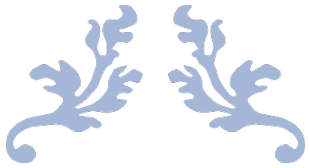
2017

Procedural Generation: An Algorithmic Analysis of Video Game Design and Level Creation

Logan Bond
Ouachita Baptist University



PROCEDURAL GENERATION

An Algorithmic Analysis of Video Game Design and Level Creation

April 26, 2017

Logan Bond

Carl Goodson Honors Program

TABLE OF CONTENTS

CONVENTIONS IN THIS TEXT

- All written text will be in Times New Roman, 12pt font – font size varies with headings

  This sentence is an example.

- Any code snippet will be in Courier New, 12pt font *with* syntax highlighting (barring a color printer)

```
public void Start()
 {
       ArrayList gamePieces = GenerateList();
 }
```

- Comments on parts of code will be used to explain key components.

```
// A comment will follow the characters '//'
// all characters on the line after '//'
// are ignored during compilation
```

- Code supplied in the Appendix will have file names corresponding to the environment it was typed in. A `.pde` file extension corresponds to Processing while a `.cs` file extension corresponds to a C# script from Unity.
- There is also a bit of *pseudo*-code, which is a simplified version of code. As the name suggests, it is *fake* code; it will not compile. It is a basic sample of what might be very complex code, or code written in a language that differs from the two mentioned in Materials Used. The pseudo-code is labeled appropriately and should not be considered *runnable* code.

MATERIALS USED

- All code is written for use with Unity3D (Personal Edition) or Processing.
- All code is written with Visual Studio (Community Edition 2015), Processing, or Unity's built-in MonoDevelop.
- Languages of choice are C# and Processing (comparable syntax, see Conventions Used).
- All code is compiled and run on either a custom-built desktop PC or a MacBook Pro.
- Game Design Documents are documents that outline development for a game. They are a guideline for the developer as the developer begins implementing the game. Part of this analysis is to provide practical applications for each algorithm. Therefore, at the end of each section will be posted a Game Design Document. This document will not define or outline an entire game, but will give short description of its expected behavior.

PREFACE

I happened upon Computer Science by chance; never in a million years could I have foreseen my desire to pursue computing. However, once I wandered into Programming 1, I knew that I had discovered what I was meant to do. I have always liked puzzles – not the type that one breaks out and puts together with family at the dinner table. No, I instead prefer brain benders that required deep thought and strategy to complete. This love of challenge combined with the newfound respect for Computer Science led me to this thesis topic: procedural generation.

Procedural generation is a necessity that most game developers use at one point or another. Whether it be generating random worlds, levels, terrains, dungeons, obstacles, meaningful dialogue, game boards, or data in general, any game developer of esteem should have procedural generation in his or her toolkit. My approach to this thesis is to conduct an in-depth analysis of a few procedural generation techniques and analyze the code of each. I also propose a practical application (game) centered around each algorithm of subject. My goal is to target some of the most popular approaches to problems and examine each in depth. Every video game is ideally unique; thus, there is no perfect way to explain and describe every type, example, and use-case for every procedural generation algorithm in existence.

It is necessary to add a disclaimer to say that every bit of code in this document was written by me. As part of the analysis, I must examine through writing; I steer far from copy and paste. This does not mean that I originally designed each bit of code. I have not produced an expansive library of my own games that involve procedural generation and therefore cannot produce original code for each one. I do not have the mental capacity to come up with an idea for a game to provide a specific use-case for every possible procedural generation algorithm. Credit

will be given where credit is due, but the point of this disclaimer is the contracted agreement between the reader and me that each code snippet was written by me, whether pulled from my own mind or another's.

INTRODUCTION

Procedural generation is a method for generating mass quantities of data algorithmically rather than manually. One perfect example of this is the recently famous *No Man's Sky*, a video game where the entire marketing scheme was structured around its procedurally generated universe. The game's trailer and advertisements promised its players 18,446,744,073,709,551,616 unique planets[1], all of which were procedurally generated. In other words, the developers did not create exclusive profiles for every single planet, but instead programmed the game in such a way that the planets were built from the code. This method of content creation is the essence of procedural generation.

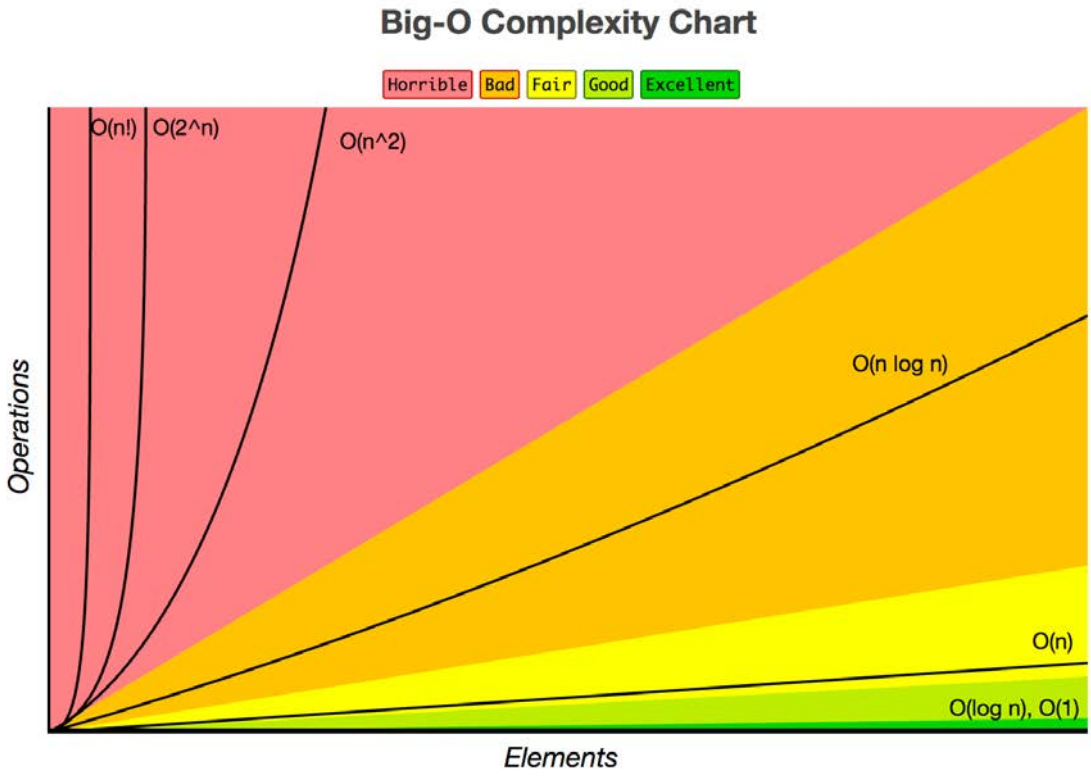
Procedural generation has a broader application; it is a solution to many problems. It can take many different forms, and there are many different algorithms that have been written to accomplish many scenarios. The purpose of this analysis was to choose four common and interesting procedural generation algorithm archetypes, analyze each of them in the same way, and then demonstrate a practical application for each of them. Procedural generation techniques are some of the more convoluted programming techniques in existence; therefore, extra emphasis is placed on writing, demonstrating, and applying each algorithm.

Each algorithm's runtime complexity was also studied. The complexity of an algorithm is denoted as a function of input (Big-O): $O(n)$. The input, n, varies between algorithms but represents the size of the data necessary for the algorithm. Thus, define n each time the complexity is mentioned. Algorithmic runtime complexities typically vary mathematically from

[1] "How 4 Designers Built A Game With 18.4 Quintillion Unique Planets." Accessed April 16, 2017

$O(1)$ to $O(n!)$. Figure 0.1 is a guide to further explain this concept. As input size (the number of elements) increases, so too does the amount of operations, and thus the more time and computationally expensive an algorithm becomes. Due to this fact, it is extremely important to note the complexity of each algorithm because game development relies heavily on user experience. If the algorithm is not efficient enough to smoothly generate game content for the user, then the developer must find another design.

Figure 0.1 – Big-O runtime complexity chart.



Picture from http://bigocheatsheet.com/

One other item to note is the category of each algorithm. There are several categories of procedural generation, but the algorithms of this study fall into one of two types: ontogenetic or teleological procedural generation. The difference between the two is in the approach to a given problem. *Ontogenetic* algorithms are ad hoc algorithms; they try to replicate an environment through any means necessary. *Teleological* algorithms attempt to recreate an environment by

reproducing its nature. For example, suppose a game that required a procedurally generated three-dimensional terrain. An ontogenetic algorithm would try to replicate a real world environment by *randomly* creating mountains and placing objects like trees or rocks at random. A teleological approach would *model* the movement of the Earth's crust to replicate tectonic plate movements and build mountains. Once mountains formed, the teleological approach would *simulate* rain to erode the mountains into a proper form.[2] In practice, ontogenetic algorithms are faster to write and usually faster in terms of complexity. Ontogenetic algorithms are more common in game design because of this fact. However, the teleological approach would generate a much more realistic landscape. Each algorithm in this study is categorized in this way to further examine their behavior and purpose.

These four algorithms mean something to me as a game developer. Each one of them could easily be used as a cornerstone for a game that I have prospected. Therefore, I have proposed a basic Game Design Document for each of the algorithms. The Game Design Documents follow a very trimmed version of the industry standard for this type of document; however, they provide a practical way to demonstrate how each game could (and is planned to) be made. It is my hope that the reader enjoy their laughableness as much as I do.

[2] Procedural Content Generation Wiki." Accessed April 16, 2017

ANALYSIS

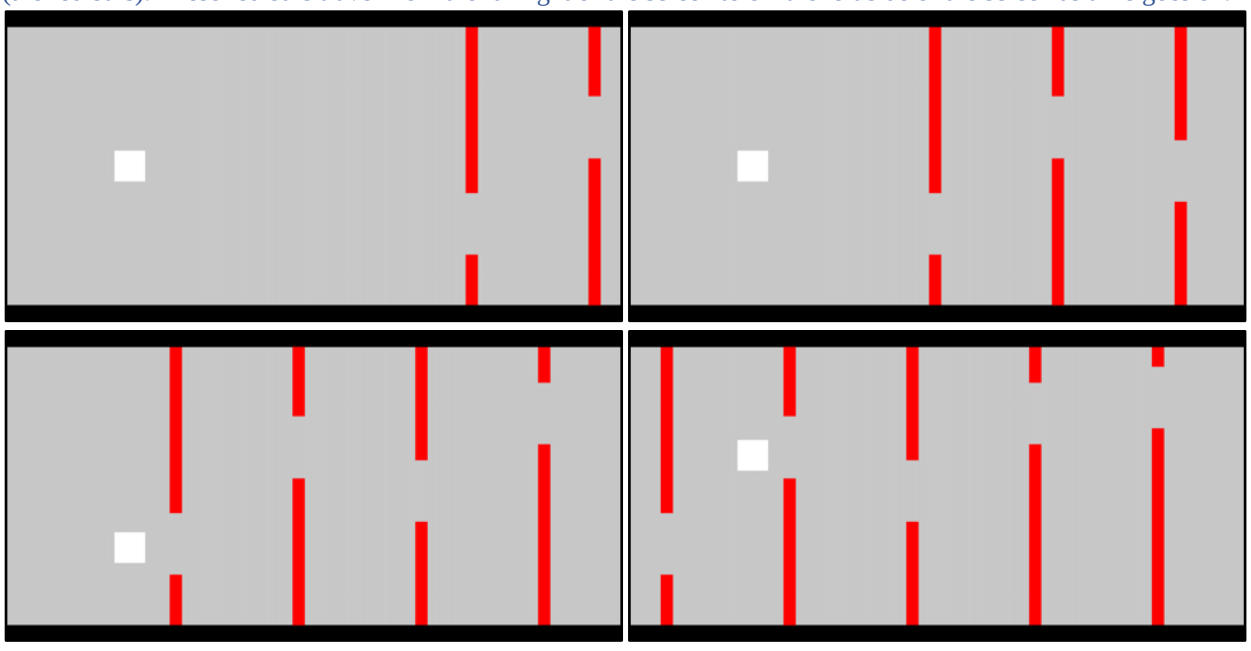
I chose to analyze four of many algorithms for procedural generation, and the same analysis was applied to each algorithm. The algorithms were chosen based on interest, applicability, and variety. These algorithms are some of the most common forms of procedural generation used today. For each algorithm, I first give an overview of where the algorithm originated, then briefly explain what it is and does, and give a demonstration referencing snippets of code. I also venture into more detail about each algorithm's complexity and a little of the math behind it. This detailed examination further explains how the algorithm works. Every analysis then ends with a section on the algorithm's applicability with a Game Design Document. These documents were created based on code that I wrote either in Unity or in Processing. They outline how each algorithm can be applied to a simple game idea that I have had throughout my experience as an amateur game developer. I cannot possibly create a full-fledged version of each game within a reasonable amount of time. Therefore, these Game Design Documents simply give a general overview of each game, its key features, and expound on how each procedural generation technique makes the game unique.

CHUNK-BASED

Chunk-based procedural generation exemplifies the definition of ontogenetic procedural generation. These algorithms are on-the-fly, ad hoc programs that are commonly used for infinitely generated games such as infinite runners, of which there are a plethora of late. These algorithms are considerably easier to conceptualize than most procedural generation algorithms. Most other procedural generation methods have papers and articles written about them once they are contrived; however, chunk-based procedural generation algorithms are so varied that they would be impossible to categorize in this way.

Many recently popular games rely heavily on chunk-based, generated content. A few of these include but are not limited to: Crossy Road, Flappy Bird, Cube Runner, Chicken Scream, Temple Run (all of them), Super Mario Run, and Super Mega Neo Pug. This field of algorithms is one of the simplest forms of procedural generation in existence. Chunk-based generated content can be defined as content generated in chunks. These algorithms usually follow the same

Figure 1.1 – A chunk-based algorithm game. The player (the square) must traverse between the obstacles (the red bars). These red bars travel from the far right of the screen to off the left side of the screen as time goes on.



three-step process for content creation. Firstly, they generate a pool of objects, obstacles, course sections, or whatever content the game requires. Secondly, these algorithms select an object randomly from this pool and slide it toward the player. Lastly, once an object has fallen off the screen, it is returned to the pool of objects to randomly select from again. Though there may be slight changes to the content in the pool as the game moves on such as making the content more difficult or spawn faster, but the foundation of the algorithm is this three-step process.

Figure 1.1 depicts a basic game implementing a chunk-based procedural generation algorithm. The red obstacles depicted in Figure 1.1 are generated with gaps at random heights and then placed into a list. Items are then randomly selected from the list, and the selected chunk begins traveling toward the player whose latitudinal position does not change. As a chunk travels off the left side of the screen, the chunk is toggled to reposition itself back to just off the right edge of the screen. Chunks are continually and randomly selected from the pool based on a time interval, which creates the gap amongst obstacles.

Again, it is important to note that these algorithms vary in terms of breadth; however, their variance in depth is not as complicated. They follow the same approach to problems, and these algorithms have a typical runtime complexity of $O(n)$ at game start, an efficient runtime complexity, where n is the size of the chunk pool. Once the game has started, the complexity trims to $O(k)$, each frame, where k is the number of chunks on the screen and $k \leq n$. Every frame, offset by the gap between chunks, the pool is sifted through to find the next chunk to render. Next, each chunk currently on the screen must be examined to check that it has not traveled beyond the scope of the play space. This runtime complexity is very quick, and the algorithm is also memory-efficient with the recycling nature of the elements on screen.

The code used to generate Figure 1.1 is in the Appendix. There are three key elements: the driver, the player, and the chunk. The player is simply the cube rendered on the screen. The chunk consists of two vertical bars, as shown in Figure 1.1. The sizes of the vertical bars are randomly generated based on a random number generator while ensuring there is space enough for the player to fit between the gap in the two bars. The driver creates a pool of these chunks and begins drawing them to the screen at random, checking for those going out of view to be returned to the pool.

HoverCube

Game Design Document

Description of Game:

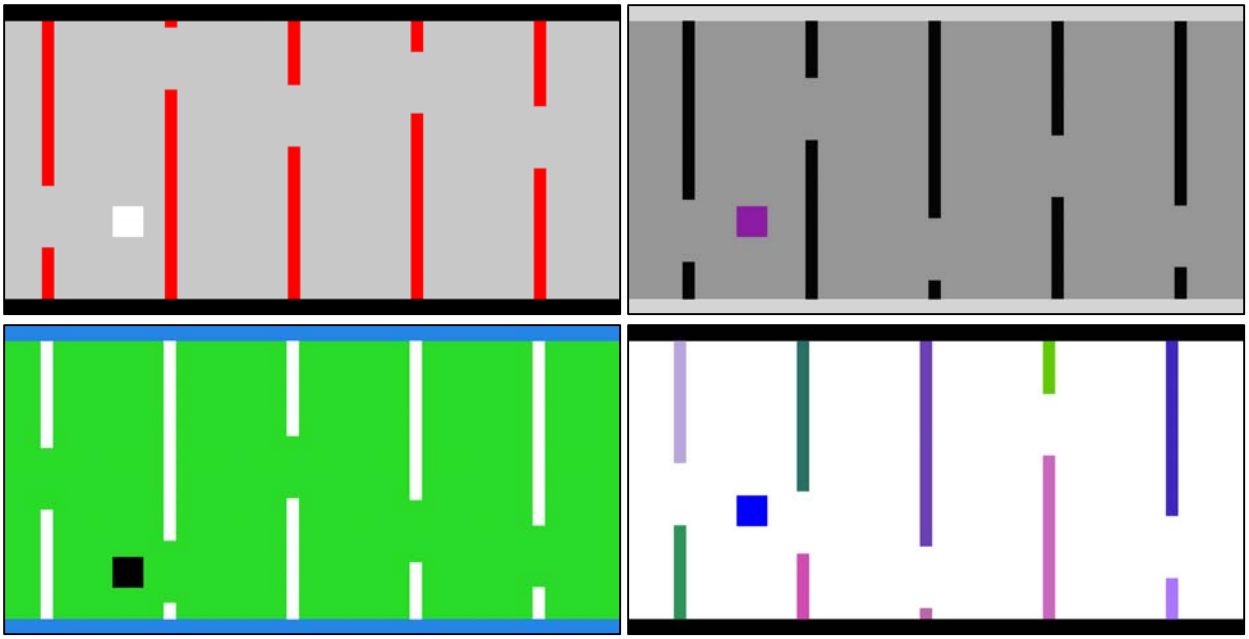
HoverCube is another infinite runner-type game. It can easily be viewed as a cross between Crossy Road, the infamous Flappy Bird, and the new Chicken Scream. The physics and style mimic that of Flappy Bird; objects approach the player as the player tries to navigate them. Score is based on the player's ability to dodge the obstacles. The player moves based on the scream mechanic from Chicken Scream: the louder the player screams, the more vigorous the jump. The crossover from Crossy Road is the artwork and character selection. As the player accumulates points, more characters are unlocked or bought. The player can also gain currency through watching advertisements. Once the player selects a different character, the entire artwork is reskinned to match the theme of the character. The goal is not to reinvent the wheel with this game; well-done infinite runners are the favorites of today.

Key Features:

- The menu screen is generic with emphasis on the cube.
- The menu includes a character selection screen where the player may select from any number of characters to play.
- Each character comes preloaded with a unique theme and skin for the gameplay.
- The player has unlocked, default characters that are just recolors of the original artwork.
- Every character is cube-based, to further add to the ridiculousness.
- The obstacles are randomly generated based on a random number generator.

- The pool is created from these obstacles and they are randomly rendered based on another random number generator.
- The player falls with gravitational acceleration, each character maintaining their own weight.
- The player must scream into the device microphone to maneuver the player upward, scaled to the weight of the character.
- The score is based on the number of pixels traversed, normalized to 100 pixels per point.

Screen Shots of Default Character Reskinned Artwork:

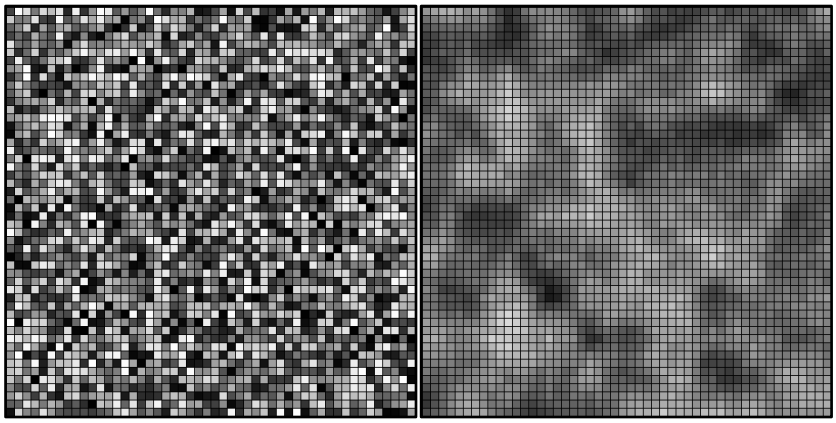


PERLIN NOISE

Perlin noise, created by Ken Perlin in 1983, was created to develop "highly realistic Computer Generated Imagery," as Perlin himself put it in his 1985 article "An Image Synthesizer".[3] He was later awarded an Academy Award for Technical Achievement for his work in 1997. He never applied for any patents for his algorithm; it was merely born out of his frustration for the current method of computer generated imagery. However, later, he was granted a patent for some use cases such as simplex noise, which simplifies his original, more complicated algorithm.[4]

The Processing Reference defines Perlin noise as "a random sequence generator producing a more natural, harmonic succession of numbers than that of [a] standard random [number generator]."[5] This description places the Perlin noise algorithm into the ontogenetic category because rather than model a natural process, it recreates environments based on a random number generator, though with smoothing. This *smoothing* aspect is a practical definition for *gradient noise*, meaning the generated grids have a more gradient color scheme. It is widely used for

Figure 2.1 – Randomly generated noise (Figure 2.1A) on the left, and Perlin noise (Figure 2.1B) on the right.

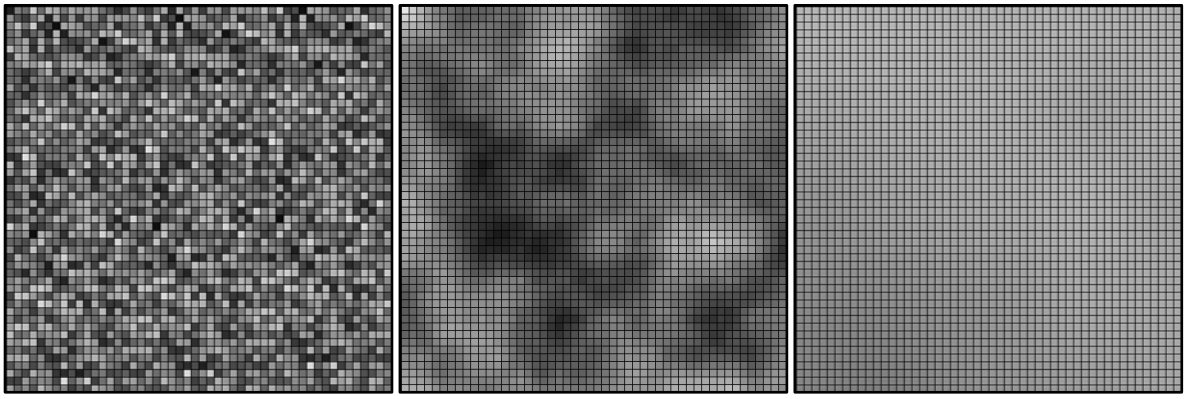


[3] Perlin, Ken. "An image synthesizer." *ACM SIGGRAPH Computer Graphics* 19, no. 3 (1985): 287-96.
[4] Perlin, Ken. Original Perlin Noise Source Code. Raw data.
[5] Processing Foundation. "Noise() \ Language (API) \ Processing 2."

computer graphics because of its ability to smooth edges and create more lifelike textures. The noise has a pseudo-random appearance, even though all its cells are the same size. Thus, the algorithm is scalable and controllable to two, three, four, or any number of dimensions. It can also be copied and used upon itself to create even more variety.

Figure 2.1 is a demonstration for comparison. Perlin noise generates smooth textures via the following three steps: divide a canvas into a grid, assign values to each node in the grid, and then smooth across each node. Figure 2.1A is an example of randomly generated noise, created by means of a random number generator; this figure shows no smoothing whatsoever. Figure 2.1B shows a grid with the Perlin algorithm applied. Both grids represent a 50x50 grid of values between 0 and 255 calculated using Processing's API. These values are then used as to create a grayscale fill, 0 for black and 255 for white. The grayscale fill was applied to each node before it

Figure 2.2 – Three grids with the Perlin noise algorithm applied, each with their own noise scale value to demonstrate smoothness. In order from left to right, the noise scale values are 1, 0.1, and 0.01.



was drawn. The grid with the applied Perlin algorithm calculated a *noise value* per node based on a scale, again relating back to the Perlin algorithm's ability to be scalable.

This *noise scale* value determines the amount of smoothness between associated nodes. Figure 2.2 demonstrates this concept with three different grids calculated with distinct noise scale values. These noise scale values are typically low, decimal values; the Processing

Figure 2.3 – Processing code to create a random number generator based grid.

```
void setup() {
  size(500, 500);
  int gridSize = 500;

  int sideLength = 10;

  //Random, no Perlin
  for (int i = 0; i < gridSize; i++) {
    for (int j = 0; j < gridSize; j++) {
      int noise = (int)random(0, 256);
      fill(noise);
      rect(i * sideLength, j * sideLength,
           sideLength, sideLength);
    }
  }
}
```

Reference defaults this value to 0.2 in its demonstration. Figure 2.1B was generated with the default value while Figure 2.2 exhibits values ranging from 1 to 0.01.

The code to generate Figures 2.1 and 2.2 is remarkably simple. Figures 2.3 and 2.4 show the Processing code snippets which were used to generate Figures 2.1 and 2.2. The noise scale value is set to its default value in Figure 2.4. The grids were generated using Processing's built in function for generating the Perlin value.

There are a few simplifications of Perlin's algorithm, most notably the Simplex Noise algorithm. Perlin's original algorithm ran with a complexity of $O(2^n)$, an exponential time algorithm, where n is the number of dimensions, which is inefficient in practice. For each node within the grid, the

Figure 2.4 – Processing code to generate Perlin noise based grid.

```
void setup() {
  size(500, 500);
  int gridSize = 500;

  int sideLength = 10;

  //Perlin Stuff
  float noiseScale = 0.2;
  for (int i = 0; i < gridSize; i++) {
    for (int j = 0; j < gridSize; j++) {
      float noiseVal = noise(i*noiseScale, j*noiseScale);
      fill(noiseVal*255);
      rect(i * sideLength, j * sideLength, sideLength,
sideLength);
    }
  }
}
```

distance is computed between each corner of the node and the point at which the node resides. This distance is then multiplied by the noise value associated with the node. In terms of vectors, the dot product between the gradient vector (noise value), and the distance vector is computed. Thus, in a two-dimensional system, with each node having four corners, eight distances and dot products must be calculated. This computation results in the $O(2^n)$ run-time complexity. Simplex noise simplifies this complexity to $O(n^2)$, a more reasonable, polynomial time algorithm. The Simplex noise algorithm was also developed by Perlin because he knew the limitations of his algorithm. Simplex noise is limited to usage in higher dimensions, however, but contains fewer "directional artifacts", meaning less overhead. It does not perfectly recreate the refined results of the original algorithm. Simplex noise is much easier in practice though, and it is also easier to implement in hardware.[6]

Perlin maintains his original algorithm and has it posted online on his personal blog.[7] The original code is written in C, a language residing outside of the scope of this research. Therefore, pseudo-code based on the original algorithm is provided in the Appendix. It is quite well documented, though the actual code looks extremely convoluted. The pseudo-code only represents the algorithm as applied to a two-dimensional system for ease of implementation.

[6] "Perlin noise." Wikipedia. March 22, 2017.
[7] Perlin, Ken. Ken's Academy Award. Accessed April 25, 2017.

Perlin noise is often used for generating maps and textures in video games. Usually, a map or game world is a mesh of differing height values at different points across a plane. This scenario is exactly how Perlin noise can be used to generate smoother maps then dividing a flat plane into a grid and then assign random values to correspond to height values. This approach would generate a jagged map with stark transitions. Figure 2.5 shows a procedurally generated mesh via Perlin noise in Unity with coloring based on the noise values for each node in the grid. Figure 2.6 shows how these noise values can be extrapolated to correspond with height values. In Unity, these height values can then be applied to the mesh to create a three-dimensional terrain. The colorization is rough due to the nature of the demonstration. If more careful attention were paid to the color selection, the mesh could be become even more realistic.

Figure 2.5 – Procedurally generated 2D mesh in Unity with colorization based on noise values.

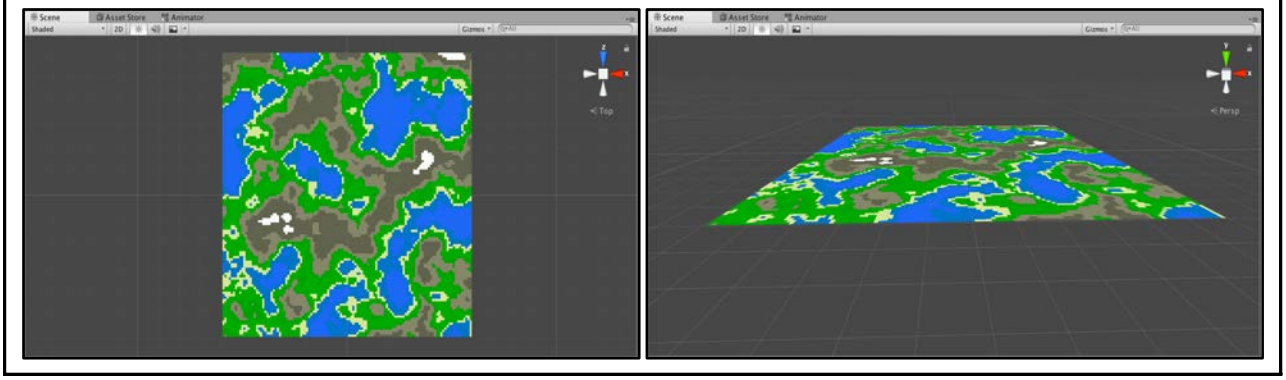
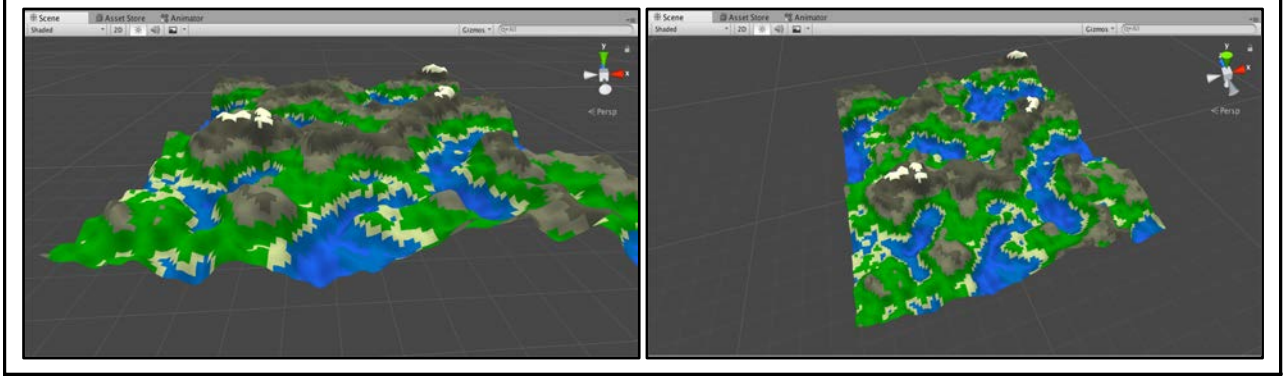


Figure 2.6 – The two-dimensional mesh from Figure 1.5, but noise values extrapolated to height values.



Survive the Night

Game Design Document

Description of Game:

Survive the Night is a nightmare survival game, with a twist. The game begins during the day, and the player can rummage around to look for supplies, weapons, ammo, and manufacturables. As the sun sets, the music gets more eerie, and it becomes obvious to the player that shelter must be either built or found. Once night officially falls, enemies begin to spawn. This cycle continues in waves, where each day/night cycle is one wave. As the game advances, the player has the ability to build more complex items to help defend through the night. Also, more enemies spawn each wave, hounding the player and creating a heightened sense of danger. The game cannot be won, it is simply a survival game where highscores are based on how long players survive. The catch is that enemies only move while the player is looking away from them; in essense, when the camera viewport is facing away. This gives the player the option to hunt the enemies down. As the distance between the player and a given enemy minimizes, the enemy begins to move faster if it can move.

Key Features:

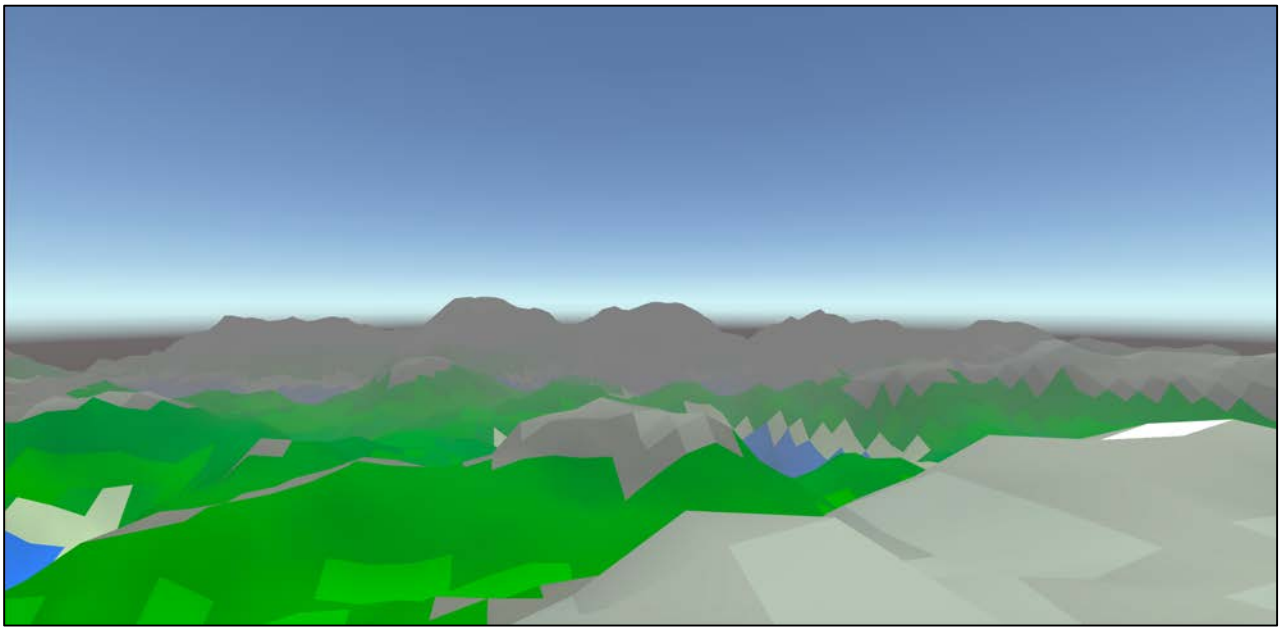
Survive the Night is a game of options:

- The world is procedurally generated (via Perlin Noise) each time the game loads.
- The supplies, ammo, or any interactable item are also procedurally placed across the game world, as are trees and environmental structures.
- To reduce the complexity of the procedurally generated content, there will be a small set of *standard* structures generated in every map (e.g. 1 cabin, 1 estate, and a standard

weapons loadout); these items will be procedurally placed each game, but it will be the same list of items.

- The player's initial spawn point is chosen at random, within a safe node of the generated Perlin noise grid.
- The enemies will also procedurally spawn, provided a minimum safe distance from the player (which grows smaller as the waves advance).
- Player health and HUD will be the same each game; essentially the player will start with nothing each round and have a given amount of time to prepare for the night.

First Person View of Perlin Noise Generated World (Daytime):



Separation Steering

Separation Steering, or flocking, is a unique procedural generation algorithm with two distinct primary applications. The most common example is the *boids* demonstration. A *boid* is a bird-oid object created by Craig Reynolds for his artificial life algorithm in 1986. His flocking algorithm was a successful attempt to model the movement of birds (or other flocking animals or insects) in real life without individually programming the movement of each bird. In fact, Reynolds' algorithm is not technically a procedural *generation* algorithm because it does not generate anything; however, pieces of his algorithm can be utilized for explicit generation, which I describe later. In his paper "Flocks, Herds, and Schools: A Distributed Behavioral Model", Reynolds states that the boids mimic the average particle system. "The simulated flock is an elaboration of a particle system, with the simulated birds being the particles. The aggregate motion of the simulated flock is created by a distributed behavioral model much like that at work in a natural flock; the birds choose their own course."[8] Thus, flocking is a teleological algorithm because it simulates the nature of birds, allowing them to select their own paths.

Figure 3.1 – Graphical demonstration of the three behavioral units of the flocking algorithm.

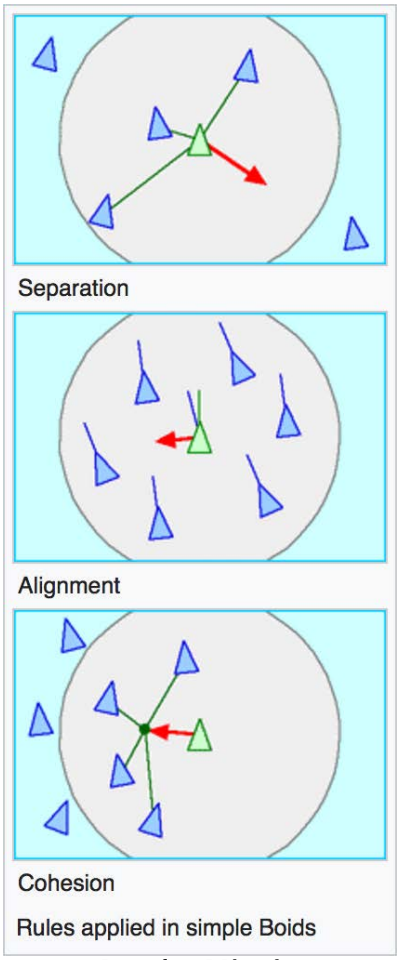


Image from Wikipedia: https://en.wikipedia.org/wiki/Boids

[8] Reynolds, Craig W. "Flocks, Herds and Schools: A Distributed Behavioral Model."

Reynolds' flocking algorithm involves three key components: separation, alignment, and cohesion. To determine how a group of entities will steer, a single entity will look at each entity within a *neighborhood radius*, some predefined radius extending out from the center of the entity. Each neighbor within this radius is added to a list of neighbors for a single entity, and this process is done for every entity in the considered system.

Separation is applied to each boid's list of neighbors, examining each neighbor. The goal is for the boids to be minimally distant from one another but not on top of each other; thus, the necessity of the separation step. This examination considers the distance from each neighbor to the boid in question. Each neighbor's directional vector is then normalized and divided by the distance from the considered boid. This value is added to a static *separation force*, which is used to literally steer the considered boid.

Alignment, after separation, tries to maintain the direction of each boid in congruence with its neighbors. The list of neighbors is iterated through again, averaging the neighbors' directional vectors, which returns the desired direction. The current directional vector of the considered boid is then subtracted from this desired direction and reapplied to itself. This is applied to each boid, iterating through and calculating a desired direction for all boids in the system.

Lastly, *cohesion* steers each boid inward toward its neighbors. Again, the list of neighbors is iterated over, however, cohesion calculates the average position vector of each neighbor, not direction. Cohesion is also applied to each entity; this is an ad hoc center of mass calculation, which is used to maintain each entity flocking by its neighbors. Cohesion, with

separation and alignment, creates the flocking simulation.[9] A larger demonstration of the bird flocking algorithm can be found in the Appendix.

With little modification, and the addition of a few extra calculations, the flocking algorithm can be extrapolated to random dungeon generation, an ontogenetic algorithm. One can draw a random group of rectangles (dungeon rooms) to the screen and then apply the same logic to the squares from Reynolds' flocking algorithm. Figure 3.2 demonstrates this fact. The few modifications include removing the cohesion factor. The separation steering component of the flocking algorithm is the key; however, the cohesive factor becomes moot (hence the name: *separation steering*). The goal is to procedurally generate dungeon rooms that are not overlapping and spread apart, exhibiting modified cohesion. Thus only a few rooms are kept once they have all been separated. This selection process is based on some population percentage density. Figure 3.3 shows the final product with crude lines drawn to signify tunnels between the rooms. These room mappings can be used as individual dungeons or individual floors within a dungeon. The crux of the algorithm is still

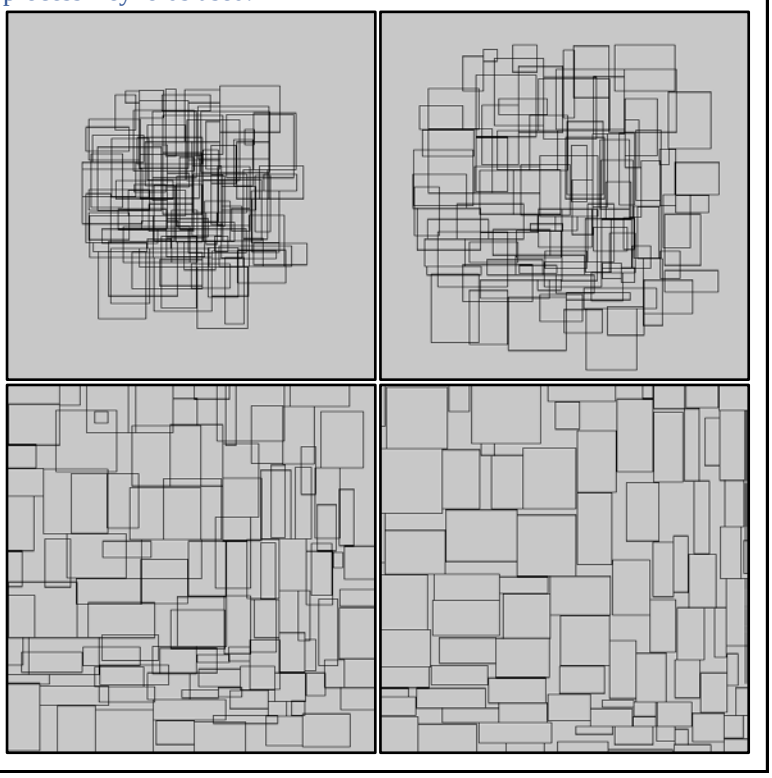
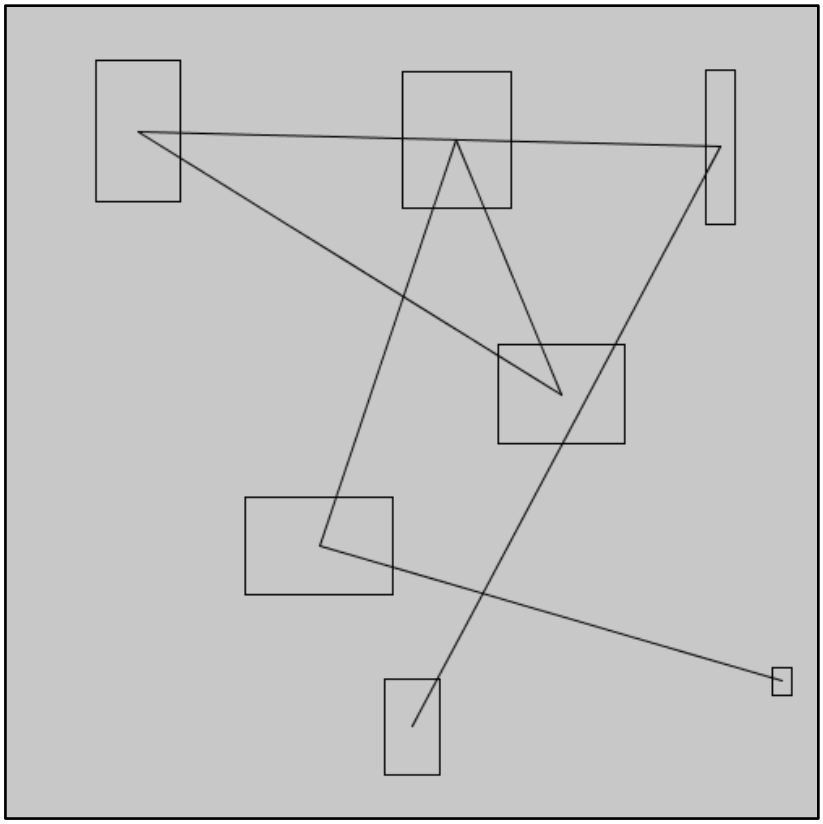


Figure 3.2 – An example of how the flocking algorithm can be applied to create random dungeons. First, random rooms are created and drawn to the screen, then separated out. This is the exact process Reynolds used.

[9] Buckland, Mat. *Programming Game AI by Example*. Plano, TX: Wordware Publ., 2009. 113-19.

Reynolds' flocking algorithm, just with modification to cohesion. Another key difference is the amount of iterations. Reynolds' boid flocking must update each frame to continue the movements of the boids indefinitely, but separation steering must only continue until all rooms have been separated.

Figure 3.3 – Procedurally generated dungeon room mapping. This map is the final point in the process. The rooms have been separated, some randomly selected based on a population percentage value of 15%, and lines were crudely drawn between them to signify tunnels.



The code written to generate Figures 3.2 and 3.3 is posted in the Appendix. It is Processing code, using Processing's built in rectangle-drawing function to represent rooms. The Room class contains the method that computes the steering force, the heart of Reynolds' algorithm. This separation steering algorithm has a runtime complexity of $O(n^2)$, where n is the number of rooms initially generated. Each room must find each of its neighbors, and in this demonstration, the neighbor radius encompasses all initially spawned rooms.

Geometrimon

Game Design Document

Description of Game:

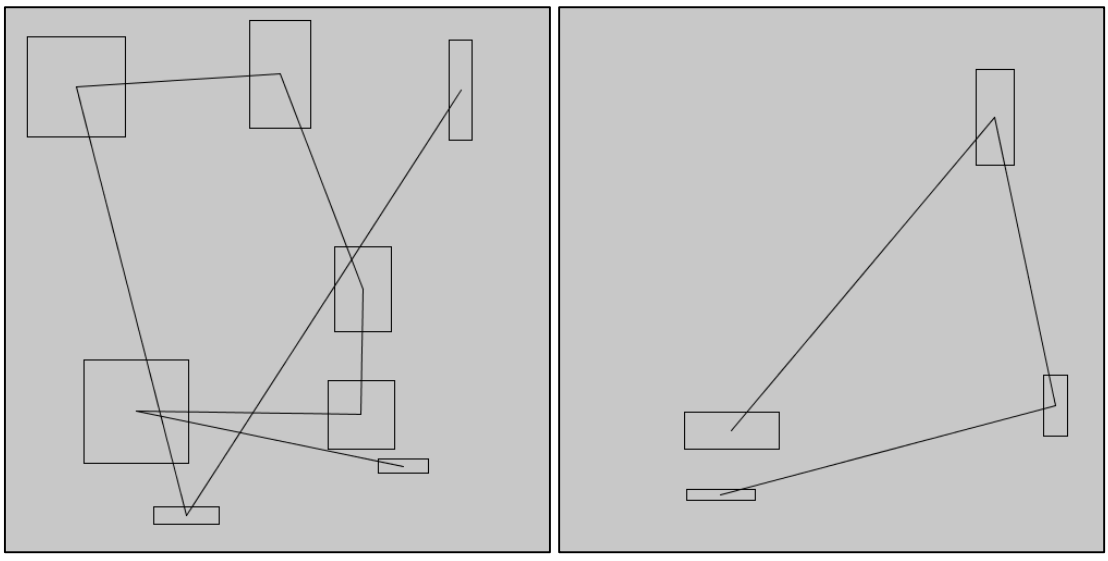
Geometrimon is a classic dungeon crawler. It is loosely modeled after an old game, Dragon Warrior Monsters, once a popular Gameboy game in Japan. The game is designed for mobile devices, and the game centers around satirical humor and sarcasm. The player is thrown into a world filled with mathematicians. Mathematicians have become too powerful and created monsters to do their bidding. These monsters now threaten existence due to a miscalculation. Also, mathematicians are not artistically inclined, and these monsters consist only of geometric shapes with more powerful monsters containing more vertices and dimensions. The monsters have grown and evolved to have their own god, the sphere, a shape that has surpassed vertices. The mathematicians have sequestered these monsters into their own dimension by bending space-time at points across the world with the help of physicists. The player (the *consultant*) can enter this alternate dimension to conquer the monster world. To do this, the player must gather a team of monsters by recruiting them along the way. Each of these points of bent space-time leads to a dungeon, procedurally generated via separation steering. The player must conquer all dungeons and defeat the sphere to win the game.

Key Features:

- Every mathematician is modeled after a real-life mathematician.
- The player is gifted their first monster, who does not obey initially.
- The monsters have attributes and gain experience.
- Not all attributes are directly linked to experience (example: tameness).
- Opposing monsters grow steadily more challenging as the player advances.

- Monsters can *evolve*, meaning they gain another vertex (a point becomes a line, etc.).
- Monsters can be bred, adding the parents' vertex count together minus some offset.
- Encounters throughout each dungeon are battles between the player and enemy monsters that can be recruited – these encounters happen once ever few steps.
- Each dungeon has a preset number of floors, increasing in number as the game progresses.
- Each dungeon keeps the same artwork, though the rooms are procedurally generated.
- Every ten floors in a dungeon will contain a transition in artwork and a more significant increase in difficulty.
- Player movement and dungeon mapping will be based on a gridded system in compliance with the artistic style.

Dungeon (Complex and Simple) Example Maps:



CELLULAR AUTOMATA

Cellular automata, or cellular automation, is another procedural generation algorithm with many uses. It can be both teleological or ontogenetic depending on its usage. The idea, originating from Stanislaw Ulam and John von Neumann, is to have the computer replicate a task autonomously. Ulam and von Neumann began working separately on two different ideas, but came together in the late 1950s. This new project considered a liquid system as a group of cells, measuring how the movement of each cell was affected by its neighbors. Thus, the first model of cellular automata was born.[10]

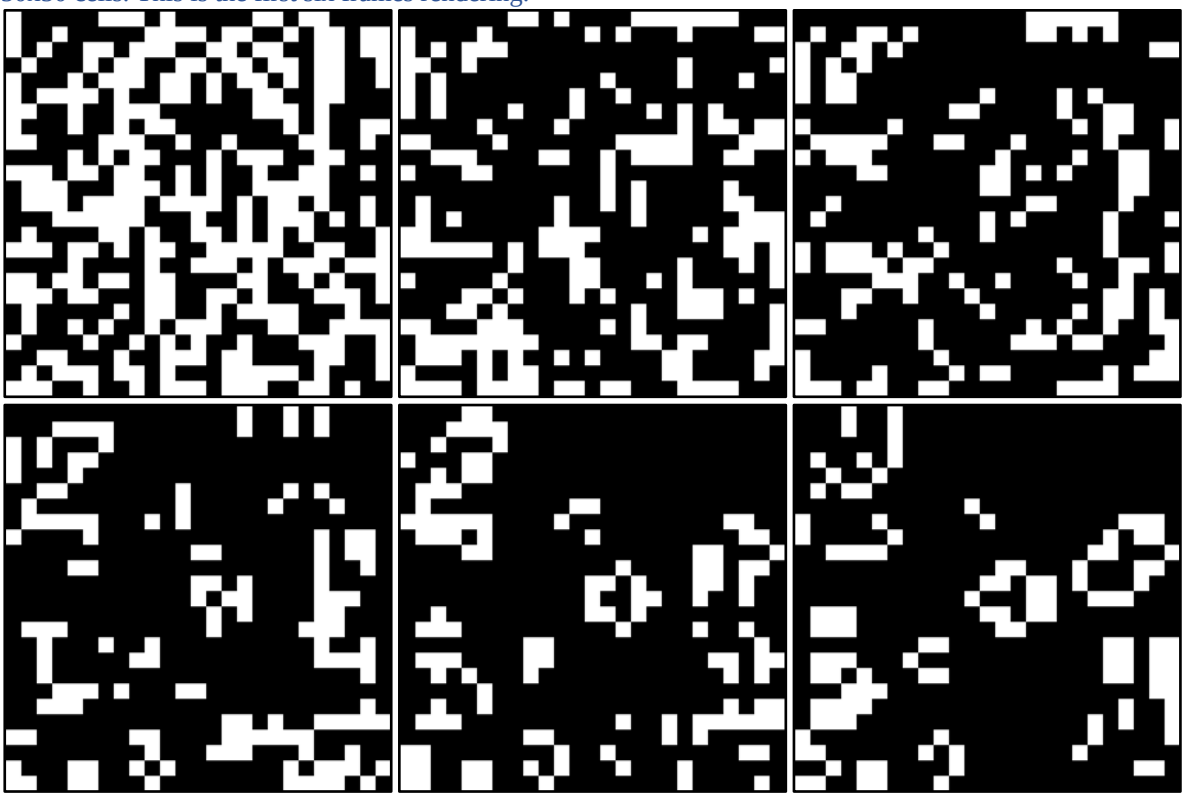
Cellular automata operate on a three-step process: divide a system into a grid of cells assigning each cell an initial state along the way, iterate through each cell once the grid has been initialized and examine its neighbors, and finally recalculate each cells' new state based on some system of rules. The most well-known example of this is *Conway's Game of Life*. The game arose from Conway's interest in a problem presented by von Neumann; he wanted to build a machine that could then build more of itself sustainably, indefinitely, and autonomously. His Game of Life is a game in which there are no players, the program plays itself through cellular automaton. The game begins by dividing the play space into a grid and randomly assigning each cell a state, dead or alive. The goal of the game is to simulate life, hence the title, with each cell representing a person or population. The initial randomization of the grid can be seeded or based on a random fill percentage defined by the developer. Figure 4.1 shows screen shots of Conway's Game of Life with an initial fill percentage of 50%. I found the original rules for the

[10] Bialynicki-Birula, Iwo, and Iwona Bialynicka-Birula. *Modeling reality: how computers mirror life*.

game, which are the same rules used in Figure 4.1, on Wikipedia. The code is supplied in the Appendix. The rules are as follows:

1. Any live cell with fewer than two live neighbors dies, simulating under population.
2. Any live cell with exactly two or three neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, simulating overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, simulating reproduction.

Figure 4.1 – An example of Conway's Game of Life. The initialization of the grid is based off a random fill percentage of 50%. Black indicates a dead cell while white indicates an alive cell. The grid is 50x50 cells. This is the first six frames rendering.

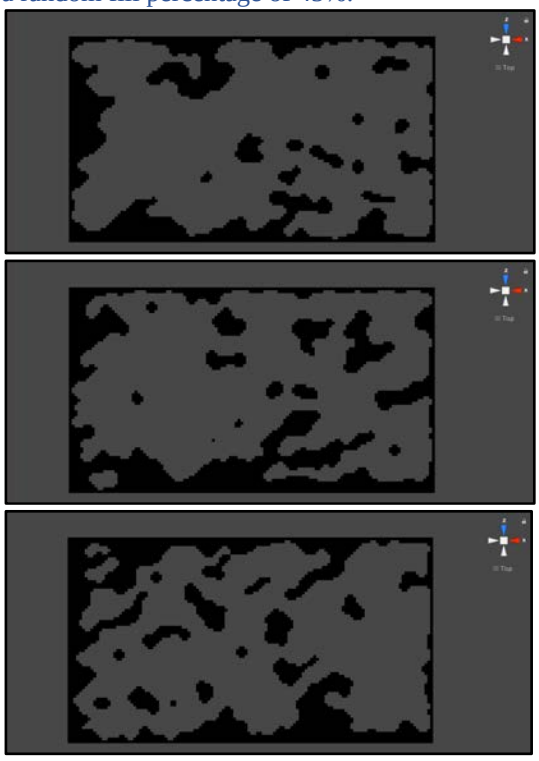


These rules are applied each iteration, or *game tick*, and after the initial running of the program, the game continues play by itself. These rules are applied to each cell's eight neighbors, a three-by-three grid surrounding the cell in question, unless the cell is a border cell.[11]

[11] "Cellular automaton." Wikipedia. April 06, 2017.

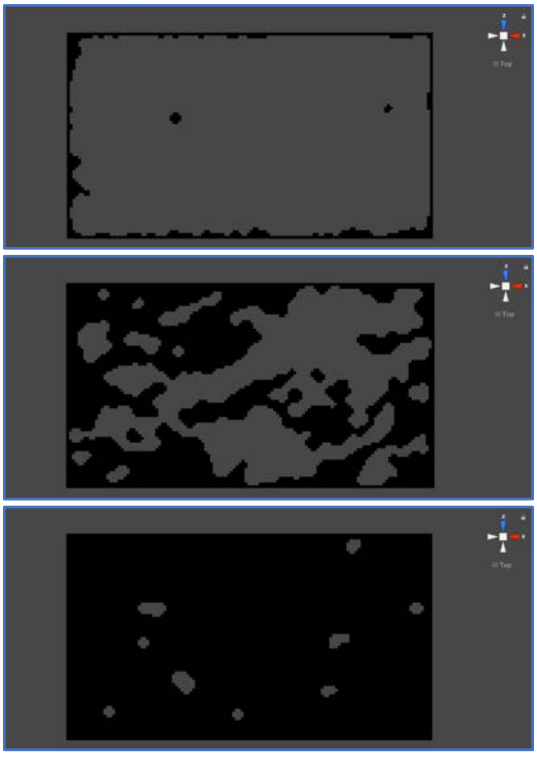
The original game, however, is designed to be played on an infinite grid without borders, but that is impossible in demonstration. Also, people have changed and created their own rulesets, some taking it upon themselves to try to create unique and interesting patterns with specific rule declarations.

Figure 4.2 – Procedural cave generation in Unity. Each of these caves were generated with a random fill percentage of 45%.



In games, other than no-player games like Conway's Game of Life, cellular automata can be used to do many things. One example that I came across was Sebastian Lague's procedurally generated caves for games like Terraria.[12] Figure 4.2 demonstrates the result of applying the same process used in Conway's Game of Life to generating textures and meshes for games. The caves in Figure 4.2 were also drawn based on the same random fill percent. Figure 4.3 shows three caves that were drawn with three different random fill percentages.

Figure 4.3 – Procedurally generated caves based on three different random fill percentages. In order from top to bottom, the random fill percentages were 30%, 50%, and 60%.



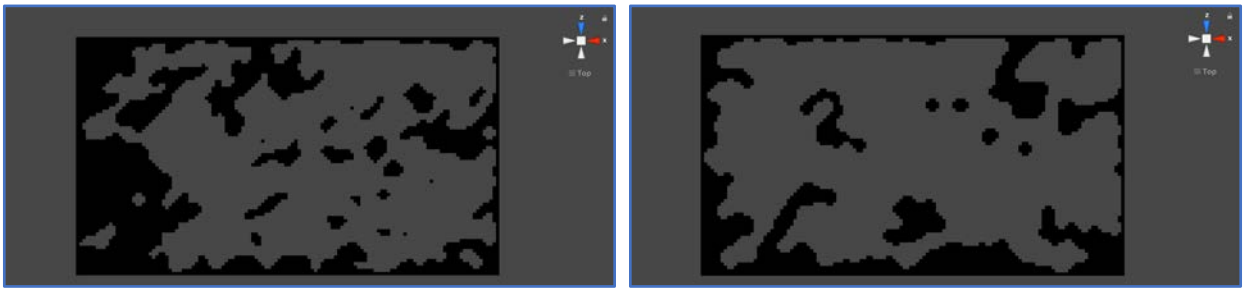
The key difference to note between Conway's Game of Life and this cave generation technique is the amount of smoothing iterations applied.

[12] Lague, Sebastian. "Procedural Cave Generation tutorial." Unity.

Conway's game continues indefinitely while the caves in Figures 4.2 and 4.3 only went through four iterations of smoothing. The amount of iterations can be varied to create unique results. Figure 4.4 demonstrates this phenomenon.

The cellular automata procedural generation technique has a runtime complexity of $O(n^2)$, where n is the side lengths of the grid. This assumes that the grid is a square; if it were not a square, then the runtime complexity would be $O(nk)$, where n and k are the length and width of the canvas, respectively. Even this runtime complexity is in simplified form because each cell's neighbors must also be considered each iteration. For the purposes of this study and Conway's game, the grids were all two-dimensional and each cell's immediate eight neighbors were examined. Other variations of the algorithm may change the runtime complexity depending on the number of neighbors examined, the calculations involved with the neighbors, or the dimensions of the grid used.

Figure 4.4 – An example showing differences between caves generated with differing amounts of smoothing iterations. The cave on the left went through only one iteration of smoothing while the cave on the right underwent 25 iterations. The differences are subtle, but the cave on the left has more jagged features and more noise. The cave on the right has more rounded edges and not as much noise scattered through its center regions.



Pixel.Defender

Game Design Document

Description of Game:

Pixel.Defender is a tower defense game with a twist. The game progresses like any normal tower defense game: the player spends his or her initial time setting up a base, arms the pathway to their base with towers, and enemies spawn in waves making their way toward the player's base. The twist is that each time the player begins a new play session, the map is procedurally generated with cellular automata. It is seeded, so the player can replay any map that they particularly enjoy, and the player has the option to use any seed or just randomize it each time before he or she begins.

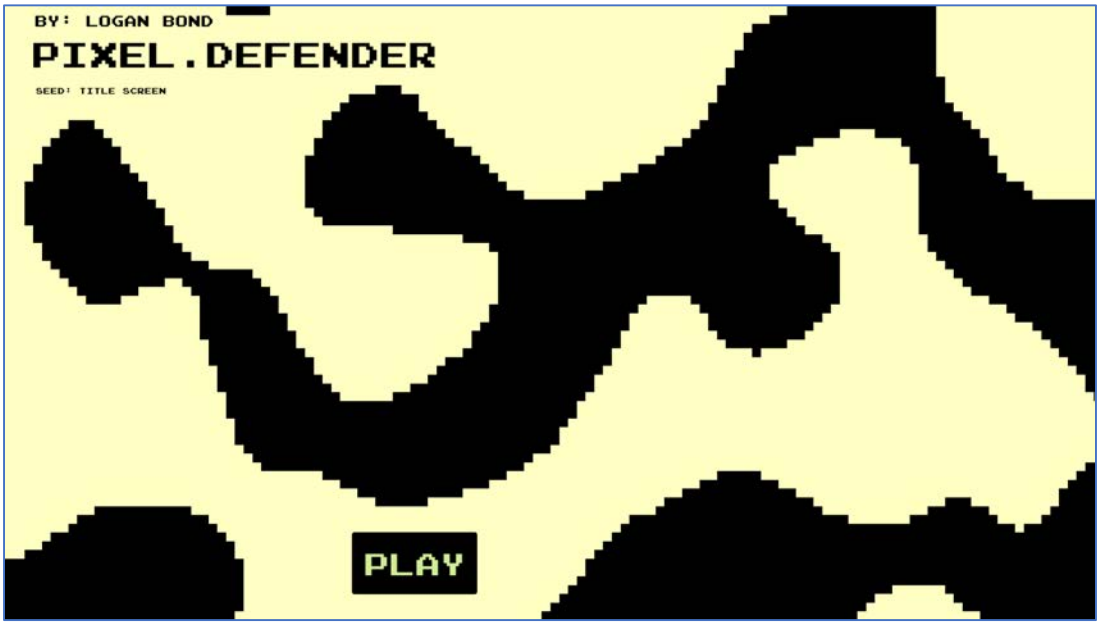
Due to the nature of procedural generation, it is hard to programmatically place the player's and enemy's bases. Therefore, the player chooses where each of the bases spawn on startup. The player may place either base on only valid tiles, which are denoted by color. The color of each tile is determined programmatically with cellular automata. The algorithm may produce sections of the map that are closed off from other portions; however, the player is still allowed to place there. Should the enemies run into walls that are blocking them from the players, they will simply begin tunneling through the wall. Each tile has a life point value associated with it, and should a tile's life points reach zero, the tile becomes passable and changes color.

Key Features:

- Player towers may only be placed on non-passable tiles.
- The player may only place towers on one color while the enemy may only pass over the other color.

- Each tower is a square, maintaining the pixelated artwork and title to the game.
- Towers that are bigger squares are harder to place, due to the procedural generation; however, these towers provide more defense.
- Enemies come in waves, allowing the player time to place more towers and purchase more defense during the downtime.
- Players may also purchase power-ups which may build up or tear down impassible areas of the map.
- The map is generated based on the Procedural Cave Generation code in the Appendix, promoting impassible areas of the map to take up the perimeter of the game.

Title Screen Procedurally Generated with Seed “Title Screen”:



CONCLUSION

I conducted this study to increase my ability to program complex algorithms, as well as understand them. Now, reaching the end of my analysis, I feel much more confident in my ability to examine and understand complicated code when it is presented to me. In this study, I analyzed and wrote four different procedural generation algorithms. These four algorithms were chosen based on interest, familiarity, and variety with emphasis placed on writing the actual code myself.

One interesting and key observation came out of this analysis. Each game contains some amount of crossover between algorithms, or each game could be further improved if written in conjunction with one of the other algorithms in the study. For example, good game design involves presenting content to the player little at a time. As the levels progress, so does the challenge and amount of content. It is bad practice to give the player all the content on first load in; that would overwhelm the player. HoverCube, according to the Game Design Document, randomly generates each chunk's gap that the player must traverse. It would be interesting to instead generate each chunk with Perlin noise at the beginning of the game. This would create smoother transitions between chunks and not create as much of a challenge for the player at the beginning of the game. As the player progresses, the noise scale value could be modified to increase the amount of stark changes in gap position. This, of course, would change the chunk-based algorithm slightly. The chunks are still randomly generated, but have more of a smoothness between them. The problem lies with randomly selecting which chunk to render; as the chunks fall off the screen, they would have to be put back in correct order in the pool.

Another interesting example of crossover is the similarity between Perlin noise and cellular automata. The Pixel.Defender screenshot of the title screen could have been rendered in either algorithm. They are very similar in the way they work: divide the screen into a grid, generate values for each cell based on a scale and value, and then run some calculations. The difference is that Perlin noise generates a range of numbers, generally from 0.0 to 1.0, while cellular automata gives each cell a state. However, cellular automata can give each cell any number of states, increasing the difficulty of the algorithm, but further mimicking Perlin noise. On the other hand, Perlin noise could be given conditions to further mimic cellular automata. For example, every cell given a value between 0.0 and 0.5 could be normalized to 0.0, and likewise with 1.0 at the other end of the spectrum. This method somewhat defeats the purpose of the Perlin noise algorithm, which is to create smoother transitions. However, the important thing to note is not the negation of the algorithm, just that it takes one simple condition to turn the algorithm into some mock-cellular automata algorithm. Each algorithm appears to be just a few mutations from one another, though the code is very different.

There are many different algorithms that embody procedural generation. Each algorithm is uniquely suited to solve a niche problem that developers everywhere have experienced over the years. After examining several of the more common algorithms and getting to personally write each one, I plan to create a working alpha-version of each projected game in this document at the very least. It is much easier to write a complicated algorithm to generate content procedurally rather than to hardcode every piece of content in a game by way of menially configuring each piece.

APPENDIX

This Appendix contains code, pseudo-code, or other related material that took up too much space in the middle of the document. Thus, posted sections that follow this statement are most likely large. They will be grouped together with their relevant constituents (for example, classes and methods associated with a single algorithm), and they will be separated from non-related objects via a page break, such as the one after this sentence.

Chunk-Based Procedural Generation Example

Chunk_Generator.pde

```
// driver class
Player player;
Chunk[] chunks;

final int NUM_CHUNKS = 100;
final int PLAYER_SIZE = 50;
final int BOUNDS_HEIGHT = (PLAYER_SIZE/2) - 1;

int gap, count;

void setup() {
  size(1000, 500);

  rectMode(CENTER);

  count = 0;
  gap = width/5; // gap to be between the rendered chunks

  player = new Player(width/5, height/2, PLAYER_SIZE);
  chunks = new Chunk[NUM_CHUNKS];

  initializeChunks(); // generate the pool
}

void draw() {
  background(200);
  noStroke();
  player.drawPlayer();

  // draw bounds
  fill(0);
  rect(width/2, BOUNDS_HEIGHT/2, width, BOUNDS_HEIGHT);
  rect(width/2, height-(BOUNDS_HEIGHT/2), width, BOUNDS_HEIGHT);

  // create the gap between obstacles
  if (count == 0) {
    chunks[selectRandomIndex()].toggleChunk();
  }

  // draw and move the chunks on the screen
  for (int i = 0; i < chunks.length; i++) {
    if (chunks[i].onScreen) {
```

```
      chunks[i].drawChunk();
      chunks[i].moveChunk();
    }
  }

  checkIfChunksOffScreen();
  count = (count+1) % gap; // creates the gap between chunk
renderings
}

// create the chunks
void initializeChunks() {
  for (int i = 0; i < chunks.length; i++) {
    chunks[i] = new Chunk(width, PLAYER_SIZE, BOUNDS_HEIGHT);
  }
}

// check for chunks fallen off of the screen
void checkIfChunksOffScreen() {
  for (int i = 0; i < chunks.length; i++) {
    if (chunks[i].x < 0) {
      chunks[i].toggleChunk(); // toggle that it is now off
screen
    }
  }
}

int selectRandomIndex() {
  int rand = (int)random(0, NUM_CHUNKS);
  while (chunks[rand].onScreen) {      // if the selected chunk
is already rendered
    rand = (int)random(0, NUM_CHUNKS); // continue selecting a
random chunk
  }
  return rand;
}

// moving the player based on input
void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP || keyCode == RIGHT) {
      player.moveUp();
    } else if (keyCode == DOWN || keyCode == LEFT) {
      player.moveDown();
    }
  }
}
```

Chunk.pde

```
// a chunk consists of two vertical bars, or obstacles
// each chunk is added to a pool in the driver class
class Chunk {

  int x;
  int padding, obsWidth, offset;
  int height1, height2;
  int boundsHeight;

  boolean onScreen;

  Chunk(int x, int playerSize, int boundsHeight) {
    onScreen = false; // each chunk starts off screen
    padding = playerSize+obsWidth;
    obsWidth = 20; // width of obstacle

    offset = obsWidth/2;
    this.x = x + offset;
    this.boundsHeight = boundsHeight;

    // calculating the random heights
    // height1 is based on a random number generator
    // height2 is based on height one to account for the player
gap
    this.height1 = (int)random(0, height-playerSize-padding-
boundsHeight*2);
    this.height2 = height -
(height1+playerSize+padding+boundsHeight*2);

    }

  // drawing the two red, vertical bars
  void drawChunk() {
    fill(255, 0, 0);
    rect(x, height1/2 + boundsHeight, obsWidth, height1);
    rect(x, height-(height2/2)-boundsHeight, obsWidth, height2);
  }

  void moveChunk() {
    x--;
  }

  void toggleChunk() {
   this.onScreen = !this.onScreen; // detect whether the chunk
                                   // is currently rendered
```

```
    x = width + offset; // reset the chunk to off the right side
                        // of the screen
  }
}
```

Player.pde

```
// the player
class Player {

  int x, y, size;

  private int w, h;

  Player(int x, int y, int size) {
    this.x = x;
    this.y = y;
    this.size = size;
    w = size;
    h = size;
  }

  void drawPlayer() {
    fill(255);
    rect(x, y, w, h);
  }

  void moveUp() {
    if (y - size > 0) {
      y -= size;
    }
  }

  void moveDown() {
    if (y + size < height) {
      y += size;
    }
  }
}
```

Perlin Noise Algorithm Pseudo Code[13]

```
// Function to linearly interpolate between a0 and a1
// Weight w should be in the range [0.0, 1.0]
function lerp(float a0, float a1, float w) {
     return (1.0 - w)*a0 + w*a1;
}

// Computes the dot product of the distance and gradient
vectors.
function dotGridGradient(int ix, int iy, float x, float y) {

     // Precomputed (or otherwise) gradient vectors at each grid
node
     extern float Gradient[IYMAX][IXMAX][2];

     // Compute the distance vector
     float dx = x - (float)ix;
     float dy = y - (float)iy;

     // Compute the dot-product
     return (dx*Gradient[iy][ix][0] + dy*Gradient[iy][ix][1]);
}

// Compute Perlin noise at coordinates x, y
function perlin(float x, float y) {

     // Determine grid cell coordinates
     int x0 = (x > 0.0 ? (int)x : (int)x - 1);
     int x1 = x0 + 1;
     int y0 = (y > 0.0 ? (int)y : (int)y - 1);
     int y1 = y0 + 1;

     // Determine interpolation weights
     // Could also use higher order polynomial/s-curve here
     float sx = x - (float)x0;
     float sy = y - (float)y0;

     // Interpolate between grid point gradients
     float n0, n1, ix0, ix1, value;
     n0 = dotGridGradient(x0, y0, x, y);
     n1 = dotGridGradient(x1, y0, x, y);
     ix0 = lerp(n0, n1, sx);
     n0 = dotGridGradient(x0, y1, x, y);
```

[13] Pseudo Code provided by Wikipedia, “Perlin Noise”

```
        n1 = dotGridGradient(x1, y1, x, y);
        ix1 = lerp(n0, n1, sx);
        value = lerp(ix0, ix1, sy);

        return value;
    }
```

Perlin Noise Mesh Generator in Unity[14]

Noise.cs

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

public static class Noise {

    public static float[,] GenerateNoiseMap(int mapWidth,
     int mapHeight, int seed,float scale, int octaves,
     float persistance, float lacunarity, Vector2 offset)
    {
        float[,] noiseMap = new float[mapWidth, mapHeight];

        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves];
        for(int i=0; i < octaves; i++)
        {
            float offsetX = prng.Next(100000, 100000)
               + offset.x;
            float offsetY = prng.Next(100000, 100000)
               + offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if (scale <= 0)
        {
            scale = 0.0001f;
        }

        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        float halfWidth = mapWidth / 2f;
        float halfHeight = mapHeight / 2f;

        for(int y = 0; y < mapHeight; y++)
        {
            for(int x = 0; x < mapWidth; x++)
            {
```

[14] Code inspired by Sebastian Lague in his video series “Procedural Landmass Generation”. It is undocumented due to the sheer mass of it.

```
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;
                for (int i = 0; i < octaves; i++)
                {
                    float sampleX = (x - halfWidth) / scale
                        * frequency + octaveOffsets[i].x;
                    float sampleY = (y - halfHeight) / scale
                        * frequency + octaveOffsets[i].y;

                    float perlinValue = Mathf.PerlinNoise(
                        sampleX, sampleY) * 2 - 1;

                    noiseHeight += perlinValue * amplitude;

                    amplitude *= persistance;
                    frequency *= lacunarity;
                }

                if(noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                }
                else if(noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                noiseMap[x, y] = noiseHeight;
            }
        }

        for (int y = 0; y < mapHeight; y ++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                noiseMap[x, y] = Mathf.InverseLerp(
                    minNoiseHeight, maxNoiseHeight,
                    noiseMap[x, y]);
            }
        }

        return noiseMap;
    }
}
```

MapGenerator.cs

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

public class MapGenerator : MonoBehaviour {

    public enum DrawMode { NoiseMap, ColorMap, Mesh }
    public DrawMode drawMode;

    public int mapWidth;
    public int mapHeight;
    public float noiseScale;
    public bool autoUpdate;

    public float meshHeightMultipler;

    public int octaves;
    [Range(0,1)]
    public float persistance;
    public float lacunarity;

    public string seed;
    public Vector2 offset;

    public TerrainType[] regions;

    public void Start()
    {
        GenerateMap(seed);
    }

    public void GenerateMap(string seed)
    {
        float[,] noiseMap = Noise.GenerateNoiseMap(mapWidth,
          mapHeight, seed.GetHashCode(), noiseScale, octaves,
          persistance, lacunarity, offset);

        Color[] colorMap = new Color[mapWidth * mapHeight];
        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                float currentHeight = noiseMap[x, y];
```

```
                for(int i = 0; i < regions.Length; i++)
                {
                    if(currentHeight <= regions[i].height)
                    {
                        colorMap[y * mapWidth + x] =
                              regions[i].color;
                        break;
                    }
                }
            }
        }

        MapDisplay display = FindObjectOfType<MapDisplay>();
        if (drawMode == DrawMode.NoiseMap)
        {
            display.DrawTexture(
               TextureGenerator.TextureFromHeightMap(noiseMap));
        }
        else if(drawMode == DrawMode.ColorMap)
        {
            display.DrawTexture(
               TextureGenerator.TextureFromColorMap(colorMap,
               mapWidth, mapHeight));
        }
        else if(drawMode == DrawMode.Mesh)
        {
            display.DrawMesh(
               MeshGenerator.GenerateTerrainMesh(noiseMap,
               meshHeightMultipler),
               TextureGenerator.TextureFromColorMap(colorMap,
               mapWidth, mapHeight));
        }
    }

    void OnValidate()
    {
        if(mapWidth < 1)
        {
            mapWidth = 1;
        }
        if(mapHeight < 1)
        {
            mapHeight = 1;
        }
        if(lacunarity < 1)
        {
            lacunarity = 1;
```

```
        }
        if(octaves < 0)
        {
            octaves = 0;
        }
    }
}

[System.Serializable]
public struct TerrainType
{
    public string name;
    public float height;
    public Color color;
}
```

MapDisplay.cs

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

public class MapDisplay : MonoBehaviour {

    public Renderer textureRender;
    public MeshFilter meshFilter;
    public MeshRenderer meshRenderer;

    public void DrawTexture(Texture2D texture) {


        textureRender.sharedMaterial.mainTexture = texture;
        textureRender.transform.localScale = new Vector3(
          texture.width, 0,texture.height);
    }

    public void DrawMesh(MeshData meshData, Texture2D texture)
    {
        meshFilter.sharedMesh = meshData.CreateMesh();
        meshRenderer.sharedMaterial.mainTexture = texture;
    }
}
```

MeshGenerator.cs

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

public static class MeshGenerator
{

    public static MeshData GenerateTerrainMesh(
          float[,] heightMap, float heightMultiplier)
    {
        int width = heightMap.GetLength(0);
        int height = heightMap.GetLength(1);
        float topLeftX = (width - 1) / -2f;
        float topLeftZ = (height - 1) / 2f;

        MeshData meshData = new MeshData(width, height);
        int vertexIndex = 0;

        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                meshData.vertices[vertexIndex] =
                    new Vector3(topLeftX + x,
                    heightMap[x, y] * heightMultiplier,
                    topLeftZ - y);

                meshData.uvs[vertexIndex] =
                    new Vector2(x / (float)width,
                    y / (float)height);

                if (x < width - 1 && y < height - 1)
                {
                    meshData.AddTriangle(vertexIndex,
                         vertexIndex + width + 1,
                    vertexIndex + width);
                    meshData.AddTriangle(
                         vertexIndex + width + 1,
                         vertexIndex, vertexIndex + 1);
                }
                vertexIndex++;
            }
        }

        return meshData;
```

```
    }
}

public class MeshData
{
    public Vector3[] vertices;
    public int[] triangles;
    public Vector2[] uvs;

    int triangleIndex;

    public MeshData(int meshWidth, int meshHeight)
    {
        vertices = new Vector3[meshWidth * meshHeight];
        uvs = new Vector2[meshWidth * meshHeight];
        triangles = new int[(meshWidth - 1)
               * (meshHeight - 1) * 6];
    }

    public void AddTriangle(int a, int b, int c)
    {
        triangles[triangleIndex] = a;
        triangles[triangleIndex + 1] = b;
        triangles[triangleIndex + 2] = c;
        triangleIndex += 3;
    }

    public Mesh CreateMesh()
    {
        Mesh mesh = new Mesh();
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.uv = uvs;
        mesh.RecalculateNormals();
        return mesh;
    }
}
```

TextureGenerator.cs

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

public static class TextureGenerator {
```

```
    public static Texture2D TextureFromColorMap(
          Color[] colorMap, int width, int height)
    {
        Texture2D texture = new Texture2D(width, height);
        texture.filterMode = FilterMode.Point;
        texture.wrapMode = TextureWrapMode.Clamp;
        texture.SetPixels(colorMap);
        texture.Apply();
        return texture;
    }

    public static Texture2D TextureFromHeightMap(
          float[,] heightMap)
    {
        int width = heightMap.GetLength(0);
        int height = heightMap.GetLength(1);

        Texture2D texture = new Texture2D(width, height);

        Color[] colorMap = new Color[width * height];
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                colorMap[y * width + x] =
                    Color.Lerp(Color.black, Color.white,
                    heightMap[x, y]);
            }
        }
        return TextureFromColorMap(colorMap, width, height);
    }
}
```

Boids, An Example of the Flocking Algorithm[15]

Boid.pde

```
// "bird-oid"
class Boid {
  PVector position;
  PVector velocity;
  PVector acceleration;

  float r;
  float maxforce; // maximum steering force
  float maxspeed; // maximum speed

  Boid(float x, float y) {
    acceleration = new PVector(0, 0); // no acceleration

    float angle = random(TWO_PI);
    velocity = new PVector(cos(angle), sin(angle));

    position = new PVector(x, y);
    r = 2.0;
    maxspeed = 2;
    maxforce = 0.03;
  }

  void run(ArrayList<Boid> boids) {
    flock(boids);
    update();
    borders();
    render();
  }

  void applyForce(PVector force) {
    acceleration.add(force);
  }

  // new acceleration is calculated
  // based on separation, alignment, cohesion
  void flock(ArrayList<Boid> boids) {
    PVector sep = separate(boids);
    PVector ali = align(boids);
    PVector coh = cohesion(boids);
```

[15] Processing Foundation. "Flocking \ Examples \ Processing.org."

```
  // weight the forces arbitrarily
  sep.mult(1.5);
  ali.mult(1.0);
  coh.mult(1.0);

  // add these forces to new acceleration
  applyForce(sep);
  applyForce(ali);
  applyForce(coh);
}

void update() {
  // update the velocity
  velocity.add(acceleration);
  // limit speed
  velocity.limit(maxspeed);
  position.add(velocity);
  // reset the acceleration back to 0
  acceleration.mult(0);
}

// calculates and applies a steering force in a direction
PVector seek(PVector target) {
  PVector desired = PVector.sub(target, position);

  // scale to max speed
  desired.normalize();
  desired.mult(maxspeed);

  // steering = desired - velocity
  PVector steer = PVector.sub(desired, velocity);
  steer.limit(maxforce);
  return steer;
}

// drawing the boids
void render() {
  // draw a triangle rotated in direction of velocity
  float theta = velocity.heading2D() + radians(90);

  fill(200, 100);
  stroke(255);
  pushMatrix();
  translate(position.x, position.y);
  rotate(theta);
  beginShape(TRIANGLES);
  vertex(0, -r*2);
```

```
    vertex(-r, r*2);
    vertex(r, r*2);
    endShape();
    popMatrix();
  }

  // making the boids wrap around the screen
  void borders() {
    if (position.x < -r) position.x = width+r;
    if (position.y < -r) position.y = height+r;
    if (position.x > width+r) position.x = -r;
    if (position.y > height+r) position.y = -r;
  }

  // finds neighbors and steers away from them
  PVector separate(ArrayList<Boid> boids) {
    float desiredSeparation = 25.0;
    PVector steer = new PVector(0, 0, 0);
    int count = 0;

    // loop through all the other boids to find neighbors
    for (Boid other : boids) {
      float d = PVector.dist(position, other.position);
      if (d > 0 && d < desiredSeparation) {
        // calculate vector pointing away from neighbor
        PVector diff = PVector.sub(position, other.position);
        diff.normalize();
        diff.div(d);
        steer.add(diff);
        count++; // count thy neighbors
      }
    }
    // average it out
    if (count > 0) {
      steer.div((float)count);
    }

    // if that doesn&apos;t zero-out the vector
    if (steer.mag() > 0) {
      // Reynolds&apos; algorithm
      steer.normalize();
      steer.mult(maxspeed);
      steer.sub(velocity);
      steer.limit(maxforce);
    }
    return steer; // return the steering force
  }
```

```
// calculate the average velocity of all neighbor boids
PVector align (ArrayList<Boid> boids) {
  float neighborDist = 50;
  PVector sum = new PVector(0, 0);
  int count = 0;
  for (Boid other : boids) {
    float d = PVector.dist(position, other.position);
    if (d > 0 && d < neighborDist) {
      sum.add(other.velocity);
      count++;
    }
  }

  // computing average
  if (count > 0) {
    sum.div((float)count);

    // Reynolds&amp;apos; algorithm
    sum.normalize();
    sum.mult(maxspeed);
    PVector steer = PVector.sub(sum, velocity);
    steer.limit(maxforce);
    return steer;
  } else {
    return new PVector(0, 0);
  }
}

// calculate "center of mass" of all neighbors and steer there
PVector cohesion(ArrayList<Boid> boids) {
  float neighborDist = 50;
  PVector sum = new PVector(0, 0); // start with empty vector
  int count = 0;
  for (Boid other : boids) {
    float d = PVector.dist(position, other.position);
    if (d > 0 && d < neighborDist) {
      sum.add(other.position); // add the position
      count++;
    }
  }
  if (count > 0) {
    sum.div(count);
    return seek(sum); // steer toward the position
  } else {
    return new PVector(0, 0);
  }
```

```
  }
}
```

Flock.pde

```
class Flock {

  ArrayList<Boid> boids; // list of the boids

  Flock() {
    boids = new ArrayList<Boid>();
  }

  void run() {
    for (Boid b : boids) {
      b.run(boids);
    }
  }

  void addBoid(Boid b) {
    boids.add(b);
  }
}
```

Flocking.java

```
Flock flock;
void setup() {
  size(640, 360);
  flock = new Flock();
  // add an initial set up boids
  for (int i = 0; i < 150; i++) {
    flock.addBoid(new Boid(width/2, height/2));
  }
}

void draw() {
  background(50);
  flock.run();
}

// to add another boid
void mousePressed() {
  flock.addBoid(new Boid(mouseX, mouseY));
}
```

Separation Steering: Random Dungeon Generation[16]

Room.pde

```
class Room {
  float x, y, w, h;
  float xPressure, yPressure;

  Room(float x, float y, float w, float h) {
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
  }

  void drawRoom() {
    rect(x, y, w, h); // Processing's API
  }

  // checking whether a room overlaps another
  boolean overlaps(Room other) {
    return x < (other.x+other.w) && (x+w) > other.x &&
      y < (other.y+other.h) && (y+h) > other.y;
  }

  // calculating the separation pressure
  void separationPressure(ArrayList<Room> rooms) {
    int count = 0;
    for (Room r : rooms) {
      // calculate distance from this room to neighbors
      float distance = dist(x, y, r.x, r.y);

      // check that this room does not overlap with others
      // calculate pressure
      if (this != r && overlaps(r)) {
        float xDiffNorm = (r.x - x)/distance;
        float yDiffNorm = (r.y - y)/distance;

        xPressure += xDiffNorm;
        yPressure += yDiffNorm;

        count++;
      }
    }
```

[16] Code inspired by Dr. Jeff Matocha

```
    // normalize the pressure across all neighbors
    if (count > 0) {
      xPressure /= count;
      yPressure /= count;

      float vectorHypotenuse =
            sqrt(pow(xPressure, 2) + pow(yPressure, 2));

      xPressure /= vectorHypotenuse;
      yPressure /= vectorHypotenuse;

      x -= xPressure;
      y -= yPressure;
    }
  }
}
```

Dungeon_Generator.pde

```
ArrayList<Room> rooms;
ArrayList<Room> drawnRooms;
boolean pause;
int count;

final int NUM_ROOMS = 100; // number of initial rooms

void setup() {
  size(500, 500);
  noFill();

  rooms = new ArrayList<Room>(); // a list of all of the rooms
generated
  drawnRooms = new ArrayList<Room>(); // a list of only drawn
rooms in the finished product
  initializeRooms();

  pause = false;
  count = 0;

  selectAndDraw(); // pruning the rooms
  drawLines(); // drawing crude lines between rooms

}
```

```
// the draw() method is commented out because it is unnecessary
for room generation
// this method demonstrates the separation/steering parts of the
// algorithm,
// but not the finished product, which contains pruning of rooms
// and lines
//void draw() {
//  if (!pause) {
//    background(200);
//    for (Room r : rooms) {
//      r.drawRoom();
//    }
//    separate();
//  }
//}

void selectAndDraw() {
  int i = 0;
  while (i < 1000) {
    separate(); // separate all of the rooms
    i++;
  }

  background(200);
  int percent = 15;
  for (int j = 0; j < NUM_ROOMS; j++) {
    // if room isn&apos;t out of bounds, generate random number
    // rooms are only drawn if random number is less than
population percent
    if (!outOfBounds(rooms.get(j)) && (int)random(0, 100) <
percent) {
      rooms.get(j).drawRoom();
      drawnRooms.add(rooms.get(j)); // the list of drawn rooms
    }
  }
}

void drawLines() {
  for (int j = 0; j < drawnRooms.size(); j++) {
    try {
      Room r1 = drawnRooms.get(j);
      Room r2 = drawnRooms.get(j+1);
      line(r1.x+(r1.w/2), r1.y+(r1.h/2), r2.x+(r2.w/2),
r2.y+(r2.h/2)); //drawing crude lines
    }
    catch (Exception e) {
      // do nothing
```

```
    }
  }
}

boolean outOfBounds(Room r) {
  // checking bounds, want to remove all rooms off-screen
  return (r.x+r.w) > width-10 || (r.y+r.h) > height-10
              || r.x < 10 || r.y < 10;
}

void separate() {
  for (Room r : rooms) {
    r.separationPressure(rooms); // calculating the force
  }
}

void initializeRooms() {
  int offset = 100;
  for (int i = 0; i < NUM_ROOMS; i++) {
    // initializing the rooms to be random widths/heights
    rooms.add(new Room(random((width/2)-offset,
(width/2)+(offset/2)),
      random((height/2)-offset, (height/2)+(offset/2)),
      random(10, 100), random(10, 100)));
  }
}

void keyPressed() {
  if (key == &apos;p&apos; || key == &apos;P&apos;) {
    pause = !pause;
    save("Dungeon_Creater_"+count+".png");
    count++;
  }
}
```

Conway's Game of Life Example with Original Rules

Conway_Game_of_Life.pde

```
final int RANDOM_FILL_PERCENT = 50;
final int SQUARE_SIZE = 1;

// use 2 grids:
// calculate 1 grid, dump it into the 2nd
// then switch out the grids
int[][] grid1;
int[][] grid2;

int count = 0;

void setup() {
  size(500, 500);

  grid1 = new int[width/SQUARE_SIZE][height/SQUARE_SIZE];
  grid2 = new int[width/SQUARE_SIZE][height/SQUARE_SIZE];

  initializeGrid();
  frameRate(10); // slow down the process
  noStroke();
}

void draw() {
  background(0);
  if (count % 2 == 0) {
    simulate(grid1);
  } else {
    simulate(grid2);
  }

  drawGrid(frameCount % 2 == 0 ? grid2 : grid1);
  count++;
}

// begin by randomly filling the grid
// each state is randomly dead or alive
// this is based on some RANDOM_FILL_PERCENT
// changing this value gives varying results
void initializeGrid() {
  for (int i = 0; i < grid1.length; i++) {
    for (int j = 0; j < grid1[0].length; j++) {
```

```
      grid1[i][j] = (int)random(0, 100) < RANDOM_FILL_PERCENT ?
0 : 1;
      grid2[i][j] = 0;
    }
  }
}

// draw the grid to the screen
void drawGrid(int[][] grid) {
  for (int i = 0; i < grid.length; i++) {
    for (int j = 0; j < grid[0].length; j++) {
      if (grid[i][j] == 0) {
        fill(0); // black = dead
      } else if (grid[i][j] == 1) {
        fill(255); // white = alive
      }
      rect(i*SQUARE_SIZE, j*SQUARE_SIZE, SQUARE_SIZE,
SQUARE_SIZE);
    }
  }
}

// simulate life happening autonomously
void simulate(int[][] grid) {
  // calculate the entire grid and store it in 2nd grid
  // switch out the two grids every frame
  int[][] otherGrid = grid == grid1 ? grid2 : grid1;
  for (int i = 0; i < grid.length; i++) {
    for (int j = 0; j < grid[0].length; j++) {
      // first calculate the value of all neighbors
      int neighborVal = calculateNeighborhood(i, j, grid);
      // apply rules
      if (grid[i][j] == 1 && neighborVal < 2) {
        // this cell dies due to underpopulation
        otherGrid[i][j] = 0;
      } else if (grid[i][j] == 1 &&
           neighborVal == 2 || neighborVal == 3) {
        // this cell continues to next generation
        otherGrid[i][j] = 1;
      } else if (grid[i][j] == 1 && neighborVal > 3) {
        // this cell dies due to overpopulation
        otherGrid[i][j] = 0;
      } else if (grid[i][j] == 0 && neighborVal == 3) {
        // this cell becomes alive due to reproduction
        otherGrid[i][j] = 1;
      } else {
        otherGrid[i][j] = 0; // clear out the cell otherwise
```

```
      }
    }
  }
}

// calculate the value of all neighbors
// parameters are x and y position of cell to look at
int calculateNeighborhood(int x, int y, int[][] grid) {
  int neighborVal = 0;
  for (int neighborX = x-1; neighborX <= x+1; neighborX++) {
    for (int neighborY = y-1; neighborY <= y+1; neighborY++) {
      if (inBounds(neighborX, neighborY)) { // make sure it is a
legal cell
        if (neighborX != x || neighborY != y) { // don&apos;t
count cell in question
          neighborVal += grid[neighborX][neighborY];
        }
      }
    }
  }
  return neighborVal;
}

// check that a cell is valid and in the grid
boolean inBounds(int x, int y) {
  return x >= 0 && x < width/SQUARE_SIZE && y >= 0 && y <
height/SQUARE_SIZE;
}
```

Procedural Cave Generation in Unity: Cellular Automata[17]

MapGenerator.cs

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

public class MapGenerator : MonoBehaviour {

    public int width;
    public int height;

    public string seed;
    public bool useRandomSeed;

    [Range(0,100)]
    public int randomFillPercent;

    int[,] map;

    // initialization
    void Start() {
        GenerateMap ();
    }

    // generate new maps on mouse clicks
    void Update() {
        if (Input.GetKeyDown(KeyCode.Space)) {
            GenerateMap ();
        }
    }

    // create the grid of cells
    void GenerateMap() {
        map = new int[width, height];
        RandomFillMap ();

        for (int i = 0; i < 5; i++) {
            SmoothMap ();
        }
    }
```

[17] Code inspired by Sebastian Lague in his video series “Procedural Cave Generation” on the Unity website.

```
// smoothing the map to have rounded edges
void SmoothMap() {
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            int neighborWallTiles =
                GetSurroundingWallCount (x, y);

            if (neighborWallTiles > 4) {
                map [x, y] = 1;
            } else if (neighborWallTiles < 4) {
                map [x, y] = 0;
            }
        }
    }
}

// want to know how many neighbors are walls
int GetSurroundingWallCount(int gridX, int gridY) {
    int wallCount = 0;
    for (int neighborX = gridX - 1;
           neighborX <= gridX + 1; neighborX++) {
        for (int neighborY = gridY - 1;
           neighborY <= gridY + 1; neighborY++) {
            //bounds checking
            if (neighborX >= 0 && neighborX < width
                && neighborY >= 0 && neighborY < height) {
                // check against looking at self
                if (neighborX != gridX ||
                     neighborY != gridY) {
                    // grid values are 0 or 1
                    // will only increment if 1,
                    // which is a wall
                    wallCount += map [neighborX, neighborY];
                }
            } else {
                // encourage growth of walls around edge
                wallCount++;
            }
        }
    }
    return wallCount;
}

// randomly fill the map
void RandomFillMap() {
    if (useRandomSeed) {
        seed = Time.time.ToString ();
```

```
            }

            // pseudo-random number generator
            System.Random pseudoRand =
              new System.Random (seed.GetHashCode ());

            for (int x = 0; x < width; x++) {
                for (int y = 0; y < height; y++) {
                    if (x == 0 || x == width - 1
                        || y == 0 || y == height - 1) {
                        map [x, y] = 1;
                    } else {
                        // assign 0 or 1 based on randomFillPercent
                        map [x, y] = (pseudoRand.Next (0, 100)
                             < randomFillPercent ? 1 : 0);
                    }
                }
            }
        }

        // render to Unity window
        void OnDrawGizmos() {
            if (map != null) {
                for (int x = 0; x < width; x++) {
                    for (int y = 0; y < height; y++) {
                        Gizmos.color = (map [x, y] == 1) ?
                             Color.black : Color.clear;
                        Vector3 pos = new Vector3
                             (-width / 2 + x + 0.5f, 0,
                             -height / 2 + y + 0.5f);
                        Gizmos.DrawCube (pos, Vector3.one);
                    }
                }
            }
        }
    }
```

REFERENCES

1. Wilson, Mark. "How 4 Designers Built A Game With 18.4 Quintillion Unique Planets." Co.Design. July 20, 2015. Accessed April 16, 2017. https://www.fastcodesign.com/3048667/how-4-designers-built-a-game-with-184-quintillion-unique-planets.

2. "Procedural Content Generation Wiki." Teleological vs. Ontogenetic - Procedural Content Generation Wiki. Accessed April 16, 2017. http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic.

3. Perlin, Ken. "An image synthesizer." *ACM SIGGRAPH Computer Graphics* 19, no. 3 (1985): 287-96. Accessed April 16, 2017. doi:10.1145/325165.325247.

4. Perlin, Ken. Original Perlin Noise Source Code. Raw data. http://mrl.nyu.edu/~perlin/doc/oscar.html.

5. Processing Foundation. "Noise() \ Language (API) \ Processing 2." Accessed April 16, 2017. https://processing.org/reference/noise_.html.

6. "Perlin noise." Wikipedia. March 22, 2017. Accessed April 16, 2017. https://en.wikipedia.org/wiki/Perlin_noise.

7. Perlin, Ken. Ken's Academy Award. Accessed April 25, 2017. http://mrl.nyu.edu/~perlin/doc/oscar.html.

8. Reynolds, Craig W. "Flocks, herds and schools: A distributed behavioral model." *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*, 1987. doi:10.1145/37401.37406.

9. Buckland, Mat. *Programming Game AI by Example*. Plano, TX: Wordware Publ., 2009. 113-19.

10. Bialynicki-Birula, Iwo, and Iwona Bialynicka-Birula. *Modeling reality: how computers mirror life*. Oxford: Oxford University Press, 2004.

11. "Cellular automaton." Wikipedia. April 06, 2017. Accessed April 16, 2017. https://en.wikipedia.org/wiki/Cellular_automaton.

12. Lague, Sebastian. "Procedural Cave Generation tutorial." Unity. Accessed April 16, 2017. https://unity3d.com/learn/tutorials/projects/procedural-cave-generation-tutorial.

13. Processing Foundation. "Flocking \ Examples \ Processing.org." Accessed April 16, 2017. https://processing.org/examples/flocking.html.