Procedural City Generation

Tom Reilly

Student Number: 170239882

May 2020

Computer Science

Supervisor: Dr Gary Ushaw

Word Count: 14,286

Abstract

This dissertation puts forward the case of using Procedural Generation over the use of artists when it comes to creating a city. One of the most time and resource consuming aspects for any game development is the creation and layout of complex structures. This dissertation looks at several Procedural Generation techniques along with a summary of related works on this subject. The project presents and evaluates the results of when three of these are implemented into a system. These are Perlin Noise, L-Systems and the author's own procedural generation algorithm.

The motivation for this project originally stemmed from a love of playing open world and city builder games such as GTA V and City Skylines. The ultimate intention is to comprehend how one might create a city generator where the user can design their own bespoke city, using a set of inputs, boundary conditions and rules, which could be exported into their own project and assist in the development process.

The dissertation also makes an assessment of whether increases in user input can lead to a more creative and visually appealing outcome.

Declaration

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

Acknowledgements

I would first like to thank my supervisor Dr Gary Ushaw for the advice and support provided throughout this project. I would also like to thank Dr Rich Davison and Dr Giacomo Bergami for their additional support during this process.

Table of Contents

List of Figures

Chapter 1 – Introduction

1.1 – Context and Motivation

Modern 3d games are often established within large open worlds with many of these taking place in a single expansive urban environment. This tends to require a large amount of time-consuming content creation. In meeting demands for realistic and detailed content, studios have tended to go through resource challenges leading to highly damaging product launch issues, brand damage and share price challenges. Increasing the number of artists deployed in the design process is not always generating a proportional amount of quality content showing that the production pipeline is not linearly scalable. Extending deadlines is also not a long-term solution as consumers and shareholders tend to become disillusioned losing faith in the studio and its ability in game development.

Procedural Generation is a concept gaining widespread appeal. Instead of having multiple artists manually creating these urban areas, a programmer can produce a set of rules to generate the content. Modern games have also increased in detail and complexity with the amount of space needed for storage on the hard drive also increasing. With Procedural Generation, worlds can be infinite with only the terrain close to any player being generated.

Currently, there are many ways to implement Procedural Generation. Notable examples have been No Man's Sky and Minecraft. The former cultivated a lot of public interest with its claim of being able to generate eighteen quintillion unique worlds. Upon release however, it was met with criticism about the lack of variety contained within said "unique" worlds. This leads me to believe that Procedural Generation could be managed better in a smaller and more urban environment. Cities while still being varied and complex would be perfect for Procedural Generation. Open world games such as GTA V or Watchdogs Legion both contain detailed and handcrafted cities, which consume a lot of time. If Procedural Generation was included in the process, it could have reduced development time and reduced costs. The resources saved in reducing the effort required for the basic cityscape could be redeployed in improving gameplay or creating more compelling experiential features, likely improving the profitability of the product.

Currently fractals, L-Systems, Perlin noise and tiling systems are all techniques employed in city generation. One of the fundamental issues with most procedural city generators is the realistic or "natural" look of the finished city. In this project I will be investigating the efficiency and other characteristics such as loading time and scalability of different procedural generation techniques. The chosen techniques will be Perlin Noise, L-Systems, and my own algorithm.

1.2 – High Level Aim

To develop a natural looking Procedural City Generator (PCG) on the Unity Engine and evaluate the efficiencies of the different applied techniques.

1.3 – Objectives

1. To investigate how Perlin noise can be used for zoning in city generation

   Through research I have found that Perlin noise has been widely used in the generation of terrain for more rural areas. By applying the same concepts to that of an urban area it will allow me to generate different heights of buildings in a city through sampling the greyscale value at corresponding world and texture coordinates.

2. To find out how L-Systems can be used to generate complex road networks.

After conducting further research into procedurally generated content, I have found that L-Systems and its methodology of rewriting are suitable for the generation of cities. This is due to their succinct nature and computational efficiency.

3. To design and develop three distinct Procedural City Generators.

This objective will be used to put the theory learned from objectives 1 and 2 into practice. This objective will be the visual representation of the project and will be the key factor in deciding which techniques are viable for Procedural City Generation.

4. To investigate the efficiencies of different procedural generation techniques.

This last objective will be used to assess whether procedural generation techniques are indeed useful when it comes to creating cities. By using the six criteria, which will be explained below, it will provide an assessment framework of each generation method and provide an insight into the validity of the concept of using procedural generation for cities.

1.4 – System Requirements

To achieve the objectives stated above, the following requirements will be needed. These requirements will be split into four separate groups.

Building and Grid Generation

1. The system should have three distinct types of buildings.
2. The system should be able to generate different heights of the three types of buildings for variation and give a city a natural looking feel.
3. The systems should ensure that buildings do not overlap or clip into each other.
4. The system should be able to generate roads.
5. The system should be scalable.

Random Generation

6. The system should be able to randomly generate a unique looking city.

Perlin Noise

7. The system must be able to create a Perlin noise texture based on user input.
8. The system must be able to sample the created textures.
9. The system should be able to generate a city based on sampled data.

L-System

10. The system should allow the user to be able to create new rules and sentences to be input back into the system.
11. The system is responsible for the encoding of the sentences and ensure that symbols have been given their legitimate corresponding function.
12. The system must be able to construct a road network established from the user's initial sentence and rules.

1.5 – The Criteria

Does it look Realistic/ Valid?

If the generated city does not look entirely realistic then it must at least be credible for a player to enjoy the environment if it was ever exported into a playable demo/ product. Players should be consistently immersed within the city and broken geometry will not help this. The city structure should not look completely random or 'thrown together'.

Is the city scalable in size i.e., Flexible?

The generator should be able to generate multiple different cities with different looks.

What is the loading time?

The city should be generated within a satisfactory amount of time. Moving around the city with the built in Unity camera should also be above 60 fps.

How much user input is needed?

What is the minimal amount of input data required to generate the city and what is the optimal data set for each technique?

How economical is the overall process?

Does the implementation time of the Procedural Generator outweigh the visual result?

Is the generation deterministic?

If the same seed is put through the generator is the same city produced? This is important as the user has definitive control over the output and might be wanting to achieve a specific look/ design for their city.

1.6 – Summary of Results

After implementing the different techniques, I have found that each offers the user something different. The first technique offers a system where the user has very little parametric control, the second, where the user has more parametric control and the third where the user has almost complete control over the entire city layout at their own discretion. The L-System which offers the most user input can create (with the right set of rules) European style road networks, which allows for more variety than the standard Manhattan grid used for the first two implementation. The second implementation offers a Perlin Nosie approach to city generation. This approach creates more natural looking building layouts for a city.

1.7 – Dissertation Outline

Introduction

The introduction for this dissertation will provide an insight into why I have chosen this project, the problem I will be addressing, as well the project's aim and objectives.

Technical Background

In this section of the dissertation, I will be discussing the concepts and theories that I have been implementing in my Procedural City Generator. I will also be examining other related works to this field and evaluating them to my set criteria.

What was done and how?

This will be a detailed account of the methods and technologies I employed to achieve my aim for this project. This section will also focus on the diagrams and pseudocode to explain how the implemented algorithms work.

Results and Evaluation

This section will focus on the outputs produced from the different Procedural Generation techniques that were used in this project. This section will also be where the six criteria will be deployed to assess and evaluate the efficiency of each of the three techniques.

Conclusion

In the concluding section of this dissertation, I will be discussing the extent to which the original aim and objectives have been met. I will also be considering what went well, what could have been done to improve the project and what could be done in the future in terms of increasing the scope and the features of this project.

Chapter 2 – Technical background

2.1 – Procedurally Generated Content

Procedurally Generated Content, PGC, in games has been well-defined as *"the algorithmic creation of game content"* (Togelius, et al., 2011). Essentially it refers to a piece of software that can create playable game content on its own or in unison with human designers. Nevertheless, PGC can be used for a multitude of facets when it comes to game content creation. One such example is in the 2D side scroller game *Spelunky* that utilises PGC to generate variants of its levels (Spelunky, 2008). The main property of PGC is that it defines geometry, textures, and effects in terms of a *"sequence of generation instructions rather than as a static block of data"* (Kelly & McCabe, 2006). These instructions can be called upon with user specific descriptions when needed in order to instantiate assets of the same type with varying characteristics. In *Texturing and Modelling: A Procedural Approach* (Ebert, 2014), David S Ebert identified three key features a procedural technique must have. These are *Abstraction*, *Flexibility* and *Parametric Control*.

> *Abstraction* in relation to PGC means that geometric and texture details are abstracted into a set of procedures. This set (or algorithm) is then handled by the operator which can call on them when needed.
>
> *Flexibility* is the second key feature. This feature means that an appropriate PGC system must be able to produce a wide range of results from parameters which are not always restricted to the controls of the original model.
>
> *Parametric Control* is when the developer can define and adjust specific behaviours to be used as useful controls for artists. An example of this would be being able to change the noise scale in a noise generator algorithm.

Procedurally Generated Content allows for simple parameters and brief descriptions of instantiated objects to be used to create detailed textures and geometric structures. This effect is what Ebert called *data amplification*[4]. Procedural Generation is an area that seems easier on the surface than it is. The constant weighing of order against chaos so that scenes appear credible but not too predictable makes every decision (in design and implementation) a careful planning act.

2.2 – City Generation

A city can be defined as *"an inhabited place of greater size, population, or importance than a town or village"*. Cities have often been divided into zones. The Burgess/ Concentric Model states that there are six main zones (Park & Burgess, 1925). Starting from the outside to the middle, Commuter, Residential, Working Class, Transition, Factories and then the Central Business District (CBD).

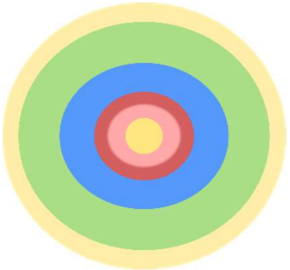


Figure 1: The Burgess Concentric Model (Burgess)

For this dissertation I will be merging these zones to form three new usage groups. These will be Residential, Industrial and Commercial. From making these fewer generalisations it emerges a useful

mechanism for modelling the functions of a city. Unlike the real world, cities in games do not need to be completely logical. Electricity lines or petrol stations do not necessarily have to be added. These simplifications allow us to create a significantly reduced model as cultural and architectural influences on the city are not considered, this leads to far less complexity.

My goal for this project is to create an accurate city generator. Therefore, when it comes to the project, I have found three distinct generation stages. These will be roads, districts and buildings.

2.3 – Online vs Offline

When developing a system, using procedural content generation, it can be either be online or offline (Shaker, et al., 2016).

Offline PGC is generated before the player starts playing the game as generating the content would be too slow. This means that the generation would be undertaken by the developers before release. The type of content to be generated would usually be maps and environments as they are the most complex and resource consuming.

Online however is when content generation is happening as the player is playing the game. This opens the possibility of generating player-adapted content. For example, in Left for Dead 2 the game employs an AI Director. This director controls the generation of new items and enemies in the game based on physical and emotional state of the player's character (Booth, 2009).

2.4 – Perlin Noise

To be able to describe and explain what Perlin Noise is, we must first ask what is noise? Lagae stated in the paper *State of the Art in Procedural Noise Functions* that noise is *"the random number generator of computer graphics"* (Lagae, et al., 2010). Perlin Noise is a type of Gradient Noise. Gradient Noise contains lattices of pseudo-random gradients within a coordinate system. Dot products of these gradient points are then interpolated to obtain a noise value between the lattices. The obtained value will be between 0.0 and 1.0.

One of the first implementation of the gradient noise function was Perlin Noise. Ken Perlin developed it in 1982 for the film *Tron* and won an Academy Award for Technical Achievement in 1996 for creating the algorithm. The noise does not contain a completely random value at each point but rather consists of "waves" whose values gradually increase and decrease across the pattern (Unity, 2022). Perlin Noise can be implemented from 2D to 4D (Perlin, 1985). In this project we will be considering 2D Perlin Noise as well as looking at Ken Perlin's updated implementation from his paper *Improving Noise* (Perlin, 2002).

How it works:

The algorithm expects two arguments, an x and a y coordinate. These two values are normalised to create a point (where an individual pixel lives) between two integer lattices. A gradient vector (with a random direction) from each lattice point to this pixel point is calculated. They are called gradients because the noise function will increase in the direction of each gradient vector.

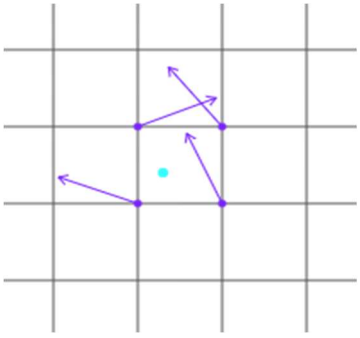


Figure 2.1: Random Gradient Vectors Assigned

For each neighbouring lattice point we take the dot product of their gradients with the difference vector of the normalised point and each neighbouring grid point.

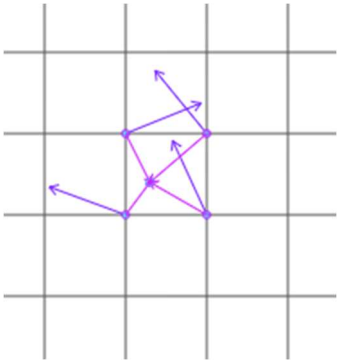


Figure 2.2: Calculate Dot product

The resultant noise is the value calculated by interpolating these dot products from each Pixel.

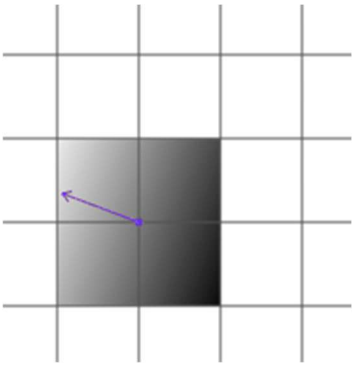


Figure 2.3: Interpolate Dot products

The last stage is to apply the fade function to the x and y axis, which acts a drop off filter to allow smoother transitions between gradients. **Fade(t) = $6t^5-15t^4+10t^3$**

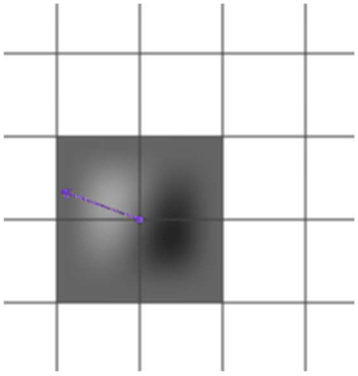


Figure 2.4: Introduce Fade function

2.5 – Voronoi Diagrams

Voronoi diagrams were defined by Imperial Russian mathematician Georgy Vorony. It wasn't until Worley's paper *"A Cellular Texture Basis Function"* (Worley, 1996) in 1996 that these types of diagrams were proposed as a method for Procedurally Generated Content. Worley demonstrated an algorithm that partitioned space into a pseudorandom array of cells that generated cellular type textures.

Figure 3: Coloured cell Voronoi Diagram made from 20 points (Ertl, 2015)

The technique was initially developed to be used in generating textures with a cellular nature to them like skin or leaves. It was also to be used in conjunction with other PGC techniques such as Perlin Noise.

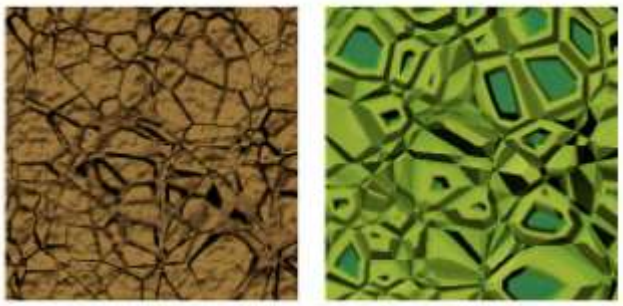


Figure 4: Cellular textures created using Worley's algorithm

The algorithm works by partitioning an area into cells using lines which are plotted using initial data points on the map. Each line is placed an equal distance between every point and its neighbours. This results in the final diagram.

The algorithm also achieves Ebert's rules for *Abstraction* and *Parametric Control* (Ebert, 2014). It provides these by offering a small set of parameters called "Worley Constants" that control the operations it performs while also delivering large variations on the output.

2.6 – Fractals

In 1975 mathematician Benoit Mandelbrot coined the term Fractal. Fractals hold a property called self-similarity. This is where as Mandlebrot said, *"a rough or fragmented geometric shape that can be split into parts, each of which is a reduced-size copy of the whole"* (Mandlebrot, 1983). An example of this is the Koch Snowflake as seen below.

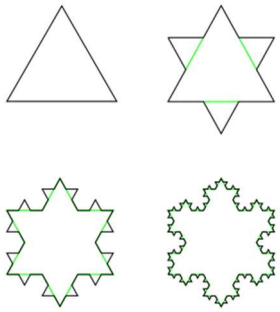


Figure 5: Koch Snowflake (Wikimedia Commons, 2007)

The above example shows four recursions taking place and this is known as scale invariance and can theoretically continue infinitely. Fractals however are limited to self-similar structures and providing an effective abstraction for natural objects (Ebert, 2014).

2.7 – L-Systems

A more flexible alternative to a Fractal algorithm is a Lindenmayer System, or more commonly known as the L-System. Originally developed by biologist Astrid Lindenmayer for the use of studying bacteria replication and Algae growth (Lindenmayer, 1968) they have been used in fractal geometry generating plants within the field of computer graphics.

L-Systems are a class of formal grammar whose defining feature is parallel rewriting (Burgess). Rewriting is the technique for defining complex objects by replacing parts of a simple initial object, using a set of rewriting rules, resulting in a much longer consecutive sentence. The Thue-Morse Sequence shows the L-System and its components in action (Ardnt, 2011) :

V (variables) – this is the alphabet containing a set of elements that can be replaced.

S (Constants) – this contains a set of symbols that cannot be replaced.

ω (axiom i.e., start) – this is a string of variables and constants that represent the initial state of the system
P (Rules) – this is a set of productions that defines the way variables can be replaced with combinations of constants and variables.

Consider a system **K,** which is defined by the four components above:
K = {V, S, ω, P}
V = {0,1}
ω = 0
P_1 = 0 –> 01
P_2 = 1 –> 10

For,
n = 0: 0
n = 1: 01
n = 2: 0110
n = 3: 01101001
n = 4: 0110100110010110

Here we see that with every iteration not only the resulting sentence grows but also the rate at which it grows. This shows us that L-Systems could be used for defining more complex objects.

Visualisation of the L-System

One way to visualise L-Systems is to use the generated string as instructions for an agent to move across a surface and trace its path. The commands given to this agent will be to move forwards and turn left or right (at a predetermined angle).

F: move and draw forwards

+: turn left at **θ** degrees

–: turn left at **θ** degrees

Using these new symbol definitions another set of productions can be prepared.

Consider a system **K**, which is defined by only one rule with the following axiom and angle:

F –> F + F – F – F + F

ω = F

θ = 90°

For,

n = 0: F

n = 1: F + F – F – F + F

n = 2: F + F – F – F + F + F + F – F – F + F – F + F – F – F + F – F + F – F – F + F + F + F – F – F + F

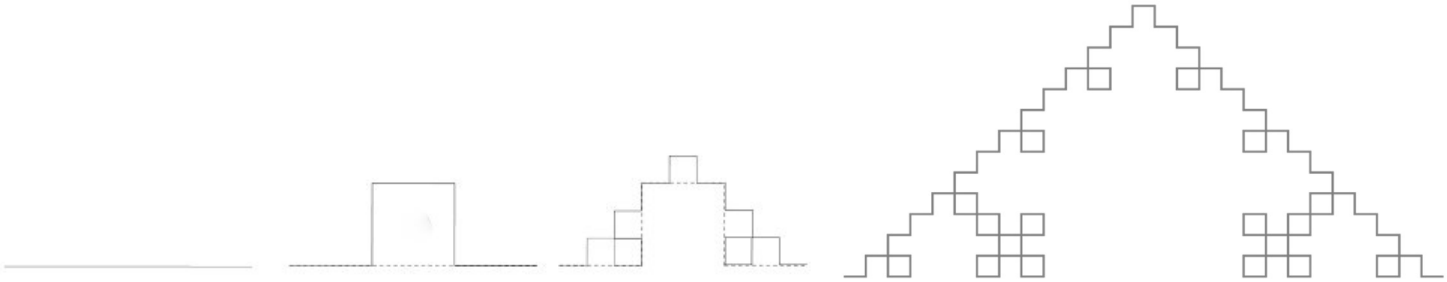


Figure 6: Square Koch Fractal Curve after 0, 1, 2 and 3 iterations. (Anon., 2007) (Academic, 2002)

By interpreting these strings as the instructions stated earlier, we begin to see how a sequence of symbols can be realised as a complex fractal structure.

Nevertheless, this method does come with its own constraints. Having a system designed this way means that the whole structure must be drawn without the agent's "pen" leaving the paper. Lindenmayer gives us a solution to this, the Bracketed L-System (Lindenmayer, 1968). This type of L-

System introduces two more symbols, [and], each with their own behaviours. The [will save the current location of the agent to a stack (i.e., push) and the] will reload the last saved position (i.e., pop). This method allows for the generation of natural looking objects such as plants.

Lindenmayer gives an example system:

F –> F [+ F] F [– F] [F]

ω = F

θ = 20°

Figure 7: Branching Structure after five iterations (Przemyslaw & Lindenmayer, 1990)

Here we see that with the bracketed extensions the L-System can be used to create natural complex flora. Using L-Systems allows for complex models to be defined and visualised as their concise set of rules can be easily adjusted, which makes them a good example for Procedural Generation.

2.8 – Tile-Based Generation

One of the most straightforward procedural generation techniques applied to game development is Tiling. Tile-based generation techniques are generally used in multi-texturing. This allows for highly detailed textures made from multiple layers of base textures. The most common example for tile-based generation is terrain texturing (Lefebvre & Neyret, 2003). Rock, snow, sand, and grass texture layers can be merged with differing levels of influence on the resultant texture. These final textures can be applied to a terrain according to height, gradient, or an image map. This satisfies Ebert's *Parametric Control* as these three parameters are specified by the artist. One high resolution texture would not be able to cover a large map, however this method means that large terrain maps can have detailed textures placed onto them.

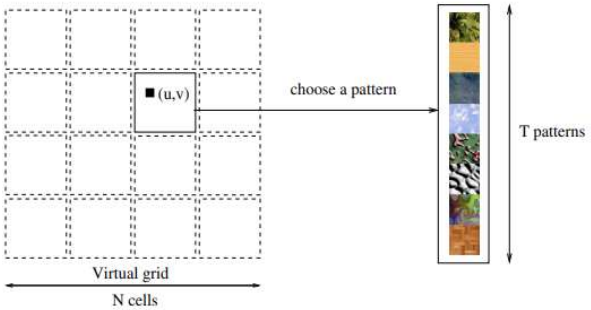


Figure 8: Pattern based procedural textures: A pattern is chosen on the fly (Lefebvre & Neyret, 2003)

It is worth noting that by adding a probability distribution map, which is an image map that stores the probability of using several texture tiles, one could create a vast number of unique worlds. If a particular world was appealing to the user, then they could write down the seed used to create that world and then be able to recall it whenever they wanted to.

This method of being able to input a seed for a desired world is a good example of why procedural generation is useful. This is because the storage and memory requirements are sizably reduced making it possible to efficiently store and render large worlds in real-time.

2.9 – Related Works

In this section I will be discussing previous works that have been accomplished on the topic of Procedural Generation of Cities.

CityEngine

L-Systems have traditionally been used to generate natural objects such as plants and trees. Nevertheless, the 2001 paper by Parish and Muller (Parish & Muller, 2001) is perhaps the first attempt published to procedurally generate realistic cities for the use of computer graphics. Parish and Muller use a hierarchical L-system for both road and building generation. The system generates both major and minor roads that follow one of the predefined road styles over a given population density and road pattern map.

CityEngine used an extended version of L-Systems used in the road generation, which are called *"Self-Sensitive L-Systems"*. This means that the system takes the current road 'growth' into account. The system takes input in the form of geographical information. This is information on water, vegetation, elevation and population density.

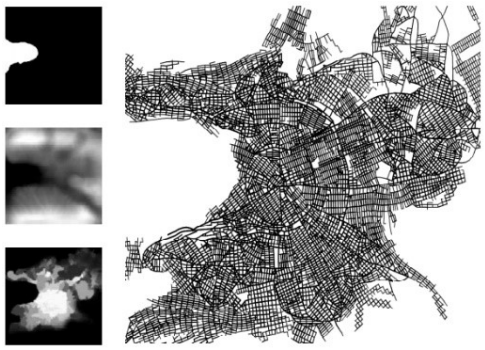


Figure 9: Left column (from the top down) showing water, elevation and population density input image maps. Right side picture showing resultant roadmap generated. (Parish & Muller, 2001)

The generation of the road network above is completed by using two functions, Global Goals and Local Constraints. The Global Goals initially propose a set of parameters for a road which are aligned with the goals set by the Road Pattern. The "*localConstraints"* function is then called to adjust the proposed parameter values so that the road can 'fit in' with local environment.

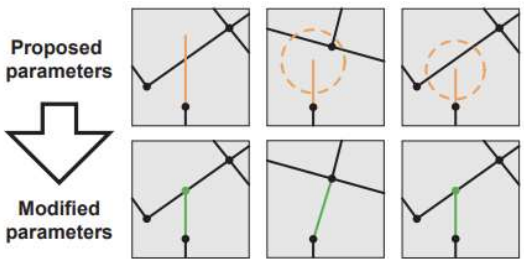


Figure 10: Example of local constraints adjusting a global goal's proposed parameters. (Parish & Muller, 2001)

Parish and Muller use a second L-System to generate buildings, thier geometry and the textures placed onto them. Before the buildings can be generated allotments for the buildings need to be calculated from the newly formed city blocks. Where the roads are the blocks borders.

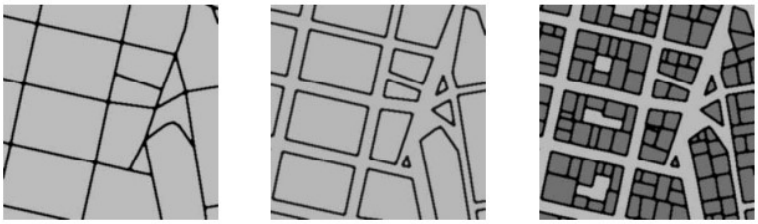


Figure 11: Left side: roadmap. Middle: Blocks created from road network. Right side: generated lots with cull if allotment has no street access. (Parish & Muller, 2001)

The buildings are the generated using a parametric stochastic L-System. For every valid allotment there is one building. The type of building to be generated is determined through zoning rules, which are controlled by an image map. Each building's initial state (axiom) is a bounding box, with the output of each iteration "*can be interpreted as a refining step of the geometry"*.

Figure 12: Five iterations to generate Empire State Building. Bounding box axiom allow for easy Level-Of-Detail (LOD) generation. (Parish & Muller, 2001)

Figure 13: Virtual Manhattan generated by CityEngine. 13,000 buildings were generated in under 10 minutes. (Parish & Muller, 2001)

CityGen

During "*The Fifth Annual International Conference in Computer Game Design and Technology*" in Liverpool, George Kelly and Hugh McCabe presented their own system for Procedural City Generation called Citygen (Kelly & McCabe, 2007). Unlike CityEngine, CityGen was designed for use in games and other graphical applications and not just for the modelling of cities. This new system was to give the user more control over the generation process. In addition, the system also includes a real-time feedback feature, so the user gets to see an immediate change to their generated city when they manipulate the parameters.

The city is generated in three stages: primary road generation, secondary road generation and building generation. The primary road network is implemented as a *"connected graph"* and once a region of terrain is enclosed by a road network this is called a City Cell. To extract the cells a Minimal Cycle Basis algorithm is executed on the enclosing primary road network, putting the individual cell data into a self-contained unit. This allows the cells to be independent of another, the data shared between them in minimal and enables efficient parallel execution of secondary road generation.

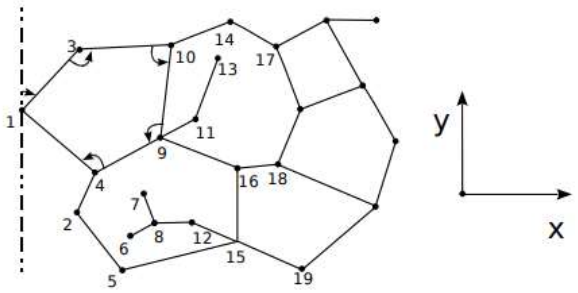


Figure 14: Illustration of City Cells and MCB algorithm execution. (Kelly & McCabe, 2007)

The secondary road network process is automatically started after a City Cell is created. The secondary road network is generated using growth-based algorithm inspired by L-Systems. The control parameters of this generation are segment size, degree, snap size, and connectivity. The snap size alters the maximum radius of the end of a road to connect to existing infrastructure and the connectivity controls the probability that segments will connect. This would affect the road network flow. These controls form a cell seed that ensures all generated variants are *"100% reproducible"*.

The enclosed regions of both primary and secondary road networks are then subjected to the Lot Subdivision algorithm. This process subdivides each region into two or more allotments. The algorithm runs recursively until a target lot size has been achieved. Allotments that do not have any road access are excluded from the process.

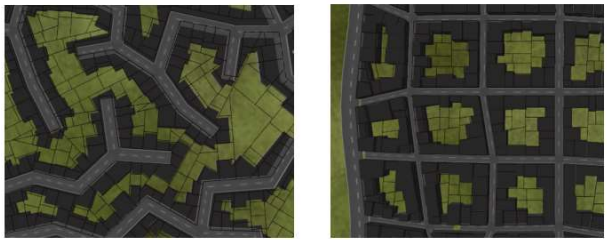


Figure 15.1: Lot division done on suburbs and city districts. (Kelly & McCabe, 2007)

After the Lot Subdivision the buildings are generated in their neighbourhoods in accordance with user input from the initial control parameter set.

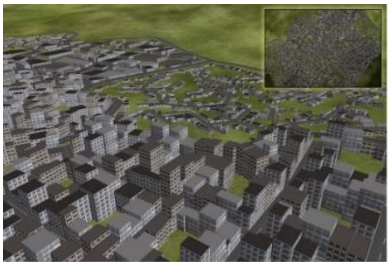


Figure 15.2: Real-time screenshot capture (Kelly & McCabe, 2007)

Template Based Generation of Road Networks for Virtual City Modelling – Sun & Baicu

In 2002 Sun, Baicu, Yu and Green published their paper titled *"Template Based Generation of Road Networks for Virtual City Modelling"* (Sun, et al., 2002). This paper proposes the approach of using a collection of simple templates and a population-based template to procedurally generate the road network of a virtual city.

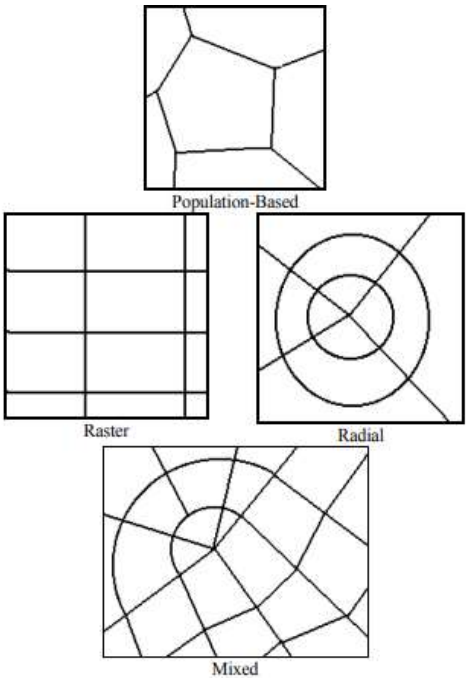


Figure 16: Simple Road pattern templates (Sun, et al., 2002)

The population-based road pattern as the top of the image above is a Voronoi Diagram while the two beneath (Raster and Radial) are simple growing patterns like that of a L-System. The last pattern template is "mixed". This is a compound of all three template techniques above it.

The paper states that the system also requires inputting several more 2D image maps. These are:

- A colour image map that holds local geographical data.
- An elevation map to specify height of the area.
- A population density map to determine road network density.
- A district map to defines different road patterns.

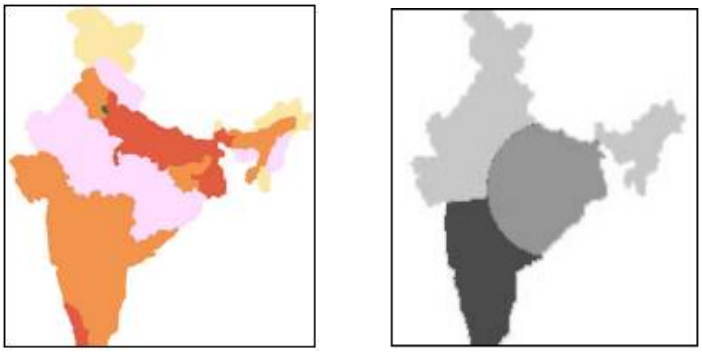


Figure 17.1: Population desnity map of India and Figure 17.2: District map of India

The picture on the left (Sun, et al., 2002) shows a population density map of india, with the darker colours representing more people living there. The picture on the right is a district map with the light grey representing a population-based road network, the middle grey being a radial road pattern and the dark grey signifying the raster road pattern. The image below is the resultant road map network of mixed patterns.

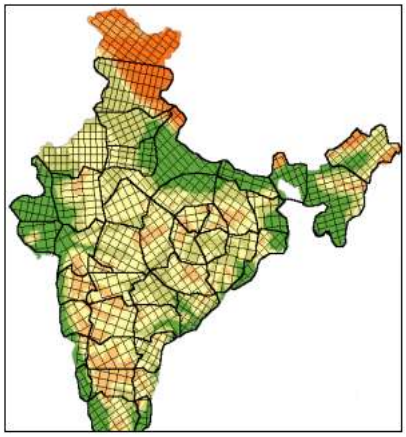


Figure 17.3: Generated road map of India

Chapter 3 – System Design and Implementation

3.1 – Introduction

This chapter will provide a detailed account of what was accomplished in the project, how it was accomplished and why that feature was incorporated. Within the chapter we will look at how the System requirements (as stated in chapter one) were achieved. Advantages and disadvantages regarding decisions for the design of the system will also be discussed. The chapter will also focus on the implementation of the project as well as the tools used in this process.

3.2 – Planning

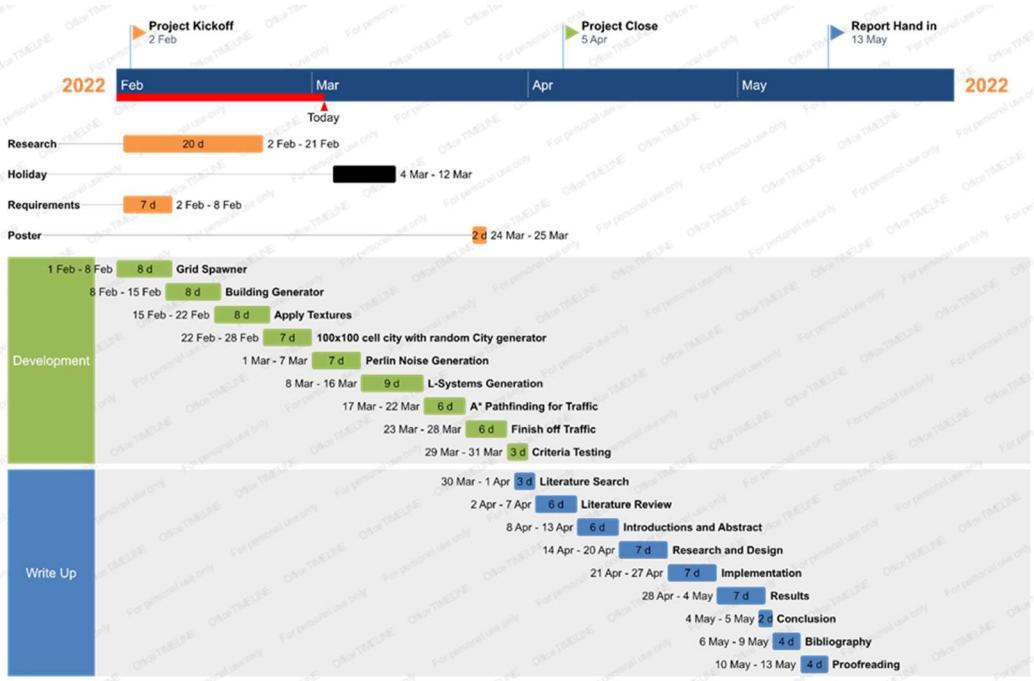


Figure 18: Gantt Chart of project timeline

The Gant chart above shows the breakdown of tasks that I must complete to get the project finished on time. I have decided to opt for using the SCRUM methodology having weekly meetings with my supervisor and having a new sprint task to complete before the next meeting. Due to the nature of the project, comparing "visual" results, starting the development before any write up began made sense. The procedural generation techniques used here however are concepts that I had not come across before. Therefore, while developing the base of the project i.e., grid and building spawners, I have also been researching into procedural generation techniques and deciding which ones I believe will be viable for a city generation project.

3.3 – Tools and Technologies

Unity

This project was implemented using the Unity Engine with the scripts being written in C#. While the project could have been implemented without the use of a game engine, using one helped reduce the amount of initial code that would have been required. Using an engine would also help tackle performance issues. Other games engines such as Unreal were also considered however this too would have increased the initial workload as time would have to be allocated to becoming familiar with the engine and the C++ programming language. Unity also comes with a great deal of resources with its wealth of documentation, vast code libraries and strong online community. The engine comes with a strong debugging feature that allows for simple and easy investigation when problems arise.

The final reason the decision was made to use Unity is because of my familiarity with the tool and belief that this is the best approach to fulfilling the objectives and achieving the System Requirements.

3.4 – Initial Design and Implementation

Version 1.0, (V1.0)

Design

The chosen design layout for the city generator is a Manhattan Grid. This style was chosen as having a city made of blocks which themselves are made of allotments for buildings creates a straightforward hierarchical structure. This could then be implemented and modified in an easier fashion than that of a European style city, such as London or Paris, which have been growing for centuries and contain many conflicting urban planning models.

Figure 19: Snippet of lower Manhattan taken from google maps

The initial design for the city was to generate the city blocks on top of a ground plane which would then function as the roads for the city. In this project a block is defined as an area enclosed by two or more roads. When generated the city blocks are assigned a district number, 1, 2 and 3. These numbers would correspond to a district type of either residential, industrial, or commercial. When assigned a district type the block would then generate buildings of the same type for each allotment is contains.

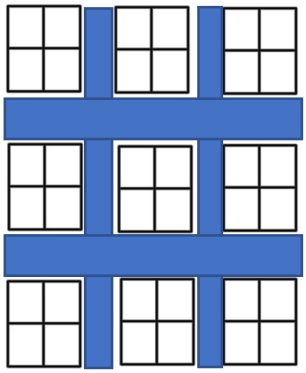


Figure 20: Initial design for part of the city

The buildings in this project are placeholders to show the layout of generated city. They are to be made up of cube shaped objects with the dimensions 1x1x1, where one is a Unity unit (equivalent to one metre).

Implementation

How the V1.0 of the system worked was by having an empty *GridSpawner* object use the GridSpawner script to generate *Building Block* objects. These *Building Block* objects are placed a certain distance

apart that the user inputs, this is the grid offset. When a block is instantiated, it is randomly assigned a district number, in the range of 1-3, using Unity's Random.Range function.

The buildings are generated by the block object. The *Building Block* object also contains a GridSpawner script but has different inputs to that of the *GridSpawner*. The city block instead generates the buildings but only separates them with a grid offset of one. This tightly packs the objects together to ensure there are not gaps between the 1x1x1 dimension cubes. Before the building is spawned the city block checks which district it was assigned. This allows for the district to know at what height it should stop generating cubes and what material should be attached to them. The buildings are then assigned to be a child for the block. This helps the user understand which buildings belong to a certain city block when looking through the scene hierarchy. It also helps any further implementation that would need to modify the buildings within a block.

```
void SpawnGrid()
{
     // Go along the x axis first
    for (int x = 0; x < gridX; x++)
    {
         // Go along z axis for current x
        for (int z = 0; z < gridZ; z++)
        {
            if(prefab == "Block")
            {
                // Instantiate building block prefab and randomly set district number
            }

            if (parentObject == "Grid Spawner")
            {
                StackCubes(x, z);
            }
         }
```

Pseudocode for grid generation

```
void StackCubes(int x, int z)
{
    if (areaBlock == 1)
    {
        for (int a = 0; a <= smallHeight; a++)
        {
         // Instantiate city cube prefab, attach residential material and set city block object
         as parent
        }
    }

    if (areaBlock == 2)
    {
        for (int a = 0; a <= mediumHeight; a++)
        {
         // Instantiate city cube prefab, attach industrial material and set city block object as
         parent
        }
    }

    if (areaBlock == 3)
    {
        for (int a = 0; a <= largeHeight; a++)
        {
         // Instantiate city cube prefab, attach commercial material and set city block object as
         parent
        }
    }
}
```

Pseudocode for building generation

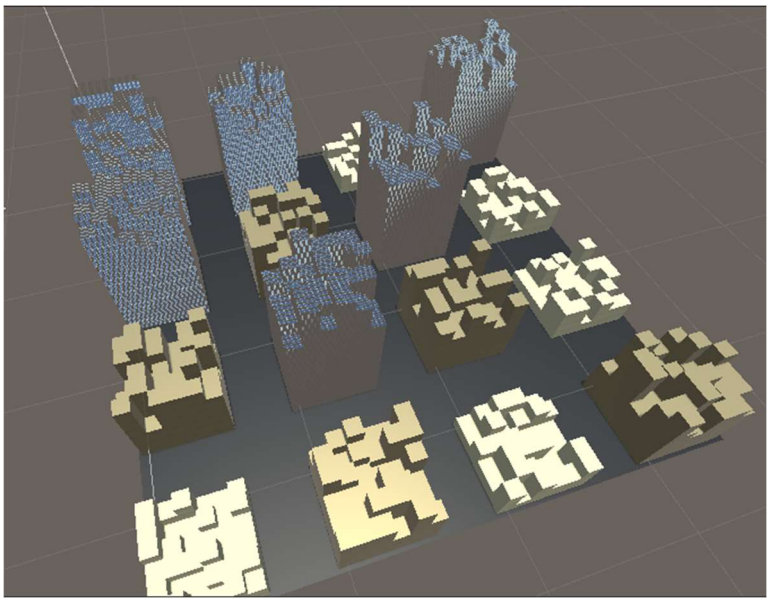


Figure 21: Visual output from V1.0

3.5 – Revised Design and Version 2.0, (V2.0)

Why

After the completing the initial stages of the project the design of the system had to be revised. This is because the design would not be able to meet the requirements 4 and 5. These state that the systems should be able to generate roads and should be scalable. In its current state the system did not generate the terrain which provided the road network. Furthermore, if the initial grid size was increased it would be difficult to know what dimensions to scale it to. Version 1.0 did also not fully satisfy requirement two, which stated that there should be variation throughout the city. In this initial version the system generated city blocks all the same building type and therefore similar heights. This not only makes the city look less realistic but also rather dull looking. Additionally, due to the city blocks being placed at the same distance from one another the "roads" also formed the same distance from one another which is not the case in realistic Manhattan grid style cities. Rather than abandon the design all together I sought to evolve it using David S Ebert's principles on Procedurally Generated Content and adapt it to be more *Flexible* and provide more *Parametric Control*.

Grid Structure

Design

The grid used to spawn the blocks and in turn be used to spawn the buildings has been repurposed. In V2.0 only one grid spawner is used. The user will now be able to map out the entire size of the city to their needs. The city will be made up of *Cell* objects that will be offset by a single Unity unit forming a structure like the one below. If a city was to be 100x100 cells long then the first cell to be generated has the coordinates (0, 0) and the last cell will have the coordinates (99, 99). These are the x and z coordinates of the cell. The y coordinate here is redundant as all generated cells hold a y position of zero.

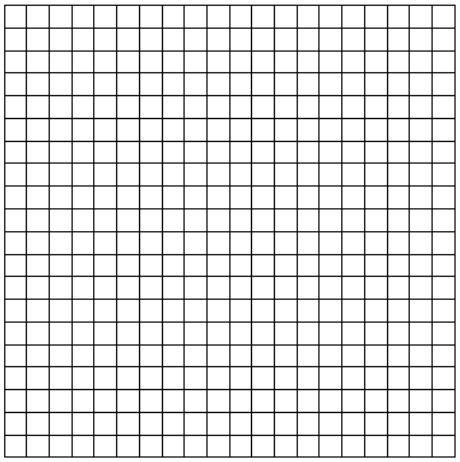


Figure 22: 20x20 grid of empty cells.

These Cells are then to be stored in a 2D array for easy access. For example, to locate a cell with coordinates (3, 27) you would search the array at location 2D*ArrayOfCells*(3, 27). By using this grid structure and storing the cells like this it allows for city blocks to contain more variety as a cell/ allotment's building type is defined by a user or by an algorithm.

Having the grid set up this way will also simplify any data sampling processes from textures. For example, if the city layout of size 256x256 is decided by a texture that is also of size 256x256 then each individual texel would explicitly define the building type for a cell with the same coordinates.

Using a grid system and an offset of 1 Unity unit between each Cell fulfils the 3rd System Requirement. With each building being only 1 Unity unit in length and width there is extremely low probability that buildings will clip into each other.

Implementation

The Grid Spawner in V2.0 works the exact same as it did in V1.0 but with some modifications. The system instead of city blocks generates a 2D grid of Cell objects. These Cell objects contain a *CellID* component. This *CellID* is an integer constant defined in the Grid script which is characterized as a Cell's *State.* The possible states a Cell can be in is Undefined (0), Road (1), Residential (2), Industrial (3), Commercial (4) or an Empty Lot (5). When a Cell object is generated, it is generated with a Grid.State Identifier of 0 and therefore unassigned to any building type. The *CellIDs* are then stored in a 2D array with their world position stored as their index. The Grid Spawner is then assigned as the Cells parent.

```
void SpawnGrid()
{
    for (int x = 0; x < gridX; x++)
    {
        for (int z = 0; z < gridZ; z++)
        {
            // Update is called once per frame
            cells[x, z] = Cell.GetComponent<CellID>();
            // Update is called once per frame
        }
    }
}
```

Pseudocode for V2.0 grid generation

Road and Block Generation

Design

After the city allotments (Cells) have been generated the next stage to is to generate the roads and blocks. Since the city block is just an abstract representation of a rectangular area surrounded by streets the city block is just the result of the roads that surround them. Therefore, to create a city block, roads must be generated. The first step will be to generate the main roads running through the city from the bottom of the grid to the top and the second step will be to generate secondary roads from left to right of the grid. From looking at example cities such as New York you can see that the largest width roads travel from North to South with smaller roads connecting the larger ones together and occurring more frequently. Once all the roads have been generated the city will have been divided into a non-uniform grid of city blocks.

Implementation

Unlike V1.0, this version of the project contains road generation. As stated earlier the first roads to be generated are the main roads. The main roads are to run vertically through the city i.e., along the z-axis. The system starts at position 0,0 and goes across the x-axis finding start positions for main road generation. As main roads occur less frequently in grid style cities a *roadGap* is introduced. This gap can only be between sizes 10 – 14 and it changes every time a new main road is selected. To visually differentiate main roads from secondary roads the neighbour cells in the column to the right are also selected to be a main road.

Secondary roads are generated in the same way, but they run along the x-axis. As they occur more frequently than main roads the *roadGap* is shortened to be between 5 and 10. Unlike Version 1.0 this allows for variation in block size and provides the non-uniform grid structure.

```
while (!mainRoadsComplete)
{
    for (lastRoadX = 0; lastRoadX < gridX; lastRoadX++)
    {
        // Main roads occur less so large gap needed
        int roadGap = Random.Range(10, 14);
        lastRoadX += roadGap;

        if(lastRoadX >= gridX)
        {
            break;
        }

        for(int z = 0; z < gridZ; z++)
        {
            cells[lastRoadX, z].ID = State.ROAD;
            // Convert Cell next to current position into a road
            if (lastRoadX + 1 < gridX)
            {
                cells[lastRoadX + 1, z].ID = State.ROAD;
            }
        }
```

Code for main road generation

```
while (!smallRoadsComplete)
{
   for (lastRoadZ = 0; lastRoadZ < gridZ; lastRoadZ++)
   {
       // Small roads occur more so smaller gap needed
         than main roads
       int roadGap = Random.Range(5, 10);
       lastRoadZ += roadGap;

       if (lastRoadZ >= gridZ)
       {
           break;
       }

       for (int x = 0; x < gridX; x++)
       {
           cells[x, lastRoadZ].ID = State.ROAD;
       }
   }
}
```

Code for secondary road generation

Building Generation

Design

The last stage of the generation process is that of the buildings. When the system reaches this point a full grid must have been developed with the road network having been generated, forming city blocks. Implemented algorithms have access to the 2d array stored within the Grid object. This approach allows them to be able to access each individual Cell and their State ID. The system will only allow the State ID to be changed to a named integer constant, the names are of a building type. When this is changed, the Cell will generate a building of that type on its current position.

Implementation

The building and road generation is performed by the Cell objects that make up the grid. When a specific *CellID* is selected by an algorithm it can change its initial State from Unassigned into a road or specific building type. When one of these are selected the Cell becomes "locked". This means that a Cell cannot change back to being unassigned or switch to another building type. If a building type is selected it, will one of Residential, Industrial, or Commercial. To ensure variety throughout the city, each building type has a minimum and maximum height it could be. The *updated* Boolean check ensures the satisfaction of the 3rd System Requirement. This makes sure that a Cell is not used twice and that buildings are not overlapping.

```
if (ID == Grid.State.ROAD)
{
    if (!updated)
    {
        // Instantiate road object and set Cell as its parent
        updated = true;
    }
}

if (ID == Grid.State.COMMERCIAL)
{
    if (!updated)
    {
        int largeHeight = Random.Range(7, 15);

        for (int a = 0; a <= largeHeight; a++)
        {
          // Instantiate commerical building block and set
          Cell as its parent
        }
        updated = true;
    }
}
```

```
if (ID == Grid.State.RESIDENTIAL)
{
   if (!updated)
   {
     int smallHeight = Random.Range(0, 3);

     for (int a = 0; a <= smallHeight; a++)
     {
         // Instantiate residential building block and
         set Cell as its parent
     }
     updated = true;
  }
}
if (ID == Grid.State.INDUSTRIAL)
{
   if (!updated)
   {
      int mediumHeight = Random.Range(2, 6);

      for (int a = 0; a <= mediumHeight; a++)
      {
         // Instantiate industrial building block and set
         Cell as its parent
      }
      updated = true;
   }
```

Pseudocode for road and building generation

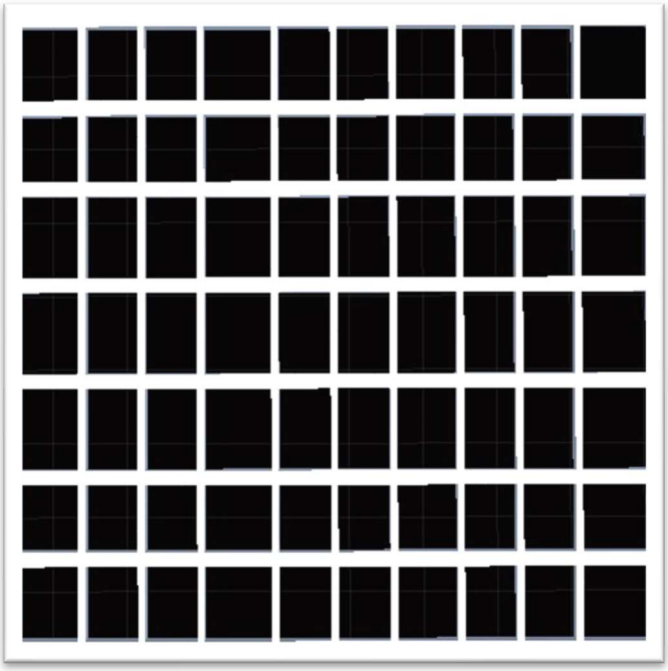


Figure 23: Generated 100x100 Grid and road network forming 70 City blocks

Random

Why

The first algorithm to be implemented is the “Random” one. This algorithm was implemented because it provided a strong starting point in showing that a city could be generated. A Random building generator algorithm was chosen to see what would happen if there was no real Parametric Control in the system for the user.

Design

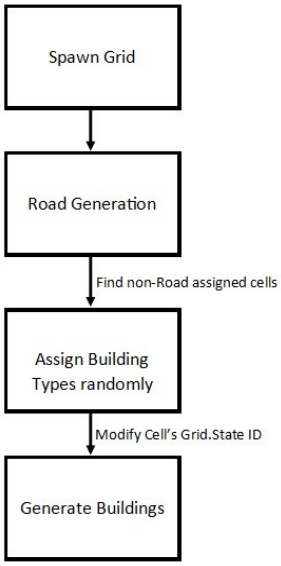


Figure 24: Program Structure for Random implementation of the System

The system should allow for the selection of a pseudorandom generated city. When selected the algorithm should search through the grid for unassigned Cells. When a Cell is located the algorithm will assign the Cell a random building type. After assigning the Cell a building type the algorithm will move on to the next applicable Cell.

Implementation

The algorithm’s first step is to search through the 2D array of *CellIDs*. The algorithm then checks if the current Cell is assigned to a road. If the Cell is assigned to a road, then it will skip over that Cell and continue along the z-axis. If the Cell is found to be unassigned then it will randomly assign a number between zero and nine to an integer variable, this satisfies System Requirement six. If the number is five or below then it will assign that cell to be a residential building. If the number is eight or below it will assign the industrial building type to that cell with only values of nine assigning cells to the commercial building type. This makes the probability of a cell producing a building type look like this, Residential > Industrial > Commercial. The reasoning for this is because of land use requirements in cities a residential building occurs more frequently than any other building type, and industrial buildings tend to procure more space than taller commercial office buildings.

```
private void GenerateRandomDistricts()
{
    for (lastGridX = 0; lastGridX <= 99; lastGridX++)
    {
        for (lastGridZ = 0; lastGridZ <= 99; lastGridZ++)
        {
            if (cells[lastGridX, lastGridZ].ID != State.ROAD)
            {
                int buildingType = Random.Range(0, 10);

                if (buildingType <= 5)
                {
                    cells[lastGridX, lastGridZ].ID = State.RESIDENTIAL;
                }
                else if (buildingType <= 8)
                {
                    cells[lastGridX, lastGridZ].ID = State.INDUSTRIAL;
                }
                else if (buildingType <= 10)
                {
                    cells[lastGridX, lastGridZ].ID = State.COMMERCIAL;
                }
            }
```

Algorithm for Generating Random City

Perlin

Why

The second technique implemented in this project is Perlin Noise. Perlin Noise was chosen because it is a widely used concept within the realm Procedurally Generated Content and that it is feasible to implement within a short amount of time. As stated in Chapter 1, one of the main reasons for using PGC is that it reduces the amount of time needed to generate content. This makes Perlin Noise appropriate for this project as there would not be any use for it if the implementation time was greater than the time it would take to create a city by hand. The technique offers reproducible textures that can be sampled creating minimal storage requirements and efficient generation. Perlin noise tends to produce pseudorandom noise that looks more natural than it does random, making it a useful tool for wanting to generate a natural looking city.

Design

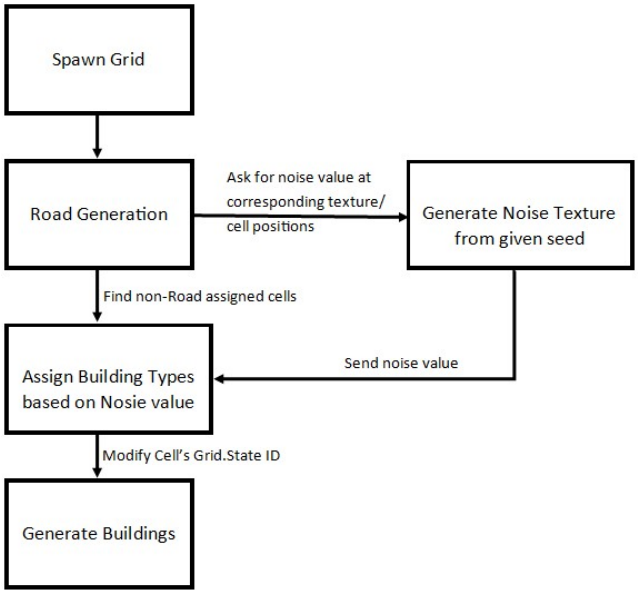


Figure 25: Program Structure for Perlin Noise implementation of the System

The system should allow a city to be generated using the Perlin Noise technique. Once the Grid has been spawned and the roads have generated to provide the non-uniform set of city blocks, the system will begin to search through the 2D array of Cells to find unassigned allotments. When an unassigned

Cell has been located it will request, using the Cell's x and z coordinates, the corresponding greyscale value from a pixel within a generated Perlin Noise texture.

To ensure Parametric Control and Flexibility the texture will be generated using a seeded function, which will allow for the same texture to be generated each time if the parameters are not changed. The algorithm also allows for a change in the noise's persistence. The persistence is the amount of "zooming out" on the texture. A low persistence will give the appearance of a smooth texture while a high persistence will generate a jagged noisier texture. If the user is comfortable with a chosen level of persistence but would like a different layout of the city, the system is able to implement a random offset feature that can provide this.

Once the Perlin Noise has been generated and the greyscale values have been requested and sampled by the Grid, the Cells are allocated their building type. This is similar to the "Random" technique as a Cell's likely assigned building type will be structured as follows, Residential > Industrial > Commercial.

Implementation

Generating Noise

To achieve a Procedurally Generated City using Perlin Noise a noise texture must first be generated. This texture is created when the system starts and checks if the Perlin Noise method is enabled. If enabled the algorithm will perform a "random offset check" so see if the user would like to use a random seed for this city generation. If this too is enabled, then a Vector 2 variable, *perlinOffset*, will be assigned with x and y coordinates within the range of 0-99999.

The next step of the Noise generation process is to create a 2D texture to store the noise map. The size of which is also determined by the user. After the texture is created the system traverses through each pixel of the texture and samples the Perlin Noise at the same position as the pixel.

To sample Perlin noise the relative x and y coordinates need to be established. This is achieved through dividing the texture coordinates with the size of the texture and then multiplying by the *noiseScale* (persistence). To allow for flexibility and an element of randomness these coordinates can be adjusted using the *perlinOffset*. These coordinates are then sent through Unity's Perlin Noise function that outputs a float that are stored as the RBG values for a Colour variable. This *perlinColour* is subsequently returned and set as the original pixels colour. This process is performed for all pixels on the texture. Due to how Unity works whenever colour data is changed on a texture it must then be applied to the texture, and this is the last stage in the noise generation procedure. The resultant texture and code below show that the 7th System Requirement of being able to generate a Perlin Noise texture based on user input has been realised.

```
Color SampleNoise(int x, int y)
{
    float xSample = (float)x / textureSizeX * noiseScale + offset.x;
    float ySample = (float)y / textureSizeY * noiseScale + offset.y;

    float sample = Mathf.PerlinNoise(xSample, ySample);
    Color noiseColour = new Color(sample, sample, sample);

    return noiseColour;
}
```

Code for generating Perlin Noise for a given texel position

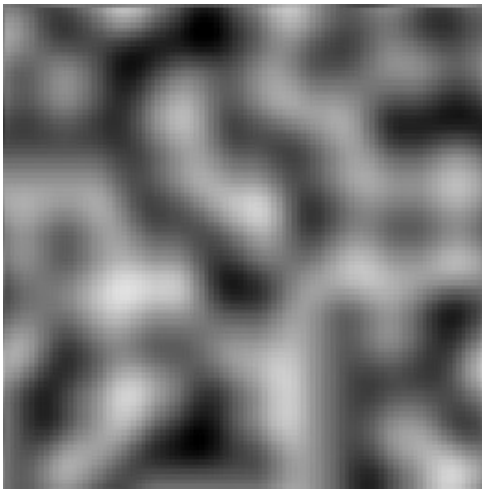


Figure 26: Perlin Noise map generated using seed: Perlin Offset (22888, 53843) and Noise Scale = 6.

Sampling the Nosie Texture

To satisfy the 8th System Requirement of being able to sample generated textures the system must first find an unassigned Cell. When an unassigned Cell has been located the system then tries to acquire the relevant greyscale value from the generated noise map texture. The algorithm first goes about this by converting the Cell's world space coordinates to an approximate position within the Perlin noise texture. After the x and y coordinates to sample have been calculated the next stage is to work out the *gridStepSize*. To work this out the *perlinGridStepSizeX* and *perlinGridStepSizeY* variables containing the length of each grid axis are provided. This then divides the *textureSize* variables, allowing the system to sample the Perlin texture at a different resolution. For example, if the grid was of 100x100 Cells and the texture was of 256x256 then the city would be generated using a lower resolution Perlin Noise texture with a stepped-down size of 100x100.

These sample coordinates and *stepSizes* can then be used to identify the correct pixel within the texture and access its greyscale attribute.

```
public float SampleStepped(int x, int y)
{
    int gridStepSizeX = textureSizeX / perlinGridStepSizeX;
    int gridStepSizeY = textureSizeY / perlinGridStepSizeY;

     float sampledGreyscale = perlinTexture.GetPixel((Mathf.FloorToInt(x * gridStepSizeX)),
     (Mathf.FloorToInt(y * gridStepSizeY))).grayscale;

    return sampledGreyscale;
}
```

Code for Sampling noise texture

Assigning Building type

Once the greyscale data has been obtained it be used to sort Cells to their specific building type. If the greyscale value is below 0.4 the Cell is assigned as a residential allotment. If the greyscale value is below 0.75 then the Cell is designated an industrial lot and anything above 0.75 is assigned as a commercial lot.

L-System

Why

The final PGC technique in this project for generating cities is the Bracketed L-System. This method has been proven to be effective as it can produce infinite variations of complex models in a short amount of time using a concise set of rules that are easily adjustable.

Design

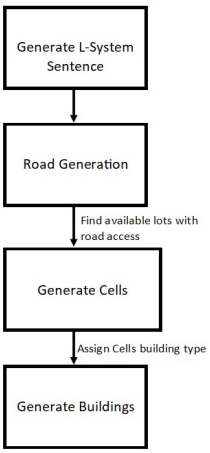


Figure 27: Program Structure for L-System implementation of the System

As stated in Chapter 2, L-Systems are a type of formal grammar that need to have a set of rules. Due to this nature of L-Systems the *GridSpawner* used in the first two PGC techniques will not be deployed for this implementation. For this part of the project, I have chosen a set of rules that will create a realistic road network.

These rules are as follows:

> Where **F** is place road forward, **–** is turn left, **+** is turn right, **[** is save position to stack and **]** is load position from top of stack.
>
> **F –> [+ F] [– F]**
>
> **ω = [F] – – F**
>
> **θ = 90o**

After the first iteration the resultant sentence is **[F [+ F] [– F]] – – F [+ F] [– F].** The system will be able to return a visualised road that is shown below.

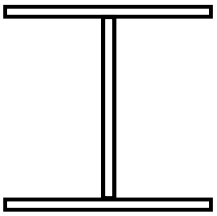


Figure 28: Visual Representation of L-System after first iteration

To be a real Procedural Content Generator the system must allow for *Flexibility.* Therefore, this method allows for the creation of new rules, modification of the rules and the use of multiple rules within the same generation. The addition of having multiple rules allows for different outputs every time the scene on Unity is played.

The generated roads are to be stored in a dictionary with their relative world positions. Once all roads have been generated the system will start to generate the buildings. The buildings in this technique are only placed where there is road access. This is unlike the previous two techniques where all unassigned Cells i.e., empty lots, were filled with a specific buildings type.

Implementation

L-System Sentence Creator

The first stage of the implementation was to create the Rule class. This class is important as it allows the creation of multiple production rules (**[+ F] [– F]**) and the assignment of an input character (**F**). The Rule class extends *ScriptableObject*. This allows the user to create assets that contain the Rule information and reuse the rule multiple times if more than one city was being generated. This also satisfies the 10th System Requirement stating that the system should allow the user to create new rules to be input back into the system.

The next stage is to implement the L-System Generator. One of the variables this script will take in is a public array of Rules which will be the space where we input all the rules we would like to use in this L-System. The class will also take in an axiom or *rootSentence* and *iteration* limit that can only be in the range of 0 to 10.

The first and only public method in this class is the *GenerateSentence* function. This function sets the initial starting *word* and returns the output from the recursively grown *word* from the L-System based on the number of iterations.

To grow the word into a sentence the word and rules must be processed recursively. To do this StringBuilder *resultantWord* is used to hold the final sentence. A StringBuilder is used instead of a regular String as its value is mutable and can manage larger Strings without affecting performance. The next stage is to loop through each character in the *word* and append it to the *resultantWord*. After this the *resultantWord,* the character and the iteration index are passed through to *ProcessRulesRecursively* function. This process happens for every character in the rootSentence.

The final function *ProcessRulesRecursively* starts by looking through each rule within the rules array. The system should then check if the Rule's input letter is the same as the character sent from the previous function. If this is true, then the *resultantWord* will have call and append the output of the *GrowRecursive* function as there are new symbols within the Rule that now must be processed as long as the iteration limit is not reached.

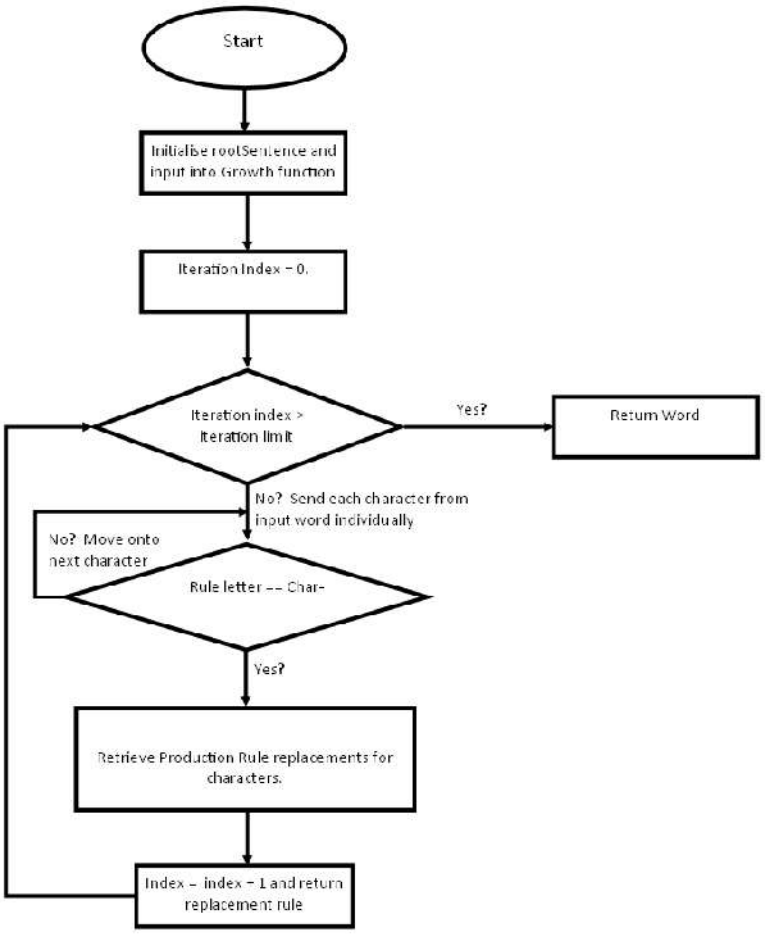


Figure 29: Flowchart for L-System Generator

```
private string GrowRecursive(string word, int iterationIndex = 0)
{
    if (iterationIndex >= numOfIteration)
    {
        return word;
    }
    StringBuilder resultantWord = new StringBuilder();

    foreach(var a in word)
    {
        resultantWord.Append(a);
        ProcessRulesRecursively(resultantWord, a, iterationIndex);
    }

    return resultantWord.ToString();
}

private void ProcessRulesRecursively(StringBuilder resultantWord, char a, int iterarionIndex)
{
    foreach (var rule in rules)
    {
        if(rule.letter == a.ToString())
        {
            resultantWord.Append(GrowRecursive(rule.GetResult(), iterarionIndex+1));
        }
    }
}
```

Recursive algorithm for growing L-System sentence

Road Generation and Visualisation of the L-System Sequence

Agent

As stated in Chapter 2 of this dissertation, “one way to visualise L-System is to use the generated string as instructions for an agent to move across a surface and trace its path”. This “Agent” is the next stage of the L-System implementation. The Agent class is a helper class that will store its position, direction, and length to “draw”.

Encoding the Sequence

This RoadVisualiser class will be responsible for drawing the visualisation of what has been outputted from the L-System Generator. When starting the visualisation process, a *sequence* (L-System sentence) is generated and then sent to the *VisualiseSequence* function.

A public enumerator, *EncodingLetters*, is established so that there is a way to map the sequence of characters to their correct commands for the Agent to use. This addition satisfies my 11th System Requirement that states the system is responsible for encoding the sentences and ensuring that symbols have been given their legitimate corresponding function.

The first step in the *VisualiseSequence* method will be to implement a collection that will gather saved and load agent parameters when the appropriate commands is called. This collection is to be a stack as it provides the needed functionality of last in first out.

The next step of this method is to iterate throughout all the characters in the *sequence* and ensure that each letter in the returned L-System sequence is mapped inside the enumerator. If a character in the sequence is mapped within the Encoding, a Switch statement is performed with each command defining a separate case.

When the system comes across a “[“ i.e., save, in the sequence it pushes the Agent’s current position, direction and draw length onto the stack, *savePoints*. If a “]” i.e., load, is found in the sequence then the *savePoints* stack is to pop the last Agent information into the current Agent’s parameters. If the Encoded command of “F” appears in the sequence then the Agent will place a road down from its current location for a certain length, in its current heading. This Agent length is then subtracted by two creating a road network where the roads become shorter the further away they are from the city centre. Here a fractal look as stated in Chapter two is achieved where one is defined as *"a rough or fragmented geometric shape that can be split into parts, each of which is a reduced-size copy of the whole”*. The last two commands that could be in a sequence are “+” and “–“. If these appear in the sequence, then the Agent’s direction will be changed ±90°.

Building the Road

When the draw command is found in a sequence the Agent will send a starting position, direction and draw length to a road builder function. To build the road correctly and make sure there are no duplicates, the position of each possible road to be built is checked a *roadDictionary*. This dictionary stores the position of the road and the road object itself. If a road has already been placed at the queried position in the dictionary, then the road builder moves onto the next position. If a queried position is not in the dictionary, then the road builder instantiates a road at that position and then adds it to the dictionary. The visual outputs below show that the 12th System Requirement of being able to construct a road network, from a user’s initial sentence and rules, has been satisfied.

Figure 30.1: Road generation after 0 iterations

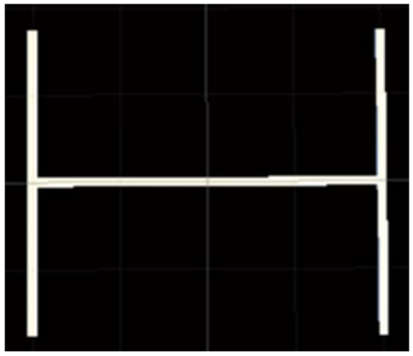


Figure 30.2: Road generation after 1 Iteration

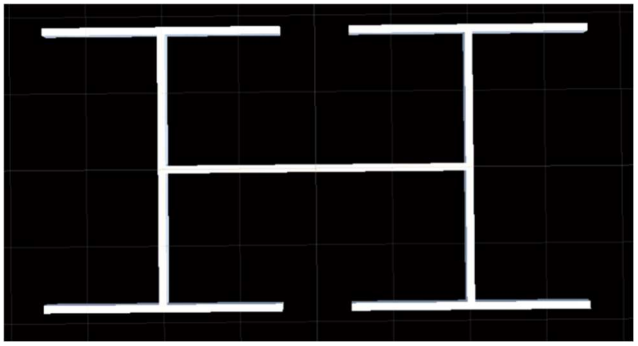


Figure 30.3: Road Generation after 2 iterations

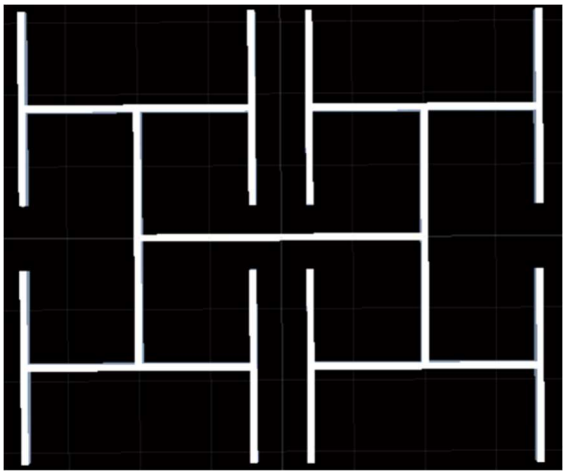


Figure 30.4: Road generation after 3 iterations

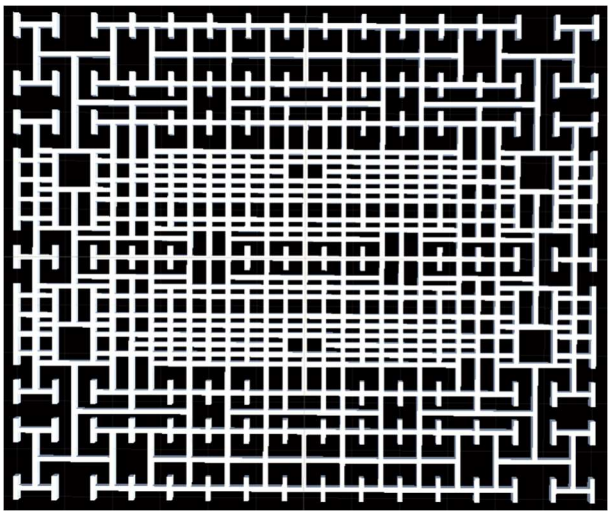


Figure 30.5: Road Generation after 10 iterations

Building Generation

The last stage of this City generator it the placement of Cells and assigning building types to them. Once the roads have been generated and placed into the road dictionary the system goes through the dictionary to find empty coordinates/ allotments with road access. The system will generate a Cell every 1 Unity unit around each road object.

To assign a building type a simple layout for the city zones has been designed. If the *CellID* is within the *smallRange*, which is 0 to 20, from the origin then the Cell will be assigned to a commercial building. The system then assigns any Cells between the *largeRange,* which is *smallRange/2* to 40, from the origin as an industrial building. All other plots of land with road access are then assigned to be residential buildings. As these ranges are updated for every road tile it produces an element of randomness as well as creating a border competition effect. Having these update for every road block also means that every city building layout is unique for each generation.

3.6 –Testing

As stated in Chapter One there are six criteria to which the City Generators will be judged on. The third of these is the amount of time it took to generate the city. To measure this the *Stopwatch* class from the System.Diagnostics library was used. The *Stopwatch* measures the elapsed time by counting the ticks from the core timer mechanism.

For each technique, the *Stopwatch* is started before the generation starts. The *Stopwatch* then stops after the city generation has taken place. The elapsed time of the stopwatch is then output to the console in milliseconds.

3.7 – Summary

This chapter has been used to present a detailed approach to the design and implementation of the project with all 12 System Requirements that were set out in Chapter One being satisfied.

Chapter 4 – Results and Evaluation of Techniques

4.1 – Overview

This chapter provides the results and evaluation of the project and the Procedural Content Generation techniques used. This will be done using the six Criteria that were set out in the introduction. By using these criteria, the benefits and drawbacks of each approach can be discussed easily.

It is worth noting that when it came to running and testing this project the computer used had the following technical specifications:

CPU = AMD Ryzen 5900x

RAM = 32GB

GPU = GeForce RTX 3070

4.2 – Random

The first technique to be assessed by the six criteria is the pseudorandom technique where the user's parametric control is limited.

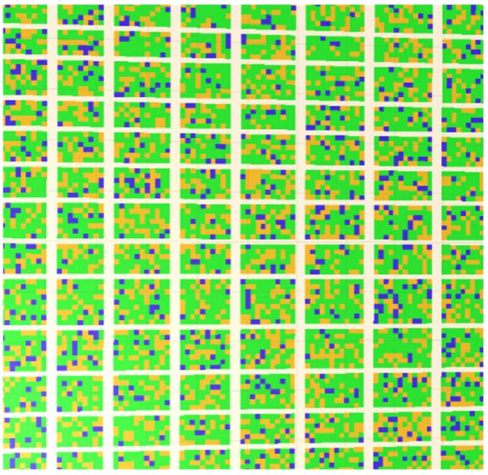


Figure 31.1: Bottom-up view of generated city

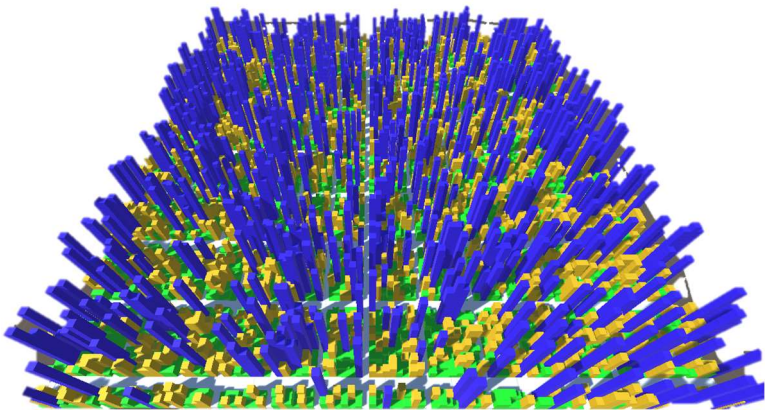


Figure 31.2: Generated City using random technique

Does it look realistic?

The "Random" technique generates a city where the user input is little to none. From the bottom-up view you can see that the majority of the Cells were assigned a Residential building type. Yet, in the figure on the right the taller commercial buildings, while significantly less in

number, seem to overpower the look of the city. This provides an unrealistic interpretation of what a western city should look like, but it does provide a look similar to some cities further East such as Seoul or Moscow which have 16,359 and 12,317 high rise buildings respectively (Keegan, 2019).

Figure 32: Picture of high rises in Seoul City in South Korea (Keegan, 2019)

Is the city scalable in size?

Due to the nature of the Grid format the city is scalable in size. The city however can only be resized in the shape of a rectangle or square.

What is the loading time?

Through the use of the System Stopwatch function I was able to determine the average loading time for the Random City Generation method. The average loading time was 124.45ms (milliseconds). The minimum amount of time taken for the city to be generated was 124ms and the maximum amount of time taken to generate was 127ms.

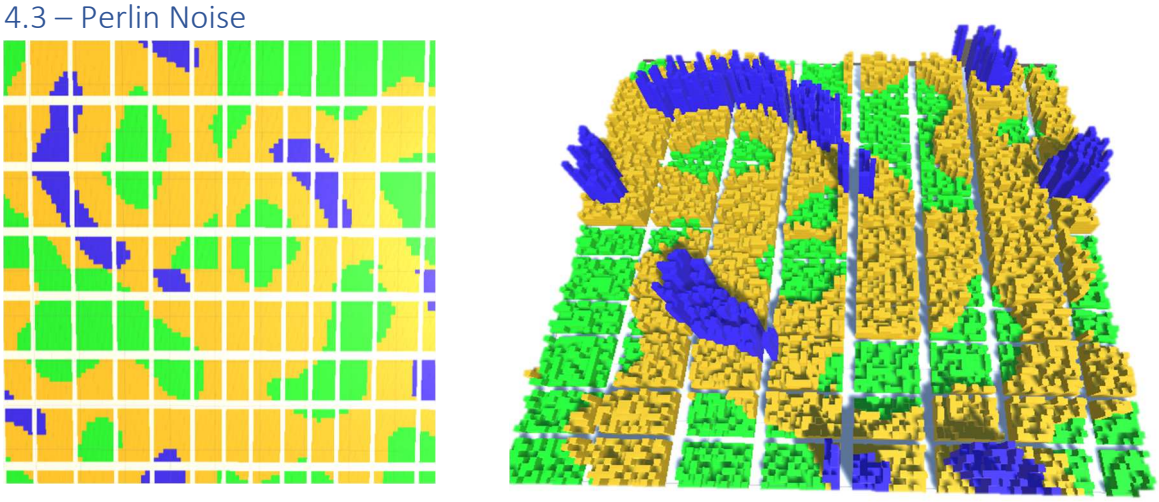
How much user input is needed?

For this technique the only user input needed is for the length of the grid's x and z axis. All other inputs for the placement of the buildings are handled by the algorithm.

How efficient is the overall process?

The overall process from design to implementation is fairly straightforward. The generated city is not a complex one which meant that the design of the algorithm didn't need to be complicated either.

4.3 – Perlin Noise



Figures 33.1 and 33.2 showing generated city and its layout using seed: Perlin Offset (22888, 53843) and Noise Scale = 6.

Does it look realistic?

Depending on the used seed, a generated city can range from looking realistic with a good mix of buildings appropriately clustered together (like the figure above) or, if the noise scale is set too high, a city similar to the Random technique discussed earlier. With this method and appropriate input seed a viable city is generated with a natural looking formation of districts.

Is the city scalable in size?

The Perlin Noise technique generates a city that is scalable. This is done by either changing the length of the grid's x and z axis or by sampling from a texture that has a smaller size axis than that of the grid.

What is the loading time?

The average loading time using the hardware as specified earlier was 126.4ms. The minimum amount of time taken to generate a city using this method was 126ms with the maximum being 127ms.

How much user input is needed?

The technique requires some user input into the system. The user must first know the size of the grid that they will be using so that they place this into the *PerlinGridStepSize* parameters. This allows for the texture to be sampled at a resolution stepped up or down from its own. The User may also input a Perlin offset if a desired seed is wanted but this can be randomised at the user's discretion. The last parameters that have to be input into the system are the Perlin Noise texture size and the Persistence (Noise Scale).

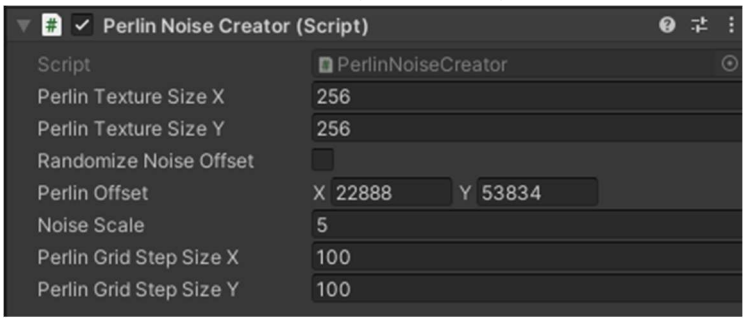
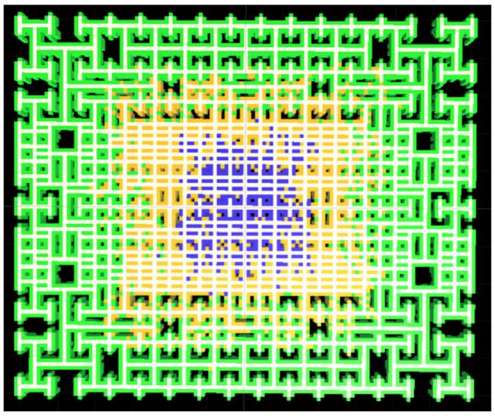


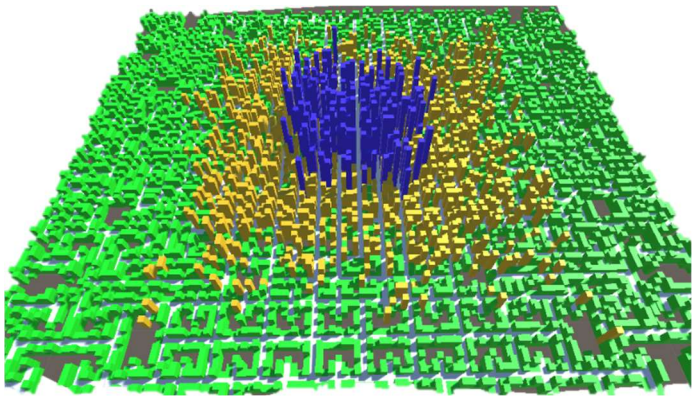
Figure 34: The user input needed for Perlin City Generation via inspector window

How efficient is the overall process?

The overall process is quite efficient with a rather short implementation time that produces a wide range of unique visual results. The design itself could perhaps be modified so instead of creating a texture to store the Perlin Noise map, the noise data itself could be directly sent to the building type assignment part of the algorithm. This would speed up the amount of time taken to generate the city.

4.4 – L-System





Figures 35.1 and 35.2 showing generated city after 10 iterations of L-System with rules F –> [+ F] [– F], ω = [F] – – F, θ = 90º.

Does it look realistic?

The generated Cities using L-Systems do appear realistic. This is mostly due to the generated road network and elements of border competition that are taking place. The road system however does appear quite symmetrical but with additional rules and a change in angle this problem will disappears immediately. This can be seen below where a L-System which has the same rules but a changed angle of 42^0 generates coastal harbour type city.

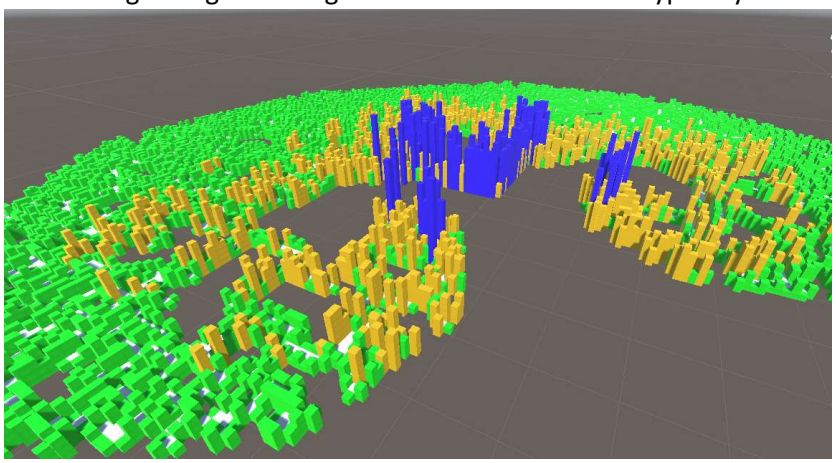


Figure 36: City generated after 10 iterations with rules F –> [+ F] [– F], ω = [F] – – F, θ = 42º.

Is the city scalable in size?

The L-System city generator can produce cities of various scales by changing the road drawing length of the Agent and the angle at which the Agent should turn at. Increasing the drawing length however does lead to a decrease in overall performance and the number of cubes being drawn increases rapidly.

What is the loading time?

With this technique the Loading time varies on the number of iterations taking place, the amount of draw commands in a sequence, the length of the draw command, the angle at which to turn the Agent and the ranges from the origin for commercial and industrial buildings. The following times are for the generation using the rules applied to the first visual example.

- 0 iterations = 4ms
- 1 iteration = 6ms
- 2 iterations = 10ms
- 3 iterations = 15ms
- 4 iterations = 24ms
- 5 iterations = 38ms
- 6 iterations = 57ms
- 7 iterations = 73ms
- 8 iterations = 105ms
- 9 iterations = 119ms
- 10 iterations = 127ms

For comparison using the same production rules but at a different angle of 42° instead of 90° the time taken to generate the harbour type city after 10 iterations was 185ms. This is due to the agent being less likely to revisit past road locations and therefore more roads and buildings are generated.

How much user input is needed?

To use this technique knowledge of L-Systems is needed. If the user understands the concept and how it works then they should be able to come up with their own rules to be input into the system. Other user input such as turning angle, drawing length and number of iterations also need to be put into the system however these are easily modifiable through the inspector.

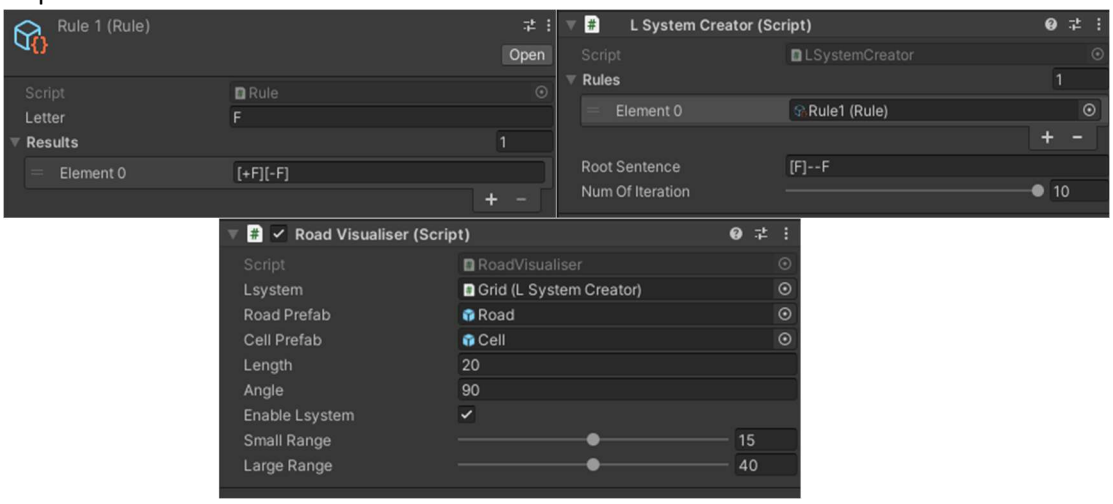


Figure 37: The user input via the inspector for the L-System

How efficient is the overall process?

The L-System technique took the most time to design and implement. The concept of the L-System presented the biggest learning curve for the project. Once the concept was understood designing the system was easier and quicker than initially expected but it did take longer than any of the other techniques. Nevertheless, this technique is an example of the end justifying the means. The L-System method allows for unique city layouts allowing for infinite variations at any scale. This PGC technique allows for distinct natural city layouts to be generated almost instantly, proving that the time taken to develop this method is worth the results.

Chapter 5 – Conclusion

5.1 – Introduction

This chapter will reflect on the project as a whole. It will make a case for why the aim and objectives may or may not have been met and will look to see what has been established within the project. This chapter will also look at what could have been done further/ better as well as what work could be done in the future if the project was to continue on.

5.1 – Satisfaction of Objectives

Objective 1 – To investigate how Perlin noise can be used for zoning in city generation

This objective was fulfilled through research in Chapter 2. In this chapter I didn't just learn how Perlin Noise was generated but also how to approach the task of creating a city using Perlin Noise. Through the work accomplished by Sun and Baicu on "Template Based generation.. for Virtual City Modelling" I came to the solution of sampling data from a 2D image map. This image map would be filled with the Perlin Noise data that could be sampled by the system.

Objective 2 – To find out how L-Systems can be used to generate complex road networks

To satisfy this objective I investigated how others such as Parish and Muller had developed their road networks using L-Systems. The type of L-System they used however was their own extended version. This led to me to read Lindenmayer's book The Algorithmic Beauty of Plants, which made me realise that the methods he used to procedurally generate plants could be adapted to road networks. This could be done by using simpler production rules.

Objective 3 – To design and develop three distinct Procedural City Generators.

This objective was fulfilled in Version 2.0 of the project which was outlined in Chapter 3. Three distinct Procedural City Generators were developed. These were the Random City Generation method, the Perlin Noise Generation method and the L-System Generation method. Each method produces a diverse range of cities each with their own benefits and drawbacks.

Objective 4 – To investigate the efficiencies of different procedural generation techniques.

This objective was satisfied in Chapter 4. To satisfy this objective six criteria were created to assess each procedural generation technique. These criteria were used to provide an overview of each generation method and provide and insight into the viability of each technique as a useful tool for creating cities within the Unity Game engine.

5.2 – What has been established?

In this project three different techniques for Procedural City Generation have been developed. Through evaluating the methods using the six criteria in Chapter 4 it could be said that the best City Generator to produce a viable city is the L-System City Generator. This is due to the fact that the user has more control over each stage of the city generation, allowing for dynamic and unique cities to be generated each time the scene is played.

5.3 – What went well?

The project offers a user-friendly system that allows for easy modification of the inputs. For example, in the Perlin Noise implementation the user can change the persistence of the noise to create natural looking city layouts. The L-System implementation allows for the user to come up with their own rules and design a city to their specific design needs. This user first approach allows an almost infinite number of different styles of cities to be generated without too much prior knowledge of how the concepts behind each implementation actually work.

5.4 – What could have been done better?

The first technique used which generated building types randomly across the city could have been done better. Throughout the process of creating this project I have realised that if more user input is offered, the generated city begins to look more natural. Therefore, instead of randomly assigned each allotment a building type, the generated city blocks could be assigned a district type randomly with the System asking the user how many of each type of district they would like. Border competition could then be introduced to allow blocks to have multiple types of buildings contained within them.

The third technique which generated the city using L-Systems could also be improved. Under certain input angles the generated city could look as if it is all roads with buildings placed round the edges. This is not desirable so a limit on angles could be introduced, or a check could be made on current road width. The building placement in this technique also needs work as currently commercial buildings only spawn around the origin (0, 0).

5.5 – What could be done in the future?

A possible solution to the unnatural building placement in the L-System technique could be to incorporate Perlin Noise. By combining the two techniques I believe it would create a much more naturalistic looking city. The Perlin Noise technique offered a more visually appealing building placement, but the Manhattan grid structure can quickly become unattractive. This is the opposite of the L-System technique where the road network is visually appealing, but the building placement is rather generic.

Another technique that could be explored is that of using a colour image map. Other works by Sun & Baicu and Parish & Muller have used image maps as the basis for their city layouts. Having a user drawn map that could be sampled by the system to generate a city would allow for total design freedom. This is something not fully offered by the three techniques implemented in this project.

Other features such as implementing different types of land use structures would also be welcomed for this project. Currently there are only 4 types of structures that are used to generate a city, and all are made up of cube meshes. These are roads, residential, industrial and commercial buildings. By diverting away from cubes and using actual models of buildings will help make the city feel more natural. The addition of parks, schools and warehouses will make generated cities more diverse and natural feeling for users.

Another feature that would accompany this project well is that of a traffic system. Having cars driving around and obeying the highway code will offer the generated cities the ability to feel lived in.

5.6 – Satisfaction of Aim

The project undertaken was to *"develop a natural looking Procedural City Generator (PCG) on the Unity Engine and evaluate the efficiencies of the different applied techniques"*. This aim has been met with three distinct city generators being developed; one - where the user has very little parametric control; two - where the user has more parametric control; and three - where the user has almost complete control over the entire city layout at their own discretion. These have had their loading times tested and their visual results evaluated. The evaluations have shown that each technique has its own unique advantages and disadvantages and with future extension work planned by the author to merge the concepts into one Procedural City Generator, there is some confidence that this will almost certainly lead to a natural looking city which meets all of the six criteria.

References

Academic, 2002. *En-Academic.* [Online]
Available at: https://en-academic.com/dic.nsf/enwiki/30071
[Accessed 16 04 2022].

Anon., 2007. *WikiMedia Commons.* [Online]
Available at: https://commons.wikimedia.org/wiki/File:Square_koch_3.svg
[Accessed 16 04 2022].

Ardnt, J., 2011. The Thue-Morse sequence. In: *Matter Computational: Ideas, Algorithms, Source Code.* s.l.:Springer, p. 44.

Booth, M., 2009. *The AI systems of Left 4 Dead. In: Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference.* s.l., AIIDE.

Burgess, n.d. [Online]
Available at: https://en.wikipedia.org/wiki/Concentric_zone_model
[Accessed 15 04 2022].

Ebert, D. S., 2014. *Texturing and Modelling: A Procedural Approach.* ISBN: 978148329706 ed. s.l.:s.n.

Ertl, B., 2015. [Online]
Available at:
https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg#/media/File:Euclidean_Voronoi_diagram.svg
[Accessed 14 04 2022].

Keegan, M., 2019. *The Guardian.* [Online]
Available at: https://www.theguardian.com/cities/2019/jul/16/which-is-the-worlds-most-vertical-city
[Accessed 10 05 2022].

Kelly, G. & McCabe, H., 2006. A Survey of Procedural Techniques. *ITB,* Issue 14.

Kelly, G. & McCabe, H., 2007. *Citygen.* [Online]
Available at: https://www.citygen.net/files/citygen_siggraph07.pdf
[Accessed 2022 04 2].

Kelly, G. & McCabe, H., 2007. *Citygen: An Interactive System for Procedural City Generation.* Liverpool, GDTW.

Lagae, A. et al., 2010. *State of the Art in Procedural Noise Functions.* s.l., EUROGRAPHICS.

Lefebvre, S. & Neyret, F., 2003. Pattern Based Procedural Textures. *SIGGRAPH - I3D*, April, pp. 203-212.

Lindenmayer, A., 1968. Mathematical Models for Cellular Interaction in Development Parts I and II. *Elsevier,* 18(3), pp. 280-315.

Mandlebrot, B. B., 1983. *The Fractal Geometry of Nature,* s.l.: Macmillan.

Parish, Y. I. H. & Muller, P., 2001. *Procedural Modelling of Cities.* New York, s.n., pp. 301-308.

Park, R. E. & Burgess, E. W., 1925. The Growth of the City: An Introduction to a Research Project. *University of Chicago Press,* pp. 47-62.

Perlin, K., 1985. An Image Synthesizer. *SIGGRAPH,* 19(3), pp. 287-296.

Perlin, K., 2002. Improving Noise. *ACM Transactions on Graphics,* 21(3), pp. 681-682.

Przemyslaw, P. & Lindenmayer, A., 1990. The Algorithmic Beauty of Plants. In: s.l.:s.n., p. 25.

Shaker, N., Togelius, J. & Nelson, M. J., 2016. *Procedural Content Generation in Games.* 1st ed. s.l.:Springer.

Spelunky. 2008. [Game] Directed by Derek Yu. s.l.: Mossmouth.

Sun, J., Baicu, G., Xiaobo, Y. & Green, M., 2002. *Template-based generation of road networks for virtual city modelling.* New York, s.n., pp. 33-40.

Togelius, J., Katsbjerg, E., Schedl, D. & Yannakakis, G., 2011. What is procedural content generation?: Mario on the borderline. *PCGames '11: Proceedings of the 2nd International Workshop on Procedural Content Generation in Games,* Issue 3, pp. 1-6.

Unity, 2022. [Online]
Available at: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html
[Accessed 14 04 2022].

Wikimedia Commons, 2007. [Online]
Available at: https://en.wikipedia.org/wiki/Koch_snowflake
[Accessed 15 04 2022].

Worley, S., 1996. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques.* s.l., s.n., pp. 291-294.