Procedural Generation in Game design

Evan Barry – T00202376

Final Year Project

B.Sc. in Computing With Games Development

**Table of Contents**

[Chapter 1: Introduction 2](#_Toc54703847)

[Chapter 2: Procedural Generation 3](#_Toc54703848)

[2.1 Introduction 3](#_Toc54703849)

[2.2 History of Procedural Generation 3](#_Toc54703850)

[Chapter 3: Procedural Generation Algorithms 6](#_Toc54703851)

[3.1 Introduction 6](#_Toc54703852)

[3.2 Noise Algorithms 6](#_Toc54703853)

[3.2.1 Value Noise 7](#_Toc54703854)

[3.2.2 Cubic Noise 7](#_Toc54703855)

[3.2.3 Perlin Noise 7](#_Toc54703856)

[3.2.4 Simplex Noise 7](#_Toc54703857)

[3.2.5 Diamond Square 7](#_Toc54703858)

[3.3 Procedural Level Generation Pipeline 7](#_Toc54703859)

[3.3.1 Base Graph 7](#_Toc54703860)

[3.3.2 Minimum Spanning Tree 7](#_Toc54703861)

[3.3.3 Level Graph and Merging 7](#_Toc54703862)

[3.3.4 Geometric Representation 7](#_Toc54703863)

[3.4 Voronoi Diagram 7](#_Toc54703864)

[3.5 Fractals 7](#_Toc54703865)

Chapter 1: Introduction

Leave until end

Chapter 2: Procedural Generation

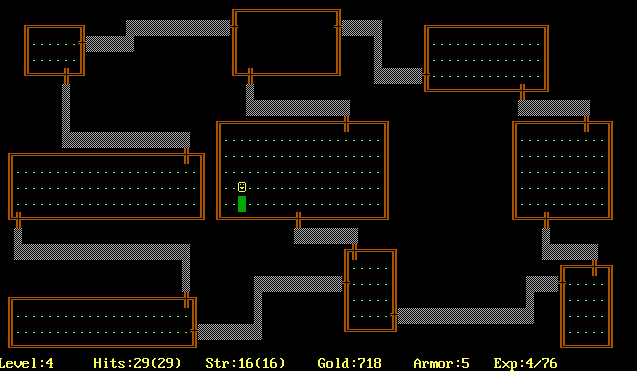
2.1 Introduction

Procedural Generation is a method by which data is created algorithmically instead of manually. Procedural Generation is usually used for the creation of video game or animated movie content, such as 3D objects, landscapes, and character designs. Procedural Generation, in the video game industry, has become more widely used as the production of game content has grown to the point that it has become a bottleneck in companies’ schedules and budgets (insert reference to A. Amato here). Many companies have resorted to procedural generation to produce their content in an attempt to reduce their workload. Utilising Procedural Generation to develop content for games ,reduces human workload as it uses algorithms that requires limited or no human contribution to produce content(Lucas et al., 2013).

2.2 History of Procedural Generation

The earliest uses of procedural generation in video games can be found in the games “Beneath Apple Manor”(1978) and “Rogue”(1980), the game which was used as the basis of a new genre, “Roguelike”. A roguelike game derives concepts from “Dungeons and Dragons”, where the players takes controller of a character that has customizable features, such as class, race, and gender, and the character can be assigned attribute points and skills as the player sees fit.

The gameplay of a roguelike is considered a “dungeon crawl”, where the player moves their character through a dungeon to find treasure, items, and weapons, and encounter enemies. As these games were released in the late 70s and early 80s, they were limited by the hardware available at the time. Rogue was released for 8-bit systems such as the Commodore 64 and the Atari 8-bit home computer series. These systems only had 64 kilobytes of RAM, so storing fully detailed dungeons was not a possibility for the developers of Rogue. To overcome this problem, the developers used procedural generation algorithms to generate the dungeon at the start of an adventure. The dungeon took the form of 3-by-3 tic tac toe grid, with a room occupying a space in the grid and hallways connecting the rooms (see Diag. 2.1). This procedural generation of dungeons allowed for Rogue to be played on the 64 kilobyte systems and allowed for a unique playthrough each time as treasure, items, weapons, and enemies were randomly placed on the grid.



Diag. 2.1: Dungeon in Rogue. The Rooms are aligned in 3x3 pattern with hallways connecting most rooms

As computer systems evolved, so did the application of procedural generation for game content. The roguelike genre continued to grow throughout the 1980’s, producing classic games such as Moria and NetHack. As game consoles were released in the late 1980’s, the number of roguelikes began to dwindle as the new game consoles had external game cartridges that stored the game data and were able to run pre-made worlds. The most notable of these consoles was the Nintendo Entertainment System (NES). It was not until the mid-2000’s that procedural generation in game design became popular again. The resurgence in the use of procedural generation coincides with independent(indie) game development. One of the most popular indie games of all-time is “Spelunky”. It is a roguelike platformer, where you control a character to explore a cave by jumping across platforms (see Diag. 2.2). The caves are entirely procedurally generated, and all platforms and items are procedurally generated to provide a unique experience with each playthrough.

A close up of a sign

Description automatically generated

Diag. 2.2: Full view of a "Mine" cave from Spelunky

As game consoles have further evolved, the need for procedural generation has diminished as the more modern game consoles are very powerful and have up to 1TB of storage space and 8GB of RAM. Procedurally generated content is still used in current gaming but not to the extent of uses in previous game consoles. The first-person shooter games Borderlands 2 and Borderlands 3 both use a procedural generation system to generate different guns. It is said that Borderlands 3 has over 1 billion unique guns.

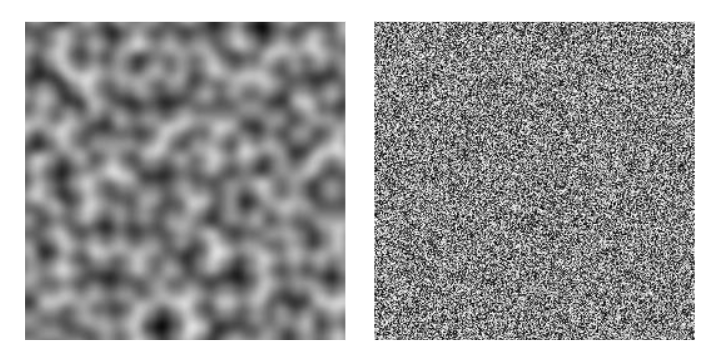
Chapter 3: Procedural Generation Algorithms

## **3.1 Introduction**

There are many different algorithms used for procedural generation. Specific areas of procedural generation have specific algorithms. Other areas will have numerous different algorithms that can be used , these their own strengths and weaknesses depending on the use of the algorithm or the desired result.

## **3.2 Noise Algorithms**

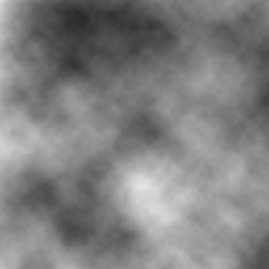
Noise algorithms are generally used for procedural terrain generation. Nosie algorithms are used to assign elevation values to each pixel in a heightmap programmatically. Co-ordinates of a each pixel in a heightmap are the input to a noise algorithm and the output is the elevation value of the pixel. The most basic method to produce these elevation values is to iterate through each pixel and assign a random value for the elevation but this can result in volatile and unrealistic terrain. This is known as “White Noise”. The solution to this is apply a specific method that gives each pixel awareness of nearby vertices. These solution result in “Coherent Noise”, as opposed to white noise (see Diag. 3.1). Coherent noise is characterized by its three constraints; Any input value will always result in the same output value, small changes to the input value results in small changes to the output value and large change to the input value will result in more random changes in the input value. (Vitacion and Liu, 2019)



Diag. 3.1: White Noise(left) vs. Coherent Noise(Right)

## **3.2.1 Value Noise**

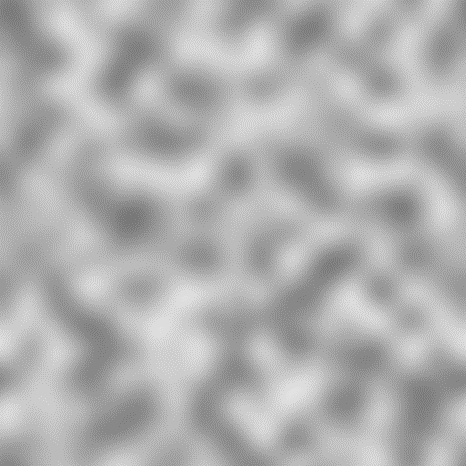
The value noise assigns each point of the heightmap pseudorandom values. For each point, the final elevation value is acquired by interpolating its value with those of the surrounding points. Doing so provides awareness of each point of the elevation values of its neighbours. This is characteristic of Consistent noise. You can use different variants of value noise to attain this result by using varying interpolation functions. In this case , the basic variant of value noise tested uses Quintic interpolation in order to alleviate variations at each point in the heightmap.(Vitacion and Liu, 2019)



Diag. 3.2: Value Noise

## **3.2.2 Cubic Noise**

The cubic noise algorithm generates a map of white pseudorandom noise greater than the desired heightmap, then this map is scaled down to the desired resolution using cubic scaling. This process takes each pseudorandom value of the original white noise and combines it with the nearby noise values with the overall effect, through cubic interpolation, of smoothing out the variations. Through this mechanism, each position in the final heightmap is given awareness of the value of nearby positions, creating coherent noise.



Diag. 3.3: Cubic Noise

## **3.2.3 Perlin Noise**

One of the oldest coherent noise algorithms, Perlin Noise can be used to generate heightmaps. Conceived by Ken Perlin in 1983, Perlin noise works by first generating a unit length grid. At each intersection, gradient vectors originate that point in pseudorandom directions. The position of each noise value being calculated is placed within each cell such that each coordinate is surrounded by four grid points, each with their own gradient vector. Next, four vector distances are calculated representing the distance from the noise value coordinates and the corners of the cell surrounding it are calculated. Lastly, the dot products between each vector of the gradient and each corresponding distance vector is calculated and interpolated to produce the final noise value. Because the computation of each noise value is dependent on the distance vectors, each point is given awareness of the value of its neighbours and coherent noise is achieved(Vitacion and Liu, 2019)



Diag. 3.4: Perlin Noise

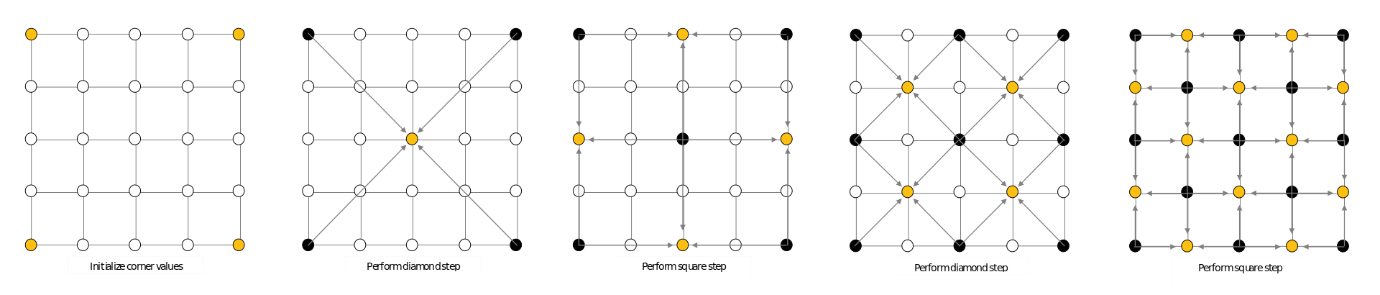
## **3.2.4 Simplex Noise**

Another coherent noise algorithm is Simplex Noise. Designed in 2001 by Ken Perlin and serves as an optimization of his earlier algorithm on Perlin Noise. Simplex noise simplifies Perlin Noise by the use of a simplex grid in which instead of a square, each cell is a triangle. This leads to less distance vectors needed for each height value to be calculated since each cell has just three corners. In addition, Simplex Noise instead of interpolation, it uses a summation process to calculate the final noise values. Simplex noise is computationally less complex and scalability is better than Perlin noise. Directional artifacts, by using Simplex grids, are minimized as well. (Vitacion and Liu, 2019)

## **3.2.5 Diamond Square**

Another consistent noise algorithm is the Diamond Square Algorithm. The Diamond Square Algorithm is used to generate heightmaps. The algorithm makes use of the midpoint displacement principle for the purpose of assigning elevation values for individual coordinates. The algorithm begins with the four corner coordinates of a square coordinate grid initialized to pseudorandom values. It then measures the elevation average of these four points plus or minus an offset-pseudorandom. After that, this value is assigned to the elevation of the point in the grid's direct centre. A grid of four smaller squares is then subdivided into four and the process is repeated for each of them. This is done until each value in the elevation value has been assigned to the grid.

At each level of subdivision, the magnitude of the pseudorandom offset decreases to ensure sharp changes do not occur as small changes in close coordination between coordinates in close proximity are calculated by the smaller subdivision steps. The Diamond Square Algorithm, via this mechanism, produces coherent noise. It is essential to note that the Diamond Square Algorithm is distinctive from the other algorithms because is inherently fractal. Consequently, it is unnecessary to use a technique like Fractional Brownian. Nonetheless, there are several limitations to the algorithm. For the original grid, In order to subdivide fully, the original grid must have 2n+1 dimensions. In addition, the algorithm can only produce square heightmaps due to its design.(Vitacion and Liu, 2019)



Diag. 3.5: Diamond Square Algorithm steps

## **3.3 Procedural Level Generation Pipeline**

The procedural level generation pipeline is used for the efficient creation of complex indoor levels, minimizing limitations on level and designers’ creativity. This pipeline was created by Bartosz von Rymon Lipinski and Simon Seibt of the Nuremberg Institute of Technology, and Johannes Roth and Dominik Abé of Mimimi Games GmbH. Mimimi Games are best known for developing the 2016 real-time tactics video game, Shadow Tactics: Blades of the Shogun.

### 3.3.1 Base Graph

### 3.3.2 Minimum Spanning Tree

### 3.3.3 Level Graph and Merging

### 3.3.4 Geometric Representation

## 3.4 Voronoi Diagram

## 3.5 Fractals