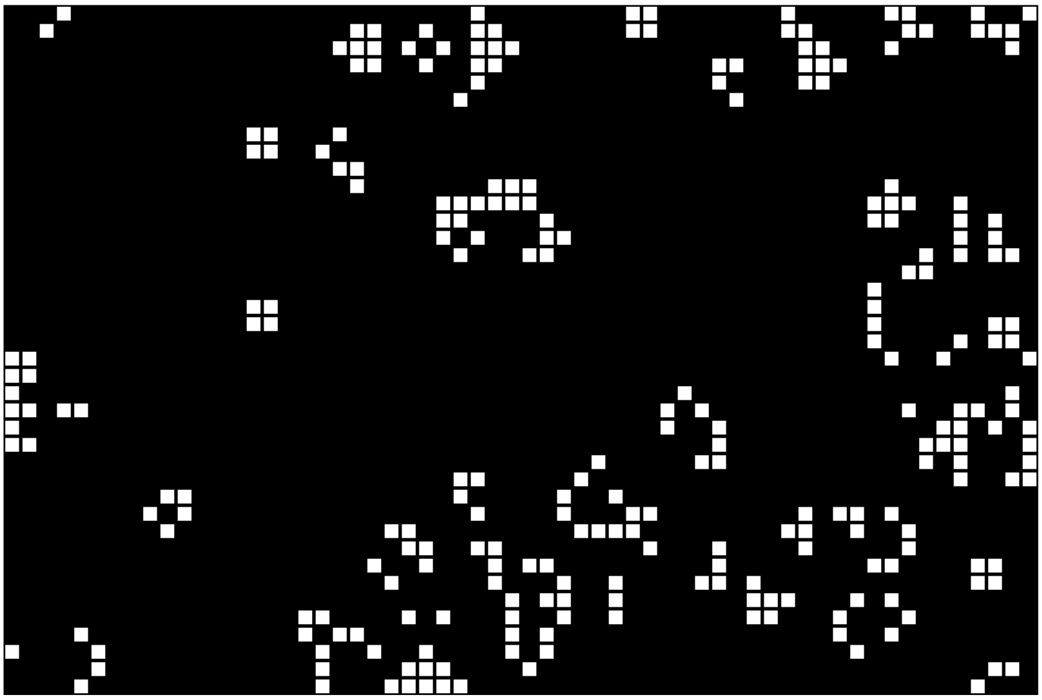


Figure 11 The Game of Life

Discussion

It is worth mentioning that in order to apply this algorithm, it does not necessitate a perfect grid of cells. I have used a 2D grid of cells just for demonstration purposes. It could be, for instance, a bunch of tiled triangles, or arbitrary polygons. But keep in your mind, that this change has a huge impact on the rulesets because it governs how many neighbors each cell is going to have. For example, if each cell has a hexagon shape, then it has six neighbors instead of 8 and so on.

Chapter 6

Sequence Generation

1. Markov Chains

This concept introduced for the first time by Andrey Markov in 1906 **[30]**. Markov chains are widely used in different kinds of systems. They can be used, among other things, in financial systems, scientific data, weather patterns, predicting earthquakes based on some set of sensor readings, and much more. Each Markov chain consists of multiple states. Each state has a certain probability of either moving on to a new state, or looping back onto itself, starting the process over. What Markov chains actually do is analyzing a sequence of states, and then generate outcomes based on that sequence. For the sake of simplicity, think of those states as the different states of a baby; playing, eating and crying.

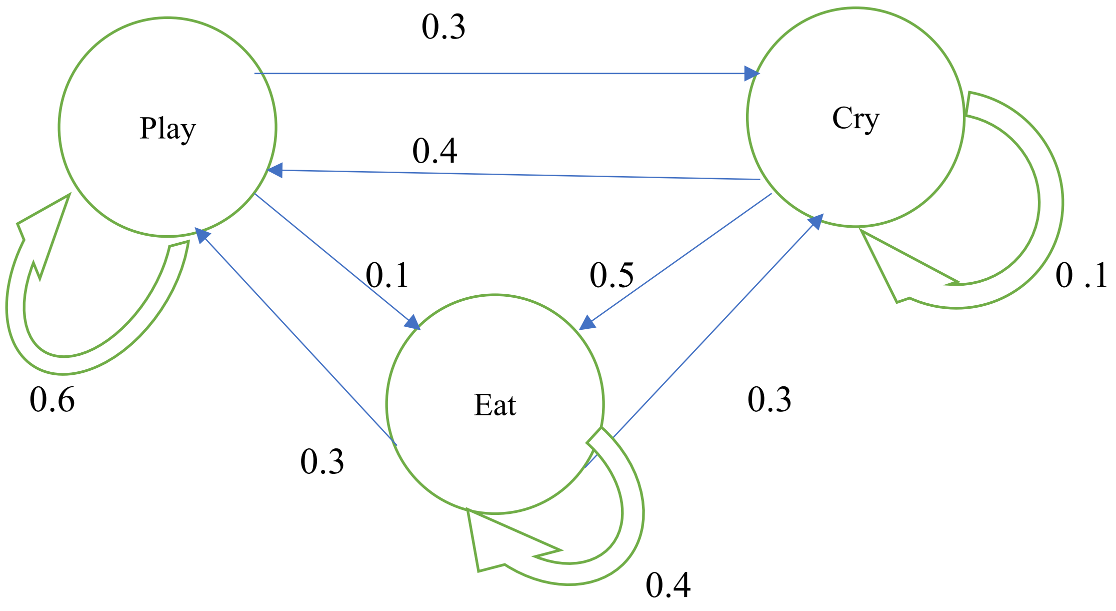


Figure 12.1 Markov chain for three states

Imagine that the baby can only be in one of these three states at a time. What really matters is the probability of moving between these states. Let us say that when the baby plays, there is a chance of 30% to cry if he does not like what he plays with or if he needs something from his mother for example. On the contrary of that, the chance to move from crying state to playing state is 40%. That is reasonable because whenever the baby cries, there is a high chance that his mother plays with him or gives him a toy to play with. The probability to move from playing state to eating state is not too high because when the baby plays, he does not ask for food unless he is hungry. Therefore, a 10% is reasonable. Each state has an internal transition where it moves to itself. Let us say that if the baby plays, then the likelihood to continue playing without crying or asking for food is 60%. Likewise, we can analyze the likelihood between all other states and come up with the likelihood for each possible transition(table 5).

State	**Playing**	**Eating**	**Crying**
Playing	60%	10%	30%
Eating	40%	30%	30%
Crying	40%	50%	10%

Table 5 Markov chain likelihoods for three states

The whole chain can be represented as a graph, and each state as a node belong to that graph. The transitions between states represent the edges in the graph. The weight on each edge represents the probability of the corresponding transition **[31]**. How Markov chains are generated, can be described by the following steps:

1. Start at a particular node.
2. The node´s value is considered as the first value of the sequence.
3. Choose the next node randomly from the nodes that branches from the current node.
4. The new node´s value is considered as a new value in the sequence.
5. If the termination condition does not hold, Go to step 3. Otherwise, Go to step 6.
6. Terminate the process.

The termination criterion depends entirely on the application. For example, if our purpose is to generate a chain of length equals to 10, then the termination condition will be a specific length of the sequence (in this case length = 10).

Implementation

In order to examine Markov chains in practice, let us implement an interesting application that uses Markov chains. Imagine the following scenario. Your wife gives birth, and you are hesitant which name you should opt for your darling newborn. Do not worry, Markov chains can be certainly of help to you. The purpose of this application is to procedurally generate a text (a name for your baby in our case).

Input: A set of random names (50 names in our implementation).

Output**:** The generator should generate10 different names, so you can choose one as a name to your baby. If you are not satisfied with the generated names, take it easy, and run the generator again. Each time you run the generator, you will get a new set of names. How flexible our generator is!

Before implementing this application, let us have a quick look at how a text can be generated using Markov Chains.

Text Generation

Generating a text using Markov chains is very popular and it is used in many applications. For example, you can use it to generate a title for your new book, or a poetry or music, and much more. Markov chains can be implemented in different ways. However, the core idea of any implementation is still exactly the same. Generally speaking, any implementation will contain at least the following steps somehow or another:

- Read input data, and split it into a big array of separate words.
- For each word (state), find the probability of a list of words that comes after/before it (possible transitions). This step may hugely vary from an implementation to another. You may only interest in the words that come after a specific word. Other developer may prefer to check the words before a given word, and so on. Another factor that affects the final result is how many words you are interested in after/before a given word. This step is usually referred to as building a Markov chains dictionary.
- Using the previously built dictionary, generate the next word (state) in the final chain as long as the termination condition has not held yet.

Remember that the more words in your input data are, the better results you get.

For the sake of simplicity, let us say that our input data contains only 10 names. Let us apply the previous steps and see what we will get.

Input: [Alex, Peter, Kevin, Alex, David, Tim, Peter, Alex, Kevin, Tim]

The next step is to represent the probability of each state. We start by creating a state for each word, and then assign it with its transition probability.

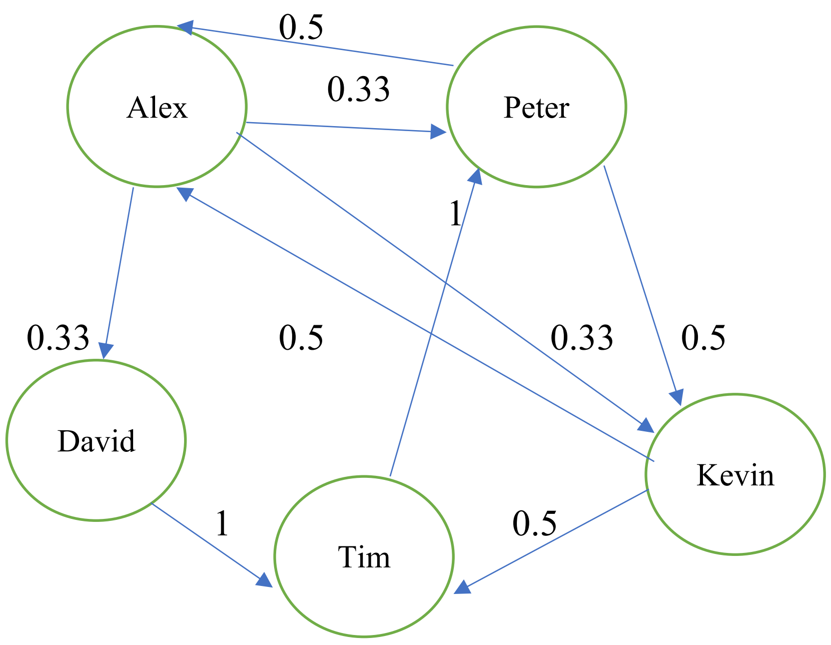


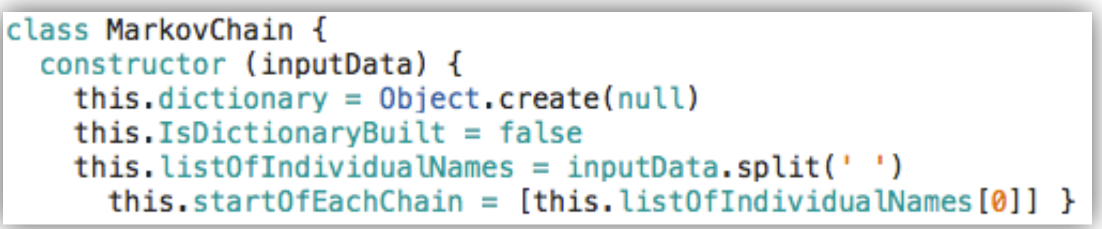
Figure 12.2 Markov Chain for five states.

The point of this example is just to demonstrate how we can build the Markov chain using text. Looking at the previous graph, we can predict that whenever the word Alex occurs in the chain, the likelihood for any of the following states (Peter, Kevin, David) is 33.33% to come after it. Likewise, whenever the state David occurs, it will always be followed by the state Tim. We analyze all other states in a similar manner.

Code

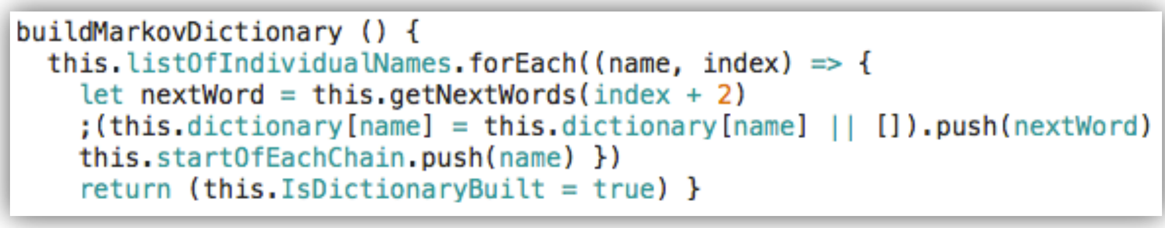
The way I implement this task is by creating a class called MarkovChain. In the constructor, the following are done:

1. Create an empty dictionary.
2. Split the input data into an array of individual names.



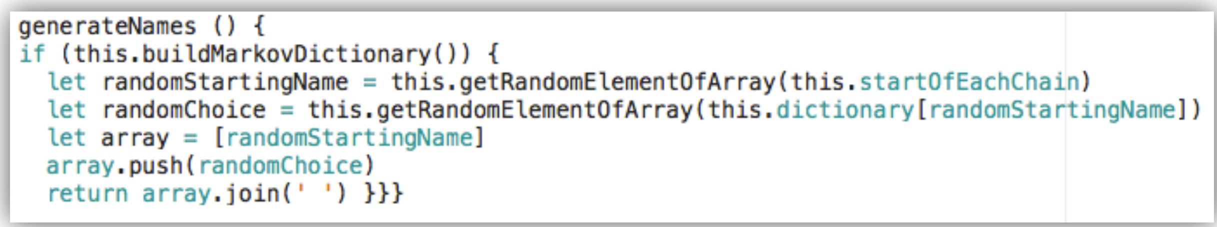
```
class MarkovChain {
  constructor (inputData) {
    this.dictionary = Object.create(null)
    this.IsDictionaryBuilt = false
    this.listOfIndividualNames = inputData.split(' ')
      this.startOfEachChain = [this.listOfIndividualNames[0]] }
```

3. The next step is building Markov dictionary:



```
buildMarkovDictionary () {
  this.listOfIndividualNames.forEach((name, index) => {
    let nextWord = this.getNextWords(index + 2)
    ;(this.dictionary[name] = this.dictionary[name] || []).push(nextWord)
    this.startOfEachChain.push(name) })
    return (this.IsDictionaryBuilt = true) }
```

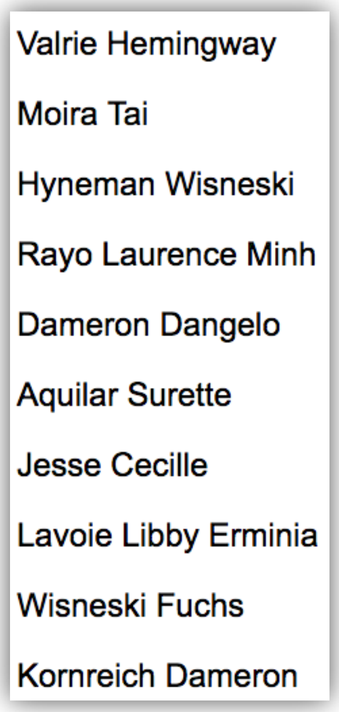
Lastly, a function that generate next name depending on the probabilities inside the dictionary.



```
generateNames () {
if (this.buildMarkovDictionary()) {
  let randomStartingName = this.getRandomElementOfArray(this.startOfEachChain)
  let randomChoice = this.getRandomElementOfArray(this.dictionary[randomStartingName])
  let array = [randomStartingName]
  array.push(randomChoice)
  return array.join(' ') }}}
```

There are some other utility functions in my implementation. You can find the complete code in the appendix.

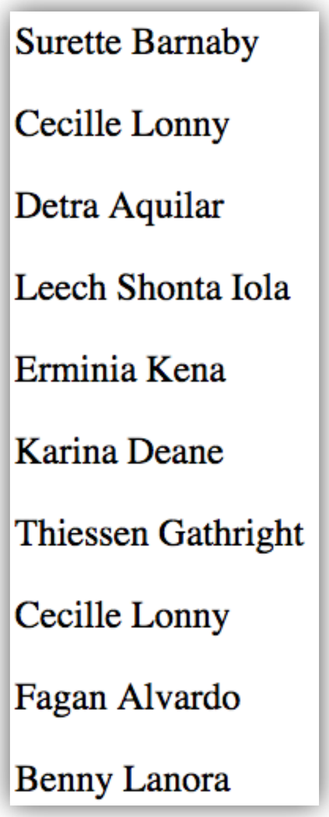
Running this generator generates the following sequence:



```
Valrie Hemingway
Moira Tai
Hyneman Wisneski
Rayo Laurence Minh
Dameron Dangelo
Aquilar Surette
Jesse Cecille
Lavoie Libby Erminia
Wisneski Fuchs
Kornreich Dameron
```

Figure 12.3 Generation of 10 random names using Markov chains

Running it again will generate another list of names, and so on:



```
Surette Barnaby
Cecille Lonny
Detra Aquilar
Leech Shonta Iola
Erminia Kena
Karina Deane
Thiessen Gathright
Cecille Lonny
Fagan Alvardo
Benny Lanora
```

Figure 12.4 Generation of 10 random names using Markov chains

2. Fractals

This term introduced for the first time in 1975 by Benoit Mandelbrot **[32].** In general, all shapes can be categorized into two main sub categorizes: Simple shapes and complex shapes. The former contains all kinds of well-known simple shapes such as a triangle, a rectangle, a circle, oval and so on. The latter contains any compound shapes that are made up of two or more simple shapes. Any car is can be considered as a complex shape. It has four circular tires, the mirrors have different shape, the chassis has another different shape and so on. In regards to fractals, a fractal is nothing more than a special kind of complex shapes. What make a fractal shape stands out, is its special structure that comprises sub-shapes identical to the main shape. This kind of symmetry is called self-similarity. To be more precise, the sub-shapes do not need to be completely identical to form a fractal, but the similarity to a great extent is very obvious. The most important aspect of fractals is that they are infinite **[33]**, at least from a theoretical point of view. Practically, the recursive loop of creating sub-shapes of the same fractal should be terminated somehow or another at some point. In other words, we have to limit how many times the recursion should be performed.

One of the most prominent fractal patterns is Sierpinski triangle which is named after Waclaw Sierpinski **[34]**. This pattern starts with one big equilateral triangle. At each iteration, the triangle is subdivided into smaller equilateral triangles, and the process goes over and over again until a predefined termination condition holds.

There exist seven different common ways to construct this pattern **[35]**.

1. Removing triangles.
2. Shrinking and duplication.
3. Chaos game.
4. Arrowhead construction of Sierpinski gasket.
5. Cellular automata.
6. Pascal´s triangle.
7. Towers of Hanoi.

The method I have chosen to implement this pattern is *chaos game.* You may wonder why I have particularly chosen this method. My answer is just because this method is the simplest one with regard to coding, in my humble opinion.

2.1 Chaos Game

This method introduced by Michael Barnsley **[36]**. The way this method works can be described by the following steps **[37]**:

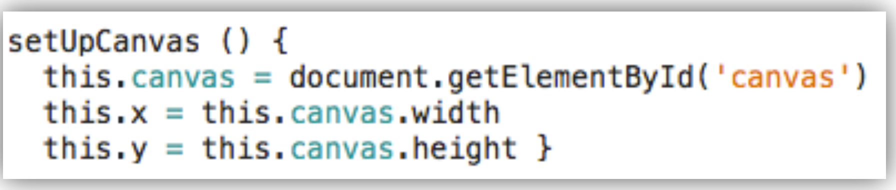
1. Choose any three points in the space (canvas in our case) to constitute an equilateral triangle.
2. Choose randomly any point inside the previous triangle.
3. The chosen point is considered to be the current position in the space.
4. Choose randomly any one of the three vertex points (from step 1).
5. Calculate half the distance between the current position and the chosen point in step 4.
6. Move to the new position, and plot it.
7. Check the termination condition:
   7.1 If you are done, Go to step 8.
   7.2 Otherwise, go to step 3 again.
8. Terminate the process.

Implementation

My implementation is very straightforward. It is just coding the previous steps one by one.

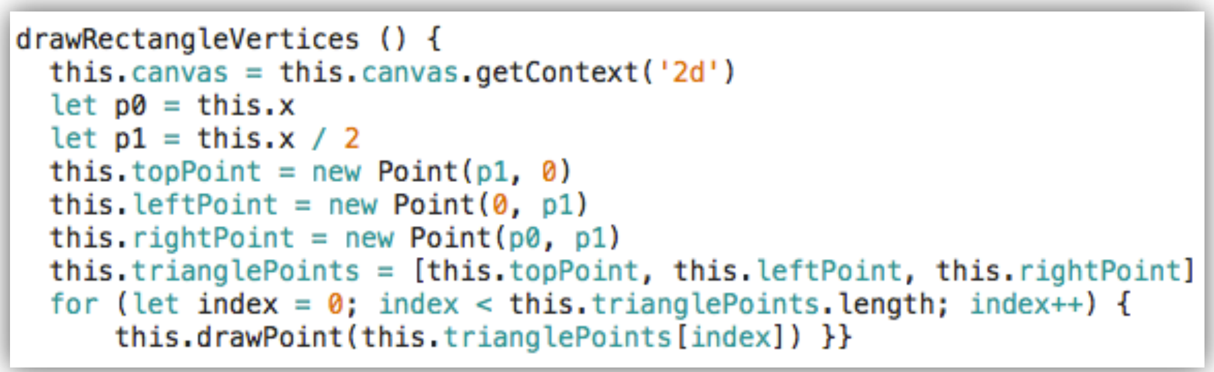
Code

Firstly, set up the plane(canvas in my implementation) to be ready for drawing .



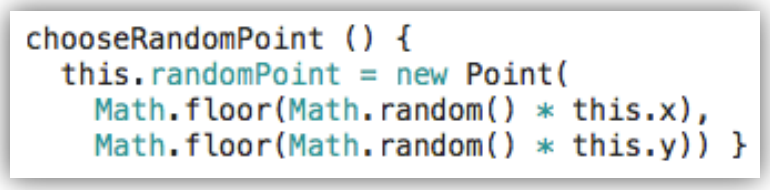
```
setUpCanvas () {
  this.canvas = document.getElementById('canvas')
  this.x = this.canvas.width
  this.y = this.canvas.height }
```

Then, step 1:



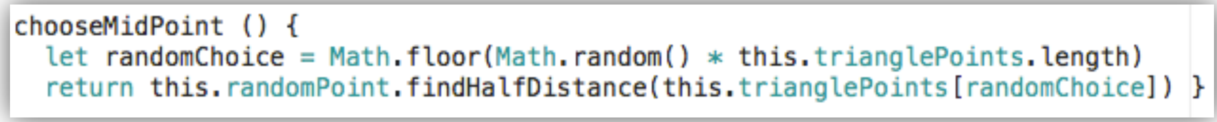
```
drawRectangleVertices () {
  this.canvas = this.canvas.getContext('2d')
  let p0 = this.x
  let p1 = this.x / 2
  this.topPoint = new Point(p1, 0)
  this.leftPoint = new Point(0, p1)
  this.rightPoint = new Point(p0, p1)
  this.trianglePoints = [this.topPoint, this.leftPoint, this.rightPoint]
  for (let index = 0; index < this.trianglePoints.length; index++) {
      this.drawPoint(this.trianglePoints[index]) }}
```

After that, step 2:



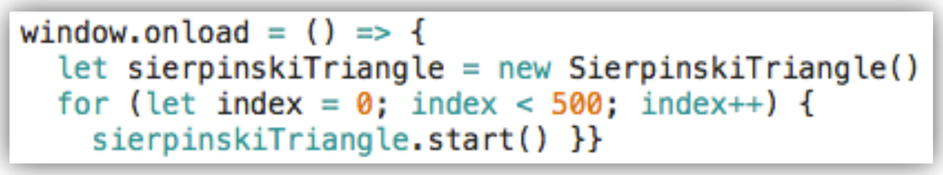
```
chooseRandomPoint () {
  this.randomPoint = new Point(
    Math.floor(Math.random() * this.x),
    Math.floor(Math.random() * this.y)) }
```

Then step 4 & 5 :



```
chooseMidPoint () {
  let randomChoice = Math.floor(Math.random() * this.trianglePoints.length)
  return this.randomPoint.findHalfDistance(this.trianglePoints[randomChoice]) }
```

Finally, check the termination condition (step 7).



```
window.onload = () => {
  let sierpinskiTriangle = new SierpinskiTriangle()
  for (let index = 0; index < 500; index++) {
    sierpinskiTriangle.start() }}
```

The complete code can be certainly found in the appendix.

Let us run this algorithm for different numbers of iterations (table 6).

Number of iterations	Result
5	Figure 13.1
50	Figure 13.2
500	Figure 13.3
5000	Figure 13.4

Table 6 Different number of iterations for Chaos Game

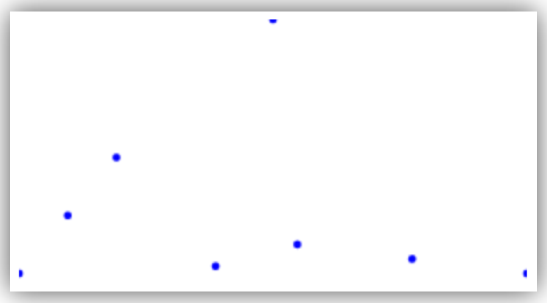


Figure 13.1 Sierpinski triangle -1-

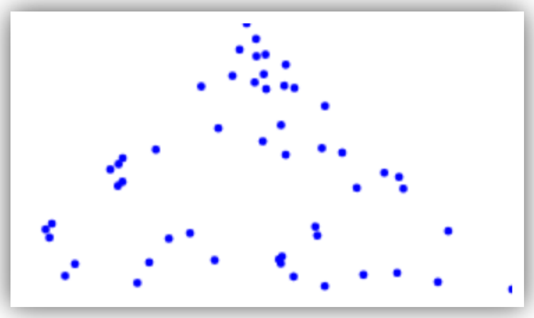


Figure 13.2 Sierpinski triangle -2-

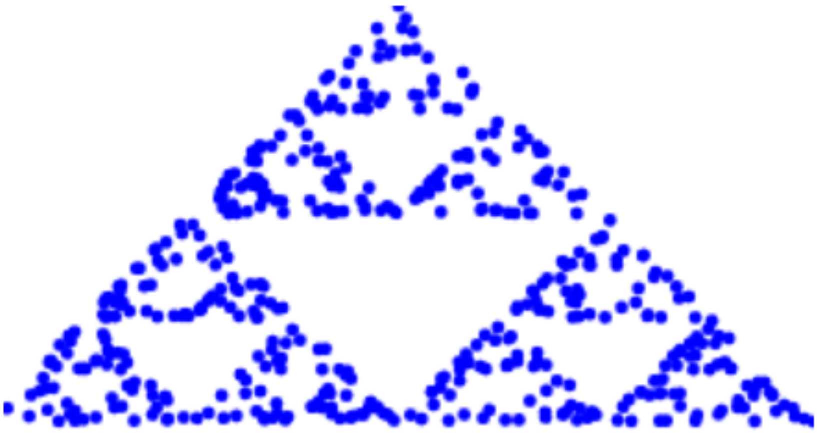


Figure 13.3 Sierpinski triangle -3-

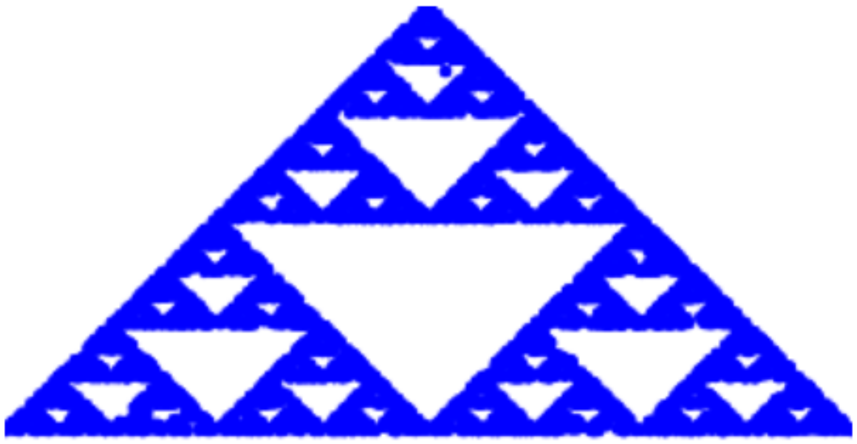


Figure 13.4 Sierpinski triangle -4-

<u>**Discussion**</u>

For a few iterations, we ended up with a figure that had only a few points distributed randomly inside the main triangle. In fact, that seems a sort of chaos.

For 500 iterations, the pattern became clearer (figure 13.3). It is still, however, not totally unmistakable. In order to get a satisfied result, a huge number of iterations should be performed (figure 13.4). The last experiment executed for 5000 iterations.

According to the huge number of experiments I have done on this method, it is worth to mention that interestingly enough :

- The order of the vertices does not matter at all as long as they are chosen randomly and with equal probability.

2.2. Space Colonization Algorithm[SCA]

This algorithm is all about colonizing a two-dimensional space of leaves in order to build a tree iteratively. It starts with a collection of attraction points represents the leaves of the tree, and one or more tree nodes.

Let D1: the distance between a tree node and an attraction point.

D2: influence distance.

D3: kill distance.

There are two rules govern how this algorithm works **[38]**:

1- An attraction point attracts a tree node if: D1 < D2.

2- An attraction point is removed if: D1< D3.

In other words, for any attraction point(leaf) to be relevant, the following criterion should be true: D3 < D1 < D2.

The way this algorithm works can be described as follows **[38]**:

1- The tree structure contains initially random number of tree nodes and attraction points.

2- Start a new iteration, and do the following:

3- Each attraction point is associated with the closest tree node if the rule-1- holds.

4- The normalized vectors from each node to its attraction point are found.

5- The vectors are added and their sum is normalized again causing new nodes to be added to the tree.

6- Check rule-1- and rule-2- each time new tree nodes are added.

7- Construct the tree again depending on the results of step 6.

8- Go to step 2.

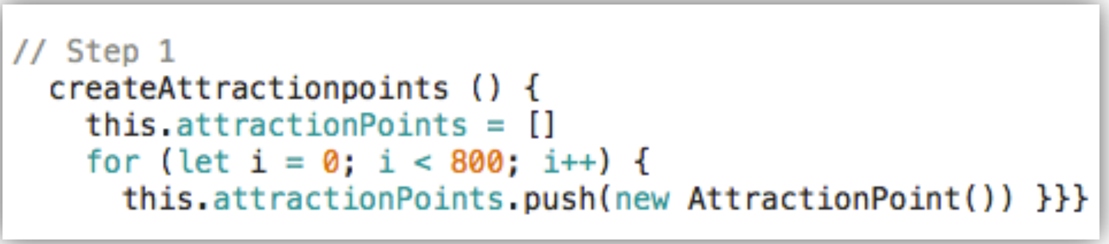
Implementation

This algorithm will be tested for different numbers of leaves (table 7).

Code

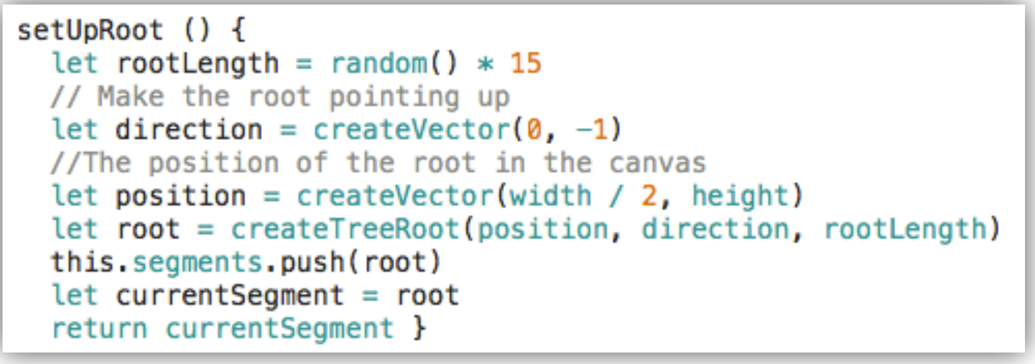
To end up with a tree, two things must be done firstly:

1. Create a number of attraction points (leaves) inside the space (the canvas in our case).



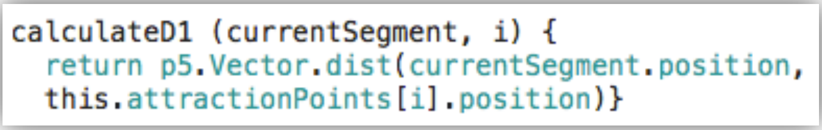
```
// Step 1
  createAttractionpoints () {
    this.attractionPoints = []
    for (let i = 0; i < 800; i++) {
      this.attractionPoints.push(new AttractionPoint()) }}}
```

2. No tree can ever be existed without a root. Therefore, the root of should be created.



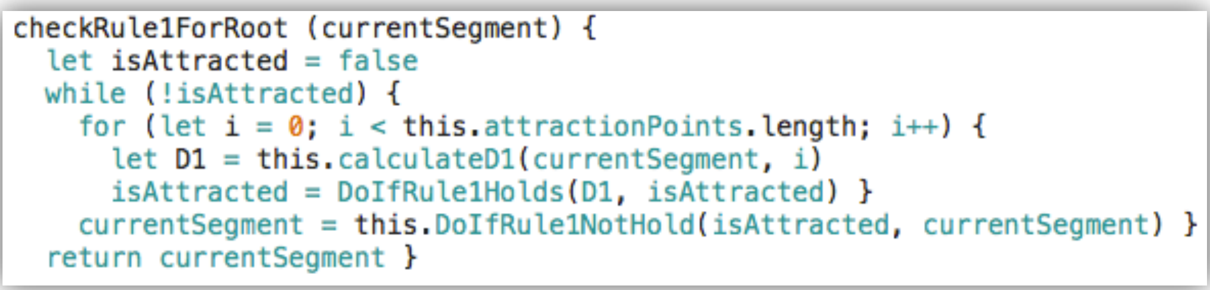
```
setUpRoot () {
  let rootLength = random() * 15
  // Make the root pointing up
  let direction = createVector(0, -1)
  //The position of the root in the canvas
  let position = createVector(width / 2, height)
  let root = createTreeRoot(position, direction, rootLength)
  this.segments.push(root)
  let currentSegment = root
  return currentSegment }
```

At this point, the space contains both leaves and root. Let us begin by checking if the root hopefully locates in a good position to be attracted by one of the attraction points. To do that, the distance D1 should be calculated beforehand:



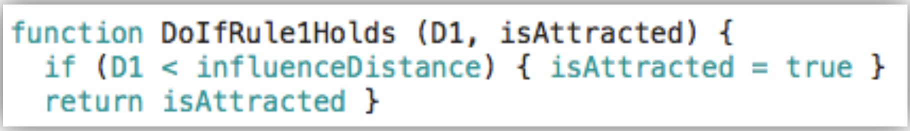
```
calculateD1 (currentSegment, i) {
  return p5.Vector.dist(currentSegment.position,
  this.attractionPoints[i].position)}
```

Then we check if the rule-1- holds for the root:



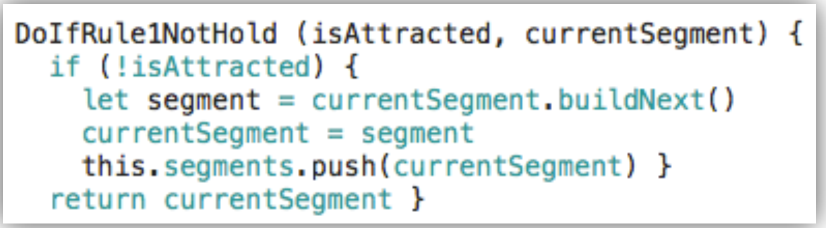
```
checkRule1ForRoot (currentSegment) {
  let isAttracted = false
  while (!isAttracted) {
    for (let i = 0; i < this.attractionPoints.length; i++) {
      let D1 = this.calculateD1(currentSegment, i)
      isAttracted = DoIfRule1Holds(D1, isAttracted) }
    currentSegment = this.DoIfRule1NotHold(isAttracted, currentSegment) }
  return currentSegment }
```

This function calls another two functions. The first one when the rule-1- holds. The other one when the rule-1- does not hold. Remember, if the rule holds, the root will be attracted.



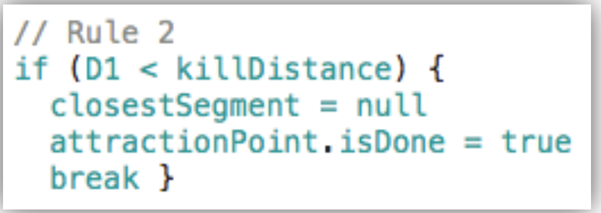
```
function DoIfRule1Holds (D1, isAttracted) {
  if (D1 < influenceDistance) { isAttracted = true }
  return isAttracted }
```

If that is not the case, do not give up. Discard the root, create a new segment and continue your adventure. The new segment becomes a new root, and the process goes on over and over again until a root gets attracted. That is reasonable because it does not make sense to build the body of a tree before getting the root in place.



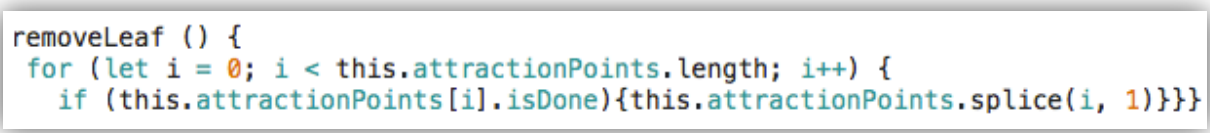
```
DoIfRule1NotHold (isAttracted, currentSegment) {
  if (!isAttracted) {
    let segment = currentSegment.buildNext()
    currentSegment = segment
    this.segments.push(currentSegment) }
  return currentSegment }
```

The next step is to expand the tree. First of all, we have to find the closest branch to each leaf, and check if it locates within the range [D2, D3]. In other words, we have to check the rules 1&2 for every single attraction point inside the canvas. Let us start by rule-2- to figure out if an attraction point is done.



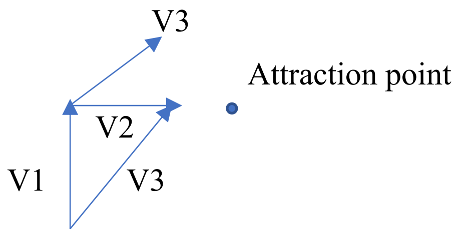
```
// Rule 2
if (D1 < killDistance) {
  closestSegment = null
  attractionPoint.isDone = true
  break }
```

If that is the case, this particular attraction point should be removed because it is no longer has a role in the algorithm.

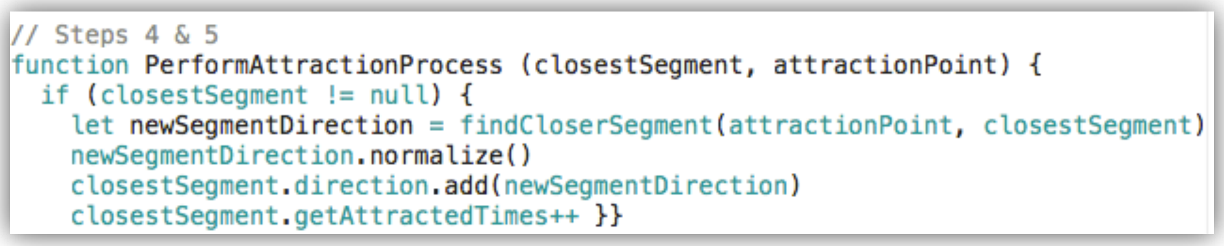


```
removeLeaf () {
 for (let i = 0; i < this.attractionPoints.length; i++) {
   if (this.attractionPoints[i].isDone){this.attractionPoints.splice(i, 1)}}}
```

Then we check the rule-1-. If it holds, there exists closest segment for this particular attraction point. Assume that v1: closest segment, v2: normalized vector. Then v3= v1+v2.

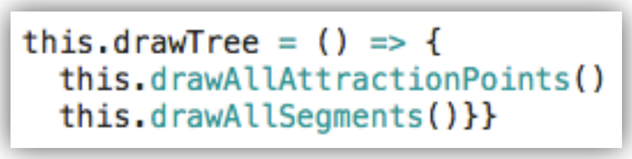


Whenever exists closest segment, the steps 4 & 5 should be applied. A normalized vector should be found and added to the closest segment vector in order to constitute the new vector v3 that is closer to the attraction point. The normalized vector is found by simply subtracting the current position of the attraction point from the position of the closest segment.



```
// Steps 4 & 5
function PerformAttractionProcess (closestSegment, attractionPoint) {
  if (closestSegment != null) {
    let newSegmentDirection = findCloserSegment(attractionPoint, closestSegment)
    newSegmentDirection.normalize()
    closestSegment.direction.add(newSegmentDirection)
    closestSegment.getAttractedTimes++ }}
```

When everything is done, the tree should be drawn on the canvas:



```
this.drawTree = () => {
  this.drawAllAttractionPoints()
  this.drawAllSegments()}}
```

The algorithm was tested for the values listed in table 7:

D2	D3	Number of leaves	Result
150	10	80	14.1
150	10	800	14.2
150	10	8000	14.3
150	10	80000	14.4

Table 7 Tested values used in SCA

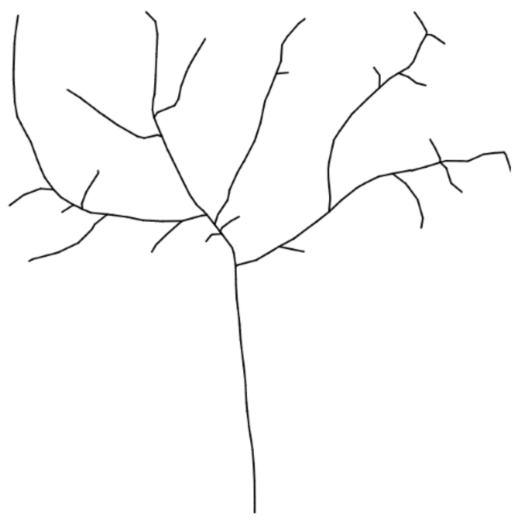


Figure 14.1 Generated tree SCA-1-

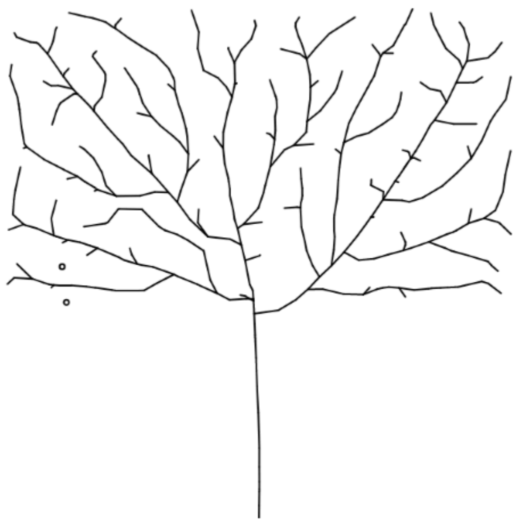


Figure 14.2 Generated tree SCA-2-

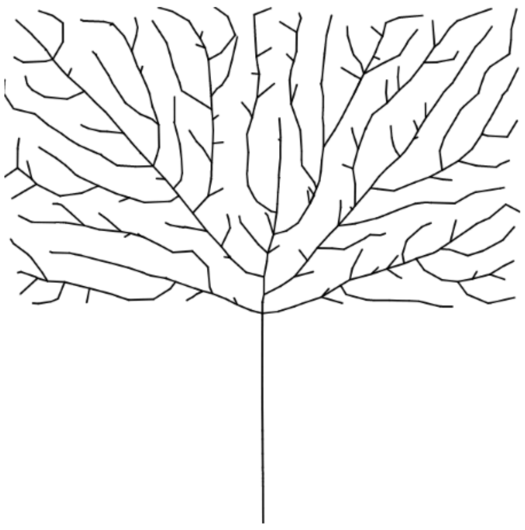


Figure 14.3 Generated tree SCA-3-

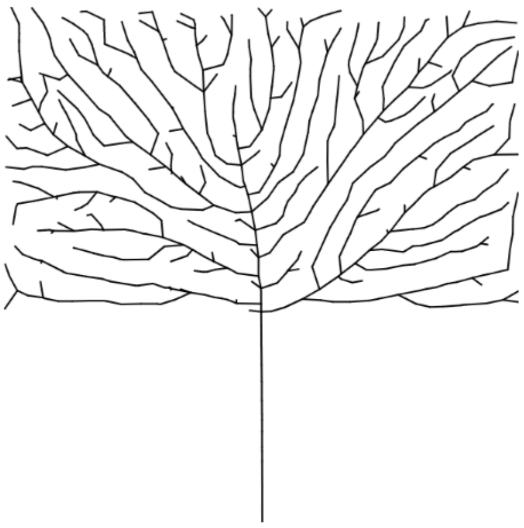


Figure 14.4 Generated tree SCA-4-

Discussion

The experiments showed that

1- The density of the tree increases when the kill distance decreases.
2- The density of the tree increases when the number of leaves increases.
3- If any two attraction points are by coincidence located at the same distance from the any tree node, then both points will be ignored. That is, they exist in the space alone (figure 14.2).

2.3. Lindenmayer Systems (L-Systems)

L-Systems was named after Aristid Lindenmayer when he introduced this concept for the first time by in 1968 **[39]**. L-System is used to generate fractal patterns. This topic is broad in the sense that there are tons of L-systems that you can make. We can describe it as a recursive way of generating sentences over and over using string replacement. L-system at its core is really a grammar. It is rewriting system. It is a way of looking at strings of characters, and rewriting those strings of characters again in a recursive way.

An L-system has **[40]** :

- An alphabet represents the allowed characters in this particular L-system.
- An axiom represents the beginning of a particular L-system.
- A set of rules represents the production rules. We can have one or more rules.

Implementation

Two different systems will be implemented in this section. The first one is a simple and classic example. The second one is interesting and challenging at the same time.

Experiment 1

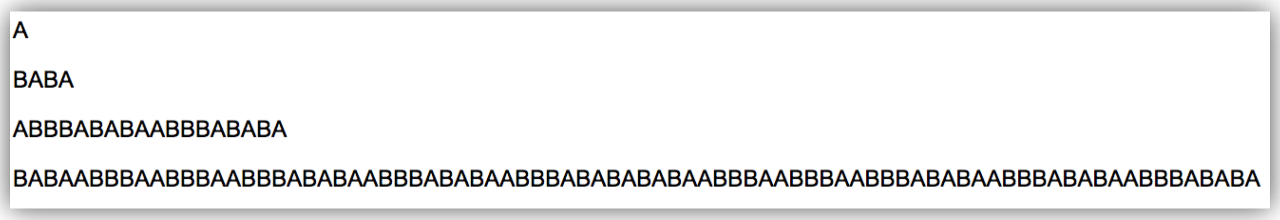
Let us experiment first L-system by using two simple rules. The choice of rules is completely random. Any change in the rules will affect the generated content. For example, if we start by the following two randomly chosen rules:

- Alphabet: A and B.
- Axiom is A.
- Ruleset:

  1- $A \longrightarrow$ BABA.

  2- $B \longrightarrow$ ABBBA.

Running this system for 3-generations generates the following sentences:



```
A
BABA
ABBBABABAABBBABABA
BABAABBBAABBBAABBBABABAABBBABABAABBBABABABABAABBBAABBBAABBBABABAABBBABABAABBBABABA
```

Figure 15.1 Generated sentences using simple L-System

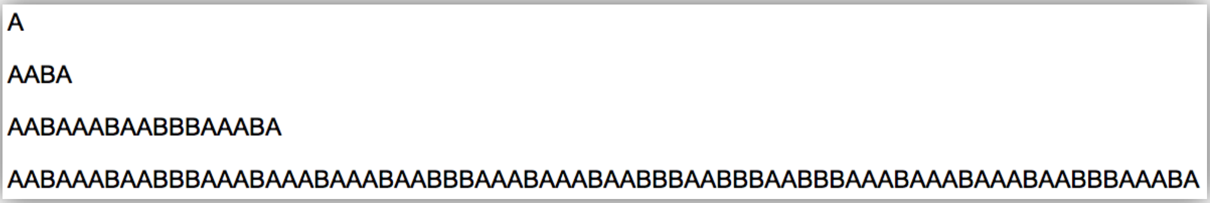
Let us make a tiny change in the first rule by just changing the first letter to becomes A instead of B. Everything else remains the same.

Ruleset :

A $\longrightarrow$ AABA.

B $\longrightarrow$ ABBBA.

The new generated sentence is:



```
A
AABA
AABAAABAABBBAAABA
AABAAABAABBBAAABAAABAAABAABBBAAABAAABAABBBAABBBAABBBAAABAAABAAABAABBBAAABA
```

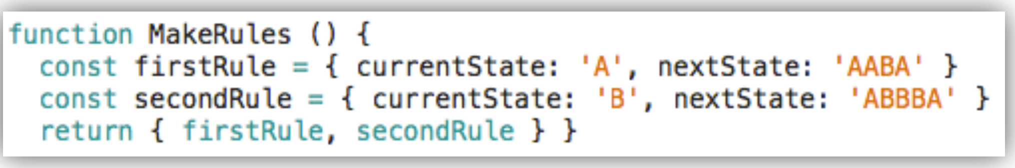
Figure 15.2 Generated sentences using simple L-System

Unmistakably, the two set of sentences are entirely different.

<u>**Code**</u>

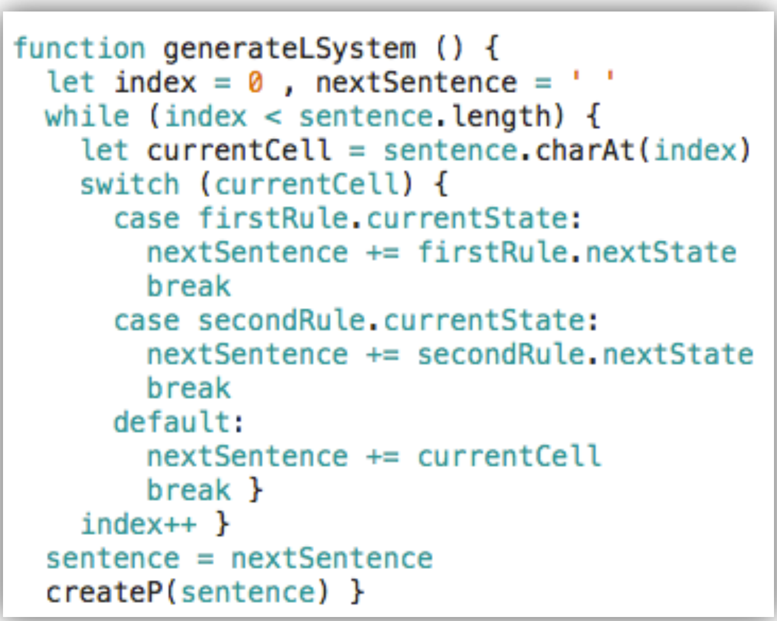
The coding process can be divided into three main parts.

Firstly, generating the rules:



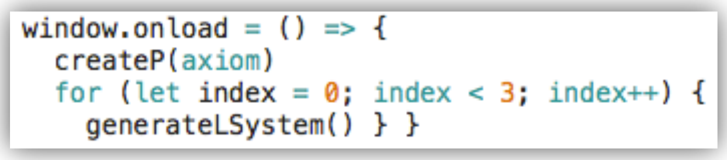
```
function MakeRules () {
  const firstRule = { currentState: 'A', nextState: 'AABA' }
  const secondRule = { currentState: 'B', nextState: 'ABBBA' }
  return { firstRule, secondRule } }
```

Then, generating a number of sentences depending on the previously built rules:



```
function generateLSystem () {
  let index = 0 , nextSentence = ' '
  while (index < sentence.length) {
    let currentCell = sentence.charAt(index)
    switch (currentCell) {
      case firstRule.currentState:
        nextSentence += firstRule.nextState
        break
      case secondRule.currentState:
        nextSentence += secondRule.nextState
        break
      default:
        nextSentence += currentCell
        break }
    index++ }
  sentence = nextSentence
  createP(sentence) }
```

Finally, checking the termination condition, and display the generated sentences so far.



```
window.onload = () => {
  createP(axiom)
  for (let index = 0; index < 3; index++) {
    generateLSystem() } }
```

The generated sentences can be interpreted in unlimited number of ways. It depends totally on your application. For example, you may want to read the generated sentences as a poetry. Or perhaps you would like to use it to be a song, or you tie it to graphics and so on. It is all up to you. After implementing the simplest form of L-system, let us go one step further and implement something more interesting.

Experiment 2

In the first implementation, the alphabet included only two characters A and B. Let us expand it a little bit to contain five characters F, +, -, [,]. Likewise, let us replace the simple rules with a complex one. So, our new L-system looks like:

Alphabet: F + - [ ]

Axiom: F

Ruleset: F $\longrightarrow$ FF+[+F-F-F]-[-F+F+F]

Again, whenever you make any change in the ruleset, the generated content will vary accordingly. The purpose of this implementation is to read each generated sentence, and then draw it on the screen (canvas). The drawing in this case represents our interpretation of the generated sentences. But before doing that, we have to decide exactly how to interpret the generated sentences. Suppose that our interpretation is something similar to the following:

- F: means move forward and draw a line.
- +: means turns right.
- -: means turns left.
- []: means save where you are. This is useful for backtracking.

There are absolutely no must-follow patterns or rules at all on how to interpret the generated sentences. You can, for instance, interpret F as a rotation with 60 degrees, and + as a drawing a rectangle, and so on and so forth. Just play with the generated sentences, and interpret it differently each time, and see how impressive is that.

Let us inspect the generation process step by step, and running the system for four different generations (table 8).

Number of generations	Result
1	Figure 15.3
2	Figure 15.4
3	Figure 15.5
4	Figure 15.6

Table 8 Tested values for L-System

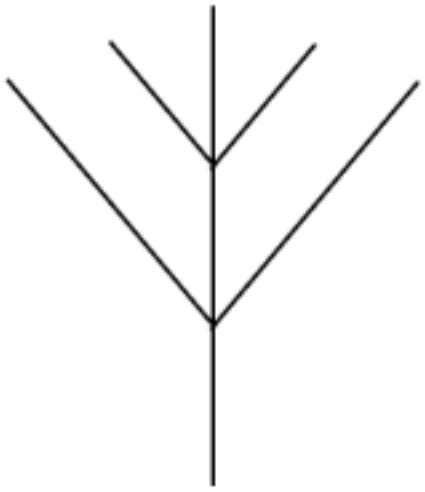


Figure 15.3 One generation L-system

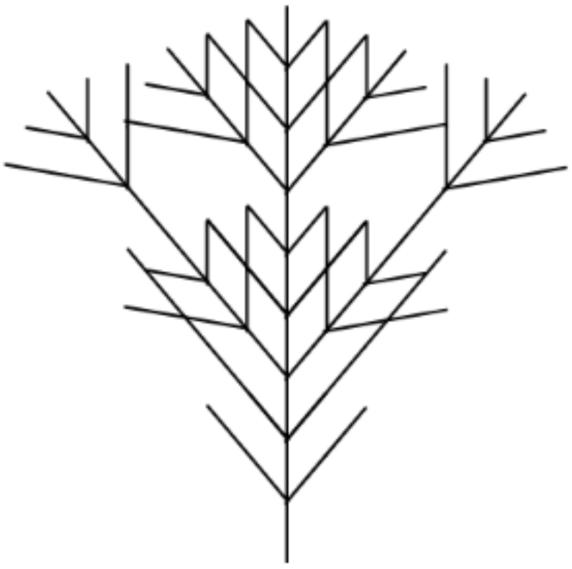


Figure 15.4 Two generations L-system

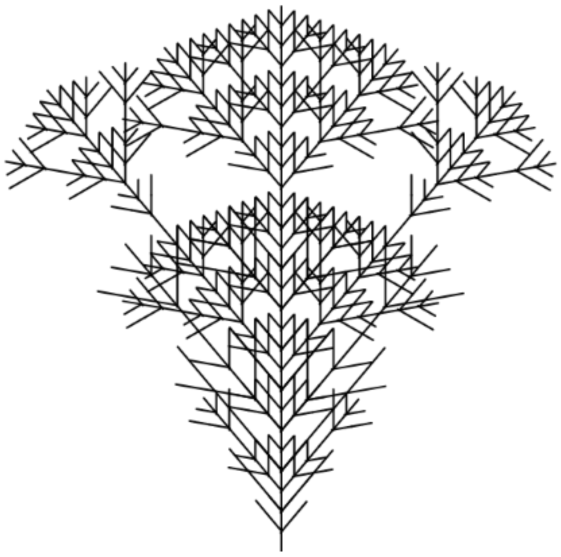


Figure 15.5 Three generations L-system

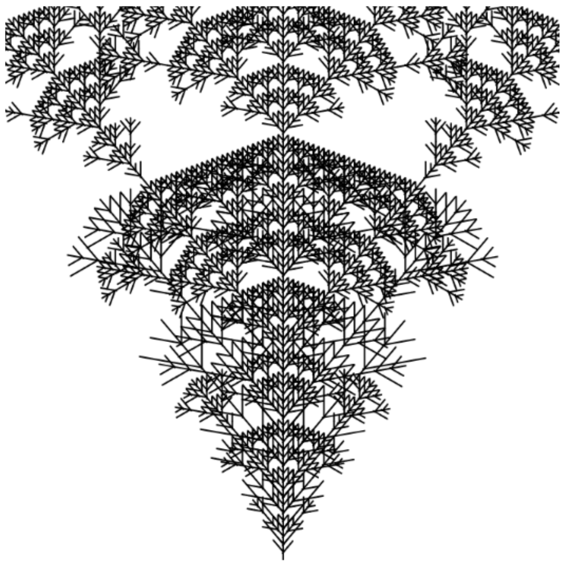
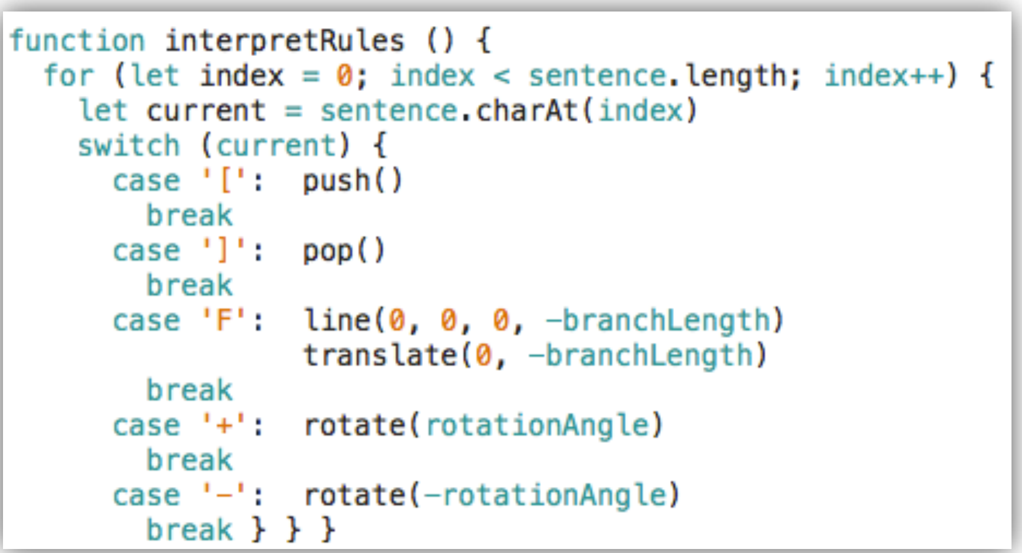


Figure 15.6 Four generations L-system

You can surely go further in generation process as many generations as you want as long as it makes sense to you.

<u>Code</u>

Basically, the code is exactly as the previous one, but the main difference is where we interpret the generated sentences.



```
function interpretRules () {
  for (let index = 0; index < sentence.length; index++) {
    let current = sentence.charAt(index)
    switch (current) {
      case '[':  push()
        break
      case ']':  pop()
        break
      case 'F':  line(0, 0, 0, -branchLength)
                 translate(0, -branchLength)
        break
      case '+':  rotate(rotationAngle)
        break
      case '-':  rotate(-rotationAngle)
        break } } }
```

Discussion

There exist a set of pre-tested rules that generate some well-known patterns **[41]**. But you can anytime play with the ruleset, and change the axiom and the alphabet. You will eventually end up with your own patterns. Congratulations!

2.4. Barnsley Fern

This concept was introduced by the British mathematician Michael Barnsley in 1993 **[42]**. There are four different transformations that should be used in order to end up with Barnsley fern. Each one has the following formula **[42]**:

$$f(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

The constants in this equation a, b, c, d, e, f represents the coefficients of the equation. There is another important variable that is denoted as p represents the probability for each transformation to be chosen. The values Barnsley proposed are listed in the following table:

Transformation number	A	b	c	d	E	f	P
1	0	0	0	0.16	0	0	0.01
2	0.85	0.04	-0.04	0.85	0	1.6	0.85
3	0.20	-0.26	0.23	0.22	0	1.6	0.07
4	-0.15	0.28	0.26	0.24	0	0.44	0.07

Table 9 Coefficient values in Barnsley Fern affine transformations

All the numbers in this table have a meaning. It represents rotation, scaling and shifting on y-axis (f). Moreover, p represents the weight (probability) which means how important the rule is. All the weights add up to one.

The way this algorithm works can be described as follows **[36]**:

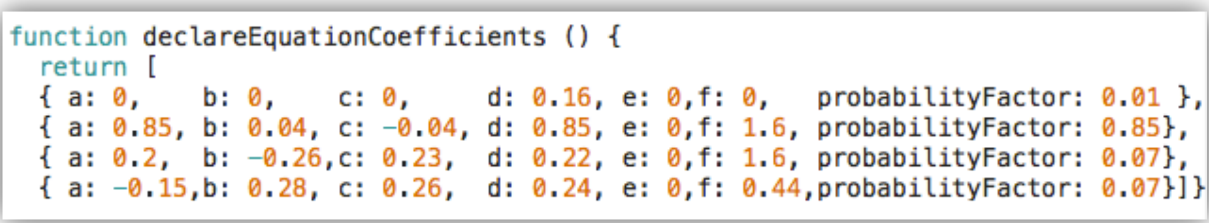
1. starts with a single random point inside a 2D space (canvas in our case).
2. Choose randomly one of four options on each iteration. More precisely, the random in this algorithm is *weighted* according to predefined constants.
3. Depending on the choice from step 2, a specific transformation applies to the chosen point using a 2D transformation matrix.
4. Draw the point after applying the correct transformation.
5. Check the termination condition:
    - If it holds, Go to step 6.
    - Otherwise, Go to step 2.
6. Terminate the process.

<u>Implementation</u>

Let us implement this algorithm in 2D space. Firstly, the values in table-9- will be used without any change. Then we will play around with those values and check which patterns we can come up with.

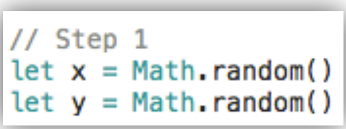
Code

The coding process is just step by step implementation of the above steps. First of all, we declare all the constants that are listed in the table above:



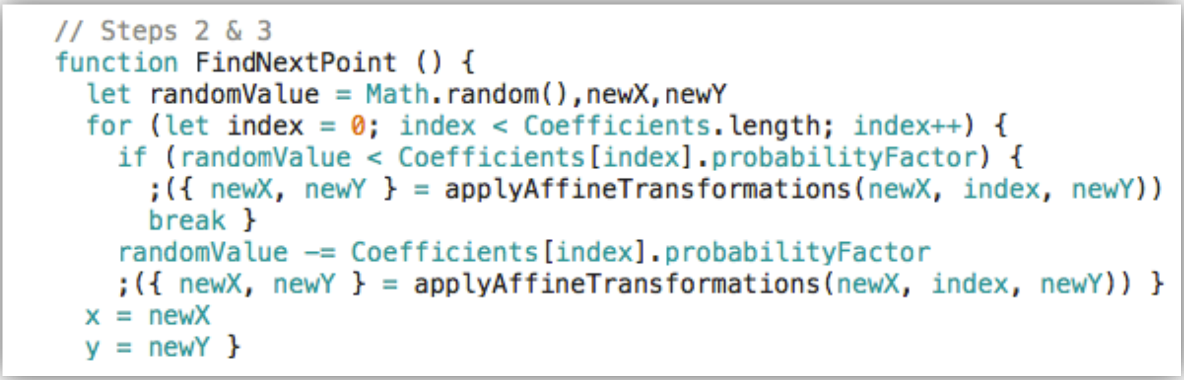
```
function declareEquationCoefficients () {
  return [
  { a: 0,     b: 0,     c: 0,      d: 0.16, e: 0,f: 0,   probabilityFactor: 0.01 },
  { a: 0.85, b: 0.04, c: -0.04, d: 0.85, e: 0,f: 1.6, probabilityFactor: 0.85},
  { a: 0.2,  b: -0.26,c: 0.23,  d: 0.22, e: 0,f: 1.6, probabilityFactor: 0.07},
  { a: -0.15,b: 0.28, c: 0.26,  d: 0.24, e: 0,f: 0.44,probabilityFactor: 0.07}]}
```

Then we create a random point inside the canvas

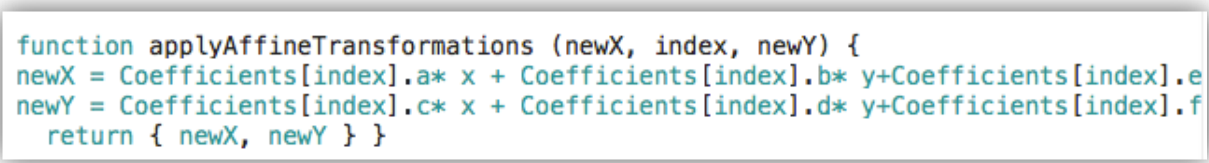


```
// Step 1
let x = Math.random()
let y = Math.random()
```

Then we opt an option from the four available options, and apply the transformation.

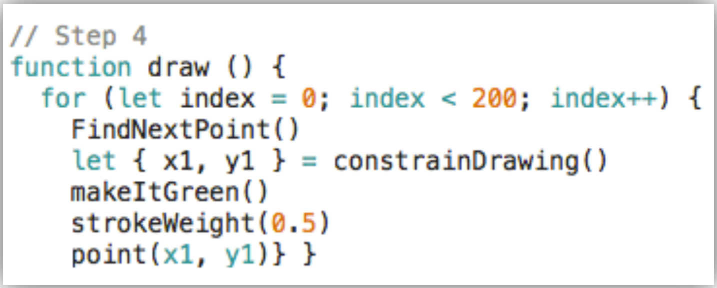


```
  // Steps 2 & 3
  function FindNextPoint () {
    let randomValue = Math.random(),newX,newY
    for (let index = 0; index < Coefficients.length; index++) {
      if (randomValue < Coefficients[index].probabilityFactor) {
        ;({ newX, newY } = applyAffineTransformations(newX, index, newY))
        break }
      randomValue -= Coefficients[index].probabilityFactor
      ;({ newX, newY } = applyAffineTransformations(newX, index, newY)) }
    x = newX
    y = newY }
```



```
function applyAffineTransformations (newX, index, newY) {
newX = Coefficients[index].a* x + Coefficients[index].b* y+Coefficients[index].e
newY = Coefficients[index].c* x + Coefficients[index].d* y+Coefficients[index].f
  return { newX, newY } }
```

After that we draw the point :



```
// Step 4
function draw () {
  for (let index = 0; index < 200; index++) {
    FindNextPoint()
    let { x1, y1 } = constrainDrawing()
    makeItGreen()
    strokeWeight(0.5)
    point(x1, y1)} }
```

Running the code, we get the following gorgeous fern:

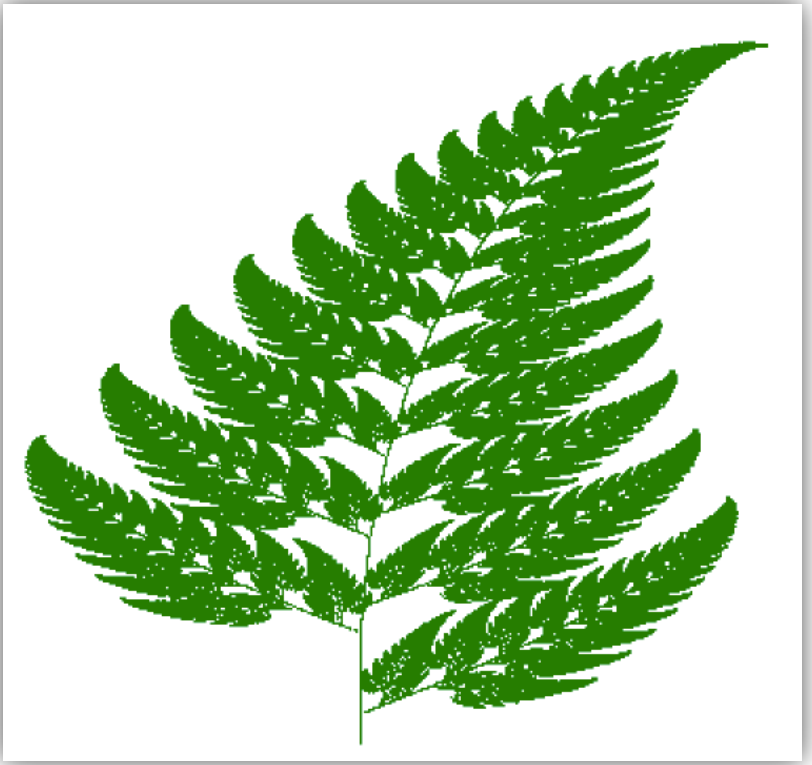


Figure 16.1 Barnsley Fern -1-.

Discussion

You may wonder (and I did) what would happen if some of the values in table-9- are changed. The answer is very simple. You will get a new pattern each time you make a change. Occasionally, very weird patterns may show up. Let us alter some of those values a little bit, and see which patterns appear. For the sake of simplicity, we will make all the changes in just one row of the table. Let us choose the second line of the table 10, and apply the following changes on a, c, e, probabilityFactor :

Coefficient	Old value	New value	Result
a	0.85	0.5	Figure 16.2
c	- 0.04	- 0.7	Figure 16.3
e	0	- 0.5	Figure 16.4
probabilityFactor	0.85	0.5	Figure 16.5

Table 10 Tested values for coefficients a and c

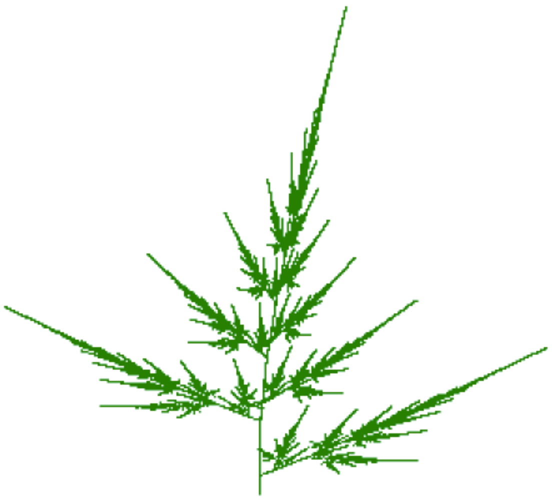


Figure 16.2 Barnsley Fern -2-

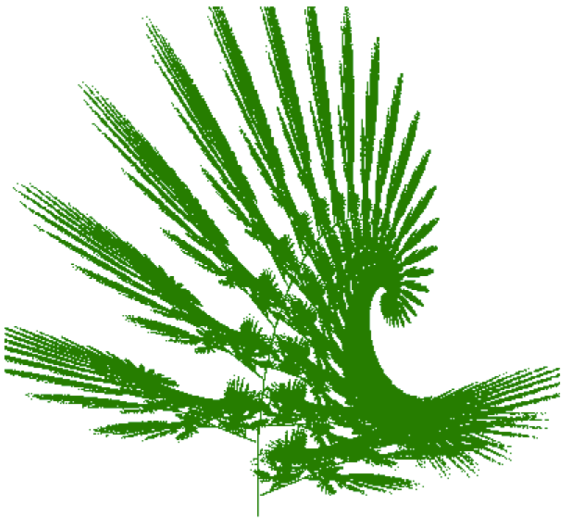


Figure 16.3 Barnsley Fern -3-

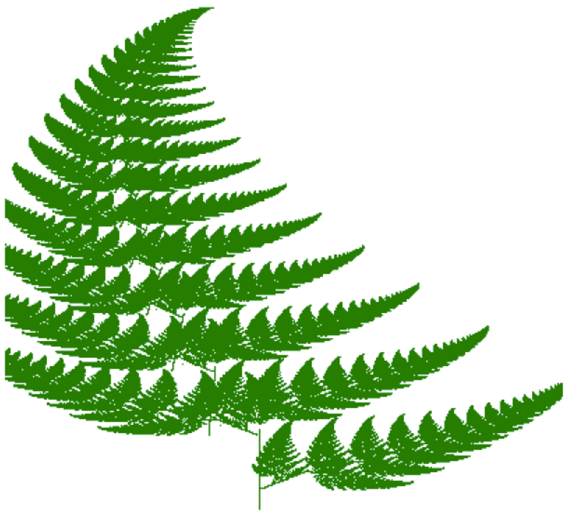


Figure 16.4 Barnsley Fern -4-

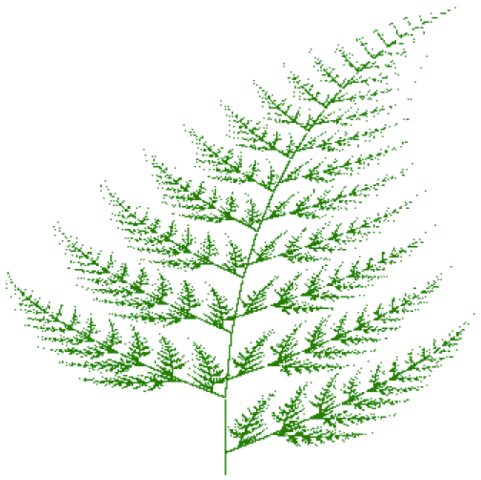


Figure 16.5 Barnsley Fern -5-

Chapter 7

Partitioning Space

Whenever you use PCG, you do not absolutely expect to get an arranged, partitioned space as an output from your generator after first attempt. The content, in most cases, is normally an open and unpartitioned space. This content needs to be polished several times by split it into a set of regions. For instance, the generator may generate a map haphazardly, and then you split it into political areas to give it a meaning somehow or another.

1. Binary Space Partitioning(BSP)

This term was developed in the context of 3D computer graphics **[43].** BSP is just a fancy way of saying you cut something in half a bunch of times. BSP at its core is a very simple concept. You can use BSP in a couple of different ways. You simply take a space, and cut it in half a certain number of times. This process stops when a predefined criterion holds. The criterion could be either a specific number of splits, a specific size for the resulting spaces, or something else. Depending on the size of the space, and the number of times you are going to split it, you can determine how many rooms you have. It is basically an exponential function 2^x. So, if you split the space one time, that is, x=1, then you will get two new, smaller, spaces. If you split the space twice, that is x=2, you will get 4 rooms, and so on and so forth. This means you are rapidly going to end up with a lot of smaller spaces after a few times of splitting.

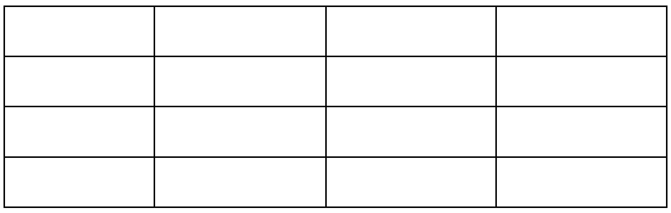


Figure 17.1 Partitioning the space for x=4

This tool is really powerful because it allows you to create equidistant sort of regions. But in other cases, you may need a bunch of smaller spaces that are not necessarily equal. So, what you might do is to determine a range of possible sizes. For example, you can say that you want to split a room in two, and you want it to be anywhere from 30-70%. In other words, you do not want the room to be smaller than 30%, and likewise not above 70%. So, you can constrain your generated rooms to some required values. For example, if the random number generator produces 35% first time, then you will get two rooms. Now, let us say that for the first room the generator gives 50%, and for the second room it gives 65%. This way you can generate a number of rooms that have different sizes (figure 17-2).

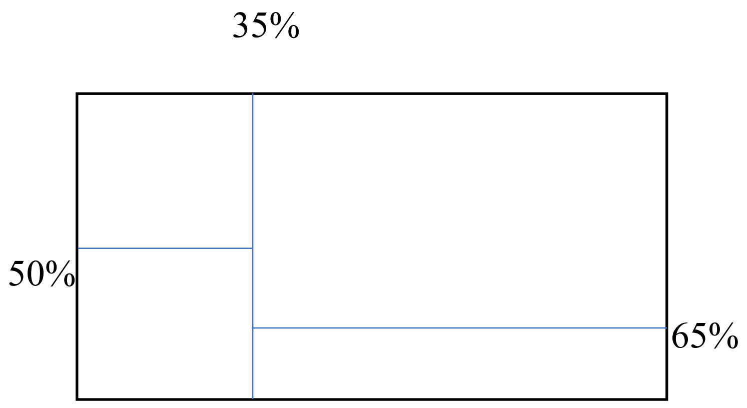


Figure 17.2 Generated rooms with different sizes

2. Voronoi Diagram

Voronoi diagram is named after Georgy Voronoi when he introduced it for the first time in 1908 **[44].** Voronoi diagrams are arguably the most well-known concept in computational geometry, and have many useful applications in practical cases **[45]**. The basic idea is that instead of being concerned with the initial data points that exist in the data set, we are actually more concern about the data points that surround them. An important concept is Voronoi region which is basically a single region that surrounds one of the input data points. A single Voronoi region defines all the points outside the original data set which are closer to that data point (the one that is centered in the region) than to any other data point in the data set. So, you basically define regions in space that are closer to your input points than to any of the other input points. So, it is the sort of the closest neighbor type of analysis. All Voronoi regions are always convex without exception **[46]**. That is, polygons in 2D, and polyhedral in 3D. For example, let us say that you are hopefully a businessman, and you have a plan to start a new business in your city. Your business is basically to open a number of groceries that hold your name. Using a Voronoi diagrams jargon, we can say that each point in the data set represents a grocery store in the city. Likewise, each Voronoi region contains the residences that are closest to a particular grocery store. In order to succeed in your business, your groceries should be located in the city in a way that guarantees a sort of balance. In other words, the residences should be used each of your groceries approximately in an equal ratio.

Implementation

To exemplify this algorithm, let us implement it to generate a diagram for a data set consists of 100 input points (sites). The way I implemented this algorithm can be described as follows:

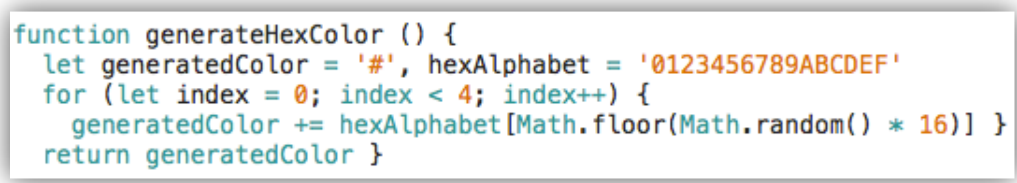
- Create a collection of points (100 points in our case).
- Distribute the points randomly in a 2D space.
- For each pixel, figure out which point is the closest to it.
- Color the pixel based on that.

More details in the code section below.

Code

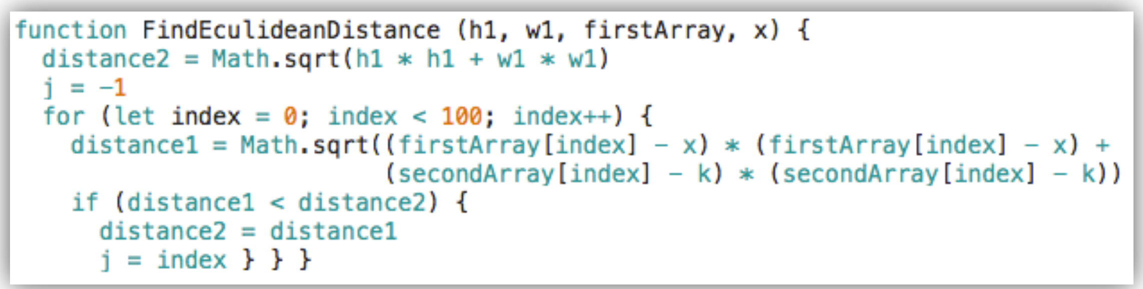
The code for this part is somewhat long, and therefore it would be better to go through it step by step.

To provide a better look to the generated diagram, the Voronoi regions will be colored randomly. A function generates a hex-color is responsible for that:



```
function generateHexColor () {
  let generatedColor = '#', hexAlphabet = '0123456789ABCDEF'
  for (let index = 0; index < 4; index++) {
    generatedColor += hexAlphabet[Math.floor(Math.random() * 16)] }
  return generatedColor }
```

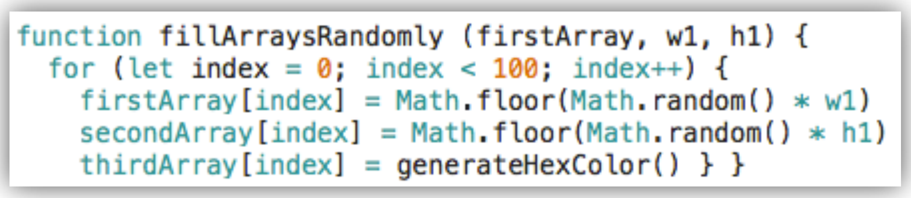
The core idea behind Voronoi diagram is to calculate the distance between each point in the data set and its neighbors. Therefore, a function to calculate the Euclidean distance is a demand.



```
function FindEculideanDistance (h1, w1, firstArray, x) {
  distance2 = Math.sqrt(h1 * h1 + w1 * w1)
  j = -1
  for (let index = 0; index < 100; index++) {
    distance1 = Math.sqrt((firstArray[index] - x) * (firstArray[index] - x) +
                          (secondArray[index] - k) * (secondArray[index] - k))
    if (distance1 < distance2) {
      distance2 = distance1
      j = index } } }
```

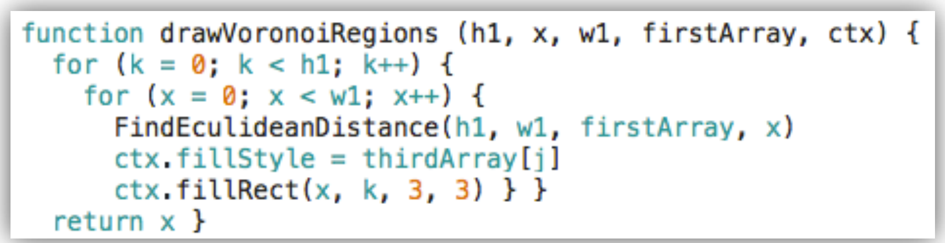
All the points in the data set are distributed randomly in the space (canvas in our case). The coordinates for each point will be chosen from two arrays. Both arrays are filled with random values. We need another randomly filled array to be used as a source to color every Voronoi region.

The function that is responsible for that is the following one:



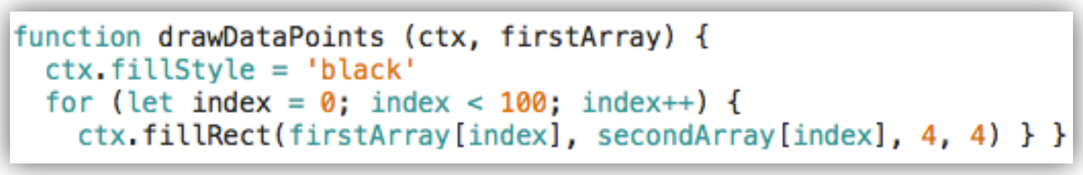
```
function fillArraysRandomly (firstArray, w1, h1) {
  for (let index = 0; index < 100; index++) {
    firstArray[index] = Math.floor(Math.random() * w1)
    secondArray[index] = Math.floor(Math.random() * h1)
    thirdArray[index] = generateHexColor() } }
```

To draw the diagram, we need two different functions. The first one draws Voronoi regions.



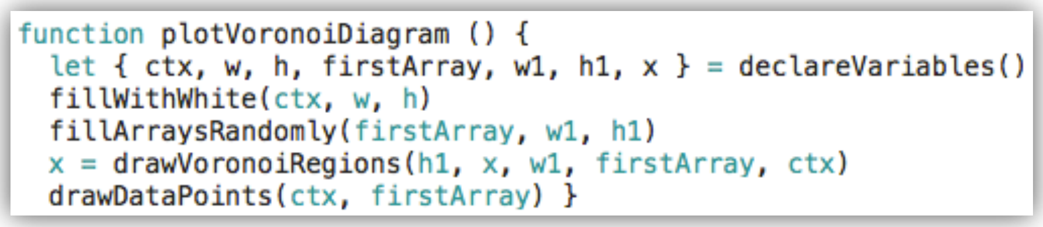
```
function drawVoronoiRegions (h1, x, w1, firstArray, ctx) {
  for (k = 0; k < h1; k++) {
    for (x = 0; x < w1; x++) {
      FindEculideanDistance(h1, w1, firstArray, x)
      ctx.fillStyle = thirdArray[j]
      ctx.fillRect(x, k, 3, 3) } }
  return x }
```

And the second one draws all the points in the data set (100 points in our case).



```
function drawDataPoints (ctx, firstArray) {
  ctx.fillStyle = 'black'
  for (let index = 0; index < 100; index++) {
    ctx.fillRect(firstArray[index], secondArray[index], 4, 4) } }
```

Using both of them, we can generate the whole diagram.



```
function plotVoronoiDiagram () {
  let { ctx, w, h, firstArray, w1, h1, x } = declareVariables()
  fillWithWhite(ctx, w, h)
  fillArraysRandomly(firstArray, w1, h1)
  x = drawVoronoiRegions(h1, x, w1, firstArray, ctx)
  drawDataPoints(ctx, firstArray) }
```

Running this algorithm will generate the following diagram.

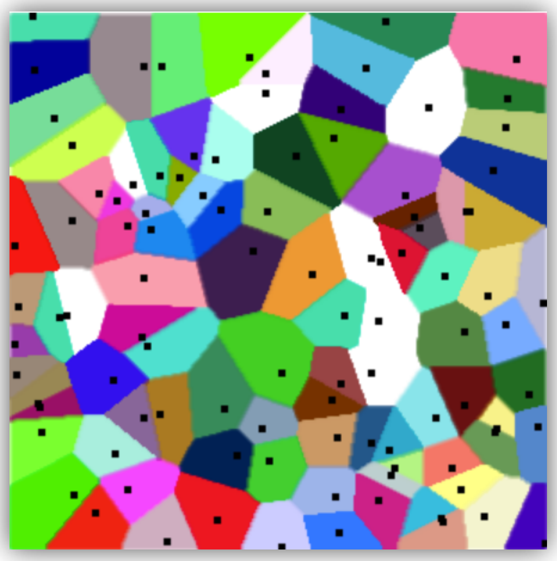


Figure 18.1 Voronoi diagram -1-

Running this generator again, it generates another diagram, and so on:

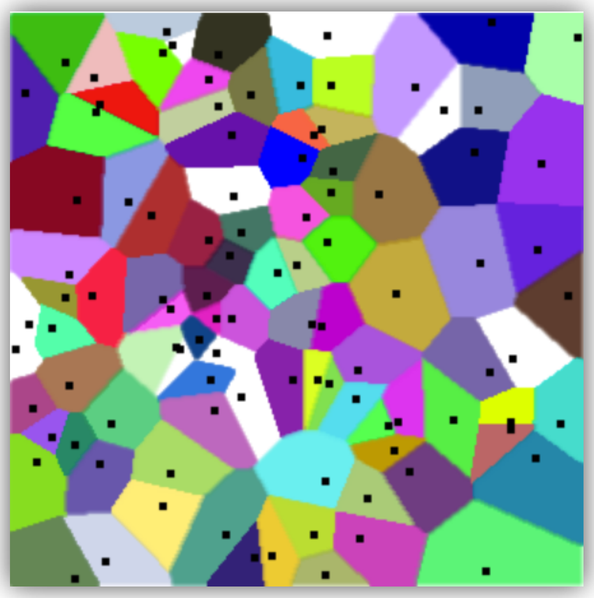


Figure 18.2 Voronoi diagram -2-

<u>Note</u>

It is worthwhile to mention here that Voronoi diagrams is a very broad subject, and a whole master thesis can be done only to investigate this important concept. In my case, however, I would like to admit that I have barely scratched the surface of this topic. This would be a promising future research for me, or for any other ambitious student. Highly recommended topic!

Chapter 8

Maze Generation

As a matter of fact, a master thesis about procedural generation would be not valuable without raising maze generation for discussion. There exist many different algorithms to generate mazes procedurally. To demonstrate this concept clearly, I have chosen two different algorithms to be demonstrated and implemented. They are recursive backtracker algorithm and reaction diffusion algorithm.

1. Recursive Backtracker[RB]

In fact, this algorithm is one of the easiest maze generation algorithms. It is used by another algorithm called depth-first search (DFS) in order to generate a maze. RB normally uses a stack as a data structure in its implementation. Its downside, however, that it leads to a few dead ends. That means it generates relatively less-complicated mazes comparing to other maze generation algorithms. In fact, if you want to build a game the includes mazes, then the following statement always holds: the more complicated and intricate the mazes are, the more interesting and enjoyable your game is. The following steps describe how this algorithm works as proposed in **[47]** :

1- Choose an initial cell within the grid.

2- Check if its neighbors have been visited before. If not, continue to step 2.1. Otherwise, go to step 3.

  2.1- Pick random neighbor.

  2.2- Put the neighbor on the stack.

  2.3- Remove the wall between the current cell and the chosen neighbor.

  2.4- Mark a path from the current cell to the selected neighbor.

  2.5- Go to step 2, and repeat the process

3- Mark the current cell as visited, and pop it off the stack as long as the stack is not empty. Otherwise, go to step 5.

4- Go to the cell at the top of the stack and go to step 2.

5- Stop when the stack is empty.

Implementation

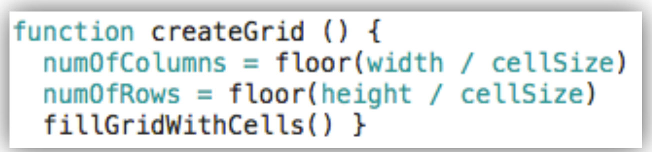
In my suggested implementation, the algorithm is going to figure out which walls should be removed to generate a nice maze pattern eventually. Each cell in the grid represents an object, and it should mainly know two things:

- Its location on the grid. That is, what is its row and column coordinates?
- The situation of its surrounding walls. That is, is it opened or closed?

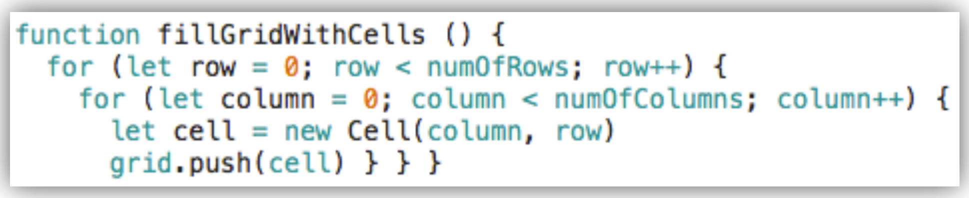
Code

Only the most important parts of the code will be demonstrated. The complete code can be found in the appendix. The coding process can be divided into the following steps:

1. Create a 2D grid consists of a number of cells.

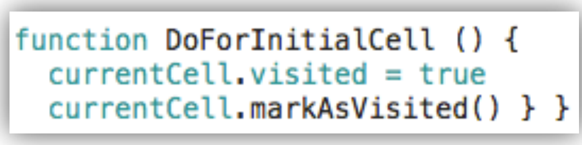


```
function createGrid () {
  numOfColumns = floor(width / cellSize)
  numOfRows = floor(height / cellSize)
  fillGridWithCells() }
```



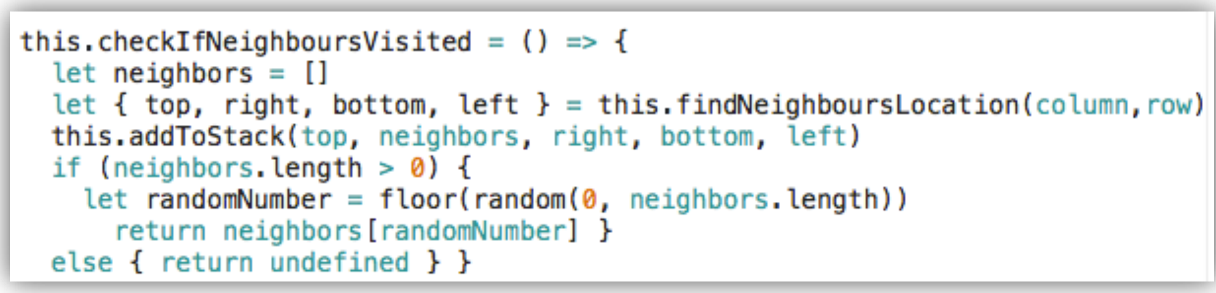
```
function fillGridWithCells () {
  for (let row = 0; row < numOfRows; row++) {
    for (let column = 0; column < numOfColumns; column++) {
      let cell = new Cell(column, row)
      grid.push(cell) } } }
```

2. Choose an initial cell to start the algorithm.



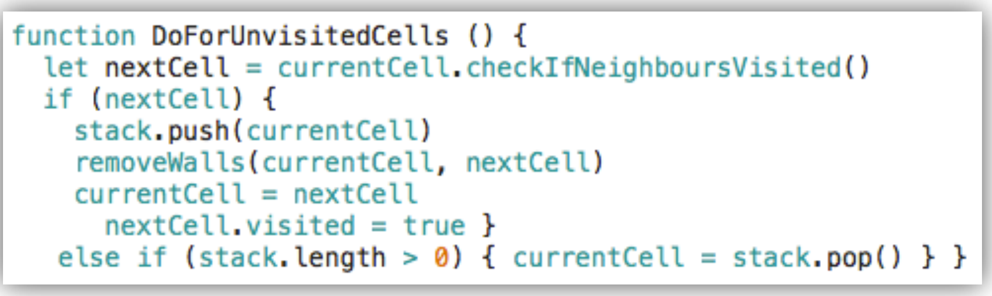
```
function DoForInitialCell () {
  currentCell.visited = true
  currentCell.markAsVisited() } }
```

Then check its neighbors to see if there are any unvisited cells in the neighborhood. If that is the case, choose randomly one neighbor. This represents the step 2.1 from the steps above.



```
this.checkIfNeighboursVisited = () => {
  let neighbors = []
  let { top, right, bottom, left } = this.findNeighboursLocation(column,row)
  this.addToStack(top, neighbors, right, bottom, left)
  if (neighbors.length > 0) {
    let randomNumber = floor(random(0, neighbors.length))
      return neighbors[randomNumber] }
  else { return undefined } }
```

3. For any unvisited neighbor, perform the steps 2.2 & 2.3 & 2.5 & 3.

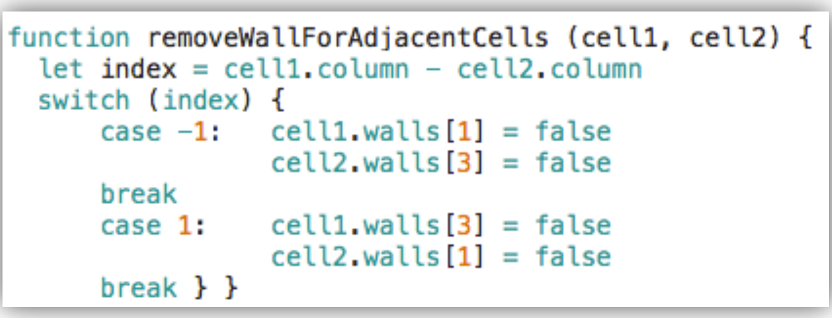


```
function DoForUnvisitedCells () {
  let nextCell = currentCell.checkIfNeighboursVisited()
  if (nextCell) {
    stack.push(currentCell)
    removeWalls(currentCell, nextCell)
    currentCell = nextCell
      nextCell.visited = true }
   else if (stack.length > 0) { currentCell = stack.pop() } }
```

<u>Note</u>:

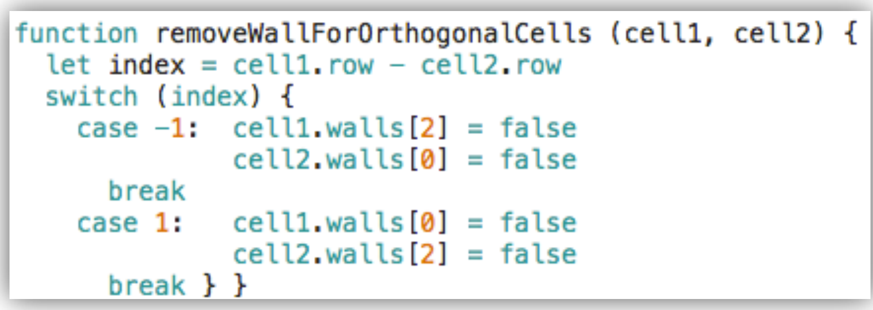
In order to end up with a grid similar to a maze, the algorithm should continuously remove the walls between adjacent cells. Depending on how the cells are located, we can distinguish between two different wall-removing processes:

- For adjacent cells, a vertical wall that separates them will be removed:



```
function removeWallForAdjacentCells (cell1, cell2) {
  let index = cell1.column - cell2.column
  switch (index) {
      case -1:    cell1.walls[1] = false
                  cell2.walls[3] = false
      break
      case 1:     cell1.walls[3] = false
                  cell2.walls[1] = false
      break } }
```

- For orthogonal cells (the cells on the top of each other's), a horizontal wall that separates them will be removed:



```
function removeWallForOrthogonalCells (cell1, cell2) {
  let index = cell1.row - cell2.row
  switch (index) {
    case -1:  cell1.walls[2] = false
              cell2.walls[0] = false
      break
    case 1:   cell1.walls[0] = false
              cell2.walls[2] = false
      break } }
```

Running this algorithm will generate the following nice maze.

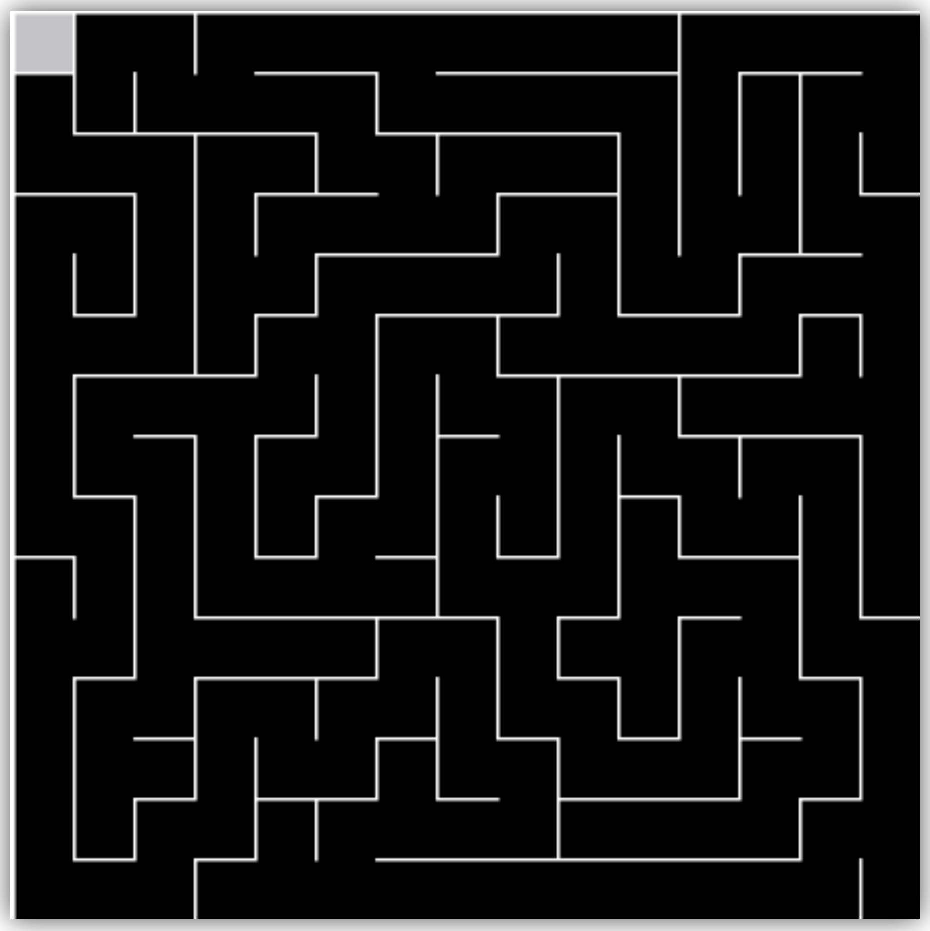


Figure 19 Generated maze using recursive backtracker algorithm

2. Reaction Diffusion Algorithm [RDA]

Reaction diffusion systems can be implemented in different ways. The way I have chosen is called *Gray Scott Model* of reaction diffusion **[48]**. This algorithm simulates the reaction and diffusion between those two chemicals A and B on a two-dimensional grid. In our case, we are going to use a JavaScript canvas to display the generated content. The canvas is going to be filled with chemical B, and then chemical A will be added at a specific "feed" rate into the canvas. This causes the reaction to start immediately. There are two different phases in this algorithm. The reaction phase in which chemical A turns into chemical B. The diffusion phase in which chemical B is removed at a specific "kill" rate. This system can be described by the equations underneath. There are some key constants in those equations. The first constant is the feed rate which represents the ratio of adding chemical A to the reaction. The second constant is the kill rate which represents the ration of removing chemical B from the system. Each cell on the grid contains certain amount of chemicals A and B. This amount is represented by a value between 0 and 1. If the value equals 1, there is a lot of chemical A. if it is 0, there is none. This value is used to set the color for a pixel on the canvas. This algorithm is built on the following two equations **[49]**:

$$\frac{\delta A}{\delta t} = D_A \nabla^2 A - AB^2 + f\,(1 - A)$$

$$\frac{\delta B}{\delta t} = D_B \nabla^2 B + AB^2 - (k + f)B$$

At first glance, those equations look intimidating, but they are actually mathematically simple. The new values of A and B can be found depending on:

- The previous known values of A and B.
- The values of its neighbors.
- Some other given constants.

For each cell in the grid, a convolution of 3x3 matrix should be applied. The convolution in our context means that for any cell in the grid, each of its neighbors is multiplied by a specific value called weight.

Implementation

This algorithm will be implemented in a 2D space. Only the most important parts of the code will be highlighted. The whole code can be found in the appendix.

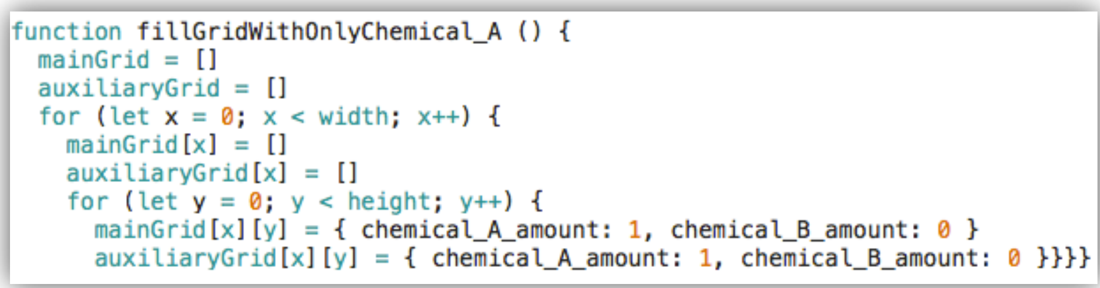
Code

First of all, I would like to mention that all the values in my code are exactly as proposed in **[49]**.

Two grids will be used, a main grid and an auxiliary grid. Each cell in the grid contains some amount of chemical A or chemical B, or potentially a combination of both A and B. The auxiliary grid is useful when we moved from one generation to another. The new calculated values are saved temporarily in the auxiliary grid, and then moved to the main grid. This process occurs at each movement from one generation to another.

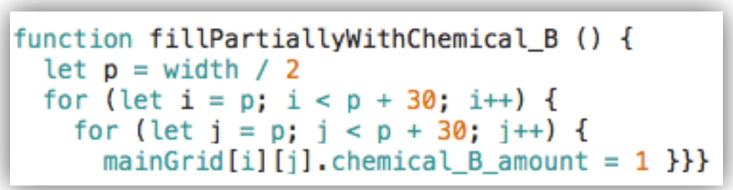
The coding process can be divided into the following steps:

1- Fill the entire main grid with only chemical A.



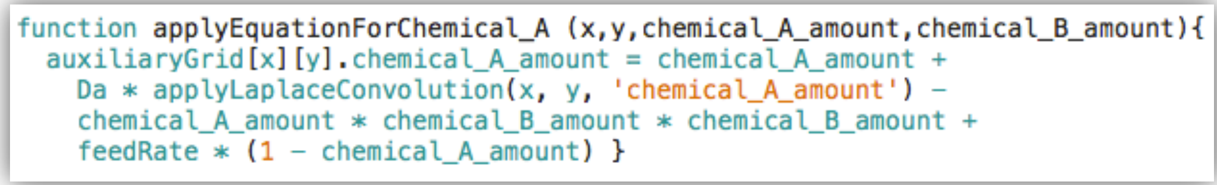
```
function fillGridWithOnlyChemical_A () {
  mainGrid = []
  auxiliaryGrid = []
  for (let x = 0; x < width; x++) {
    mainGrid[x] = []
    auxiliaryGrid[x] = []
    for (let y = 0; y < height; y++) {
      mainGrid[x][y] = { chemical_A_amount: 1, chemical_B_amount: 0 }
      auxiliaryGrid[x][y] = { chemical_A_amount: 1, chemical_B_amount: 0 }}}}
```

2- Insert chemical B into a small area of the main grid (in my implementation, chemical B is inserted into only 30 cells).



```
function fillPartiallyWithChemical_B () {
  let p = width / 2
  for (let i = p; i < p + 30; i++) {
    for (let j = p; j < p + 30; j++) {
      mainGrid[i][j].chemical_B_amount = 1 }}}
```

3- After adding chemical B, the reaction starts. There is a specific equation that should be applied for every chemical A.



```
function applyEquationForChemical_A (x,y,chemical_A_amount,chemical_B_amount){
  auxiliaryGrid[x][y].chemical_A_amount = chemical_A_amount +
    Da * applyLaplaceConvolution(x, y, 'chemical_A_amount') -
    chemical_A_amount * chemical_B_amount * chemical_B_amount +
    feedRate * (1 - chemical_A_amount) }
```

And another reaction equation for every chemical B.

```
function applyEquationForChemical_B (x,y,chemical_B_amount,chemical_A_amount){
  auxiliaryGrid[x][y].chemical_B_amount = chemical_B_amount +
    Db * applyLaplaceConvolution(x, y, 'chemical_B_amount') +
    chemical_A_amount * chemical_B_amount * chemical_B_amount -
    (killRate + feedRate) * chemical_B_amount }
```

4- Once the reaction occurs, it causes the previously amounts of both chemical A and B to be changed. The new values depend on:

- Some given constants [Da, Db, killRate, feedRate] **[49]**.
- The current amount of both chemical A and B.
- A Laplacian function that is performed with 3x3 convolution, using the following weights **[49]**:
  (a) -1 for the center cell.
  (b) 0.2 for adjacent neighbors.
  (c) 0.05 for diagonals.

The Laplacian function is performed using the following methods:

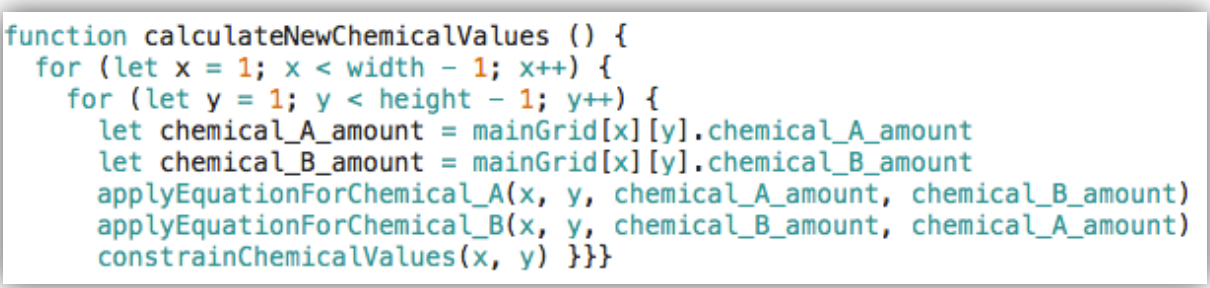
```
function applyLaplaceConvolution (x, y, value) {
  let wieght = 0
  let centerCellWieght = mainGrid[x][y][value] * -1
  wieght += centerCellWieght
  wieght = calculateAdjacentCellsWeight(wieght, x, y, value)
  wieght = calculateDiagonalCellsWeight(wieght, x, y, value)
  return wieght }
```

```
function calculateAdjacentCellsWeight (wieght, x, y, value) {
  wieght += mainGrid[x - 1][y][value] * 0.2
  wieght += mainGrid[x + 1][y][value] * 0.2
  wieght += mainGrid[x][y + 1][value] * 0.2
  wieght += mainGrid[x][y - 1][value] * 0.2
  return wieght }
```

And this one:

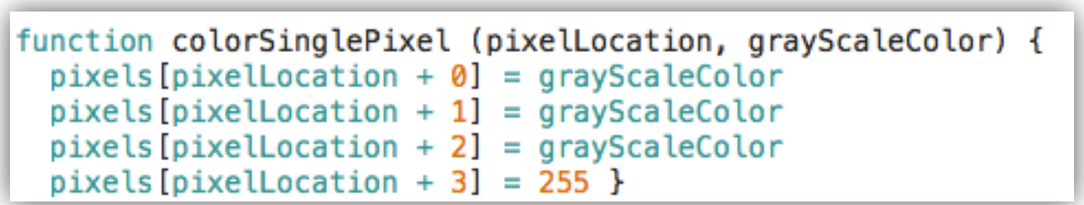
```
function calculateDiagonalCellsWeight (wieght, x, y, value) {
  wieght += mainGrid[x - 1][y - 1][value] * 0.05
  wieght += mainGrid[x - 1][y + 1][value] * 0.05
  wieght += mainGrid[x + 1][y - 1][value] * 0.05
  wieght += mainGrid[x + 1][y + 1][value] * 0.05
  return wieght }
```

5- The applying the Laplacian function, the new values for both A and B can be found using the following function:



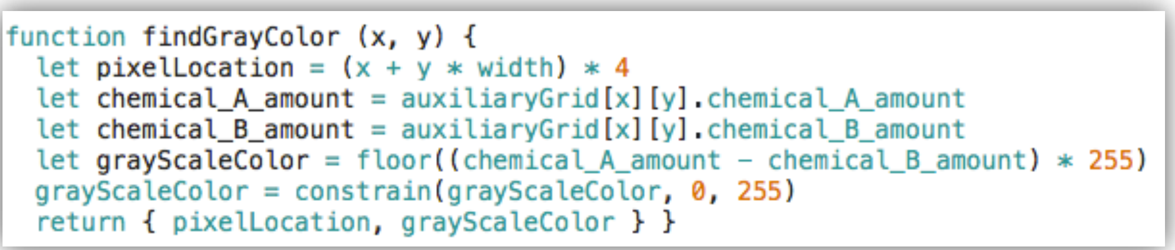
```
function calculateNewChemicalValues () {
  for (let x = 1; x < width - 1; x++) {
    for (let y = 1; y < height - 1; y++) {
      let chemical_A_amount = mainGrid[x][y].chemical_A_amount
      let chemical_B_amount = mainGrid[x][y].chemical_B_amount
      applyEquationForChemical_A(x, y, chemical_A_amount, chemical_B_amount)
      applyEquationForChemical_B(x, y, chemical_B_amount, chemical_A_amount)
      constrainChemicalValues(x, y) }}}
```

6- Finally, in order to draw the maze, a color is attached to each cell (pixel) in the main grid.



```
function colorSinglePixel (pixelLocation, grayScaleColor) {
  pixels[pixelLocation + 0] = grayScaleColor
  pixels[pixelLocation + 1] = grayScaleColor
  pixels[pixelLocation + 2] = grayScaleColor
  pixels[pixelLocation + 3] = 255 }
```

By the way, the color is calculated depending on the values of both chemical A and B:



```
function findGrayColor (x, y) {
  let pixelLocation = (x + y * width) * 4
  let chemical_A_amount = auxiliaryGrid[x][y].chemical_A_amount
  let chemical_B_amount = auxiliaryGrid[x][y].chemical_B_amount
  let grayScaleColor = floor((chemical_A_amount - chemical_B_amount) * 255)
  grayScaleColor = constrain(grayScaleColor, 0, 255)
  return { pixelLocation, grayScaleColor } }
```

<u>Experiments</u>

In order to test this algorithm, and to see the variety of pattern it can produce, four different experiments will be done. The values in the table will be used. The values in the first two lines are proposed in **[49]**.

DA	DB	feedRate	killRate	Result
1	0.5	0.0545	0.062	Figure 20.1
1	0.5	0.0367	0.0649	Figure 20.2
1	0.5	0.045	0.0625	Figure 20.3
0.4	0.2	0.045	0.0625	Figure 20.4

Table 11 Tested values in RDA

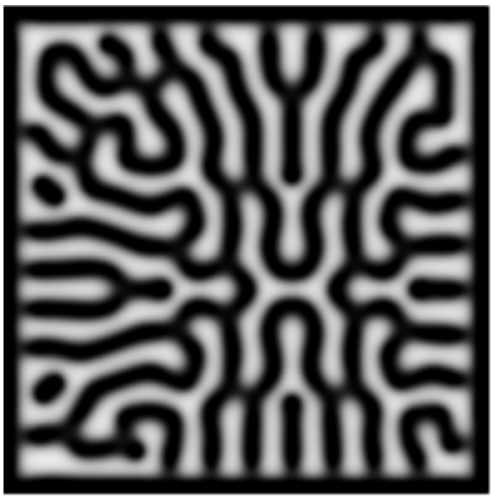


Figure 20.1 RDA-1

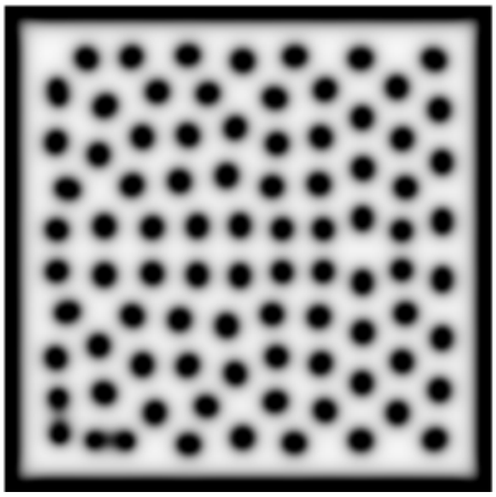


Figure 20.2 RDA-2

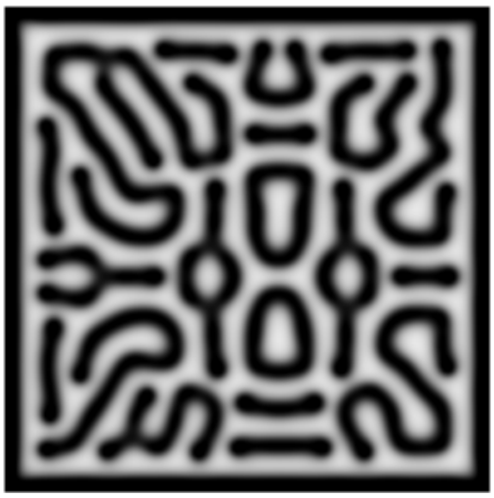


Figure 20.3 RDA-3

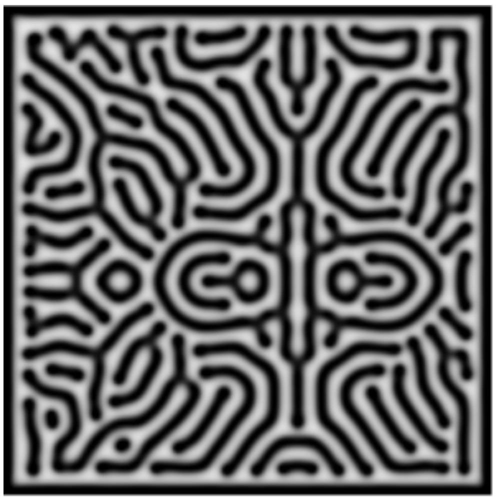


Figure 20.4 RDA-4

Chapter 9

Procedural Flowers

If you would like to generate various flower patterns, there are a lot of different algorithms for this specific purpose. The algorithm I have chosen is built completely around the concept of polar coordinates. We all know that if (x, y) are the coordinates in cartesian coordinates system, and (r, θ) the coordinates in a polar coordinates system, then as it is proposed in**[50]**:

$$x = r\cos(\mathrm{k}\theta)$$

$$y = r\sin(\mathrm{k}\theta)$$

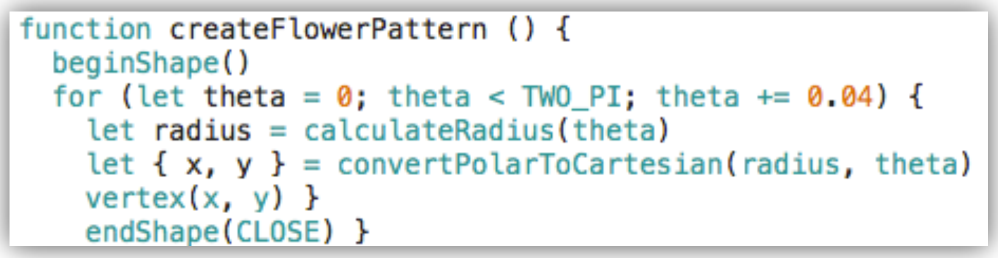
Where k is a constant.

Implementation

All the experiments will be done in a 2D space.

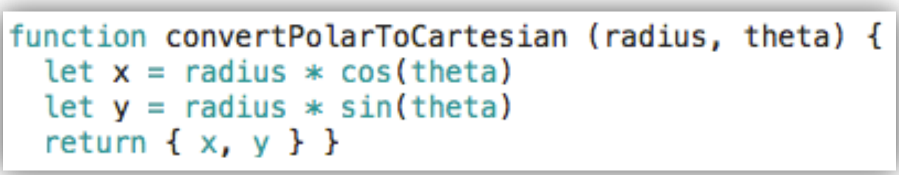
Code

The following function is responsible for creating a specific pattern each time we change the value of the constant k.

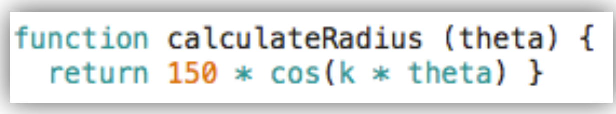


```
function createFlowerPattern () {
  beginShape()
  for (let theta = 0; theta < TWO_PI; theta += 0.04) {
    let radius = calculateRadius(theta)
    let { x, y } = convertPolarToCartesian(radius, theta)
    vertex(x, y) }
    endShape(CLOSE) }
```

This function calls another two functions:



```
function convertPolarToCartesian (radius, theta) {
  let x = radius * cos(theta)
  let y = radius * sin(theta)
  return { x, y } }
```



```
function calculateRadius (theta) {
  return 150 * cos(k * theta) }
```

Note that you are not restricted to use only cos() function as we did in this implementation. You can replace it with any other function. Each time you change the function, you will get a new pattern. For example, try to use sin() function or tan() function instead of cos().
No matter the function you choose, you can get infinite number of different flower patterns by simply changing the value of the constant k. Different values will be tested (table 12).

Value of constant K	**Figure**
0.9	21.1
0.09	21.2
0 8	21.3
1000	21.4

Table 12 Tested values for the constant k

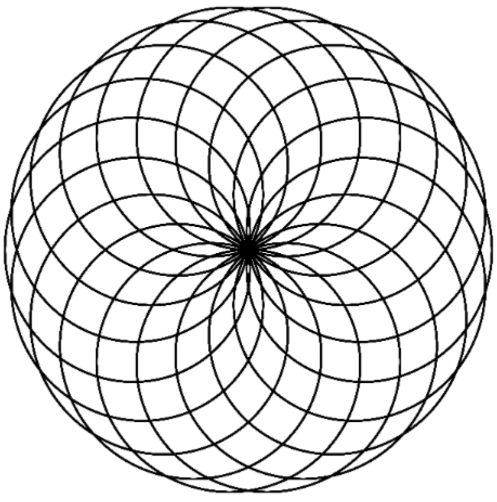


Figure 21.1 Flower pattern -1-

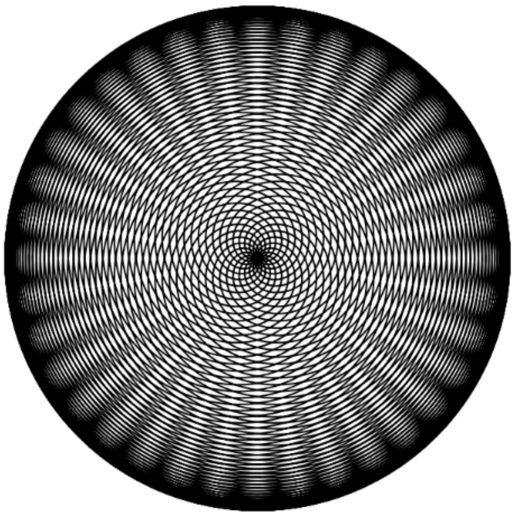


Figure 21.2 Flower pattern -2-

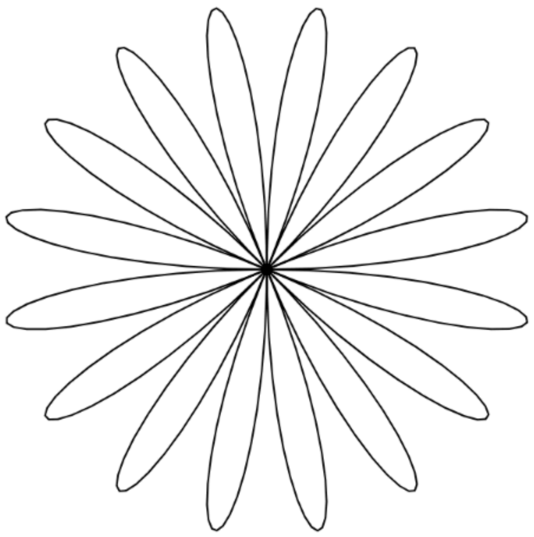


Figure 21.3 Flower pattern -3-

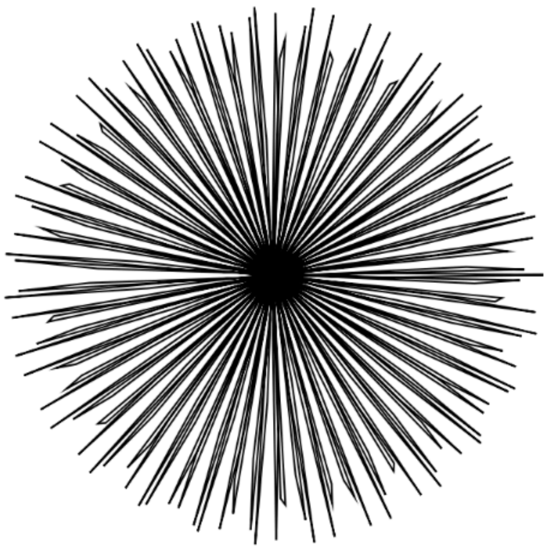


Figure 21.4 Flower pattern -4-

Chapter 10

Procedurally Generated Infinite Terrain

Some notes on the fly:

- This project is built using Unity 2018.3.4f1 version.
- Nowadays, neither Java nor JavaScript can be used with Unity. In fact, Unity supports only C#. Therefore, the code for this project is written completely using C#.

1 A simple & static terrain

If you ask any designer about the most obvious and simple way to create a terrain, the answer most likely would be to use one of the tiling techniques. My answer is no exception, and I will choose tiling as well if you ask me. In fact, tiling is the intuitive way to start with.

1.1 Tiling

I have started the project by randomly picking up an image from Google (figure 22.1), and divided it into a number of equivalent squares. Each square represents a small tile (160 squares in my code) as depicted in figure 22.2. This image represents the texture that will be used to build the terrain.

Figure 22.1 Grass image

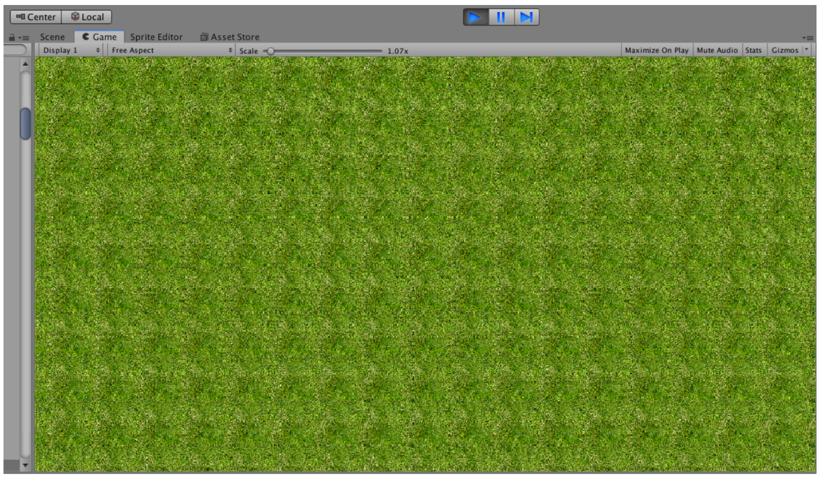


Figure 22.2 Dividing an image into small tiles.

To do that, the code is very straightforward.

```
private void DivideIntoTiles() {
    CoordinateAdjustment = new Vector3(0 - width / 2, 0 - height / 2, 0);
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            Tile = new GameObject();
            Tile.transform.position = new Vector3(x, y, 0) + CoordinateAdjustment;
            RenderTheImage(); } } }
```

1.2 Make it random

Instead of using one texture to build the terrain, using multiple textures makes the terrain certainly looks better. Let us add two new textures to our terrain. The first texture is sand and the second one is rocks. These textures will be tiled before adding them to the terrain. To build the terrain, one tile from the three textures will be randomly chosen at each iteration until the terrain is completely filled with tiles. The generated terrain is a varied terrain that consists of three different textures.

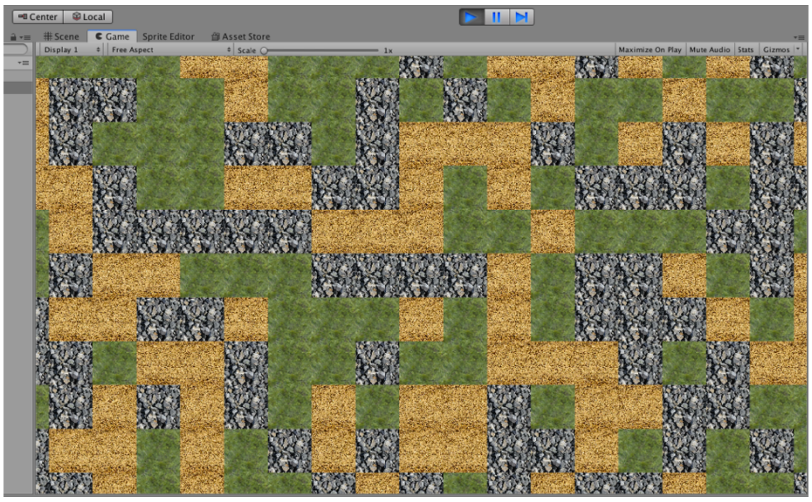


Figure 22.3 Randomly generated terrain using tiling.

Each time we run the generator, a completely different terrain will show up. That is due to the randomness that dominates our generator so far.

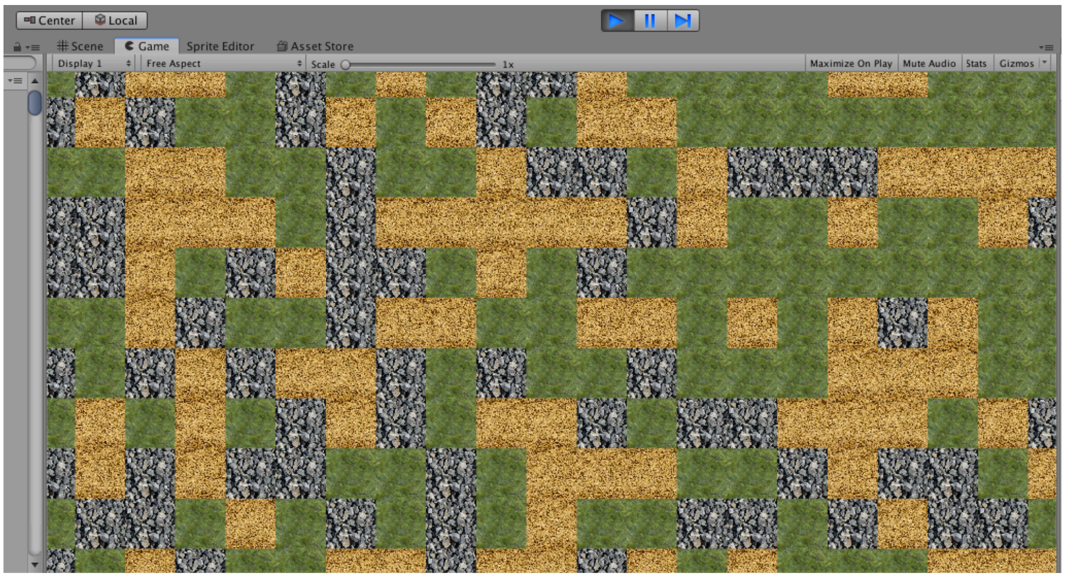


Figure 22.4 Randomly generated terrain using tiling

Until now the generated terrain is uncontrollable. The generation process is still random, and we cannot control by any means how the different textures are distributed in the terrain. Let us go one step further, and make the process controllable somewhat or another.

1.3 Pseudorandom instead of random

It would be a huge improvement if we can find a way to make the generator remembers each terrain it generates somehow. This way, whenever we need to get back a previously generated terrain, we can do that. To add this feature, we have to use a method that return a pseudorandom number instead of a pure random. Earlier in this thesis, pseudorandom numbers generators and Perlin noise were discussed in detail (chapters 3 & 4). These techniques should look familiar to us now. Let us use one of them in order to replace the completely random behavior with a pseudorandom one. A particular value called seed will be used to generate the pseudorandom numbers. This way, whenever you use the same seed, the generator generates the exact same terrain. Likewise, whenever we provide the generator with a different seed, a new terrain will be generated. Let us use a Perlin noise function to achieve our purpose.

The following two methods do exactly what we want.

```
private void DivideIntoTilesWithPerlinNoise() {
    CoordinateAdjustment = new Vector3(0 - width / 2, 0 - height / 2, 0);
    for (int x = 0; x < width; x++) {
        X1 += Seed;
        for (int y = 0; y < height; y++) {
            Y1 += Seed;
            Tile = new GameObject();
            Tile.transform.position = new Vector3(x, y, 0) + CoordinateAdjustment;
            RenderTheImage(); }}}
```

```
private void RenderTheImage() {
    SpriteRenderer = Tile.AddComponent<SpriteRenderer>();
    SpriteRenderer.sprite = mainImages[Mathf.RoundToInt(Mathf.PerlinNoise(X1, Y1) * 2)]; }
```

Let us run the generator for different seed values, and see what we will get. For a seed value equals 2.45, the generator generates the following terrain:

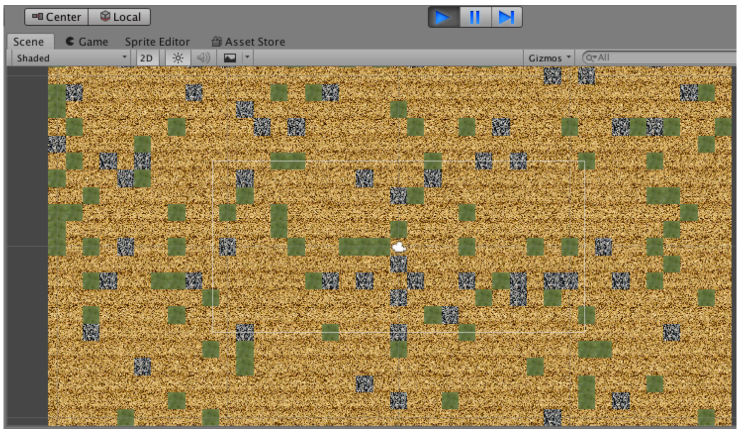


Figure 22.5 Generated terrain for seed=2.45

For a tiny modification in the seed to be 2.46, many changes occur in the generated terrain.

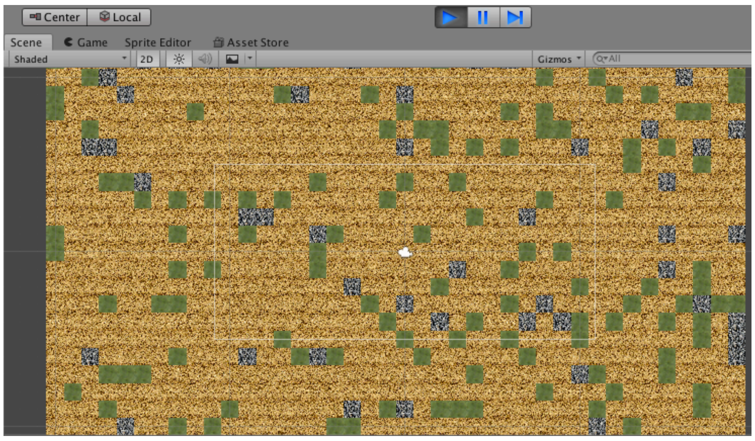


Figure 22.6 Generated terrain for seed=2.46

Let us assume that for some reason, we need to get the first terrain again. This task becomes now very simple. Just provide the generator with the seed value 2.45, and the generator will regenerate the previous terrain (figure 22.7).

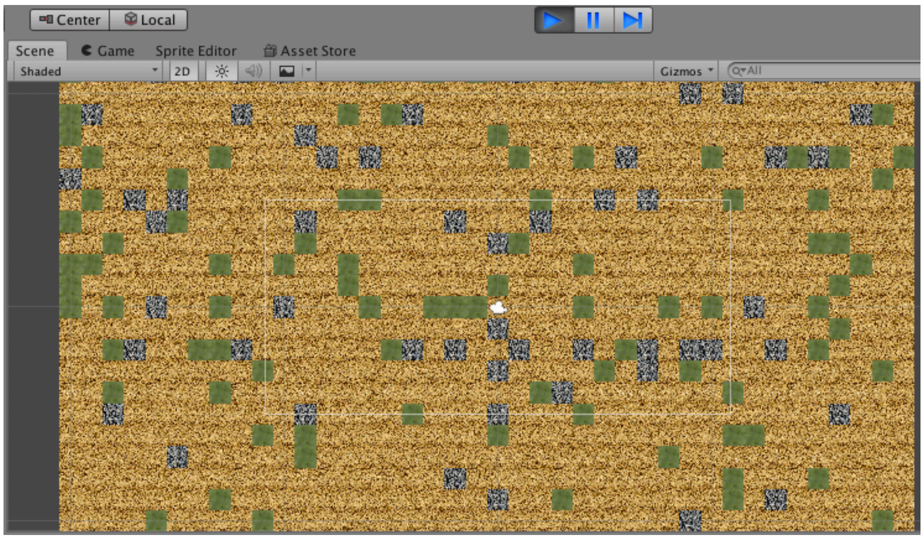


Figure 22.7 Generated terrain for seed=2.45

Our progress has been satisfactory up to now. So far so good !

But, wait a minute. *Would not it be dreamy if there is a way to make our small terrain goes somehow infinitely, and forever?*

But I know, it is just a fantasy……………

1.4 Fill the entire space and enjoy the infinity

What we have done so far is getting some images on the screen, tiling them, and display them in the scene. Our current terrain has specific dimensions, and if you move in any direction, you will inevitably hit one of its four boundaries (figure 22.8).

But what if we want to expand our nice terrain indefinitely, and forever?

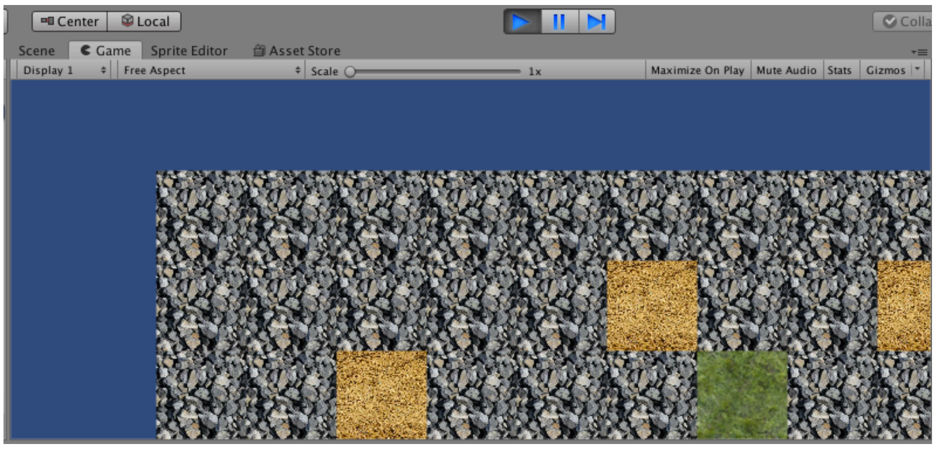


Figure 22.8 Terrain boundary

To make our dreams come true, the generator has to ***redraw*** the terrain forever.
The idea is very simple. Let us say that:
P: is a player or a user moves inside the terrain.
D: represents the distance between the current player location and the center of the terrain.
The player starts from the center. No matter the direction of the movement, whenever the player moves a distance equals to D in any direction, the generator will redraw the terrain again. This process repeats over and over again as long the redrawing criterion holds. In fact, what happens behind the scene is that the generator moves the blocks (tiles) in specific direction according to the player movement each time we ask the generator to redraw the terrain. Even though it seems a simple trick, but it does exactly what we want. At least it gives us the illusion of infinity. Is not it so ?

2. A gorgeous & dynamic terrain

2.1 Polish the generated Terrain

You might ask yourself: well, our generator generates an infinite terrain that goes on forever. That is our main purpose of this chapter. So, why do we need any further enhancements? Well, on one hand, we are actually done, and we have generated an infinite terrain using tiling. But, on the other hand, tiling are certainly not the best choice to do that. The reason is you have to control every way the transition occurs between any adjacent tiles. Tiling is very straightforward. That is, what you see is exactly what you get on the scene. For every tile, you should control how does it border and connect to other tiles. Our implementation so far is just a collection of tiles that are placed next to each other. There is no interaction or overlapping between the tiles. Our terrain is very static so far. Therefore, it looks far away from a real terrain. If you insist on building the dynamic terrain with only tiling, and your aim is to allow different textures to overlap, your first intuitive suggestion might be to place the images over each other's. But you know what, you will get into a big trouble. That is because:

- You have to control how the textures will connect to each other's.
- A huge amount of extra work with graphics will be added.

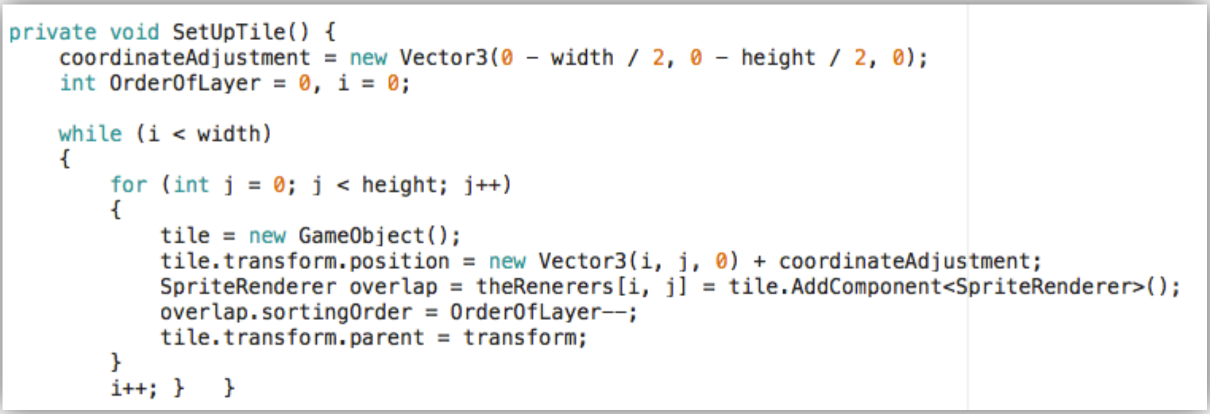
Keep in your mind that the more textures you add to get a more real terrain, the more complex it becomes in regards to graphics.

- From code perspective, whenever you add any texture, you need to take four tiles at once into your account. That is, what is the tile at each direction: above, to the right, to the left and at the bottom? Just imagine, that your terrain may consist of thousands of different tiles. How do you manage all that in your code? For example, for a terrain that has only two types of textures (grass and water). There are 16 different combinations for each tile. That is: WWWW- WGWG-GWGG- and so on (W refers to water and G to grass). To check out the potential combinations for every single tile in your code is certainly impractical.

That being said, how could we decrease this complexity ?

2.2 Moving smoothly between different textures

The way I have chosen to overcome the previous problem is by using a texture tile that fades into the next texture using transparencies. In other words, using a graphics that overlap each other instead of just sitting side by side. Each texture will be located at its own separate layer. The order of each layer is important and it should be taken into account because it affects the generated terrain. That is, whenever the order of any layer changes, the generated terrain changes. When we use tiling, there exist only one layer. Therefore, the order of layers does not make sense. But now with different layers located on the top of each other's, the order of each one really matters. The way the layers are located resembles the concept of the queue. That is, First In First Out [FIFO]. Any new layer will be located at the bottom of the previously located layers. To implement this idea in code, the first texture is given the highest priority. Before adding any new texture to the terrain, the priority should be decreased by one, and so on. But how can we display some parts from all the layers at the generated terrain if all of them are underneath the top texture? To make that possible, a transparency layer to some parts of each layer has been added. This way, the generated terrain contains parts from all the layers no matter the priority they have. The following method perform this task.



```
private void SetUpTile() {
    coordinateAdjustment = new Vector3(0 - width / 2, 0 - height / 2, 0);
    int OrderOfLayer = 0, i = 0;

    while (i < width)
    {
        for (int j = 0; j < height; j++)
        {
            tile = new GameObject();
            tile.transform.position = new Vector3(i, j, 0) + coordinateAdjustment;
            SpriteRenderer overlap = theRenerers[i, j] = tile.AddComponent<SpriteRenderer>();
            overlap.sortingOrder = OrderOfLayer--;
            tile.transform.parent = transform;
        }
        i++; }   }
```

The generated terrain looks now much better than before. It feels real to us (figure 22.9).

Running the generator gives the following terrain:

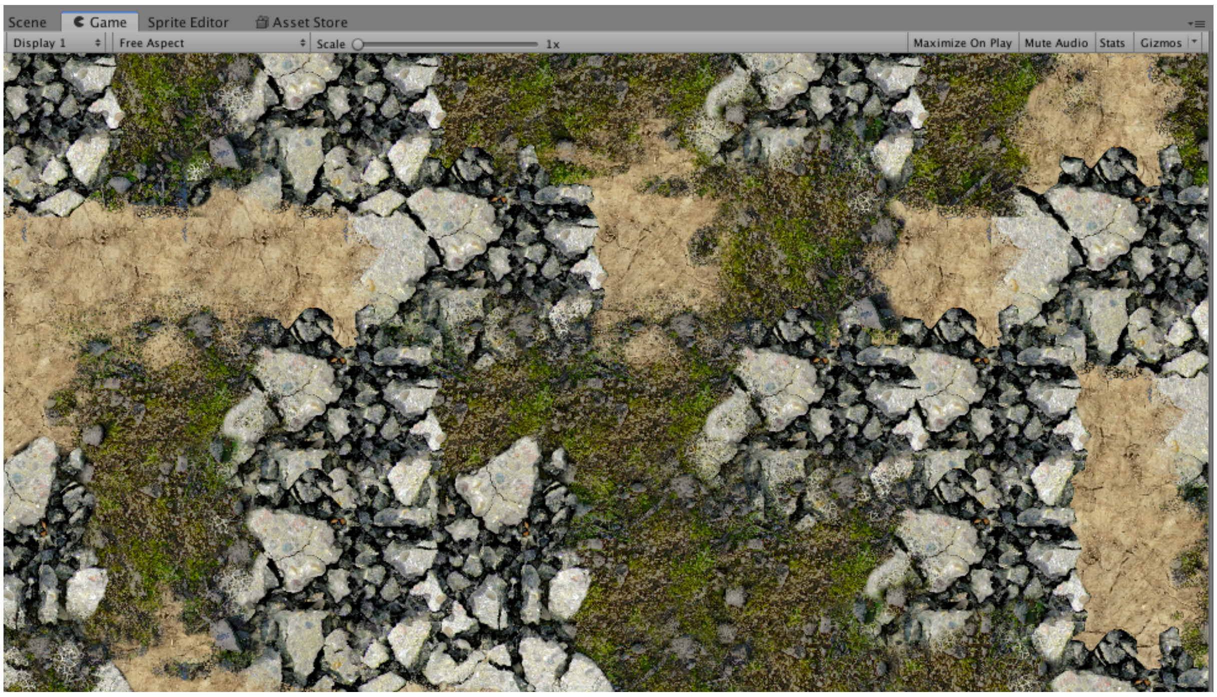


Figure 22.9 Generated terrain using transparencies between layers

Changing the order of some layers, the generated terrain differs (figure 22.10).

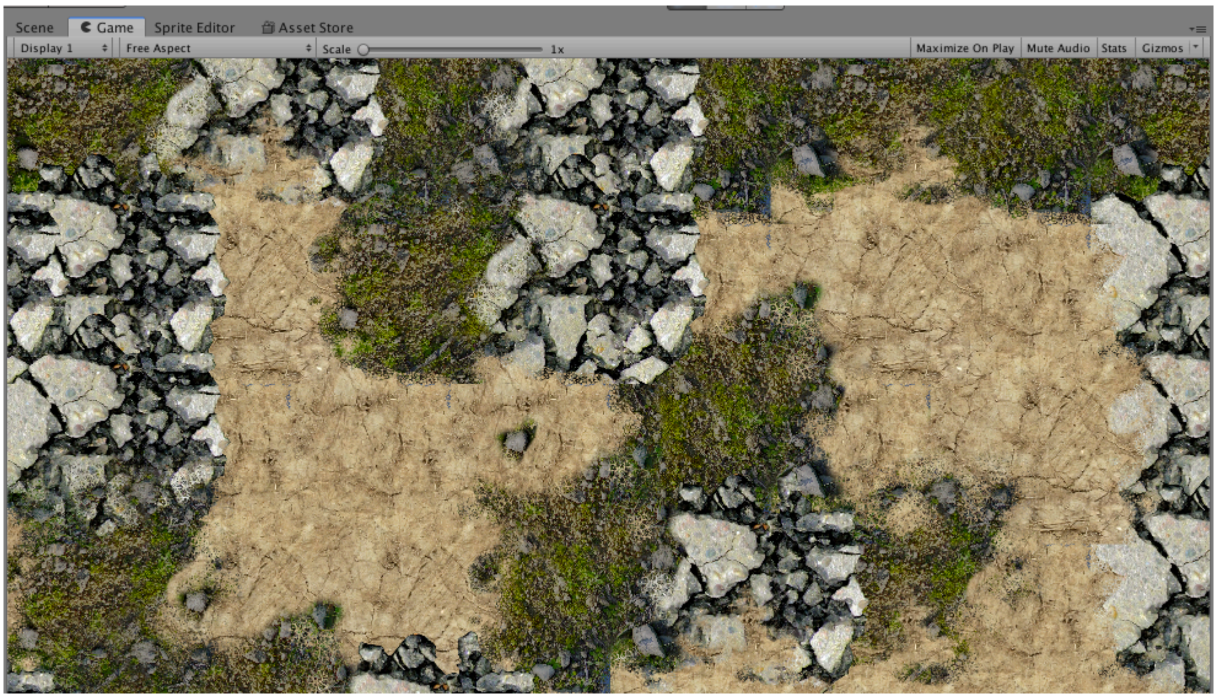


Figure 2210 Generated terrain using transparencies between layers

The complete code can be found in the appendix.

Chapter 11

Discussion

Throughout the previous chapters, more than 20 algorithms have been illustrated, implemented and tested. Let us discuss and recap what we have done so far. In chapter 3, we demonstrated three different algorithms: MSM, LCG and MT. the main purpose of those algorithms is to generate pseudorandom numbers instead of pure random ones. Each one has its own benefits and drawbacks as well. MSM is good choice because:

- The simplest one in regards to implementation.
- It is very fast.

But its huge drawback, however, is:

- It has a very short period.

Keep in your mind that in any pseudorandom number generator, a short period is considered to be as a fatal flaw. The experiments we performed on MSM were intimidating. This algorithm was unable to create even 100 different numbers before it created duplicates. Luckily, not all hope is lost, because there are various PRNGs available that aim to bridge this gap in particular. Of them is LCG that comes as lifesaver because:

- It has a great period, and it is much superior of MSM.

But it still has deficiency when:

- It is used for security purposes. In such cases, it fails immediately.

Both MMS and LCG are not categorized as a high-quality PRNGs. Therefore, we demonstrated one of those which is called MT. This algorithm is :

- The most widely-used PRNG.
- The default PRNG in many programing languages, frameworks and libraries.
- It has a very long period.
- It passes huge number of tests including Diehard tests.

However, it has some detrimental side effects:

- Due its high quality and great performance, it needs much more memory.
- It is not the best choice to be used as a cryptographic technique.

Bottom line is, which PRNG should you use is entirely dependent on your application and the resources available. If the memory is not a big concern, then you should for sure opt MT.

In chapter 4, Perlin noise was illustrated and tested. It is very popular in procedural systems, and it is considered as the classic noise. A Perlin noise function was also used in chapter 10 to generate an infinite terrain. One of its useful aspects is:

- The smooth transition it provides between values.

However, it has some weaknesses. The major one is:

- It is extremely slow for higher dimensions.

To overcome this problem, Ken Perlin came up with improved version of Perlin noise in 2001. It is called Simplex noise **[51].** According to Ken Perlin, this noise is:

- Easy to implement.
- The complexity of computational is less compared to classic Perlin noise.
- Faster and easy to extend to multiple dimensions.

In chapter 5, three different algorithms for filling the space were discussed and tested. The first one is random walks. Two variant implementations were discussed. The classical random walks where the direction of movement is restricted to the dimensions of the space. Then an improvement was suggested to free the movement from the previous constrains. That is, the walker can move in the space randomly in unlimited number of possible directions. Random walks are good if your purpose is:

- Filling the space haphazardly.

If that is not the case, you should avoid using random walks.
The second algorithm was DLA, where a number of attraction points located randomly in the space, and a number of random walkers fire alternatively in the space until it gets attracted by one of the attraction points. Depending on the specific content that you want to fill the space with, you can choose between random walks and DLA. The classis random walks generate a pattern similar to map. On the other hand, DLA generates a pattern similar to tree somewhat.
The third concept was cellular automata. CA are very useful to describe many biological phenomena. It is a great choice if:

- The system is spatially heterogeneous **[52]**.

And a bad choice if:

- The system is spatially homogeneous. In Such cases, ODEs (Ordinary Differential Equations) and DDEs (Discrete Difference Equations) should be used **[52].**

In chapter 6, sequence generation techniques were demonstrated. We started with Markov chains, and implemented it to generate a text. A very useful merit of Markov chains **[53]**:

- Its ability to model the majority of systems where there exist dependencies between states.

However, it has a known drawback, which is **[53]**:

- Its need to a huge number of states to represent some simple systems.

After that, three different methods to generate fractals were studied. Chaos game was the first method which served as a tool to generate Sierpinski triangle. Then another method called Space Colonization Algorithm was illustrated. This algorithm is used to generate a like-tree pattern. At the end of this chapter, L-Systems was demonstrated and two examples were implemented. Fractals are a huge topic, and there exists tremendous number of algorithms to generate fractals.

Moving to chapter 7, two methods to partition the space were studied. The first one is Binary Space Partition, which is a common tool to divide a space into smaller parts. The subdivision process of the space can be represented by a tree data structure known as BSP tree. These trees have some advantages and disadvantages. A big advantage is **[54]**:

- It is very efficient to detect collisions in the map in a fast manner.

A huge drawback, however **[54]:**

- To achieve high performance, this algorithm needs additional requirements which turns BSP trees to be somehow complicated.

After BSP, Voronoi diagram was demonstrated. A real-world example was implemented using Voronoi diagrams.

In chapter 8, two methods were studied to generate like-maze patterns. Even though recursive backtracker and reaction diffusion algorithm generates eventually a pattern similar to a maze, there are a major difference. RDA has the ability to generates enormous number of different patterns. Each time a tiny modification for one of its parameters, a completely new maze shows up. Recursive backtracker, however, will always generate almost the same pattern.

In chapter 9, a method to procedurally generate flowers was demonstrated. Even though only one function was used to generate flowers (cosine function), there are plenty of functions that can be used to generate different flower patterns compared to what we got in our experiments.

Finally, in chapter 10 something different was implemented. No algorithms or techniques were demonstrated. An infinite terrain was implemented step by step. We ended up eventually with a terrain very close to a real one.

Chapter 12

Conclusion

Throughout this thesis, different topics were studied, implemented and discussed. Firstly, a survey of applications of procedural generation was introduced. In chapter 2 particularly, the main focus was to draw attention to the cases where using PCG is considered to be either a merit or a demerit. Then a categorizing of the techniques, design patterns and approaches that are used in PCG were demonstrated. The main categories in this thesis are:

1- Random numbers.
2- Perlin Noise.
3- Filling Space.
4- Sequence generation.
5- Partitioning space.
6- Maze generation.
7- Procedural flowers.

Then in chapter 11, the strength and weaknesses of the different strategies were highlighted. With regards to further developments and future works, i would like to mention that this topic is a broad discipline. Honestly, with roughly 140 pages in this thesis, I have to admit that I have barely scratched the surface. Every single category in this thesis is a good candidate as a separate master thesis. Moreover, the infinite terrain that was implemented in chapter 10 can be effectively used as the cornerstone of unlimited number of games. An infinite space is the backbone for almost every game nowadays. Our generated infinite terrain can be easily polished by any designer/developer to be consistent with different games. How valuable our infinite terrain is!

Bibliography

1. Short, T.X. and T. Adams, *Procedural Generation in Game Design*. 2017: AK Peters/CRC Press.
2. Freeman, E., et al., *Head First Design Patterns: A Brain-Friendly Guide*. 2004: " O'Reilly Media, Inc.".
3. Stackoverflow, *https://stackoverflow.com/*.
4. Shiffman, D., *https://www.youtube.com/user/shiffman*.
5. Wikipedia, *Minecraft* *https://en.wikipedia.org/wiki/Minecraft*.
6. Wikipedia, *No Man´s Sky* *https://en.wikipedia.org/wiki/No_Man%27s_Sky*.
7. L'Ecuyer, P., *Random Numbers for Simulation*. Vol. 33. 1990. 86-87.
8. Francillon, A. and C. Castelluccia. *TinyRNG: A cryptographic random number generator for wireless sensors network nodes*. in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on*. 2007. IEEE.
9. Schoo, M., K. Pawlikowski, and D.C. McNickle, *A survey and empirical comparison of modern pseudo-random number generators for distributed stochastic simulations.* 2005.
10. Von Neumann, J.J.A.M.S., *Various techniques used in connection with random digits.* 1951. **12**(36-38): p. 36-38.
11. Bolte, J.J.W.D.P., *Linear congruential generators.*
12. O'Neill, M.E.J.A.T.o.M.S., *PCG: A family of simple fast space-efficient statistically good algorithms for random number generation.* 2014: p.4-8.
13. Niederreiter, H.J.B.o.t.A.M.S., *Quasi-Monte Carlo methods and pseudo-random numbers.* 1978. **84**(6): p. 1000-1003.
14. Press, W.H., et al., *Numerical Recipes with Source Code CD-ROM 3rd Edition: The Art of Scientific Computing*. 2007: Cambridge University Press.
15. Matsumoto, M., T.J.A.T.o.M. Nishimura, and C. Simulation, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.* 1998. **8**(1): p. 5-8

.

16. Marsaglia, G., *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*

. 2005.

17. Matsumoto, M., Y.J.A.T.o.M. Kurita, and C. Simulation, *Twisted gfsr generators ii.* 1994. **4**(3): p. 254-266.
18. Pontes, E.A.S., *A Brief Historical Overview Of the Gaussian Curve: From Abraham De Moivre to Johann Carl Friedrich Gauss.* International Journal of Engineering Science Invention (IJESI), 2018: p. 28-34.
19. Saghir, A., et al., *Weighted distributions: A brief review, perspective and characterizations.* 2017. **6**(3): p. 109-131.
20. Perlin, K.J.A.S.C.G., *An image synthesizer.* 1985. **19**(3): p. 287-296.

21. Pearson, K.J.N., *The problem of the random walk.* 1905. **72**(1867): p. 342.
22. OpenCourseWare, M., *Random Walks.pdf.* 2010. **20**: p. 533-539.
23. Witten Jr, T. and L.M.J.P.r.l. Sander, *Diffusion-limited aggregation, a kinetic critical phenomenon.* 1981. **47**(19): p. 1400.
24. Offringa, A., *Diffusion Limited Aggregation.pdf.* p. 2-11.
25. Manneville, P., et al., *Cellular Automata and Modeling of Complex Physical Systems: Proceedings of the Winter School, Les Houches, France, February 21–28, 1989*. Vol. 46. 2012: Springer Science & Business Media.
26. Burzyński, M., et al., *Cellular automata: structures and some applications.* 2004. **42**(3): p. 461-465.
27. Schiff, J.L., *Cellular automata: a discrete view of the world*. Vol. 45. 2011: John Wiley & Sons.
28. Kari, J., *Cellular automata.* 2013.
29. Gardner, M.J.S.A., *Mathematical games: The fantastic combinations of John Conway's new solitaire game "life".* 1970. **223**(4): p. 120-123.
30. Gagniuc, P.A., *Markov chains: from theory to implementation and experimentation*. 2017: John Wiley & Sons.
31. TOLVER, A., *AN INTRODUCTION TO MARKOV CHAINS.* p. 13-14.
32. Mandelbrot, B.B., *The fractal geometry of nature*. Vol. 2. 1982: WH freeman New York.
33. Churchill, M., *Introduction to Fractal Geometry.* 2004: p. 1-9.
34. Conversano, E. and L.T.J.J.A.M. Lalli, *Sierpinski triangles in stone, on medieval floors in Rome.* 2011. **4**: p. 114-122.
35. Wikipedia, *Sierpinski triangle* https://en.wikipedia.org/wiki/Sierpinski_triangle
.
36. Barnsley, M.F., *Fractals everywhere*. 2014: Academic press.
37. Pyke, R., *Fractals and the Chaos Game.* 2009.
38. Adam Runions, B.L., and Przemyslaw Prusinkiewicz, *Modeling Trees with a Space Colonization Algorithm.* 2007: p. 3-6.
39. Prusinkiewicz, P. and J. Hanan, *Lindenmayer systems, fractals, and plants*. Vol. 79. 2013: Springer Science & Business Media.
40. Shaker, N., J. Togelius, and M.J. Nelson, *Procedural content generation in games*. 2016: Springer.
41. Prusinkiewicz, P. and A. Lindenmayer, *Graphical modeling using L-systems*, in *The Algorithmic Beauty of Plants*. 1990, Springer. p. 25.
42. Barnsley, M., *Fractals everywhere. 1993*. Academic Press.
43. Schumacker, R.A., et al., *Study for applying computer-generated images to visual simulation.* 1969, GENERAL ELECTRIC CO DAYTONA BEACH FL APOLLO AND GROUND SYSTEMS.
44. Dong, P.J.C. and Geosciences, *Generating and updating multiplicatively weighted Voronoi diagrams for point, line and polygon features in GIS.* 2008. **34**(4): p. 411-421.

45. Gavrilova, M.L., *Generalized voronoi diagram: a geometry-based approach to computational intelligence*. Vol. 158. 2008: Springer.
46. Franz Aurenhammer , R.K., *Voronoi Diagrams.pdf.* p. 1-6.
47. Foltin, M., *Automated maze generation and human interaction*. 2011, Masarykova univerzita, Fakulta informatiky.
48. Abelson, A., Coore, Hanson, Nagpal, Sussman, *Gray Scott Model of Reaction Diffusion*

.
49. Sims, K., *Reaction-Diffusion Tutorial, http://www.karlsims.com/rd.html*

50. Cundy, H.M. and A.P. Rollett, *Mathematical models*. 1961: Clarendon Press Oxford.
51. Perlin, K.J.R.-T.S.S.C.N., *Noise hardware.* 2001.
52. Unit, B.S.J.C.S., *Differential equations and cellular automata models of the growth of cell cultures and transformation foci.* 2001. **13**: p. 347-380.
53. Boyd, M.A. and S. Lau, *An introduction to Markov modeling: Concepts and uses.* 1998.
54. Ranta-Eskola, S. and E.J.U.m.s.t.i.c.s. Olofsson, *Binary space partioning trees and polygon removal in real time 3d rendering.* 2001.

Appendix

All the lists in the appendix are written in JavaScript. In order to run any of them, you need HTML file. The following file is general-purpose file. That is, you can use with any one of the JavaScript files. All you have to do is write the name of the file you want to run in the mentioned area below. Do not forget to add the correct extension. For example, if you want to run the code to test Middle Square Method, you have to copy the list-1- from the next page into a text file, and save the file using the extension .js. Then use the HTML template below, and write Middle Square Method.js in the placeholder in line 13.

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.    <head>
4.      <meta charset="UTF-8" />
5.      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6.      <meta http-equiv="X-UA-Compatible" content="ie=edge" />
7.      <title>Procedural Generation</title>
8.    </head>
9.    <body>
10.     <canvas id="canvas" width="500" height="500"></canvas>
11.     <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.6.1/p5.js"></script>
12.     <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.6.1/addons/p5.dom.js"></script>
13.     <script src=" Here you have to write the name of the file you want to run"></script>
14.   </body>
15. </html>
```

List 1 - Middle Square Method

****

```
let { seed, index, randomNumber, period } = declareVariables()

window.onload = () => {
  printResult()
  findPeriod() }

function declareVariables () {
  let index
  let randomNumber
  let period = []
  let seed = 1234
  return { seed, index, randomNumber, period } }

function setup () {}

function squareNumber () {
  let numOfDigits = 4
  let pseudoRandomNumber = (seed * seed).toString()
  return { numOfDigits, pseudoRandomNumber } }

function middleSquareAlgorithm () {
  let { numOfDigits, pseudoRandomNumber } = squareNumber()
  padResult()
  pullMiddleDigits()
  return seed

  function pullMiddleDigits () {
    let start = Math.floor(numOfDigits / 2)
    let end = start + numOfDigits
    seed = parseInt(pseudoRandomNumber.substring(start, end)) }

  function padResult () {
    while (pseudoRandomNumber.length < numOfDigits * 2) {
      pseudoRandomNumber = '0' + pseudoRandomNumber } } }

function printResult () {
  for (index = 0; index < 10; index++) {
    createP(middleSquareAlgorithm()) }}

function checkRepetition () {
  for (index = 0; index < 100; index++) {
    randomNumber = middleSquareAlgorithm()
    if (period[randomNumber]) { break }
    period[randomNumber] = true }
  return index }

function findPeriod () {  createP(checkRepetition()) }
```

List 2- Linear Congruential Generator [LCG]

****

```
let index, period, seed = 123456,result = []

function setup () {
  createCanvas(600, 600) }

window.onload = () => { printResult() }

function linearCongruentialGenerator () {
  const { a, c, m } = defineConstants()
  applySeedEquation(a, c, m)
  return seed }

function applySeedEquation (a, c, m) {
  seed = (a * seed + c) % m }

function defineConstants () {
  const a = 1664525
  const c = 1013904223
  const m = Math.pow(2, 32)
  return { a, c, m } }

function checkRepetition () {
  for (index = 0; index < 1000000; index++) {
    let randomNumber = linearCongruentialGenerator()
    if (result[randomNumber]) { break }
    result[randomNumber] = true }
  return index }

function findPeriod () {
  period = checkRepetition()
  return period }

function printResult () {
  for (let index = 0; index < 10; index++) {
    createP(linearCongruentialGenerator()) }
  createP(findPeriod()) }
```

List 3- Mersenne Twister

****

```
// This implementation uses values for ID= MT 11213B.
//Source: http://www.math.sci.hiroshima-u.ac.jp/
//      ~m-mat/MT/ARTICLES/mersenneTwister.pdf
// All these values are listed in the table in page.8
const seed = 123434,a = 0xccab8ee7,m = 175,degreeOfRecurrence = 351,
    last32Bit = 18122433253 & 0xffffffff
let y

window.onload = () => {
  printRandomSequence() }

function setup () {
  createCanvas(500, 500) }

function performSuccessiveTransformations (mersenneTwister, i) {
  // Tempering parameters
    const { TEMPERING_SHIFT_U, TEMPERING_SHIFT_S, TEMPERING_MASK_B,
        TEMPERING_SHIFT_T, TEMPERING_MASK_C, TEMPERING_SHIFT_L } =
        defineTemperingConstants();
  // Step 4
  y = mersenneTwister[i]
  y ^= TEMPERING_SHIFT_U
  y ^= TEMPERING_SHIFT_S & TEMPERING_MASK_B
  y ^= TEMPERING_SHIFT_T & TEMPERING_MASK_C
  y ^= TEMPERING_SHIFT_L
  // Step 5
  i = (i + 1) % degreeOfRecurrence
  return y }

class MersenneTwister {
  constructor (seed) {
    let i = 0
    let mersenneTwister = [seed]
    for (let j = 1; j < degreeOfRecurrence; j++) {
      mersenneTwister[j] =
      last32Bit * (mersenneTwister[j - 1] ^ (mersenneTwister[j - 1] >> 30)) +1}
    this.nextNumber = () => {
      if (i === 0) { generateNextNumber() }
      return performSuccessiveTransformations(mersenneTwister, i) }

    function generateNextNumber () {
      for (let i = 0; i < degreeOfRecurrence; i++) {
        // Step 2
          y = mersenneTwister[i] & 0x80000000 ||mersenneTwister[(i + 1)
              % degreeOfRecurrence] & 0x7fffffff
        // Step 3
    mersenneTwister[i]=mersenneTwister[(i + m) % degreeOfRecurrence] ^ (y >> 1)
```

```
      if (y % 2 !== 0) {mersenneTwister[i] = mersenneTwister[i] ^ a }}}}}

function defineTemperingConstants() {
  const TEMPERING_MASK_B = 0x31b6ab00;
  const TEMPERING_MASK_C = 0xffe50000;
  const TEMPERING_SHIFT_U = y >> 11;
  const TEMPERING_SHIFT_S = y << 7;
  const TEMPERING_SHIFT_T = y << 15;
  const TEMPERING_SHIFT_L = y >> 17;
    return { TEMPERING_SHIFT_U, TEMPERING_SHIFT_S, TEMPERING_MAS
K_B,
      TEMPERING_SHIFT_T, TEMPERING_MASK_C, TEMPERING_SHIFT_L };
 }

function printRandomSequence () {
  let MT = new MersenneTwister(seed)
  for (let i = 0; i < 10; i++) {
    let nextRandomNumber = MT.nextNumber()
    createP(nextRandomNumber) } }
```

List 4- Normal Probability Distribution

****

```
let index
function setup () {
    createCanvas(500, 500) }

window.onload = () => {
  let probabilities = []
  fillWithZeros(probabilities)

function generateRandomRange (min, max) {
    return min + Math.random() * (max - min) }

function findNormalProbabilityDistribution () {
    let numOfRandomNumbers = 3
   let total = FindfSumOfRandomNumbers(numOfRandomNumbers, generateRandomRange)
    increaseProbability()

function increaseProbability () {
      average = Math.floor(total / numOfRandomNumbers)
      probabilities[average] += 1 } }


function draw () {
    for (index = 0; index < 150; index++) {
      let h = probabilities[index] * -20
      rect((width / 100) * index, height, width / 100, h)
      fill(0) } }


function update () {
    findNormalProbabilityDistribution()
    draw()
    requestAnimationFrame(update) }
    update() }

function FindfSumOfRandomNumbers (numOfRandomNumbers, generateRandomRange) {
  let total = 0
  for (index = 0; index < numOfRandomNumbers; index++) {
    total += generateRandomRange(0, 100) }
  return total }

function fillWithZeros (probabilities) {
  for (index = 0; index < 500; index++) {
    probabilities[index] = 0 } }
```

List 5- Weighted Random

****

```
function setup () {
  createCanvas(600, 600)
  divideCanvasVertically()
  for (let index = 0; index < 100; index++) {
    let tails = setPriorities()

    let { x, y } = generateRandomlyPixelCoordinate()
    x = moveToRightHandSide(tails, x)
    drawResultOnCanvas(x, y) } }

function moveToRightHandSide (tails, x) {
 if (!tails) { x += width / 2 }
 return x }

function generateRandomlyPixelCoordinate () {
 let y = Math.random() * height
 let x = (Math.random() * width) / 2
  return { x, y }}

function divideCanvasVertically () {
 line(width / 2, 0, width / 2, height) }

function setPriorities () {
 return Math.random() > 0.5 }

function drawResultOnCanvas (x, y) {
 ellipse(x, y, 20, 20)
 fill(0) }
```

List 6- Perlin Noise

```
let pixelLocation, brightness
function setup () {
  createCanvas(300, 300)
  pixelDensity(1) }

function colorEverypixel (pixelLocation, brightness) {
  // Color each pixel using RGB
  let i = 0
  while (i < 3) {
    pixels[pixelLocation + i] = brightness
    i++ }
  // alpha value
  pixels[pixelLocation + 3] = 255 }

function applyPerlinNoise (verticalCoordinate,horizontalCoordinate,
  horizontalOffset, verticalOffset,increment) {
  verticalCoordinate = 0
  while (verticalCoordinate < width) {
    findPixelLocation(verticalCoordinate, horizontalCoordinate)
    calculateBrightness(horizontalOffset, verticalOffset)
    colorEverypixel(pixelLocation, brightness)
    horizontalOffset += increment
    verticalCoordinate++ }
  return { verticalCoordinate, horizontalOffset } }

function calculateBrightness (horizontalOffset, verticalOffset) {
  brightness = noise(horizontalOffset, verticalOffset) * 255 }

function findPixelLocation (verticalCoordinate, horizontalCoordinate) {
  pixelLocation = (verticalCoordinate + horizontalCoordinate * width) * 4 }

function draw () {
  let increment = 0.08
  let horizontalCoordinate = 0
  let verticalCoordinate
  let verticalOffset = 0
  loadPixels()
  while (horizontalCoordinate < height) {
    let horizontalOffset = 0
        ;({ verticalCoordinate, horizontalOffset } = applyPerlinNoise(
            verticalCoordinate,horizontalCoordinate,horizontalOffset,
            verticalOffset, increment ))
    verticalOffset += increment
    horizontalCoordinate++ }
  updatePixels() }
```

List 7- Random Walk

****

```
let p1 = (p2 = 120)

window.onload = () => {
  createCanvas(500, 500)
  background(0) }

function walkRandomly () {
  let randomNumber = floor(random(4))
  switch (randomNumber) {
    case 0:   p1 += 2 // Right-movement
      break
    case 1:   p1 -= 2 // Left-movement
      break
    case 2:   p2 += 2 // Up-movement
      break
    case 3:   p2 -= 2 // Down-movement
      break } }

function draw () {
  stroke(255, 255)
  strokeWeight(1.8)
  point(p1, p2)
  walkRandomly() }
```

List 8- Improved Random Walk

****

```
let currPosition, previousPosition

function setup () {
  createCanvas(500, 500)
  background(0)
  currPosition = createVector(250, 250)
  previousPosition = createVector(250, 250)
  copyVector() }

function increasevectorLength (step) {
  let vectorAmplification = random(25, 75)
  step.x *= vectorAmplification
  step.y *= vectorAmplification }

function copyVector () {
  previousPosition.x = currPosition.x
  previousPosition.y = currPosition.y }

function addVector (step) {
  currPosition.x += step.x
  currPosition.y += step.y }

function stepAccordingToProbability (step) {
  let probability = random(1)
    if (probability < 0.0125) { increasevectorLength(step) }
    else {
    let magnitude = sqrt(step.x * step.x + step.y * step.y)
    step.magnitude } }

function draw () {
  stroke(250)
  strokeWeight(1.8)
  let step = p5.Vector.random2D()
  line(currPosition.x, currPosition.y, previousPosition.x, previousPosition.y)
  previousPosition.set(currPosition)
  stepAccordingToProbability(step)
  addVector(step) }
```

List 9- Diffusion Limited Aggregation

```
let {growingObject,maxWalkers,walkers,iterations,particleRadius} = DeclareVariables()

function setup () {
  createCanvas(500, 500)
  locateParticleRandomly()
  createRandomWalkers() }

// Step 1
function locateParticleRandomly () {
  let p1 = floor(random() * width)
  let p2 = floor(random() * height)
  growingObject[0] = new Walker(p1, p2) }

// Step 2
function createRandomWalkers () {
  for (let index = 0; index < maxWalkers; index++) {
    walkers[index] = new Walker() } }

function draw () {
  background(255)
  displayOnCanvas()
  fillWalkersArray();
  fireWalkersInSpace()

  function displayOnCanvas () {
    displayGrowingObject()
    displayWalkers() } }

function fillWalkersArray() {
  while (walkers.length < maxWalkers) {
    walkers.push(new Walker()); } }

function displayGrowingObject () {
  for (let index = 0; index < growingObject.length; index++) {
    growingObject[index].show() } }

function displayWalkers () {
  for (let index = 0; index < walkers.length; index++) {
    walkers[index].show() } }

// Steps 3 & 4
function fireWalkersInSpace () {
  let index = 0
  while (index < iterations) {
    for (let index = 0; index < walkers.length; index++) {
      walkers[index].moveRandomlyInSpace()
      if (walkers[index].checkCollision(growingObject)) {
        attachParticle(index) } }
    index++ } }

// Step 5
function attachParticle (index) {
  growingObject.push(walkers[index])
  // Remove the walker from the array after attaching it
  walkers.splice(index, 1) }

function createWalkerRandomly () {
  return createVector(random(width), random(height)) }

function euclideanDistance (par1, par2) {
  let horDistance = par2.x - par1.x
  let verDistance = par2.y - par1.y
  return horDistance * horDistance + verDistance * verDistance }

class Walker {
  constructor () {
    this.setUpWalker()
    this.radius = particleRadius
    this.moveRandomlyInSpace = () => {
      this.increaseWalkerSpeed();
      this.constrainWalkerMovement() }

    this.checkCollision = otherParticles => {
      for (let index = 0; index < otherParticles.length; index++) {
        let distance = this.calculateDistance(otherParticles, index)
        if (distance <= 3 * particleRadius) {
          this.collision = true
          return true } }
      return false }
    this.show = () => {
      if (this.collision) { fill(0) }
       else { fill(360, 0, 255) }
      ellipse(this.position.x,this.position.y,this.radius * 2,this.radius * 2 ) } }

  increaseWalkerSpeed() {
    let velocity = p5.Vector.random2D();
    this.position.add(velocity); }

  setUpWalker () {
    this.position = createWalkerRandomly()
    this.collision = false }

  calculateDistance (otherParticles, index) {
    return euclideanDistance(this.position, otherParticles[index].position) }

  constrainWalkerMovement () {
    this.position.x = constrain(this.position.x, 0, width)
    this.position.y = constrain(this.position.y, 0, height) } }


```

```
function DeclareVariables () {
  let growingObject = []
  let walkers = []
  let maxWalkers = 50
  let iterations = 1000
  let particleRadius = 2
  return { growingObject, maxWalkers, walkers, iterations, particleRadius } }
```

List 10- Game Of Life (Cellular Automata)

****

```
let checkBoard ,numOfColumns, numOfRows, cellSize = 10

function setup () {
  createCanvas(600, 600)
  numOfColumns = width / cellSize
  numOfRows = height / cellSize
  createCheckboard() }

function createCheckboard () {
  // Determine the size for each cell
  numOfColumns = width / cellSize
  numOfRows = height / cellSize
  checkBoard = create2DArray(numOfColumns, numOfRows)
  fillGridRandomly() }

function fillGridRandomly () {
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
        checkBoard[i][j] = floor(random(2)) } } }

function draw () {
  background(0)
  whitenSurvivalCells()
  determineNextState() }

// Step 4
function whitenSurvivalCells () {
  // Loop through each cell in the checkerboard
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
      // If its current value is one, then it survives
      if (checkBoard[i][j] == 1) {
        fill(255) // Whiten it.
        rect(i * cellSize, j * cellSize, cellSize, cellSize) }}}}

function create2DArray (numOfColumns, numOfRows) {
  let arr = new Array(numOfColumns)
  for (let i = 0; i < arr.length; i++) {
    arr[i] = new Array(numOfRows) }
  return arr }

// Step 3
function determineNextState () {
  let tempCheckBoard = create2DArray(numOfColumns, numOfRows)
  for (let i = 0; i < numOfColumns; i++) {
    for (let j = 0; j < numOfRows; j++) {
      let cellState = checkBoard[i][j]
      checkNeighborsState(i, j, cellState, tempCheckBoard)
    } } checkBoard = tempCheckBoard }
```

```

function checkNeighborsState (i, j, cellState, tempCheckBoard) {
  let numOfLiveNeighbors = countLiveNeighbors(checkBoard, i, j)
  applyConwaysLaws(cellState, numOfLiveNeighbors, tempCheckBoard, i, j) }

function applyConwaysLaws (cellState, numOfLiveNeighbors, tempCheckBoard, i, j )
{
  // Reproduction phase
  if (cellState == 0 && numOfLiveNeighbors == 3) { tempCheckBoard[i][j] = 1 }
  // Death Phase
 else if (cellState == 1 && (numOfLiveNeighbors < 2 || numOfLiveNeighbors > 3))
  {
    tempCheckBoard[i][j] = 0 }
  // Survival Phase
  else { tempCheckBoard[i][j] = cellState} }

// Find the number of live neighbors for each cell
function countLiveNeighbors (checkBoard, cellColumn, cellRow) {
  let currentCellValue = checkBoard[cellColumn][cellRow]
  let totalLiveNeighbors = 0
  let neighborColumn
  let neighborRow
  let horizontaloffset, verticaloffset
  for (horizontaloffset = -1; horizontaloffset <= 1; horizontaloffset++) {
    for (verticaloffset = -1; verticaloffset <= 1; verticaloffset++) {
      neighborColumn =
        (cellColumn + horizontaloffset + numOfColumns) % numOfColumns
      neighborRow = (cellRow + verticaloffset + numOfRows) % numOfRows
      totalLiveNeighbors += checkBoard[neighborColumn][neighborRow] }}
  return totalLiveNeighbors - currentCellValue }
```

List 11- Markov Chains
****

```
let inputData = `Moira Jurek Tai Pflug Alfonzo Cohan Floria Grahm
Barbera Golden Karina Clonts Deane Lauria Esteban Broder
Susann Canada Aiko Balcom Benny Manley Lanora Lightsey
Allison Thiessen Edmund Gathright Thomasena Broadfoot
Clarice Kornreich Laurel Dameron Coleen Dangelo Penni Leech
Shonta Rankins Iola Soloman Arlinda Vickery Detra Rousey
Violet Aquilar Jesse Surette Cecille Barnaby Lonny Lavoie
Libby Bondurant Erminia Fagan Kena Alvardo Bianca Irvin
Ethelyn Hyneman Hoyt Wisneski Vincent Fuchs Valrie Rayo
Laurence Hemingway Minh Yoshimura Brady Claar Thad Krueger`

function setup () {
  createCanvas(500, 500)
  for (let index = 0; index < 10; index++) {
    createP(new MarkovChain(inputData).generateNames()) } }

class MarkovChain {
  constructor (inputData) {
    this.dictionary = Object.create(null)
    this.IsDictionaryBuilt = false
    this.listOfIndividualNames = inputData.split(' ')
      this.startOfEachChain = [this.listOfIndividualNames[0]] }

  getRandomElementOfArray (array) {
      return array[Math.floor(Math.random() * array.length)] }

  // get the nextWord set of listOfIndividualNames as a string
  getNextWords (index) {
    return this.listOfIndividualNames.slice(index, index + 1).join(' ') }

  buildMarkovDictionary () {
    this.listOfIndividualNames.forEach((name, index) => {
      let nextWord = this.getNextWords(index + 2)
      ;(this.dictionary[name] = this.dictionary[name] || []).push(nextWord)
      this.startOfEachChain.push(name) })
      return (this.IsDictionaryBuilt = true) }

  // generateNames new text from a markov lookup
    generateNames () {
    if (this.buildMarkovDictionary()) {
    let randomStartingName = this.getRandomElementOfArray(this.startOfEachChain
)
        let randomChoice = this.getRandomElementOfArray(this.dictionary
            [randomStartingName])
    let array = [randomStartingName]
      array.push(randomChoice)
      return array.join(' ') }}}
```

List 12- Fractals / Chaos Game

****

```
window.onload = () => {
  let sierpinskiTriangle = new SierpinskiTriangle()
  for (let index = 0; index < 500; index++) {
    sierpinskiTriangle.start() } }

class Point {
  constructor (x, y) {
    this.x = x
    this.y = y }

  findHalfDistance (p) {
    let distance = new Point(this.x - p.x, this.y - p.y)
    return new Point(this.x - distance.x / 2, this.y - distance.y / 2)}}

class SierpinskiTriangle {
  constructor () {
    this.setUpCanvas()
    this.drawRectangleVertices()
    this.chooseRandomPoint() }

  setUpCanvas () {
    this.canvas = document.getElementById('canvas')
    this.x = this.canvas.width
    this.y = this.canvas.height }

  drawRectangleVertices () {
    this.canvas = this.canvas.getContext('2d')
    let p0 = this.x
    let p1 = this.x / 2
    this.topPoint = new Point(p1, 0)
    this.leftPoint = new Point(0, p1)
    this.rightPoint = new Point(p0, p1)
    this.trianglePoints = [this.topPoint, this.leftPoint, this.rightPoint]
    for (let index = 0; index < this.trianglePoints.length; index++) {
      this.drawPoint(this.trianglePoints[index]) } }

  chooseRandomPoint () {
    this.randomPoint = new Point(
      Math.floor(Math.random() * this.x),
      Math.floor(Math.random() * this.y)) }

  drawPoint (point) {
    this.canvas.beginPath()
    this.canvas.arc(point.x, point.y, 2.4, 0, Math.PI * 2)
    this.canvas.closePath()
    this.canvas.fillStyle = 'blue'
    this.canvas.fill() }


 chooseMidPoint () {
    let randomChoice = Math.floor(Math.random() * this.trianglePoints.length)
    return this.randomPoint.findHalfDistance(this.trianglePoints[randomChoice]) }

  start () {
    let MiddlePoint = this.chooseMidPoint()
    this.drawPoint(MiddlePoint)
    this.randomPoint = MiddlePoint }}
```

List 13- Fractals / Space Colonization Algorithm

****

```
let { tree, influenceDistance, killDistance } = declareVariables();

function setup () {
  createCanvas(500, 500)
  tree = new Tree() }

function draw () {
  background(0)
  tree.drawTree()
  tree.extendTree() }

class Segment {
  constructor (position, direction, parent, len) {
    this.position = position
    this.direction = direction
    this.parent = parent
    this.origDirection = this.direction.copy()
    this.getAttractedTimes = 0
    this.len = len
    this.reset = () => {
      this.direction = this.origDirection.copy()
      this.getAttractedTimes = 0 }

    this.buildNextSegment()

    this.drawSegment = () => {
      if (parent != null) {
        stroke(255)
        line(this.position.x,this.position.y,this.parent.position.x,this.parent.position.y)}}
}

  buildNextSegment () {
    this.buildNext = () => {
      let nextDirection = p5.Vector.mult(this.direction, this.len)
      let nextPosition = p5.Vector.add(this.position, nextDirection)
      let nextSegment = new Segment(nextPosition,this.direction.copy(),
        this,random() * 15)
      return nextSegment }}}

class AttractionPoint {
  constructor () {
    // Locate the point randomly in the space
    this.position = createVector(random(width), random(height - 170))
    this.isDone = false
    this.drawAttractionPoint = () => {
      ellipse(this.position.x, this.position.y, 3, 3) }} }



```

```
class Tree {
  constructor () {
    this.createAttractionpoints()
    this.segments = []
    let currentSegment = this.setUpRoot()
    currentSegment = this.checkRule1ForRoot(currentSegment)

    this.extendTree = () => {
      for (let i = 0; i < this.attractionPoints.length; i++) {
        let attractionPoint = this.attractionPoints[i]
        // Step 6
        let attractionDistance = influenceDistance
        let closestSegment = null
        for (let j = 0; j < this.segments.length; j++) {
          let segment = this.segments[j]
          let D1 = p5.Vector.dist(attractionPoint.position, segment.position)
          // Rule 2
          if (D1 < killDistance) {
            closestSegment = null
            attractionPoint.isDone = true
            break }
          // Rule 1
          else if (D1 < attractionDistance) {
            closestSegment = segment
            attractionDistance = D1 }}

        PerformAttractionProcess(closestSegment, attractionPoint) }

      // Remove attraction point after it is done
      this.removeLeaf()
      // Do only for segments that get attracted at least one time
      for (let i = 0; i < this.segments.length; i++) {
        let segment = this.segments[i]
        if (segment.getAttractedTimes > 0) {
          findAverageOfAttractionTimes(segment)
          this.segments.push(segment.buildNext())
          segment.reset() } } }

    // Draw the tree
    this.drawTree = () => {
      this.drawAllAttractionPoints()
      this.drawAllSegments() }}

  drawAllSegments () {
    for (let i = 0; i < this.segments.length; i++) {
      this.segments[i].drawSegment() } }

  drawAllAttractionPoints () {
    for (let i = 0; i < this.attractionPoints.length; i++) {
      this.attractionPoints[i].drawAttractionPoint() } }

  removeLeaf () {
      for (let i = 0; i < this.attractionPoints.length; i++) {
        if (this.attractionPoints[i].isDone) {
          this.attractionPoints.splice(i, 1) }}}

    checkRule1ForRoot (currentSegment) {
      let isAttracted = false
      while (!isAttracted) {
        // Calculate the distance between the currentSegment segment and every
        // attraction point in the space
        for (let i = 0; i < this.attractionPoints.length; i++) {
          let D1 = this.calculateD1(currentSegment, i)
          isAttracted = DoIfRule1Holds(D1, isAttracted) }
        currentSegment = this.DoIfRule1NotHold(isAttracted, currentSegment) }
      return currentSegment }

    calculateD1 (currentSegment, i) {
      return p5.Vector.dist(
        currentSegment.position,
        this.attractionPoints[i].position ) }

    // if the current segment can not be attached to any attraction point
    // Then create a new one and make it the current segment
    DoIfRule1NotHold (isAttracted, currentSegment) {
      if (!isAttracted) {
        let segment = currentSegment.buildNext()
        currentSegment = segment
        this.segments.push(currentSegment) }
      return currentSegment }

    setUpRoot () {
      // The position of the root in the center of the canvas
      let position = createVector(width / 2, height)
      // Make the root pointing up
      let direction = createVector(0, -1)
      let rootLength = random() * 24
      let root = createTreeRoot(position, direction, rootLength)
      this.segments.push(root)
      let currentSegment = root
      return currentSegment }

    createAttractionpoints () {
      this.attractionPoints = []
      for (let i = 0; i < 80; i++) {
        this.attractionPoints.push(new AttractionPoint()) } } }

  function findAverageOfAttractionTimes (segment) {
    segment.direction.div(segment.getAttractedTimes + 1) }
```

```
        // Steps 4 & 5
        function PerformAttractionProcess (closestSegment, attractionPoint) {
          if (closestSegment != null) {
            let newSegmentDirection = findCloserSegment(attractionPoint,
    closestSegment)
            newSegmentDirection.normalize()
            closestSegment.direction.add(newSegmentDirection)
            closestSegment.getAttractedTimes++ } }

        function findCloserSegment (attractionPoint, closestSegment) {
          return p5.Vector.sub(attractionPoint.position, closestSegment.position) }

        function DoIfRule1Holds (D1, isAttracted) {
          if (D1 < influenceDistance) {
            isAttracted = true }
          return isAttracted }

        function createTreeRoot (position, direction, rootLength) {
          return new Segment(position, direction, null, rootLength) }

        function declareVariables() {
            let tree;
            let influenceDistance = 150;
            let killDistance = 10;
            return { tree, influenceDistance, killDistance }; }

```

List 14- Fractals / L-Systems – 1-

****

```
window.onload = () => {
  createP(axiom)
  for (let index = 0; index < 3; index++) {
    generateLSystem() } }

const axiom = 'A'
let sentence = axiom
const { firstRule, secondRule } = MakeRules()

function setup () {}

function MakeRules () {
  const firstRule = { currentState: 'A', nextState: 'AABA' }
  const secondRule = { currentState: 'B', nextState: 'ABBBA' }
  return { firstRule, secondRule } }

function generateLSystem () {
  let index = 0 , nextSentence = ' '
  while (index < sentence.length) {
    let currentCell = sentence.charAt(index)
    switch (currentCell) {
      case firstRule.currentState:
        nextSentence += firstRule.nextState
        break
      case secondRule.currentState:
        nextSentence += secondRule.nextState
        break
      default:
        nextSentence += currentCell
        break }
    index++ }
  sentence = nextSentence
  createP(sentence) }
```

List 15- Fractals / L-Systems -2-

****

```
window.onload = () => {
  for (let index = 0; index < 4; index++) {
    generateFractalTree() } }

let rules = [],rotationAngle,axiom = 'F',sentence = axiom,branchLength = 80

makeRule()

function generateFractalTree () {
  branchLength *= 0.5
  let nextSentence = ''
  for (let index = 0; index < sentence.length; index++) {
    let current = sentence.charAt(index)
    let existed = false
    for (let j = 0; j < rules.length; j++) {
      if (current == rules[j].currentState) {
        existed = true
        nextSentence += rules[j].nextState
        break }}
    if (!existed) {
      nextSentence += current } }
  sentence = nextSentence
  createP(sentence)
  drawFractalTree() }

function drawFractalTree () {
  resetMatrix()
  stroke(0)
  translate(width / 2, height)
  interpretRules() }

function interpretRules () {
  for (let index = 0; index < sentence.length; index++) {
    let current = sentence.charAt(index)
    switch (current) {
      case '[':  push()
        break
      case ']':  pop()
        break
      case 'F':
                line(0, 0, 0, -branchLength)
                translate(0, -branchLength)
        break
      case '+':  rotate(rotationAngle)
        break
      case '-':  rotate(-rotationAngle)
        break } } }

function setup () {
  createCanvas(350, 350)
  rotationAngle = radians(40)
  createP(axiom)
  drawFractalTree() }

function makeRule () {
  rules[0] = {
    currentState: 'F',
    nextState: 'F[+FF][-FF]F[-F][+F]F' } }
```

List 16- Fractals / Barnsley Fern

****

```
let Coefficients = declareEquationCoefficients()
let x = Math.random()
let y = Math.random()

function declareEquationCoefficients () {
  return [
    { a: 0,     b: 0,     c: 0,     d: 0.16,  e: 0,  f: 0,    probabilityFactor: 0.01 },
    { a: 0.85,  b: 0.04,  c: -0.04, d: 0.85,  e: 0,  f: 1.6,  probabilityFactor: 0.85},
    { a: 0.2,   b: -0.26, c: 0.23,  d: 0.22,  e: 0,  f: 1.6,  probabilityFactor: 0.07 },
    { a: -0.15, b: 0.28,  c: 0.26,  d: 0.24,  e: 0,  f: 0.44, probabilityFactor: 0.07 }
 ]}

function setup () {
  createCanvas(400, 400) }

function draw () {
  for (let index = 0; index < 200; index++) {
    FindNextPoint()
    let { x1, y1 } = constrainDrawing()
    makeItGreen()
    strokeWeight(0.5)
    point(x1, y1) } }

function makeItGreen () {
  let greenColor = color(0, 128, 0)
  stroke(greenColor) }

function constrainDrawing () {
  let x1 = map(x, -2.5, 3, 0, width)
  let y1 = map(y, -1, 11, height, 0)
  return { x1, y1 } }

function FindNextPoint () {
  let randomValue = Math.random(),newX, newY
  for (let index = 0; index < Coefficients.length; index++) {
    if (randomValue < Coefficients[index].probabilityFactor) {
      ;({ newX, newY } = applyAffineTransformations(newX, index, newY))
      break }
    randomValue -= Coefficients[index].probabilityFactor
    ;({ newX, newY } = applyAffineTransformations(newX, index, newY))}
  x = newX
  y = newY }

function applyAffineTransformations (newX, index, newY) {
  newX =x * Coefficients[index].a +y * Coefficients[index].b +Coefficients[index].e
  newY =x * Coefficients[index].c +y * Coefficients[index].d +Coefficients[index].f
  return { newX, newY }}
```

List 17- Voronoi Diagrams

****

```
window.onload = () => plotVoronoiDiagram()
function declareVariables () {
  let cvs = document.getElementById(`canvas`)
  let ctx = cvs.getContext('2d')
  let w = cvs.width
  let h = cvs.height
  let x = (k = distance1 = distance2 = j = 0)
  let w1 = w - 2
  let h1 = h - 2
  let firstArray = new Array(100)
  secondArray = new Array(100)
  thirdArray = new Array(100)
  return { ctx, w, h, firstArray, secondArray, thirdArray, w1, h1, x } }

function fillArraysRandomly (firstArray, w1, h1) {
  for (let index = 0; index < 100; index++) {
    firstArray[index] = Math.floor(Math.random() * w1)
    secondArray[index] = Math.floor(Math.random() * h1)
    thirdArray[index] = generateHexColor() }}

function generateHexColor () {
  let generatedColor = '#'
  let hexAlphabet = '0123456789ABCDEF'
  for (let index = 0; index < 4; index++) {
    generatedColor += hexAlphabet[Math.floor(Math.random() * 16)] }
  return generatedColor }

function fillWithWhite (ctx, w, h) {
  ctx.fillStyle = 'white'
  ctx.fillRect(0, 0, w, h) }

function FindEculideanDistance (h1, w1, firstArray, x) {
  distance2 = Math.sqrt(h1 * h1 + w1 * w1)
  j = -1
  for (let index = 0; index < 100; index++) {
    distance1 = Math.sqrt( (firstArray[index] - x) * (firstArray[index] - x) +
        (secondArray[index] - k) * (secondArray[index] - k) )
    if (distance1 < distance2) {
      distance2 = distance1
      j = index }}}

function plotVoronoiDiagram () {
  let { ctx, w, h, firstArray, w1, h1, x } = declareVariables()
  fillWithWhite(ctx, w, h)
  fillArraysRandomly(firstArray, w1, h1)
  x = drawVoronoiRegions(h1, x, w1, firstArray, ctx)
  drawDataPoints(ctx, firstArray) }

function drawVoronoiRegions (h1, x, w1, firstArray, ctx) {
  for (k = 0; k < h1; k++) {
    for (x = 0; x < w1; x++) {
      FindEculideanDistance(h1, w1, firstArray, x)
      ctx.fillStyle = thirdArray[j]
      ctx.fillRect(x, k, 3, 3)}}
  return x }

function drawDataPoints (ctx, firstArray) {
  ctx.fillStyle = 'black'
  for (let index = 0; index < 100; index++) {
    ctx.fillRect(firstArray[index], secondArray[index], 4, 4) }}
```

List 18- Maze Generation / Recursive Backtracker

****

```
let { numOfColumns, cellSize, numOfRows, grid, currentCell, stack } =
declareVariables();

function createGrid () {
  numOfColumns = floor(width / cellSize)
  numOfRows = floor(height / cellSize)
  fillGridWithCells() }

function fillGridWithCells () {
  for (let row = 0; row < numOfRows; row++) {
    for (let column = 0; column < numOfColumns; column++) {
      let cell = new Cell(column, row)
      grid.push(cell) } } }

function generateMaze () {
  DoForInitialCell()
  DoForUnvisitedCells()

  function DoForUnvisitedCells () {
    let nextCell = currentCell.checkIfNeighboursVisited()
    if (nextCell) {
      stack.push(currentCell)
      removeWalls(currentCell, nextCell)
      currentCell = nextCell
        nextCell.visited = true }
     else if (stack.length > 0) { currentCell = stack.pop() } }

  function DoForInitialCell () {
    currentCell.visited = true
    currentCell.markAsVisited() } }

class Cell {
  constructor (column, row) {
    this.walls = [true, true, true, true]
    this.column = column
    this.row = row
    this.visited = false
    this.checkIfNeighboursVisited = () => {
      let neighbors = []
      let { top, right, bottom, left } = this.findNeighboursLocation(column,row)
      this.addToStack(top, neighbors, right, bottom, left)
      if (neighbors.length > 0) {
        let randomNumber = floor(random(0, neighbors.length))
          return neighbors[randomNumber] }
      else { return undefined } }
    this.markAsVisited = () => {
      let { x, y } = this.findCellLocation()
      colorVisitedCell(x, y) }
    this.displayCell = () => {
```

```
    let { x, y } = this.findCellLocation()
    stroke(255)
    this.drawWalls(x, y) } }

  findCellLocation () {
   let x = this.column * cellSize
   let y = this.row * cellSize
   return { x, y } }

  drawWalls (x, y) {
   this.drawTopWall(x, y)
   this.drawRightWall(x, y)
   this.drawBottomWall(x, y)
   this.drawLeftWall(x, y) }

  drawLeftWall (x, y) {
   if (this.walls[3]) line(x, y + cellSize, x, y) }

  drawBottomWall (x, y) {
   if (this.walls[2]) line(x + cellSize, y + cellSize, x, y + cellSize) }

  drawRightWall (x, y) {
   if (this.walls[1]) line(x + cellSize, y, x + cellSize, y + cellSize) }

  drawTopWall (x, y) {
   if (this.walls[0]) line(x, y, x + cellSize, y) }

  addToStack (top, neighbors, right, bottom, left) {
   addTopNeighbour(top, neighbors)
   addRightNeighbour(right, neighbors)
   addBottomNeighbour(bottom, neighbors)
   addLeftNeighbour(left, neighbors) }

  findNeighboursLocation (column, row) {
   let top = topNeighbourLocation(column, row)
   let right = rightNeighbourLocation(column, row)
   let bottom = bottomNeighbourLocation(column, row)
   let left = leftNeighbourLocation(column, row)
   return { top, right, bottom, left } } }

function leftNeighbourLocation (column, row) {
  return grid[findCellIndex(column - 1, row)] }

function bottomNeighbourLocation (column, row) {
  return grid[findCellIndex(column, row + 1)] }

function rightNeighbourLocation (column, row) {
  return grid[findCellIndex(column + 1, row)] }

function topNeighbourLocation (column, row) {
  return grid[findCellIndex(column, row - 1)] }

```

```
function addLeftNeighbour (left, neighbors) {
  if (left && !left.visited) neighbors.push(left) }

function addBottomNeighbour (bottom, neighbors) {
  if (bottom && !bottom.visited) neighbors.push(bottom) }

function addRightNeighbour (right, neighbors) {
  if (right && !right.visited) neighbors.push(right) }

function addTopNeighbour (top, neighbors) {
  if (top && !top.visited) neighbors.push(top) }

function colorVisitedCell (x, y) {
  fill(250, 250, 255, 200)
  rect(x, y, cellSize, cellSize) }

function findCellIndex (column, row) {
  return column < 0 || row < 0 || column > numOfColumns - 1 ||
  row > numOfRows - 1  ? -1 : column + row * numOfColumns }


function removeWalls (cell1, cell2) {
  removeWallForAdjacentCells(cell1, cell2)
  removeWallForOrthogonalCells(cell1, cell2) }

function removeWallForOrthogonalCells (cell1, cell2) {
  let index = cell1.row - cell2.row
  switch (index) {
    case -1:  cell1.walls[2] = false
              cell2.walls[0] = false
      break
    case 1:   cell1.walls[0] = false
              cell2.walls[2] = false
      break } }

function removeWallForAdjacentCells (cell1, cell2) {
  let index = cell1.column - cell2.column
  switch (index) {
      case -1:   cell1.walls[1] = false
                 cell2.walls[3] = false
      break
      case 1:    cell1.walls[3] = false
                 cell2.walls[1] = false
      break } }

function setup () {
  createCanvas(300, 300)
  createGrid()
  currentCell = grid[0] }
```

```
        function draw () {
          background(0)
          showGrid()
          generateMaze() }

        function showGrid () {
          for (let column = 0; column < grid.length; column++) {
            grid[column].displayCell() } }

        function declareVariables() {
           let currentCell, stack = [], grid = [], cellSize = 20, numOfColumns,
                                numOfRows;
    return { numOfColumns, cellSize, numOfRows, grid, currentCell, stack };}
```

List 19- Maze Generation / Reaction Diffusion Algorithm

****

```
let { mainGrid, auxiliaryGrid, Db, killRate, feedRate, Da } = declareVariables();

function setup () {
  createCanvas(300, 300)
  pixelDensity(1)
  fillGridWithOnlyChemical_A()
  fillPartiallyWithChemical_B() }

function fillPartiallyWithChemical_B () {
  let p = width / 2
  for (let i = p; i < p + 30; i++) {
    for (let j = p; j < p + 30; j++) {
      mainGrid[i][j].chemical_B_amount = 1 } } }

function fillGridWithOnlyChemical_A () {
  mainGrid = []
  auxiliaryGrid = []
  for (let x = 0; x < width; x++) {
    mainGrid[x] = []
    auxiliaryGrid[x] = []
    for (let y = 0; y < height; y++) {
      mainGrid[x][y] = { chemical_A_amount: 1, chemical_B_amount: 0 }
      auxiliaryGrid[x][y] = { chemical_A_amount: 1, chemical_B_amount: 0 }}}}

function draw () {
  background(255)
  calculateNewChemicalValues()
  loadPixels()
  colorPixels()
  updatePixels()
  swap() }

function colorPixels () {
  for (let x = 0; x < width; x++) {
    for (let y = 0; y < height; y++) {
      let { pixelLocation, grayScaleColor } = findGrayColor(x, y)
      colorSinglePixel(pixelLocation, grayScaleColor) } } }

function colorSinglePixel (pixelLocation, grayScaleColor) {
  pixels[pixelLocation + 0] = grayScaleColor
  pixels[pixelLocation + 1] = grayScaleColor
  pixels[pixelLocation + 2] = grayScaleColor
  pixels[pixelLocation + 3] = 255 }

function findGrayColor (x, y) {
  let pixelLocation = (x + y * width) * 4
  let chemical_A_amount = auxiliaryGrid[x][y].chemical_A_amount
  let chemical_B_amount = auxiliaryGrid[x][y].chemical_B_amount
  let grayScaleColor = floor((chemical_A_amount - chemical_B_amount) * 255)
  grayScaleColor = constrain(grayScaleColor, 0, 255)
  return { pixelLocation, grayScaleColor } }

function calculateNewChemicalValues () {
  for (let x = 1; x < width - 1; x++) {
    for (let y = 1; y < height - 1; y++) {
      let chemical_A_amount = mainGrid[x][y].chemical_A_amount
      let chemical_B_amount = mainGrid[x][y].chemical_B_amount
      applyEquationForChemical_A(x, y, chemical_A_amount, chemical_B_amount)
      applyEquationForChemical_B(x, y, chemical_B_amount, chemical_A_amount)
      constrainChemicalValues(x, y) }}}

function constrainChemicalValues (x, y) {
  auxiliaryGrid[x][y].chemical_A_amount = constrain(
  auxiliaryGrid[x][y].chemical_A_amount,0,1)
  auxiliaryGrid[x][y].chemical_B_amount = constrain(
    auxiliaryGrid[x][y].chemical_B_amount, 0,1 ) }

function swap () {
  let tmp = mainGrid
  mainGrid = auxiliaryGrid
  auxiliaryGrid = tmp }

function applyEquationForChemical_B (x,y,chemical_B_amount,chemical_A_amount) {
  auxiliaryGrid[x][y].chemical_B_amount =chemical_B_amount +
    Db * applyLaplaceConvolution(x, y, 'chemical_B_amount') +
    chemical_A_amount * chemical_B_amount * chemical_B_amount -
    (killRate + feedRate) * chemical_B_amount }

function applyEquationForChemical_A (x,y,chemical_A_amount,chemical_B_amount) {
  auxiliaryGrid[x][y].chemical_A_amount = chemical_A_amount +
    Da * applyLaplaceConvolution(x, y, 'chemical_A_amount') -
    chemical_A_amount * chemical_B_amount * chemical_B_amount +
    feedRate * (1 - chemical_A_amount) }

function applyLaplaceConvolution (x, y, value) {
  let wieght = 0
  let centerCellWieght = mainGrid[x][y][value] * -1
  wieght += centerCellWieght
  wieght = calculateAdjacentCellsWeight(wieght, x, y, value)
  wieght = calculateDiagonalCellsWeight(wieght, x, y, value)
  return wieght }

function calculateDiagonalCellsWeight (wieght, x, y, value) {
  wieght += mainGrid[x - 1][y - 1][value] * 0.05
  wieght += mainGrid[x - 1][y + 1][value] * 0.05
  wieght += mainGrid[x + 1][y - 1][value] * 0.05
  wieght += mainGrid[x + 1][y + 1][value] * 0.05
  return wieght }

```

```
function calculateAdjacentCellsWeight (wieght, x, y, value) {
  wieght += mainGrid[x - 1][y][value] * 0.2
  wieght += mainGrid[x + 1][y][value] * 0.2
  wieght += mainGrid[x][y + 1][value] * 0.2
  wieght += mainGrid[x][y - 1][value] * 0.2
  return wieght }

function declareVariables() {
  let mainGrid, auxiliaryGrid, Da = 1, Db = 0.5, feedRate = 0.0545, killRate = 0.062;
  return { mainGrid, auxiliaryGrid, Db, killRate, feedRate, Da }; }

```

List 20- Procedural Flowers

****

```
function setup () {
  createCanvas(600, 600) }

function createFlowerPattern () {
  beginShape()
  for (let theta = 0; theta < TWO_PI; theta += 0.04) {
    let radius = calculateRadius(theta)
    let { x, y } = convertPolarToCartesian(radius, theta)
    vertex(x, y) }
  endShape(CLOSE) }

function calculateRadius (theta) {
  return 150 * cos(k * theta) }

function convertPolarToCartesian (radius, theta) {
  let x = radius * cos(theta)
  let y = radius * sin(theta)
  return { x, y } }

function startDrawingFromCenter () {
  translate(width / 2, height / 2) }

function draw () {
  background(255)
  startDrawingFromCenter()
  createFlowerPattern() }
```

List 21- Infinite Terrain / Main

****

```
using UnityEngine;
using System.Collections;
using System;

public class Main : MonoBehaviour
{
    public float x1 = 0.1F;
    public float y1 = 0.1F;
    public float seed = 0.25F;
    public Vector3 coordinateAdjustment;
    public GameObject tile;
    public Sprite[] mainImages;
    public int width = 40;
    public int height = 40;
    public Transform Player;
    public float DistanceBeforeRedrawing = 9;
    public SpriteRenderer[,] theRenerers;

    private void RedrawTerrainForever()
    {
        MoveTerrainToSurroundPlayer();
        RenderTerrainUsingPerlinNoise(); }

    private void MoveTerrainToSurroundPlayer() {
        transform.position = new Vector3((int)Player.position.x, (int)Player.position.y,
Player.position.z); }

    private void RenderTerrainUsingPerlinNoise() {
        for (int i = 0; i < width; i++) {
            x1 += seed;
            for (int j = 0; j < height; j++) {
                y1 += seed;
                theRenerers[i, j].sprite = mainImages[ Mathf.RoundToInt((
Mathf.PerlinNoise(x1,y1))*8)]; } } }

    private void SetUpTile() {
        coordinateAdjustment = new Vector3(0 - width / 2, 0 - height / 2, 0);
        int OrderOfLayer = 0, i = 0;
        while (i < width)
        {
            for (int j = 0; j < height; j++)
            {
                tile = new GameObject();
                tile.transform.position = new Vector3(i, j, 0) + coordinateAdjustment;
                SpriteRenderer overlap = theRenerers[i, j] = tile.AddComponent
<SpriteRenderer>();
                overlap.sortingOrder = OrderOfLayer--;
                tile.transform.parent = transform;
            }
            i++; }   }

    private bool ShouldRedraw() {
        return DistanceBeforeRedrawing < Vector3.Distance(Player.position, transform.position) ? true : false; }

    private void Start() {
        theReneres = new SpriteRenderer[width, height];
        SetUpTile();
        RedrawTerrainForever(); }

    private void Update() {
        if (ShouldRedraw()) {
            RedrawTerrainForever(); } } }
```

List 22- Infinite Terrain / Move

****

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour
{
    private float defaultSpeed = 1;
    private KeyCode increaseSpeed = KeyCode.S;

    public float DefaultSpeed { get => defaultSpeed; set => defaultSpeed = value; }
    public KeyCode IncreaseSpeed { get => increaseSpeed; set => increaseSpeed =
value; }

    private Vector3 selectDirection() {
     return   new Vector3(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical"), 0);
}

    private void moveTheScene() {
        if (Input.GetKey(increaseSpeed))
        { fastSpeed(); }
        normalSpeed(); }

    private void normalSpeed() {
        transform.Translate(selectDirection() * defaultSpeed * Time.deltaTime);}
    private void fastSpeed()
    { transform.Translate(selectDirection() * 3 * Time.deltaTime); }

    void Start() { }

    void Update() { moveTheScene(); } }
```