



Newcastle University

Investigating if Perlin Noise Heightmaps can be used to create accurate approximations of Manhattan

Matt Alton

160201930

BSc Computer Science (Games Engineering)

May 2021

Supervisor: Dr Gary Ushaw

14540 words

Abstract

This dissertation investigates the use of Perlin noise heightmaps to establish if it is possible to create a program that can create accurate approximations of Manhattan with Perlin noise.

A summary of the background research that was undertaken is presented, as well as including the fully documented development of a program which utilises Perlin noise heightmaps to set the heights of buildings within a procedurally generated city. The program is tested with various noise levels to determine if Perlin noise can produce accurate approximations of a Manhattan city structure. An evaluation of these results is presented, along with a discussion of how future work can further the ideas presented in this dissertation.

Declaration

"I declare that this dissertation represents my own work, except where otherwise stated."

Acknowledgements

Firstly, I would like to thank my dissertation supervisor Dr Gary Ushaw, for directing me to pursue research into a field I have acquired a newfound interest in. I would also like to thank my friends and family for helping motivate me and always being there for me.

Table of Contents

Abstract.....	2
Declaration	3
Acknowledgements.....	4
Table of Contents.....	5
Table of Figures	7
Table of Code.....	8
Table of Tables.....	8
1: Introduction.....	10
1:1 Introduction	10
1:2 Subject Area	10
1:3 Aim and Objectives	11
1:4 Dissertation Structure	11
2: Background Research	14
2:1 Introduction	14
2:2 Procedural Generation.....	14
2:2:1 Rogue-like Video Games	14
2:2:2 Procedurally Generated Landscapes	15
2:3 Noise Functions.....	17
2:3:1 Lattice Gradient and Perlin noise.....	17
2:4 Procedural City Generation.....	19
2:4:1 Citygen and Road Networks.....	19
2:4:2 L-Systems for Modelling Cities.....	19
2:4:3 Texturing	21
2:4:4 Unity3D Prefabs	21
2:5 Manhattan City Grid	22
2:5:1 Street Plan.....	22
2:5:2 Blocks	23
2:5:3 Criteria	25
2:6 Summary	26
3: Design and Implementation.....	28
3:1 Introduction	28
3:2 Planning	28

3:3 Tools and Technologies.....	28
3:3:1 Unity3D	28
3:3:2 C#	29
3:3:3 Visual Studio	29
3:4 Program Specification	30
3:5 Development.....	30
3:5:1 Building Generator.....	30
3:5:2 Block Generator	32
3:5:3 Grid Generator.....	34
3:5:4 Perlin Noise Generator	36
3:5:5 Switch Camera	40
3:6 Summary	41
4: Results and Evaluation	43
4:1 Introduction	43
4:2 Testing and Evaluation Technique	43
4:3 Finding a Building Height Range	44
4:4 Noise Levels	47
4:5 Evaluation	48
4:7 Screenshots of Procedural Cities	51
4:8 Summary	55
5: Conclusion	57
5:1 Satisfaction of the Aim and Objectives.....	57
5:2 Reflection	58
5:2:1 Project Conclusion	58
5:2:2 What has been learnt?.....	58
5:2:3 What could have been done differently?	58
5:2:4 What remains to be done?	59
References.....	61
Appendix A – Screenshots of Procedural Buildings	65
Appendix B – Source Code.....	72
Appendix C – Test Results.....	73

Table of Figures

Introduction:

Figure 1 Marvel's Spider-Man procedurally generated city [1]	10
--	----

Background Research:

Figure 2 Spelunky, Derek Yu [5]	14
Figure 3 No Man's Sky, Hello Games [9]	15
Figure 4 Stack of grids for the landscape, Fischer et al. [11]	16
Figure 5 Noise texture.....	17
Figure 6 Random vs Perlin Values	18
Figure 7 Buildings modelled with L-systems, Parish & Müller [17]	20
Figure 8 Building Textures, Greuter et al. [20].....	21
Figure 9 Lots in a city block, The HouseShop [26]	23
Figure 10 Block containing 25 x 100 feet lots	24
Figure 11 Block containing 50 x 100 feet lots	24

Design and Implementation:

Figure 12 Top-down view of blocks produced in the program.....	25
Figure 13 First example of building generation	31
Figure 14 Positioning the buildings into blocks	31
Figure 15 Building generated by ProceduralBuildings	32
Figure 16 Block generated with ProceduralBlocks (Top-down).....	33
Figure 17 Block generated with ProceduralBlocks (Side view).....	33
Figure 18 City generated with ProceduralGrid (Top-down)	35
Figure 19 City generated with ProceduralGrid (Side view).....	35
Figure 20 City generated with low Perlin noise level.....	39
Figure 21 City generated with medium Perlin noise level	39
Figure 22 City generated with high Perlin noise level.....	39
Figure 23 Alternate angle of city generation	40

Results and Evaluation:

Figure 24 Frequency of Manhattan building heights.....	44
Figure 25 Frequency of building heights between 3 and 53 meters	45
Figure 26 Frequency of building heights between 3 and 35 meters	46
Figure 27 Comparison of the Mean, Median and Variance of Noise levels between 3 and 53 meters	48
Figure 28 Comparison of the Mean, Median and Variance of Noise levels between 3 and 35 meters	48
Figure 29 Comparison of the frequency of building heights of Noise levels between 3 and 53 meters	49
Figure 30 Comparison of the frequency of building heights of Noise levels between 3 and 35 meters	50
Figure 31 Random buildings between 3 and 35 meters	54

Table of Code

Code 1 Pseudocode to create, size and position building	31
Code 2 Code to create a block	33
Code 3 Code to create the city grid	34
Code 4 Pseudocode to apply Perlin noise to texture.....	36
Code 5 Pseudocode to obtain Perlin float value.....	36
Code 6 Pseudocode to create building with Perlin noise	37
Code 7 Code to create a 2D texture of Perlin noise.....	37
Code 8 Code to return the Perlin noise value.....	37
Code 9 Code to generate a building using Perlin noise	38

Table of Tables

Table 1 Mean, Median and Variance of Manhattan building heights	44
Table 2 Values for city generation between 3 and 53 meters.....	45
Table 3 Mean, Median and Variance of building heights between 3 and 53 meters.....	45
Table 4 Values for city generation between 3 and 35 meters.....	46
Table 5 Mean, Median and Variance of building heights between 3 and 35 meters.....	46
Table 6 Perlin noise levels for testing	47
Table 7 Screenshots of Perlin Noise Cities	52
Table 8 Images of Manhattan	54

1: Introduction

1:1 Introduction

This section introduces the dissertation to the reader by elaborating on the author's motivation and rationale behind their choice to research into this area. The aim and objectives of the dissertation are stated as well as describing the overall structure of the dissertation.

1:2 Subject Area

Manhattan is one of the five boroughs of New York City and is perhaps the most widely recognised cityscape in the world. Hosting iconic landmarks such as the Empire State Building and the Chrysler Building, sightseeing in Manhattan is truly something to behold. With advancements in technology, it is now possible to create such structures virtually through means of computer algorithms. This process is known as *Procedural Generation* and is demonstrated in Marvel's *Spider-Man* [1] video game. This game boasts an entire virtual New York City which is procedurally generated for the player to explore.



Figure 1 Marvel's Spider-Man procedurally generated city [1]

Another interesting application of procedural generation is the use of *heightmaps*. Heightmaps are raster images that can be used to create randomised landscapes on surfaces. One of the more notable uses of this technique is in Mojang Studio's *Minecraft* [2], whose terrain is set using 2D heightmaps that are affected by Perlin noise. By using Perlin noise heightmaps, each world that is generated is entirely randomised and unique. However, this randomness is not entirely random, rather it is *controlled* randomness. A controlled form of randomness in this instance means that every randomly generated point has a coordinate which is in relation to the coordinate of a point next to it. This is because of Perlin noise and it is this effect that this dissertation shall be focusing on.

Past research has shown effective methods of implementing Perlin noise heightmaps to set the structure of cityscapes. Mirrorfishmedia created a system entitled *ImaginaryCities* [3], which utilises a Perlin noise heightmap to set the heights of buildings in a city grid. Whilst this is an effective tool to generate a wide variety of city structures under Pelin noise, there has been little research to detect if the Perlin noise algorithm can be used to create accurate approximations of Manhattan.

This dissertation focuses on implementing this technique to shape procedurally generated cities specifically to Manhattan using a researched criteria the program must follow.

The results of this dissertation establish comparisons between buildings distributed in Manhattan and buildings distributed with Perlin noise. Despite these results not showing conclusive evidence Perlin noise can be used to accurately create Manhattan, additional work is discussed into what could be done to develop an altered Perlin noise algorithm that would be an effective tool to accomplish this.

1:3 Aim and Objectives

The aim of the project is to:

“Establish if the Perlin noise algorithm can be used to procedurally generate an accurate approximation of Manhattan”

This aim shall be met with the following objectives:

- To investigate the Perlin noise algorithm and how it can be used in procedural city generation
- To investigate the Manhattan city structure to define a set of criteria a building generation algorithm must follow
- To develop a program that procedurally generates a city structure with Perlin noise
- To test at least 5 levels of Perlin noise to compare similarities between Perlin noise building distribution and Manhattan building distribution

1:4 Dissertation Structure

This dissertation is divided into the following sections:

Introduction

An introduction into the subject area this dissertation shall be focusing on as well as defining the aim and objectives the project hopes to fulfil

- Subject Area
- Aim and Objectives

Background Research

A discussion of the research that was undertaken into the subject area, including a review of existing academic works and any algorithms to be implemented

- Procedural Generation
- Noise Functions
- Procedural City Generation
- Manhattan City Grid

Design and Implementation

A detailed discussion on what work was completed and why

- Planning

- Tools and Technologies
- Program Specification
- Development

Results and Evaluation

Summarising the results from testing the developed program as well as detailing an understood evaluation

- Testing and Evaluation Technique
- Finding a Building Height Range
- Testing Noise Levels
- Evaluation
- Screenshots of Procedural Cities

Conclusion

A finalised conclusion on the results of the work that was carried out whilst laying a foundation for future work to build from

- Satisfaction of the Aim and Objectives
- Reflection

2: Background Research

2:1 Introduction

This section alludes to the background research that was undertaken for this dissertation.

2:2 Procedural Generation

This sub-section looks at the techniques used for procedural generation and how they can be used in my dissertation.

Smith describes Procedural Content Generation as the *“use of a formal algorithm to generate game content that would typically be produced by a human”* [4]. This content can include 3D objects which is what I shall be using to generate the buildings in my program. However, there are plenty of other types of content that could be created such as entire landscapes, dungeon levels, in-game items, vegetation and much more.

2:2:1 Rogue-like Video Games

Rogue-like games are a sub-genre of the RPG genre that involve the creation of randomly generated levels through procedural generation such as *“Spelunky”* [5] and *“Dead Cells”* [6]. Often set in dungeons, typically the player is tasked with traversing the generated dungeons with the goal of finding items, killing enemies, and finding the exit.



Figure 2 Spelunky, Derek Yu [5]

Smith and Bryson observe that Roguelike games typically comprise of *“a 2D grid of navigable tiles, such that the environment is discrete rather than continuous”* [7], and that *“each level will comprise of a number of rooms, of varying size and shape, connected by corridors that will be positioned such that neither the rooms or the corridors overlap or crossover”* [7].

İzgi outlines a framework [8] for roguelike video games development with a set of common features of a roguelike game:

- *Procedural content creation*
- *Permanent death*
- *Turn-based game in the grid-based world*
- *Single Character with an Inventory*
- *Discoverability*

It is important that when the levels are generated that precautions have been set in the code that allow for the player to be able to complete the level. An example of this could be ensuring the player can jump high enough to access a room with an important item such as a key to complete the level.

Although this technique of procedural content generation is interesting and worth discussing, it is not entirely relevant to this dissertation, so I shall avoid going into depth regarding the algorithms to generate levels. However, it is worth pointing out the idea that content is generated by a pre-defined set of rules. This is how my program will generate the grid of the city, with blocks including buildings of set sizes.

2:2:2 Procedurally Generated Landscapes

Procedural generation can also be applied to create entirely randomised and unique surfaces for developers to take advantage of when creating a video game. One such example of this is Hello Games' *No Man's Sky* [9], a survival game involving expansive exploration of over 18 quintillion planets [10].



Figure 3 *No Man's Sky*, Hello Games [9]

Whilst there are several aspects of the game that are procedurally generated such as all the galaxies, the names of planets and even the structures on planets; this section focuses on the procedural generation occurring to form the landscape of the planets.

The planets created when playing the game are produced by algorithms that shape the surface of each planet to form different biomes such as mountains, rainforests, and oceans.

Fischer *et al.* crafted a pipeline approach for a system that allows the creation of realistic terrains [11]. This approach involves using a stack of grids affected by noise functions, which I shall go into in more detail later. Each grid represents different aspects of the environment, which are the terrain, temperature, wind, moisture, precipitation, and biomes.

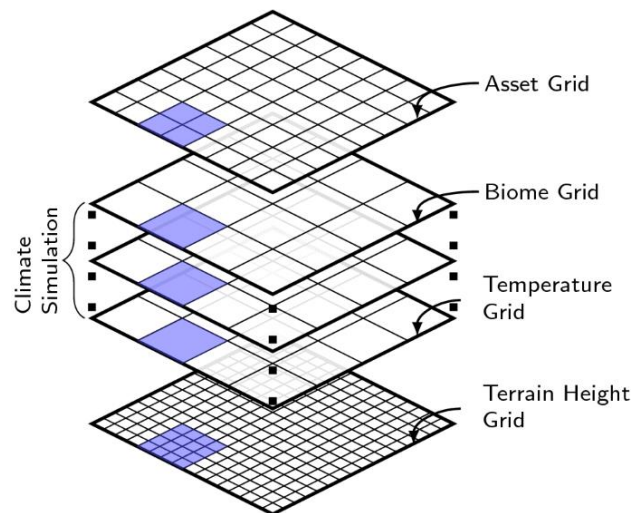


Figure 4 Stack of grids for the landscape, Fischer et al. [11]

By combining these grids together, a final terrain is formed. Due to all the possibilities each grid layer can be, this leads to infinite terrains that can be generated when each grid is combined. Whilst the procedural generation technique utilised in *No Man's Sky* is largely kept secret, it is safe to say that the system shares many similarities.

The technique used here is very effective and one I wish to use in my program. By using a grid layer, I can apply the noise functions onto it to set the heights of the buildings. A similar technique is demonstrated by mirrorfishmedia, who uses a Perlin noise heightmap to set the heights of buildings in a procedural generator entitled *ImaginaryCities* [3].

Here an instance of Perlin noise is applied to an empty 2D texture. For each building to be generated, a position is selected and a value is sampled from the 2D texture from the coordinates in that position. This value is then used to set the height of the desired building, which is repeated for every building within every block inside the grid. There are variables that can be altered to edit the size and length of blocks, the gaps between them, how many blocks there are and much more. However, the interesting factor about this technique is that the scale of the noise can be altered in real time to generate vastly different results.

ImaginaryCities utilises a slider to alter the scale of Perlin noise in the program. When the slider is moved the heights of the buildings are updated after a key press. A higher noise value makes the building heights appear more spread out while a lower noise value makes the building heights much flatter and closer together.

This technique of applying Perlin noise to a texture map to set the heights of the buildings would work perfectly in this dissertation. I shall aim for my program to utilise a similar method but focus more on shaping the city to appear closer to a Manhattan grid layout. I shall do this by hard coding values that the user is unable to alter. This way it will be ensured that the city being generated is always shaped like Manhattan.

Before implementing this technique, it is important to research into not only existing procedural city generation techniques, but also into what noise functions are, how any relevant algorithms work, and how I can ensure the cities being produced share a similar grid to that of Manhattan.

2:3 Noise Functions

This sub-section looks at noise functions, giving detailed explanations of any algorithms and how they can be utilised in this dissertation.

Defined by Lagae *et al.*, noise is “*the random number generator of computer graphics*” [12]. The effect of noise produces results that are random and unstructured. This is useful when extensive detail is needed that is lacking in structure.



Figure 5 Noise texture

Noise functions are a fast approach to produce detailed and random results, making them ideal for computer applications.

Ebert blueprints a set of properties a typical noise function must have [12]. Noise must:

- *Have a range* – Usually between -1 and 1
- *Not show obvious patterns*
- *Be repeatable with use of inputs*
- *Be band-limited* – It is smooth
- *Be stationary* – Results are invariant to translation
- *Be isotropic* – Results are invariant to rotation

In procedural generation, noise can have several uses for many graphical effects such as clouds, waves, and particle effects such as fire or rocket trails. Noise functions can be classified into three categories: lattice gradient noises, explicit noises, and sparse convolution noises. For this dissertation I shall be using Perlin noise, which is an example of lattice gradient noise. So, I shall explain this type of noise in detail.

2:3:1 Lattice Gradient and Perlin noise

Lagae *et al.* define lattice gradient noise as a function that “*generates noise by interpolating or convolving random values and/or gradients defined at the points of the integer lattice*” [12]. An example of this type of noise is Perlin noise, which is what I shall be using for this dissertation. This algorithm was developed in 1985 by Ken Perlin, who created an image synthesiser implementing the algorithm [13].

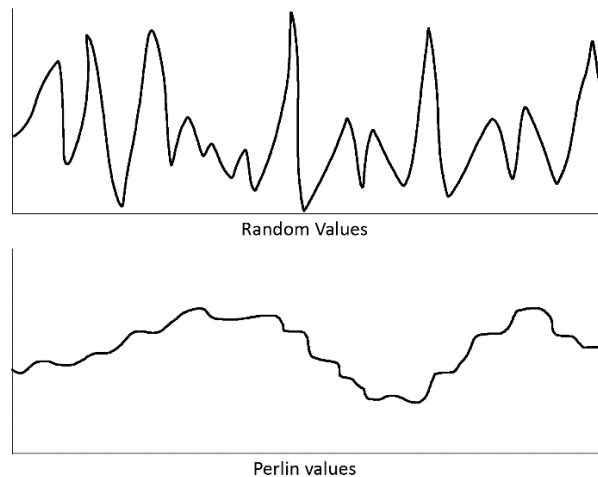


Figure 6 Random vs Perlin Values

Figure 6 roughly shows the difference between values determined randomly and values determined by Perlin noise. The effects of Perlin noise produce results that look more controlled and organic, as opposed to the random values which are much more varied and jagged. This is due to the fact that for each point on the graph for Perlin noise, the next point is interpolated between a point that is within one of the nearest vertices. Simply put, this means that each point along the graph is related to the point next to it by being within a certain range of it. As points along the random graph have no relation to one another, this leads to various points with massive differences between them, looking very unnatural. Values interpolated with Perlin noise are said to be under normal distribution.

This effect leads to natural looking curves, which make them ideal for landscapes in video games. One of the more notable examples of Perlin noise in a video game is *Minecraft* [2], which utilises Perlin noise for its landscapes to form the biomes with hills, rivers, and oceans.

The creator of *Minecraft*, Notch, details the earliest version of *Minecraft* terrain generation as utilising a “2D Perlin noise heightmap to set the shape of the world” [14]. The algorithm for the latest version is complex and is kept a secret but is said to still use “2D elevation and noisyness maps” [14].

This is a similar technique to what was mentioned in the previous section in mirrorfishmedia’s *ImaginaryCities* system [3], which involves the use of a Perlin noise heightmap. In the previous section I explained how the program operates to set the heights of buildings being generated, but here I shall give more of an explanation into how the Perlin noise algorithm is used and how I can adapt it for use in this dissertation.

As stated before, in *ImaginaryCities* an empty 2D texture is created. For every coordinate inside of the grid, a sampled value is calculated and used to set a pixel in the 2D texture. The value is calculated using a function called *SampleNoise*. The function takes the coordinates which are then recalculated using the inputted *perlinOffset*, which ensure the buildings generated are in line with the city grid, and a *noiseScale* which multiplies the effect of Perlin noise on each coordinate. A sample value is then calculated using Unity3D’s *Mathf.PerlinNoise* function. Unity’s documentation states that this function takes two parameters, an x and y coordinate, and returns a float value between 0.0 and 1.0 [15]. This sample value is stored in order to set a new *Color* value, which is then

returned and applied to the 2D texture. After all the coordinates have been assigned a value to the texture, the texture can be accessed to set the heights of all the buildings.

This technique proves to be very effective in mirrorfishmedia's *ImaginaryCities* [3]. I shall utilise the *Mathf.PerlinNoise* and *Color* values in my program to create a 2D texture to also set the heights of all the buildings. However, there are a vast number of features in the program that are not necessary, so I shall only be using the math functions provided here in my program.

Now that I have provided an insight into what Perlin noise is and how it can be used to affect an empty 2D texture, I need to research how the program can generate each of the buildings, blocks, and roads, and also define a criteria they must meet so I can shape them to approximate a Manhattan grid layout.

2:4 Procedural City Generation

This sub-section looks at the techniques used for procedural generation of virtual cities and how they can be used in my dissertation.

Procedural virtual cities are one of the most explored avenues of procedural generation. Whether it's implementing dynamic road networks, detailed buildings and paths or even exploring an infinitely expanding city, there are various aspects of procedural city generation that can be explored and adapted.

Here I shall be looking into existing city generators to observe if there are any techniques I can use for road and building generation.

2:4:1 Citygen and Road Networks

Citygen is a fully interactive procedural city generator which automates terrain, road and building generation using a simple, intuitive interface developed by Kelly and McCabe [16]. This interface allows the user to manipulate various aspects of the city, for instance where roads intersect, which is then updated in real time.

Whilst *Citygen* does build its cityscapes on top of elevating terrain, it also introduces a technique for the generation of "*primary and secondary roads*". Road networks consist of "*nodes*" which are stored in "*adjacency lists*" to keep track of which nodes are directly connected to each other. It is because of these nodes that the user is able to alter various aspects of the city by changing the properties of the nodes such as their position and what other nodes they connect to.

Despite *Citygen* being able to yield flexible and eye-catching road networks through the use of nodes, they have no practical use in this dissertation. As I will be generating a Manhattan city structure, road networks are usually kept in a grid-like fashion in straight lines at 90 degrees to each other. If this dissertation were to generate city structures with various road directions, then using nodes for my road networks would allow me to shape the city to what is needed. However, *Citygen* does introduce the idea of generating buildings inside of areas formed by roads, which is what I shall be needing for my program.

2:4:2 L-Systems for Modelling Cities

In 2001, Parish and Müller proposed a procedural generation system entitled *CityEngine*, which utilises "*L-Systems*" to model cities [17]. The name L-systems originates from the name *Lindenmayer systems* [18], who originated the theory in relation to plant branches. After years of study, L-systems

eventually became an effective tool for modelling in computer programming developed by Prusinkiewicz and Lindenmayer [19].

These modelling tools are handy for creating plant models, as L-systems produce an effective branching system. When used for the roads in *CityEngine* [17], roads are able to branch outwards and connect with each other, a technique also implemented into *Citygen* [16]. With these roads in place, the sections between them are divided into “lots”. It is within these lots where the buildings are then generated. For the modelling of the buildings, L-systems are used again in *CityEngine* to shape the geometry of the buildings using a “*parametric, stochastic L-system*”. There are three types of buildings that are generated: skyscrapers, commercial buildings, and residential houses, each determined by what shape the base of the building starts out as.



Figure 7 Buildings modelled with L-systems, Parish & Müller [17]

Figure 7 shows the effect of these L-systems on a starting cuboid building. The L-systems consist of “*transformation modules (scale and move), an extrusion module, branching and termination modules, and geometric templates for roofs, antennae*”. The effect of these modules effectively trims and shapes the starting cuboid to produce a high level of visual complexity amongst all the buildings.

For the purpose of this dissertation, I am going to avoid using L-systems in my program as I am only investigating into the Perlin noise value that gives the closest approximation to Manhattan. However, if I were to develop my program further it would be a good idea to incorporate L-systems into it, as then my program would be able to produce the same results but with an added level of detail. It is worth looking at L-systems in my research for procedural city generation as many studies implement them as they produce fantastic results, so if any future work were to occur to make my program better it would be to include more varied and detailed city generations.

2:4:3 Texturing

Procedurally texturing the buildings in my program would be another great way to efficiently provide an extra level of detail. One example of this is a study by Greuter *et al.* [20] in which they explore procedural texturing during real time procedural generation.



Figure 8 Building Textures, Greuter *et al.* [20]

In this system, there are multiple single window textures, all of which have different appearances as shown in *Figure 8*. The length of each side of every building has to be divisible by the length of a single window texture. So, for each building being generated, the total length is divided by the size of the window texture. A random texture is chosen from the selection and is applied to the side of the building that many times. When this is completed for each side, the entire building is textured.

Like L-systems, it is not a necessity to add texturing into my program but it would not be a particularly difficult task to achieve. The technique described above is relatively simple and wouldn't be too much work to add into my program. However, since I am using Unity3D for this project, there exists a much simpler way of generating buildings and applying textures to them. This is possible through the use of Unity3D Prefabs.

2:4:4 Unity3D Prefabs

The Unity documentation defines Prefabs as a system that “allows you to create, configure, and store a *GameObject* complete with all its components, property values, and child *GameObjects* as a reusable Asset” [21]. The documentation also states, “you can nest Prefabs inside other Prefabs to create complex hierarchies of objects that are easy to edit at multiple levels”, and it is with this nesting system that I aim to generate the buildings, blocks, and entire city grid.

In my program, I can create a grid generator that generates the entire area of the city, inside of which I can have a block generator that generates a single block in the city. Then inside of that I can have a building generator that generates a single building in a block. This is all possible through the nested Prefab system. All that would be required of the grid generator is to generate a block at regular intervals until the grid is full of blocks. Similarly, all that would be required of the block generator is to generate a building until the block is full.

As for the building generator, I aim to use a similar nesting Prefab system that incorporates the Perlin noise algorithm into it. As the Perlin noise algorithm outlined earlier returns a float value between 0.0 and 1.0, it will be possible to set a maximum height for any building which can then be multiplied using that float value to set the Perlin noise height of the building. With the height set the program can create a building, put it into position and set the appropriate dimensions.

In mirrorfishmedia's *ImaginaryCities* [3], a different technique is used to set the height of a building, using an actual model of a building for the Prefab. Instead of stretching the prefab to the height of

the building, the building is built up in layers of prefabs. There are three types of layers: a base, middle, and top. Each type has an array of different models, so when the Prefab is created, a random piece is selected each time. By doing this, the buildings are built up fully decorated and with even more randomness in their appearance. However, I will not be using this technique because decoration is not the focus of this dissertation. Whilst it would be an added benefit that would be simple to implement, it is better in this instance to stretch a single Prefab as opposed to adding multiple layers, as adding multiple Prefabs for every building would be costly on performance. So, there is no need to implement texturing or models for the focus of this dissertation.

With the Perlin noise algorithm and the building, block, and grid generation algorithms explained, all that is left to research is how I will shape the city to appear like Manhattan.

2:5 Manhattan City Grid

This sub-section looks at the properties of a Manhattan grid layout, in order to define a set of criteria my program must fulfil when procedurally generating a city.

The reasoning behind my decision to choose a Manhattan grid is due to its simplicity as well as being easily recognisable. Creative Commons Attribution-Share state that Manhattan adopts a uniform grid plan consisting of streets and avenues running at right angles to each other [22]. Walks of New York suggest that the city also becomes “*easy to navigate*” using the block and avenue number system [23].

2:5:1 Street Plan

Walks of New York defines a set of rules that makes up New York’s street plan [23]. The key rules are:

- *Streets run east-west*
- *Street numbers ascend as they move northward*
- *Avenues run south-north, with numbers beginning on the east side of the island and ascending to the west*
- *Fifth Avenue is Manhattan’s central dividing line*
- *Street address numbers begin at Fifth Avenue, and increase as they move outward*

As the procedural generator will not concern street numbers, the two rules I will use from this street plan are that streets run east-west and avenues run south-north. The system will also not need a central dividing line, as it is only meant to produce a small area of the city. As I will be using Unity3D, instead of east-west and south-north I will use the x, y, and z axes.

In order to define a set of rules for my system to follow, I will need to conduct research into the proportions of a New York block.

2:5:2 Blocks

As New York uses a typical grid plan, blocks tend to be rectangular and encased within streets and avenues which are all the same width. Different cities that use a grid plan have different spacing between streets, buildings, and paths.

In Manhattan, the sizes of different blocks can vary. A post made by B.P. contains the widths and lengths of the blocks in Manhattan [24], while Solomon states that the average length of a north-south block runs “approximately 264 feet” and the lengths are “roughly 750 feet” [25].

From analysing the dimensions of the blocks and from data collected by B.P. and Solomon, I am going to set the length of each block to approximately four times the size of the width. All of the blocks created shall be the same size. So, upon runtime, a small area of the city shall be generated by creating a flat grid which contains multiple evenly spaced and sized blocks, which are then filled with buildings. In order to fill these blocks with buildings, they need to be divided into *lots*.

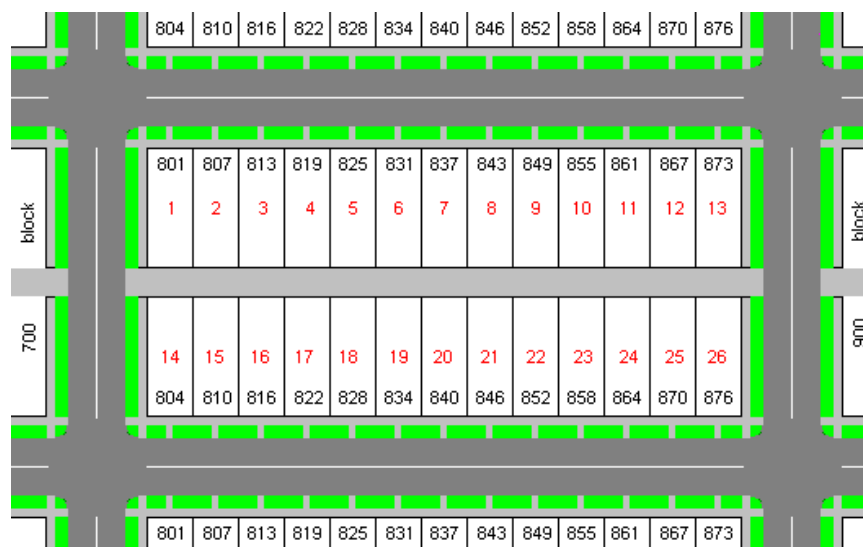


Figure 9 Lots in a city block, The HouseShop [26]

In a typical grid plan, blocks are divided into evenly sized lots, as shown in *Figure 9*. The block is split down the middle, so all lots are facing outward to the nearest street. However, in Manhattan some buildings can take up multiple lots, forming different building shapes that are more cubic or more rectangular. It is important that the finalised criteria contain a fixed size for a lot. So, I shall need to research into the dimensions of a single lot and how many lots there are inside a block.

“The greatest evil which ever befell New York City was the division of the blocks into lots of 25 x 100 feet.” - Ernest Flagg [27].

Whilst this evil is concerned with the problem of confined living space with minimal lighting, it does provide the dimensions of each lot within a block the program needs to produce. However, as previously mentioned, the sizes of blocks tend to vary slightly, so I will need to decide how many lots will be put inside of a block.

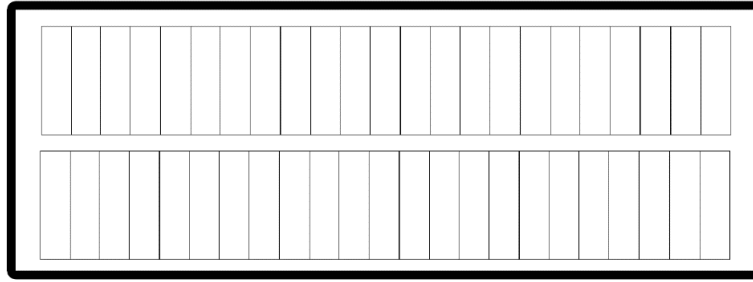


Figure 10 Block containing 25 x 100 feet lots

Figure 10 shows an example of a block containing lots with dimensions of 25 x 100 feet. Looking at this block, I believe there are too many lots inside and that they appear too thin. Whilst the length of the block could be reduced to include fewer lots, it would take away from the distinctive shape of the New York Block. So, I feel that increasing the width of the lots to be an acceptable solution to fix this.

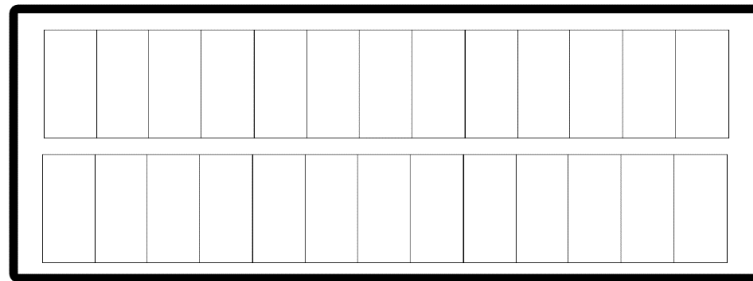


Figure 11 Block containing 50 x 100 feet lots

Figure 11 shows a block containing lots which are 50 x 100 feet, basically just double the width of the previous lots. I believe this representation of lots inside of blocks to appear more natural and fitting for a New York block. In this image there are 13 lots along the length of the block, making 26 lots in total. I believe this to be an acceptable amount. So, the criteria will include a rule that lots have dimensions of 50 x 100 feet.

As for the heights of the buildings, they will be random yet still affected by Perlin noise as described earlier. The minimum and maximum heights are difficult to say, as the heights change greatly in different areas of Manhattan. In Manhattan there are staggeringly tall buildings such as the One World Trade Center standing at 541m tall according to Emporis [28], which would massively skew the results of the program if the maximum height were simply set that high.

This is something that will be altered in testing to determine the best fitting minimum and maximum height for approximations for Manhattan. Along with testing for a suitable range, the testing will include using different values of Perlin noise. Testing different ranges and different Perlin noise values will help determine the best approximation for Manhattan using Perlin noise.

NYCOpenData is a site that provides free open data to any members of the public. One of the data sets they provide concerns information of over two thousand buildings in New York, which includes the heights of the buildings in stories [29]. In my testing for the final program, I will be able to use the data of Manhattan against the results from my program. This will also help quantitatively decide the best approximation for Manhattan using Perlin noise.

2:5:3 Criteria

After researching into the grid plan that Manhattan uses, I have accumulated the following criteria:

- Streets run east-west
- Avenues run south-north
- Blocks are evenly spaced
- Blocks are roughly 264 by 750 feet
- Blocks are divided into 50 by 100 feet lots

With these criteria I can create a mock-up image of what a generation from my program would look like.

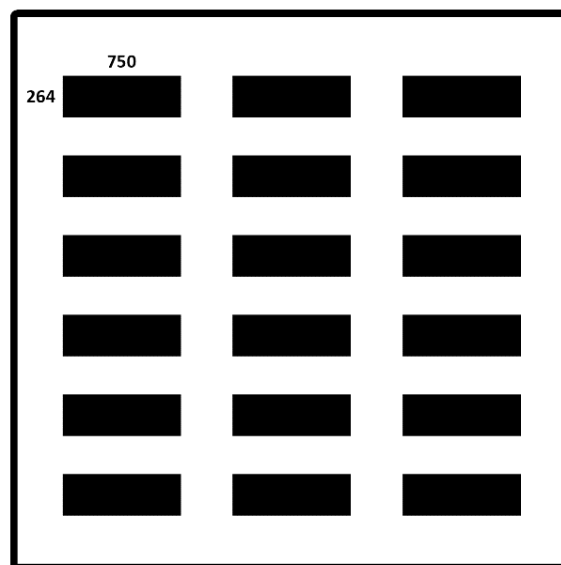


Figure 12 Top-down view of blocks produced in the program

Figure 12 shows what a generation of my program will look like from a top-down perspective. The blocks use the 264 by 750 feet dimensions I found from research. These blocks fill a plane which is generated on run time, with all the blocks being evenly spaced. Even though this is a rough estimate of how my program will look, I am happy with these results, so I shall use these dimensions for my program.

Now that I have a set of criteria, I need to adjust these criteria to work in Unity3D and C#, as I cannot work with feet as a measurement. I shall need to use the units in Unity3D, which are an arbitrary measurement. So, the revised criteria are:

- Streets run along the x axis
- Avenues run along the z axis
- Blocks are evenly spaced along the x and z axes
- Blocks are 13 lots wide and 2 lots deep
- A lot is 1 unit wide and 2 units deep

With this criteria in place, I have completed all the research needed to implement the appropriate algorithms to complete my objectives.

2:6 Summary

This section summarises the entirety of the background research that was undertaken, and what I will be implementing into my program to complete the objectives.

The basic aspects of procedural generation were covered, including the use of heightmaps to create randomly shaped surfaces to have the appearance of landscapes for planets. Mirrorfishmedia's *ImaginaryCities* [3] was introduced, which is a program that sets the heights of buildings using a Perlin noise heightmap.

A detailed breakdown of what Perlin noise is and how it is used to create more organic and controlled randomness appropriate for landscapes was discussed. Math calculations I can use in my program to successfully apply Perlin noise to a 2D texture in Unity3D was also covered.

Existing procedural city generation techniques available such as L-systems were mentioned. I also covered the use of Unity3D Prefabs [21], and how using a nested Prefab system I can create a program that can easily generate a city grid, which fills with blocks which in turn fills with buildings.

Finally, the criteria my program will need to use in order to appear close to an approximation of Manhattan was created:

- Streets run along the x axis
- Avenues run along the z axis
- Blocks are evenly spaced along the x and z axes
- Blocks are 13 lots wide and 2 lots deep
- A lot is 1 unit wide and 2 units deep

3: Design and Implementation

3:1 Introduction

This section discusses the approach for the planning of the entire dissertation, whilst elaborating on my decision for the tools I used in the program. An in-depth discussion on how the program was implemented in-line with a program specification is also given, as well as discussing the parts of the program I felt were successful and not successful.

3:2 Planning

At first a waterfall approach was taken with the project, as I felt it was easier for me to understand the task at hand if I completed each step one at a time. Originally, the project was focused around exploring the use of advanced performance techniques in real-time procedural generation. This idea was abandoned quite late into the project. I had completed the background research during March, but when it came to implementing the system with the performance techniques, the great difficulty of focusing on performance techniques was apparent. This meant I had to restart the project, instead focusing on a different area I had still performed research into; that being Perlin noise.

With the project coming to a halt, I chose to swap to an agile approach. This was because I had to go back and forth between the various areas of the dissertation. I already had a system in place to generate buildings in a city, so I decided to implement the Perlin noise algorithm into what I had. Previously building heights were entirely randomised, but the new system involved the Perlin noise heightmap setting the heights instead. After successfully doing so, I went back to focus the background research more on Perlin noise [2:3:1 Lattice Gradient and Perlin noise] and its application in mirrorfishmedia's *ImaginaryCities* [3].

Seen as though I was already shaping my previous city generation around a Manhattan city structure, it made sense for me to not make all the research on that pointless. So, I decided to focus the dissertation around Perlin noise and Manhattan, which led to me wanting to discover if Perlin noise can be used to create accurate approximations of Manhattan.

3:3 Tools and Technologies

This section looks at the various tools and programs I used for this project and what was learnt by doing so.

3:3:1 Unity3D

Unity3D is a popular game engine that allows the development of 2D and 3D games. The program comes packed with many features, which made it the essential choice for this project. The program along with its features are free, with many helpful tutorial series to get started in Unity3D, which made the program very accessible for me to get into.

It was decided from the start of the project that I was to use Unity3D. This was due to my previous experience with the program from other university modules, and its easy-to-use interface.

One of the features that ensured my decision to use Unity3D for this project was the use of Prefabs, more specifically nesting Prefabs [2:4:4 Unity3D Prefabs].

In a meeting with my supervisors, I was told about the *Instantiate* function that Unity3D also provides. This function allows game objects to be spawned in world space from code written in C#, which also factored in my decision to choose Unity3D, as this function works perfectly with the nesting Prefab system.

Another reason behind my decision was that Unity3D includes a C# scripting API and built-in Visual studio integration, making the process of setting up the procedural generation program simple.

3:3:2 C#

C# is an object-orientated and type-safe programming language. It is a language that branches from C and has many similarities to C, C++, Java, and JavaScript. I had not previously used C# before, but since I had experience in all of the similar programming languages, I was not against using it. While C++ is one of the most widely used programming languages for game development, and it is a language I have prior experience with, C# was my language of choice. This was due to the reason previously mentioned; it came packed in with Unity3D as a scripting API.

This meant there were several functions I could write in my scripts that worked perfectly for procedural generation.

Firstly, Unity3D Prefabs are very easily integrated into C# [2:4:4 Unity3D Prefabs]. *GameObject* variables can be created and the Prefab can be assigned to them. If these variables are made *public* then the Prefabs can be assigned from the Unity3D interface, which made initialising Prefabs into C# scripts very simple. Prefabs can also be transformed inside of the C# script, which was essential for shaping and positioning the buildings and blocks.

Secondly, C# can use various mathematical functions that are available alongside Unity3D. The *Mathf.PerlinNoise* function is vital for my program to operate [2:3:1 Lattice Gradient and Perlin noise], as well as Unity3D's *Instantiate* function previously mentioned. Two other functions that proved useful were the *GetPixel* and *SetPixel* functions to interact with a 2D texture. This meant the script could set Perlin noise onto the texture that was created to set the heights for the buildings.

3:3:3 Visual Studio

Visual Studio also comes integrated into Unity3D. This was useful for me as scripts that are written in Visual Studio are immediately updated and compiled in Unity3D when saved. This meant if there were any issues in my code Unity3D would detect them before I ran my program, which saved situations where the program could crash.

Visual Studio includes an easy to read syntax for C#, as well as GitHub integration, to allow me to push/pull my saved work onto an online repository. This meant any changes I made I could easily understand when looking back at my code, and also that there was a backup had I lost any of my work.

3:4 Program Specification

It was important that I understood what my program needed to be able to do prior to developing it. This section outlines the specification my program needed to fulfil as well as including any Unity3D specific features my program could achieve.

City Grid Generation

- On run time, the program must generate an area of a city grid
- The city grid must be filled with blocks
- The blocks must be filled with buildings
- The city must follow the researched criteria [2:5:3 Criteria]

City Alterations

- The program must allow the user to alter variables in scripts
- The variables must alter values related to the heights of the buildings
- The variables must not relate to values set to follow the researched criteria

Camera Switching

- The user must be able to control the camera perspective

3:5 Development

This section looks at the various stages of the project that were completed to achieve the final program. This is split into the different sections I completed in order of each other. This starts with the process I took to create the building generator, then the block and grid generator. Following this I implemented a Perlin noise generator which resulted in slight changes to the previous generators. Finally, I added a script to switch the camera perspective.

3:5:1 Building Generator

The first task I took to begin development was to generate a building. Initially, I thought I could only generate objects by creating the individual vertices and placing them in world space. It was only after speaking to my supervisors when I learnt about the *Instantiate* function that Unity3D provides.

I took to instantiating cube objects into world space. To give the appearance of buildings I scaled their heights randomly and moved their position in the y axis half the amount that was scaled in the y axis. If I had not positioned the buildings in the y axis, half of the building would be below the grid.

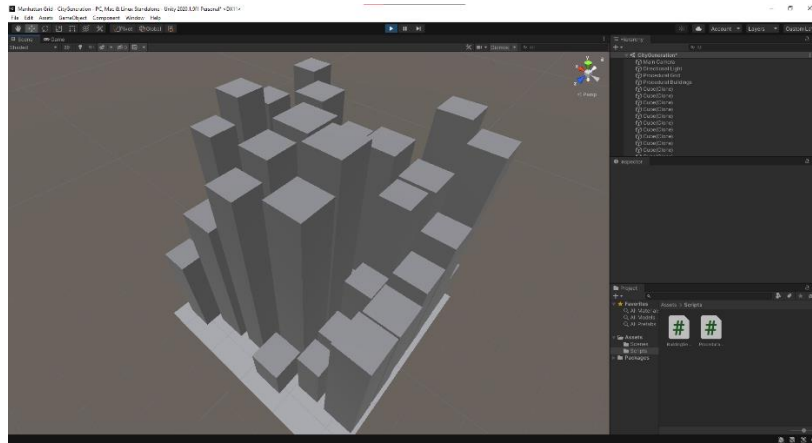


Figure 13 First example of building generation

The idea I had was to define a grid size and fill it with buildings with random heights. *Figure 13* demonstrates this idea.

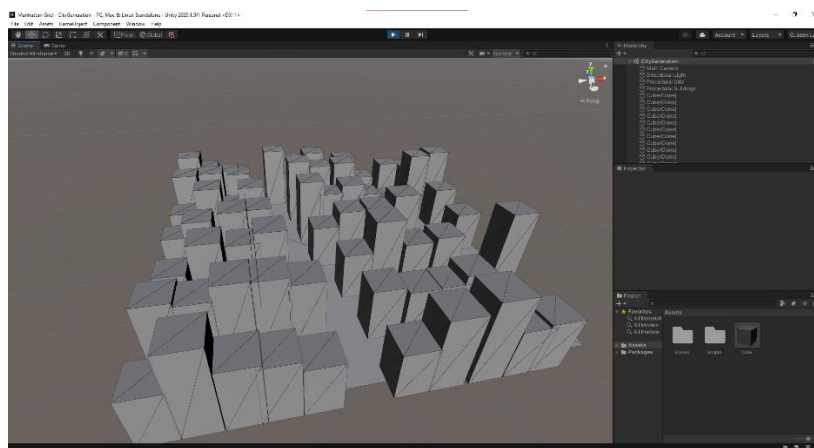


Figure 14 Positioning the buildings into blocks

I then decided to position the buildings into blocks to fill the grid. This involved defining a set size for a block which would be filled with buildings. Then a gap would occur and another block created. This process was repeated for the next row and so forth until the grid was full. Whilst this gave the appearance of Manhattan blocks, it was incredibly tedious to work with. It was at this point I researched further into Unity3D's Prefabs and discovered the nested Prefab system [2:4:4 Unity3D Prefabs].

Instead of trying to generate buildings in place to form blocks I created a block Prefab and a building Prefab. The building Prefab contained a *ProceduralBuildings* script, which when called on run time, would generate a single building of a random height.

```
BEGIN
CREATE (BUILDING)
SCALE_HEIGHT = MAX_HEIGHT * RANDOM_FLOAT
SCALE_BUILDING(1, SCALE_HEIGHT, 2)
POSITION_BUILDING(SCALE_HEIGHT / 2)
END
```

Code 1 Pseudocode to create, size and position building

The script accepts a *GameObject* Prefab which is just a cube object. The cube Prefab is *instantiated* into world space. A scale height is then determined by multiplying the maximum height a building can be by a random float between 0.0 and 1.0. The cube is then scaled in the y axis and translated in the y axis by half the amount so the base of the building will remain at the same height. The cube is also scaled by 2 in the z axis. This is because the cube needs to appear as a lot [2:5:2 Blocks and 2:5:3 Criteria], which are 1 unit wide and 2 units deep.

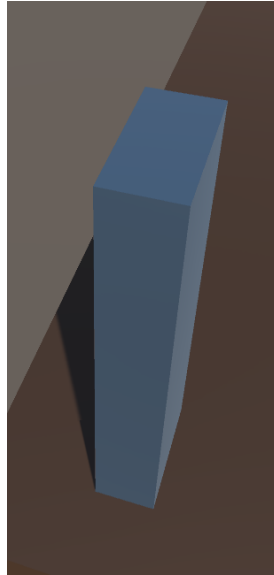


Figure 15 Building generated by *ProceduralBuildings*

This pseudocode effectively creates a single building to fit in a lot and is close to the final version of the program. I will explain the changes that were made to incorporate Perlin noise in the Perlin Noise Generator section [3:5:4 Perlin Noise Generator], but the very first implementation of the program produced a single building as shown in *Figure 15*. This shows a single cube that has been scaled in the y axis by a random number and scaled in the z axis by two. These transformations make the building fit the dimensions of a lot which fits the criteria [2:5:3 Criteria].

After completing this building generator, I moved onto creating the block generator.

3:5:2 Block Generator

This is the first instance of nested Prefabs. A block Prefab contains a *ProceduralBlocks* script which accepts a *GameObject*. The *GameObject* that is used is a building Prefab. So, whenever a building Prefab is instantiated in this generator, a single building is generated with a random height.

The *ProceduralBlocks* script contains two fixed variables for the dimensions of a block: *blockLength* and *blockDepth*. Blocks fit two sets of thirteen lots inside of them, so *blockLength* is set to thirteen and *blockDepth* is set to two. This script then instantiates a building Prefab for each lot inside a block.


```

int blockLength = 13;
int blockDepth = 2;
for (int x = 0; x < blockLength; x++)
{
    for (int z = 0; z < blockDepth; z++)
    {
        int buildingSpaceX = 1;
        int buildingSpaceZ = 2;
        Instantiate(building, transform.position + new Vector3(buildingSpaceX * x, 0, buildingSpaceZ * z), transform.rotation);
    }
}

```

Code 2 Code to create a block

In *ProceduralBlocks*, the nested for loops execute for the length of the block, then the depth of the block. Inside the nested for loops are two integers, *buildingSpaceX* and *buildingSpaceZ*. These integers are used to ensure that buildings are kept in the proper position, as otherwise all the buildings would overlap. These integers can be increased to increase the gap between buildings. However, this would not be appropriate for a Manhattan block according to the criteria [2:5:3 Criteria], so these values are kept fixed. For the case of the Manhattan block, *buildingSpaceZ* needs to be double *buildingSpaceX*. This is due to the fact that lots are 1 units wide but 2 units deep, so the space needs to be doubled in the z axis to accommodate for that.

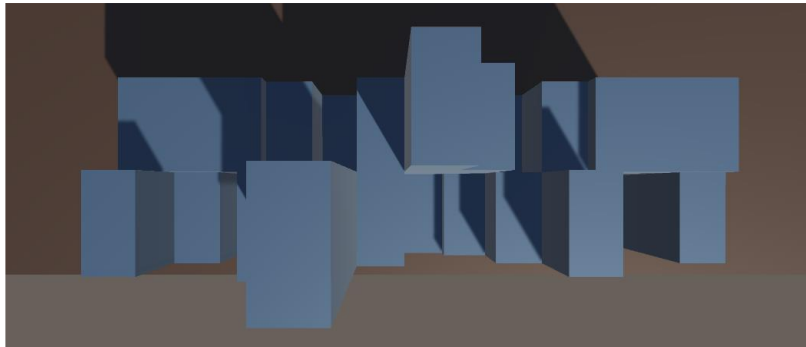


Figure 16 Block generated with *ProceduralBlocks* (Top-down)

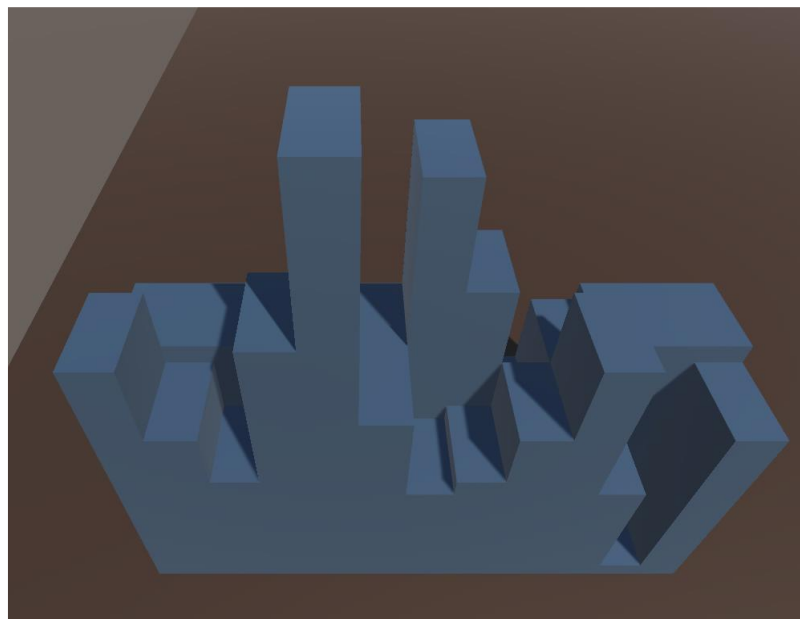


Figure 17 Block generated with *ProceduralBlocks* (Side view)

Figure 16 and 17 show an example of a block that can be generated using *ProceduralBlocks*. At this stage of development building heights were entirely random, but the shape of a Manhattan block is there. Each building takes up a 1x2 lot in the block, with thirteen buildings being generated twice.

For the implementation of Perlin noise, this script did not need changing, so the code in *Code 2* represents the final code of the block generator.

Following the implementation of the block generator, I needed to repeat the same process of filling a block with buildings, but this time filling the entire grid with blocks.

3:5:3 Grid Generator

The grid generator operates similarly to the block generator. The script *ProceduralGrid* accepts a *GameObject*, in this program the game object is a block Prefab. When a block Prefab is instantiated, the *ProceduralBlocks* script is called, which then instantiates a building Prefab. This is the nested Prefab system [2:4:4 Unity3D Prefabs].

```
int numBlocksX = 4;
int numBlocksZ = 7;
for (int x = 0; x < numBlocksX; x++)
{
    for (int z = 0; z < numBlocksZ; z++)
    {
        int blockSpacingX = 16;
        int blockSpacingZ = 7;
        Vector3 blockAlign = new Vector3(-30, 0, -22.5f);
        Instantiate(block, transform.position + blockAlign + new Vector3(blockSpacingX * x, 0.5f, blockSpacingZ * z), transform.rotation);
    }
}
```

Code 3 Code to create the city grid

In *ProceduralGrid*, there are two integers that are used for the nested for loops, *numBlocksX* and *numBlocksZ*. These two variables represent the number of blocks to be created along the x axis and the z axis, respectively. For this program there are four blocks along each row and there are seven rows in total. There is no particular reason behind these numbers other than to fill a pre-made plane.

Inside the nested for loops there are another two integer variables: *blockSpacingX* and *blockSpacingZ*. These integers are used to set the gap in between each block along the x and z axes, respectively. All the integer variables in *ProceduralGrid* cannot be altered in Unity3D as this would mean that the criteria outlined in from research is not met [2:5:3 Criteria]. This is because in this instance the gaps between the blocks using these numbers create equal distance. *BlockAlign* is a Vector3 value that is used to ensure the blocks are aligned with the city plane.

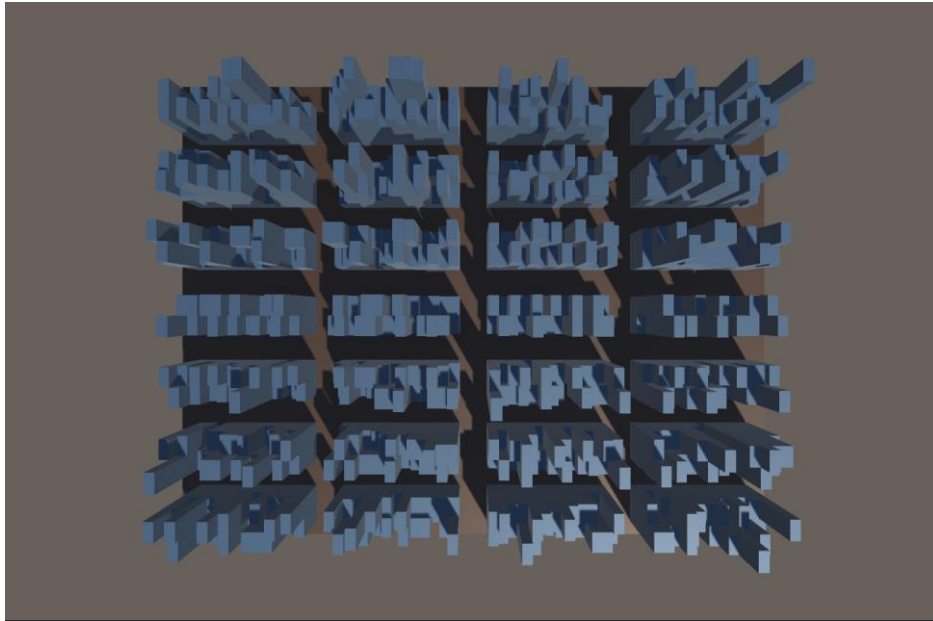


Figure 18 City generated with ProceduralGrid (Top-down)

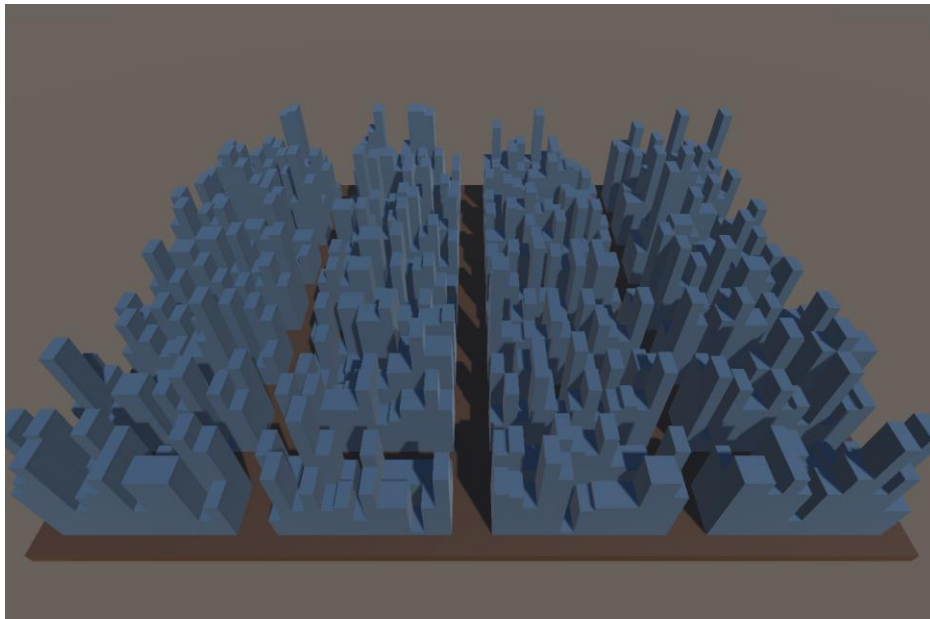


Figure 19 City generated with ProceduralGrid (Side view)

Figure 18 and *19* show an example of an area of a city that is generated using *ProceduralGrid* with all the heights of buildings remaining randomised. The script that was developed at this time is the script that is included in the final program, as implementing Perlin noise did not affect any of this code. During this stage of development, I had successfully implemented a system that on run time generated a city grid which was filled with blocks which were filled with buildings. *Figure 18* and *19* show that the gaps between blocks are evenly spaced. This matches the criteria to allow for evenly sized roads between blocks [2:5:1 Street Plan and 2:5:3 Criteria].

The main area of testing will be with the heights of buildings, so the user can alter the minimum and maximum building heights as well as the Perlin noise value in the interface. Any variables that would

alter the city that meet the criteria cannot be changed, such as the spacing between blocks. The user can also alter the number of blocks along each axis, which could be used in testing as well.

All that was left to do following this was implementing Perlin noise, alter *ProceduralBuildings*, and implement a feature to switch between two cameras using user input.

3:5:4 Perlin Noise Generator

At this stage of development, it was time to implement Perlin noise. After conducting research into the area, I discovered mirrorfishmedia's *ImaginaryCities* [3] which used a Perlin noise heightmap to set heights of buildings in a similar fashion [2:2:2 Procedurally Generated Landscapes and 2:3:1 Lattice Gradient and Perlin noise]. I implemented this technique of applying Perlin noise to a 2D texture into my script *PerlinNoise*.

The Perlin noise generator needed to operate similarly to the existing system, that being there is a maximum height that buildings can be that is multiplied by a float between 0.0 and 1.0. Before implementing Perlin noise this was entirely randomised, so I needed the Perlin noise function to return a float between the same range, except that the floats that are next to each other are related to one another as described in the Perlin noise section [2:3:1 Lattice Gradient and Perlin noise].

```
BEGIN
  TEXTURE2D NEW_TEXTURE
  FOR 'EVERY X COORDINATE IN NEW_TEXTURE' DO
    FOR 'EVERY Y COORDINATE IN NEW_TEXTURE' DO
      FLOAT PERLIN_NOISE(X,Y)
      NEW_COLOUR(FLOAT)
      SET_COLOUR(NEW_TEXTURE, NEW_COLOUR)
    END
  END
END
```

Code 4 Pseudocode to apply Perlin noise to texture

The pseudocode described in *Code 4* works in a similar fashion to the Perlin noise generator in *ImaginaryCities* [3]. It starts by creating a new empty texture. For every coordinate in that texture a Perlin noise value is determined between 0.0 and 1.0. This value is used to obtain a new colour which is applied to the texture in that position. In my program, this function is called *Start*, which is a Unity function that is called before any other function in the script. After all the coordinates have been assigned, the entire texture has been affected by Perlin noise, and is ready for use in setting the heights of buildings.

In order to do this, a second function is required. The purpose of this function is to return a float value that will be between 0.0 and 1.0. This is the value that is multiplied by the maximum height of the building to set the actual height.

```
BEGIN
  GET_COLOUR(X,Y, PERLIN_TEXTURE)
  COLOUR_TO_FLOAT()
  RETURN FLOAT
END
```

Code 5 Pseudocode to obtain Perlin float value

This function takes a x and y coordinate. The coordinates are used to obtain the colour value that is in that position in the 2D texture created that was affected by Perlin noise. This colour value is

converted into a float between 0.0 and 1.0, which is then returned and used to set the height of the building.

In the Building Generator section [3:5:1 Building Generator], I mentioned that the building generator algorithm had to be altered to accommodate for the Perlin noise implementation and was the only script to do so.

```
BEGIN
    SCALE_HEIGHT = MAX_HEIGHT * PERLIN_HEIGHT(X,Y)
    IF 'SCALE_HEIGHT IS GREATER THAN 0' DO
        CREATE(BUILDING)
        SCALE_BUILDING(1, SCALE_HEIGHT, 2)
        POSITION_BUILDING(SCALE_HEIGHT / 2)
    END
END
```

Code 6 Pseudocode to create building with Perlin noise

In the altered pseudocode, the random float is replaced with the float returned from the *PerlinHeight* function. A check is performed to see if the height is greater than zero, if it is then the building is created. This is to avoid buildings being created with a height of zero.

With all the pseudocode created, I could then start to implement them in my actual program.

```
void Start()
{
    perlinInstance = this;
    int xTexture = 65;
    int yTexture = 50;
    perlinTexture = new Texture2D(xTexture, yTexture);
    Vector2 randomNoise = new Vector2(Random.Range(0, 99999), Random.Range(0, 99999));
    for (int x = 0; x < xTexture; x++)
    {
        for (int y = 0; y < yTexture; y++)
        {
            float xCoord = ((float)x / xTexture) * noiseLevel + randomNoise.x;
            float yCoord = ((float)y / yTexture) * noiseLevel + randomNoise.y;
            float perlinValue = Mathf.PerlinNoise(xCoord, yCoord);
            Color colour = new Color(perlinValue, perlinValue, perlinValue);
            perlinTexture.SetPixel(x, y, colour);
        }
    }
    perlinTexture.Apply();
}
```

Code 7 Code to create a 2D texture of Perlin noise

Code 7 shows the *Start* function inside of *PerlinNoise*. When this script is called, an instance of Perlin noise is created. This is used in *ProceduralBuildings*, as an instance is called there to use for the buildings. Following that an empty 2D texture is created and each coordinate is used to create a *perlinValue* using the *Mathf.PerlinNoise* function [2:3:1 Lattice Gradient and Perlin noise]. This value is used to obtain a colour which can then be applied to the 2D texture. This script also contains a function called *PerlinHeight*.

```
public float PerlinHeight(float xPosition, float zPosition)
{
    return perlinTexture.GetPixel(Mathf.FloorToInt(xPosition), Mathf.FloorToInt(zPosition)).grayscale;
}
```

Code 8 Code to return the Perlin noise value

In this function, the x and z coordinates are used against the x and y coordinates of the *perlinTexture*. These coordinates are converted to an integer so they can be used by the *GetPixel* function, which retrieves the value of a colour in a given position. This colour is then converted into a grayscale value, which is a float between 0.0 and 1.0, perfect for the program to use to set the height of buildings.

```
float perlinHeight = PerlinNoise.perlinInstance.PerlinHeight(transform.position.x, transform.position.z);
float maxPerlinHeight;
if (randomHeights)
{
    maxPerlinHeight = minHeight + ((maxHeight - minHeight) * Random.Range(0.0f, 1.0f));
}
else
{
    maxPerlinHeight = minHeight + ((maxHeight - minHeight) * perlinHeight);
}
if (maxPerlinHeight >= 0)
{
    GameObject building = Instantiate(buildingPiece, this.transform.position, transform.rotation);
    if (writeBuildings)
    {
        WriteBuildingHeight(maxPerlinHeight);
    }
    building.transform.localScale = new Vector3(1, maxPerlinHeight, 2);
    building.transform.position += new Vector3(0, maxPerlinHeight / 2, 0);
}
```

Code 9 Code to generate a building using Perlin noise

Code 9 shows the final amendment to the code in *ProceduralBuildings*. In this script, an instance of Perlin noise is created. This effectively creates a 2D texture which can be sampled to obtain a value to set the height of a building. Before that occurs though, a check is performed to see if the user wishes to create randomised building heights or heights set with Perlin noise. This feature is here for testing purposes because although I am testing various Perlin noise values against Manhattan, I believe it would be interesting to also compare the results against entirely randomised buildings. I am testing this as it may be found in results that Manhattan is closer to randomised building distribution as opposed to Perlin noise.

Another check is performed to see if the height of the building is greater than or equal to zero. This is needed so no buildings are created with negative heights. Afterwards, the building is instantiated but another important function is used for testing purposes. This function is *WriteBuildingHeight* and is used to write the heights of buildings to a file. This is needed for testing later as the heights will need to be recorded to allow comparisons to the heights of Manhattan. These heights can be used to obtain different graphs and charts as needed.

Finally, the building is scaled and positioned appropriately.

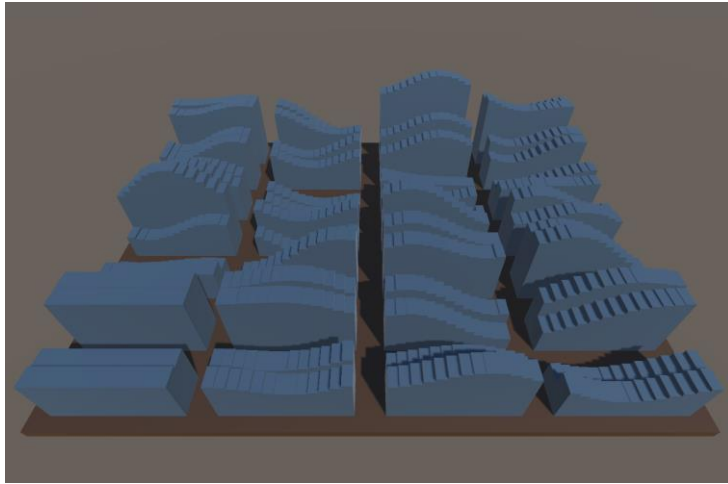


Figure 20 City generated with low Perlin noise level

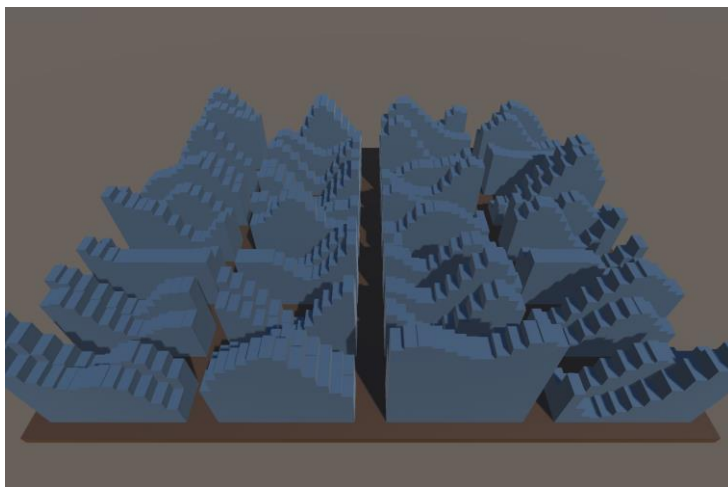


Figure 21 City generated with medium Perlin noise level

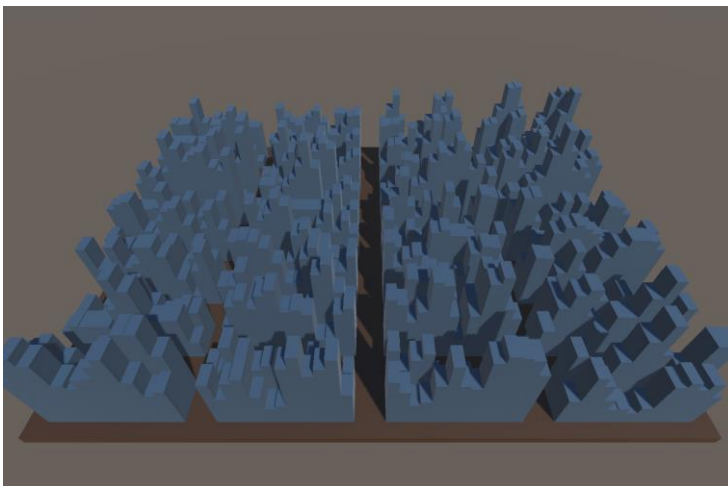


Figure 22 City generated with high Perlin noise level

Figure 20, 21 and 22 demonstrate different examples of a city generated using Perlin noise. The *noiseLevel* variable can be altered within Unity3D to increase the effect of Perlin noise in the program. A lower level of Perlin noise shows the waving effect described in background research

[2:3:1 Lattice Gradient and Perlin noise], which clearly shows the relation between the heights on each building and how they can only be within a certain range of the building near them. As *noiseLevel* is increased the effect is still visible but starts to become less obvious. By the time *noiseLevel* is increased to the level shown in *Figure 22*, any relation between buildings starts to become less noticeable. It is this effect that I shall be investigating and evaluating in the next research using specific sets of values I can change to generate the closest approximation to a Manhattan city structure.

3:5:5 Switch Camera

The purpose of this script is to change the camera perspective with user input. In my program, the user can press the space bar and the angle will change to a top-down perspective.

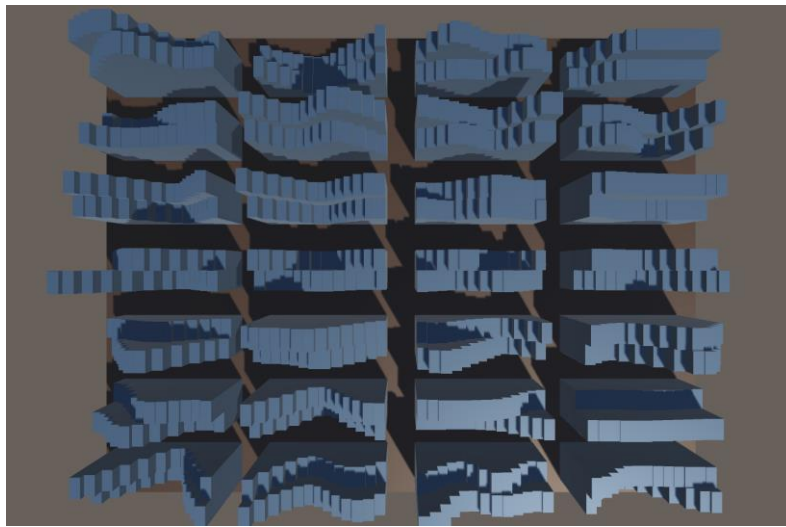


Figure 23 Alternate angle of city generation

This script functions by first enabling the first camera and disabling the second camera on start-up. Then using Unity3D's *Update* function, which is a function that runs every frame, a check is performed to see if the space bar has been pressed, if it has then the enabled states of both cameras are switched around. Thus, swapping the camera perspective.

This feature works great. However, when starting development of the camera I wanted to include a feature so the user could control the camera and fly around the city. Unfortunately, the frame rate of the program dropped significantly as this technique of generating the buildings is not optimised. This made it very difficult for the camera to navigate around the city properly. Thus, the idea was scrapped and replaced with a camera swapping technique.

The camera swapping feature is rather simple but can be used to provide alternate angles of the city that is generated. This may be needed for testing purposes to provide another angle to compare to Manhattan. However, as previously mentioned, this camera feature is not as developed as I wish it could be.

3:6 Summary

Despite being held back by having to restart the dissertation to implement Perlin noise, I am pleased with how the final program was implemented. The program specification was fully completed as the project developed [3:4 Program Specification].

This section still elaborated the course of the development to create a program that can procedurally generate a city using Perlin noise in order to evaluate what Perlin noise level is a close approximation to Manhattan.

The chosen program for the project was also discussed as well as the choice for choosing the program, along with the programming language and code editor.

The process of the development was covered including the implementation of Perlin noise to set the heights of buildings that are generated. The Perlin noise algorithm was shown to have appeared to be implemented into the program, through screenshots showing the waving building heights described in the Perlin noise research [2:3:1 Lattice Gradient and Perlin noise].

4: Results and Evaluation

4:1 Introduction

This section discusses the testing of the Perlin noise algorithm that was implemented against data found from Manhattan. The algorithm is tested with different building heights and Perlin noise levels to determine if Perlin noise is suitable for generating an approximation of Manhattan.

4:2 Testing and Evaluation Technique

Testing for this dissertation will involve drawing comparisons between measurements of actual buildings in Manhattan and the buildings generated in my program.

The purpose of this testing is to satisfy the aim of the dissertation which is to establish if the Perlin noise algorithm can be used to create an approximation of Manhattan. In order to accomplish this testing, I will split it into two parts.

The first part will be to analyse the data provided by *NYCOpenData* [29] outlined in the background research [2:5:2 Blocks], and the data generated from my program. The data provided by *NYCOpenData* measures the buildings in stories. For the purposes of these tests, I shall be using meters as opposed to stories, so all the heights of the buildings in this data will be multiplied by 3.3, as that is how many meters are in a story [30]. By recording the building height distribution of Manhattan, I can use different sets of values in my program to attempt to create a similar distribution. I can also record the mean, median and variance to observe numerical similarities in the averages between the data sets. By the end of this part, I will be able to deduce a set of quantitative values I can use for the next part of testing.

The second part will involve using the data sets found from the first part of testing to approximate Manhattan using different Perlin noise levels. With these data sets, the Perlin noise level will be altered. Each Perlin noise level will be tested ten times. The averages of these tests will be used to record a mean, median, variance and distribution of building heights. The results will be compared to the findings of Manhattan. With these comparisons I aim to determine if any Perlin noise level is suitable to create approximations of Manhattan.

As well as quantitatively determining if Perlin noise produces results close to Manhattan, I will also compare screenshots of the generations that are produced to images of Manhattan. By doing this I can also qualitatively identify if there are any similarities or differences that the quantitative findings could not show.

By the end of these tests, I will be able to evaluate if Perlin noise is a suitable algorithm to create approximations of Manhattan, and if not, I can elaborate on any alterations that need to be made to the algorithm in order to do so.

All testing results will be uploaded as supplementary material in an Excel spreadsheet.

4:3 Finding a Building Height Range

Using the heights provided by *NYCOpenData* [29], the buildings of Manhattan produce the following statistics:

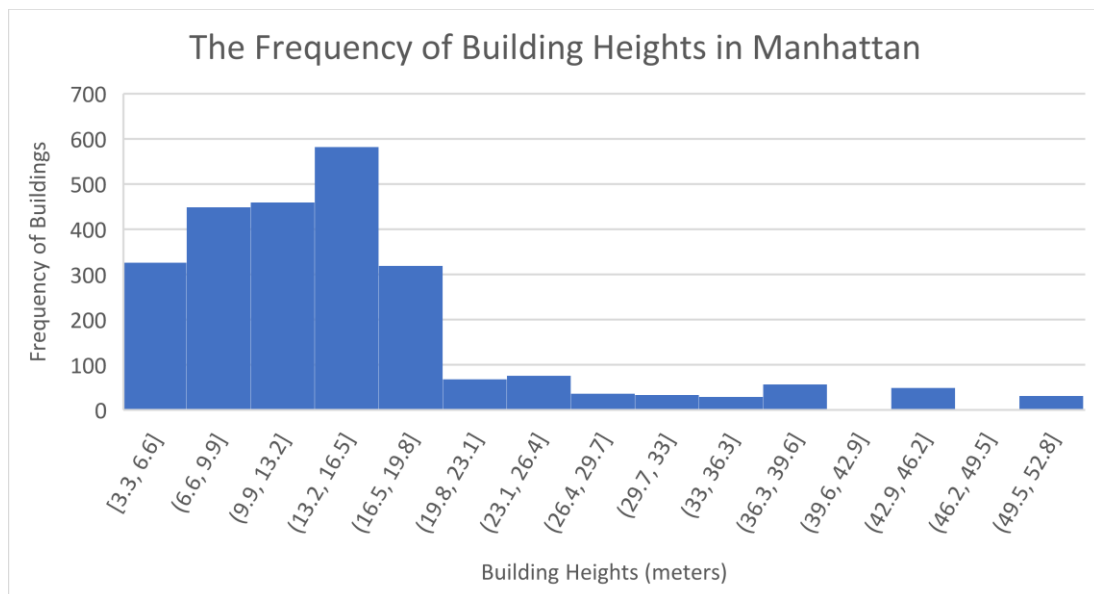


Figure 24 Frequency of Manhattan building heights

Manhattan Building Heights (meters)	
Mean	16.331981
Median	16.5
Variance	83.078969

Table 1 Mean, Median and Variance of Manhattan building heights

Looking at the distribution of the building heights in *Figure 24*, there appears to be a skew to the left, with a sharp drop off past the peak.

Firstly, in order for comparisons to be accurate, a similar sample size is required. Ignoring the outliers, the sample size of the buildings from *NYCOpenData* is 2514 [29]. In my program, a single block includes 26 buildings. If we divide the number of buildings from the data above by 26, we have the number of blocks the program needs to create.

$$2514 / 26 = 96.692307692307692307692307692308$$

I will round this number to 96 as it is simpler to work in even numbers. Having 96 blocks in total allows the program to shape the city with 8 blocks along the x axis and 12 blocks along the z axis, giving 2496 buildings per generation. Each set of values that are tested will be tested ten times to ensure that results are more accurate. A Perlin noise level of 1.0 will be used. This is because altering the Perlin noise level will not alter the distribution of the results, so this value does not matter right now.

For the first set of buildings, I shall use the same range that is used in *NYCOpenData*'s findings [29]. So, the values I will use are:

Values for Procedural City	
Minimum Building Height	3
Maximum Building Height	53
Number of Blocks on X axis	8
Number of Blocks on Z axis	12

Table 2 Values for city generation between 3 and 53 meters

Using these values, the following results are obtained:

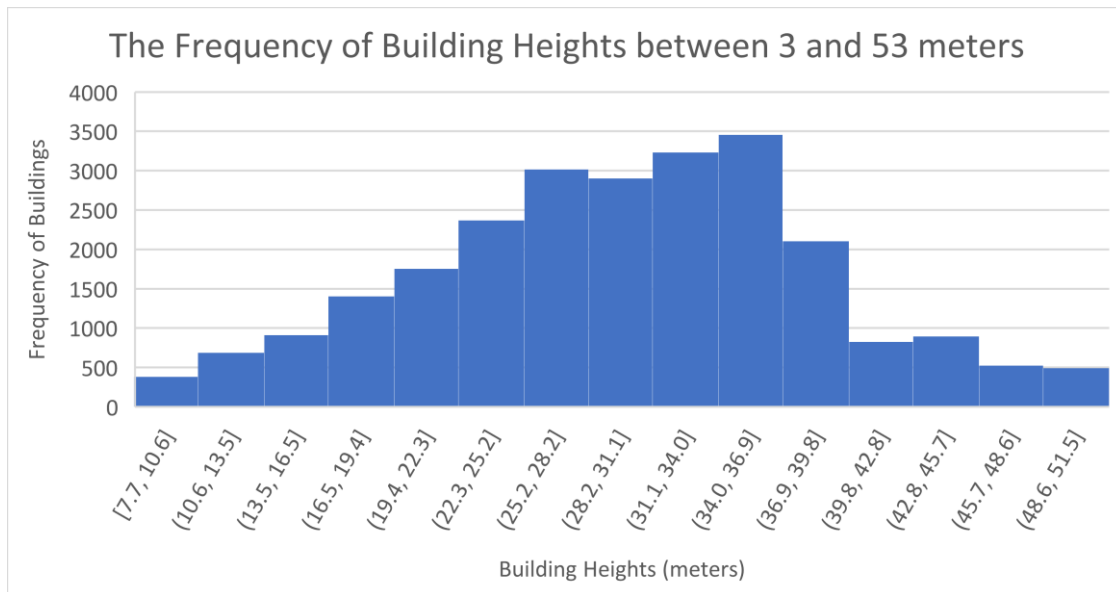


Figure 25 Frequency of building heights between 3 and 53 meters

Building Heights between 3 and 53 meters	
Mean	29.67145
Median	30.12157
Variance	78.67844

Table 3 Mean, Median and Variance of building heights between 3 and 53 meters

Comparing the distribution found in *Figure 24*, we can see that Manhattan does not include a normal distribution. Whereas the distribution in *Figure 25* appears to be under normal distribution, which matches the description of Perlin noise with its waving appearance described in the Perlin noise research [2:3:1 Lattice Gradient and Perlin Noise].

Taking a look at the mean, median and variance of *Table 1* and *2* we can see that the mean and median greatly differ. However, the variance values in each set are close together. This is due to the fact that the sets of data are both within an identical range. So, if we use a minimum height of 3 meters and a maximum height of 53 meters, we can generate a city that has a similar variance.

On top of this, if we accept the fact that the Perlin noise algorithm is distributed normally, that means the mean and median will lie close to the middle of the range. So, if the mean and median of Manhattan's buildings are both around 16, then this value will lie in the centre of the minimum and maximum building heights we can use. If we still set the minimum building height to 3, then the

maximum building height would be 35. This means we can produce a city that has a similar mean and median building height as Manhattan.

So, the values we can test next are:

Values for Procedural City	
Minimum Building Height	3
Maximum Building Height	35
Number of Blocks on X axis	8
Number of Blocks on Z axis	12

Table 4 Values for city generation between 3 and 35 meters

Using these values, we get the following results:

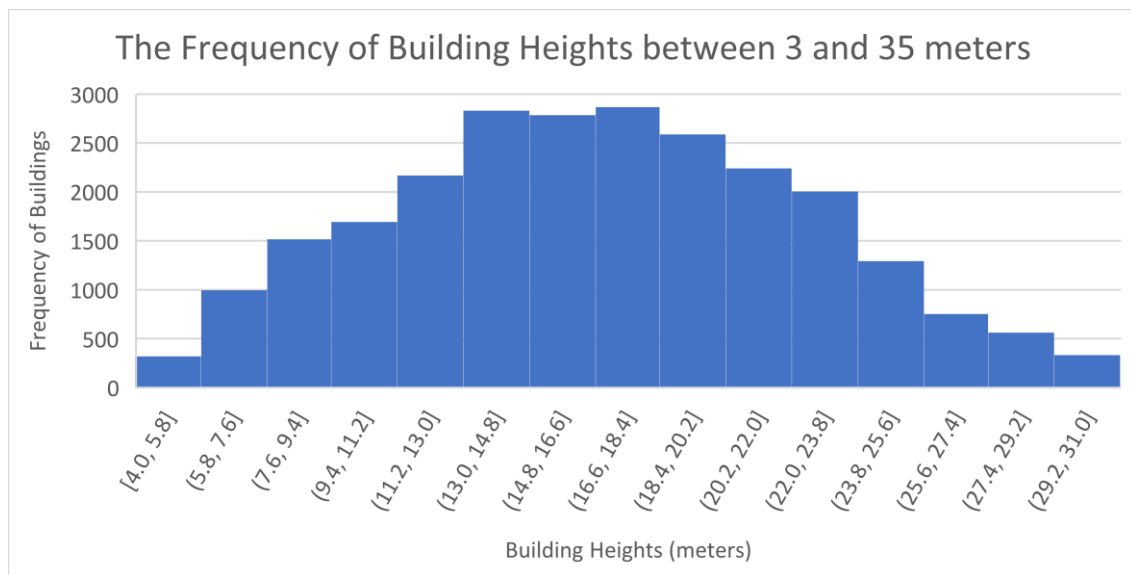


Figure 26 Frequency of building heights between 3 and 35 meters

Building Heights between 3 and 35 meters	
Mean	16.77829
Median	16.58824
Variance	32.6284

Table 5 Mean, Median and Variance of building heights between 3 and 35 meters

Looking at these results, again the buildings appear to be under normal distribution. However, this time the mean and median of the building heights are nearly the same as the mean and median building heights of Manhattan. As the range of building heights was decreased, the variance naturally dropped significantly.

This does now mean though that we have two sets of values that we can use to give approximations of Manhattan using Perlin noise. One set of values approximates Manhattan with a similar variance and building height range. The other set approximates Manhattan with a similar mean and median.

Unfortunately, as discussed earlier, the distribution of the Perlin noise algorithm cannot be altered. So, the distribution shown in the building heights of Manhattan cannot be recreated using the algorithm I have in my program. This is something I will elaborate more on for future improvements.

With these values in place, I can continue onto the second part of the testing.

4:4 Noise Levels

From the first part of testing, we have established two sets of fixed values we can use to approximate Manhattan. There will be a city grid with 8 blocks along the x axis on 12 rows on the z axis, making 96 blocks in total.

One set of values will use a minimum and maximum building height of 3 and 53 meters, respectively. The results from these tests should produce cities with a similar variance but also use the same building height range to that of Manhattan.

The other set of values will use a minimum and maximum building height of 3 and 35 meters, respectively. The results from these tests should produce cities that have a similar mean and median to that of Manhattan.

So, for these tests I will be using the following Perlin noise levels:

Perlin Noise Levels for Testing
0.5
1.0
2.5
5.0
10.0
25.0
Random

Table 6 Perlin noise levels for testing

I have selected these levels to test as I believe they will show enough variety to produce interesting results. The values start with 0.5, which will produce very minimal changes to the heights of buildings but I feel is worth testing as it is an interesting comparison. This is also the case with a level of 25.0 and random. I feel that using a value of 25.0 is quite high and could possibly push the program to its breaking point. Not in terms of the program crashing, but more for the values that will be produced. Random is being used for comparison purposes to Manhattan and with the other Perlin noise levels. This value will ensure that the heights of buildings are entirely randomised, so there will be no relation between any heights at all.

Each of the levels listed will be tested ten times, for both data sets of building height ranges. So, now we can test these values and obtain some results.

4:5 Evaluation

This section includes the results of the Perlin noise levels established in the previous section.

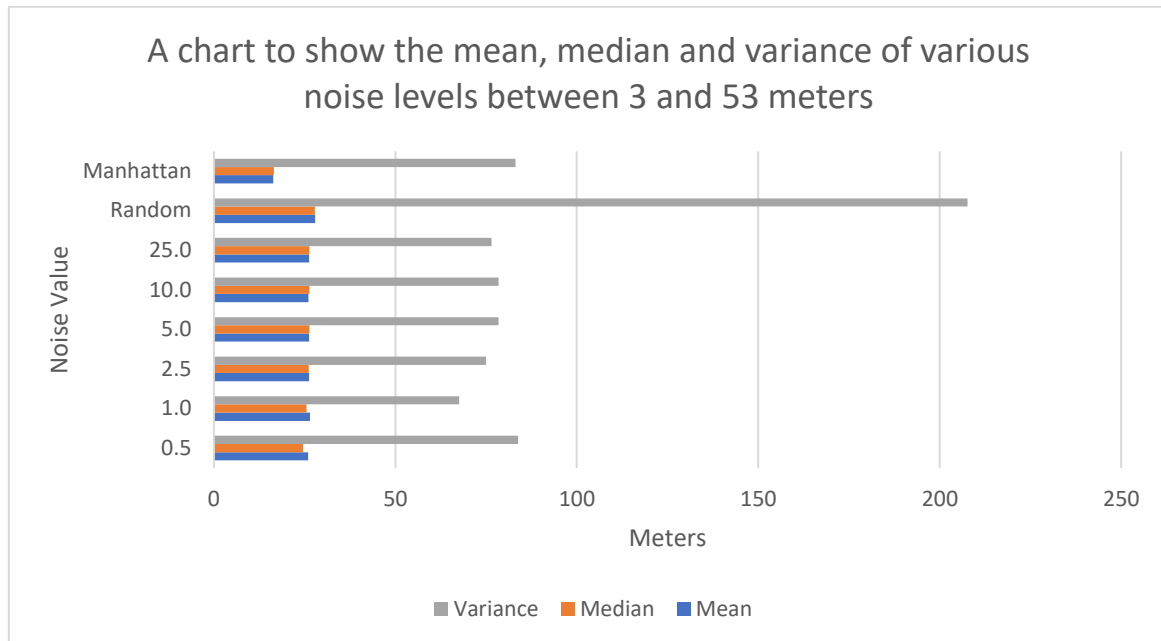


Figure 27 Comparison of the Mean, Median and Variance of Noise levels between 3 and 53 meters

As the same range is being used in the program as the range of heights in Manhattan, the variance was expected to remain at a similar level. Looking at *Figure 27*, we can see that the variance for each noise value does remain at a similar level. Except for random whose variance is significantly higher, although this is also to be expected. Each of the noise values used has a higher mean and median than that of Manhattan, and we can see that there is not much change when using different values. Again, this was to be expected as changing the scale of the noise does not affect the distribution of the algorithm. So, from these results we can see that using these values only gives a similar variance, but nothing else.

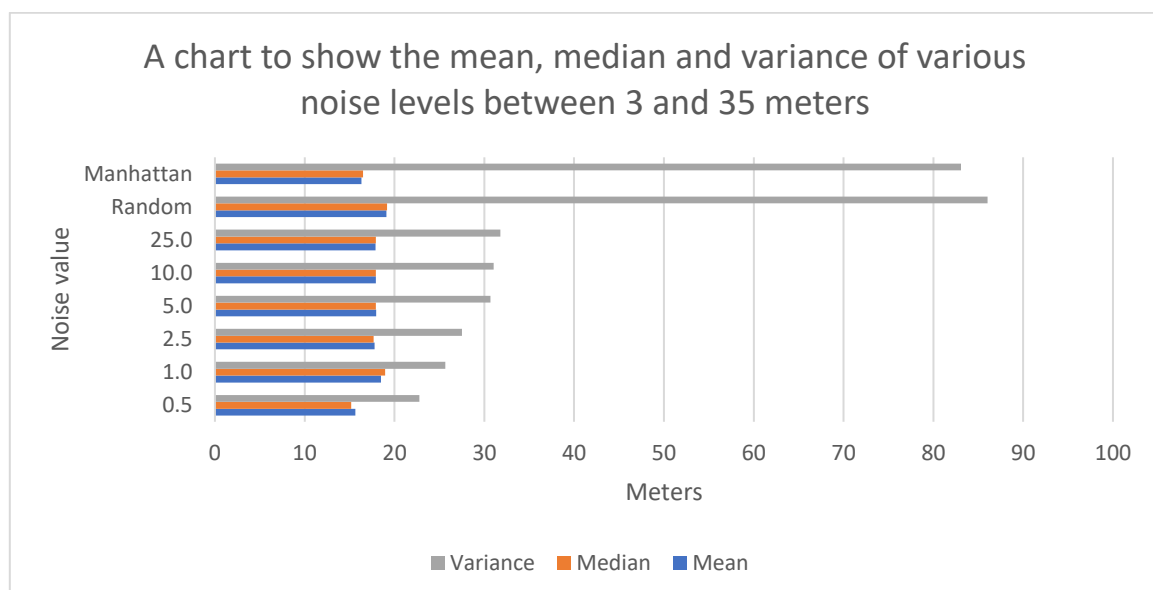


Figure 28 Comparison of the Mean, Median and Variance of Noise levels between 3 and 35 meters

Figure 28 shows the results from a range of 3 to 35 meters. These values were expected to produce a mean and median that would remain at a similar level with nothing else changing. Looking at Figure 28 we can see that each of the noise values produces a similar mean and median as expected, with the variance being significantly different. However, one interesting takeaway from this chart is that when using random building heights between 3 and 35 meters, a very similar mean, median and variance is produced. Each of these results appear to only be a few meters higher than that of Manhattan, so if the test with random was completed again with a range of 0 to 32 meters, it is possible that near identical numerical results would be produced.

Overall, from these two charts we can conclude that Manhattan does not share a similar relation with Perlin noise with the mean, median and variance of the building heights. However, using random building heights does produce a very similar quantitative result.

Next, we can observe the distribution of building heights amongst Manhattan and various noise levels.

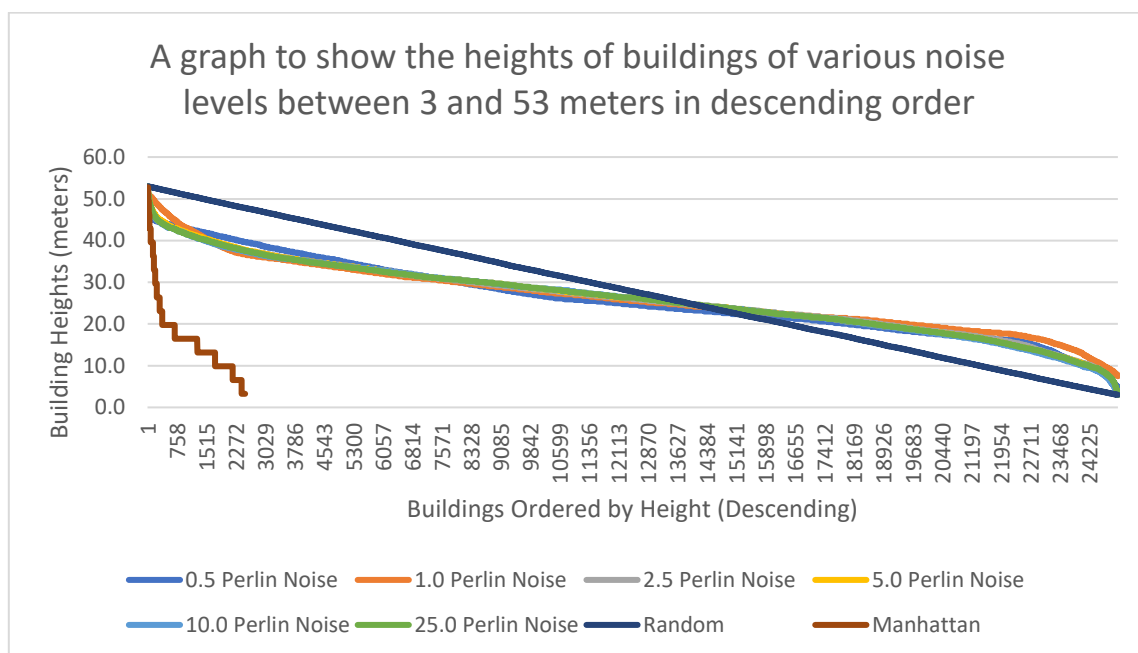


Figure 29 Comparison of the frequency of building heights of Noise levels between 3 and 53 meters

Figure 29 shows the heights of buildings in meters for each of the noise levels between 3 and 53 in descending order. As random values are uniformly distributed, there is a straight uniform line for the buildings set with a random height. Perlin noise is normally distributed, so curves can be observed with all the buildings set with a Perlin noise level. However, we can see that none of the buildings produced with Perlin noise share any similarities with the distribution of buildings for Manhattan. Manhattan shows a sharp drop from the tallest buildings before levelling out with buildings under 20 meters tall.

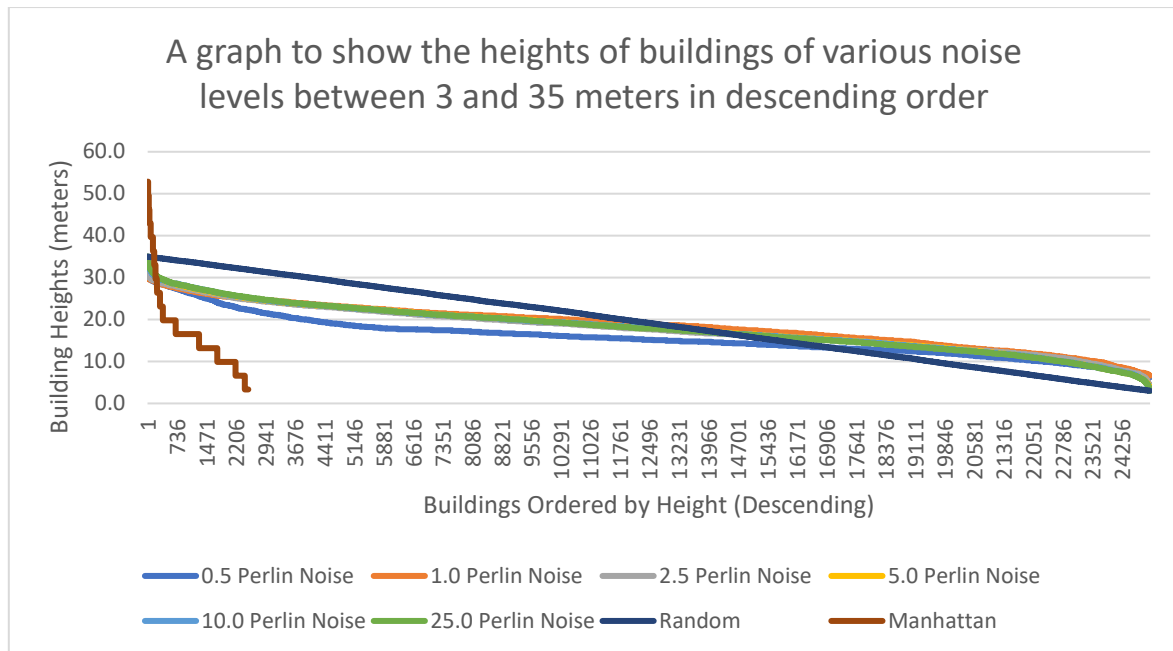


Figure 30 Comparison of the frequency of building heights of Noise levels between 3 and 35 meters

Looking at the data shown in *Figure 30*, we can see again that there is no correlation between the building heights of Manhattan and that of buildings generated with any noise level.

After collecting these results, we can quantitatively conclude that the Perlin noise algorithm is not suitable for generating a close approximation to Manhattan. However, using these figures we can make slight alterations to the Perlin noise algorithm that would be suitable for Manhattan.

Firstly, looking at the mean, median and variance values in *Figure 27* and *28*. We can see that the mean and median of Manhattan buildings are typically lower than that of buildings produced using Perlin noise. In order for a Perlin noise algorithm to produce a closer approximation to Manhattan it would need to ensure that the mean and median lie closer to the minimum building height than the middle.

Secondly, looking at the graphs in *Figure 29* and *30*, we can see that there are very few buildings that have heights close to the maximum building height. The altered Perlin noise algorithm would need to generate very few building heights that lie closer to the maximum building height, with the majority of buildings being closer to the minimum building height, as shown by the mean and median.

Finally, an alteration would be needed for the variance of buildings using Perlin noise. However, due to the Perlin noise algorithm operating by producing values that are related to values next to it, this may be quite difficult to achieve. But with the changes described to the mean, median and distribution this would help increase the variance naturally closer to the variance of Manhattan.

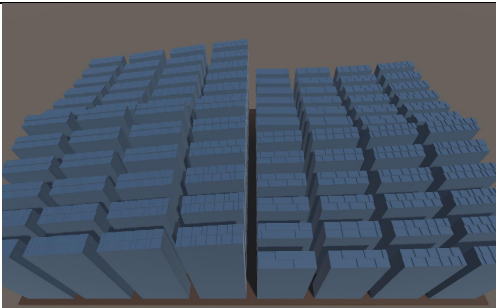
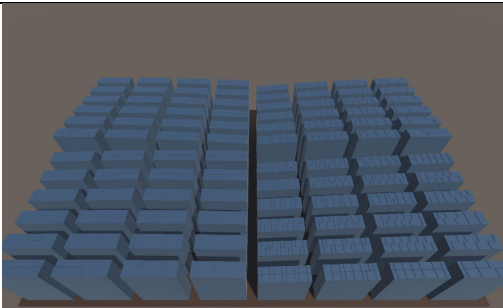
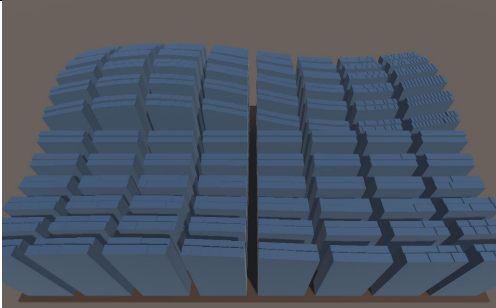
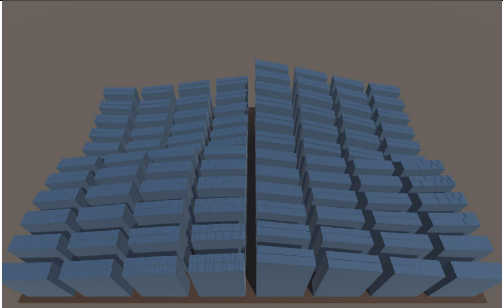
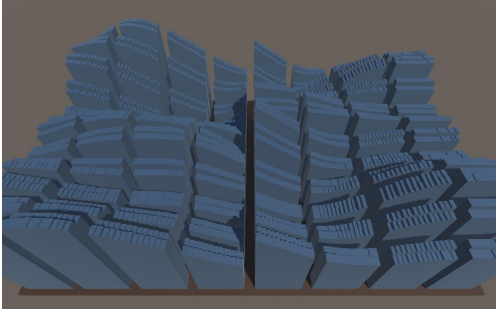
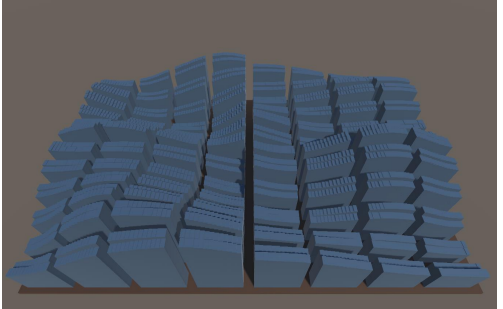
4:7 Screenshots of Procedural Cities

This section includes a table of screenshots of the different city shapes that can be generated using different Perlin noise levels.

Despite quantitatively concluding that the Perlin noise algorithm cannot be used to generate an approximation of Manhattan, we can still observe the screenshots of the different building heights. By doing this we can see if there are any qualitative similarities between the Perlin noise buildings and buildings of Manhattan.

I will first show the procedurally generated buildings in a table then show another table using images I have found that look close enough to draw some comparisons.

All images can be seen in full resolution in the appendix.

Screenshots of Procedurally Generated Cities		
Noise Level	3-53 meters	3-35 meters
0.5		
1.0		
2.5		

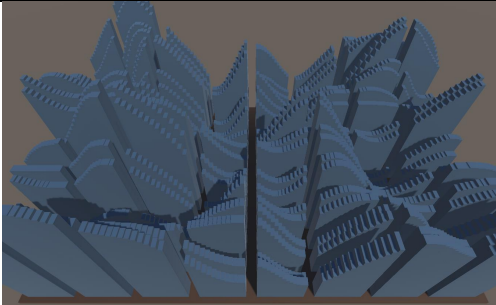
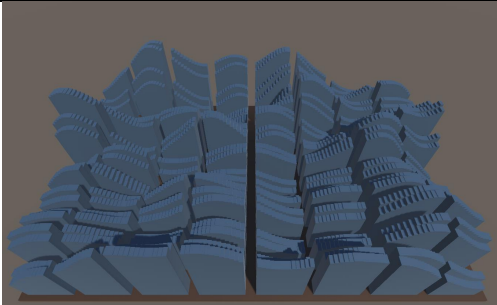
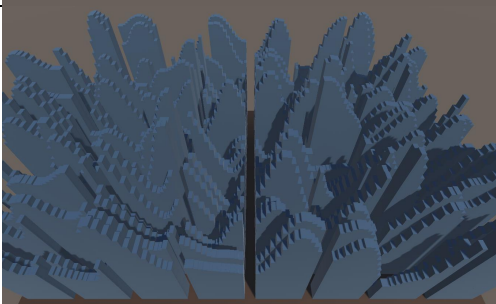
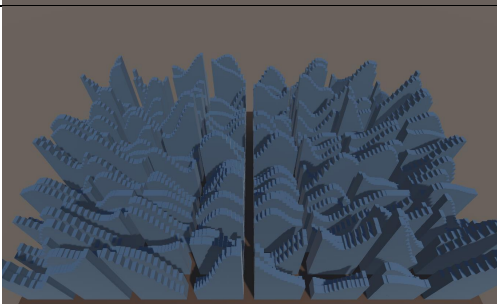
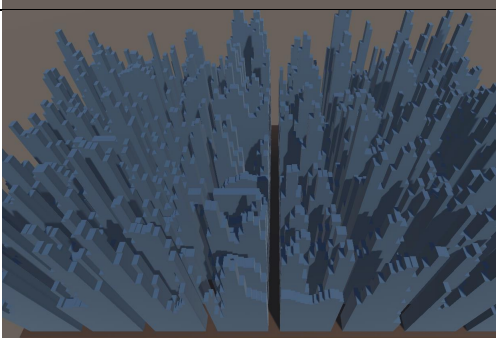
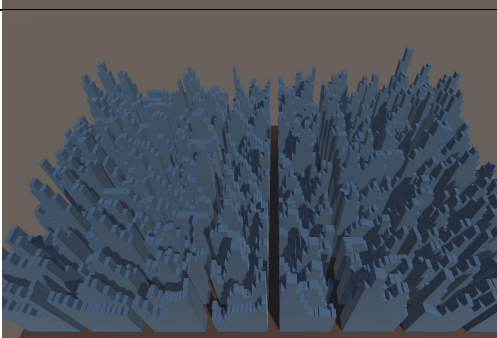
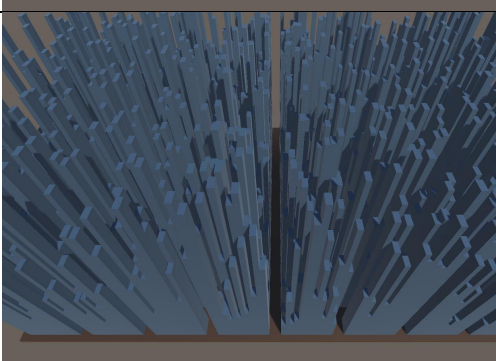
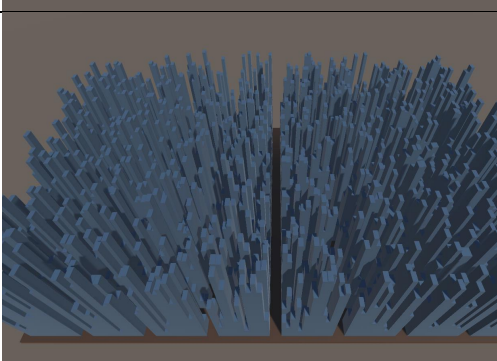
5.0		
10.0		
25.0		
Random		

Table 7 Screenshots of Perlin Noise Cities

Images of Manhattan	
Image	Reference
 An aerial photograph of Lower Manhattan, New York City, showing a dense cluster of skyscrapers. The Empire State Building is prominent on the right side of the frame. The image is taken from a high angle, looking down on the city.	[31]
 An aerial photograph of Midtown Manhattan, New York City, showing a dense cluster of skyscrapers. The image is taken from a high angle, looking down on the city.	[32]
 An aerial photograph of Upper Manhattan, New York City, showing a dense cluster of skyscrapers. The image is taken from a high angle, looking down on the city.	[33]

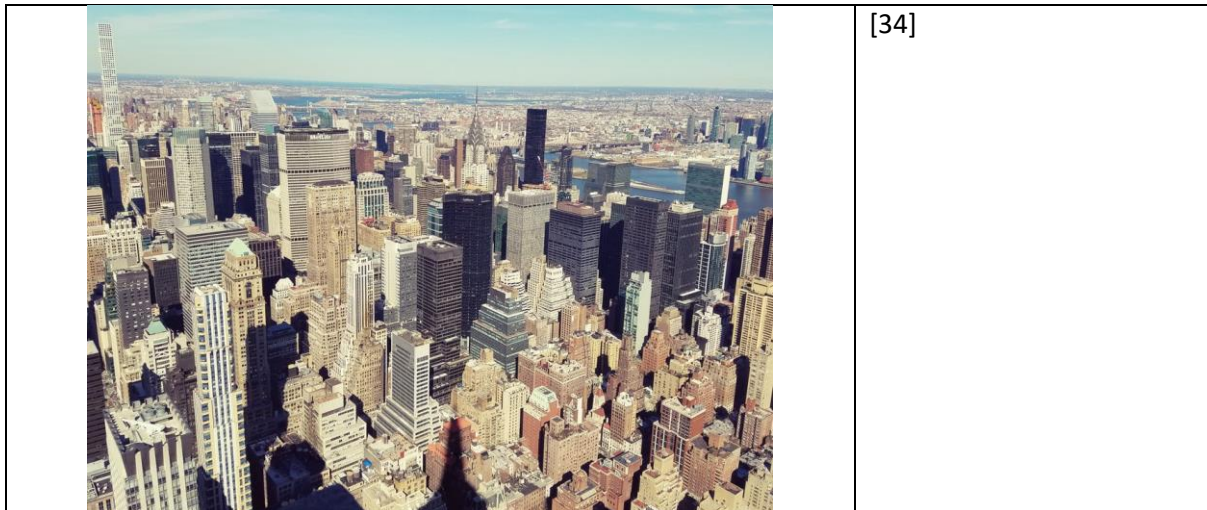


Table 8 Images of Manhattan

From analysing these results, I can conclude that neither the Perlin noise algorithm nor just using random building heights produce results that appear close to Manhattan. This is backed up by the numbers found from testing, as the distribution of buildings is what needs to be altered.

However, from the findings it was shown that buildings between 3 and 35 meters that are randomly generated have a similar mean, median and variance to that of Manhattan.

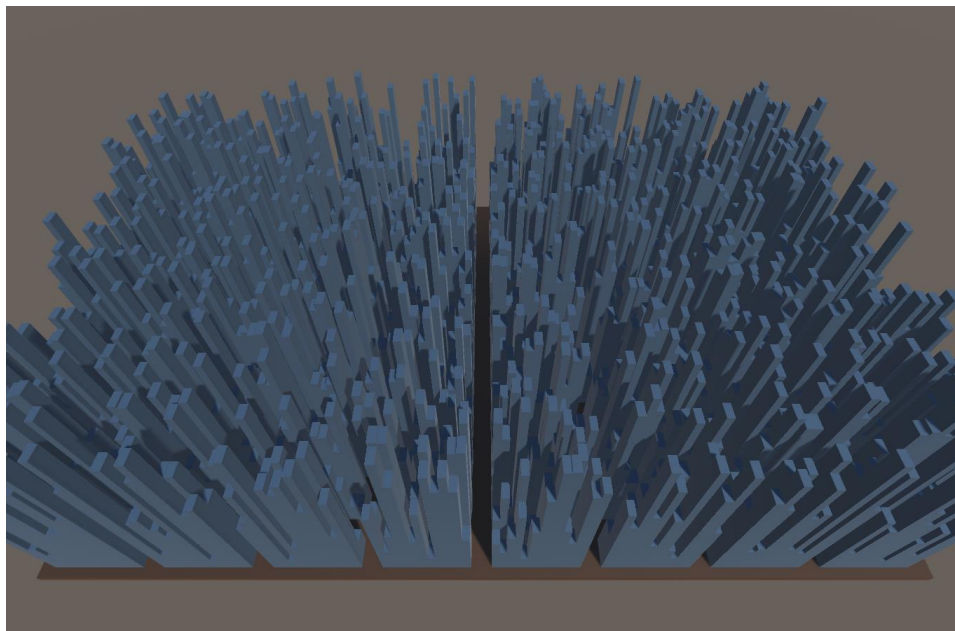


Figure 31 Random buildings between 3 and 35 meters

When comparing *Figure 31* to the images shown in *Table 8* it is still difficult to draw any similarities. However, there is one observation I can make from these screenshots that is not present in the quantitative findings. In my program and shown in *Figure 31*, all the buildings are just set as 1x2 lots, so all the buildings appear to be very thin, especially when they are at the tallest heights. Looking at the images of Manhattan, the tallest buildings are generally cube shaped rather than rectangular and thin. This is something that my program did not include a function for and would need to be included in the altered algorithm. I believe the cities produced in my program would appear much

closer to Manhattan if there was code in place to ensure that tall buildings were grouped together to form a single building.

4:8 Summary

In this section we have discussed the findings from the different minimum and maximum buildings heights along with the different Perlin noise values used in my program.

It was concluded through quantitative results that Perlin noise is not suitable for creating an approximation of Manhattan. However, there are alterations that could be made to the algorithm to ensure better results.

Looking at the results, if the mean and median of the Perlin noise algorithm was closer to the minimum building height then it would have a closer appearance to Manhattan. Each generation would also need to include very few buildings that are close to the maximum building height.

The results also showed that using a random building height in between 3 and 35 meters would generate a city with a similar mean, median and variance.

However, from analysing the screenshots of each city produced and comparing to images of Manhattan, we could see that in Manhattan the tallest buildings are generally cube shaped. Whereas in my program, as buildings are stretched from a 1x2 shape, all the buildings appear very thin, which does not look like Manhattan. This is another alteration that would need to be made to the program to ensure a closer approximation.

5: Conclusion

This section is a conclusion to the aim and objectives created in the Introduction section (1:3 Aim and Objectives), which is based upon if the objectives have been met to satisfy the aim. I also discuss how the dissertation went, as well as any improvements I feel could have been made and what future work can be completed.

5:1 Satisfaction of the Aim and Objectives

The overall aim of the dissertation was to:

“Establish if the Perlin noise algorithm can be used to procedurally generate an accurate approximation of Manhattan”

Depending on how the following objectives have been met, it will establish if this aim has been reached.

“To investigate the Perlin noise algorithm and how it can be used in procedural city generation”

This objective has been met as my final program includes Perlin noise which demonstrates the same effects seen in the patterns of the heights of the buildings. Heightmaps were discovered to have been used in games such as Minecraft to set the shape of the landscape. A heightmap technique outlined by mirrorfishmedia [3] was successfully implemented using Unity3D's `Mathf.PerlinNoise` function to set the heights of buildings in the final program.

“To investigate the Manhattan city structure to define a set of criteria a building generation algorithm must follow”

This objective has been partly met with the following criteria:

- Streets run along the x axis
- Avenues run along the z axis
- Blocks are evenly spaced along the x and z axes
- Blocks are 13 lots wide and 2 lots deep
- A lot is 1 unit wide and 2 units deep

One aspect that was not met was the shape of the buildings. Whilst it was researched that buildings take up lots in blocks, it was not in the code that a single building can take up different lots to form different shaped buildings. The effect this had on the final program was that buildings generally looked too thin, which does not represent the appearance of Manhattan. As for the shape of blocks and roads, this part of the objective was met.

“To develop a program that procedurally generates a city structure with Perlin noise”

This objective has been fully met after implementing mirrorfishmedia's technique of a Perlin noise heightmap. Using Unity3D's Prefab system, nested Prefabs were used to generate a buildings within blocks, and blocks within a city grid. The Perlin noise heightmap was successful in setting the heights of the buildings that were generated.

“To test at least 5 levels of Perlin noise to compare similarities between Perlin noise building distribution and Manhattan building distribution”

This objective was also fully met by testing Perlin noise values of 0.5, 1.0, 2.5, 5.0, 10.0 and 25.0 which pushed the limit of the Perlin noise algorithm. Random building heights were also tested for comparison purposes. The results showed that none of the Perlin noise values or the random building heights shared a similar building distribution to that of Manhattan. However, a discussion was made regarding future work that can be made to alter the Perlin noise algorithm to allow closer approximations to Manhattan using Perlin noise.

5:2 Reflection

5:2:1 Project Conclusion

Despite the fact that Perlin noise was discovered to not be appropriate for generating approximations of Manhattan, the aim of the dissertation was merely to establish if it was possible to. In addition to showing that Perlin noise is not an appropriate use for generating buildings with similar distributions to Manhattan, a discussion was had about future work that can be completed in order to alter the Perlin noise algorithm to distribute building heights in a similar fashion to Manhattan.

Whilst there were various aspects of the project that could be improved upon, the objectives were fully met and the aim realised.

5:2:2 What has been learnt?

In conclusion the author is pleased with the outcome of the project. Procedural generation can be widely adapted in game development. By undertaking research into the field, the author has gained a solid understanding of the basics of procedural generation, along with experience of the application of Perlin noise, which has significant importance in many popular games such as Minecraft. Despite not obtaining the results that were wanted, the author takes pride in knowing the research and implementation was not in vain by understanding how there is an opportunity for future work to be built off what has been put in place.

5:2:3 What could have been done differently?

Even though Perlin noise was proved to not be appropriate for procedurally generating Manhattan, that does not mean that other generation techniques are not. Existing techniques such as Simplex noise and Value noise exists which are built on the same premise of randomness but could perhaps have shown different results for this project. Had the author focused the project on other techniques as well as Perlin noise, then other results could have been included which may be fit for procedurally generating Manhattan.

Various aspects of the program could also have been developed differently to improve usability with a user interface. The variables in the program can only be changed via the Unity3D interface rather than including any input options when in the game scene. A slider to increase and decrease the level of Perlin noise or a textbox to input an integer for the building heights would have made altering generations much easier. Instead, this cost quite a lot of time during testing having to awkwardly keep restarting the game scene and changing the values in Unity3D. In future there needs to be more consideration for the end user who may be new to the program.

5:2:4 What remains to be done?

As previously mentioned, there are slight alterations that could be made to the existing Perlin noise algorithm that could produce accurate approximations of Manhattan. An altered algorithm could shift the distribution of Perlin noise so the majority of points are closer to the minimum value, which in turn shifts the mean and median too. If there are also very few points towards the maximum value then the algorithm would fit for procedurally generating Manhattan.

For added detail in this testing, buildings could include modelling and texturing as discussed in the background research section [2:4:2 L-systems for Modelling Cities and 2:4:3 Texturing]. This would have proved useful when comparing screenshots of the procedurally generated cities to pictures of Manhattan for improved qualitative results.

References

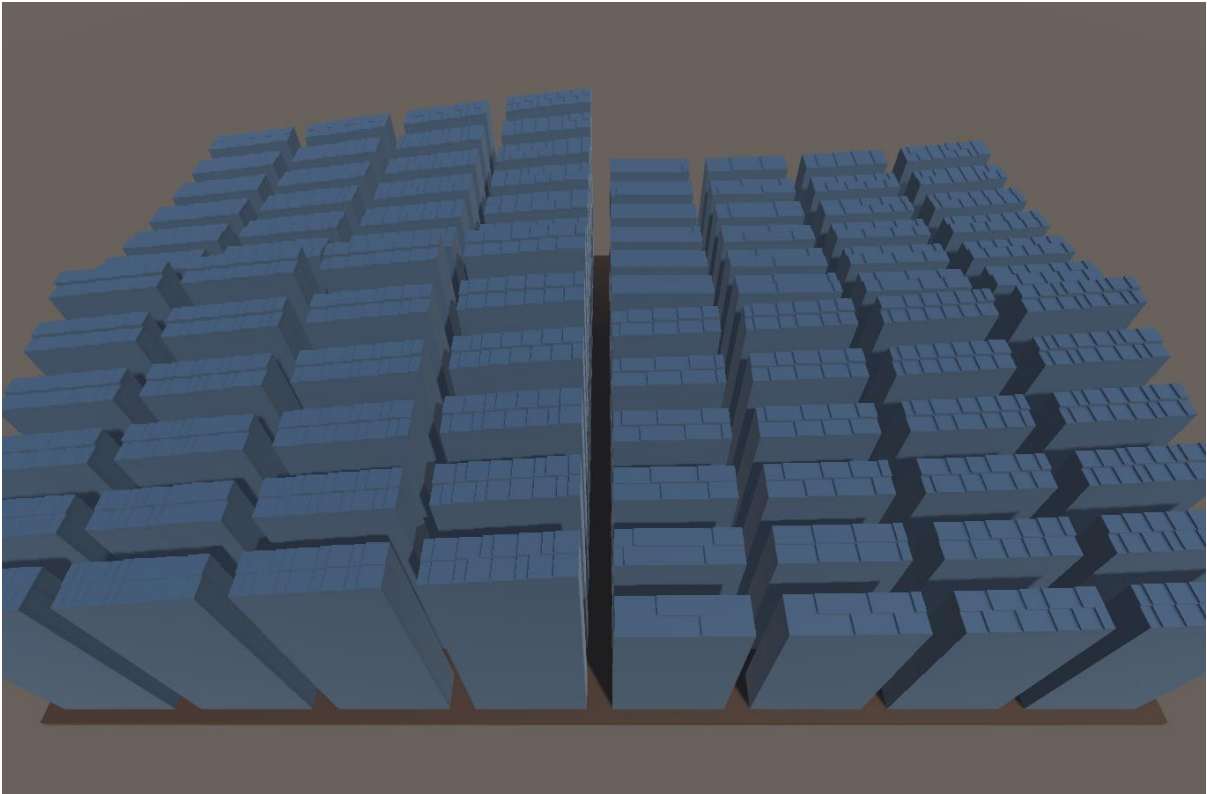
- [1] Marvel. (2018) *Marvel's Spider-Man*. Available at: <<https://www.marvel.com/games/marvel-s-spider-man>> [Accessed: 05 May 2021]
- [2] Mojang Studios. (2009) *Minecraft*. Available at: <<https://www.minecraft.net/en-us/>> [Accessed: 26 April 2021]
- [3] Mirrorfishmedia. (2019) *ImaginaryCities*. Available at: <<https://github.com/mirrorfishmedia/ImaginaryCities>> [Accessed: 25 April 2021]
- [4] Smith, Gillian. (2015) 'An Analog History of Procedural Content Generation', *Playable Innovative Technologies Lab*, Available at: <<http://sokath.com/home/wp-content/uploads/2018/01/smith-fdg15.pdf>> [Accessed: 26 February 2021]
- [5] Yu, Derek. (2008) *Spelunky*. Available at: <<https://spelunkyworld.com/>> [Accessed: 24 April 2021]
- [6] Motion Twin. (2017) *Dead Cells*. Available at: <<https://dead-cells.com/>> [Accessed: 24 April 2021]
- [7] Smith, A.J. & Bryson, J.J. (2014) 'A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. *Proceedings of the 50th Anniversary Convention of the AISB*.
- [8] İzgi, E. (2018) *Framework for Roguelike Video Games Development*.
- [9] Hello Games. (2018) *No Man's Sky*. Available at: <<https://www.nomanssky.com/>> [Accessed: 24 April 2021]
- [10] Scimeca, Dennis. (2020) *This is how No Man's Sky creates over 18 quintillion planets*. Available at: <<https://www.dailydot.com/parsec/no-mans-sky-procedural-generation-galaxy-creation/>> [Accessed: 24 April 2021]
- [11] Fischer, Roland. Dittmann, Philipp. Weller, René & Zachmann, Gabriel. (2020) *AutoBiomes: procedural generation of multi-biome landscapes*. Available at: <<https://link.springer.com/article/10.1007/s00371-020-01920-7>> [Accessed: 25 April 2021]
- [12] Lagae, A. Lefebvre, S. Cook, R. DeRose, T. Drettakis, D.S. Lewis, J.P. Perlin, K. Zwicker, M. (2010) 'A Survey of Procedural Noise Functions', *The Authors Computer Graphics Forum*. Available at: <<https://onlinelibrary.wiley.com/doi/full/10.1111/j.1467-8659.2010.01827.x>> [Accessed: 26 April 2021]
- [13] Perlin, Ken. (1985) 'An image synthesizer', *ACM SIGGRAPH Computer Graphics*. Available at: <<https://dl.acm.org/doi/10.1145/325165.325247>> [Accessed: 26 April 2021]
- [14] Notch. (2011) *Terrain generation, Part 1*. Available at: <<https://notch.tumblr.com/post/3746989361/terrain-generation-part-1>> [Accessed: 28 February 2021]
- [15] Unity. (2020) *Mathf.PerlinNoise*. Available at: <<https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html>> [Accessed: 27 April 2021]

- [16] Kelly, George. & McCabe, Hugh. (2007) 'Citygen: An Interactive System for Procedural City Generation', *Department of Informatics*, Available at:
<http://www.citygen.net/files/citygen_gdtw07.pdf > [Accessed 01 March 2021]
- [17] Parish, H. Müller, Pascal. (2001) 'Procedural Modeling of Cities', *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, Available at:
<https://cgl.ethz.ch/Downloads/Publications/Papers/2001/p_Par01.pdf > [Accessed: 13 March 2021]
- [18] Astrid, L. (1968) 'Mathematical models for cellular interaction in development. Parts I and II.', *Journal of Theoretical Biology*, 18, pp.280-315
- [19] Lindenmayer, A. Prusinkiewicz, P. (1990) *The algorithmic beauty of plants* (Vol. 1). New York: Springer-Verlag
- [20] Greuter, Stefan. Parker, Jeremy. Stewart, Nigel. Leach, Geoff. (2003) 'Real-Time Procedural Generation of Pseudo Infinite Cities', *Association for Computing Machinery*, Available at:
<<https://dl.acm.org/doi/abs/10.1145/604471.604490> > [Accessed: 13 March 2021]
- [21] Unity. (2020) *Prefabs*. Available at: <<https://docs.unity3d.com/Manual/Prefabs.html> >
[Accessed: 28 April 2021]
- [22] Creative Commons Attribution-Share. (2012) *Grid Plan*. Available at:
<<https://jokesels.files.wordpress.com/2012/11/grid-plan.pdf> > [Accessed: 13 March 2021]
- [23] Walks of New York. (2014) *Manhattan Streetsmarts*. Available at:
<<https://www.walksofnewyork.com/blog/manhattan-streetsmarts> > [Accessed: 13 March 2021]
- [24] B.P. (2012) *All New York City Streets Are Not Created Equal*. Available at:
<<http://stuffnobodycaresabout.com/2012/11/19/all-new-york-city-streets-are-not-created-equal/> >
[Accessed: 27 March 2021]
- [25] Solomon, Zachary. (2018) *How Many NYC Blocks Are in a Mile?* Available at:
<<https://streeteasy.com/blog/how-many-nyc-blocks-are-in-one-mile/> > [Accessed: 27 March 2021]
- [26] Anonymous. (2018) 'City block'. *The HouseShop*. Available at:
<<https://www.thehouseshop.com/property-blog/city-block/> > [Accessed: 27 March 2021]
- [27] Flagg, Ernest. (1894) 'The New York Tenement-House Evil and Its Cure', *Scribner's magazine* : v.16 (1894:July-Dec.). New York (State): Charles Scribners Sons, 1887, pp. 108-117. Available at:
<https://books.google.co.uk/books?id=EEVRNIVR5mkC&lpg=PA108&ots=JKLcDKSYN1&dq=the+wors+t+curse+which+ever+afflicted+any+great+community&pg=PA108&redir_esc=y#v=onepage&q&f=false > [Accessed: 26 March 2021]
- [28] Emporis. (2021) *Buildings in New York City (existing)*. Available at:
<<https://www.emporis.com/city/101028/new-york-city-ny-usa/status/existing/1> > [Accessed: 27 April 2021]
- [29] NYOpenData. (2021) *Local Law 44 – Building*. Available at:
<<https://data.cityofnewyork.us/Housing-Development/Local-Law-44-Building/hu6m-9cfi> >
[Accessed: 02 May 2021]

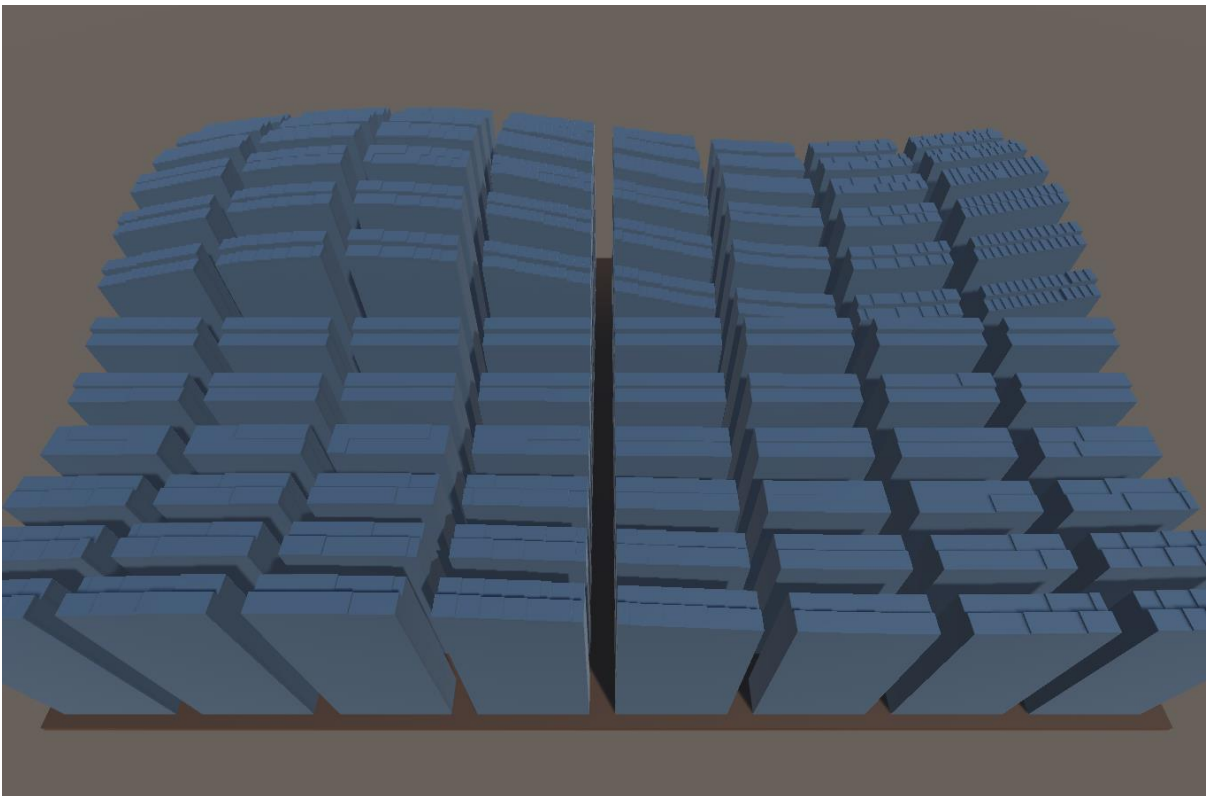
- [30] ConvertUnits. (2021) *Convert meters to story – Conversion of Measurement Units*. Available at: <<https://www.convertunits.com/from/meters/to/story>> [Accessed: 05 May 2021]
- [31] Parker, Charles. (2020) *Residential and commerce buildings placed in downtown of megapolis*. Available at: <<https://www.pexels.com/photo/residential-and-commerce-buildings-placed-in-downtown-of-megapolis-5847765/>> [Accessed: 04 May 2021]
- [32] Okos, Dora. (2019) *Aerial Photo of City Buildings*. Available at: <<https://www.pexels.com/photo/aerial-photo-of-city-buildings-3379625/>> [Accessed: 04 May 2021]
- [33] Pixabay. (2016) *Gray Concrete Building*. Available at: <<https://www.pexels.com/photo/gray-concrete-building-48896/>> [Accessed: 04 May 2021]
- [34] Buchi, David. (2018) *New York City*. Available at: <<https://www.pexels.com/photo/new-york-city-1038531/>> [Accessed: 04 May 2021]

Appendix A – Screenshots of Procedural Buildings

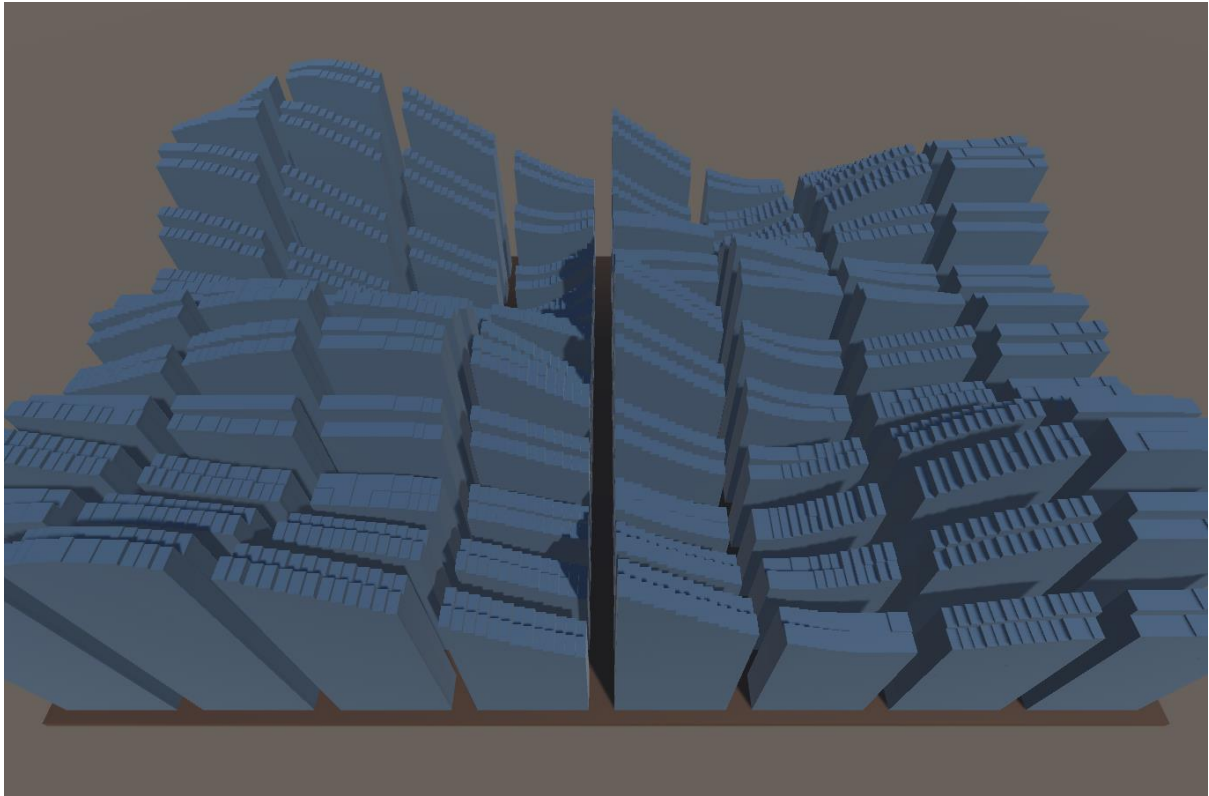
0.5 Perlin Noise (3 – 53 meters)



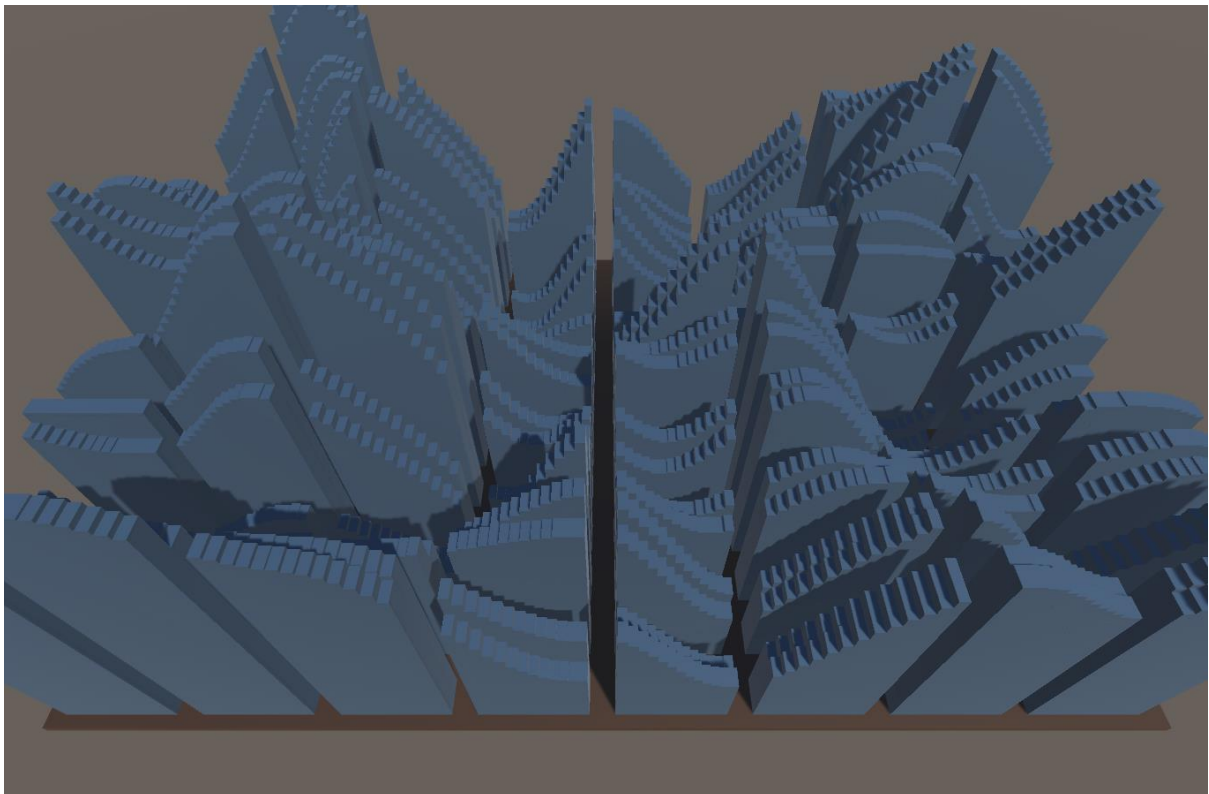
1.0 Perlin Noise (3 – 53 meters)



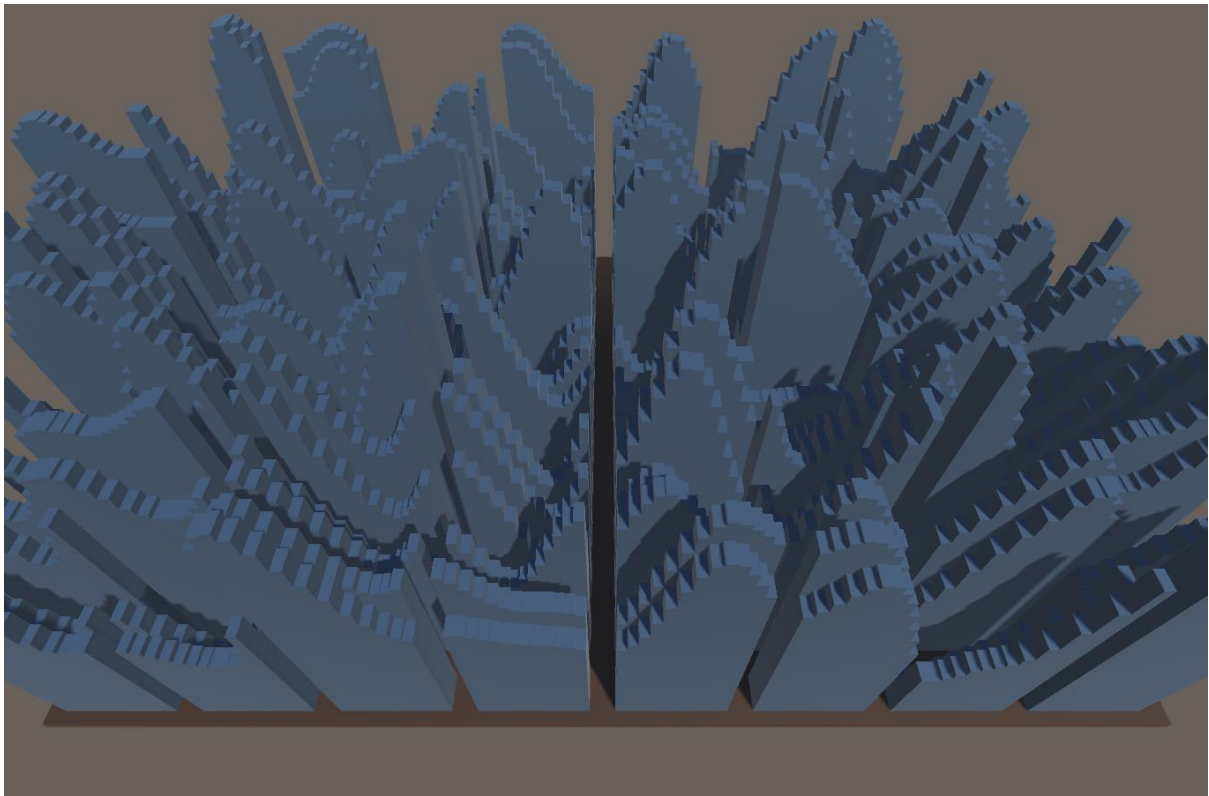
2.5 Perlin Noise (3 – 53 meters)



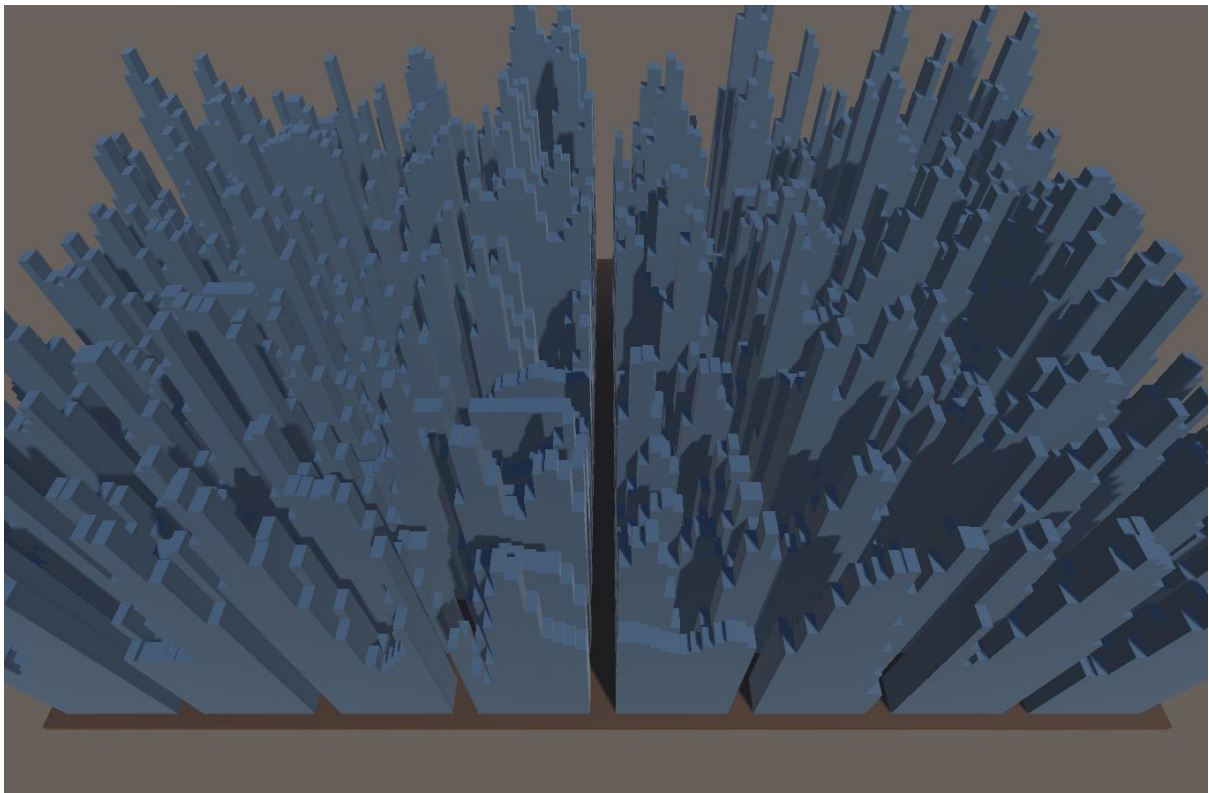
5.0 Perlin Noise (3 – 53 meters)



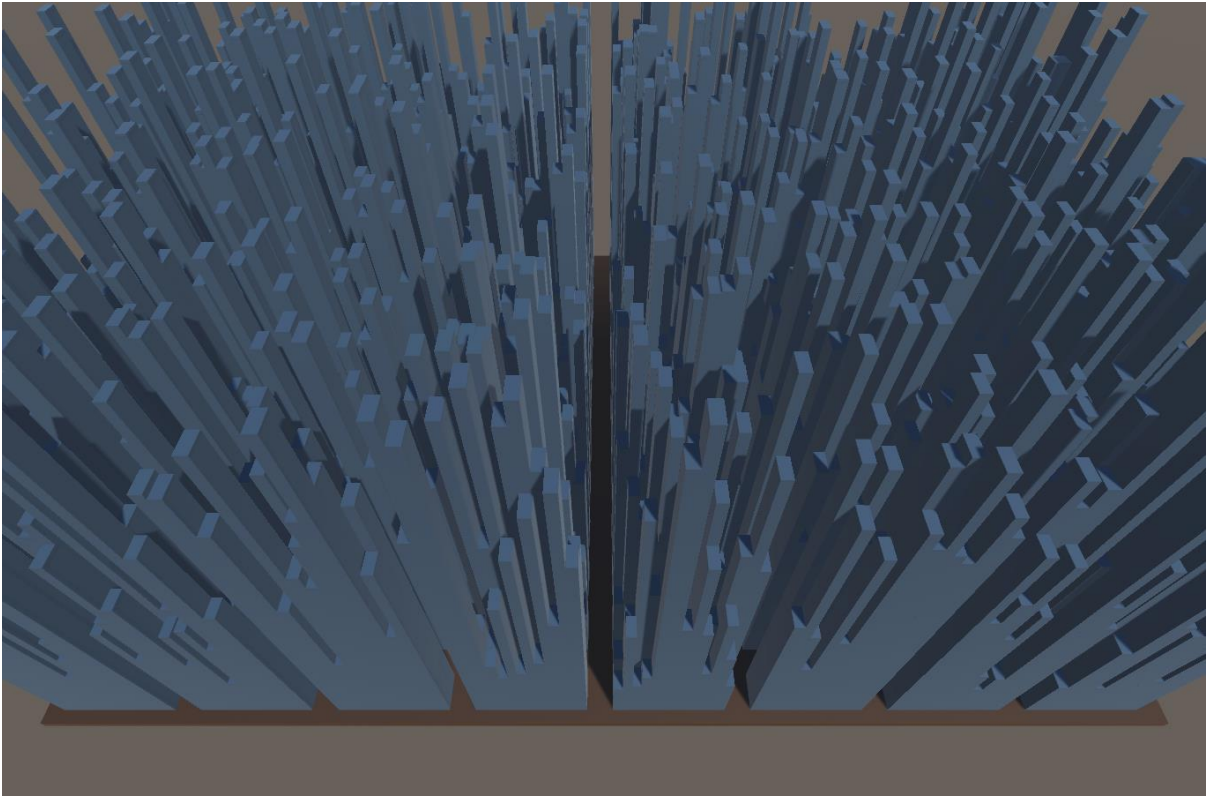
10.0 Perlin Noise (3 – 53 meters)



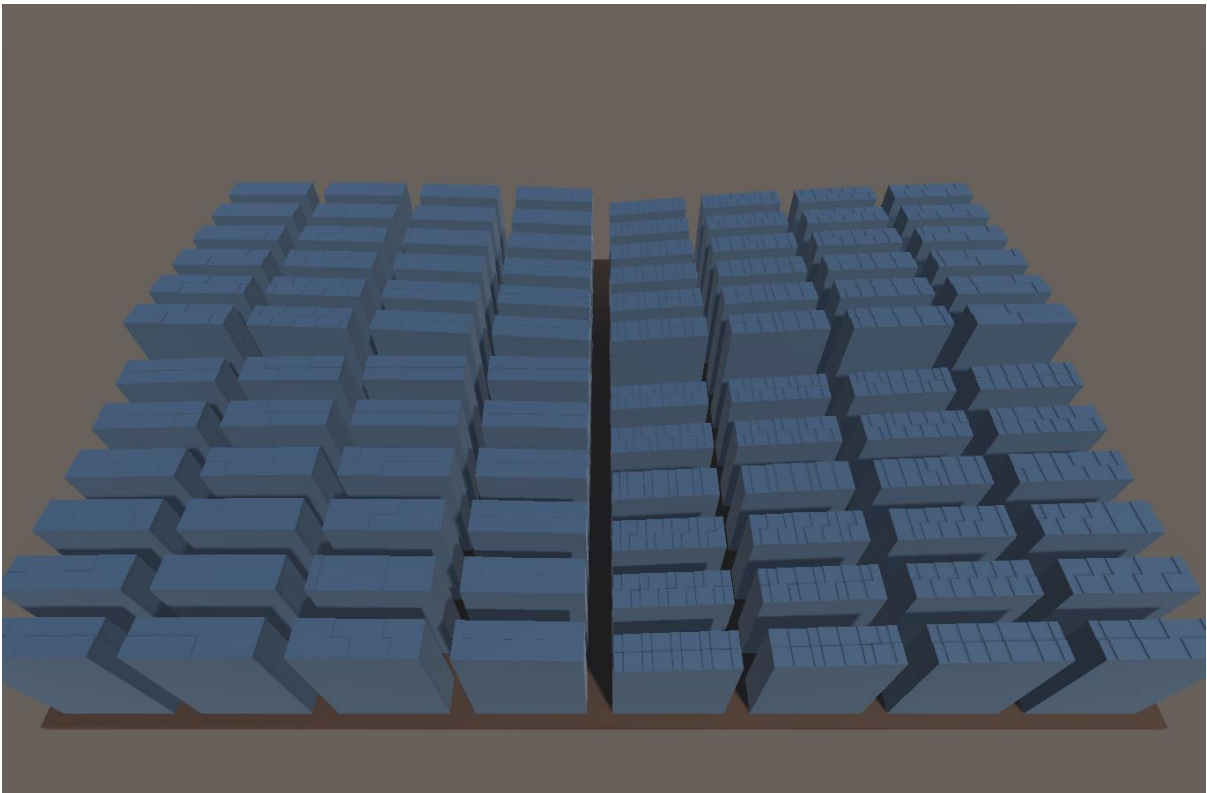
25.0 Perlin Noise (3 – 53 meters)



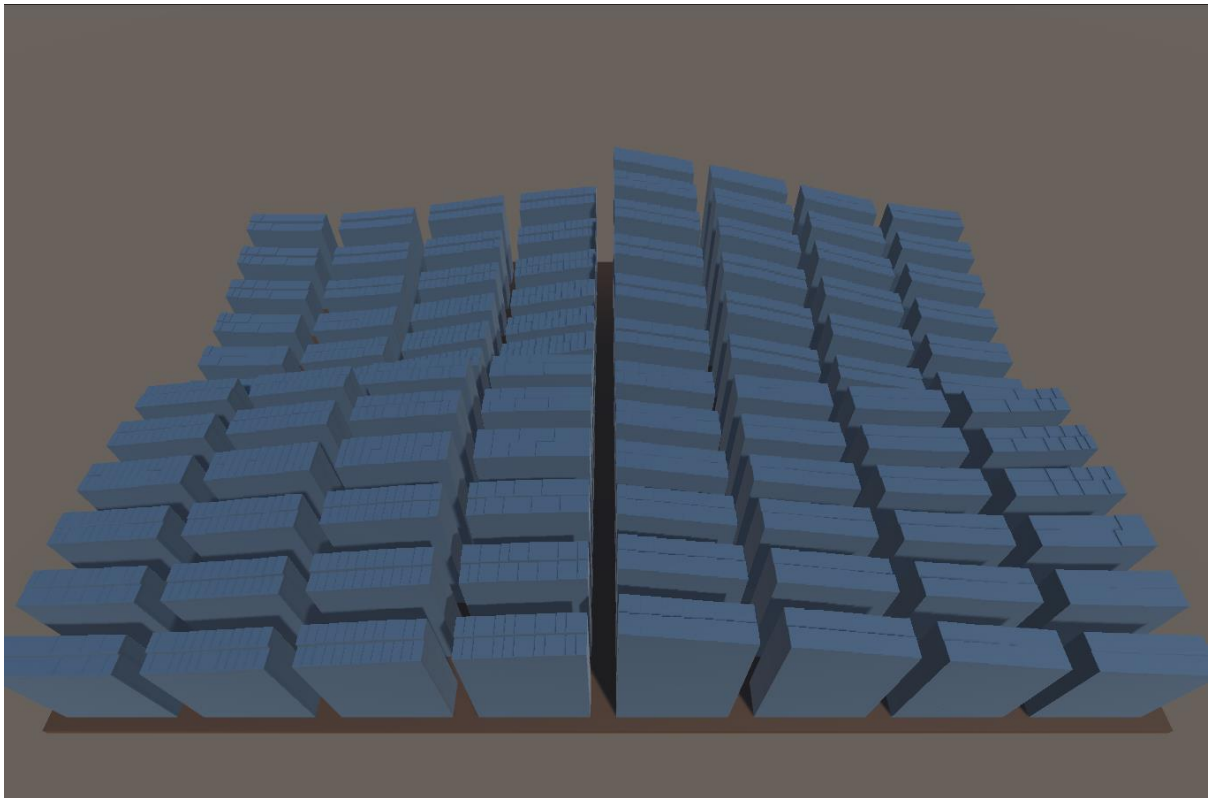
Random Heights (3 – 53 meters)



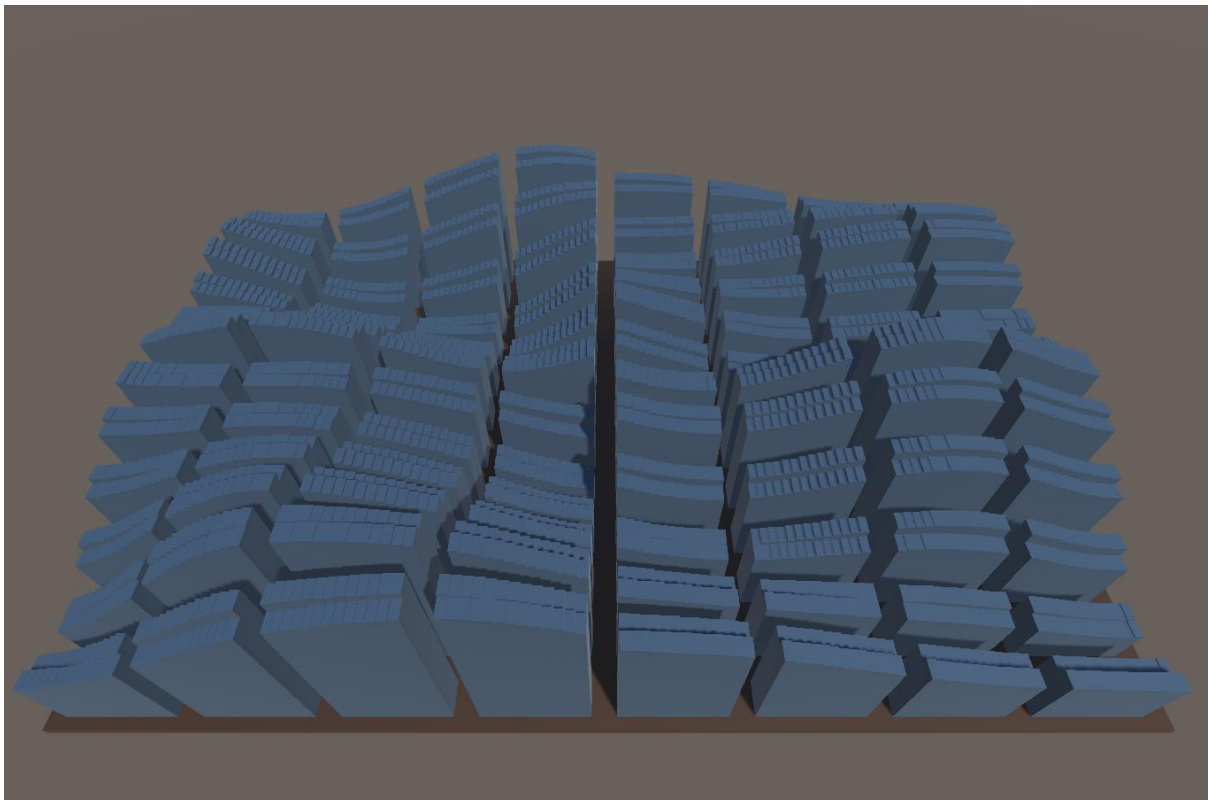
0.5 Perlin Noise (3 – 35 meters)



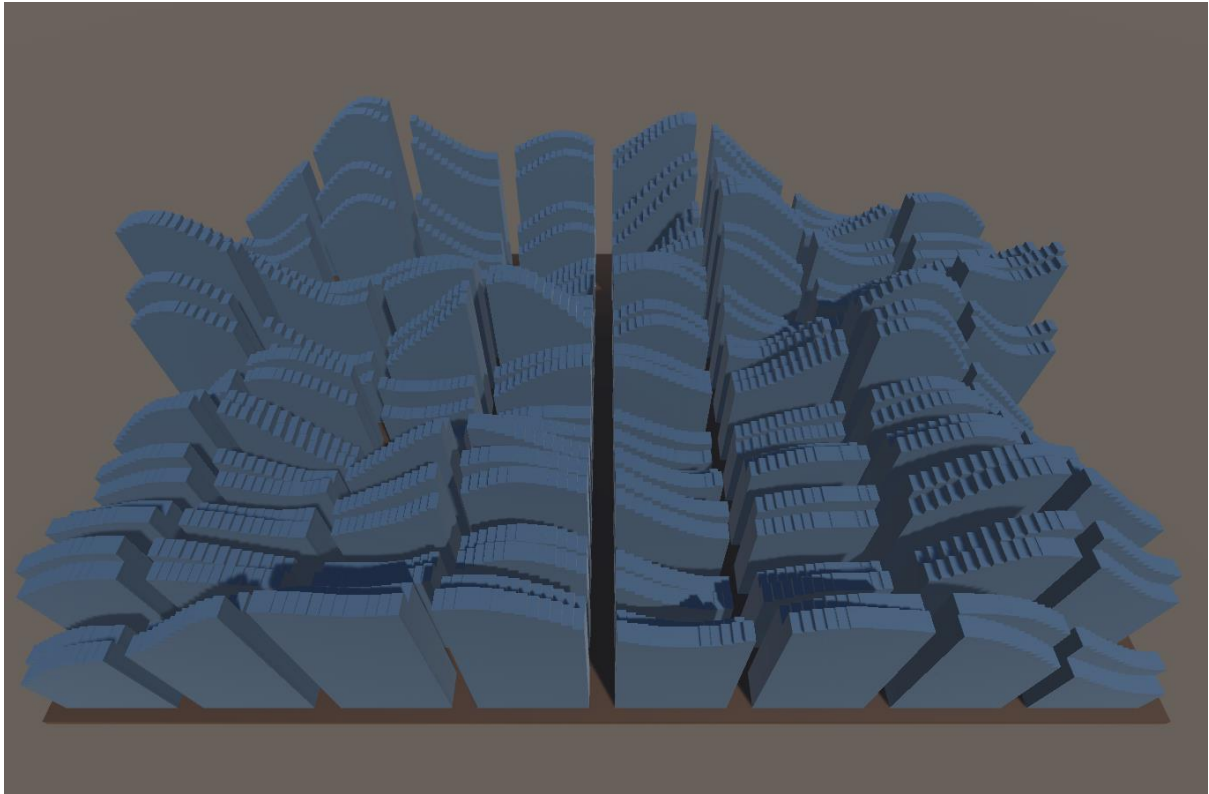
1.0 Perlin Noise (3 – 35 meters)



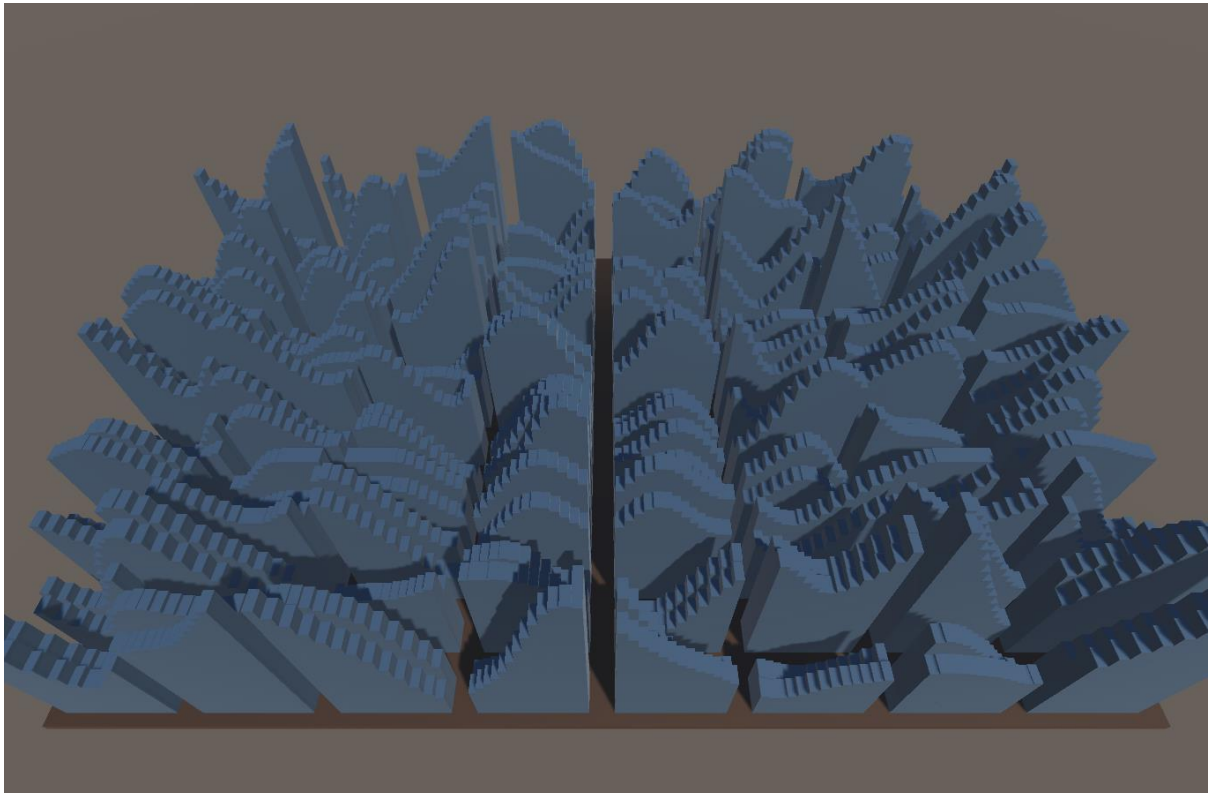
2.5 Perlin Noise (3 – 35 meters)



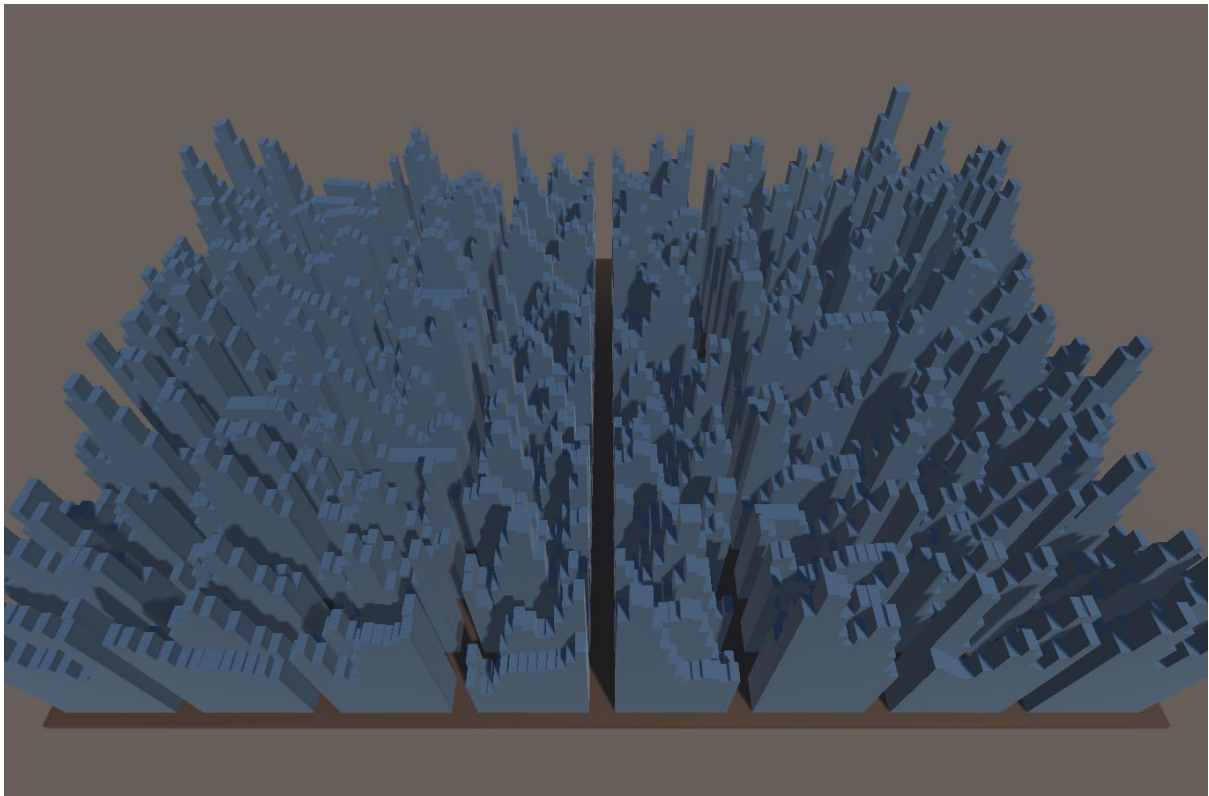
5.0 Perlin Noise (3 – 35 meters)



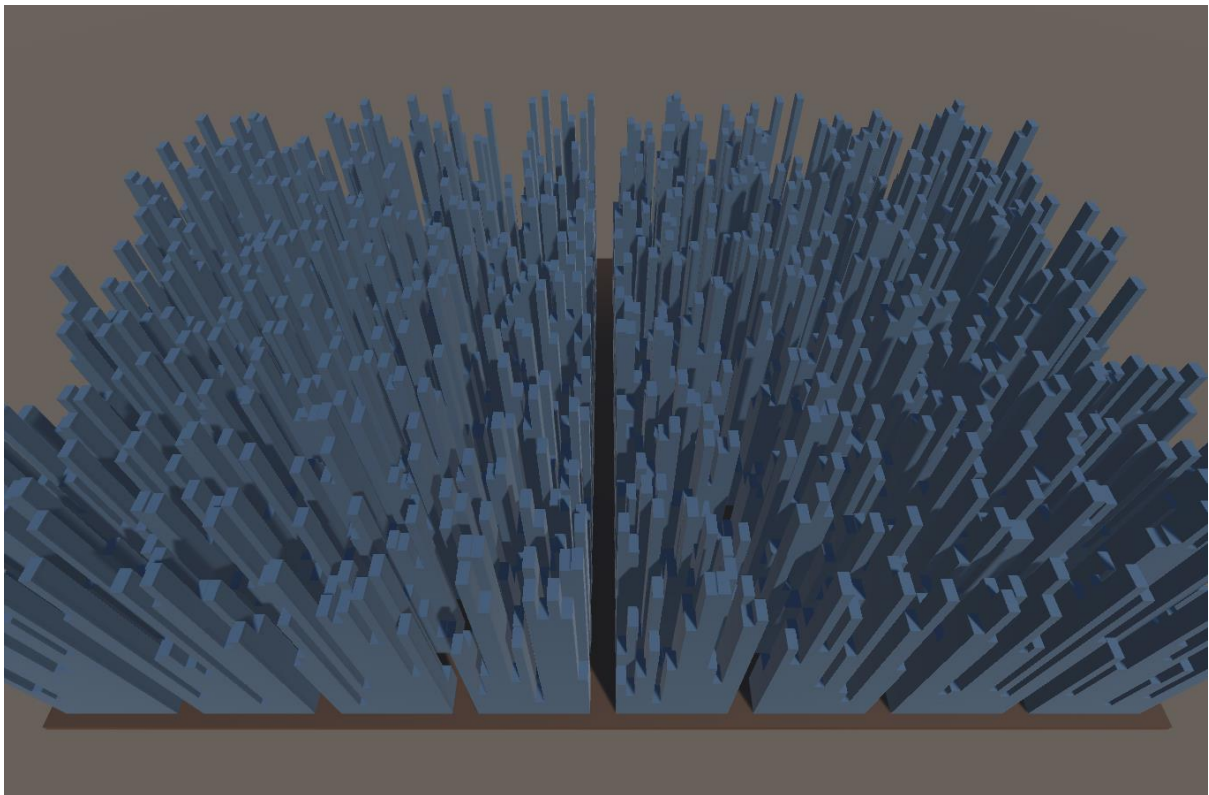
10.0 Perlin Noise (3 – 35 meters)



25.0 Perlin Noise (3 – 35 meters)



Random Heights (3 – 35 meters)



Appendix B – Source Code

The final source code for the dissertation can be found with the supplementary materials inside a folder titled “*Appendix B – Source Code*”. In addition to being uploaded as supplementary material, the source code is also uploaded to a *GitHub* repository.

The main code can be found by going into the Assets folder then into Scripts.

Materials for game objects can be found within the Materials folder inside Assets.

Prefabs can be found within the Prefabs folder inside Assets.

Variables can be altered within the Unity interface by clicking on the various Game Objects within the Hierarchy window. Additional variables can be altered by going into the Prefabs folder within the Unity interface and selecting the Prefabs. All the variables to be edited are visible within the Components inside the Inspector window.

Appendix B – Source Code/

```
|---Assets
  |---Materials
    |---Building.mat
    |---Grid.mat
  |---Prefabs
    |---Block.prefab
    |---Building.prefab
    |---BuildingPiece.prefab
  |---Scenes
  |---Scripts
    |---PerlinNoise.cs
    |---ProceduralBlocks.cs
    |---ProceduralBuildings.cs
    |---ProceduralGrid.cs
    |---SwitchCamera.cs
```

The link for the GitHub depository can be found here:

<https://github.com/malton108/MA-Dissertation>

Appendix C – Test Results

The results from testing the program can be found uploaded with the supplementary materials inside a folder titled “*Appendix C – Test Results*”. In addition to being uploaded as supplementary material, the source code is also uploaded to a *GitHub* repository.

The file is called “*TestResults.xlsx*” and contains the following spreadsheets:

Comparisons

Includes information about data provided by NYCOpenData for the heights of Manhattan buildings. Also contains the first testing of building height ranges between 3 and 53 meters, and 3 and 35 meters.

3-53 Test Results

Includes data for all Perlin noise levels tested for building heights between 3 and 53 meters. Each value was tested 10 times. A mean, median and variance of each level is also displayed.

3-35 Test Results

Includes data for all Perlin noise levels tested for building heights between 3 and 35 meters. Each value was tested 10 times. A mean, median and variance of each level is also displayed.

The link for the GitHub depository can be found here:

<https://github.com/malton108/MA-Dissertation>