

Conor Creagh

Final Report

An investigation and implementation of procedural map generation techniques in games.

Student number: 13454222

**Table of contents**

1. Introduction and literature review
2. Technical Review
3. Code and process dissection
4. Observations
5. Conclusions

**CHAPTER 1** **INTRODUCTION**

Brief Description:

This project will aim to highlight and explain the main methodologies in the area of procedural map generation in game design while also producing a working implementation of a combination of some of these techniques. This project entails a mixture of practical work and research. The first few months being spent on research into techniques used by professionals in the field and the theory behind these techniques. After this research is completed the project will then shift into the practical aspect, which will deal with implementing some of the techniques researched in the previous stage.

What is procedural generation?

Procedural generation is process of automatic generation that follows a set process and rules. This technique has a wide range of applications from video games to the cinema industry.

**Needs expanding**

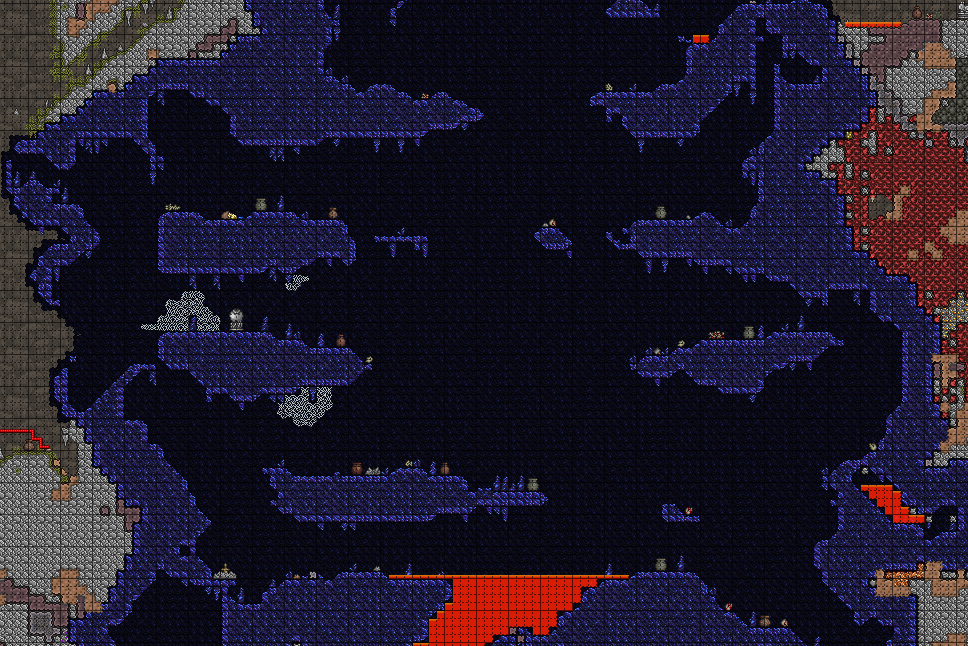
Procedural generations place in the gaming industry:

Overview:

In modern times the want/need for a game to have a map that has been procedurally generated has increased tenfold. In many ways this is due to the smash hit game Minecraft created and developed by Markus Persson and later Mojang. This game propelled the idea of a randomly procedurally generated map into the media spotlight, with any new open world sandbox game being compared to Minecraft. Of course Minecraft wasn’t the first game to employ procedural generation, procedural generation has been about since the advent of gaming, from games such as dwarf fortress to the Age of empires series.

It now seems nearly essential that a new game have some form of procedural generation, be it the whole world or just having certain areas such as caves procedurally generated.

Procedural generation is very popular in indie game development, again Minecraft comes to mind. Although it can be easy to think of only 3D procedural generation perhaps the most elegant form of procedural generation comes in the 2D implementations. This can be seen in **Terraria developed by ReLogic,** this game provided an insight into 2D complete world generation, with algorithms being executed to form subterranean constructs and above ground constructs, something which only came along to Minecraft’s 3D generation much later in the game.

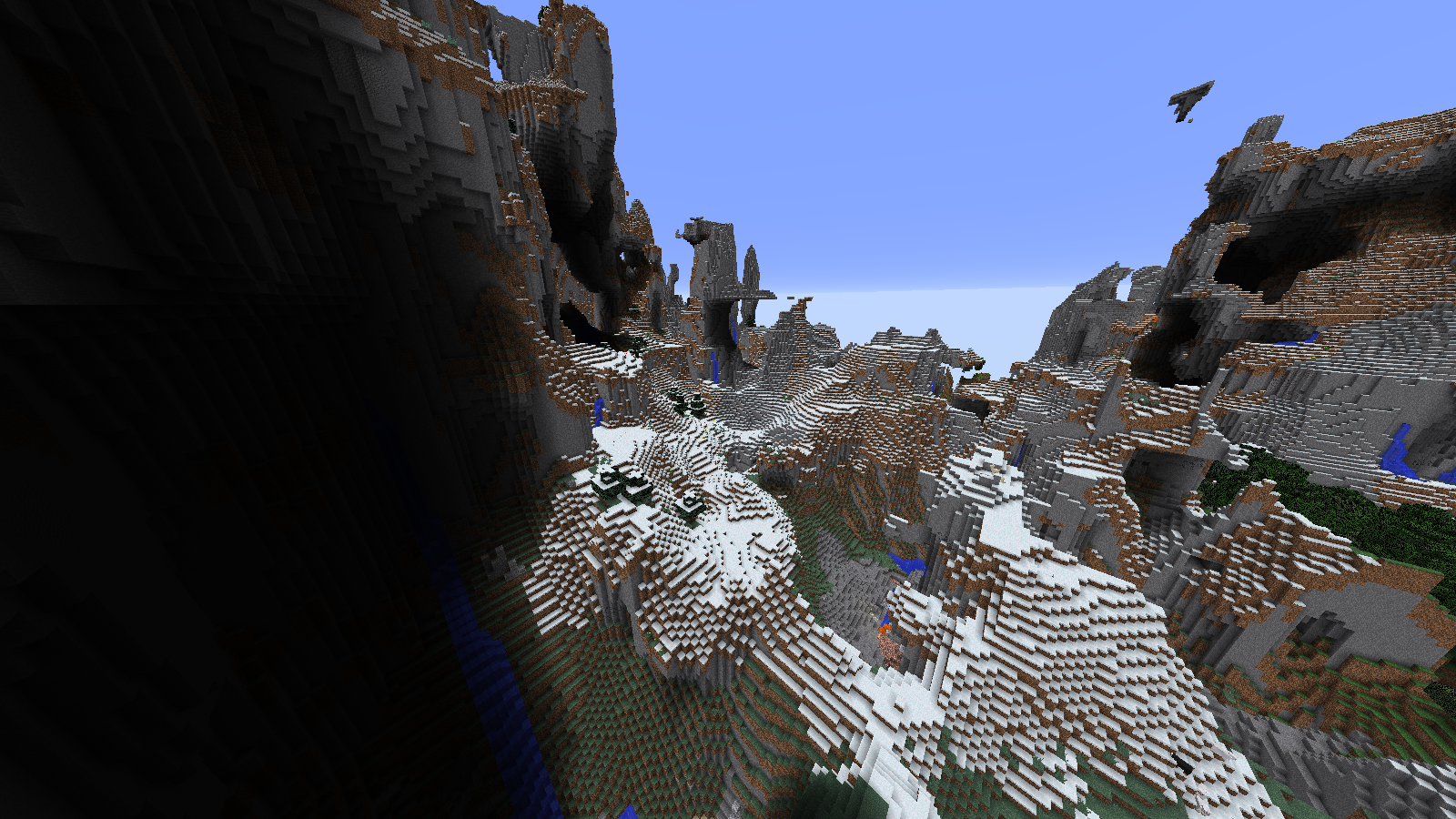


The two images above demonstrate the procedural generation in Terraria, on the left is a cave that was generated using the terraria engine, and to the right is an example of a structure that can be generated in the engine. These structures come in many different sizes and styles and can be found in all areas of the map, be it in the sky, as shown above, or deep in a cave.

Minecraft:

Upon researching the techniques used in the PMG in Minecraft I discovered a blog entry by the creator Notch going into some detail on his algorithm for Minecraft. Although he does not reveal the exact algorithm he used to create the terrain, he gives us an insight into the techniques used and his thinking behind the initial stages.

Initially Notch states that he used 2D Perlin noise to generate height maps for his terrain, similar to what I hope to accomplish in my implementation. However he does not simply use one Noise value for his terrain he uses a couple, in order to simulate other aspects which he would like to take into account in his terrain generation, such as roughness and detail. He then combined these separate values into a mathematical formula in order to generate the height. Although he later changed this system to one based of 3D Perlin noise as he found the previous to be “rather dull”, but did not go into as much detail on this algorithm.



The above image shows a portion of the terrain generation in Minecraft taken from the latest release of the game. Notch however does not still work on the game, but I am sure that elements if not the entirety of his generation algorithm comes into play here. In the left part of the image we see an overhang, which he alludes to in his blog as being one of the reasons he switch to 3D Perlin noise.

I chose this image due to its representation of Noise being used to generate a 3D landscape, this world being generated using the Amplified world type in order to demonstrate this, albeit exaggerated due to this.

Procedural Generation an academic view:

From the outside the concept of procedural map generation (PMG) may seem rather trivial to some but a closer inspection will reveal a complex host of algorithms and techniques. At the forefront of the academic side of PMG is noise generation, noise generation is the process of applying pseudorandom fluctuation in values that will give an organic result in pattern and was originally developed by Ken Perlin in order to improve on the “machine look” of computer graphics of the time. I will take a deeper look into noise in the next chapter of this report.

At its base PMG in a 2D environment is manipulation of a 2D array and requires clever algorithms for searching and the manipulation of values and or tiles in this 2D array. The end aim of PMG is to have a “natural” looking map with interesting features

Procedural generation does not singly apply to video games it is widely used in the films industry to produce large CGI visuals rapidly and in music with “Generative music”. **Needs expanding**

**CHAPTER 2** **TECHNICAL REVIEW**

Techniques in Procedural map Generation:

Noise generation:

In 1983 Ken Perlin developed Perlin noise, he developed this algorithm in order to more realistically represent graphics in computers. Although not his intention this would create a new precedent in the area of map generation in computer games.

To elaborate on this take an n-dimensional map represented as an n-dimensional array, each entry in the array being a “height” value between 0 and 1 (can be different range depending on implementation) in the world, while running a Perlin noise algorithm on this array each position will be influenced but the positions around it. This leads to a more natural shift in values, allowing us to model regions in a map and simulate the natural flow of the land.

In essence applying Perlin noise to a 2D array will give supply us with a pseudorandom array of integers between 0 and 1, which each value naturally leading into the next, to give us the skeleton of a natural looking height map.

Perlin further went on to improve upon his initial creation with the advent of Simplex noise, Simplex noise however being Patented would lead to the creation of a open source implementation of the revised algorithm, open simplex noise. The main improvements of Simplex over base Perlin are the improvements in computational complexity in higher dimensions, **with classic Perlin noise having Big-Oh of 2^n while simplex noise have Big-Oh of n^2.**

Cellular automata:

Cellular automata is a technique used in PMG for creating caves and cave like structures. The basic idea behind cellular automata is comparing a cell to the cells around it.

As an example take a 2D grid of Boolean values, now randomly seed this grid with the values. The algorithm will go through each value in this 2D array and apply a pre-defined set of rules to the cell. An example could be if more than 5 of the cells 8 neighbours are different then change the value of this cell to that of the 5, which in this case will be simply the inverse of the cell. It is also important to note that this procedure should always read from one array and write to another in order to avoid previously computed cells to effect the current cell.

This process will give an overall smoothing effect to the grid of values, I will expand on cellular automata later in this report as I have used the core concept in order to create a smoothing function for my implementation.

Technologies used:

Git:

Git is a service that provided tools to aid in the process of incremental development. Git and Github were introduced to me in a module I had done in my first semester of final year so was in the forefront of my mind, and so was only logical to keep track of my project using Git.

Throughout my project my use of Git was not strictly enforced as there are wide gaps between some of the work done on the commits, but the feature itself helped me analyze the overall progression of the code for my project from start to finish. The project can be found on my Github account – CreaghSTC (<https://github.com/creaghstc/Final-year-project>).

(note 2 commits are from a different account due to an error that occurred when I committed on a college computer, which was signed in on their account.)

JavaScript:

JavaScript is a high-level interpreted programming language. JavaScript is used in most if not all websites and is one of the core web technologies, because of this all top end web browsers have detailed debugger and console capabilities for JavaScript.

The driving force behind my decision to use JavaScript for my implementation was my decision to use PixiJS. PixiJS being a JavaScript graphics library. Throughout my implementation the browser I used in conjunction with JavaScript and PixiJS was Mozilla Firefox. I initially used google chrome, but due to security restrictions placed by Chrome PixiJS would not function correctly, so in response I switched to using Firefox as it does not possess any of these restriction. **To be reworked/extended**

PixiJS an Overview:

PixiJS is an open source JavaScript graphics library developed by goodboy Digital ™ (<http://www.goodboydigital.com/>) and is the graphics library that I chose to use for my implementation. PixiJS is fast and easy to learn with plenty of tutorials and examples supplied on their website. There is also a GitHub page for the project that allows the community to contribute to the project.

For my project no intense rendering is needed, although PixiJS is more than capable of performing high level rendering, the height of the rendering capabilities needed will be to draw tiles to a specific coordinate of the screen.

PixiJS was recommended to me by my supervisor Dr. Sam Redfern, as he has previously used it to develop a game in conjunction with nodeJS, after much deliberation between using Microsoft’s XNA or PixiJS I decided to go with the latter, as my supervisor is well versed in the library and would better be able to answer any queries I had. Another reason I chose PixiJS over XNA was that PixiJS is still being actively developed while XNA has been retired by Microsoft.

Features of PixiJS:

PixiJS is hailed as the fasted 2D graphics renderer available for browsers, it has many key features that allows it to claim this status. First of which being its speed and the fact that PixiJS automatically will use the WebGL renderer if it is available and fall back on the Canvas renderer if WebGL is not available. This allows PixiJS to easily run on older machines that may not have WebGL without a hitch as this is all done automatically in the background. PixiJS as has multiplatform support that covers all the essential platforms.

One of the most attractive feature of PixiJS is it’s easy to learn API. In my experience with PixiJS, although I did not need to go deep into more complicated functions of the API I found it to be quite intuitive and straight forward.

One of PixiJS’ best sometimes overlooked feature of PixiJS is the fact that PixiJS is free and open source with a highly active community of developers on their GitHub page. This feature also allowed me to inspect the source code of PixiJS and gain a more intimate knowledge of how some of the API works.

A lot of the features highlighted by the community and the developers however do not influence my implementation as I am simply using PixiJS to render my map, a few of these features listed on their website include: multi-touch interactivity, WebGL filters, sprite sheet support and many more.

P5.js:

P5.js is a library for JavaScript which I discovered when I was researching Perlin noise and its implementations in PMG, what attracted me to this library was its implementation of Perlin noise and the Built in functions which allowed me to manipulate the octaves and provide a seed for the noise function.

**CHAPTER 3 CODE AND PROCESS DISECTION**

Initial stages:

My initial approach to the implementation of my project after I made the decisions on what technologies that I would use was to familiarize myself with said technologies. As I had never used JavaScript before I explored various online tutorials and read up on the documentation of the language.

In particular I done research into how JavaScript uses arrays as I knew a large portion of my implementation would be the manipulation of arrays. Another aspect of JavaScript that I inspected was the idea of Object orientated techniques in the language as it had occurred to me that I could create several objects to manipulate alongside the array. Such objects being perhaps a tile object and a map object.

From research into JavaScript I could clearly make out where the language had being influenced by other languages such as Java and python which I had studied before. I could see it had taken the object oriented java and combined it with the simple and straight forward syntax of python, which greatly increased my understanding of the language. In particular I found the tutorials on the JavaScript website itself very good and helped me progress my knowledge swiftly.

Once I had a grasp on JavaScript I decided it was time to learn the basics of PixiJS. Again the PixiJS website had plenty of informative examples which I tried to recreate on my own to help me understand the subtleties of the API. For my implementation I do not have to utilize many of PixiJS’ capabilities as the height of my rendering is drawing various tiles to the screen, although I do plan to continue using PixiJS for personal projects in the future, so there was little harm in reading further into the material.

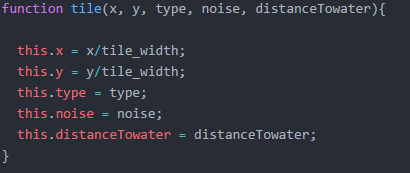
Along with its abundant host of examples PixiJS supplies us with full documentation for the API, which I frequently used in order to quickly check parameters for functions or if there was a simpler was for me to achieve my goals.

Discussion of Final code:

Basics needed to understand code snippets:

This section will provide the basic knowledge needed of my implementation to understand the more complex code snippets in the sections to come.

First we have the tile object, this is a base object that the map is built from.

Shown on the left is the tile object and its attributes. The constructor for a tile takes in an x value, y value, type and its noise value (height). These x and y values are the set to the parameter divided by the width/height of a tile….**going to change this**….. The other parameters will then be simply set. The tile object is the main focus of many of the algorithms that follow as it is the key part of the map and is what needs to be manipulated. The purpose of the type attribute is to allow me to determine what texture the tile will be drawn with and to be able to manipulate this without having to redraw the tile multiple times.

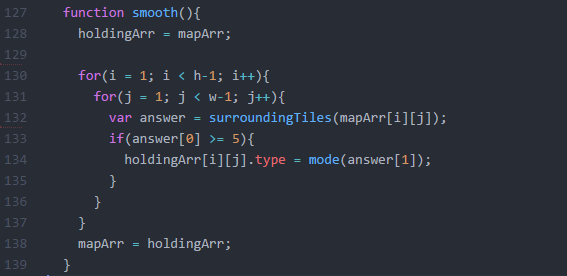
Another key bit of information needed is that the tile objects are stored in a 2D array that represents the map itself.

Next is the surroundingTiles() function which is a function that takes in a tile object as a parameter and then returns and array of useful information, 1. Number of different tiles 2. An array of types of surrounding tiles 3. An array of the actual surrounding tiles 4. **Distance to water? To be finished**

Smoothing:

As soon as I got the base map created and rendered my first objective was to manipulate the map in order to remove some of the outlying “noisy” tiles, as these tiles would interfere with any of the future tasks I wanted to perform on the map. So in a sense the following is a representation of how I went about “cleaning up” my map for further use.

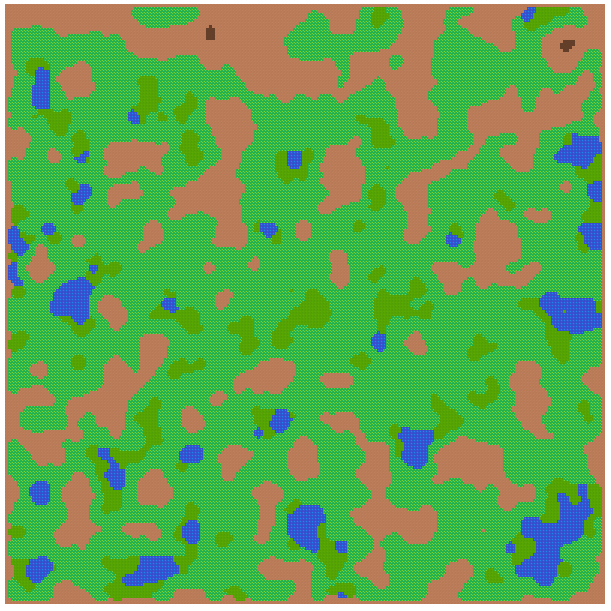
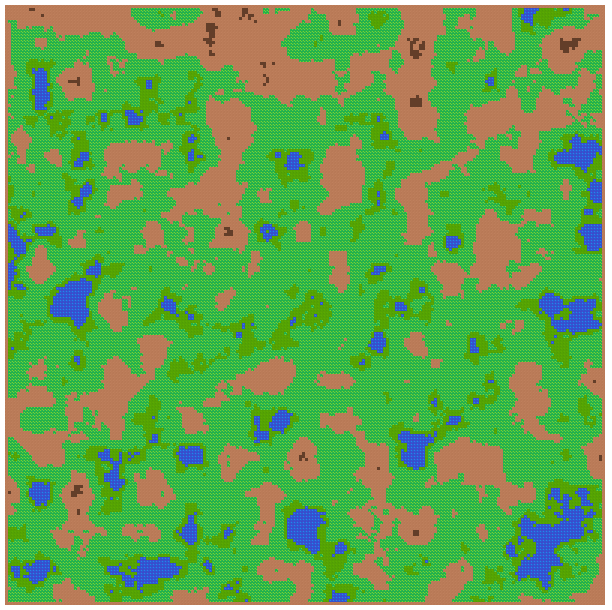
As discussed before I took the core idea of cellular automata and applied it to my map in order to remove some of the visual noise produced by the noise function. The main thing that I had to remember was to read from array and write to another, and then to set the rules of what would induce the smoothing effect.



**Should I comment out this code?**

The above code is my smoothing function. The basic idea of the algorithm is to loop through all of the tiles in the map array, mapArr, and for each tile to run the surroundingTiles() function. The surroundingTiles() function returns an array of values, only two of which are used here, the rest will be discussed when relevant. The first element of the array produced by surroundingTiles() is a counter which represents the how many of the surrounding tiles are of a different type. Using this information the algorithm will then either change the tile if it meets the correct requirements or simply move unto the next tile in the map. The requirements here being if 5 or more of the surrounding tiles are of different type, if this is the case then the corresponding value in the holding array is changed to the most common tile surrounding the current tile, this is retrieved by getting the mode of the second element of the surroundingTiles() function which is simply an array of the types of each surrounding tile. When all tiles have been passed through the algorithm mapArr is then set to holdingArr.

Upon inspection the nested for loops do not actually loop through all the tiles, this is due to design, the map itself has edges which are all mountains, if the edges were to be smoothed this would break the majority of the map boundary. So for this reason the smooth() function does not operate on these tiles.



Above is a portion of the map before and after smoothing. Notice how most if not all of the high level noise has either been removed or expanded. For this result the smooth() function was called four times, from experimentation I found four calls to be the optimal, being called once still left noise while being called more than four times stopped having an effect on the map, as it had reached an equilibrium of sorts.

Placing sand:

In order to add more detail to the map and to make it more realistic I decided to produce and algorithm that would go through the map and apply sand to the areas which sand would naturally form. I decided these areas would be around every current water source directly after the smoothing phase.

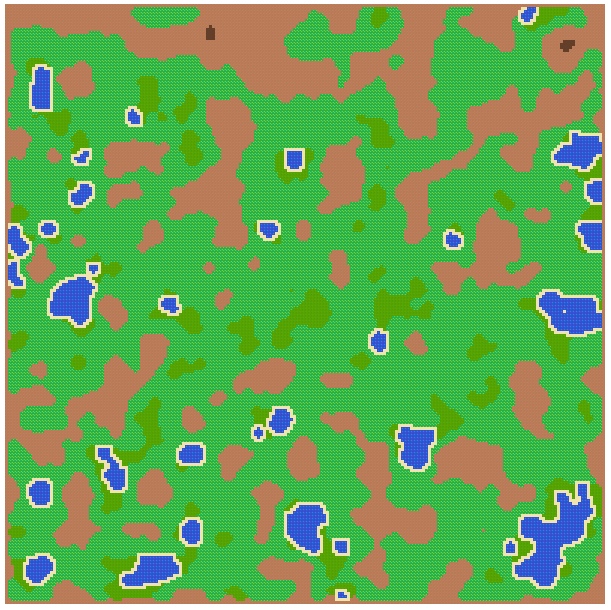


**Distance to water line, may be gone later**

The idea of the above algorithm is to basically trace every source of water with sand, also note again that this algorithm reads and writes to two separate arrays. The algorithm is as follows, again running through each of the tiles in the map excluding the edges, we initialize two variables at each tile, waterPresent and answer, waterPresent simply being a checker whether the current has an adjacent water tile or not and answer again being the array given by the surroundingTiles() function.

The first check the tile has to go through is to see if the tile has any adjacent water, the answer to this question will be stored in the waterPresent variable. The algorithm loops through the second element of answer which is an array of the surrounding tile types, if anyone of these entries is “water” then there is water present and the check is set to true, and the algorithm proceeds.

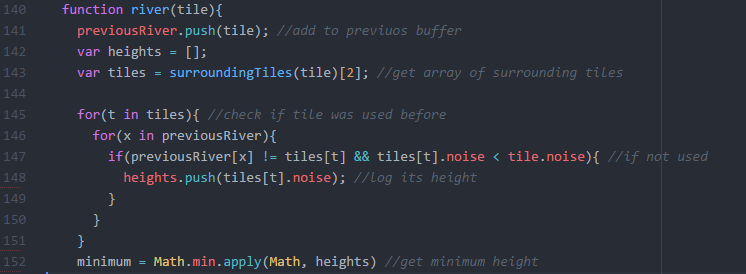
The next check that a tile has to go through is, is the tike itself water? If so then the tile will not be converted, if it is not water and there is water present then the tile will be converted to sand using the holdingArr as a proxy to the mapArr. At the end of the algorithm the mapArr is set to the holdingArr.

 To the left is the same map shown in the previous section with the sand() function applied. As you can see the function accurately traces all bodies of water with sand and emphasizes the water in the map, while also making the overall map more natural looking and pleasing to look at.

Generating and placing Rivers:

The placement of rivers is a feature I included in my implantation in order to add complexity to the tool. I also took an interest in the idea of simulating natural looking rivers. In order to do this I took what a river is at its base level, water flowing from a high point to a low point. To implement this I used a recursive algorithm which basically “snakes” a river down a path of tiles each being lower than the next and finally stopping at the lowest tile or when it hits water.

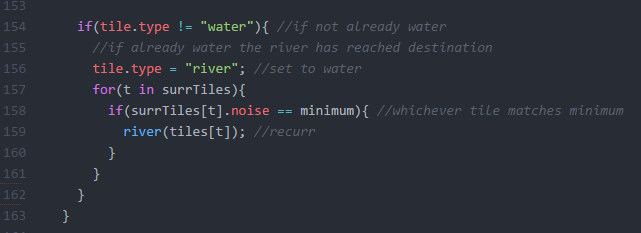
**Needs new code screen**



The above snippet of code is the first step in my algorithm, which is used to determine the minimum height in the tiles surrounding the current river tile.

The first step is to add the current tile to an array of previous river tiles, so as we can eliminate this tile as being a candidate for the next tile later. The algorithm then initializes the two arrays of heights and surrTiles, heights being all the heights of candidate next river tiles.

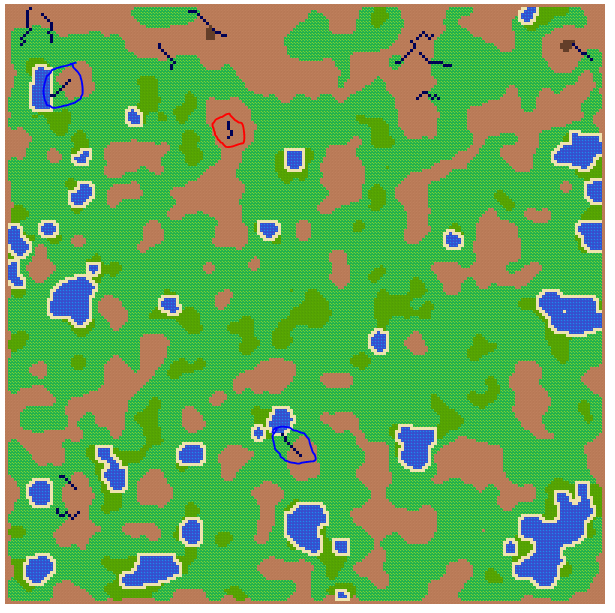
Next we loop through surrTiles and all of the previous river tiles and perform a check, the check itself is to ensure that 1: the surrounding tile has not been used before ( != ) and 2: that the surrounding tile has a noise/height value that is lower than the current tile. If a tile passes both these checks then it will be added to the heights array, We then set minimum to the lowest height in that array.



Having obtained the local minimum from the set of surrounding tiles we enter the next stage of the algorithm. We perform a check that the current tile is not a water tile, as if it were to be a water tile we would end the river here, and so the recursion will end. We then set the type of the current tile to “river” as it has passed all of the test and matches the specifications to become a river tile.

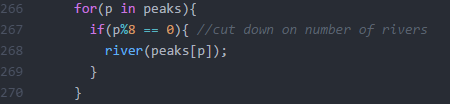
Once we have deemed the current tile to be of type river our next task is to continue the generation of this river. To do this the algorithm loops over the surrounding tiles again and checks if any of the tile noise values match the local minimum, if a tile does then the algorithm will then call the function on this tile, if not then the river ends at the current tiles location.

During the development of this algorithm I had expected the majority of the rivers to “flow” into water areas, this was not the case. An acceptable number of the rivers do end in the water bodies and very few of those that don’t still behave pleasingly. As shown in the image below.

Here I have circled two rivers in blue, these are the rivers that acted “perfectly” and flowed from a local highpoint in the mountains into a lowpoint in the water.

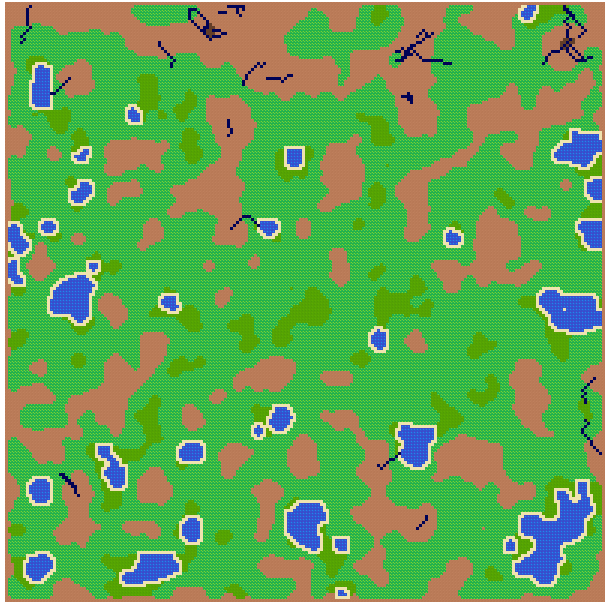
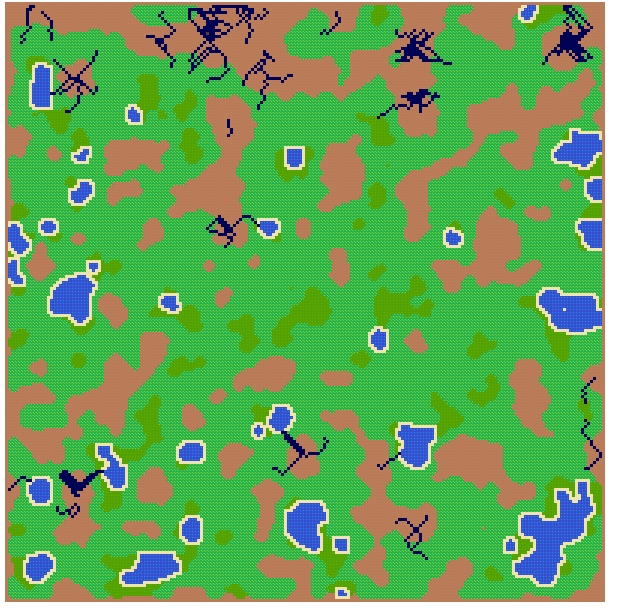
The river circled in red however and ones like it in other seeds drew my attention to a problem that can manifest using pure noise as a height map. The crux of the problem is the there is no path of continually lower heights that lead out of the mountain range. Although not entirely a failure when looking at this from a point of view of being natural looking, as this can be viewed as the formation of lakes in mountains that have no draining river, which there are numerous examples of completely isolated mountain lakes in nature.

Separate from the rivers algorithm is the method in which I select where to start the rivers, basically the map has two types of mountain, a normal; mountain and a peak. I use these peaks as the seed for my rivers. However if I use all of the peaks as seeds then the algorithm does not perform as desired at all and so I was forced to come up with a solution. My solution involves taking every 8th peak tile and using that as a seed for a river. To do this I simply loop over all of the peak tiles and call river() when the loop number modulo 8 is equal to 0.



**This my change**

In order to explain why this restraint is needed I will modify the above block of code twice times, with every peak being used as a seed and every fourth peak.



Here we see on the left tile every peak being used as a river, and on the right every fourth. The left is clearly unacceptable as it turns every mountain range into a mess of rivers, while the image on the right is acceptable but still leaves some clutter around some of the peaks, through experimentation on the different amounts of rivers being generated I found 8 to be the best at providing satisfying results.