Splitting the Work

Distributed Computing for procedural generation of a realistic geographic simulation

An honors thesis presented to the Department of Mathematics & Computer Science

at Wittenberg University

By

Jacob R. Major

Page Intentionally Left Blank

An Honors Thesis Presented to The Department of Mathematics &

Computer Science

Splitting the Work

Distributed Computing for Procedural Generation of a Realistic Geographic Simulation

Jacob R. Major

Wittenberg University

Class of 2023

24 April 2023

Contents

Abstract

Procedural content generation can require computational abilities beyond most consumer scale devices. An alternative approach to traditional generation techniques is to break a large problem into smaller more manageable problems in exchange for a longer processing time. One step farther is to break the problem down and distribute it to multiple worker devices or virtual cores. This field is called distributed or decentralized computing and is growing rapidly with the advent of decentralized applications within the Web3 movement. The content of this paper discusses the process of optimizing a traditionally complex problem into smaller subsets of the problem and proposes possible distributed optimizations that could be implemented.

Introduction

The technology industry has presented new and innovative ways to solve computationally difficult problems in cost effective and efficient manners for decades. The video game industry specifically has introduced procedural terrain generation in games like Minecraft, Elite: Dangerous, and Terraria, allowing for vast and dynamic worlds to be created quickly and efficiently. Current approaches, however, are often limited in their scalability and modularity, making these systems difficult to adapt to different game engines, stories, and distribution across multiple machines.

Procedural generation is creating a set of rules that when executed in order will result in an object that is described by the rules. In the case of this project, we will be creating the rules for creating a voxel world, cube-based, like the worlds seen in the Game Minecraft. The ultimate

goal of procedural generation is to allow computers to create digital objects that fit some sort of criteria without any aid from a human.

Procedural generation has existed longer than modern video games in the form of card, board, and roleplay games such as Dungeons and Dragons. Many of these games proposed a rule for how the game would play out and required human players to compute the results of certain actions. With the introduction of computers, games such as Elite (Acornsoft, 1984) were able to create much larger games including representations of outer space. One of the most influential games in modern times is Minecraft (Mojang, 2011) which was able to create entire worlds on most consumer systems.

The goal of this paper is to explore the same challenges that these games solved and understand what steps in the design process need to be taken to make such complex procedural generation possible. The further research for this topic would include implementing these solutions on consumer systems as well as how to convert such processes to work with distributed systems since hardware is becoming harder and more expensive to source. The chip shortages in recent years have reduced the supply of consumer computing devices despite the ever-growing demand (Moore). Therefore, I believe it will be important to propose guidelines for computationally cost-effective programs going forward.

Over the course of this project, I have created a program that attempts to mimic the core fundamentals of the original Minecraft source code and proposed suggestions to save computational costs down the production line and eventually convert a typical project into a distributed system. Most of the project is written in Java while supporting scripts are written in Python and future suggestions utilize solutions such as Apache Zookeeper.

While terrain generation is mostly utilized in game development currently, I believe that future uses for procedural content generation will expand into other industries as Artificial Intelligence(AI) and Web3 software solutions attempt to solve more complex tasks. While my current project attempts to solve terrain generation in a game environment, future procedurally generated content could be user data for enterprise level software, children's educational content, and augmented reality content.

The structure of my paper is written as a manual for the inner workings of my coding project with explanations of design choices, code structure, and visualizations of the output of the program. Afterwards, I review literature discussing procedurally generated content and give a conclusion. Finally, I have attached a bibliography, an appendix of supporting images, as well as an appendix of code snippets from the project.

Literature Review

While the game of Minecraft explores procedurally generating a world within the constraints designed and maintained by a human, in a paper from the AAAI conference on Artificial intelligence, Matthew Fontaine et al. discuss the implementation of generative adversarial networks (GANs) to explore procedurally generating valid levels for the game Super Mario Bros. The discussion centers around the idea that while human level designers intuitively know how to create a good level, AI should also be able to create playable and entertaining levels indistinguishable from human created levels. This problem is called latent space illumination (LSI) in the paper.

While this approach to evaluating procedurally generated content is fascinating, it presents a high computation cost in order to compare different approaches for solving LSI, with

over 300 trials lasting approximately 7 hours each. Although these trials were run in parallel on a university cluster of Intel Xeon L5520 processors, the typical consumer would not spend time running these tests or allowing their computer to run for 7 hours on a single application. For this reason, I think that AI will stay a product of enterprise and university level applications of procedural content generation.

In the paper *Procedural Content Generation through Quality Diversity* (Gravina et al.), various approaches of quality diversity(QD) algorithms are discussed. The paper explains that while procedural content generation (PCG) is common, it often does not take quality and diversity into account. My own project for this paper is guilty of not always optimizing the two features in a variety of ways. For example, the quality of my generated biomes is low due to my focus on splitting the terrain generation process into smaller processes rather than an elegant or realistic approach. My chunks also suffer from a lack of diversity since each chunk of the same biome look the same due to the limitation of my random logic.

The paper discusses the use of quality diversity algorithms in order to solve these issues given the benefits of generative efficiency, fitness-free search, and online expressivity analysis. Despite the generative efficiency, the ability of QD to produce high quality solutions with diverse behaviors, QD suffers from requiring a lot of computation. The paper argues that the ability of QD to take more than one dimension of interest gives it a step up against other methods. Overall I think QD solutions would be a good investment for a quality terrain generator although analysis into distributed the workload would need to be done to make it cost-effective.

GANs and QD algorithms are integral to improving the efficiency, quality, and diversity of terrain generation models. For my terrain generation, utilizing GANs would allow me to rapidly test and debug our generated terrain models so that more seeds work with the generator

without unexpected errors. QD algorithms would help create a framework for how to add more variation to the map in the form of terrain smoothing, cave and tree placements, and other post generation processes. Without these types of algorithms being applied to the output, the terrain generator would be just theoretical and not practical.

Implementation & Support Functions

The bulk of my code is based off the structure of a 3-dimensional voxel map like what is found in the game Minecraft. The world uses a 2-dimensional map of chunks to hold a total of 102,400 blocks in each chunk. A single chunk is arranged in a 32x32x100 array and the chunk map acts as a tile map where chunks can be added to the side of existing chunks. For those unfamiliar with how space is represented in the game Minecraft, a chunk is a 32x32 block section of the world with 100 blocks for height. The chunk structure can then be used to create a modular map of a massive 3d world. Each chunk has an x and y location within the map, a Biome chosen based on randomly generated noise data for rainfall and temperature and is either a land chunk or a body of water (regardless of the biome).

Overall, by using the Block, Chunk, and Map classes to manipulate the map during Map generation we are overcomplicating the problem, but afterwards do not need to be as robust and complicated with our data representation. Ultimately, the final choice of the block's type is the most crucial piece of information because any visualizations will use that data. During the generation of a new map, there are various processes that need to happen. The topmost level processes are the creation of an initial starting land mass, the generation of the rainfall and

temperature data, using the rainfall and temperature data to pick a biome for each chunk, and finally generating the chunks according to the land mass and biome data.

According to Zucconi, the starting land mass in the Minecraft source code creates a map with a 1 in 10 chance of a chunk being land and 9 in 10 chances being water (*The World Generation of Minecraft*). The land mass is then manipulated using two main operations, zoom and addIsland. The zoom operation doubles the width and length of the map while maintaining the general shape of the previous land masses. This means that for every one square on the map, four new squares must be created. Variance is introduced into the map by allowing the chance for any or all the new four squares to be different than the original one square.

The addIsland operation does not change the size of the map but instead gives squares surrounded by water the chance to turn into land, therefore creating islands. In the Minecraft source code, the addIsland method has a supporting operation called removeTooMuchOcean but I chose not to implement this method due to the small size of the maps I was only able to generate. The issue of too much ocean was much more noticeable in larger scale implementations.

The formation of the world is based upon the Minecraft source code documented by Alan Zucconi (*The World Generation of Minecraft*). In Zucconi's analysis of the source code, the Minecraft world engine is based upon multiple process stacks that combine to create the resulting map. A stack is a set of actions that will be done to the map in order and can be grouped together as a process such as creating the land, adding rivers, and biome management. The main stacks

are the legacy stack, the noise stack, the river stack and the ocean temperature stack. I have chosen to implement my own versions of the legacy and noise stack.

While just these two stacks will not create as “realistic” or “logical” terrain, they introduced enough of a computational challenge to be completed on a distributed computing system. The map can be compared to a digital photo of the world, as you zoom in to the photo of the map, more details become visible than before. At the smallest logical size for a map generated using these stacks, a 2x2 grid must be “zoomed in” twice resulting in a grid of 8x8 chunks. This is a total of 6,553,600 block objects, 64 chunk objects, and many other supporting objects and methods. The largest map I was able to generate on my home system was a 4x4 starting grid representing chunks, which resulted in a final grid of 16x16 chunks, 256 in total and over 26 million blocks.

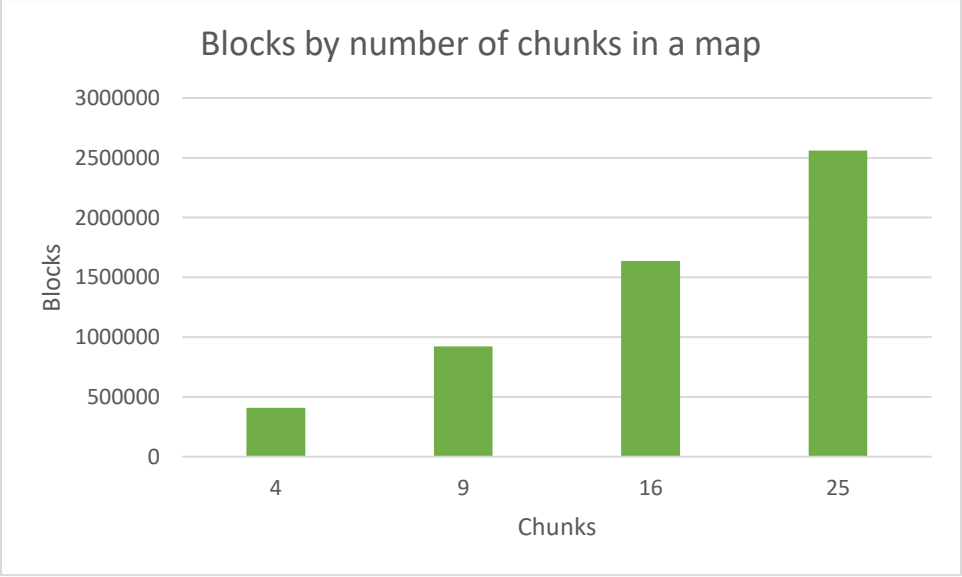


Figure 1: Scaled by map size as width and length increase by 1 chunk each.

As the size of the map, we want to generate gets larger, the number of computations and the amount of time required increases as well. In software development, the number of computations is often expressed in terms of n, n being the number of elements in a list. This is because in the worst-case scenario when we need to perform an action on a single element, we need to find that element inside the list or whatever data structure it is stored in. While there are algorithms that reduce zsmaller complexities such as log n, we can note that the number of blocks shoots up to over 25 million for a map with 25 chunks or a 5x5 map.

Below, I break down the amount of time used for each map between the time it takes to generate all the data and the time it takes to write that data to a file for use elsewhere. Unfortunately, my first implementation of storing a map to a file resulted in a large and relatively useless file. After analysis, the data that needs to be stored can be optimized by only storing the type of block for all 102,400 blocks in a chunk. Previously, the compiler had also been unable to keep all of the chunks in memory at once so each chunk is given its own file named chunkX_Y.json with X and Y being the location of the chunk in the map.

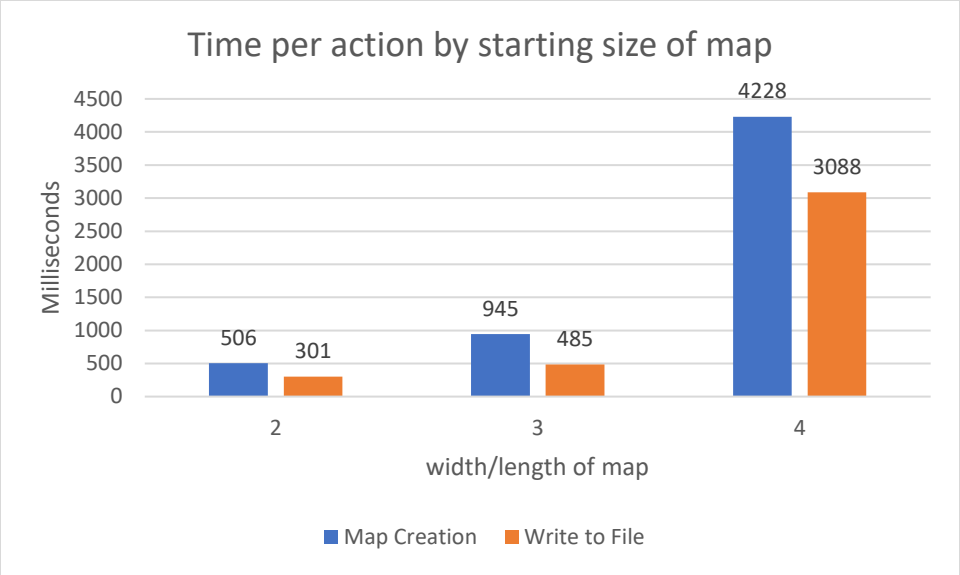


Figure 2: Showing the length of time each process takes based on size

The next step was to generate noise maps that would represent the rainfall and the temperature. The original source code contained ten possible biomes while my implementation will use four, three for actual map generation and the fourth being void and simply used for testing. After generating the noise maps, I saved both to the MapGenerator object because the map itself will not need that data, just the resulting biomes. For generating the Biomes themselves, I calculate the type of biome for each chunk using a new 2-dimensional Array that holds the Biome type as a product of the corresponding x,y position in the noise maps.

In the noise maps below we can see squares with varying intensity of gray with black indicating the highest value and white representing low values. In the figures below, you can see the noise and biome maps for three respective starting sizes. The noise and biome maps are generated only after the landmass is fully formed since that will be the final size of the map.

Rain Fall 2x2,3x3,4x4

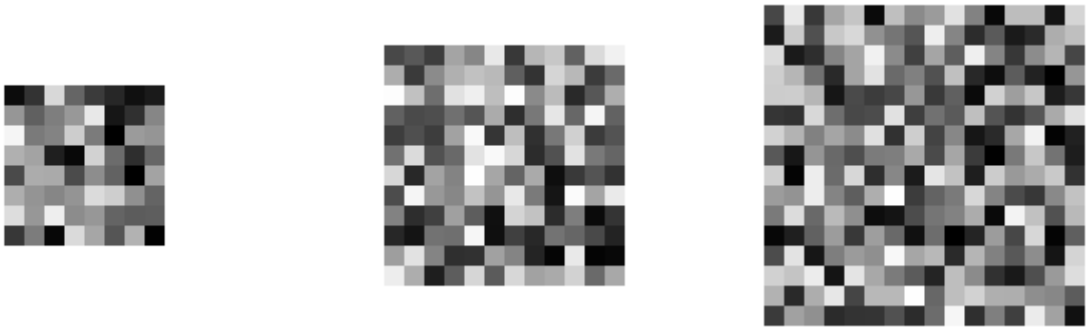


Figure 3: Rain Fall Noise Map for maps with starting size 2x2,3x3,4x4

Temperatures 2x2,3x3,4x4

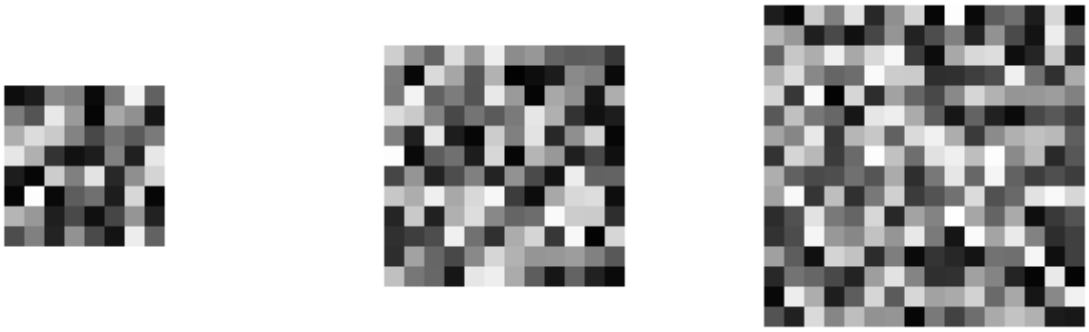


Figure 4: Temperatures Noise Map for maps with starting size 2x2,3x3,4x4

Biomes 2x2,3x3,4x4

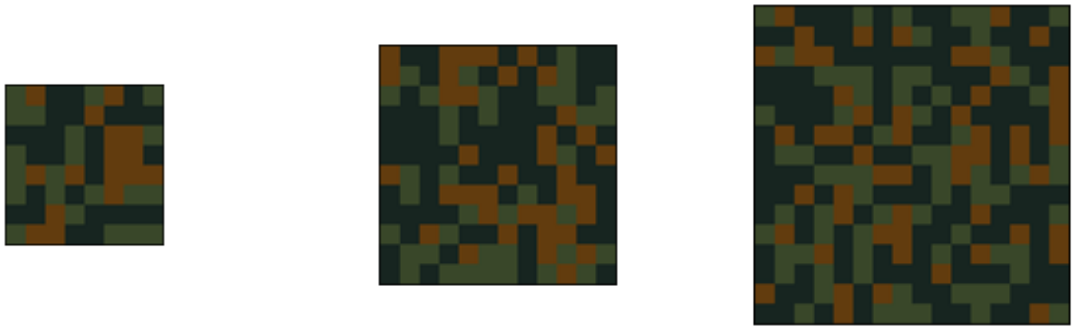


Figure 5: Biome map for maps with starting size 2x2,3x3,4x4

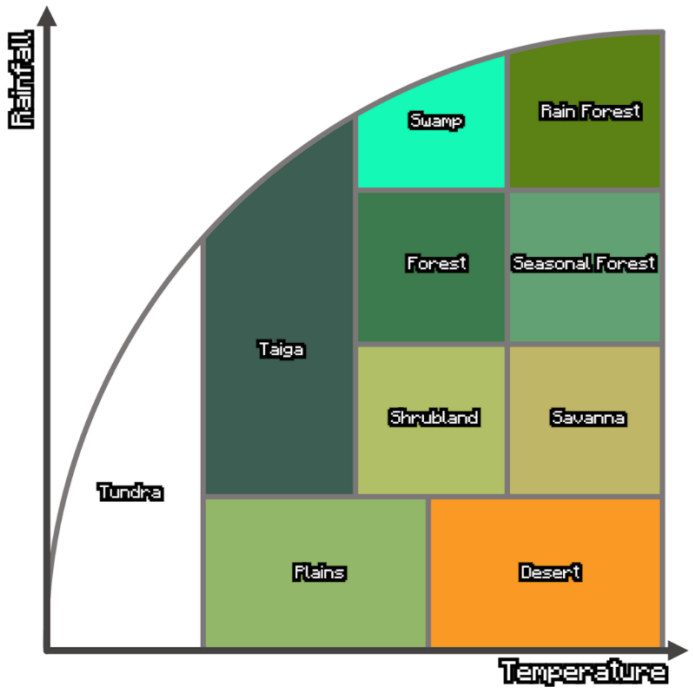


Figure 6: Minecraft Biome Chart (Alan Zuconni The World Generation of Minecraft)

Implementing a unique generation algorithm for ten different types of biomes was beyond the scope of this project so I chose three main and common biomes to implement. The forest biome is one of the most common biomes, both in real life and in Minecraft. The plains and desert biome both share relatively baren landscapes, but differ greatly in their composition of soil, dirt, and sand. In the diagram above I show how Minecraft determines which biome to assign a chunk while the bottom diagram shows my simple three-biome version.

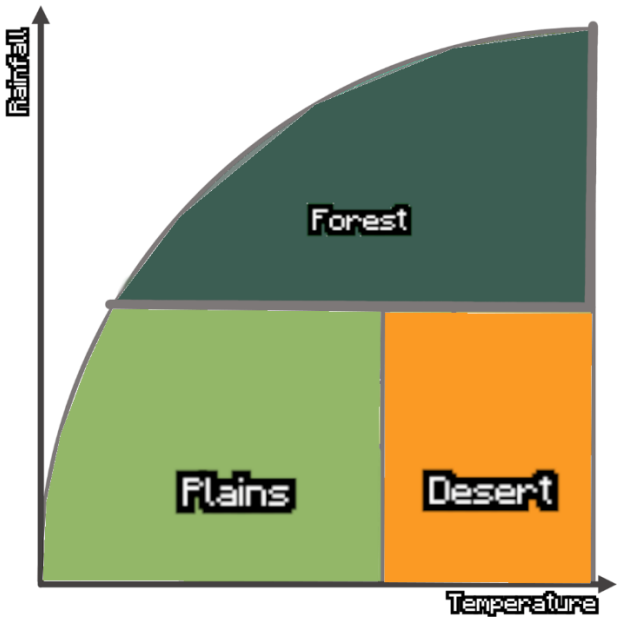


Figure 7: My three-biome chart

The final step in generating a map with this method is creating individual chunks with the corresponding data. In this case, each chunk receives an x,y position in the map, a biome assignment, whether the chunk is land or ocean, and the seed, a large value used in random number generation, used to generate the map for use generating the blocks within each chunk. The new chunks are all assembled within a 2d array and passed into the Map class for final storage. A single chunk will represent a 32x32 section of blocks within the whole map and will contain blocks depending on the biome and whether it is ocean or land.

After running the simulation a few times, it became clear that seeing the progression of each step in the generation would help with debugging and designing features for the Map Generator. Therefore, I also created methods in my original code base that would print the values of arrays such as the rainfall, temperature, Biomes, and landmass to a text file. I then also created three Python scripts, one to handle the continuous data for rainfall and temperature, one to display the difference between a land and ocean chunk, and one to display the biome of a chunk.

The Python script not only shows the data visually in a new window, but it can also process the entire folder and save each image with appropriate names.

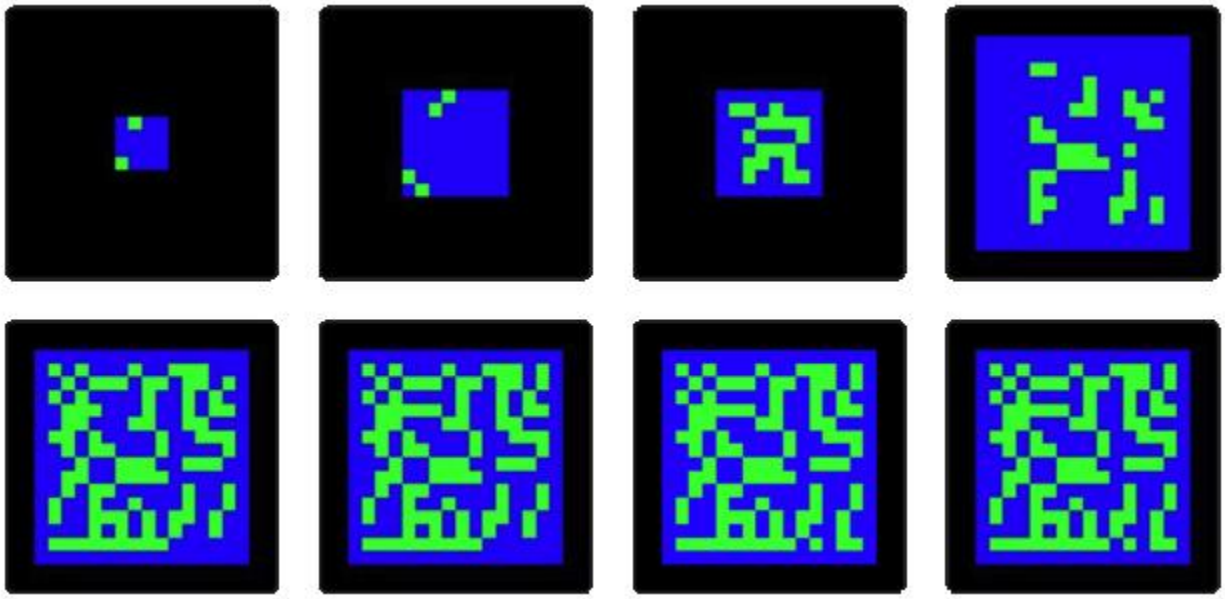


Figure 8: Land mass generation process

Being able to see what the algorithm does after each operation is very helpful when debugging why the resulting maps have the shape they do. In the figure above you can see a map with a starting size of 4x4 grow to full size and islands form. Additionally, seeing the rain fall and temperature maps will help to explain why the resulting biome map looks the way it does.

To summarize, the implementation consists of creating the landmass and ocean chunks using the process stack pictured in the figure above. Afterwards, noise maps of the final map size are created to represent rain fall and temperature using the seed value to determine the randomness. Next each chunk is assigned a biome value based on the combination of its rainfall and temperature values. Finally, each chunk is responsible for using that biome to create a block representation of that biome. By the end of the process, we will have all block types determined for every chunk in the map.

Methodology

Choice of Language

The main language I selected for this project was the language I am most comfortable with and have the most experience with, Java. I also chose to utilize Python as a supporting language due to its many libraries and fast functionality. While there are many languages available to create a distributed system, including but not limited to Java, Python, Rust, Erlang and Go, there are also drawbacks to choosing one language over another. Considering the duration and scope of this project, I spent time with both the languages I knew from that list as well as the languages I did not, and decided that Java, while perhaps not objectively the best, was the best choice for my skills and knowledge.

While I have many years of experience with Java as a language, the language is nearing three decades of use and has millions of developers and even more end users. The tasks that are completed in most educational settings that teach Java are handled in small snippets of code often called modules or deliverables. These are small sections of code that are intended to either be run on their own or as individual parts of a system. The goal of this project, however, was to create not just an individual module but a whole system, specifically a distributed system. Due to this complex task, there are numerous subjects that a novice developer must face for the first time such as using the Hypertext Transfer Protocol (HTTP), Test-Driven Development, and Git version control such as GitHub.

The system for distributed coordination that I chose for this project is Apache Zookeeper, which utilizes Java to manage a server and is highly reliable. For this, I have taken a course, *Distributed Systems & Cloud Computing with Java* taught by Michael Pogrebinsky, on how to

properly implement a Zookeeper server. Other supporting systems I used were Python's Pygame which allowed me to visualize text-based output and save the visualizations as images for use debugging and reporting results.

Java

The organization of my Java code is comprised of four classes that represent the main objects in the program, two enumerations which represent types, and a main class for running metrics on the system. I also utilized Test-Driven Development for the Block and Chunk classes which required the creation of two testing classes not pictured in the Unified Modeling Language (UML) diagram below. A UML Diagram is useful for visualizing a large system and often shows relations between classes, variables associated with each class, and methods associated with each class.

From the diagram below, it can be noted that each class object (box) has a large number of variables (middle section of each box) and methods (bottom section of each box). Most methods are getters, which get a particular variable, and setters, which set a particular variable. The MapGenerator class notably contains none of these since the focus of the MapGenerator class is to create a Map object utilizing its methods rather than store the actual map itself. Originally, I had attempted to contain the functionality of the MapGenerator class within the Map class itself, but this inflated the Map class beyond a reasonable size compared to all other classes and did not make sense.

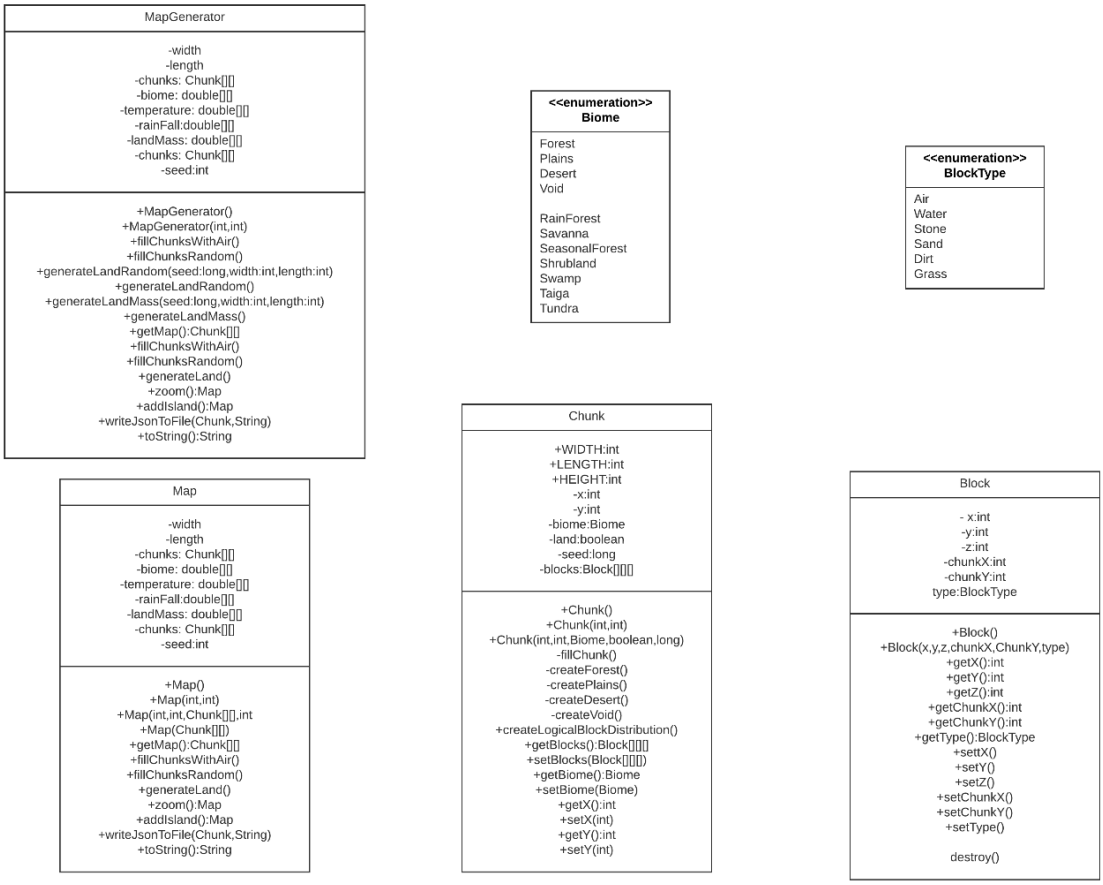


Figure 9: UML diagram showing the Java classes

Block and BlockType

To understand the code, I will start with the building blocks, literally in this case, and work my way up to the finished product. Let us begin with the instance variables of a Block object. These include the x, y, and z locations of the Block object within a chunk, and the x and y variables that represent the location of the parent chunk in the Map object relative to other chunks.

```
private int x, y, z;
private int chunkX, chunkY;
private int universalX, universalY, universalZ;
private BlockType type;
```

Figure 10: Instance variables of a Block Object

The universal x, y, and z variables represent the location of an individual block within the entire map. We can see how this is calculated below in the calculateUniversalLocation method. The universal x and y locations are the sum of the block's location within the chunk and the chunk's location in the map multiplied by the width of a chunk. Notably, the Z location is the same as the block's z location since the chunks are arranged on a 2D plane unlike the blocks which reside on a 3D plane.

```
Calculates the Universal location of this block in the map
private void calculateUniversalLocation() {
    universalX = x + (Chunk.WIDTH * chunkX);
    universalY = y + (Chunk.LENGTH * chunkY);
    universalZ = z;
}
```

Figure 11: calculateUniversalLocation method for Block Objects

I chose to represent the block's location like this while considering the scalability of my map model. As mentioned in my implementation section, the number of blocks can increase rapidly therefore requiring larger and larger numbers to represent a location. By splitting the location into the local chunk location and the universal location, I can reduce the complexity of the math done when modifying individual blocks down the road.

The final instance variable for a block is the type of block, which is selected from the enumerated list in the BlockType enum. I chose to represent the type of block as an enum to reduce the complexity required to explain what type of block is being represented. In the case of this project, the type of block is a placeholder that will not affect the map generation itself and is instead a way of determining what color or texture the block would be in a visualization of this data. The enum type in Java allows a simplification of the data into an ordinal value which allows the types listed to be represented as a simple integer number. In this case, the BlockType of AIR could be represented by the ordinal value 0 while GRASS would be 5.

```
The type of blocks possible
public enum BlockType {
    AIR, WATER, STONE, SAND, DIRT, GRASS
}
```

Figure 12: BlockType enum with possible values

Chunk

The next largest data type after a Block is a Chunk object which contains a 3-dimensional array of blocks. Like how each Block stores its Chunk's x and y value to save space, the Chunk is used to reduce the complexity of an individual section of a Map object. Currently, the width, length, and height of a chunk are static and final and allow a uniform modularity for the Map object to add and remove Chunks without affecting the integrity of the whole map. By being static, other classes can reference how large a Chunk object should be by saying Chunk.WIDTH, Chunk.LENGTH, and Chunk.HEIGHT anywhere in the code. By being final, the size of the Map is set once and not changeable.

```
public static final int WIDTH = 32;
public static final int LENGTH = 32;
public static final int HEIGHT = 100;

private int x, y;
private Biome biome;
private boolean land;
private long seed;
private Block[][][] blocks;
```

Figure 13: Instance variables of a Chunk Object

In addition to the size of the Chunk's blocks array, the Chunk class holds its x and y value within the map itself. This acts as an identifier for each Chunk and gives the blocks a universal location. Each Chunk is also assigned a Biome, whether it is land or ocean, and a seed value to generate the blocks within the chunk. The process of generating each individual Chunk is started by calling the fillChunks method which selects the appropriate method to use depending on the Biome. For this project I have implemented methods for the Forest, Plains, Desert, and Void Biomes as shown below. Implementing another Biome would start with adding an additional option in the fillChunks method.

```
Directs to the correct Fill method for the chunk's biome
private void fillChunks() {
    switch (biome) {
        case Forest:
            createForest();
            break;
        case Plains:
            createPlains();
            break;
        case Desert:
            createDesert();
            break;
        case Void:
            createVoid();
            break;
```

Figure 14: fillChunks Method for Chunk Objects

createForest and createPlains

The createForest Method is relatively simple although looking at the code directly may give an ugly impression. The basis for the method is to go one by one through the 3-dimensional array of blocks and assign a type of value based solely on the height of the block after determining whether the biome will be land or ocean. This is done by creating ranges for each layer within the chunk. The layers for the forest biome are organized as air, dirt and grass, just dirt, and finally stone. The ocean variation for this biome consists of air, water, dirt, and stone. The layering is relatively simplified due to the focus being on the ability of this system to be distributed rather than accurate.

```
Fills chunk with a dirt and grass top
private void createForest() {
    //Loop through every block
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                //set default type to AIR
                BlockType type = BlockType.AIR;
                if (land) {
                    if (h >= 70) {
                        type = BlockType.AIR;
                    } else if (h == 69) {
                        Random rand = new Random(seed);
                        if (rand.nextInt(10) > 5) {
                            type = BlockType.DIRT;
                        } else {
                            type = BlockType.GRASS;
                        }
                    } else if (h >= 60) {
                        type = BlockType.DIRT;
                    } else {
                        type = BlockType.STONE;
                    }
                } else {...}
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);
            }
        }
    }
}
```

Figure 15: createForest Method for Chunk Objects

The createPlains method is like the createForest method with a larger chance of the top layer of land being dirt. Due to the similarity, I have omitted the code for createPlains here, but it will be available in the Code section. The actual difference between the biomes is less important as long as the program is able to handle different types of biomes.

createDesert

The createDesert method goes through a similar process to the previous two biomes but is noticeably different due to its replacement of both dirt and grass with sand and stone. This substitution makes sense due to the lack of soil in a real desert.

```
Fills the chunk with sand on top
private void createDesert() {
    //Loop through every block
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                //set default type to AIR
                BlockType type = BlockType.AIR;
                if (land) {
                    if (h >= 70) {
                        type = BlockType.AIR;
                    } else if (h >= 60) {
                        type = BlockType.SAND;
                    } else {
                        type = BlockType.STONE;
                    }
                } else {...}
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);

            }
        }
    }
}
```

Figure 16: createDesert method for Chunk Objects

createVoid

The final create method is the createVoid method which simply sets all blocks in a chunk equal to the AIR block type. The use of this method is to initialize all 102,400 blocks within a chunk without having to worry about the actual type. This method can be used to test the

extremes of our map generator without worrying about the actual values generated.

```
Fills the chunk with just air
private void createVoid() {
    for (int w = 0; w < blocks.length; w++) {
        for (int l = 0; l < blocks[0].length; l++) {
            for (int h = 0; h < blocks[0][0].length; h++) {
                blocks[w][l][h] = new Block(w, l, h, x, y, BlockType.AIR);
            }
        }
    }
}
```

Figure 17: createVoid method for Chunk Objects

printBlocks and toString

The printBlocks and toString methods are particularly useful for quickly analyzing the contents of a chunk without requiring external visualization software. It does this by unwrapping the 3-dimensional array such that it is split into slices vertically rather than horizontally. It also utilizes the ordinal value of the Block's type rather than the enumerated type itself.

```
0:000000000000000000000000000000000 1:000000000000000000000000000000000 2:000000000000000000000000000000000
  000000000000000000000000000000000   000000000000000000000000000000000   000000000000000000000000000000000
  000000000000000000000000000000000   000000000000000000000000000000000   000000000000000000000000000000000
  000000000000000000000000000000000   000000000000000000000000000000000   000000000000000000000000000000000
  555555555555555555555555555555555   555555555555555555555555555555555   555555555555555555555555555555555
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  444444444444444444444444444444444   444444444444444444444444444444444   444444444444444444444444444444444
  222222222222222222222222222222222   222222222222222222222222222222222   222222222222222222222222222222222
```

Figure 18: Output of the printBlocks method of a Chunk. Each number to the left of a section represents a layer of the chunk sliced like a cake. Rows represent the chunk being sliced horizontally starting from the stop. The numbers in the sections represent ordinal values of the BlockType.

```
Outputs the blocks inside a chunk by slicing into the chunk vertically. Displays a vertical slices left to right on the screen
Returns:
public String toString() {
    String out = "";
    for (int h = blocks[0][0].length - 1; h >= 0; h--) {
        for (int l = 0; l < blocks.length; l++) {
            if (h == blocks[0][0].length - 1) {
                out += (l + ":");
            } else {
                out += ("  ");
                if (l > 9) {
                    out += (" ");
                }
            }
            for (int w = 0; w < blocks[0].length; w++) {
                out += blocks[w][l][h].getType().ordinal();
            }
            out += (" ");
        }
        out += "\n";
    }
    return out;
}
```

Figure 19: The toString method which allows the Chunk to be output as figure 18.

Map

The next actual storage data type of the program is the Map class which consists of a height and width and a 2-dimensional array of Chunk objects. The class also contains the seed used to generate the Map data as well as a Random object created using the seed.

```
public final int width;
public final int length;
private long seed;
private Random rand;
private Chunk[][] chunks;
```

Figure 20: Instance variables of a Map Object

The Map class is relatively simple with the generation of chunk data done in the MapGenerator class and the Map constructor simply taking in all the required data at construction. Setting everything except the chunks is relatively simple, while the chunks require creating a deep copy by creating a brand-new Chunk in the setChunks method. A deep copy is where you create a new object entirely rather than connecting two objects by reference which is the default method in Java. By utilizing a deep copy, we can create the same array of chunks with the MapGenerator class but have two separate map objects that can create different implementations of the same Chunk array. In other words, this allows the user to create multiple map objects with the same MapGenerator and not have to worry about sharing data references.

```
public void setChunks(Chunk[][] chunks) {
    this.chunks = new Chunk[chunks.length][chunks[0].length];
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < length; y++) {
            this.chunks[x][y] = new Chunk(chunks[x][y]);
        }
    }
}
```

Figure 21: setChunks method for Map Objects

MapGenerator

The MapGenerator class is where most of the functionality of the program is completed. The class contains the parameters for the map to be generated, the seed and random object used to generate the data, temperature and rainfall arrays, the array of biomes, and the array that determines whether a chunk is land or ocean. Finally, *the MapGenerator contains its own Map object to store the final Map generated.

```
private int mapWidth;
private int mapLength;
private long seed;
private Random rand;

private double[][] temperatures;
private double[][] rainFall;

private Biome[][] biomes;

private boolean[][] landMass;

private Map map;
```

Figure 22: Instance variables for MapGenerator Objects

MapGenerator Constructor

The steps to create a Map are best seen at a top level from the MapGenerator class constructor. After setting the starting values and creating a Random object for use in the generation, the four main steps are to generate the landmass array which sets the final size of the Map, generate the temperature and rainfall noise maps, generate the biomes using the rainfall and temperature, and finally generating the Map object itself.

```
public MapGenerator(int startWidth, int startLength, long seed) {
    this.mapWidth = startWidth;
    this.mapLength = startLength;
    this.seed = seed;
    rand = new Random(seed);

    generateLandMass();//creates final size

    temperatures = generateNoiseMap();
    rainFall = generateNoiseMap();
    generateBiomes();
    generateMap();

}
```

Figure 23: Constructor for the MapGenerator Object

generateLandMass

The first step of the Map generation process is to generate the landmass array which represents not only whether a Chunk is a land or ocean Chunk, but also the final size of the Map. This is because within the generateLandMass method, the zoom method doubles the width and the length of the map. The generateLandMass method is my implementation of the legacy stack from the original Minecraft source code. The generateLandMass method contains four main methods, the printLand method, the generateStartLandRandom method, the zoom method, and the addIsland method. By combining these methods, the generator can create a complex map with continents and oceans.

```
Creates an array of booleans indicating whether a chunk of the map will be ocean or land.
public void generateLandMass() {
    generateStartLandRandom();
    int count = 0;
    printLand(count++, "init");
    zoom();
    printLand(count++, "zoom");
    addIsland();
    printLand(count++, "addIsland");
    zoom();
    printLand(count++, "zoom");
    addIsland();
    printLand(count++, "addIsland");
    addIsland();
    printLand(count++, "addIsland");
    addIsland();
    printLand(count++, "addIsland");
    landMass = removeTooMuchOcean(landMass);
    addIsland();
    printLand(count++, "addIsland");
    printLandToFile();
}
```

Figure 24: generateLandMass method for the MapGenerator Object

printLand

The printLand method is one of the most helpful methods for debugging and visualizing the map generation process. At each step it outputs a text-based version of the map to the generateLandMassSteps folder and by utilizing a supporting Python script called createLandMassImg.py I was able to convert 0s and 1s into colored squares.

```
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

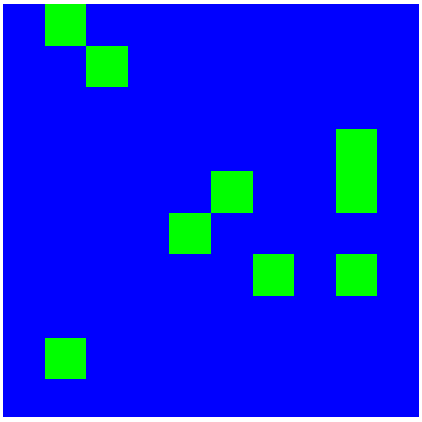


Figure 25: Text-based representation of the landmass(left) and graphical representation made with Python script (right)

generateStartLandRandom

The generateStartLandRandom method is straight forward but necessary to start the process. It starts by taking the mapWidth and mapLength of the MapGenerator object and setting the size of the boolean, true or false value, landMass array. Afterwards it assigns each element in the array a 1 in 10 chance of being land. This method creates the starting continents of the Map as indicated by green on the map. Ocean chunks are indicated in blue.

```
Creates an initial landMass randomly
private void generateStartLandRandom() {
    //initialize landMass with starting map width and length
    landMass = new boolean[mapWidth][mapLength];
    //give each boolean in the array a value using a 1/10 chance of being land
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            int isLand = rand.nextInt(10);
            landMass[x][y] = isLand < 1;
        }
    }

}
```

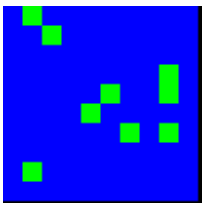


Figure 26: generateStartLandRandom method for the MapGenerator. Example 10x10 starting grid (right).

zoom

The zoom method is the main modifier for the size of the generated Map. The first step of the zoom method is to create a copy of the current landmass array so we can modify our MapGenerator's landmass array without losing data. Next, we instantiate the MapGenerator's landmass array with two times the width and length of the array before. This means that for every one square before, we create four new ones. The zoom feature also has its own chance of not replicating the original square exactly since it was zoomed out before, and the picture can be

said to be getting clearer the more we zoom in. In the green and blue map below, the islands from before are more detailed and the map is larger.

```
private void zoom() {
    //copy landmass into land
    boolean[][] land = new boolean[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            land[x][y] = landMass[x][y];
        }
    }
    //create new array 2x the size of old one
    int newMapWidth = mapWidth * 2;
    int newMapLength = mapLength * 2;
    landMass = new boolean[newMapWidth][newMapLength];
    //set every 4 tiles equal to one from the old array
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            boolean isLand = land[x][y];
            //adding randomness
            landMass[2 * x][2 * y] = isLand && rand.nextBoolean();
            landMass[(2 * x) + 1][2 * y] = isLand && rand.nextBoolean();
            landMass[2 * x][(2 * y) + 1] = isLand && rand.nextBoolean();
            landMass[(2 * x) + 1][(2 * y) + 1] = isLand && rand.nextBoolean();

        }
    }
    mapWidth = newMapWidth;
    mapLength = newMapLength;

}
```

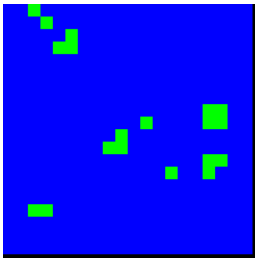


Figure 27: zoom method of MapGenerator Object. Example 20x20 zoomed grid (right).

addIsland

The addIsland method takes the previous map and adds the chance for a new island to form. First the method creates a new 2-dimensional Boolean array like the zoom method but does not double the width and length. Instead, as it copies the previous array into the new array it

counts the number of ocean chunks surrounding the current chunk. If there are more than 7 ocean chunks in a 3x3 area, the center chunk has the chance of becoming a land chunk. After every run of the addIsland method, the outer perimeter of the map becomes all ocean chunks. This is to provide a visual edge of the map using the ocean rather than having to explain that a land chunk just ends. The chance of an edge chunk being land is entered in the zoom method, but the addIsland method is the last method that modifies the map called in the land generation process.

```
private void addIsland() {
    //create new land array to hold new map with more islands
    boolean[][] newLand = new boolean[mapWidth][mapLength];
    //loop through all current chunks except edges
    for (int x = 1; x < mapWidth - 1; x++) {
        for (int y = 1; y < mapLength - 1; y++) {
            //only adding land so need to check ocean(false)
            if (!landMass[x][y]) {
                int count = 0;//check number of water tiles surrounding current block
                //counts amount of water surrounding square
                for (int i = -1; i <= 1; i++) {
                    for (int j = -1; j <= 1; j++) {
                        if (!landMass[x + j][y + i]) {
                            count++;}}                    }
                //Change chances of an island forming
                if (count > 7) {
                    //if the surrounding 8 tiles are water
                    //50% chance of turning into water
                    if (rand.nextInt() % 2 == 0) {
                        newLand[x][y] = false;
                        //50% chance of turning into land
                    } else {
                        newLand[x][y] = true;}
                } else {
                    //if less than 7 out of 9 tiles are water keep this tile water
                    newLand[x][y] = false;}
                //if the tile is already land set new array to the same
            } else {
                newLand[x][y] = true;}}}
    landMass = newLand;}
```

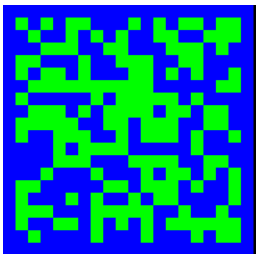


Figure 28: addIsland method of MapGenerator Object. Grid full of islands after addIsland method is called on grid from figure 27 (right).

printLandToFile

The print land to file method is a helper method that allows the program to print a final state for the landmass before passing it to the Map object via the generateMap method. First, the method creates a new directory to store the output in, and then creates a new file within that directory called landMass.txt. The true and false values of landMass are then translated into 0s and 1s for used by the supporting Python script *createLandMassImg.py*.

```
private void printLandToFile() {
    try {
        // create a new directory called "landmass"
        File directory = new File("landMass");
        if (!directory.exists()) {
            directory.mkdir();
        }
        // create a new file called "landMass.txt" in the "landmass" directory
        File file = new File(directory, "landMass.txt");
        FileWriter fw = new FileWriter(file);

        // write the contents of the landMass array to the file
        for (int y = 0; y < mapLength; y++) {
            for (int x = 0; x < mapWidth; x++) {
                if (landMass[x][y]) {
                    fw.write(1 + " ");

                } else {
                    fw.write(0 + " ");
                }
            }
            fw.write("\n");
        }
        fw.close();
    } catch (IOException e) {
        System.out.println("An error occurred while writing to the file.");
        e.printStackTrace();
    }
}
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 1 0 0 0 0 1 1 0 1 0
0 0 1 0 1 1 1 1 0 1 1 0 0 0 1 0
0 1 0 1 1 1 0 0 1 1 0 0 0 0 1 0
0 0 1 0 1 0 1 1 0 0 1 1 1 1 1 0
0 0 1 0 1 0 0 1 0 0 1 0 1 0 1 0
0 0 1 0 1 0 0 0 1 1 0 0 1 1 1 0
0 1 0 0 0 1 0 0 1 1 0 1 0 0 1 0
0 0 1 1 1 1 0 0 1 1 0 0 1 1 1 0
0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 1 1 1 1 1 0 0 1 0 1 1 1 0 1 0
0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 1 0 0 0 1 1 1 0
0 1 1 0 1 0 1 1 0 0 1 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 29: printLandToFile method of MapGenerator Object. Text-based output (right).

generateNoiseMap

The generateNoiseMap is crucial to the procedural aspect of this terrain generator. It allows us to create an array of pseudo-random decimal values, called doubles in Java, which we can use to calculate the rainfall and temperature for each map. It is also where the use of a seed is important because it allows the randomness to be repeatable for every run of the system. The method starts by creating a 2-dimensional array of doubles the size of the map at the end of the landmass generation phase. Next it utilizes the Random object rand from the MapGenerator's instance variables to assign a pseudo-random number to every spot in the array. The bottom left image below shows what the data looks like in a text file and the bottom right shows what the data looks like after going through the supporting *createGrayscaleImages.py* script. We use this method twice, once to generate the noise map, a random 2D array of doubles, once for rainfall and once for the temperatures.

```
public double[][] generateNoiseMap() {
    double[][] noiseMap = new double[mapWidth][mapLength];

    // Generate random values for the noise map
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            noiseMap[x][y] = rand.nextDouble(); // Generate random value between 0 and 1
        }
    }

    return noiseMap;
}
```

Figure 30: generateNoiseMap method for the MapGenerator class

```
0.998398 0.768664 0.702471
0.073825 0.755386 0.608392
0.772738 0.160475 0.469223
0.736526 0.197621 0.337839
0.757743 0.915027 0.364900
0.175623 0.242507 0.288063
0.049563 0.436915 0.975362
0.832059 0.871773 0.342644
0.660766 0.730922 0.473063
```

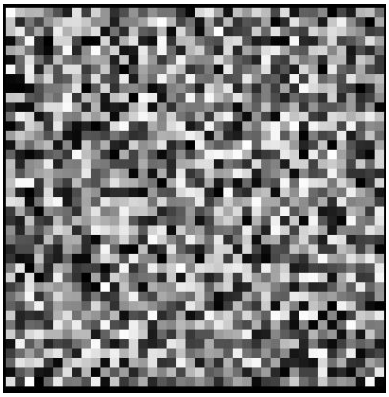


Figure 31: Text-based representation of a noise map (left) and graphical representation using a grayscale using the createGrayscaleImages.py *script (right).*

generateBiomes

The generate Biomes method uses the previously generated noise maps to calculate the type of Biome for every chunk. If the amount of rainfall for a chunk is greater than 50%, then the Biome will be Forest. If the amount of rain is less than 50% and the temperature is greater than 50%, the biome will be Desert. If neither of the previous options are true, the biome will be Plains. This selection is done for every element in the instance variable 2D Biome array called biomes. The output of the biomes array utilizes the ordinal method of the Biome's enumerated type and will therefore contain a range of values between 0 and n, where n is the number of biomes. In the bottom left image below, I show the data in a text file and the bottom right image shows the data after being run through the *createBiomeImg.py* script.

```
public void generateBiomes() {
    printNoiseMap(temperatures, "Temperatures");
    printNoiseMap(rainFall, "RainFall");
    biomes = new Biome[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            double temp = temperatures[x][y];
            double rain = rainFall[x][y];
            biomes[x][y] = Biome.Plains;
            if (rain > 0.50) {
                biomes[x][y] = Biome.Forest;
            } else {
                if (temp > 0.5) {
                    biomes[x][y] = Biome.Desert;
                }
            }
        }
    }
    printBiomes();
}
```

Figure 33: generateBiomes method of the MapGenerator Object

```
0 0 0 1 0 1 0 2 2 0 2 1 1 0
0 1 0 2 0 1 1 2 2 0 1 0 1 0
0 1 1 0 1 0 0 0 2 1 2 0 0 0
0 2 0 2 1 2 1 0 2 2 0 2 0 0
0 0 1 2 2 1 0 0 2 0 0 1 1 1
1 0 0 0 2 0 1 1 2 2 0 0 0 0
2 0 0 0 0 1 0 2 0 0 0 0 2 1
0 0 0 2 1 1 0 0 0 2 0 2 1 1
1 0 0 2 0 0 1 1 1 0 1 0 1 1
0 0 0 2 0 0 2 0 1 0 0 0 2 1
0 0 2 0 0 0 0 2 0 1 2 0 2 2
2 1 1 1 0 1 1 2 1 0 2 2 0 2
1 0 1 0 1 0 1 0 0 0 2 0 0 1
```

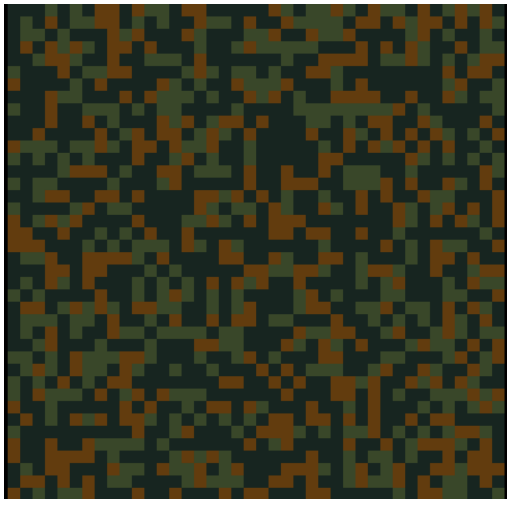


Figure 32: Text-based representation of a biome map (left) and a graphical representation of a biome map created by createBiomeImg.py *(right). Different scales.*

generateMap

Finally, the generateMap method brings together all the created arrays into one final Map object. Although the MapGenerator uses these arrays to create the Map object, the Map object only keeps a reference of the resulting biome, land or ocean, and the seed used stored in the creation of each Chunk object. Finally, the resulting 2D Chunk array is passed into the Map object's constructor with the size and seed. Currently, this way of creating a Map constrains the number of chunks that can be generated in total due to memory limits of the Java Virtual Machine (JVM).

```
private void generateMap() {
    Chunk[][] chunks = new Chunk[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            chunks[x][y] = new Chunk(x, y, biomes[x][y], landMass[x][y], seed);
        }
    }
    map = new Map(mapWidth, mapLength, seed, chunks);
}
```

Figure 34: generateMap method of the MapGenerator Object

For this reason, I propose that this would be the best place to start converting this project into a distributed system. Rather than holding everything in a 2D array, the system could store every chunk into an individual file and read and write to each file individually. Utilizing an Apache Zookeeper server would allow the creation of individual workers changing and modifying the Map at the same time. This would not only cut down on the computational cost of the main processor, but it would also speed up the generation of the map since more than one chunk could be generated at a time.

Main

For metrics used in the implementation section, I utilized the timeMapGeneration method and the timeWriteToFile method to record how long the system took to complete each action in milliseconds. This time can vary based on the system, but it allows a general idea of how long each process takes. When converting this system to a distributed system, understanding how long each process takes would help with designing where to split a process into multiple steps without going too far ahead of other processes. For example, splitting a process up so it takes less time may not make sense if that process is dependent on other slower processes to work correctly.

timeMapGeneration

```
public static Map timeMapGeneration() {
    long startTime = System.nanoTime();

    MapGenerator mapGen = new MapGenerator();
    Map test = mapGen.getMap();

    long endTime = System.nanoTime();
    long duration = (endTime - startTime) / 1000000; // convert to milliseconds
    System.out.println("Time taken: " + duration + "ms");
    return test;
}
```

Figure 35: timeMapGeneration method for the Main Class

timeWriteToFile

```
public static void timeWriteToFile(Map test) {
    Chunk[][] chunks = test.getChunks();
    String fileName;
    long startTime = System.nanoTime();
    try {
        for (int x = 0; x < chunks.length; x++) {
            for (int y = 0; y < chunks[0].length; y++) {
                fileName = "chunk" + x + "_" + y + ".json";
                MapGenerator.writeJsonToFile(chunks[x][y], fileName);
            }
        }
        System.out.println("JSON file written successfully");
    } catch (IOException e) {
        System.err.println("Error writing JSON file: " + e.getMessage());
    }
    long endTime = System.nanoTime();
    long duration = (endTime - startTime) / 1000000; // convert to milliseconds
    System.out.println("Time taken: " + duration + "ms");

}
```

Figure 36: timeWriteToFile method for the Main Class

Python

Getting file data

In all three of my supporting Python scripts, I utilized the OS library in Python to use the system directory selector so that I could select a directory of text files to convert into image files. The three steps I took were to get the directory, get all the text files in that directory, and load each file into the program for conversion into an image.

```python
# Prompt the user to select a directory
def get_directory():
    root = tk.Tk()
    root.withdraw()
    return filedialog.askdirectory()

# Get all text files in the directory
def get_text_files(directory):
    return [file_name for file_name in os.listdir(directory) if file_name.endswith('.txt')]

# Load the data from the file
def load_data(file_path):
    with open(file_path, 'r') as f:
        data = [[int(x) for x in line.split()] for line in f]
    return data
```

Figure 37: Getting file data Python code.

Displaying the file data

For each set of arrays, there were a variety of things being displayed. Therefore each Python script interpreted the data differently. The *createLandMassImg.py* script accepted 0 and 1 as possible values and displayed blue for the ocean if the value was 0 and displayed green for land if the value was 1. The script *createGrayScaleImages.py* expected a decimal value less than 1 and greater than or equal to 0 and used it to create a gray color based on the intensity of it. Since the decimal values were in the range of 0 and 1 and RGB color values are 0-255, I multiplied 255 by the decimal value to get their intensity. Finally, the *createBiomeImg.py* script accepted three possible values, 0, 1, and 2 representing forest—dark Green, plains—light green, and desert—brown biomes respectively. The void biome was not included at this time as it did not make sense at this stage of testing.

createLandMassImg.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                square_color = GREEN if data[data_row][data_col] else BLUE
            else:
                square_color = BLACK
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

Figure 38: draw_squares function of the createLandMassImg.py *script*

createGrayScaleImages.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                #print('data'  + str(data[data_row][data_col]) +' '+str(data_row)  + ' '  + str(data_col) )
                square_color = (data[data_row][data_col]*255,data[data_row][data_col]*255,data[data_row][data_col]*255)
            else:
                square_color = BLACK
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

Figure 39: draw_squares function of the createGrayScaleImg.py *script*

createBiomeImg.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            square_color = BLACK
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                if data[data_row][data_col] == 0:
                    square_color = FOREST
                elif data[data_row][data_col] == 1:
                    square_color = PLAINS
                elif data[data_row][data_col] == 2:
                    square_color = DESERT
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

Figure 40: draw_squares function of the createBiomeImg.py *Script*

Saving the data to an Image

The last step of the Python scripts was to use the *Pygame* module's save method to capture the output of the above display methods. It saved the image as a Portable Network Graphics (PNG) file with the same name as the text file that it was created from.

```
# Save the image as a PNG file
def save_image(screen, directory, file_name):
    png_name = os.path.splitext(file_name)[0] + '.png'
    pygame.image.save(screen, os.path.join(directory, png_name))
```

Further Research

While the work done in this paper demonstrated the use of various design techniques for modularizing complex processes and suggested various distributed implementations, both areas could be explored more deeply. Running more standardized tests on a variety of devices ranging from consumer to enterprise-level and university-level systems could allow a better view of where to focus optimization efforts. Additional expert knowledge into geographic terrain would allow more realistic inspiration than games such as Minecraft.

Additional work utilizing AI techniques such as GANs and QD algorithms could allow more directed research into the effectiveness of procedural content generation in games and other applications without expert knowledge. While some applications such as advanced terrain simulation and modeling would benefit from expert knowledge, games and their enjoyment benefit from soft metrics such as playability, enjoyability and portability.

The practical benefits of optimizing this program would most likely be useful only for developers and designers instead of the consumers of such content. If consumer electronics could run such software, the ability of the average consumer to contribute to the creation of games they enjoy could stem from a healthy relationship with the software they use rather than data mining and marketing strategies as often seen in the gaming industry today. Distributed systems show promise at solving the issue of consumer electronics being unable to do large computations but more research about the applications of Distributed systems needs to be done.

Conclusion

Overall, this paper explored the idea of procedurally generating content, specifically terrain generation, for games and discussed various approaches when designing potentially modular and distributed systems. Through optimizations and suggested distributed implementations for a procedural terrain generator, I discussed the effectiveness of different methods. Further research is still required to verify the effectiveness of these methods at scale and with the addition of more sophisticated methods such as AI and utilizing expert knowledge of relevant subjects.

The potential benefits of procedural content generation for terrains are vast but the overall field of content generation in games could benefit from a democratization of scalable and cost-efficient methods. As technology advances consumer electronics, AI, and Web3 solutions, I believe that procedural content will become more prevalent in the gaming industry and beyond. Continued exploration and research into refining these techniques can help unlock new possibilities for generating high-quality and diverse content throughout digital spaces ranging from terrain to entire digital experiences.

Bibliography

3, Raju Chiluvuri June, et al. “How Many Java Developers Are There in the World?: Plumbr – User Experience & Application Performance Monitoring.” *Plumbr*, 6 Apr. 2015, https://plumbr.io/blog/java/how-many-java-developers-in-the-world.

Alan Zucconi. “The World Generation of Minecraft.” *Alan Zucconi*, 30 Mar. 2023, https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/.

A. Alvarez, S. Dahlskog, J. Font and J. Togelius, "Empowering Quality Diversity in Dungeon Design with Interactive Constrained MAP-Elites," *2019 IEEE Conference on Games (CoG)*, London, UK, 2019, pp. 1-8, doi: 10.1109/CIG.2019.8848022.

Chen, M., Lv, G., Zhou, C. et al. "Geographic modeling and simulation systems for geographic research in the new era: Some thoughts on their development and construction." *Sci. China Earth Sci*. 64 (2021): 1207-1223. DOI: 10.1007/s11430-020-9759-0.

Fontaine, Matthew C., et al. "Illuminating Mario Scenes in the Latent Space of a Generative Adversarial Network." arXiv preprint arXiv:2007.05674 (2021).

Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation. SPRINGER, 2019.

Gravina, Daniele, et al. "Procedural Content Generation through Quality Diversity." *2019 IEEE Conference on Games (CoG)*, IEEE, 2019, doi: 10.1109/cig.2019.8848053.

Hendrikx, Mark, et al. "Procedural Content Generation for Games: A Survey." *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, Feb. 2013, article no. 1, pp. 1-22, doi: 10.1145/2422956.2422957.

Moore, Samuel K. “How and When the Chip Shortage Will End, in 4 Charts.” *IEEE Spectrum*, IEEE Spectrum, 28 Mar. 2023, https://spectrum.ieee.org/chip-shortage.

Risi, S. and J. Togelius. "Increasing Generality in Machine Learning Through Procedural Content Generation." *Nature Machine Intelligence*, vol. 2, 2020, pp. 428-436. DOI: 10.1038/s42256-020-0208-z.

Sarkar, Anurag and Seth Cooper. "Generating and Blending Game Levels via Quality-Diversity in the Latent Space of a Variational Autoencoder." arXiv preprint arXiv:2102.12463 (2021).

Short, Tanya X., and Tarn Adams. *Procedural Storytelling in Game Design*. Taylor & Francis, 2019.

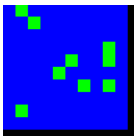
Smith, Gillian. “An Analog History of Procedural Content Generation.” *International Conference on Foundations of Digital Games (2015)*.

Viana, Breno M. F., et al. "Illuminating the Space of Enemies Through MAP-Elites." arXiv preprint arXiv:2202.09615 (2022).

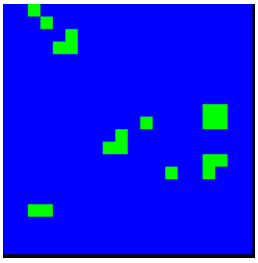
Appendix A: Images

Generating Map Steps

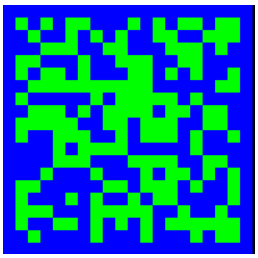
00_init



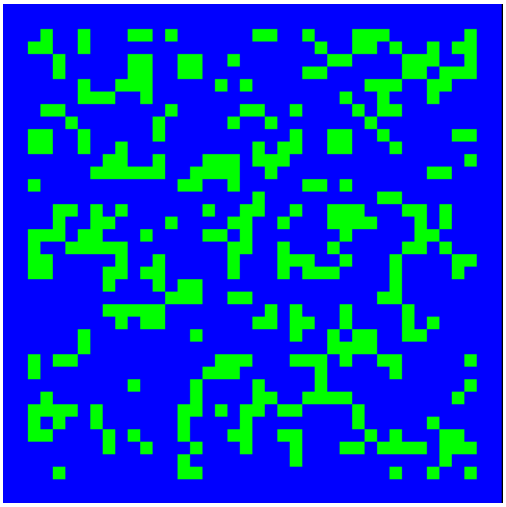
01_zoom



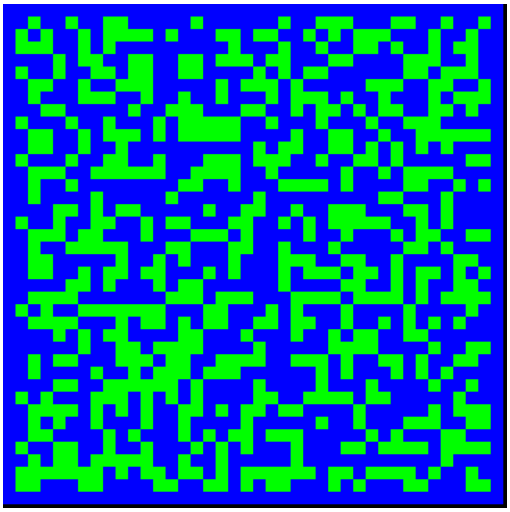
02_addIsland



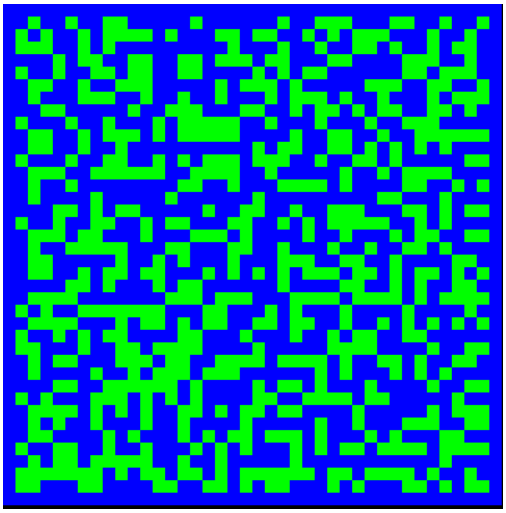
03_zoom



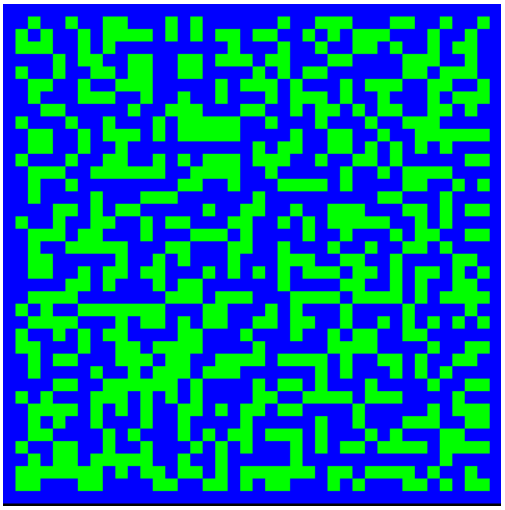
04_addIsland



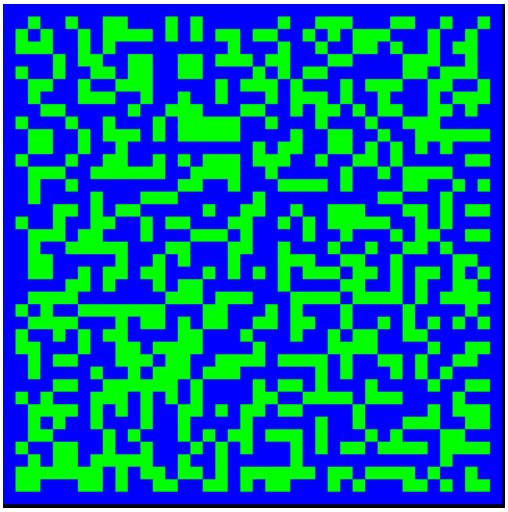
05_addIsland



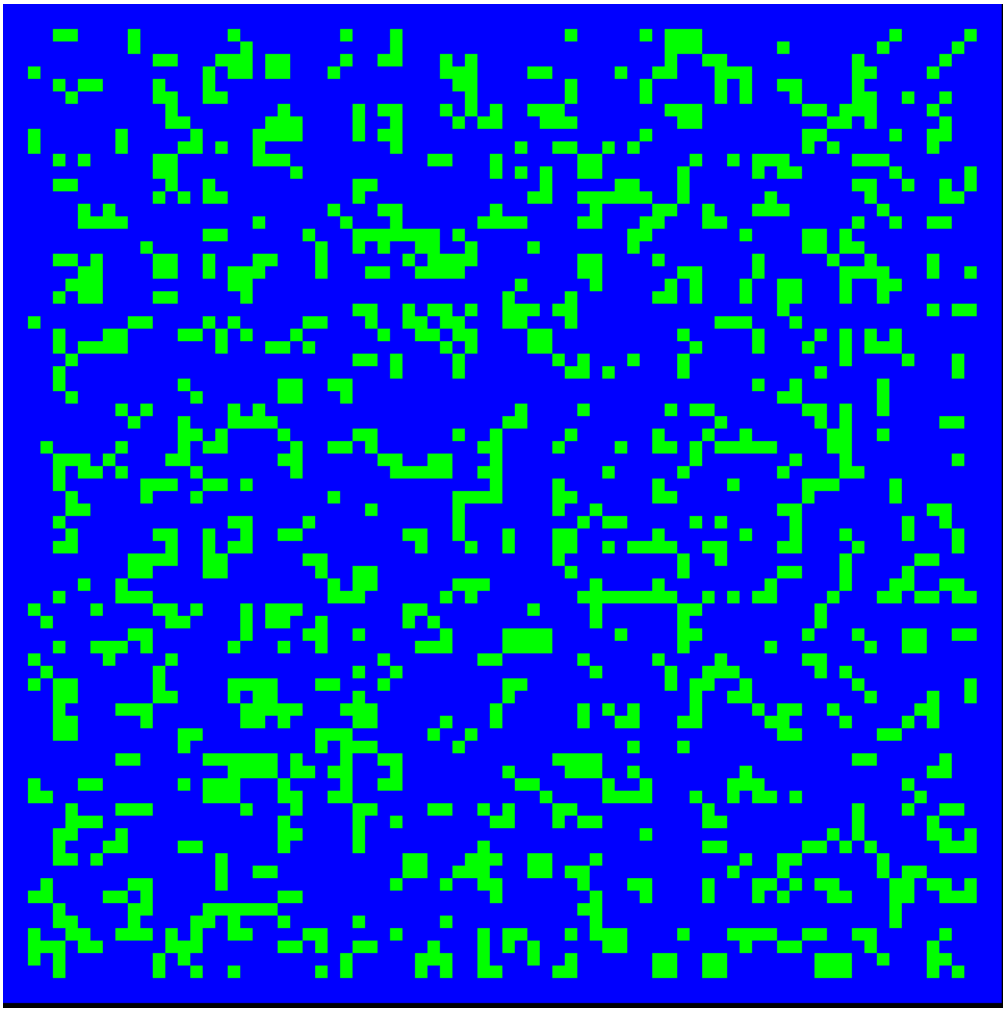
06_addIsland



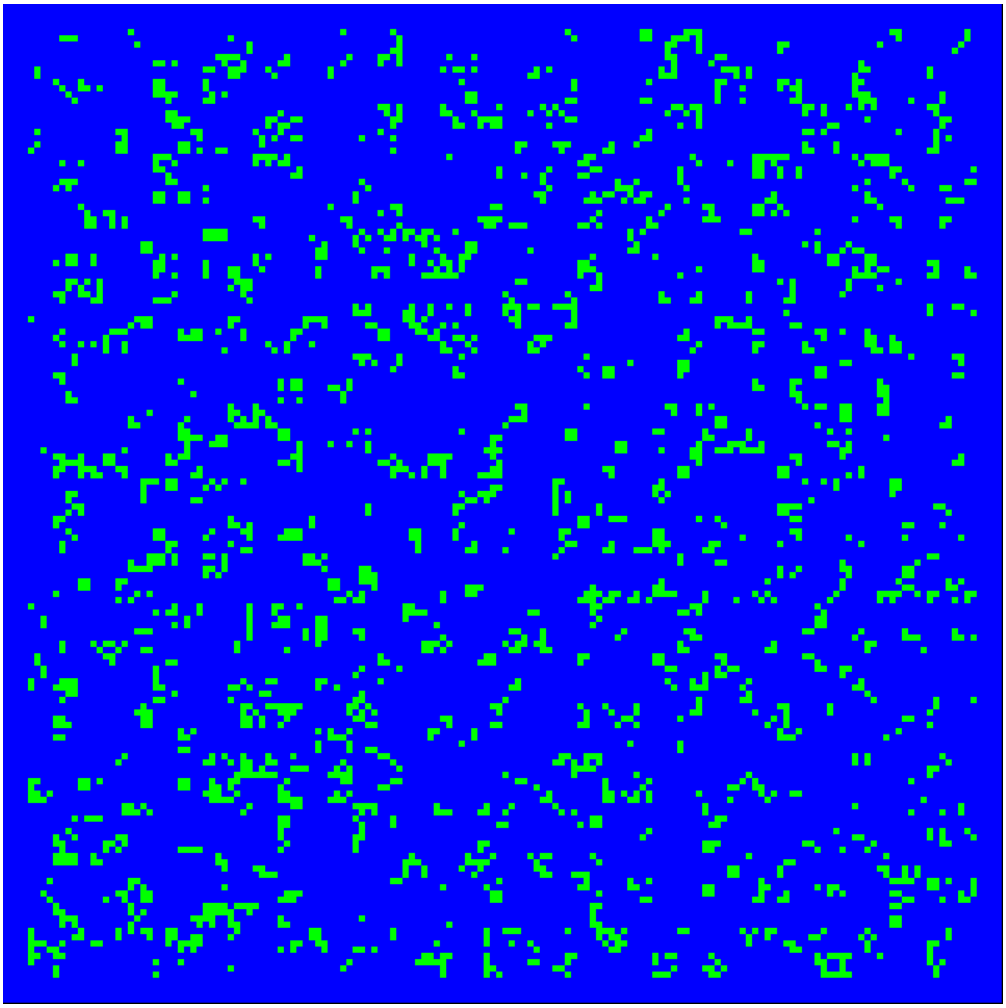
07_addIsland



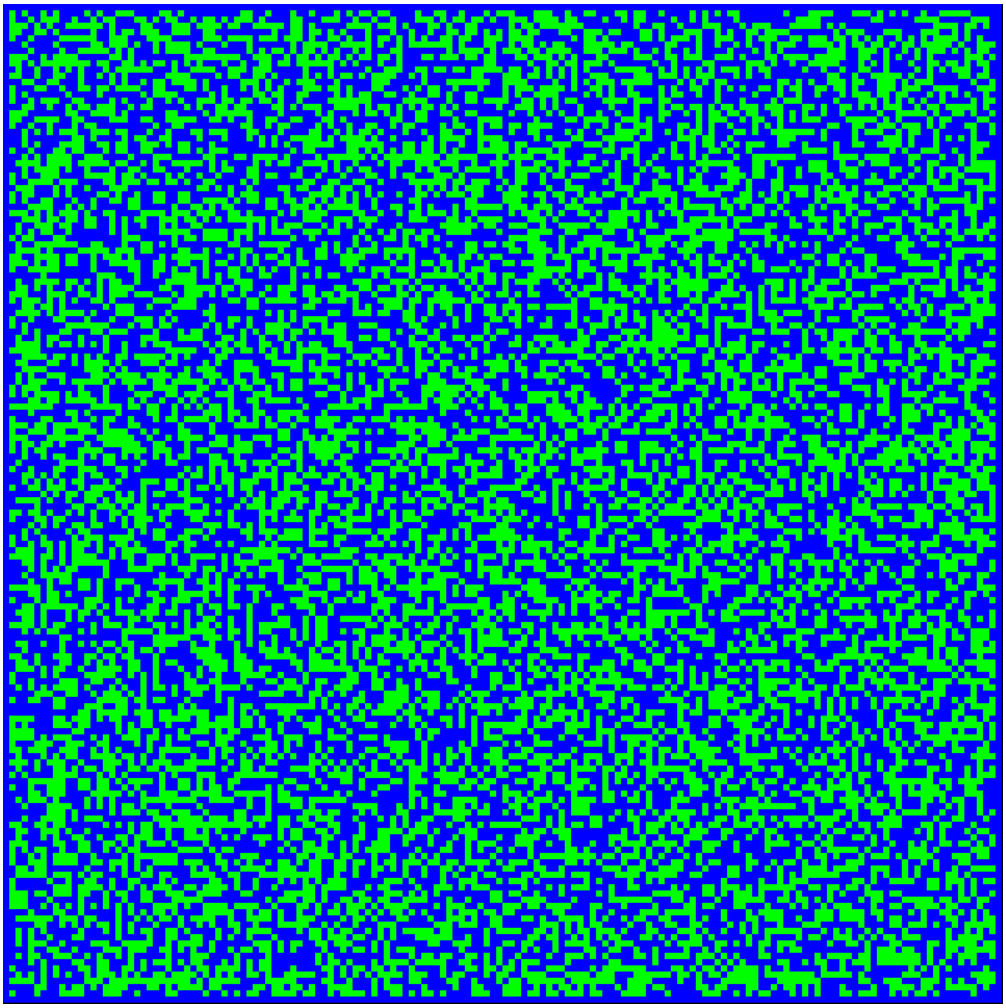
08_zoom



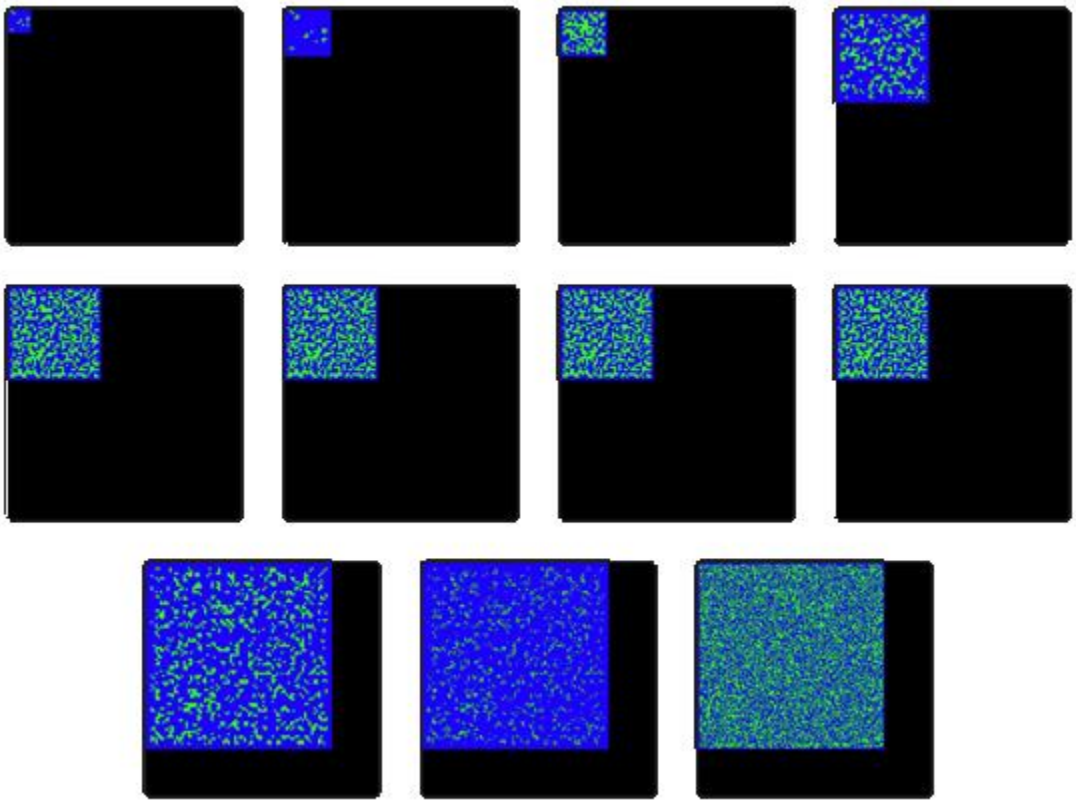
09_Zoom



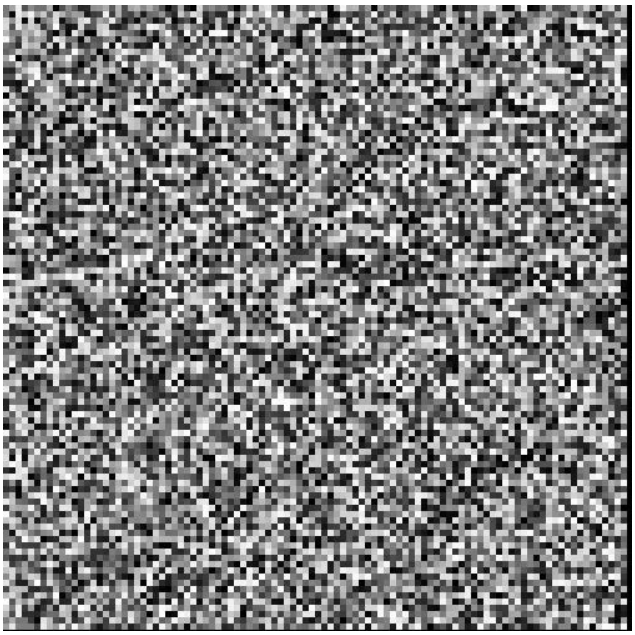
10_addIsland



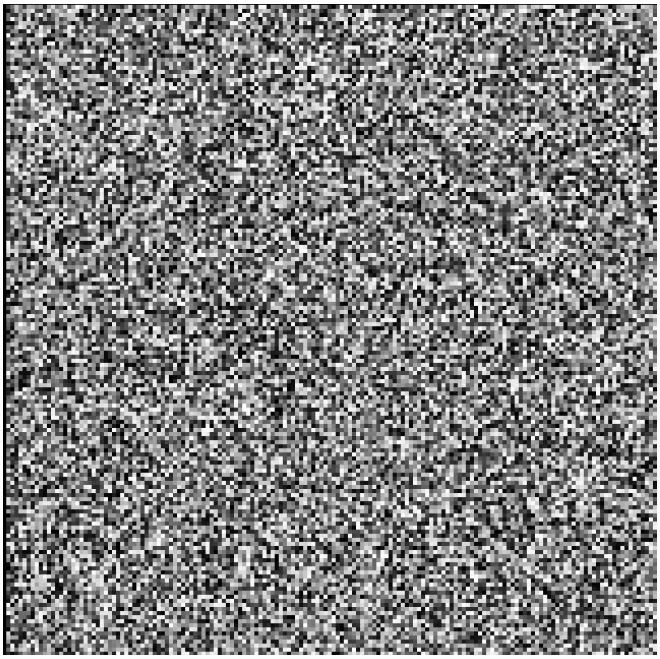
generateLandMass



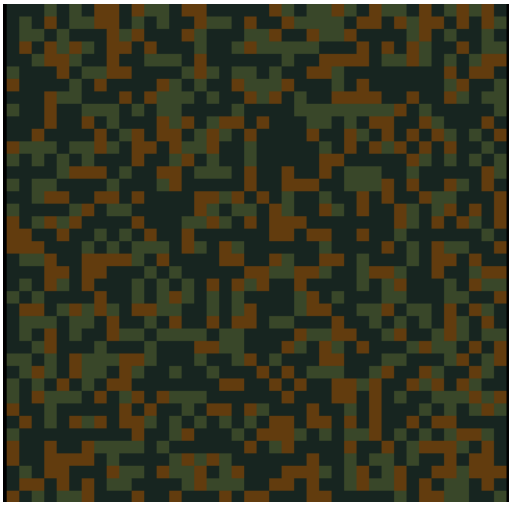
RainFall



Temperatures



Biomes

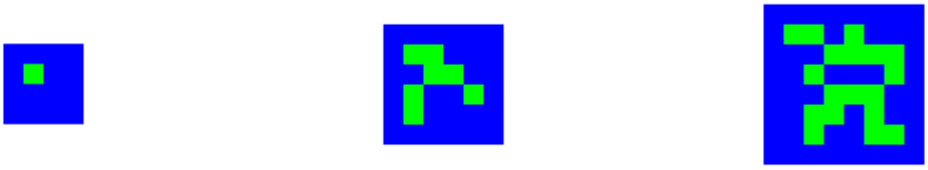


Varying Sizes

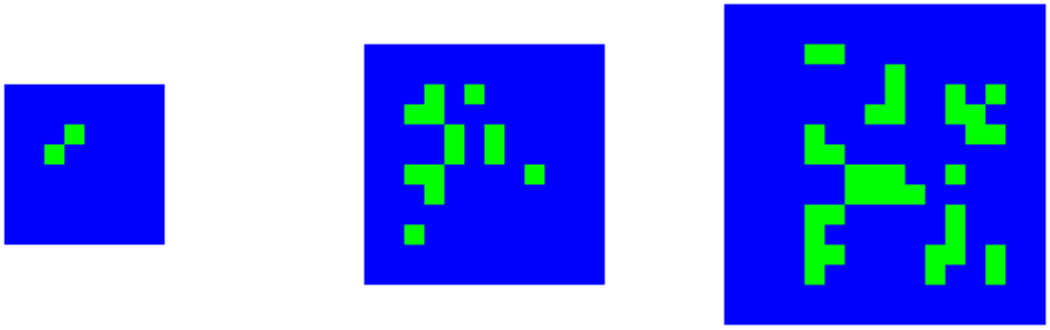
00_init 2x2,3x3,4x4

01_zoom 2x2,3x3,4x4

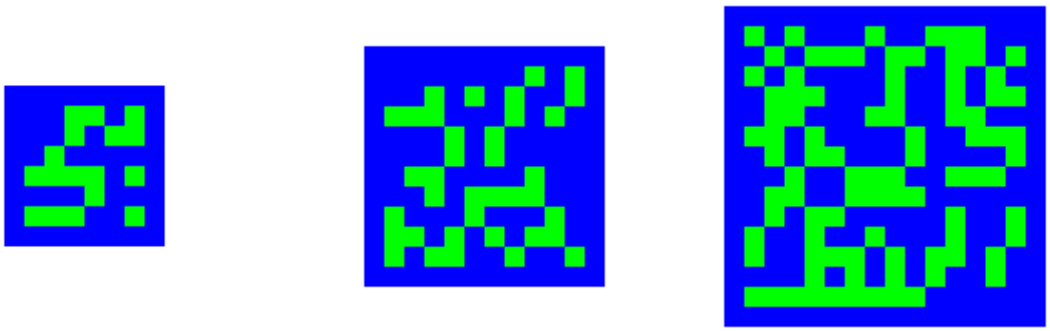
02_addIsland 2x2,3x3,4x4



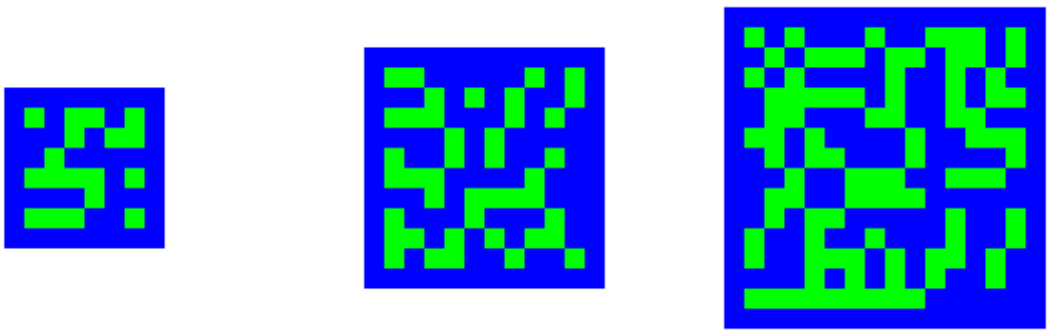
03_zoom 2x2,3x3,4x4



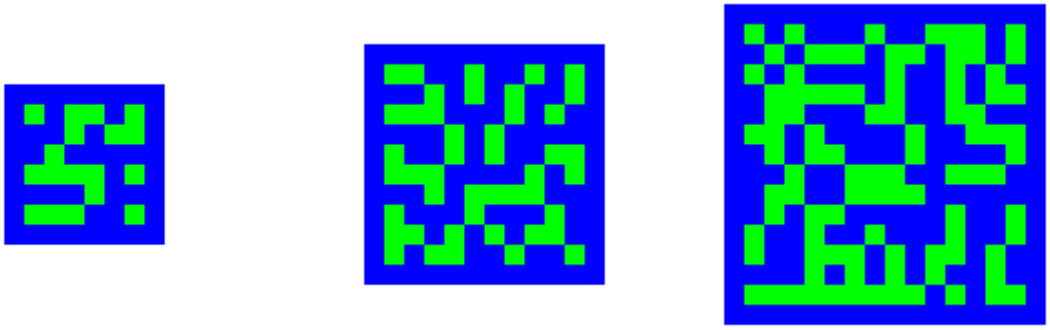
04_addIsland 2x2,3x3,4x4



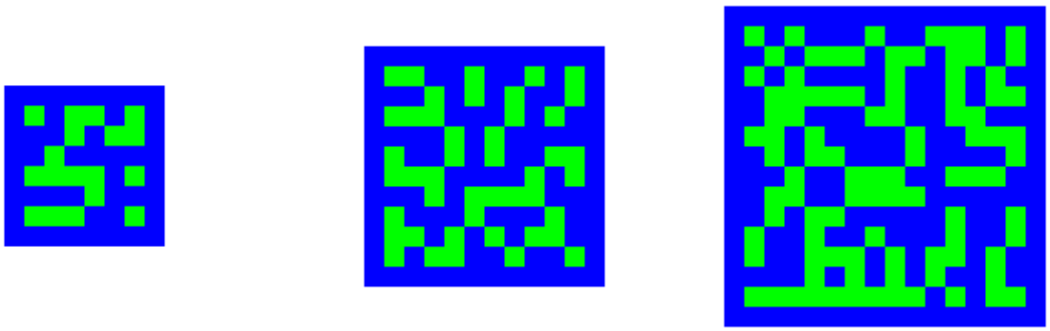
05_addIsland 2x2,3x3,4x4



06_addIsland 2x2,3x3,4x4

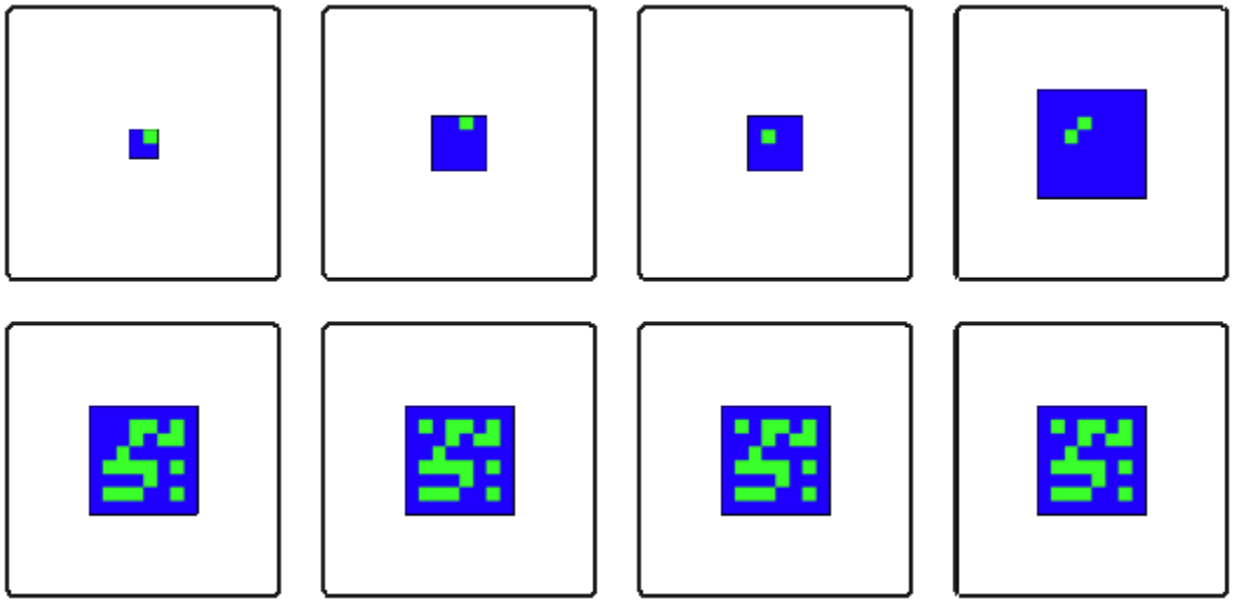


07_addIsland 2x2,3x3,4x4

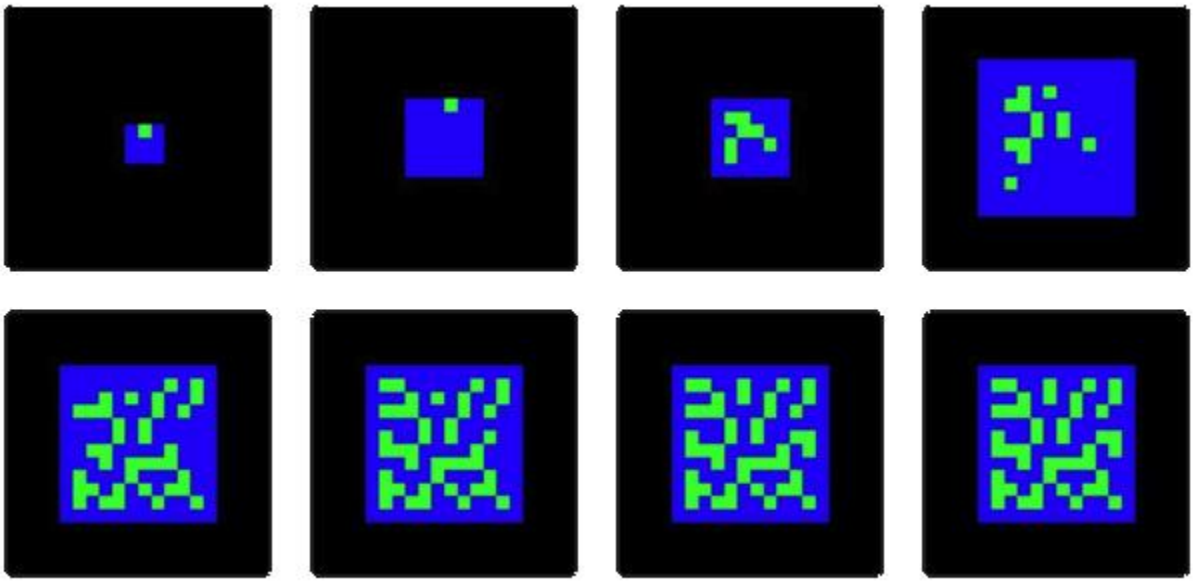


generateLandMass

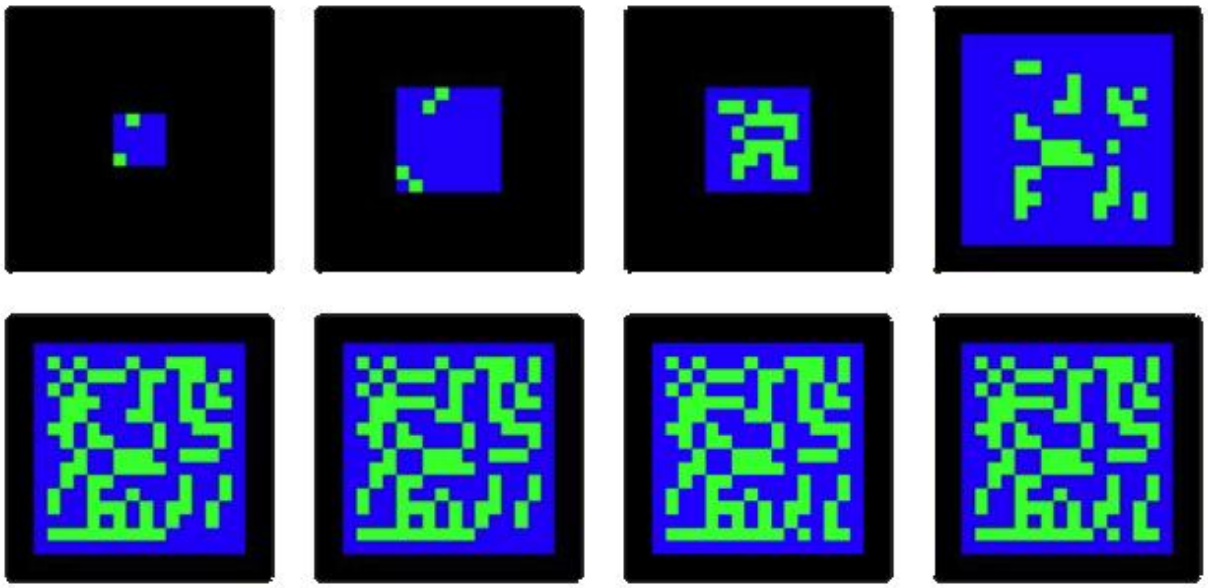
2x2



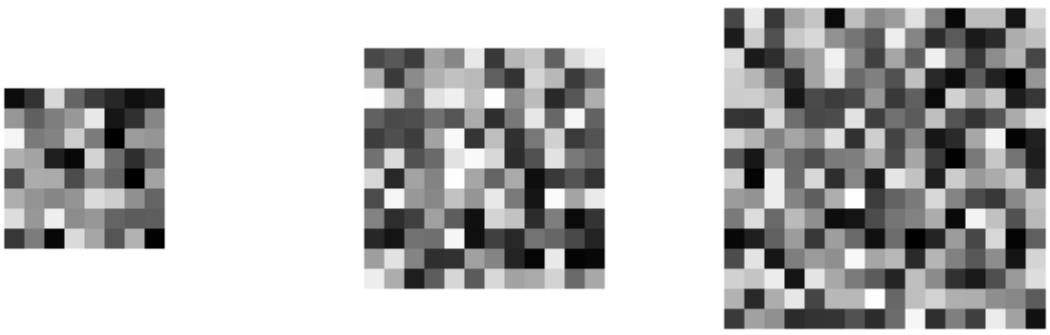
3x3



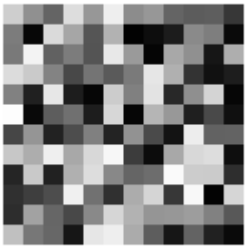
4x4

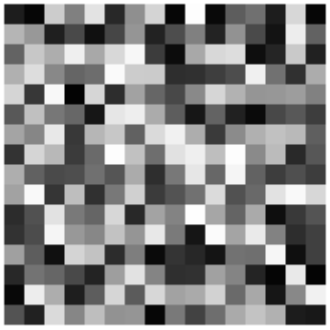


Rain Fall 2x2,3x3,4x4

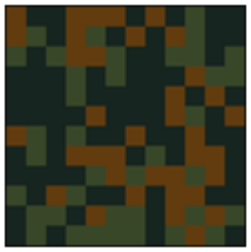


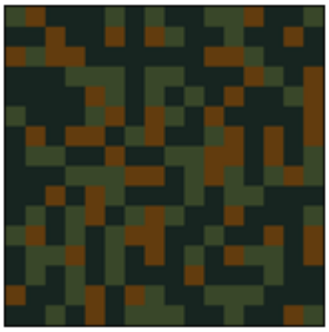
Temperatures 2x2,3x3,4x4





Biomes 2x2,3x3,4x4





Appendix B: Code

Java

Biome.java

```
public enum Biome {
    Forest, Plains, Desert, Void, RainForest, Savanna,
    SeasonalForest, Shrubland, Swamp, Taiga, Tundra;
}
```

Block.java

Block Instance variables

```
private int x, y, z;
private int chunkX, chunkY;
private int universalX, universalY, universalZ;
private BlockType type;
```

Block Constructor

A Block with a given x,y,z location relative to the parent Chunk. The parent Chunk is tracked using the Chunk's x and y location in the global map. The type is given at creation.

Params: x – x location relative to Chunk
y – y location relative to Chunk
z – z location relative to Chunk
chunkX – x location of Chunk relative to Map
chunkY – y location of Chunk relative to Map
type – type of Block

```
public Block(int x, int y, int z, int chunkX, int chunkY, BlockType type) {
    setX(x);
    setY(y);
    setZ(z);
    setChunkX(chunkX);
    setChunkY(chunkY);
    calculateUniversalLocation();
    setType(type);
}
```

calculateUniversalLocation

> Calculates the Universal location of this block in the map

```
private void calculateUniversalLocation() {
    universalX = x + (Chunk.WIDTH * chunkX);
    universalY = y + (Chunk.LENGTH * chunkY);
    universalZ = z;
}
```

BlockType.java

> The type of blocks possible

```
public enum BlockType {
    AIR, WATER, STONE, SAND, DIRT, GRASS
}
```

Chunk.java

Chunk Instance variables

```
public static final int WIDTH = 32;
public static final int LENGTH = 32;
public static final int HEIGHT = 100;

private int x, y;
private Biome biome;
private boolean land;
private long seed;
private Block[][][] blocks;
```

Chunk Constructor

> Creates a chunk of Biome biome with location x,y in the world. Sets land status and seed to passed in values
> Calls the proper fill method for the chunk's biome.
>
> Params: x
> y
> biome
> land
> seed

```
public Chunk(int x, int y, Biome biome, boolean land, long seed) {
    blocks = new Block[WIDTH][LENGTH][HEIGHT];
    setX(x);
    setY(y);
    setBiome(biome);
    this.land = land;
    this.seed = seed;
    fillChunks();

}
```

fillChunks

> Directs to the correct Fill method for the chunk's biome

```
private void fillChunks() {
    switch (biome) {
        case Forest:
            createForest();
            break;
        case Plains:
            createPlains();
            break;
        case Desert:
            createDesert();
            break;
        case Void:
            createVoid();
            break;
```

createForest

```
Fills chunk with a dirt and grass top
private void createForest() {
    //Loop through every block
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                //set default type to AIR
                BlockType type = BlockType.AIR;
                if (land) {
                    if (h >= 70) {
                        type = BlockType.AIR;
                    } else if (h == 69) {
                        Random rand = new Random(seed);
                        if (rand.nextInt(10) > 5) {
                            type = BlockType.DIRT;
                        } else {
                            type = BlockType.GRASS;
                        }
                    } else if (h >= 60) {
                        type = BlockType.DIRT;
                    } else {
                        type = BlockType.STONE;
                    }
                } else {...}
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);
            }
        }
    }
}
```

createPlains

> Fills the chunk with a majority of grass on top with dirt mixed in.

```
private void createPlains() {
    //Loop through every block
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                //set default type to AIR
                BlockType type = BlockType.AIR;
                if (land) {
                    if (h >= 70) {
                        type = BlockType.AIR;
                    } else if (h == 69) {
                        Random rand = new Random(seed);
                        if (rand.nextInt(10) > 3) {
                            type = BlockType.DIRT;
                        } else {
                            type = BlockType.GRASS;
                        }
                    } else if (h >= 60) {
                        type = BlockType.DIRT;
                    } else {
                        type = BlockType.STONE;
                    }
                } else {...}
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);
            }
        }
    }
}
```

createDesert

> Fills the chunk with sand on top

```
private void createDesert() {
    //Loop through every block
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                //set default type to AIR
                BlockType type = BlockType.AIR;
                if (land) {
                    if (h >= 70) {
                        type = BlockType.AIR;
                    } else if (h >= 60) {
                        type = BlockType.SAND;
                    } else {
                        type = BlockType.STONE;
                    }
                } else {...}
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);

            }
        }
    }
}
```

createVoid

> Fills the chunk with just air

```
private void createVoid() {
    for (int w = 0; w < blocks.length; w++) {
        for (int l = 0; l < blocks[0].length; l++) {
            for (int h = 0; h < blocks[0][0].length; h++) {
                blocks[w][l][h] = new Block(w, l, h, x, y, BlockType.AIR);
            }
        }
    }
}
```

createLogicalBlockDistribution

> Fills the chunk with a random but semi logical block distribution

```
public void createLogicalBlockDistribution() {
    for (int w = 0; w < WIDTH; w++) {
        for (int l = 0; l < LENGTH; l++) {
            for (int h = 0; h < HEIGHT; h++) {
                BlockType type = BlockType.AIR;
                if (h >= 70) {
                    type = BlockType.AIR;
                } else if (h >= 60) {
                    type = BlockType.WATER;
                } else if (h >= 40) {
                    type = BlockType.STONE;
                } else if (h >= 20) {
                    type = BlockType.DIRT;
                } else {
                    type = BlockType.SAND;
                }
                blocks[w][l][h] = new Block(w, l, h, this.x, this.y, type);

            }
        }
    }
}
```

printBlocks

```
public static void printBlocks(Block[][][] blocks) {
    //System.out.println(blocks.length);//width
    //System.out.println(blocks[0].length);//length
    //System.out.println(blocks[0][0].length);//height
    for (int h = blocks[0][0].length - 1; h >= 0; h--) {
        for (int w = 0; w < blocks.length; w++) {
            if (h == blocks[0][0].length - 1) {
                System.out.print(w + ":");
            } else {
                System.out.print("  ");
                if (w > 9) {
                    System.out.print(" ");
                }
            }
            for (int l = 0; l < blocks[0].length; l++) {
                System.out.print(blocks[w][l][h].getType().ordinal());

            }
            System.out.print(" ");
        }
        System.out.println();

    }
}
```

toString

Outputs the blocks inside a chunk by slicing into the chunk vertically. Displays a vertical slices left to right on the screen

Returns:

```
public String toString() {
    String out = "";
    for (int h = blocks[0][0].length - 1; h >= 0; h--) {
        for (int l = 0; l < blocks.length; l++) {
            if (h == blocks[0][0].length - 1) {
                out += (l + ":");
            } else {
                out += ("  ");
                if (l > 9) {
                    out += (" ");
                }
            }
            for (int w = 0; w < blocks[0].length; w++) {
                out += blocks[w][l][h].getType().ordinal();
            }
            out += (" ");
        }
        out += "\n";
    }
    return out;
}
```

Main.java

timeMapGeneration

```
public static Map timeMapGeneration() {
    long startTime = System.nanoTime();

    MapGenerator mapGen = new MapGenerator();
    Map test = mapGen.getMap();

    long endTime = System.nanoTime();
    long duration = (endTime - startTime) / 1000000; // convert to milliseconds
    System.out.println("Time taken: " + duration + "ms");
    return test;
}
```

timeWriteToFile

```
public static void timeWriteToFile(Map test) {
    Chunk[][] chunks = test.getChunks();
    String fileName;
    long startTime = System.nanoTime();
    try {
        for (int x = 0; x < chunks.length; x++) {
            for (int y = 0; y < chunks[0].length; y++) {
                fileName = "chunk" + x + "_" + y + ".json";
                MapGenerator.writeJsonToFile(chunks[x][y], fileName);
            }
        }
        System.out.println("JSON file written successfully");
    } catch (IOException e) {
        System.err.println("Error writing JSON file: " + e.getMessage());
    }
    long endTime = System.nanoTime();
    long duration = (endTime - startTime) / 1000000; // convert to milliseconds
    System.out.println("Time taken: " + duration + "ms");

}
```

Map.java

Map Instance Variables

```
public final int width;
public final int length;
private long seed;
private Random rand;
private Chunk[][] chunks;
```

Map Constructor

```
public Map(int width, int length, Biome[][] biomes, int[][] land, long seed) {
    this.width = width;
    this.length = length;
    chunks = new Chunk[width][length];

    this.chunks = chunks;
    this.seed = seed;
}
```

setChunks

```
public void setChunks(Chunk[][] chunks) {
    this.chunks = new Chunk[chunks.length][chunks[0].length];
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < length; y++) {
            this.chunks[x][y] = new Chunk(chunks[x][y]);
        }
    }
}
```

MapGenerator.java

MapGenerator Instance Variables

```
private int mapWidth;
private int mapLength;
private long seed;
private Random rand;

private double[][] temperatures;
private double[][] rainFall;

private Biome[][] biomes;

private boolean[][] landMass;

private Map map;
```

MapGenerator Constructor

```
public MapGenerator(int startWidth, int startLength, long seed) {
    this.mapWidth = startWidth;
    this.mapLength = startLength;
    this.seed = seed;
    rand = new Random(seed);

    generateLandMass();//creates final size

    temperatures = generateNoiseMap();
    rainFall = generateNoiseMap();
    generateBiomes();
    generateMap();

}
```

generateLandMass

> Creates an array of booleans indicating whether a chunk of the map will be ocean or land.

```
public void generateLandMass() {
    generateStartLandRandom();
    int count = 0;
    printLand(count++, "init");
    zoom();
    printLand(count++, "zoom");
    addIsland();
    printLand(count++, "addIsland");
    zoom();
    printLand(count++, "zoom");
    addIsland();
    printLand(count++, "addIsland");
    addIsland();
    printLand(count++, "addIsland");
    addIsland();
    printLand(count++, "addIsland");
    landMass = removeTooMuchOcean(landMass);
    addIsland();
    printLand(count++, "addIsland");
    printLandToFile();
}
```

generateStartLandRandom

```
Creates an initial landMass randomly
private void generateStartLandRandom() {
    //initialize landMass with starting map width and length
    landMass = new boolean[mapWidth][mapLength];
    //give each boolean in the array a value using a 1/10 chance of being land
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            int isLand = rand.nextInt(10);
            landMass[x][y] = isLand < 1;
        }
    }

}
```

printLand

```
private void printLand() {
    String out = "";
    //loop through the 2d array
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            if (landMass[x][y]) {
                out += 1 + " ";

            } else {
                out += 0 + " ";

            }
        }
        out += "\n";
    }
    System.out.println(out);

}
```

zoom

```
private void zoom() {
    //copy landmass into land
    boolean[][] land = new boolean[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            land[x][y] = landMass[x][y];
        }
    }
    //create new array 2x the size of old one
    int newMapWidth = mapWidth * 2;
    int newMapLength = mapLength * 2;
    landMass = new boolean[newMapWidth][newMapLength];
    //set every 4 tiles equal to one from the old array
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            boolean isLand = land[x][y];
            //adding randomness
            landMass[2 * x][2 * y] = isLand && rand.nextBoolean();
            landMass[(2 * x) + 1][2 * y] = isLand && rand.nextBoolean();
            landMass[2 * x][(2 * y) + 1] = isLand && rand.nextBoolean();
            landMass[(2 * x) + 1][(2 * y) + 1] = isLand && rand.nextBoolean();

        }
    }
    mapWidth = newMapWidth;
    mapLength = newMapLength;

}
```

addIsland

```
private void addIsland() {
    //create new land array to hold new map with more islands
    boolean[][] newLand = new boolean[mapWidth][mapLength];
    //loop through all current chunks except edges
    for (int x = 1; x < mapWidth - 1; x++) {
        for (int y = 1; y < mapLength - 1; y++) {
            //only adding land so need to check ocean(false)
            if (!landMass[x][y]) {
                int count = 0;//check number of water tiles surrounding current block
                //counts amount of water surrounding square
                for (int i = -1; i <= 1; i++) {
                    for (int j = -1; j <= 1; j++) {
                        if (!landMass[x + j][y + i]) {
                            count++;}}                    }
                //Change chances of an island forming
                if (count > 7) {
                    //if the surrounding 8 tiles are water
                    //50% chance of turning into water
                    if (rand.nextInt() % 2 == 0) {
                        newLand[x][y] = false;
                        //50% chance of turning into land
                    } else {
                        newLand[x][y] = true;}
                } else {
                    //if less than 7 out of 9 tiles are water keep this tile water
                    newLand[x][y] = false;}
                //if the tile is already land set new array to the same
            } else {
                newLand[x][y] = true;}}}
    landMass = newLand;}
```

printLandToFile

```
private void printLandToFile() {
    try {
        // create a new directory called "landmass"
        File directory = new File("landMass");
        if (!directory.exists()) {
            directory.mkdir();
        }
        // create a new file called "landMass.txt" in the "landmass" directory
        File file = new File(directory, "landMass.txt");
        FileWriter fw = new FileWriter(file);

        // write the contents of the landMass array to the file
        for (int y = 0; y < mapLength; y++) {
            for (int x = 0; x < mapWidth; x++) {
                if (landMass[x][y]) {
                    fw.write(1 + " ");

                } else {
                    fw.write(0 + " ");
                }
            }
            fw.write("\n");
        }
        fw.close();
    } catch (IOException e) {
        System.out.println("An error occurred while writing to the file.");
        e.printStackTrace();
    }
}
```

generateNoiseMap

```
public double[][] generateNoiseMap() {
    double[][] noiseMap = new double[mapWidth][mapLength];

    // Generate random values for the noise map
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            noiseMap[x][y] = rand.nextDouble(); // Generate random value between 0 and 1
        }
    }

    return noiseMap;
}
```

generateBiomes

```
public void generateBiomes() {
    printNoiseMap(temperatures, "Temperatures");
    printNoiseMap(rainFall, "RainFall");
    biomes = new Biome[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            double temp = temperatures[x][y];
            double rain = rainFall[x][y];
            biomes[x][y] = Biome.Plains;
            if (rain > 0.50) {
                biomes[x][y] = Biome.Forest;
            } else {
                if (temp > 0.5) {
                    biomes[x][y] = Biome.Desert;
                }
            }
        }
    }
    printBiomes();
}
```

printNoiseMap

```
private void printNoiseMap(double[][] noiseMap, String name) {
    String directoryName = "NoiseMaps";
    String fileName = String.format("%s.txt", name);
    String filePath = directoryName + "/" + fileName;
    File directory = new File(directoryName);
    if (!directory.exists()) {
        directory.mkdir();
    }
    try {
        FileWriter fileWriter = new FileWriter(filePath);
        // loop through the 2d array
        for (int x = 0; x < noiseMap.length; x++) {
            for (int y = 0; y < noiseMap[x].length; y++) {
                fileWriter.write(String.format("%f ", noiseMap[x][y]));
            }
            fileWriter.write("\n");
        }
        fileWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

generateMap

```
private void generateMap() {
    Chunk[][] chunks = new Chunk[mapWidth][mapLength];
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapLength; y++) {
            chunks[x][y] = new Chunk(x, y, biomes[x][y], landMass[x][y], seed);
        }
    }
    map = new Map(mapWidth, mapLength, seed, chunks);
}
```

Python

Getting file data

```
# Prompt the user to select a directory
def get_directory():
    root = tk.Tk()
    root.withdraw()
    return filedialog.askdirectory()

# Get all text files in the directory
def get_text_files(directory):
    return [file_name for file_name in os.listdir(directory) if file_name.endswith('.txt')]

# Load the data from the file
def load_data(file_path):
    with open(file_path, 'r') as f:
        data = [[int(x) for x in line.split()] for line in f]
    return data
```

Displaying the file data

createLandMassImg.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                square_color = GREEN if data[data_row][data_col] else BLUE
            else:
                square_color = BLACK
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

createGrayScaleImages.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                #print('data'  + str(data[data_row][data_col]) +' '+str(data_row)  + ' '  + str(data_col) )
                square_color = (data[data_row][data_col]*255,data[data_row][data_col]*255,data[data_row][data_col]*255)
            else:
                square_color = BLACK
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

createBiomeImg.py

```
# Draw the squares for a given data and position
def draw_squares(data, screen, start_row, start_col, max_rows, max_cols, square_size):
    for row in range(max_rows):
        for col in range(max_cols):
            square_color = BLACK
            data_row = row + ((len(data) - max_rows) // 2)
            data_col = col + ((len(data[0]) - max_cols) // 2)
            if 0 <= data_row < len(data) and 0 <= data_col < len(data[0]):
                if data[data_row][data_col] == 0:
                    square_color = FOREST
                elif data[data_row][data_col] == 1:
                    square_color = PLAINS
                elif data[data_row][data_col] == 2:
                    square_color = DESERT
            square_rect = pygame.Rect(start_col + col * square_size, start_row + row * square_size, square_size, square_size)
            pygame.draw.rect(screen, square_color, square_rect)
```

Saving the data to an Image

```
# Save the image as a PNG file
def save_image(screen, directory, file_name):
    png_name = os.path.splitext(file_name)[0] + '.png'
    pygame.image.save(screen, os.path.join(directory, png_name))
```