Conor Creagh

Final Report

An investigation and implementation of

procedural map generation techniques.

Student number: 13454222

Supervisor: Dr. Sam Redfern

Table of contents

CHAPTER 1 INTRODUCTION

Brief Description:

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

What is procedural generation?

Procedural generation is automatic generation that follows a set of guidelines and rules. The technique has a wide range of applications from video games to the cinema industry. Many aspects of digital technology and software incorporate procedural generation without users even knowing. The obvious applications for procedural generation are in games, this aspect is talked about in detail throughout the entirety of this report.

The act of procedural generation was envisioned with the aim to cut down on resource use in real time and this is exactly what it does, procedural generation also gives variety and spontaneity to whatever industry it is applied to, the degree of this depending on how rigid the applied rules are to the generator itself. The next stages of this report will further elaborate on the aspects of procedural generation and provide insights into the advantages and disadvantages which are linked to the topic.

Procedural generations place in the gaming industry

Overview:

In modern times the want/need for a game to have content or 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. Minecraft propelled the idea of a 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 games 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 its 2D implementations. This can be seen in Terraria developed by Re-Logic**,** 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.

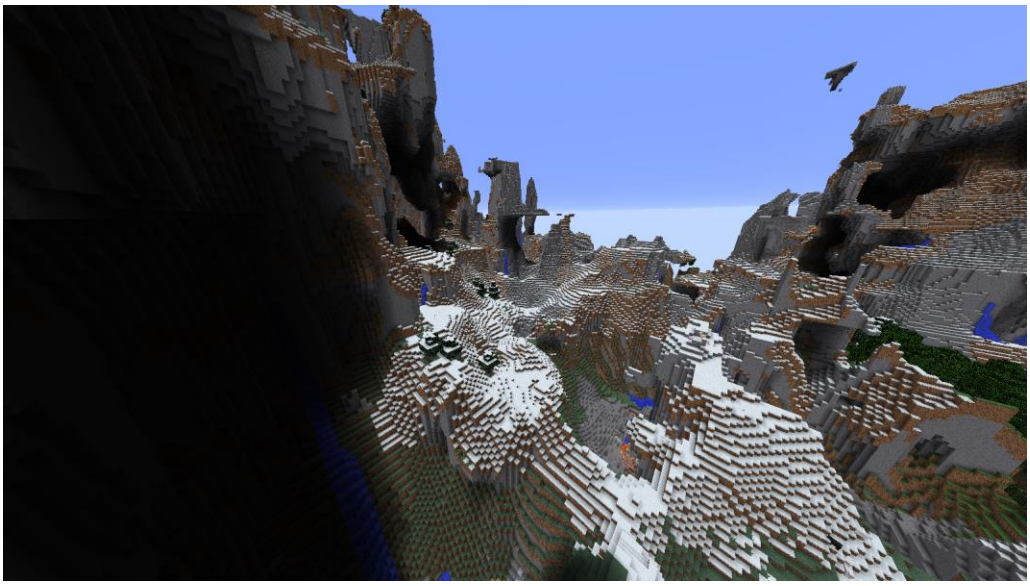
The images above demonstrate procedural generation in Terraria, on the left is a cave that was generated using Terraria's engine, and to the right is an example of a structure that can be generated in Terraria. These structures come in many different sizes and styles and can be found in all areas of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

Through my research I could find no concrete answer on how Terraria generates these cave systems, but an educated guess hints that Terraria uses cellular automata to aid in the construction of these caves. I came to this conclusion as there is strong evidence of the techniques being used in observing the overall shape of the caves, there is little to no noise and the walls of the cave appear to be uniform for the most part and stick to an overall structure, the technique of cellular automata will be discussed in the next chapter of this report in detail.

Minecraft:

Upon researching the techniques used in the procedural map generation that appears in Minecraft I discovered a blog entry[1] by the creator Notch going into some detail on his algorithms for Minecraft. Although he does not reveal the exact algorithm he used to generate 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 of values in tandem 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 much detail on this new 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 why he chose to 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 using this world type.

Procedural content:

The idea of procedural generation in games is not limited to map generation. Many games simply have procedural content generation, which can range from randomly generating the contents of a chest to procedurally generating entire enemies.

One example of content generation is in the borderlands series, which has its self-proclaimed "87 bazillion" guns. Every gun in the game is generated uniquely, with attributes such as style, bullet type, effect, modifiers and others. According to Gearbox software as of release there are over 17,750,000 different combinations of guns in the game, and guns are not the only thing generated in the game, shields, grenades and in later games in the series some enemies are procedurally generated. Borderlands in a prime example of procedural generation being used outside the area of map and terrain, in fact all maps in borderlands are pre made static areas.

Another quite controversial game on the topic of procedural generation is No Mans Sky which will be discussed in the conclusion to this report, both being a technical feat and a business blunder.

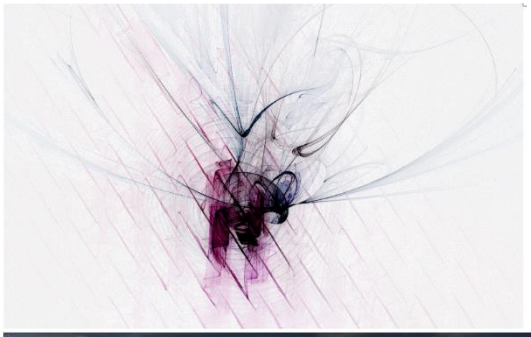
Procedural Generation an academic view:

From the outside the concept of procedural map generation may seem rather trivial to some but on closer inspection will reveal a complex host of algorithms and techniques. At the forefront of the academic side of procedural map generation 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 procedural map generation in a 2D environment involves the manipulation of a 2D array and requires clever algorithms for searching through and the manipulation of objects in this 2D array. The end aim of procedural map generation 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. One such example of the use of procedural generation in films would be the Lord of the Rings trilogy in which director Peter Jackson employs procedural generation techniques in order to create the massive battles and visuals. Procedural generation is also used for more common media such as TV advertisements.

In some cases procedural generation is used to create art using fractals and other mathematical constructs which can leave rather stunning visuals and artwork.



Here we have a piece of fractal art that was created using software called Apophysis and the art was created by username Arfink on http://chipmusic.org/arfink

Here we have a screen capture from a battle in Lord of the Rings: Return of the king, in which procedural generation was used to create both the charging and defending armies.

CHAPTER 2 TECHNICAL REVIEW

Techniques in Procedural map Generation

Noise generation:

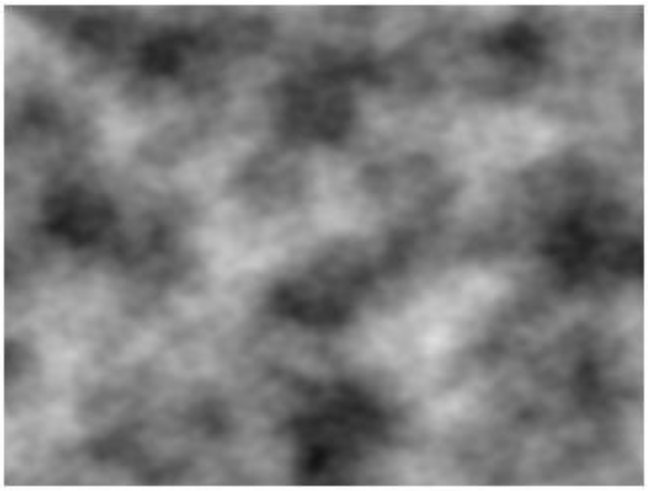
In 1983 Ken Perlin developed Perlin noise, he developed this algorithm in order to more realistically represent graphics in computers. Although not his intention this would create a new precedent in the area of map generation in computer games.

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 (This can be a different range depending on the implementation) in the world, while running Perlin noise on this array each position will be influenced by 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 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 an open source implementation of the revised algorithm, open simplex noise. The main improvements of Simplex over base Perlin are its computational complexity in higher dimensions, with classic Perlin noise having Big-Oh of 2^n while simplex noise have Big-Oh of n^2.

The JavaScript library P5.js allows me to demonstrate the core concepts of noise using its drawing capabilities to show us a grayscale representation of Perlin noise, with each pixel being affected by a noise value.



Here we can clearly see what Perlin noise entails, the smooth transition of the pixels from grey to black and back.

In the context of video game map generation it is easy to see how we could implement this as a height map, the darker the pixel the lower the height value, the lighter the higher. In my implementation I will use this concept to from a 2D map using these height values. In practice Perlin noise is not limited to 2D maps or 2 dimensions.

(http://genekogan.com/code/p5js-perlin-noise/)

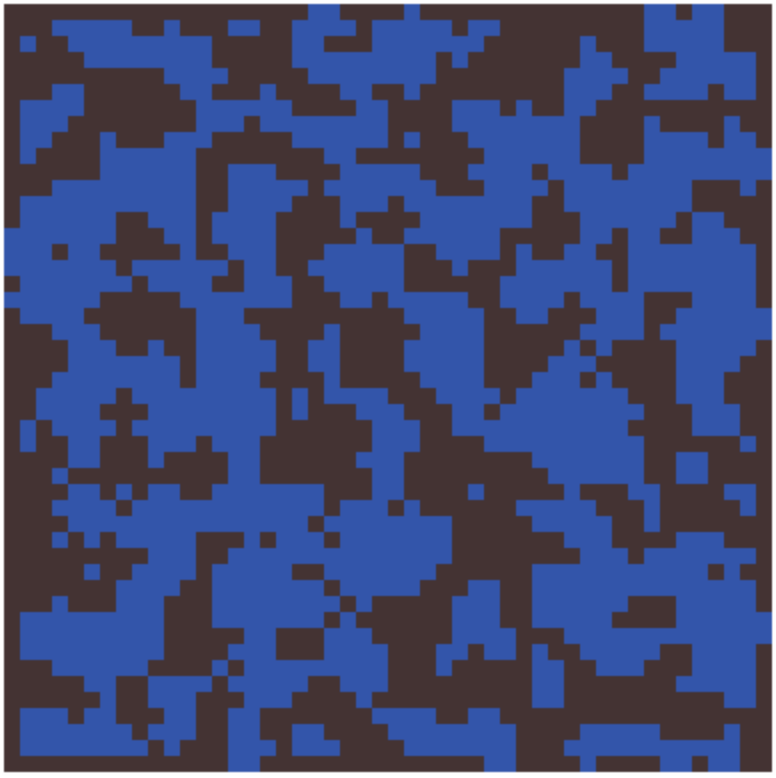
Above is an example of Perlin noise being used to generate a 3D landscape using the above height mapping technique, although more complex to render the core concepts and ideas are still the same.

Cellular automata:

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

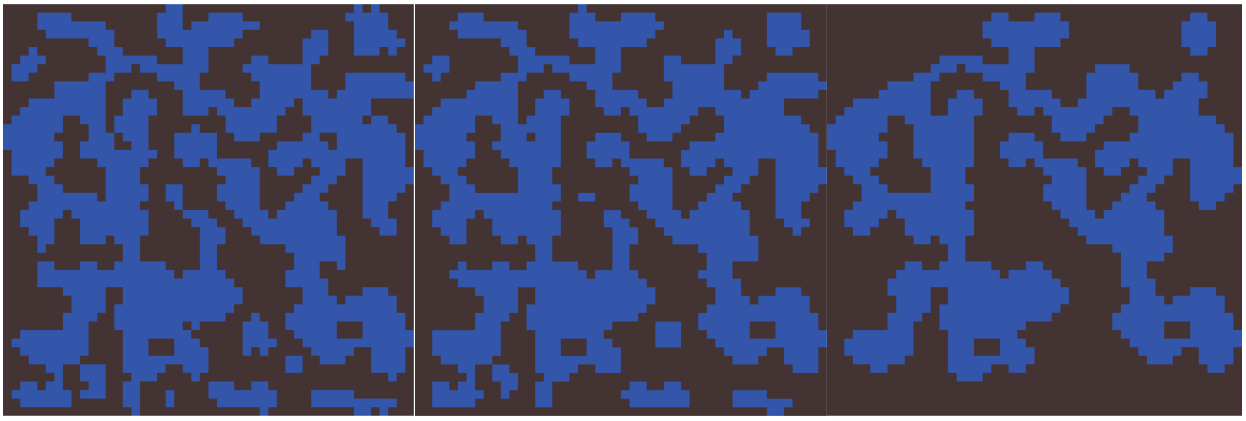
As an example take a 2D grid of Boolean values, now randomly seed this grid with values. The algorithm will go through each value in this 2D array and apply a pre-defined set of rules to the cell. An example rule could be if more than 5 of the cells 8 neighbours are different then change the value of 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 affecting the current cell.

This process will give an overall smoothing effect to the grid of values. In order to demonstrate the functionality of cellular automata I will utilize an example from an article by Michael Cook[2], the implementation from this article is a cave generator that uses cellular automata to generate caves.



On the left is the initial state of the seeded map, with each cell having a 40% chance of being “alive”. The end goal will be to produce a cave system that is smooth and possibly containing interconnecting caves.

One step of this algorithm involves each cell being run through the ruleset provided by Cook, namely if the cell is alive, how many neighbours does it have, if enough kill it and if a cell is dead, check if it has enough neighbours to be born.



The above images illustrate the algorithm running through the given map. On the left is the map after the first step, notice that in the initial step a lot of the 1-cell noise is removed, either by joining the “empty space” (alive) or by joining with a wall (dead). The middle image shows the second step being executed, this step you can clearly see that a lot of the rough edges in the walls are being smoothed out by the algorithm. The image on the right shows the end state of the map, this was achieved by running the algorithm until it no longer made any changes to the map at each step, alive and dead cells in equilibrium. Notice how almost all of the noise and rough edges of the walls have been removed leaving us with a pleasing and easy to traverse cave system.

Technologies used

Git:

Git is a service that provides 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 during the initial stages of the project, 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 in 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 programming language typically used to create intractability within a website. JavaScript is used in most if not all modern websites and is one of the three core web technologies, because of this all top end web browsers have detailed debugger and console capabilities for JavaScript which were excellent for the development of my implementation.

The driving force behind my decision to use JavaScript for my implementation was my use of PixiJS, PixiJS being the JavaScript render library that I used to render my map. 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 locally, so in response I switched to using Firefox as it does not possess any of these restrictions.

Another reason that I chose JavaScript was the fact that throughout my course I had not done much in the area of web development and so JavaScript. With JavaScript being at the forefront of the development community I decided that it would be in my own best interest to learn and become proficient with the use of JavaScript. With such tools as Node.js and electron on the rise the need for JavaScript in the software engineering industry is now higher than it ever was in the past, with electron allowing the use of web technologies to develop cross platform application with relative swiftness and ease.

The functional aspects to JavaScript also suits my needs perfectly for this project as there will be much mathematical manipulations to be done. While at the same time the object orientated aspects suit me perfectly for structuring my code with the use of objects for tiles and the generator. In some aspects JavaScript is tailored perfectly for my needs and is superior to the other platforms that I had considered such as Microsoft's XNA.

PixiJS an Overview:

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

For my project no intensive 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 specific coordinates on 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 Node.js, 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 Microsoft's XNA was that PixiJS is still being actively developed and so will continue to be relevant 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 allow it to claim this status. First of which being its speed and the fact that PixiJS will automatically use the WebGL renderer if 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 also has multiplatform support that covers all 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 complex functions of the API I found it to be quite intuitive and straight forward.

One of PixiJS' best, sometimes overlooked feature 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[3] is a library for JavaScript which I discovered when I was researching Perlin noise and its applications in procedural generation, 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 which was beneficial for testing as I could use the same seed and observe the results of my code on a map that would remain constant.

CHAPTER 3 CODE AND PROCESS DISECTION

Initial stages:

My initial approach to the implementation stage of my project after I made the decisions on what technologies that I would use was to familiarize myself with these 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 involve 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 previously utilized. 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 helpful which helped me to progress my knowledge and understanding 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 way 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 in order 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.

```
function tile(x, y, type, noise){

  this.x = x;
  this.y = y;
  this.type = type;
  this.noise = noise;
  this.distanceTowater = null;
}
```

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). 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 an array of useful information, 1. Number of different tiles 2. An array of the types of the surrounding tiles 3. An array of the actual surrounding tiles 4. Minimum distance to water.

Smoothing:

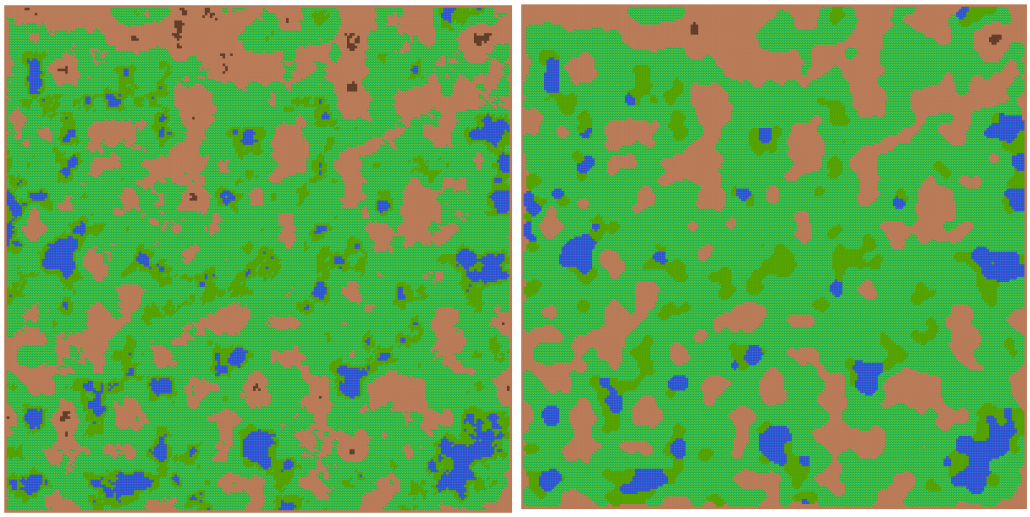
As soon as I got the base map created and rendered my first objective was to manipulate the map in order to remove some of the outlying “noisy” tiles, as these tiles would interfere with any of the future tasks I wanted to perform on the map. So in a sense the following is a representation of how I went 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 reduce the visual noise produced by the noise function. The main thing that I had to remember was to read from one array and write to another, and then to set the rules of what would induce the smoothing effect.

```
65    function smooth(){
66      var holdingArr = mapArr;
67
68      for(i = 1; i < h-1; i++){
69        for(j = 1; j < w-1; j++){
70          var answer = surroundingTiles(mapArr[i][j]);
71          if(answer[0] >= 5){ //5 or more different tiles
72            holdingArr[i][j].type = mode(answer[1]); //most common of the tiles
73          }
74
75        }
76      }
77      mapArr = holdingArr;
78    }
```

The above code snippet 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 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 you may notice 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. In most cases my algorithms do not take the edges of the map into account in the same way as above, as the edges are to be seen as a barrier and not actually integral to the map.



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:

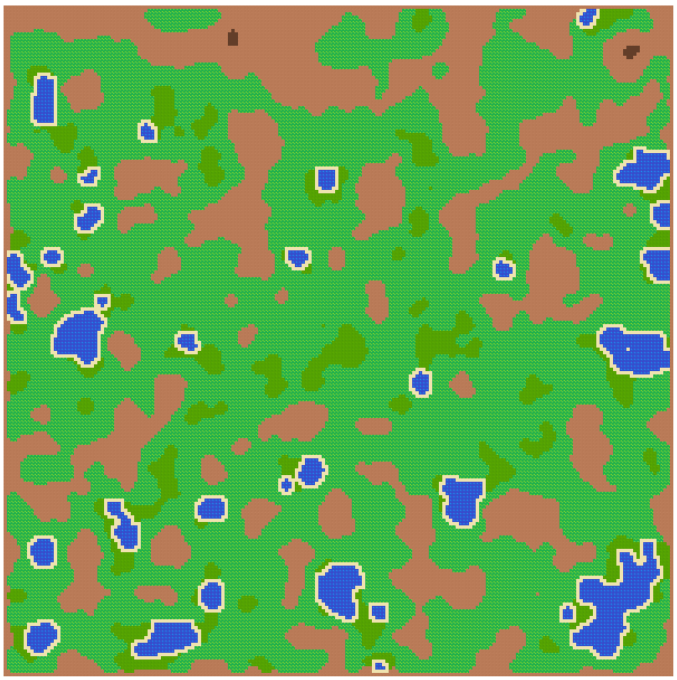
In order to add more detail to the map and to make it more realistic I decided to produce an algorithm that would go through the map and apply sand to the areas in which sand would naturally form. I decided these areas would be around every current water source directly after the smoothing phase.

```
function sand(){
  var holdingArr = mapArr;

  for(i = 1; i < h-1; i++){
    for(j = 1; j < w-1; j++){
      var waterPresent = false; //set false initially
      var answer = surroundingTiles(mapArr[i][j]);

      for(p in answer[1]){ //check each
        if(answer[1][p] == "water"){ //if water in answer[1]
          waterPresent = true; //set to true
        }
      }
      //if not water and water present
      if(mapArr[i][j].type != "water" && waterPresent == true){
        holdingArr[i][j].type = "sand"; //set to sand
      }
    }
  }
  mapArr = holdingArr;
}
```

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 that the tile has to go through is to see if it has any adjacent water tiles, 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 tile 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 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 and 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 previous and finally stopping at the lowest tile or when it hits water.

```
104    function river(tile){
105      previousRiver.push(tile); //add to previuos buffer
106      var heights = [];
107      var surrTiles = surroundingTiles(tile)[2]; //get array of surrounding tiles
108
109      for(t in surrTiles){ //check if tile was used before
110        for(x in previousRiver){
111          if(previousRiver[x] != surrTiles[t] && surrTiles[t].noise < tile.noise){ //if not used
112            heights.push(surrTiles[t].noise); //log its height
113          }
114        }
115      }
116      minimum = Math.min.apply(Math, heights) //get minimum height
```

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 river tile later. The algorithm then initializes the two arrays of heights and surrTiles, heights being all the heights of candidate next river tiles.

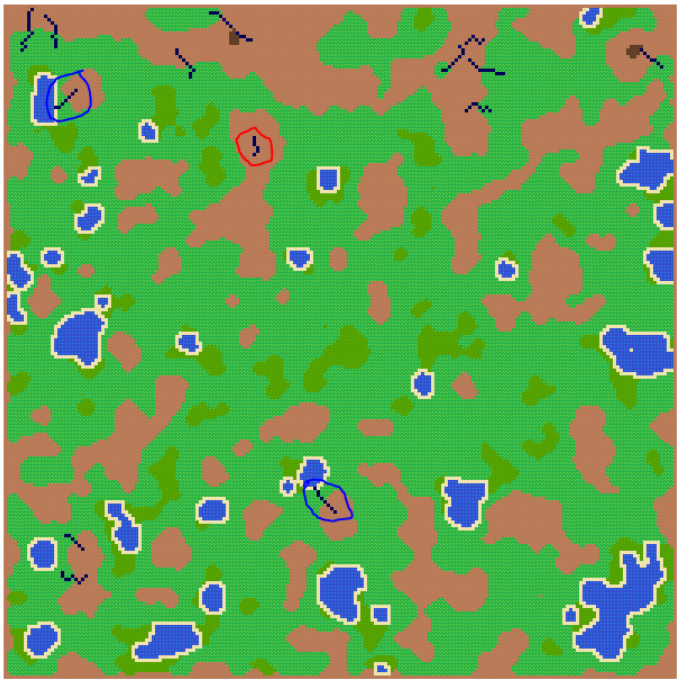
Next we loop through surrTiles and all of the previous river tiles and perform a check, the check itself is to ensure that 1: the surrounding tile has not been used before (! =) and 2: that the surrounding tile has a noise/height value that is lower than the current tile. If a tile passes both these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

```
153
154       if(tile.type != "water"){ //if not already water
155         //if already water the river has reached destination
156         tile.type = "river"; //set to water
157         for(t in surrTiles){
158           if(surrTiles[t].noise == minimum){ //whichever tile matches minimum
159             river(tiles[t]); //recurr
160           }
161         }
162       }
163     }
```

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 tile's 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.



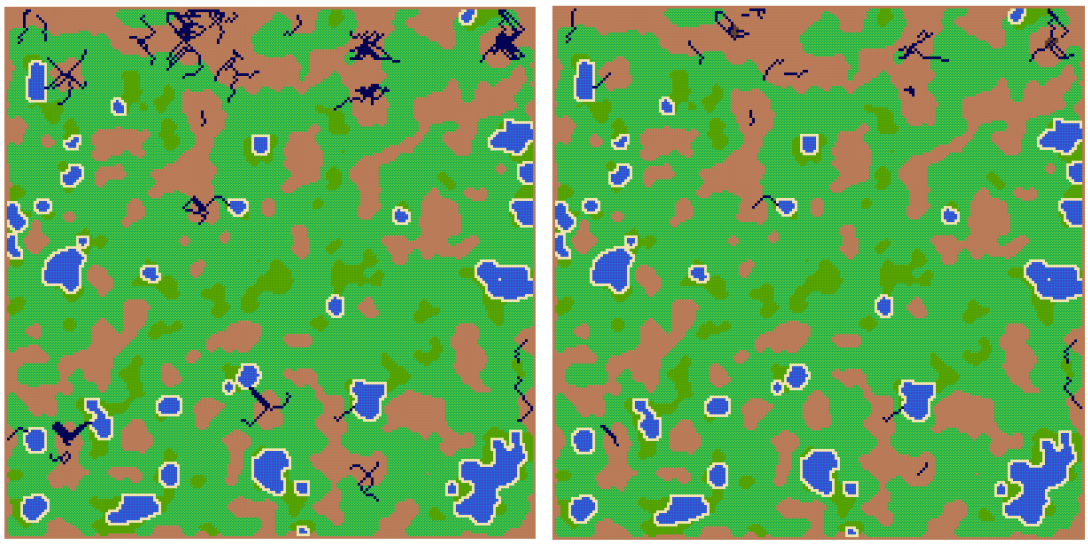
Here I have circled two rivers in blue, these are the rivers that acted “perfectly” and flowed from a local highpoint in the mountains into a lowpoint in the water.

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 as when looking at this from a point of view of being natural looking, 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.

```
266       for(p in peaks){
267         if(p%8 == 0){ //cut down on number of rivers
268           river(peaks[p]);
269         }
270       }
```

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

Calculating distance to water:

In order to add further natural detail to the map I needed some way to measure how "green" a tile was. To do this I decided to create a function that will calculate the distance to the nearest water source. The initial step of this algorithm seeds the initial distance to water values, although these initial seeds are not the correct nearest distance.

The algorithm is split up into two parts, originally two separate algorithms I joined them due to their similarity. The algorithm decides which part of the code to use depending on a Boolean parameter fed to it.

```
136   function DistanceTowater(initialSeed){
137     var holdingArr = mapArr; //assign a holding array
138     for(var i = 1; i < h-1; i++){
139       for(var j = 1; j < w-1; j++){
140
141         if (initialSeed == false) { //check if water is seeded or not
142           if(mapArr[i][j].type == "water"){ //keeps water tiles at 0
143             holdingArr[i][j].distanceTowater = 0;
144           }
145           //if there is no distance information and the minimum distance of surroundingTiles is not infinity (minimum of empty array)
146           else if(mapArr[i][j].distanceTowater == null && surroundingTiles(mapArr[i][j])[3] != Infinity){
147             holdingArr[i][j].distanceTowater = surroundingTiles(mapArr[i][j])[3] + 1; //assin to min distance + 1
148           }
149         }
150
```

Here we see the first part of the algorithm which executes if the algorithm is fed false. The algorithm will loop over the tiles, excluding the edge tiles, if a tile is water it will assign its distance value to zero. Next it will check if the distance to water of the till is null and if the minimum

distance is not infinity, this will happen when getting the minimum of an empty array which is possible due to the surroundingTiles() function. If the requisites are met then the distance for this tile will be set as the minimum of the surrounding tiles plus one, thus ending the first part of the algorithm, seeding some basic values.

These initial seeding of values do not take water to the right of the tile into account and so there could be water to the right of a tile but its distance value might be 20, the second part of the algorithm was constructed to directly address and rectify this problem.

```
151         else{ //if water is seeded
152           if(mapArr[i][j].type == "water"){ //keeps water tiles at 0
153             holdingArr[i][j].distanceTowater = 0;
154           }
155           else if(surroundingTiles(mapArr[i][j])[3] != Infinity){ //do every tile not just null tiles
156             holdingArr[i][j].distanceTowater = surroundingTiles(mapArr[i][j])[3] + 1;
157           }
158         }
159       }
160     }
161     mapArr = holdingArr; //re assign mapArr
```

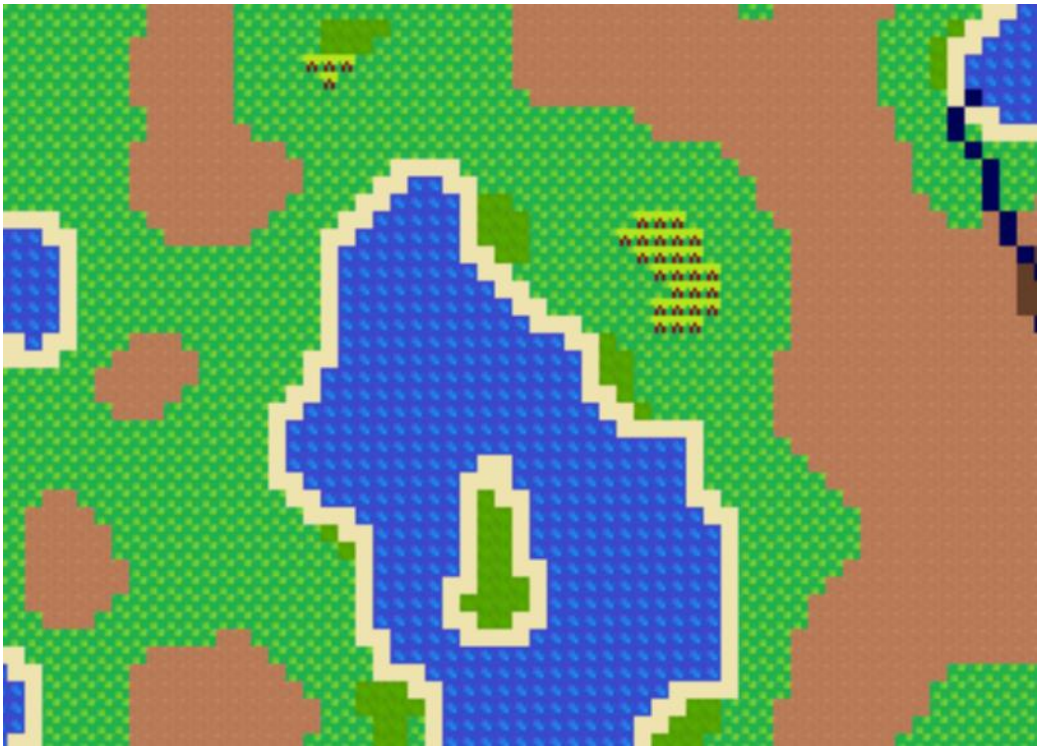
This section of code is used after the initial pass of seeding the distance values and was developed to take all water sources into account when calculating the distance. Again this section keeps all water tiles at zero but differs in the next part, it will take all tiles now not just the null tiles. Now that all tiles have a distance we can more accurately collect the minimum distance information.

To demonstrate this take the example above where a tile is next to a water source but has the distance set at 20 due to initial seeding, from the first part of this algorithm the tile did not know the tile to the right of it was water as it had not yet passed through the algorithm, now that it has the new minimum in the surrounding tiles will be zero, due to the water, which will leave this tile with a value of zero plus one, which is now correct. Using this method to correct the errors of the initial stage takes a number of passes in order to reach all of the tiles.

```
243     for(iteration = 0; iteration < 19; iteration++){
244       DistanceTowater(isWaterSeeded);
245       isWaterSeeded = true;
246
247     }
```

To the left is the code used to execute the DistanceTowater() function. The function will be executed 20 times, with the first being used to seed the values due to isWaterSeeded being initially set to false. Once the first loop is completed isWaterSeeded is set to true and so every next call on the function will use the second half of the algorithm.

The main use of this DistanceTowater() function in my implementation is to be used to put a quantifiable number on the fertility of grass and with this seed trees into the world. I have designated a second form of grass that lies within a distance to water of 10 and within a height range of water level to grass level minus one third of grass level, in order to seed plant life in the map I use this new fertile grass as a constraint and position the plant life between 3 and 8 tiles of water.



From the image above you can clearly see the fertile grass shown around the edges of the water and the plant life that has formed on the fertile grass if the conditions are correct. In my opinion this gives the map a much needed hint of life and spontaneity and brings elements of weather into the map, which will be discussed later in this report.

CHAPTER 4 OBSERVATIONS

What this section entails:

Throughout this section I will dissect the different choices and observations that I have made throughout my research and implantation, shedding light on some of the restrictions on the technologies and techniques that I have used. I will split this chapter up accordingly

Observations on Technologies

Asynchronous nature of JavaScript:

My first observation is that of the asynchronous nature of JavaScript, this did not affect me until I set out to improve the performance of my implementation. The asynchronous nature did not directly affect performance but it did affect my attempts at improving it.

My performance issues resulted from the fact that I am rendering quite a lot of tiles to the screen in my implementation, at a standard level I am rendering 40,000 tiles. This is quite intensive especially on a web browser such as Firefox. The performance was so slow that even scrolling through my map was jittery with it sometimes freezing, this was not acceptable. I consulted my supervisor due to his extensive knowledge of PixiJS hoping that he would have a way to improve performance, he pointed out to me that since none of my tiles were in fact animated I could use PixiJS' render texture functionality, which is basically taking a Pixi container object and creating a texture from it. This was ideal as I could simply create a texture from my map and then use this texture for a single sprite and so drop the amount of tiles being rendered from 40,000 to 1, removing any and all performance issues.

The problem now came on implementing this render texture functionality. Upon utilizing the function I was left with a blank screen. Which meant that the texture itself was not being created properly. When debugging this I made sure that the map itself was still being created and that the tiles were still being rendered to the Pixi container. In each case this turned out to be true, which left me at an impasse. The problem turned out to the asynchronistic JavaScript, the render texture was being created from the container before the map had been rendered to it, rectifying this was simple. I set a timeout on the creation of the render texture to ensure that the map had been added to the container and that there would not be a blank texture being created. This solution turned out to work perfectly and solved all of my performance problems. This problem lead to a firsthand experience with this aspect of JavaScript and I kept this in the back of my mind throughout the rest of my implementation, although it didn't pose a problem again.

```
53    function delay(){
54      var renderTexture = PIXI.RenderTexture.create(screen_width, screen_height); //create render texture
55      renderer.render(stage, renderTexture); //render stage container to texture
56      var stageImage = new PIXI.Sprite(renderTexture); //make new sprite using texture
57      mapImage.addChild(stageImage);
58      animate(); //draw
59    }
```

Above is the delay function that I used in order to rectify the aforementioned problem, here we can see the render texture being created using the screen height and width, then we use the renderer to render the stage, which is the Pixi container that contains all of the tiles, to the render texture. Finally we create a sprite that uses the render texture as its texture and then we add this to the mapImage, which is the container that we will render/draw to the screen, we end by calling the animate function. This delay() function is called using the setTimeout() function in JavaScript.

Observations on Techniques/Algorithms

Representing map tiles:

In the first stage of my implementation I provide a 2D array for the noise values for each cell. I then used this to select which texture to use and then to render that texture in the correct position on the map. Upon creating the tile object I also created a 1-dimenional array of these new tile objects, initially this was just for me to be able to inspect the attributes of each tile from the console. But as I continued to extend the functionality of the implementation I stuck with this 1-Dimensional array.

Sticking with this way of representing my data, was ok at the start, although this did leave the algorithms I was developing very intensive but they did not slow down the implementation, until I tried implementing the sand() function to place sand around the water bodies. Then the performance took a massive dive, and I knew why, the algorithm was performing thousands of unnecessary loops and depending on the size of the map could take several minutes to execute. However these algorithms did work, as tests on smaller maps gave out the desired results I knew the performance dive and algorithm design in general was not acceptable. This led me to redesign all of my algorithms to work with my new representation of the tiles. The new representation was that the tiles would be stored in a 2D array in the same way that the noise values were. This allowed me to directly access the surrounding tiles and specific surrounding tiles a lot more efficiently than the previous manner. Of course this meant that my old algorithms no longer worked, so I had to create new algorithms, these new algorithms greatly surpassed the old ones in performance and in their effect on the map. Particularly the sand() function which when using my new algorithm perfectly traces the water sources. The major downside of this was the amount of time that I had

to spend redoing old algorithms that could have been spent implementing new and interesting ways to represent natural phenomena within my implementation such as the topic in the next section.

Representing weather patterns:

During the early stages of development, one of the ideas that I toyed with was implementing a system that would simulate weather in the map. This would entail wind, moisture and humidity. These values could be used to realistically produce grasslands, forests, deserts and other such natural constructs, this did not happen.

From research into the area of weather simulation I came to the conclusion that weather was much too complicated to simulate with the time that I have to spend on the project. For the sake of this report I will go into the way in which I had planned to simulate weather. To simulate wind I had planned to trace a path from the north east of the map to simulate the wind coming from this directions. Depending on what tiles the wind would interact with it would take on attributes such as if the wind passed over a water body it would be high in moisture and would so leave the land in its wake green and fertile, if the wind had passed over desert the opposite would happened to the land left in its wake. The wind itself would have had a velocity attribute which would quicken when going from high to low lands and slow conversely. Mountains would also have affected this wind, as if the velocity was not fast enough or the mountain was simply too tall then the wind would break and have to disperse around the mountain. My hope in this is that it would create sheltered areas beneath mountains that would be fertile and rich in water. The moisture simulation would work hand in hand with the wind and a tiles minimum distance to water while humidity would have been a function of tile attributes, but during my research into this area I could not figure out an adequate function for humidity.

This weather pattern functionality never made it past the drawing board phase so there is no way to tell if the proposed methods mentioned above would have portrayed weather in a natural fashion**.** Though not a complete waste there are some aspects that I took away from this, one being the distance to water functionality which I used in order to seed forests and trees in the map.

Unforeseen problem with smoothing algorithm:

During my development of my algorithm to calculate a tiles distance to water I discovered bug in the smooth() function that had not occurred to me before and only really affects very niche operations.

In order to fully explain the bug I will first detail how I came across the bug. When investigating the distance to water value for tiles in order to validate them I discovered a phenomenon in a select few tiles. These tiles throughout the map had nothing in common except for that fact that their distance to water value was zero, while they were not in fact water tiles. This stumped me as there was no correlation between the tiles that had the problem, these rogue values completely through off my algorithm to calculate the distance to water.

This caused me to go through and reason out every step of the DistanceTowater() function to ensure that I had not made a mistake when assigning zero to the water tiles, it turns out as I had thought, I hadn't made a mistake. So the next step was to go through my other functions and decide if there was any code in them that would interfere with the distance function. Finally I came across the source of the problem, my constructor for the tile object. I had a parameter in the constructor for distance to water in order to initially seed water tiles as zero and all other tiles as null, which at the time made sense but I had not thought of this in the smooth() function, as when a water tile was changed in the smooth() function the tile type may have been changed but the distance to water value remained zero. This left a select amount of tiles with a zero distance value with them in fact no longer being water tiles. The solution to this was simple and obvious, remove the distance to water parameter entirely from the constructor and simply set every tile's value to null and simply put an if-water statement in the DistanceTowater() function. Now with this alteration of the constructor and DistanceTowater() function there exist no value that the smooth() function does not change that later effects functions and manipulations.

CHAPTER 5 CONCLUSIONS

Hybrid map generators:

Using procedural generation to create a map or an area in your game has its advantages and its disadvantages. Sometimes it is best to simply use a mixture of procedurally generated content and pre-made static content or to lay an overall basic structure of the content that you will be generating. A good example of such an approach is seen in the game Spelunky, this game uses procedural generation to generate its maps but places a few constraints on the blocks that are being generated.

In an article[4] the developers go through the method in which they generate their levels. In order for a level to be fit for the game it needs to have a solution path, a path that the player can take from the start point to the end point. In theory this sounds simple and self-explanatory, but when using procedural generation techniques to completely generate your level, having a solution path is not guaranteed and in most cases one will not be present.

In order to get around this problem the developers cleverly deployed a mixture of fixed attributes to their procedural generator in the form of multiple types of room. These rooms themselves are procedurally generated but with a varying attribute, the amount and positon of exits to the room. With the standard room having two exits, left and right, with other rooms that incorporate exits to the top of the room or the bottom.

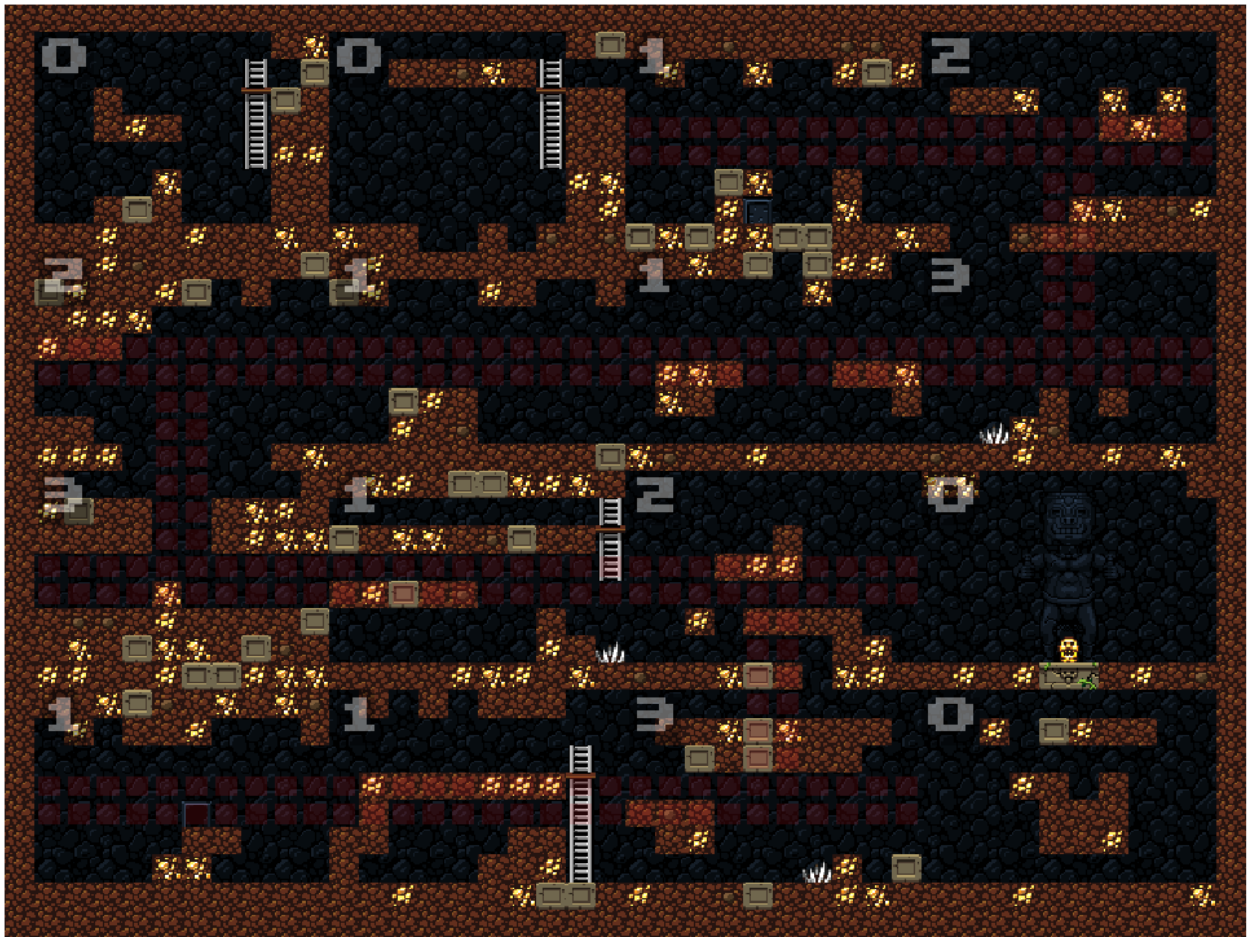
Using this system of different room types allows the map to be represented as a set of room types before/during generation. Every room will always be a standard type room with left and right exits to start with, being changed if it is deemed necessary, the next room's type being chosen in context to what positon the current room is in. As an example take if the next room hits the left edge of the map, this means that the room will have to have an exit either to the top or on the bottom of the room in order to further expand the solution path. This would also leave the room succeeding this to be a "T" shaped room to accommodate for the vertical movement.

0: a side room that is not on the solution path

1: a room that is guaranteed to have a left exit and a right exit

2: a room that is guaranteed to have exits on the left, right, and bottom. If there's another "2" room above it, then it also is guaranteed a top exit

3: a room that is guaranteed to have exists on the left, right, and top



Above we can see a map retrieved from the article that goes into Spelunky's map generator. The map is built from 16 numbered rooms, with the number on each room representing the type of the room, also explained in the article, see above image. The red line shown throughout the map illustrates the solution path. Note the rooms that are marked with a 0, these are rooms that are not on the solution path and are completely procedurally generated with no promise of an exit in any direction, these rooms are only added after the solution path is created.

Spelunky's adaptation of procedurally generated maps brilliantly illustrates methods that are used in order to overcome the pitfalls of 100% procedurally generated maps and is a good example of a hybrid system that consists of the generation and engineering of a map.

Dangers of sole reliance on procedural generation:

When developing a game it is important to not solely rely on the advantages of procedural generation as in modern times the initial surge of games that entail varying degrees of procedural generation has given consumers a very high expectation of the technique. The game that created the initial surge in popularity is without a doubt Minecraft, which in its initial stages did solely rely on its procedural map generation, being the "first" to do this gave Minecraft a pass of sorts on the lack of depth in content. The fact that the building gameplay led to a lot of replay ability did also help with its initial rise.

Later games that also used this form of generation such as Terraria and Starbound set themselves apart by focusing more on the combat and questing areas of the game, which also led to their success in the market.

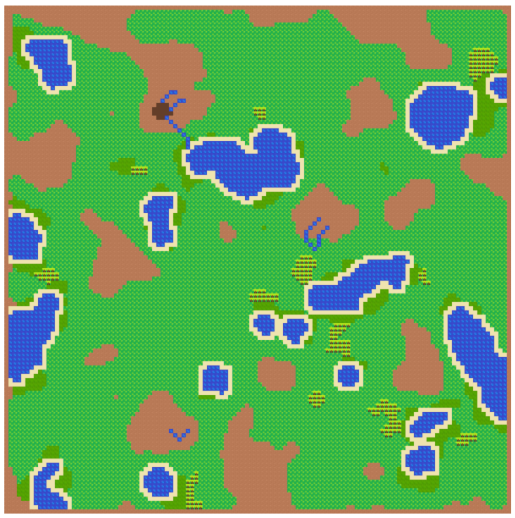
However this is not always the case, as a prime example of the dangers of relying solely on procedural generation is No Mans Sky. No Mans Sky was without a doubt the most hotly anticipated game of 2016 with many fans waiting years for the game to release. NMS promised an endless galaxy of procedurally generated planets with completely unique plant and animal life on each planet. The developers boasted that the galaxies that they were seeding were so large that they had to develop in game probes that explored the universe for the developers.

The fact that the development team of NMS stressed so much on the procedural generation should have raised a few red flags as there was little to no mention of what the player would do in the game other than explore, but the community as a whole were blinded by the sheer promise of such a massive and beautiful game.

But at release the reality of NMS hit the community like a hammer, although entirely procedurally generated the majority of the planets were dull and boring with a lot of repetition in the plant and animal life. The game also offered little else to do other than explore these sometimes lifeless and dull planets.

This lead to a public relations nightmare for the game, and highlights the possible outcomes if too much focus is put on the procedural generation in a game and not enough focus has been put into the other equally important aspects of a game such a story, content, etc.

On my implementation:



Throughout the year my implementation of procedural map generation techniques has shifted and changed in ways that I both anticipated and did not anticipate. I am quite happy with the outcome of the implementation and I am more than pleased with the capabilities of the technologies and techniques that I had chosen to incorporate into my work.

Although I had encountered many challenges and disappointments with my code, such as not being able to implement the weather patterns I am happy with the algorithms that I have done and their effect on the generated map. When looking back on my initial few maps that I had generated before the Christmas break and comparing them to the end product it hardly seems like the same engine, in many ways it is not the same engine.

I plan to conduct further development into my implementation in my spare time over the coming months and years and to someday hopefully make some resemblance of a game from this world creator that I have developed.

Final conclusions:

In conclusion I have found that procedural generation is a very powerful tool in many industries but is most prevalent in the gaming industry in modern times. When developing a game the developers should be careful not to rely solely on the technique of procedural generation whether it be for map or content generation as the danger of falling into the same trap as the developers of No Mans Sky's is very real.

In my opinion procedural generation is the way forward in the gaming industry if it used correctly and in intelligent ways such as with Spelunky. Hybrid systems that use both fixed point generation and procedural generation seem to yield the best results when it comes down to it. Another clever implementation of procedural generation is to pick one core aspect of the game and have it be procedurally driven, with well proven examples of this being the Borderlands series with their use of procedural generation in guns. When being used procedural generation can either make or break a game, and should always be used in an intelligent way and not be completely relied upon.

Sources:

1. https://notch.tumblr.com/post/3746989361/terrain-generation-part-1

2. https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664

3. https://p5js.org/

4. http://tinysubversions.com/spelunkyGen/

APPENDIX. CODE

Index.html

```
<!doctype html>
<meta charset="utf-8">
<title>Noise world gen</title>
<body>
  <script src="src/pixi.js"></script>
  <script src="src/p5.js"></script>
  <script src="src/Generate.js"></script>
  <script src="src/tile.js"></script>
  <script src="src/perlin.js"></script>


  <script>
  //Pixi containers
  var mapImage = new PIXI.Container();
  var stage = new PIXI.Container();


  var map = []; //map array
  //tile dimensions
  var tile_width = 5;
  var tile_height = 5;
  var screen_width = window.prompt("Enter width");
  var screen_height = window.prompt("Enter Height");
  var waterLevel = window.prompt("Enter water level (0-1)");
  var grassLevel = window.prompt("Enter grass level ("+waterLevel+"-1)");
  var mountainLevel = window.prompt("Enter mountainLevel ("+grassLevel+"-1)");

  var h = (screen_height/tile_height); //relative height
  var w = (screen_width/tile_width); //relative width

  //Pixi stuff
  var renderer = PIXI.autoDetectRenderer(screen_width, screen_height ,{backgroundColor : 0x000000255});
  document.body.appendChild(renderer.view);

  function animate() {
    requestAnimationFrame(animate);
    renderer.render(mapImage); //draw map to screen

  }

  function setup(){
    noiseSeed(9); //noise seef
    noiseDetail(8,0.5); // if falloff above .5, noise may return value > 1
    var map = new Generate(); //Generate object
    map.init2DMap(map,h,w); //initialise map
    map.generateNoise(map); //apply noise
    map.generateMap(map); //apply generation algorithms

    setTimeout(delay, 1000); //delay for render texture

  }//end funciton setup

  function delay(){
    var renderTexture = PIXI.RenderTexture.create(screen_width, screen_height); //create render texture
    renderer.render(stage, renderTexture); //render stage container to texture
    var stageImage = new PIXI.Sprite(renderTexture); //make new sprite using texture
    mapImage.addChild(stageImage);
    animate(); //draw
  }
  </script>
</body>
```

Generate.js

```
function Generate(){
  var mapArr = [];
  var counter;
  var previousRiver = [];
  var peaks = [];
  var isWaterSeeded = false;

  //function to find most common item in an array
  function mode(array){
    var modeMap = {};
    var maxElement = array[0];
    var maxCount = 1;

    for(var i = 0; i < array.length; i++){
      var element = array[i];
      if(modeMap[element] == null)
          modeMap[element] = 1;
        else
          modeMap[element]++;
        if(modeMap[element] > maxCount){
          maxElement = element;
            maxCount = modeMap[element];
        }
    }
    return maxElement;
                }

  function surroundingTiles(tile){
    var counter = 0;
    var typeArr = [];
    var ResultArray = [];
    var surrounding = [];
    var waterdist = [];

    //loop over tiles around current tile
    for(x = -1; x < 2; x++){
      for(y = -1; y < 2; y++){
        if(x == 0 && y == 0){} //dont count tile itself
        else{
          typeArr.push(mapArr[tile.y+y][tile.x+x].type); //log base tile type
          surrounding.push(mapArr[tile.y+y][tile.x+x]);
        }
      }
    }
    for(x in typeArr){
      if(typeArr[x] != tile.type){
        counter ++; //counter for different type
      }
    }
    for(x in surrounding){
      if(surrounding[x].distanceTowater == null){
      }
      else{ //if not null
        waterdist.push(surrounding[x].distanceTowater);
      }
    }
    ResultArray.push(counter); //number different tiles
    ResultArray.push(typeArr); //get most common tile
    ResultArray.push(surrounding); //all surrounding tiles
    ResultArray.push(Math.min.apply(Math,waterdist)); //minimum distance
    return ResultArray; //return most common tile and counter
  }
```

```
function smooth(){
  var holdingArr = mapArr;

  for(i = 1; i < h-1; i++){
    for(j = 1; j < w-1; j++){
      var answer = surroundingTiles(mapArr[i][j]);
      if(answer[0] >= 5){ //5 or more different tiles
        holdingArr[i][j].type = mode(answer[1]); //most common of the tiles
      }

    }
  }
  mapArr = holdingArr;
}

function DistanceTowater(initialSeed){
  var holdingArr = mapArr; //assign a holding array
  for(var i = 1; i < h-1; i++){
    for(var j = 1; j < w-1; j++){

      if (initialSeed == false) { //check if water is seeded or not
        if(mapArr[i][j].type == "water"){ //keeps water tiles at 0
          holdingArr[i][j].distanceTowater = 0;
        }
        //if there is no distance information and the minimum distance of surroundingTiles is not infinity (minimum of empty array)
        else if(mapArr[i][j].distanceTowater == null && surroundingTiles(mapArr[i][j])[3] != Infinity){
          holdingArr[i][j].distanceTowater = surroundingTiles(mapArr[i][j])[3] + 1; //assin to min distance + 1
        }
      }

      else{ //if water is seeded
        if(mapArr[i][j].type == "water"){ //keeps water tiles at 0
          holdingArr[i][j].distanceTowater = 0;
        }
        else if(surroundingTiles(mapArr[i][j])[3] != Infinity){ //do every tile not just null tiles
          holdingArr[i][j].distanceTowater = surroundingTiles(mapArr[i][j])[3] + 1;
        }
      }
    }
  }
  mapArr = holdingArr; //re assign mapArr
}

function sand(){
  var holdingArr = mapArr;

  for(i = 1; i < h-1; i++){
    for(j = 1; j < w-1; j++){
      var waterPresent = false; //set false initially
      var answer = surroundingTiles(mapArr[i][j]);

      for(p in answer[1]){ //check each
        if(answer[1][p] == "water"){ //if water in answer[1]
          waterPresent = true; //set to true
        }
      }
      //if not water and water present
      if(mapArr[i][j].type != "water" && waterPresent == true){
        holdingArr[i][j].type = "sand"; //set to sand
      }
    }
  }
  mapArr = holdingArr;
}
```

```
function river(tile){
   previousRiver.push(tile); //add to previuos buffer
   var heights = [];
   var surrTiles = surroundingTiles(tile)[2]; //get array of surrounding tiles

   for(t in surrTiles){ //check if tile was used before
    for(x in previousRiver){
     if(previousRiver[x] != surrTiles[t] && surrTiles[t].noise < tile.noise){ //if not used
      heights.push(surrTiles[t].noise); //log its height
     }
    }
   }
   minimum = Math.min.apply(Math, heights) //get minimum height

   if(tile.type != "water"){ //if not already water
    //if already water the river has reached destination
    tile.type = "river"; //set to water
    for(t in surrTiles){
     if(surrTiles[t].noise == minimum){ //whichever tile matches minimum
      river(surrTiles[t]); //recurr
     }
    }
   }
  }
  //function to place a tile at coordinates
  function placeTile(x,y,texture){
   //create tile sprite
   var tile = new PIXI.Sprite(texture, tile_height, tile_width);
   tile.position.x = x; //set sprite x
   tile.position.y = y; //set sprite y
   stage.addChild(tile); //add to stage container
  }


  this.generateNoise = function(map){
   var ycoord = 0;
   for(i = 0; i < h; i++){ //loop through rows
    var xcoord = 0;
    for(j = 0; j < w; j++){ //loop through columns
     xcoord += .08;
     if (i ==0 || j == 0 || i == h-1 || j == w-1){ //edges = 1
      map[i][j] = 1;
     }
     else{
      map[i][j] = noise(xcoord,ycoord); //produce nouse for each element in map
     }
    }//end j loop
    ycoord += .08;
   }//end i loop
  }//end function generateNoise

  //initialze a map at user specificatioins
  this.init2DMap = function (map, height, width){ //takes in array and relative dimensions
   map.length = height;
   for(i = 0; i < height; i++){
    map[i] = [];
    for(j = 0; j < width; j++){
     map[i][j] = random(0,1); //fill out map array
    }//end j loop
   }//end i loop
  }//end function init2DMap
```

```
this.generateMap = function(map){
   //textures and arrays
   this.init2DMap(mapArr,h,w);
   var grass = PIXI.Texture.fromImage('img/grass.png');
   var mountainTexture = PIXI.Texture.fromImage('img/brown.jpg');
   var waterTexture = PIXI.Texture.fromImage('img/blue.jpg');
   var sandTexture = PIXI.Texture.fromImage("img/yellow.jpg");
   var FertilegrassTexture = PIXI.Texture.fromImage("img/fertilegrass1.png");
   var peakTexture = PIXI.Texture.fromImage("img/peak.png");
   var treeTexture = PIXI.Texture.fromImage("img/tree.png");

   for(i = 0; i < h; i++){
    for(j = 0; j < w; j++){

      //Grass placement
      if(map[i][j] > waterLevel && map[i][j] <= grassLevel){
        var x = new tile(j, i, "grass", map[i][j]);
        mapArr[i][j] = x;
      }
      //mountain placement
      else if(map[i][j] > grassLevel && map[i][j] <= mountainLevel || map[i][j] == 1){
        var x = new tile(j, i, "mountain", map[i][j]);
        mapArr[i][j] = x;
      }
      //water placement
      else if(map[i][j] >= 0 && map[i][j] <= waterLevel || map[i][j] > 1){ // >1 for when Noise detail above .5
        var x = new tile(j, i, "water", map[i][j]);
        mapArr[i][j] = x;
      }

      else if(map[i][j] >= mountainLevel && map[i][j] < 1){ // >1 for when Noise detail above .5
          var x = new tile(j, i, "peak", map[i][j]);
          mapArr[i][j] = x;
          peaks.push(x);
        }
    }
   }
   //smooth 4 times
   for(var call = 0; call < 3; call++){
    smooth();
   }
   console.log("smoothed");
   //calculate distance to water for tiles
   for(iteration = 0; iteration < 19; iteration++){
    DistanceTowater(isWaterSeeded);
    isWaterSeeded = true;
   }

   for(var i = 1; i < h-1; i++){
    for(var j = 1; j < w-1; j++){
      //create fertile grass tiles
      if(mapArr[i][j].distanceTowater < 10 && mapArr[i][j].noise > waterLevel && mapArr[i][j].noise < grassLevel-grassLevel/3 &&
mapArr[i][j].type == "grass"){
        mapArr[i][j].type = "Fertilegrass";
      }
      //seed trees
      if(mapArr[i][j].type == "Fertilegrass" && mapArr[i][j].distanceTowater < 8 && mapArr[i][j].distanceTowater > 3){
        mapArr[i][j].type = "tree";
      }
    }
   }
   smooth(); //smooth trees and fertile grass
   DistanceTowater(true); //do distanceTowater again due to smooth
   console.log("distance to water calculated");

   sand(); //place sand
   console.log("sand placed");
```

```
//place rivers
   for(p in peaks){
     if(p%8 == 0){ //cut down on number of rivers
       river(peaks[p]);
      }
   }
   console.log("rivers placed");

   //loop over and place tiles
   for(i = 0; i < h; i++){
     for(j = 0; j <  w; j++){
       if(mapArr[i][j].type == "grass"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, grass);
       }
       else if(mapArr[i][j].type == "mountain"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, mountainTexture);
       }
       else if(mapArr[i][j].type == "water" || mapArr[i][j].type == "river"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, waterTexture);
       }
       else if(mapArr[i][j].type == "sand"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, sandTexture);
       }
       else if(mapArr[i][j].type == "Fertilegrass"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, FertilegrassTexture);
       }
       else if(mapArr[i][j].type == "peak"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, peakTexture);
       }
       else if(mapArr[i][j].type == "tree"){
         placeTile(mapArr[i][j].x*tile_width, mapArr[i][j].y*tile_width, treeTexture);
       }
     }
   }
   //log amount of tiles and array, for debugging
   console.log(w*h, "tiles", mapArr);
   console.log("Map generated");
  }
}
```

tile.js

```
function tile(x, y, type, noise){
  this.x = x;
  this.y = y;
  this.type = type;
  this.noise = noise;
  this.distanceTowater = null;
}
```