Secure Map Generation for Multiplayer, Turn-Based Strategy Games

---

A Thesis

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Stephen L. Rice

June 2014

Advisor: Chris GauthierDickey

©Copyright by Stephen L. Rice 2014

All Rights Reserved

Author: Stephen L. Rice
Title: Secure Map Generation for Multiplayer, Turn-Based Strategy Games
Advisor: Chris GauthierDickey
Degree Date: June 2014

ABSTRACT

In strategy games, players compete against each other on randomly generated maps in an attempt to prove their superior skill. Traditionally, these games rely on a client/server architecture with one player fulfilling the role of server and holding responsibility for the map generation process. We propose, analyze and evaluate a method that allows these maps to be created in a peer-to-peer fashion and thus reduce the potential for cheating. We provide an example map generation program that puts these concepts into action and demonstrate how it can be extended and customized for any game. Finally, we analyze the performance of our methods and demonstrate how it can be scaled from a two player game to an n-player game.

ACKNOWLEDGEMENTS

Stephen Rice would like to thank Chris GauthierDickey for his advice and assistance throughout the last year of work. He would also like to thank the members of his Oral Defense Committee for the time and feedback they provided on this paper. Finally, he would like to thank the faculty and staff of the University of Denver Computer Science Department for their support.

TABLE OF CONTENTS

Chapter One: Introduction 1

Chapter Two: Background 4
- Peer-to-Peer 4
- Procedural Content Generation 5

Chapter Three: Algorithms 8
- Problem Space 8
- Diamond-Square Algorithm 9
- Simplex Noise Generation 13
- Map Generation 13
- Peer-to-Peer Map Generation 16

Chapter Four: Analysis 19
- Experiment Setup 19
- Results 19

Chapter Five: Conclusion and Future Work 22

Bibliography 23

Appendices 25
- Appendix: Source Code 25

TABLE OF FIGURES

Figure 1: Sample Grid for the Diamond-Square Algorithm 10

Figure 2: Sample Fractal as generated by the Diamond-Square Algorithm 11

Figure 3: Pseudo-code for the Diamond-Square Algorithm 12

Figure 4: Examples of Perlin Noise in 2D and 3D 13

Figure 5: Layers of our Map Generator 14

Figure 6: Sample Maps as Generated by Our Map Generator 16

Figure 7: Visual Representation of the Random Seed Generation Algorithm 18

Figure 8: Graph of Program Time Observations 20

Figure 9: Table of Program Time Observations 21

CHAPTER ONE: INTRODUCTION

Strategy games, such as *Sid Meier's Civilization* or Blizzard's *StarCraft*, are defined by their focus on tactics, both in city or base building as well as in conflict between armies of varying sizes. These games can be played by one player against computer controlled opponents or can support multiple players competing (and cooperating) amongst themselves. They can be played in distinct turns (as in traditional games such as Chess) or take place in real-time. Strategy games vary in length; some can be finished in under an hour (as many StarCraft games are) and others may take more than four to five hours (as in Civilization) and can often be played in multiple sessions (using a save and resume system).

Most strategy games challenge players in two ways. First, players test their understanding of the game mechanics against each other by competing for resources, military power or political influence as they attempt to satisfy one of the game's victory conditions. Second, players test their ability to adapt by playing on randomly generated maps for each new round. These maps can be slight variations on preset terrain (i.e. new resource locations) or procedurally generated worlds with no predetermined elements. Often the first act of a player is gather information about the map so that they can develop their strategy for the rest of the game.

Traditionally, one player is selected as a host, or server and is responsible both for generating the map for that game and for acting as a server for networked play. Other

players then send their actions and moves to this player and wait for confirmation on how the game state has been changed. In real-time strategy games, network lag (caused by packet loss or a low transmission rate) can severely affect the outcome of a game, though this is less of a concern in turn-based strategy games.

This server/client set-up is vulnerable to cheating, however, as the hosting player has the ability to cheat by manipulating the map to his advantage or otherwise interfering with other players commands. Furthermore, the host player can modify the rules of the game or change the map to suit their play style.

In this paper, we propose a method that allows maps to be generated between players in a peer-to-peer fashion which prevents any one player from gaining an unfair advantage. No player can have more knowledge of the map than another player at the start of the game, nor can one player influence the appearance of the game map during the map generation process. We specifically worked with two common procedural content generation (PCG) algorithms: the Diamond-Square algorithm originally presented in Fournier's et al. *Computer Rendering of Stochastic Models* [1] and a Simplex Noise function created by Ken Perlin [2], and demonstrate how the procedure can be used in other PCG algorithms as well. We demonstrate that, by securely generating a random value between n players, each player can generate identical maps without affecting the process.

Chapter 2 of this paper will explore many of the papers that have been written on peer-to-peer game playing and procedural content generation in the last five years as well as a few older papers that lay down algorithms still in use today. Chapter 3 will discuss

the Diamond-Square algorithm in detail as well as our methodology for securely generating random numbers between multiple players. It will also lay out a clear procedure for implementing peer-to-peer map generation from start to finish. Chapter 4 will analyze the Java code constructed as a proof-of-concept and present our conclusions on the expected cost of the algorithm. Finally, chapter 5 will present our conclusion and comment on future work to be done in this field. The appendix presents the source code that is analyzed in chapter 4.

CHAPTER TWO: BACKGROUND

When we began our research, we investigated two specific fields: Peer-to-Peer (P2P) Games and Procedural Content Generation. The former was essential to understanding how to build and apply P2P concepts to the field of map generation while the latter was essential to understanding how game maps are generated in modern games.

A. Peer-to-Peer

In *Secure Peer-to-Peer Trading for Multiplayer Games* [3], GauthierDickey and Ritzdorf investigated how to support trading in a multiplayer online game without using a central server. They began by identifying two goals: that trades can be made fairly and that items cannot be duplicated (thus violating the spirit of the game). Their system consisted of a distributed hash table (DHT), which was used to uniquely identify items, and a method of signing items to provide a proof of transaction. They proved that the system works for both one-way and two-way trades and demonstrated how it can be extended to multi-item trades. Though the methods are not strictly related to P2P map generation, they can be used in a P2P based strategy game (e.g., when players trade resources amongst themselves).

The groundwork for our method is laid down in Pittman and GauthierDickey's *Match+Guardian: A Secure Peer-to-Peer Trading Card Game Protocol* [4]. The paper began by analyzing the three trading card game (TCG) play styles: sealed deck, draft deck and constructed deck. The paper then broke each of these play styles into a concrete

set of actions (such as drawing a card, shuffling a deck of cards, etc.) and provided a method for securing this step in a P2P game. The core theory behind many of these techniques is to generate a secure random number between two players such that neither player can influence what number (or card) is chosen. In this paper, we apply this technique to procedural content generation algorithms which allows them to be used fairly in P2P games.

B. Procedural Content Generation

In their paper, *Procedural Content Generation for Games: A Survey* [5], authors Hendrikx, et al. noted that the study of Procedural Content Generation in Games had been widely dispersed over a variety of research areas and that many lacked a common, unifying base. To rectify this shortcoming, they surveyed existing PCG papers in fields such as Artificial Intelligence, Pseudo-Random Number Generators or Complex System Simulations and defined two taxonomies. The first was dedicated to breaking down the elements of a game into increasingly smaller “bits” which will help researchers communicate what exactly they are attempting to generate. The second taxonomy was of various PCG techniques which might be applicable to games. This paper laid down an excellent framework for understanding and building multiple “layers” of content generation.

In *A Survey of Procedural Terrain Generation Techniques Using Evolutionary Algorithms* [6], Raffe, Zambetta and Li evaluated and compared six known Evolutionary Algorithms (EA) for the purposes of Procedural Terrain Generation (PTG), specifically for their use in interactive entertainment. Evolutionary Algorithms enabled the creation of

families of generated terrains with varying features that the authors hoped will create interesting game spaces. Each algorithm was evaluated on a number of factors, including its refinement (how well does it explore a family of terrains), its variety (how well does it explore the solution space of terrains), its control (how easy is it for a user to manipulate) and finally its game integration (how well would it work in a game space). Most of the algorithms surveyed do improve on a simple fractal terrain but often failed to create usable game spaces (due to a lack of game integration fitness checks). The one algorithm that does specifically aim to create Real Time Strategy maps failed to create maps with any sort of interesting terrain (only rounded hills and ridges are created by the EA). In their conclusion, the authors reaffirmed their belief that Evolutionary Algorithms can be used to generate Procedural Terrains for games but that more work is needed to ensure accurate and useful results.

We explicitly explored the aforementioned paper dealing with real time strategy maps, titled *Multiobjective Exploration of StarCraft Map Space* [7]. Togelius et al., developed and utilized fitness functions related to gameplay elements to guide the evolutionary algorithms such that playable and "fun" multiplayer maps are created. The authors found that selecting proper fitness functions was the key to creating usable maps. While many of the fitness functions were promising, using more than one or two at a time caused conflicts that produced undesirable results or had unreasonably long run times. The authors concluded that more research was needed on creating, combining and simplifying fitness functions and we thus turned out attention away from evolutionary algorithms and toward more traditional fractal generators.

In their paper, *Randomly Generated 3D Environments for Serious Games* [8], Noghani et al. explored a methodology to create a game space for a simple flight simulator. A Diamond-Square algorithm was used to generate the terrain with additional pseudo-random algorithms to generate decorations such as vegetation or buildings. They also explored a variety of methods to create bounding boxes for hit detection though many users criticized their use of single-axis aligned bounding boxes. Users also did not notice that terrain was repeated (when encountering the edge of a tile) though this could be due to the short test time.

The Diamond-Square algorithm was often mentioned during the discussion of fractal terrain generators and after examining *Civilization V*'s script files, we decided to further explore this algorithm. We also looked into Simplex Noise, another method used in the generation of terrain and random textures. Both of these algorithms will be investigated in more depth in the next section.

CHAPTER THREE: ALGORITHMS

We will begin by discussing our initial analysis of the problem space and the decisions we made. We will then explain the Diamond-Square algorithm and describe how a noise generator can be used to create map features such as forests or deserts. Finally we will demonstrate how we brought these algorithms together to generate maps and how we developed our secure peer-to-peer system.

A. Problem Space

We first assume that any peer-to-peer game will feature some sort of lobby that facilitates the creation of a game session and determines details such as map size, sea level, etc. As such, our procedures and analysis focus solely on the process of map generation and not on the details of establishing the initial network connection or agreeing upon match settings (i.e. size of map, ratio of landmass to water).

There are two styles of map generation that can be used in strategy games. The first is to generate the map before the game begins and the second is to generate the map during the game. Strategy games often utilize a “fog of war” where players discover the layout of the world throughout the game and we initially looked to a lazy generation method to preserve this gameplay element. We quickly realized that lazy generation had a large drawback that could have an even greater impact on the game. In order to fairly generate new map “tiles” (or regions), we knew the two players would have to communicate to ensure fairness. This could give the other player unneeded information

that could tip the game in her favor. For example, if Player Alice wanted to attack Player Bob from the flank in an area she had not previously explored, she would have to communicate with Bob to generate (or perhaps retrieve) that map tile, which would alert him that an enemy force was in the area. Though this problem is not proven to be intractable, we felt it smarter to generate the maps between players before gameplay began. Though both players would have perfect map knowledge, they would not be required to reveal sensitive information inadvertently. There is a potential third option as well that combines both approaches though it would limit the types of gameplay available. *Civilization* features a map type where all players start on a single continent and are encouraged to explore a "New World" continent with no pre-existing players (thus replicating colonialism). For such a map, it would be possible to generate the original continent before the game begins but allow players to generate the new continent in the lazy style. Though this method is promising, we opted to use traditional map generation algorithms (which are more broadly applicable) as described below.

B. Diamond-Square Algorithm

Fournier and Fussel developed the diamond-square algorithm to create natural-looking irregular objects and phenomena via stochastic processes [1]. Similar to a method known as factional Brownian motion, the algorithm generates a 2D or 3D surface by subdividing a square and randomly adjusting each new point. This process is repeated until the desired level of detail is achieved. The variation of this algorithm used in our map generator is described below. Figure 1 shows a labeled grid as might be utilized

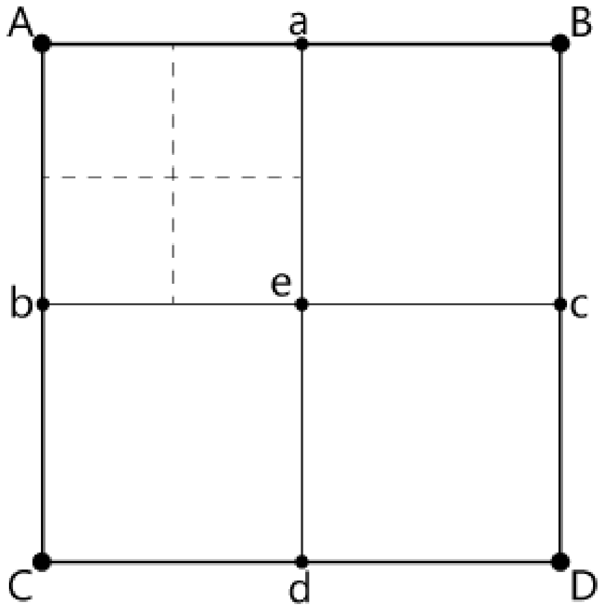

Figure 1: A sample grid where A-D represent given values and a-e represent values generated by the Diamond-Square algorithm

during the Diamond-Square algorithm. Given values for the four corner points (labeled above as A, B C and D), the "square" step of the algorithm will find coordinates for each midpoint value by averaging the corners to either side of it.

$$a = A + B / 2$$

$$b = A + C / 2$$

$$c = B + D / 2$$

$$d = C + D / 2$$

The algorithm then calculates the middle point, by averaging all four corners together and adding a random value multiplied by a variation value.

$$e = (A + B + C + D) / 4 + random * variation$$

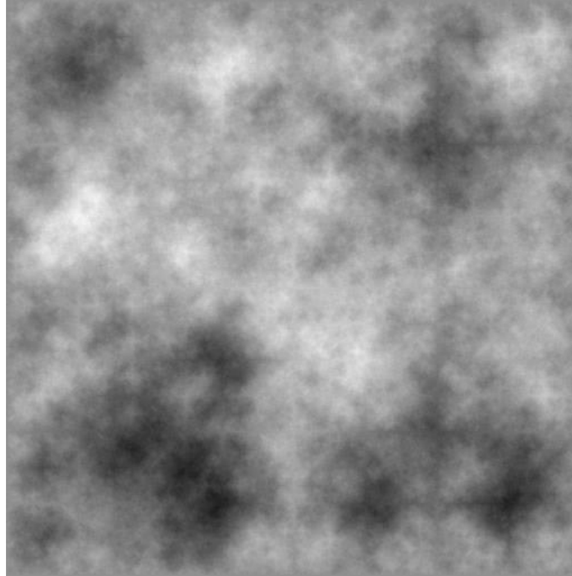

Figure 2: A sample fractal generated by the Diamond-Square algorithm

It then calls itself recursively on each of the following sub grids with a modified variance. Note that the change in variance is used to control the "noisiness" of the grid values.

Grid 1: A, a, b, e

Grid 2: a, B, e, c

Grid 3: b, e, c, d

Grid 4: e, c, d, D

The diamond step is an extra set of calculations that can be done by choosing the values generated at a, b, c, and d and using them to help generate the value at e. It is used to reduce the square-shaped artifacting that can be seen in some places in Figure 2. It can also be used to wrap the map around the edges seamlessly, though this requires an iterative approach to the algorithm instead of recursive. The algorithm can be set to

```
//Calculates five new midpoints
void sample(x1, y1, x2, y2, variance)
     //Ensure that the algorithm ceases once it has
          filled the map
     if (x2 - x1 > 1)
          //Retrieve the values of the corner points
          A = map[x1][y1]
          B = map[x2][y1]
          C = map[x1][y2]
          D = map[x2][y2]

          //Calculate the location of the new center
               point
          newX = x2 + x1 / 2
          newY = y2 + y1 /2

          //Calculate the new midpoints
          map[newX][y1] = (A + B)  / 2
          map[x1][newY] = (A + C) / 2
          map[newX][y2] = (C + D) / 2
          map[x2][newY] = (B + D) / 2
          map[newX][newY] = (A + B + C + D) / 4 +
               random * variance - variance/2)

          //Call on the four subgrids
          sample(x1, y1, newX, newY, variance/1.6)
          sample(newX, y1, x2, newY, variance/1.6)
          sample(x1, newY ,newX, y2, variance/1.6)
          sample(newX, newY, x2, y2, variance/1.6)
```

Figure 3: Pseudo-code for the Diamond-Square Algorithm's sample function

terminate after a set number of subdivisions or when the size of the sub grid reaches a point of no-consequence (i.e. an unnoticeable level of detail). We experimented with a number of variances and factors but settled upon an initial variance of .5 and a reduction factor of 1.6 during each subdivision. We found that maps with these qualities tended to have one primary continent with a few smaller islands around its periphery which we felt

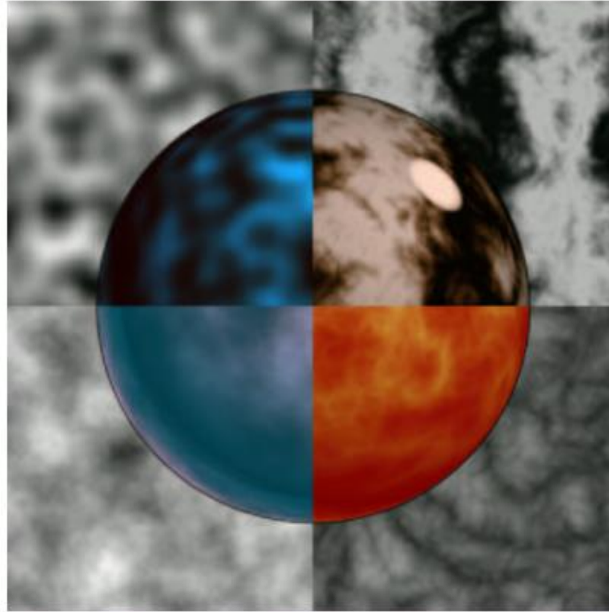

Figure 4: Four examples of Perlin Noise (2D and 3D) [2]

would make for a satisfactory play space. It terminates when the grid cannot be subdivided any further. Figure 2 shows a sample fractal map colored via grayscale. Square artifacting can also be noted in this diagram, particularly in the upper right corner. Pseudo-code for the core “sample” function can be seen in Figure 3.

We opted to not implement the diamond step of the algorithm in our program as we were more interested in the general ideas behind the algorithm than any specific improvements that could be made upon it. Furthermore, we assume a square map size of $2^n + 1$ as this simplifies the recursion process immensely. The algorithm can be modified to run on other map sizes.

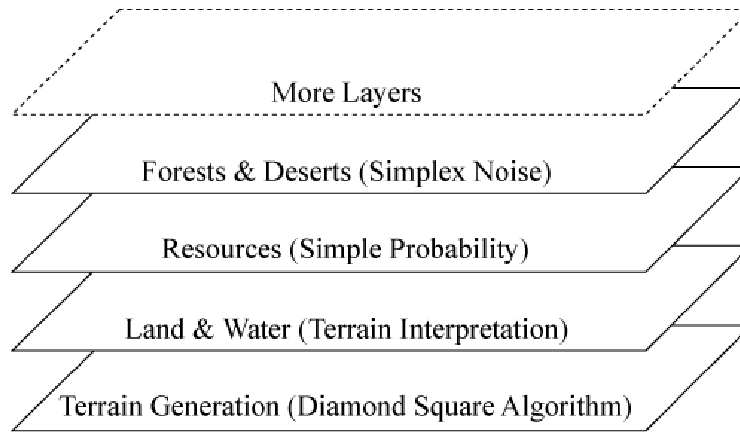

Figure 5: Layers of the map generator

C. Simplex Noise Generation

Simplex Noise is a refinement of the classic Perlin noise presented by Ken Perlin and is often used in computer graphics problems [2]. These noise functions give the appearance of randomness by generating a pseudo-random "surface" (noise can be generated in multiple dimensions). Given a coordinate, the noise function returns a "noisy" value which can then be used to generate textures or terrains.

Simplex noise is a process by which multiple noise functions (with varying amplitudes and frequencies) are combined. This allows for a finer degree of control over what form the noise takes. Figure 4 shows four sample textures as presented in Perlin's paper on Simplex noise.

D. Map Generation

Our map generator has four "layers," each of which is responsible for building one portion of the map (see Figure 5). The algorithm design was inspired by the scripting

for *Civilization V* maps which seem to use combinations of distinct functions to build unique maps (based off a user's preference for rainfall, sea level, etc.).

The first layer is responsible for generating a grid of double values via the Diamond-Square algorithm. Each corner was preset to zero which ensured the map was bounded by water (i.e. a connecting ocean). This would also allow the map to be trivially wrapped should the user desire it. A brief pass through the grid would calculate the average, maximum and minimum values which would be used during the next layer.

The Land and Water layer is responsible for converting the grid of double values into a grid of character values with each cell being either "land" or "water". This is done by comparing each double against some factor of the mean; values that are less become water, values that are greater become land. We converted all values less than the mean multiplied by 1.5 to water as we found this gave us the best land to water ratio. This value could be adjusted in either direction to create a world with more or less water, depending on user preference. We chose not to use the Diamond-Square values for mountainous terrain as we found that it did not generate the sort of mountain ranges one would expect. Instead, we imagined mountains would simply be generated by their own specific algorithm.

The third layer was perhaps the simplest; we simply generate a random value for each point in the grid and, if it is above some threshold, we designate that cell as an undefined game "resource."

The final layer implemented trees and deserts via Simplex noise. Whereas resources had no natural need for any sort of grouping, both forests and deserts are

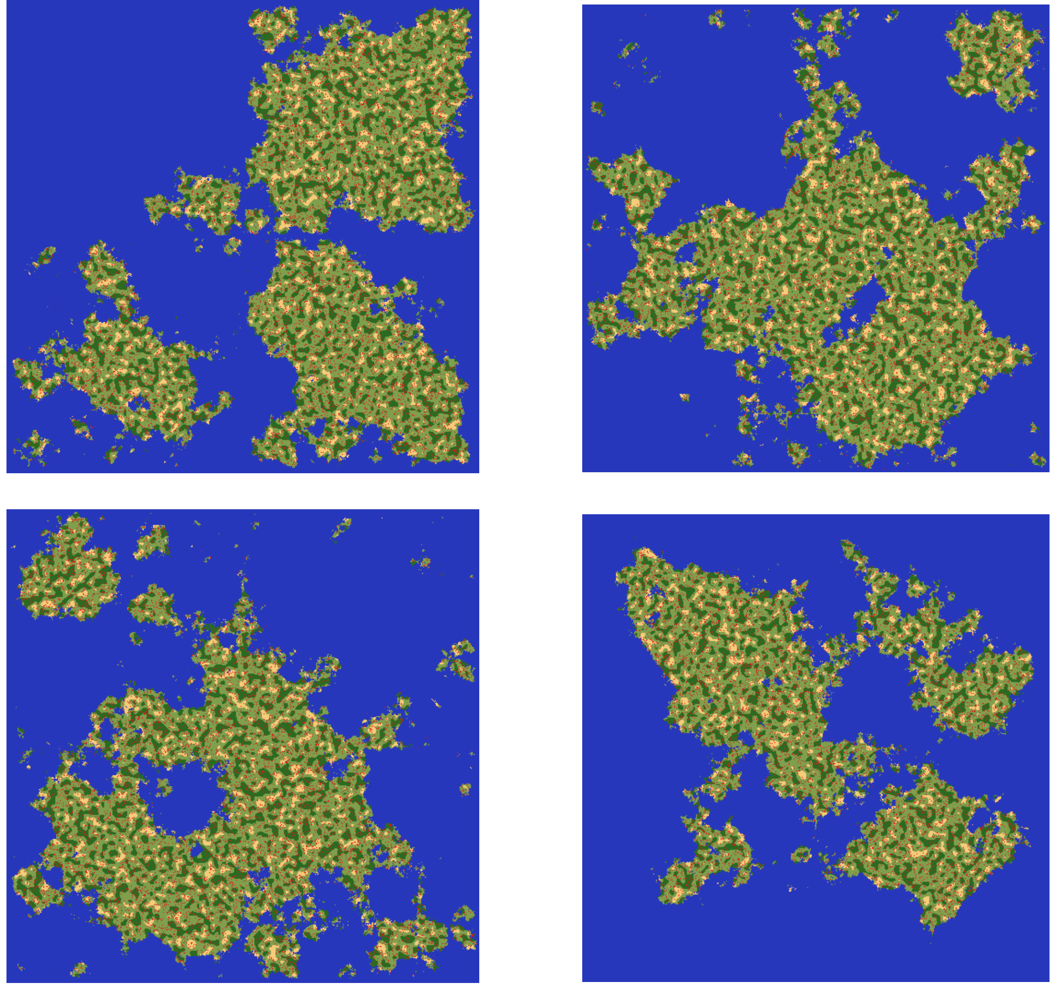

Figure 6: Four sample maps generated by our program

expected to be distinct geographical regions. Our solution was to use Simplex noise to generate these smaller regions though other algorithms could be used to the same effect. We briefly attempted a more complicated algorithm where two noise grids represented humidity and vegetation. We hoped to use a comparison of the two to generate forests and deserts (for example, forests would be found in locations with high humidity and high vegetation) but this approach proved less effective in practice.

Figure 6 shows four separate maps as generated by our four layers. Additional layers could add features such as mountains, ice or polar regions, unique resources (with varying dispersions) or lakes and rivers. The algorithms could also be tweaked to generate less world-like maps (for game such as *StarCraft* which take place on a more "planar" battlefield) by interpreting the original height points (as generated by the Diamond-Square algorithm) as mountains and valleys instead of land and water. In games like *Civilization*, many of the decisions we made about cut-off points (such as how much of the planet's surface is water or how much of the landmass is covered by forests) would be exposed as levers for players to manipulate.

E. Peer-to-Peer Map Generation

The final step in our work, and the most important, was to investigate how we could use these algorithms to build maps in a peer-to-peer game. Our goal is to provide a series of steps such that no player can cheat or otherwise impact the creation of the map without the other player being aware. The key was to have the players create a fair seed value which would then be used in the generation of random numbers for the map. If both players are playing fairly (and using unmodified game software), then the seed will be provided to identical random number generators and the maps generated will be identical.

If one player intends to cheat, however, they have one major avenue of attack: they could attempt to influence the seed itself. For this option to work, they must be able to predict how the seed affects the random number generator's output (so that they can "select" a favorable map based on a given seed) and they must be able to unfairly affect

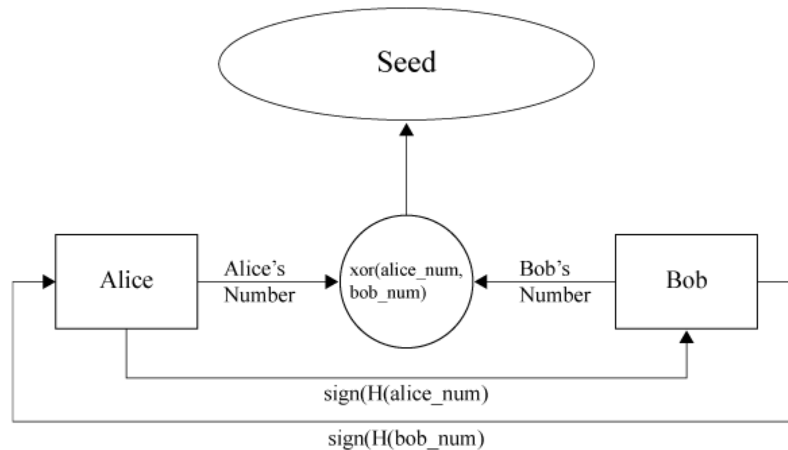

Figure 7: The process by which a fair, random seed is generated by Alice and Bob

the seed value. The process for securely generating a random seed between two parties, Alice and Bob, is as follows.

1. Alice and Bob randomly generate private numbers, which we call I_A and I_B respectively.
2. Alice and Bob sign their private numbers. Recall that a digital signature is an encrypted cryptographically secure hash of a message, i.e., $E_A(H(I_A))$ and $E_B(H(I_B))$
3. Alice and Bob transmit this digital signature to each other. Note that this message does not contain their private numbers, simply the hash of those numbers.
4. Once Alice and Bob have received the signature, they transmit I_A and I_B to each other.

5. Alice and Bob hash each other's public numbers and compare them to the original hash they received. If these hashes match, then Alice and Bob know that the other has been honest in their choice of seed contribution.
6. Alice and Bob then XOR I_A and I_B together to create a new random value, k_A which can be used as a random seed.

Because both players must commit to their number before seeing each other's public number, they cannot predict what the final XOR'd value will be. Once Alice has finished generating the map locally, she hashes the map and sends it to Bob. When she receives Bob's hash, she compares it to her own. If they do not match, then the game is aborted as the two players do not have identical maps. Otherwise, gameplay may begin.

A malicious player may possibly generate the correct map (and thus pass the checks listed above) and then attempt to play on a different map entirely. We assume that any peer-to-peer game will provide basic cheat protection (such as ensuring that each player is playing by the game rules) which would expose a scenario such as this (e.g., the cheating player would appear to be breaking game rules to a non-cheating player on a different map).

CHAPTER FOUR: ANALYSIS

A. Experiment Setup

Our program was developed as a proof-of-concept to showcase the methodology used, not with the intent to use the maps in any specific game. We developed the program in Java using Eclipse though we also used Processing as a visualization tool. We used default Java libraries (specifically the Random and Security libraries) though others could easily be used in place of these (either due to preference or for security reasons).
We used the TCP protocol for our networking over UDP due to the reliability guarantees that TCP provides. Our encryption utilized the SHA-512 encryption hash (part of the SHA-2 family of hash functions) which is proven secure enough for our needs (i.e. extremely unlikely for an adversary to derive the hashed value within a reasonable time frame).

All of our testing was done via two machines running Debian 4.6.3-14 with an Intel(R) Xeon E5405 (running at 2.00GHz) and 24 GB of RAM. Both computers were connected to the same network which does simplify some of the real world constraints our program might face. More information on TCP's performance in real world network conditions (including issues of Network Address Translation) can be found at [9] and [10]. All test values were averages recorded over 1000 trials.

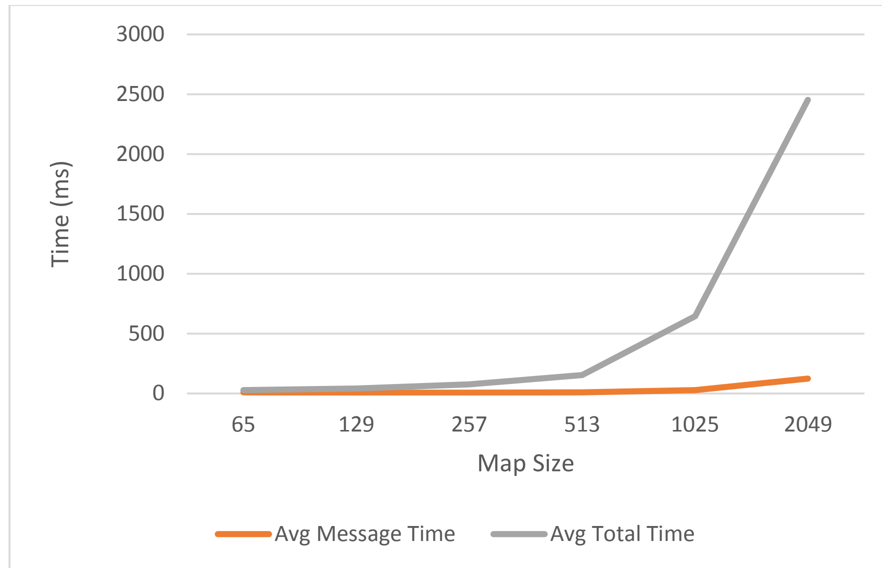

Figure 8: Average time required to complete the program as a factor of map size (graph)

B. Results

We begin our analysis by summarizing the overall cost to securely generate the seed and confirm the resulting map. Recall that the seed generation requires each player to send and receive two messages (the first is the hash of their seed contribution, the second is the actual number). The map comparison only requires one message, thus each player must only send and receive a total of six networked messages (send and receive). This number does not change based on the size of the map but would change should the procedure be adapted for *n* players.

Given *n* players, each player must send and receive 3 messages to each other player so that they can XOR all *n* seed contribution values together. This will take $3(n - 1)$ send messages and $3(n - 1)$ receive messages. In total, each player sends and receives $O(n)$ messages though the method will generate $O(n^2)$ on the system.

Map Size	65	129	257	513	1025	2049
Message Time (ms)	7	5	5	8	26	123
Total Time (ms)	28	40	74	153	644	2451

Figure 9: Average time required to complete the program as a factor of map size (table)

Figure 8 shows the average time taken for our program to complete, not including the time spent waiting for or connecting to the other player. The "Total Time" measure includes the time spent to construct and send messages, create the map and write that map to a file. The "Message Time" measure only includes the time spent to create, send and receive messages. Figure 9 shows the same data in a table format.

As expected, increasing the map size quadratically has a similar effect on the time taken to calculate the map. The required map size for a game depends both on the number players (more players usually requires a larger map) and the required level of detail for a basic map. For example, a two player *Civilization V* (referred to as a "duel" map) is a paltry 40 x 25 tiles. Even the largest map (for 12 players) is only sized at 128 x 80 tiles. Our interpreter creates maps with a pixel as a base measurement and thus tend towards the larger size (the maps displayed in Figure 6 are 513 x 513).

There is also a noticeable increase in the time spent sending and receiving messages, especially on larger map sizes. This disparity is due to the time it takes to create the hash of the map (which must be converted into a byte array first). Even in these cases, however, it is clear that the time it takes to create the map itself is most responsible for the overall running time.

CHAPTER FIVE: CONCLUSION AND FUTURE WORK

In conclusion, we have demonstrated that two players can exchange a random seed and generate a random map without either player having the ability to influence the outcome. It is further worth noting that the methods detailed above could also be applied to other genres of games that rely upon random map generation. Mojang's Minecraft, for example, relies exclusively upon a single seed value to generate worlds.

This paper provides the basis for a full peer-to-peer game but further research is needed into how to secure gameplay elements to prevent or prove when one player is cheating. Many of the concepts in this paper (and some of the others that have been cited) could be used to this effect. The ideal end goal would be a game that can be played from start to finish in a completely distributed fashion with all players knowing that the game is fair.

BIBLIOGRAPHY

[1] A. Fournier, D. Fussell, L. Carpenter, “Computer Rendering of Stochastic Models,” Communications of the ACM, vol. 25, no. 6, June 1982.

[2] K. Perlin, “Noise Hardware,” In Real-Time Shading SIGGRAPH Course Notes, 2001.

[3] C. GauthierDickey, and C. Ritzdorf, “Secure Peer-to-Peer Trading for Multiplayer Games,” ACM NetGames, November 2012.

[4] D. Pittman, C. GauthierDickey, “Match+Guardian: A Secure Peer-to-Peer Trading Card Game Protocol,” “Multimedia Systems, vol. 19, issue 3, pages 303-314, 2013.

[5] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, “Procedural Content Generation for Games: A Survey,” ACM Transactions on Multimedia Computing, Communications and Applications, vol. 9, no. 1, February 2013.

[6] W. Raffe, F. Zambetta, and X. Li, “A Survey of Procedural Terrain Generation Techniques using Evolutionary Algorithms,” WCCI 2012 IEEE World Congress on Computation Intelligence, June 2012.

[7] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. Yannakakis, “Multiobjective Exploration of the StarCraft Map Space,” 2010 IEEE Conference on Computation Ingelligence and Games, 2010.

[8] J. Noghani, F. Liarokapis, and E. Falk Anderson, “Randomly Generated 3D Environments for Serious Games,” 2010 Second International Conference on Games and Virtual Worlds for Serious Applications, 2010.

[9] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, “Modeling TCP Throughput: A Simple Model and its Empirical Validation,” SIGCOMM, 1998.

[10] B. Ford, and P. Srisuresh, “Peer-to-Peer Communication Across Network Address Translators,” 2005 USENIX Annual Technical Conference, 2005.

[11] F. Ji, and W. Deyong, “Design and Implementation of 3-D Terrain Generation Module in Game,” 2010 3rd International Conference on Advanced Computer Theory and Engineering, 2010.

[12] D. Ashlock, C. Lee, and C. McGuiness, “Simultaneous Dual Level Creation for Games,” IEEE Computational intelligence Magazine, pages 25-37, May 2011.

[13] J. Valls-Vargas, S. Ontañón, and J. Zhu, “Towards Story-Based Content Generation: From Plot-Points to Maps,” IEEE, 2013.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A Scalable Content-Addressable Network,” SIGCOMM, 2001.

[15] I. Stoica, R. Morris, D. Krager, M. Frans Kaashoek, and H. Balarkishnan, “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications,” SIGCOMM, 2001.

APPENDIX: SOURCE CODE

```
/*
 * Stephen Rice
 * P2P Map Generation
 * Created 4/10/2014
 *
 * Main.java: Takes a player number, port number (self), IP Address,
port number (other), map size
 * and seed contribution. Note that the other player's IP address and
port number are not used if
 * "1" is entered as the player number
 *
 */

import java.io.BufferedWriter;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;

public class MapGenP2P {
      public static void main(String[] args) {

            //Network Connections
            ServerSocket server = null;
            Socket client = null;
            OutputStream out = null;
            DataOutputStream dos = null;
            InputStream in = null;
            DataInputStream dis = null;

            //Seed Variables
            MessageDigest md = null;
            long SMC_Seed = 0;

            //Expected order of args: Player Number, My Port, IP, Port,
Map Size, Seed Contribution
            if(args.length < 5) {
                  System.err.println("Error: Expecting Six arguments:
                  Player Number, Local Port,  IP Address, Port, MapSize,
                  Seed Contribution");
                  System.exit(-1);
            }

            //Save user values
            int player = Integer.parseInt(args[0]);
            int myPort = Integer.parseInt(args[1]);
            String ip = args[2];
            int port = Integer.parseInt(args[3]);
```

```
            int mapSize = Integer.parseInt(args[4]);
            long seed = Long.parseLong(args[5]);

            //Check validity of provided map size
            if((mapSize - 1) % 4 != 0) {
                  System.out.println("Map Size must be of the form (2^n)
+ 1");
                  System.exit(-1);
            }

            //If Player 1, create a server and wait
            if(player == 1) {
                  //Initiate network handshake with other client
                  try {
                        server = new ServerSocket(myPort);

                        //Wait for a client to connect
                        while(true) {
                              client = server.accept();
                              System.out.println("Connected to other
player");
                              break;
                        }
                  }
                  catch (IOException e) {
                        System.err.println("Exception: Couldn't bind to
port");
                        System.exit(-1);
                  }
            }
            else {
                  //Connect to the server
                   try {
                        client = new Socket(ip, port);
                        System.out.println("Connected to player 1");
                   }
                   catch (UnknownHostException e) {
                         System.err.println("Exception: Unknown Host");
                         System.exit(-1);
                   }
                   catch (IOException e) {
                        System.err.println("Exception: Couldn't create
connection to server");
                        System.exit(-1);
                   }
            }

            //Bind inputs/outputs
            try {
                  out = client.getOutputStream();
                  dos = new DataOutputStream(out);

                  in = client.getInputStream();
                  dis = new DataInputStream(in);

            }
            catch(IOException e) {
                  System.err.println("Exception: Couldn't capture
Input/Output Streams");
            }

            //Create Hash of seed
```

```
            byte[] myHash = new byte[Long.SIZE];
            byte[] theirHash = new byte[Long.SIZE];
            long theirSeed = 0;
            try {
                  //Create a message digest with SHA-512
                  md = MessageDigest.getInstance("SHA-512");

                  //Get the bytes of the Long, then encode
                  md.update(Long.toString(seed).getBytes());
                  myHash = md.digest();

                  //Send the hash to the other player
                  dos.write(myHash);

                  //Wait for reply from other user
                  while(true) {
                        if(dis.available() > 0) {
                              dis.readFully(theirHash);
                              break;
                        }
                  }

                  //Send the unhashed integer
                  dos.writeLong(seed);

                  //Receive their unhashed integer
                  while(true) {
                        if(dis.available() > 0) {
                              theirSeed = dis.readLong();
                              break;
                        }
                  }

                  //Check the seed
                  md.update(Long.toString(theirSeed).getBytes());

                  byte[] testHash = new byte[Long.SIZE];
                  byte[] temp = md.digest();

                  //Copy into a new array to ensure length parity
                  for(int i = 0; i < temp.length; i++) {
                        testHash[i] = temp[i];
                  }

                  //Check if the two hashes are identical
                  if(!Arrays.equals(testHash, theirHash)) {
                        System.out.println("Their hashed seed does not
match the sent seed");
                        System.exit(-1);
                  }

                  //XOR the seed and save it
                  SMC_Seed = seed ^ theirSeed;

            }
            catch (NoSuchAlgorithmException e1) {
                  System.out.println("Exception: Cannot find specified
hash");
            }
            catch (IOException e) {
                  System.out.println("Error reading message");
            }
```

```
			//Agree on seed, set seed
			RandomWrapper.setSeed(SMC_Seed);
			SimplexNoise.seedP();

			//Generate the Map and save it in a new variable
			MapGenerator mapGen = new MapGenerator(mapSize);
			char[][] map = mapGen.getMap(true);

			//Convert bytes to map
			byte[] bMap = toBytes(map);
			md.update(bMap);
			byte[] myHashMap = md.digest();
			byte[] theirHashMap = new byte[Long.SIZE];

			try {
				//Send hashed map to client
				dos.write(myHashMap);

				//Wait for reply
				while(true) {
					if(dis.available() > 0)
					{
						dis.readFully(theirHashMap);
						break;
					}
				}

				//Copy my hash into a max size array
				byte[] testHash = new byte[Long.SIZE];

				for(int i = 0; i < myHashMap.length; i++) {
					testHash[i] = myHashMap[i];
				}

				//Compare
				if(!Arrays.equals(testHash, theirHashMap)) {
					System.err.println("The maps do not match.");
					System.exit(-1);
				}

			}
			catch (IOException e) {
				System.out.println("Exception: Comparing hashes");
			}

			//Write the map to a file
			writeMapToFile(map);

			System.out.println("A map was succesfully generated with the
other player");

			System.exit(1);
		}

		public static void writeMapToFile(char[][] map) {
			//Write to file
			try {
				System.out.println("Writing map to file");
				BufferedWriter writer = new BufferedWriter(new
FileWriter("map.txt"));
				for(int i = 0; i < map.length; i++) {
```

```
                        for(int j = 0; j < map[i].length; j++)
                        {
                              writer.write(map[j][i] + " ");
                        }
                        writer.write("\n");
                  }

                  writer.close();
                  System.out.println("File written successfully");
            }
            catch (IOException e) {
                  System.err.println("Problem writing to file");
            }
      }

      //Convert a character map into a byte array
      public static byte[] toBytes(char[][] map) {
            byte[] bytes = new byte[map.length*map.length];

            for(int i = 0; i < map.length; i++) {
                  for(int j = 0; j < map[i].length; j++) {
                        bytes[i + j * map.length] = (byte) map[j][i];
                  }
            }

            return bytes;
      }
}

/*
 * Stephen Rice
 * P2P Map Generation
 * Created 4/10/2014
 *
 * MapGenerator.java: Given a map size, generates a new random map
 */

public class MapGenerator {
      //Stores the map
      private double[][] map;
      private char[][] cMap;

      //Size of the map
      private int mapDim;

      //Map statistics
      private double largest, smallest, mean;

      //Map Created
      private boolean mapCreated;

      public MapGenerator(int inSize) {
            //Save the provided map size (assume size is checked prior)
            mapDim = inSize;

            mapCreated = false;
      }

      //Generate the Map
      private void generateMap() {
            System.out.println("Map Generator is starting");
            //Create the emtpy maps
```

```
			map = new double[mapDim][mapDim];
			cMap = new char[mapDim][mapDim];

			//Init corners to 0
			map[0][0] = 0;
			map[mapDim -1][0] = 0;
			map[0][mapDim -1] = 0;
			map[mapDim - 1][mapDim - 1] = 0;

			//Call recursively to sample, providing the 4 corners and a
variance of .5
			sample(0,0, mapDim - 1, mapDim - 1, .5);

			//Calculate the means and maxes
			calculateStatistics();

			//Begin populating the character map
			LandAndWater(1.5);
			Resources();
			TreesAndDesert();

			mapCreated = true;
			System.out.println("Map Generator is finished");
		}

		//Recursive Sample: Takes two corners (A, D) to determine the sub
grid
		private void sample(int x1, int y1, int x2, int y2, double random)
{
			/*
			 Capitals: Original corner points
			 Lower: New generated points

			 A - a - B
			 b - c - e
			 C - d - D

			 */

			//Check to ensure that midpoints exist
			if(x2 - x1 > 1) {
				//Get height maps
				double A = map[x1][y1];
				double B = map[x2][y1];
				double C = map[x1][y2];
				double D = map[x2][y2];

				int newX = (x2+x1) / 2;
				int newY = (y2+y1) / 2;

				//Determine a,b,c,d,e

				map[newX][y1] = (A + B) / 2; //a
				map[x1][newY] = (A + C) / 2; //b

				//Ensure first point is always positive (avoids lake
in middle)
				if(x1 == 0 && x2 == mapDim - 1) {
					map[newX][newY] = (A + B + C + D) / 4 +
(RandomWrapper.getRandom() * random/2  ); //c
				}
				else {
```

```
					map[newX][newY] = (A + B + C + D) / 4 +
					(RandomWrapper.getRandom() * random - random/2
					); //c
				}

				map[newX][y2] = (C + D) / 2; //d
				map[x2][newY] = (B + D) / 2; //e

				//Recursively call sub quadrants
				sample(x1,y1,newX, newY, random/1.60); //A-c
				sample(newX, y1, x2, newY,  random/1.6); // a-e
				sample(x1, newY, newX, y2, random/1.6); // b-d
				sample(newX, newY, x2, y2, random/1.6);// c-D
			}
		}

		//Calculate the largest, smallest and mean values in the map
		private void calculateStatistics() {
			double large = Integer.MIN_VALUE;
			double small = Integer.MAX_VALUE;
			double sum = 0;

			for(int i = 0; i < mapDim; i++) {
				for(int j = 0; j < mapDim; j++) {
					//Check Largest
					if(map[j][i] > largest) {
						small = map[j][i];
					}

					if(map[j][i] < smallest) {
						large = map[j][i];
					}

					sum += map[j][i];
				}
			}

			largest = large;
			smallest = small;
			mean = sum / (mapDim * mapDim);
		}


		/*These functions scan through the map and generate land, water,
forests, deserts
		 * and resources
		 *
		 * KEY
		 * W: Water (< mean)
		 * G: Grasslands (default land)
		 * F: Forest
		 * D: Desert
		 */

		//Based on each heighpoint's value, compare to the mean and
		//set land or water. The meanRatio will determine the amount of
land vs. amount of water
		private void LandAndWater(double meanRatio) {
			for(int i = 0; i < map.length; i++) {
				for(int j = 0; j < map[i].length; j++) {
					//Water
					if(map[j][i] < mean * meanRatio) {
```

```
				cMap[j][i] = 'W';
			}

			//Grasslands
			else {
				cMap[j][i] = 'G';
			}
		}
	}
}

	//Generates a simplex noise map and creates trees and deserts
	//Note: Simplex noise is very simple and further refinements to
the parameters (or another nosie generator)
	//may provide better results
	private void TreesAndDesert() {
		//Generate the simplex noise seed values
		//SimplexNoise.seedP();

		//Create the simplex noise object
		SimplexNoise sng = new SimplexNoise();
		float[][] humidity = sng.generateOctavedSimplexNoise(mapDim,
mapDim, 2, 2.4f, .04f);

		//Set up desert and trees
		for(int i = 0; i < mapDim; i++) {
			for(int j = 0; j < mapDim; j++) {
				if(humidity[j][i] > .8 && cMap[j][i] == 'G') {
					cMap[j][i] = 'F';
				}

				if(humidity[j][i] < -1.5 && cMap[j][i] == 'G') {
					cMap[j][i] = 'D';
				}
			}
		}
	}

	//Generic Resource Generation. Full map should use specific
resource generators
	//to create the proper distribution
	//Note: This function is also responsible for many of the random
numbers that are generated
	private void Resources() {
		for(int i = 0; i < mapDim; i++) {
			for(int j = 0; j < mapDim; j++) {
				if(RandomWrapper.getRandom() > .9 && cMap[j][i]
== 'G') {
					cMap[j][i] = 'R';
				}
			}
		}
	}

	//Return the map (or generate it if it hasn't been done)
	public char[][] getMap() {
		//If the map has not been created, make it.
		if(!mapCreated) {
			generateMap();
			return cMap;
		}
```

```
		//Otherwise return the previously generated map
		return cMap;
	}

	//Return the map, allow the user to specify the creation of a new
map, even if the old one has been created
	public char[][] getMap(boolean newMap) {
		//If the user wants a new map, do that no matter what
		if(newMap) {
			generateMap();
			return cMap;
		}

		//Otherwise check if the "old" map exists
		if(!mapCreated) {
			generateMap();
			return cMap;
		}

		//Return the old map
		return cMap;
	}
}

/*
 * Stephen Rice
 * P2P Map Generation
 * Created 4/10/2014
 *
 *RandomWrapper.java: Wraps an RNG and allows a seed to be set
 */


import java.util.Random;

public class RandomWrapper {
	private static long seed;
	private static Random random;

	//Save and set a new seed in the RNG
	public static void setSeed(long inSeed) {
		seed = inSeed;
		random = new Random(seed);
	}

	//Return the current seed
	public static long getSeed() {
		return seed;
	}

	//Return the next random number
	public static double getRandom() {
		return random.nextDouble();
	}
}

/*
 * Stephen Rice
 * P2P Map Generation
 * Created 4/10/2014
 *
```

```
 * SimplexNoise.java: Simple Simplex Noise generator put together from a
variety of Internet sources
 * The "geenerateDocteabedSimplex Noise" was taken from a post on
http://www.java-gaming.org/topics/generating-2d-perlin-
noise/31637/view.html
 * The noise function itself was taken from
http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
 *
 * Note: seedP() must be called before calling noise function
 */

public class SimplexNoise {

      private static int grad3[][] = {{1,1,0},{-1,1,0},{1,-1,0},{-1,-
1,0},
             {1,0,1},{-1,0,1},{1,0,-1},{-1,0,-1},
             {0,1,1},{0,-1,1},{0,1,-1},{0,-1,-1}};

      public static int p[] = new int[256];
      static{
             for(int i = 0; i < 256; i++) {
                   p[i] = (int) (RandomWrapper.getRandom() * 256);
             }
      }
      private static int perm[] = new int[512];

      // This method is a *lot* faster than using (int)Math.floor(x)
       private static int fastfloor(double x) {
       return x>0 ? (int)x : (int)x-1;
       }

       public static void seedP() {
             for(int i = 0; i < 256; i++)
             {
                   p[i] = (int) (RandomWrapper.getRandom() * 256);
             }

             for(int i=0; i<512; i++) perm[i]=p[i & 255];
       }

       private static double dot(int g[], double x, double y) {
             return g[0]*x + g[1]*y; }
             private static double dot(int g[], double x, double y,
double z) {
             return g[0]*x + g[1]*y + g[2]*z; }
             private static double dot(int g[], double x, double y,
double z, double w) {
             return g[0]*x + g[1]*y + g[2]*z + g[3]*w; }


      public float[][] generateOctavedSimplexNoise(int width, int
height, int octaves, float roughness, float scale){
            float[][] totalNoise = new float[width][height];
             float layerFrequency = scale;
             float layerWeight = 1;
             float weightSum = 0;

             for (int octave = 0; octave < octaves; octave++) {
                //Calculate single layer/octave of simplex noise, then
add it to total noise
                for(int x = 0; x < width; x++){
                   for(int y = 0; y < height; y++){
```

```
                    totalNoise[x][y] += (float) noise(x *
layerFrequency,y * layerFrequency) * layerWeight;
                   }
                }

                //Increase variables with each incrementing octave
                 layerFrequency *= 2;
                 weightSum += layerWeight;
                 layerWeight *= roughness;

             }
             return totalNoise;
         }

      public static double noise(double xin, double yin) {
             double n0, n1, n2; // Noise contributions from the three
corners
             // Skew the input space to determine which simplex cell
we're in
             final double F2 = 0.5*(Math.sqrt(3.0)-1.0);
             double s = (xin+yin)*F2; // Hairy factor for 2D
             int i = fastfloor(xin+s);
             int j = fastfloor(yin+s);
             final double G2 = (3.0-Math.sqrt(3.0))/6.0;
             double t = (i+j)*G2;
             double X0 = i-t; // Unskew the cell origin back to (x,y)
space
             double Y0 = j-t;
             double x0 = xin-X0; // The x,y distances from the cell
origin
             double y0 = yin-Y0;
             // For the 2D case, the simplex shape is an equilateral
triangle.
             // Determine which simplex we are in.
             int i1, j1; // Offsets for second (middle) corner of
simplex in (i,j) coords
             if(x0>y0) {i1=1; j1=0;} // lower triangle, XY order: (0,0)-
>(1,0)->(1,1)
             else {i1=0; j1=1;} // upper triangle, YX order: (0,0)-
>(0,1)->(1,1)
             // A step of (1,0) in (i,j) means a step of (1-c,-c) in
(x,y), and
             // a step of (0,1) in (i,j) means a step of (-c,1-c) in
(x,y), where
             // c = (3-sqrt(3))/6
             double x1 = x0 - i1 + G2; // Offsets for middle corner in
(x,y) unskewed coords
             double y1 = y0 - j1 + G2;
             double x2 = x0 - 1.0 + 2.0 * G2; // Offsets for last corner
in (x,y) unskewed coords
             double y2 = y0 - 1.0 + 2.0 * G2;
             // Work out the hashed gradient indices of the three
simplex corners
             int ii = i & 255;
             int jj = j & 255;
             int gi0 = perm[ii+perm[jj]] % 12;
             int gi1 = perm[ii+i1+perm[jj+j1]] % 12;
             int gi2 = perm[ii+1+perm[jj+1]] % 12;
             // Calculate the contribution from the three corners
             double t0 = 0.5 - x0*x0-y0*y0;
             if(t0<0) n0 = 0.0;
             else {
```

```
            t0 *= t0;
            n0 = t0 * t0 * dot(grad3[gi0], x0, y0); // (x,y) of grad3
used for 2D gradient
            }
            double t1 = 0.5 - x1*x1-y1*y1;
            if(t1<0) n1 = 0.0;
            else {
            t1 *= t1;
            n1 = t1 * t1 * dot(grad3[gi1], x1, y1);
            } double t2 = 0.5 - x2*x2-y2*y2;
            if(t2<0) n2 = 0.0;
            else {
            t2 *= t2;
            n2 = t2 * t2 * dot(grad3[gi2], x2, y2);
            }
            // Add contributions from each corner to get the final
noise value.
            // The result is scaled to return values in the interval [-
1,1].
            return 70.0 * (n0 + n1 + n2);
            }
}
```