Procedural Dungeon Generation Using Lindenmayer Systems

Calum John Mathison

40406464

Submitted in partial fulfilment of
the requirements of Edinburgh Napier University
for the Degree of
Games Development

School of Computing

April, 2021

Authorship Declaration

I, Calum John Mathison, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines

Signed: Calum Mathison

Calum Mathison

Date: 20/04/2021

Matriculation no: 40406464

General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below *one* of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

Calum Mathison

Calum Mathison
40406464

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

Procedural generation is becoming an increasingly large part of the games industry. As games become larger and more expensive to produce, procedural generation can be used to lower costs while also producing fun and interesting content. The issue is that current techniques can be flawed and repetitive, lacking direction, meaning that new techniques should be researched to improve upon current ones. L-Systems have proven themselves to be a useful tool in simulations but have seen little use in the games industry outside of procedural plant generation. This study looks at the potential use of L-Systems in producing procedural, yet interesting two-dimensional dungeons maps for use in games of varying genres. The project built upon this idea to produce an application that can procedurally generate these maps, while using L-Systems as the main system and algorithm. Quantitative data was gathered upon completion of the study and while more work should be carried out, does show trend of satisfaction in the technique from participants. The study found that this is a plausible technique but should be used in conjunction with other techniques such as pathfinding or cellular automata to provide a higher quality experience.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr Babis Koniaris for his support, advice, and guidance throughout the project.

I would also like to thank my second marker, Kevin Sim for his support and his advice.

I would also like to thank Dr Thomas Methven for his continued support and work throughout my time at Edinburgh Napier University.

I would like to thank the School of Computing for providing the equipment, environment, and knowledge to enable me to conduct this project and produce the body of work.

I would like to thank my friends and family for their support and guidance throughout my time at Edinburgh Napier University.

Finally, I would like to thank my wife Miranda, for her continued support and understanding, throughout my time at Edinburgh Napier University.

1 Chapter 1 | Introduction

The following chapter outlines and explains the aims and contents of the project. The aim of which is to research and create a working prototype of a piece of software that can procedurally generate two-dimensional dungeons layout through the use of a technique known as L-Systems. The body of work documented below describes in detail, the processes involved in researching, implementing, and testing the application.

1.1 Motivation

Although procedural generation is by no means a new concept within the video games industry, it is important that attempts are made to improve and challenge the varying techniques used, and indeed perform research into potentially new and better techniques, in the pursuit of successfully creating content procedurally that stands up to that created by hand. Procedural generation has been used to create a vast array of objects and its uses can include anything from names and text to models of plants and animals or even entire planets. Procedural generation provides a vital tool to companies of all sizes and can bring benefits of reduced cost or development time, to greater user play time and engagement.

L-Systems form a small but vital addition to the tools available to achieve procedural generation and while their conception lies rooted in the area of creating procedural foliage and trees, as well as fractal patterns, it has been lightly used in conjunction with other techniques to create larger objects such as levels and maps. L-Systems provide a unique advantage as once they are implemented, they are relatively cheap and easy to expand in scale and complexity.

1.2 Aim & Objectives

The aim of this project is to develop an application and analyse its effectiveness of using L-Systems to procedurally generate two-dimensional levels like those used within various video game genres.

The objective of this project is to develop an algorithm using L-Systems that can effectively generate procedural levels that are capable of being completed by a user

and to provide a similar outcome to other more established methods, like Cellular Automata. A testing phase will be undertaken using user testing gather quantitative data in an attempt to ensure that the application's methods hold up to more popular techniques already used within the games industry. Alongside this testing, there will be discussion and evidence provided which will argue that it can be used as a technique either in combination with or instead of more popular methods.

1.3 Research Questions

Over the course of the project, there are three main questions which will be considered.

1. Identify if L-Systems can produce consistently random yet constrained results?
2. How the algorithm and results compare in terms of performance and quality regarding other techniques?
3. What other techniques could be used in conjunction with L-Systems?

1.4 Scope

This subsection outlines the final deliverables expected from the project and outlines the boundaries of the project.

1.4.1 Deliverables

The items to be delivered through this project are as follows; An executable piece of software containing implementation of an L-System generator that will allow for the procedural generation of two-dimensional dungeon levels populated with game elements; The projects files and source code; The results and analysis of the projects testing phases and results.

1.4.2 Boundaries

The final deliverable is not a game or a game engine however game elements will be added for testing and integrity. The final deliverable will only include L-Systems as the main form of procedural generation.

1.5 Constraints

Several factors were identified during the planning phase that could affect the success of the project. The costliest of these factors is time; many factors can affect the time constraint such as the time required to understand and implement all the necessary algorithms. Another potential risk to this project pertains to that of coding

multiple algorithms together in a way that still uses randomness to produce a procedural result and the possibility of spending too much time implementing a single feature of the project.

1.6 Sources of Information

The study conducted for the literature review will be gathered from several sources, both academic and non-academic. The sources that will be used for this project will contain:

- Books
- Past research papers related to the field of study
- Articles
- Websites
- Technical and User Manuals

1.7 Chapter Outlines

- Background – This chapter will outline and present the research previously done into the topic.
- Methodology – This chapter will define and present the specific details and methods used during the implementation phase.
- Results – This chapter will present the results from the testing phase.
- Conclusion – This chapter reflects on how the results meet the aims and objectives and outlines future work to be carried out.

2 Chapter 2 | Background

In this chapter, we will briefly explore several popular techniques used in procedural generation that are also suitable for the problems faced in this project and why procedural generation is such a powerful tool. We will also discuss the uses and previous work done using L-Systems and why they pose a unique instrument to use within procedural generation.

2.1 Procedural Generation

A simple definition of procedural generation is the process of creating data algorithmically as apposed manually. Procedural Content Generation (PCG) can also be described as the automation of media production. This can be defined as anything a human would usually author, spread across any number of different disciplines such as poetry, paintings, music, art, and games (N. Barriga, 2019). N. Barriga states that between 30%-40% of any AAA game's budget is spent on content creation alone, even if procedural generation alone could reduce the overall cost by 10%, it would still be expected to save a AAA company up to $15 million (£11 million) in production costs by simply automating parts of their content generation. The following subsections cover popular procedural generation techniques already established in the games industry.

2.1.1 Perlin Noise

Procedural noise functions are a group of techniques that have been used extensively in computer graphics, including in procedural generation, and is a technique that produces interesting and consistent results when it comes to world generation. Perlin Noise created by Ken Perlin, constructs patterns from bands of noise, each of which is limited to a range of frequencies (K. Perlin, 2002). By creating a two-dimensional array of values and assigning random numbers (usually between 0 and 255; 0 being visualised as black and the latter being white) to each of the values, something like television static can be created. In its simplest form noise should not be considered useful to the problem of world generation however, by interpolating and smoothing between values thus using these algorithms, we can produce something akin to black and white topographical maps (R. L. Cook, T.

DeRose, 2005). By assigning the range of values in the array to height or textures we can begin to render a 3D or 2D world. This technique has been used extensively in games with large procedural worlds, popular titles include Mojang's Minecraft and Re-Logic's Terraria. Both games mentioned use noise extensively to create a near endless supply of worlds for their players. However, this technique is not the most suitable for our problem for creating dungeons due to the overly random nature issues when creating larger noises (Figure 1).

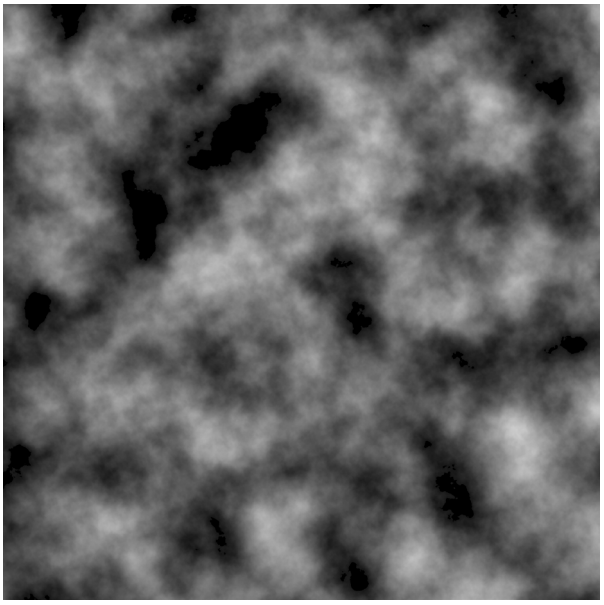


Figure 1: Example of typical Perlin Noise height map

2.1.2 Cellular Automata

Cellular Automata are described by S. Wolfram as simple mathematical idealisations of natural systems (S. Wolfram, 1983). The technique was originally conceptualised by John von Neumann in 1940 (J. Von Neumann, 1966) but popularised by Conway's Game of Life in 1970 (M. Gardner, 1970). A Cellular Automata consists of an n-dimensional grid of cells in a finite state and a set of transitional rules. Each cell of the grid can be in one of several states; in the simplest case, each cell can be on or off. The initial distribution of cell states can be randomised and constitutes the seed of the automaton in its initial state. By creating time steps within the simulation and applying the transition rules during those time steps to all cells simultaneously, we can alter the states of each cell. However, each cell is only ever aware or able to reference each cell in its neighbourhood. Each neighbourhood is usually described

as one of two designs: Moore Neighbourhoods (E. W. Weisstein, 2020) and von Neumann Neighbourhoods (V. Terrier, 2002).

The Moore Neighbourhood can be described as a square 3x3 grid with the current cell located in the centre. Whereas the von Neumann Neighbourhood can be described as a cross with the current cell located in the centre and the cells touching all sides of the current cell included (K. Pedersen, 2014)(Figure 2).

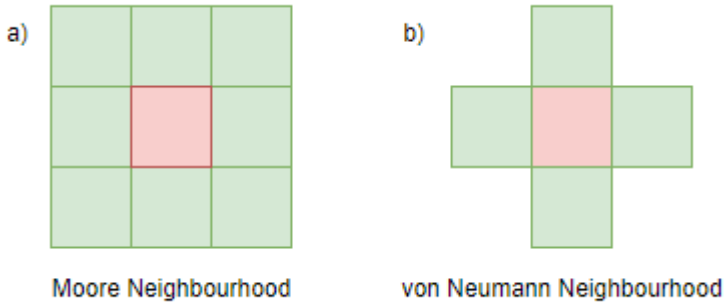


Figure 2: Diagram examples of (a) The Moore Neighbourhood and (b) The von Neumann Neighbourhood

This allows us to produce sets of rules that will alter the state of the current cell depending on the state of each of the cells within its neighbourhood. An example of rules can be derived from Conway's Game of Life.

1. A living cell with fewer than two living neighbours dies.
2. A living cell with two or three living neighbours remains living until the next generation.
3. A living cell with more than three living neighbours dies.
4. A dead cell with exactly three living neighbours becomes a living cell.

By using a similar logic and thus mapping the living/dead cells to some game logic (e.g. Living cells resembling floor or walkable surfaces and dead cells resembling walls/non-walkable surfaces), we can create a grid that begins to resemble a 2D dungeon, however adding more rules to increase complexity may be favourable.

2.1.3 Generative Grammars

Originally used to describe sets of linguistic phrases, (R. van der Linden, R. Lopes, R. Bidarra, 2014) this technique creates phrases through finite selection from a list transformational rules, which include words as terminal symbols. Terminal symbols

are the literal symbols usually displayed as ASCII characters used within the production rules. The original technique has been adapted for use with additional terminal symbols to allow for the generation of shapes and graphs. It is the Graph Grammar technique that can produce a similar solution to our problem by generating topological descriptions of levels in the form of a graph. Nodes within this graph represent the rooms however all other geometric details such as room size and material, are excluded from this technique.

Here we can denote a Grammar with G. Each element G is a replacing rule that replaces a single symbol v with another symbol u or another string A consisting of several symbols during a string rewriting process.

Example: G = { A > AB,B > b }
In a single rewriting process: AB > ABb

In the example above, when writing starts, we replace *A* with string *AB* following the first rule of *G*. At the same time, the second symbol on the original string *AB* named *B* is rewritten as *b*. The new string after the transformation is *ABb*. (J, Suh, H. Zhang, Z. Wang, 2018). As we will look at in the next section, we can think of Grammars as a prerequisite to L-Systems.

2.2 Lindenmayer Systems

Lindenmayer systems or L-Systems were conceived as a mathematical theory of plant growth and were introduced by biologist Aristid Lindenmayer in 1968 as a framework for studying the development of simple multicellular organisms (A. Lindenmayer, P. Prusinkiewicz, 1990). Originally, the concept focused less on modelling plants or fractal patterns due to a lack of detail and concentrated more on the neighbourhood relations between cells and their topology.

2.2.1 DOL-Systems

The simplest form of L-System is one that is deterministic and context-free. These systems are referred to as DOL-Systems and are comprised of an ‘axiom’ consisting of a simple string of which each letter within the string is assigned a rewriting rule, this allows us to create a string replacer that is easily implemented within other

algorithms and software. For example, if we created two rules where the letter *a* becomes *ab* and *b* becomes *a,* the following example can be derived. (Figure 3)

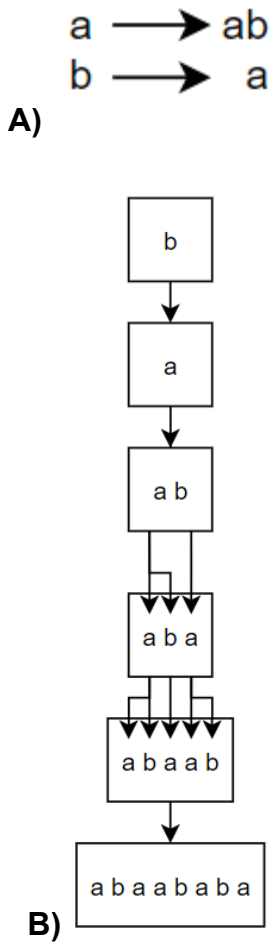


Figure 3: (a) Replacement rules for a & b. (b) Diagram iteration results using rules displayed in (a)

2.2.2 Interpretation of Strings

Now that we understand how we can generate a string of characters from the systems axiom and rules, we can look at how to derive geometry from that string. Referenced as turtle interpretation by Lindenmayer, the basic idea is as follows; A state of the 'turtle' is defined as a quaternion (*x, y, z, w*) where the Cartesian coordinates (*x, y, z*) represent the turtle's position and the angle w, referred to as the heading, is interpreted as the direction in which the turtle is facing. Given the step size *d* (how far the turtle can move over a single time step) and the angle increment *theta* the turtle can respond to commands attached to each character of the generated string. For example, given the string 'F+F+F+F+' and the following rules.

‘F’: Move straight forward of length d (1 unit). The next state of the turtle changes to (x’, y’, z’, w). A line segment between points (x, y, z) and (x’, y’, z’) are drawn.

‘+‘: Turn right by angle theta (90). The next state of the turtle is (x, y, z, w + theta)

We can give the ‘turtle’ instructions to walk forward one space and turn 90 degrees to the right for each of ‘F+’ pairs contained in the above string. Plotting the route travelled will give us a square. This is a rather simplified example as it does not include the next time step with any string replacement. (Figure 4)

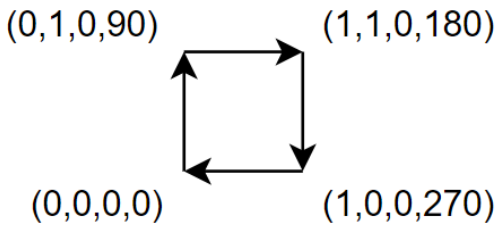
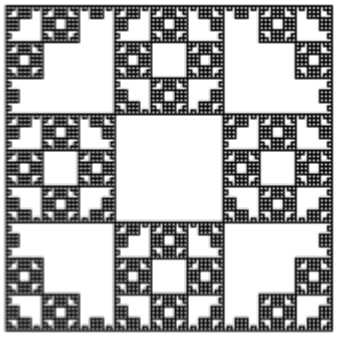


Figure 4: Diagram of turtle path using rules above.

From now on we can define an L-System as containing; the value *n* which represents the number of iterations of timesteps to loop over the string replacement, the value theta which represents the angle to be incremented at each time step, the axiom which represents the starting string value and the rules for each of the axiom values. For example, if our axiom is F then F_r will represent our rule. This now makes describing the instructions for rendering an object like the following fractal (A. Lindenmayer, 1990) (Figure 5).



n = 4, theta = 90,

Axiom: F-F-F-F

F_r: FF-F-F-F-FF

Figure 5: Fractal created over 4 iterations using the Axiom and Replacement Rule provided

2.2.3 Edge and Node Rewriting

The L-Systems discussed so far can be extended by both edge and node writing. Edge writing is the process of substituting the lines drawn by the path described the systems rules, with pairs of lines forming left or right turns. This allows us to create curves and corridors with our fractal images. These curves can be described as FASS curves (space-filled, self-avoiding, simple, and self-similar).

Node rewriting is the process of substituting new polygons for nodes of the predecessor curve. To make this possible, the interpretation described in **2.2.2** is extended by symbols representing arbitrary subfigures. Each subfigure *A* from a set of subfigures *A* is represented by the following.

- Two contact points, entry point P_A and the exit point Q_A
- Two direction vectors, entry vector p_A and the exit vector q_A.

During interpretation of a string x, a symbol *A* incorporates the corresponding subfigure into a picture. *A* represents a node that is placed and rotated to align its entry point P_A and direction p_A with current position and orientation of the interpretation. (M. Bauderon, H. Jacquet, 1999)

2.2.4 Branching Structures

So far, we have discussed how fractals are created using L-Systems and to interpret a string into a sequence of line segments. However, given the biological and botany history of L-Systems, incorporating branching into the systems will allow us to create more natural structures as opposed to the fractals discussed previously. Our branching systems can be referred to as Bracketed OL-Systems. To carry out this method we must introduce two new symbols for use in our strings.

[Push the current state of the point onto a pushdown stack. The information saved on the stack contains the points position, orientation, and possibly other attributes such as the colour and width of lines being drawn.

] Pop a state from the stack and make it the current state of the point. No line is drawn, although in general the position of the point changes.

An example of an axial tree and its string representation including the new symbols can be viewed in (Figure 6). Applying this method allows for the creation of bush-like expanding structures.

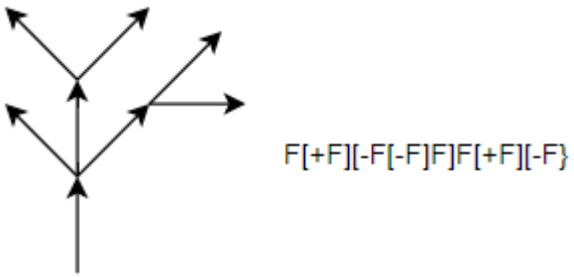


Figure 6: Example of Bracketed OL-Systems

The bracket symbols are probably the most useful we will look at however, we should look at a couple of additional symbols that could be useful in regards to the problem of the project.

!	Used to decrement the diameter or size of the current segment. This allows for the narrowing of branches or corridors further into the structure.
‘	Used to increment the current index of colour or texture.
&	Used to pitch the segment up.
^	Used to pitch the segment down.
\\	Used to roll the line segment clockwise.
/	Used to roll the line segment counter-clockwise.
|	Turn point 180 degrees

Combining these symbols allows to create incredibly complex structures like those portraited by Lindenmayer in his botany work (Figure 7).

n = 7, theta = 22.5

w : A

p_1 : A > [&FL!A]/ / / / / ‘ [&FL!A] / / / / / / / ‘ [&FL!A]

p_2 : F > S / / / / / F

p_3 : S > F L

p_4 : L > [‘ ‘ ‘ ^ ^ {-f+f+f- | -f+f+f+f}]

Figure 7: Example of three-dimensional bush-like structure (A. Lindenmayer, 1990).

2.3 Combining L-Systems and Procedural Generation of Dungeons

Having looked at both procedural generation techniques and L-Systems separately, we can now look at how those concepts can be combined.

2.3.1 Generating Rooms & Geometry

Properties of L-Systems used for room shape generation are set according to two factors: the type of space (2D/3D) and the size of a single unit of space within your game (maximum size possible for rooms and usually recorded in pixels). In that aspect we can distinguish the following elements (I. Antoniuk, P. Hoser, D. Strzeciwilk, 2019).

- The Number of alternative exchange rules for L-System keys (a single set for each symbol existing in within the axiom).
- Starting length of Axiom. This allows us to change the minimum size of the room.
- The number of iterations. We define the complexity of our room only by limiting the number of iterations.

Since we have defined our tiles size, we should also define our tiles shape and how each tile can connect to others (i.e. where corridors can be located). This will also allow us to turn off connectors should a room not require more than a specified amount (Figure 8).

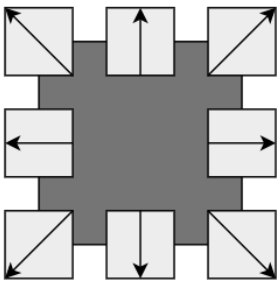


Figure 8: Example of pre-determined corridor connectors.

Using randomly generated key-sets containing production rules with different initial keys (e.g. ET representing the instruction to extend from the top corridor or EBL

representing the instruction to extend from the bottom left corridor). Symmetry of rooms can be enforced during this process if required by calling the opposing instruction (e.g. when calling the instruction to extend from the top, also calling the instruction to extend from the bottom).

2.3.2 Populating Rooms

Not only can we use L-Systems to produce the dungeon layout, but we can also use them to procedurally populate the rooms within the dungeon. We can start by defining types of rooms using symbols for our L-System grammars. The following serves as an example:

- S: The room where the player starts.
- C: A room where the player encounters a combat challenge or some other challenge.
- P: A room the contains a form of puzzle the player can complete.
- L: A room containing some form of loot or collectable for the player.
- B: A room containing a boss for the player to face.

Initially we can set our axiom or starting room as S, so that the player always has a room to start in and followed by any number of other non-terminal symbols. We can also set our terminal symbol (last symbol) as B so that there is a final boss for each dungeon. Assuming that we are generating the geometry of the dungeon first, the system used for populating must be constrained to only contain n number of room types, where n is the total room size of our geometry.

As for generating/placing visual elements within the rooms themselves after the room types have been set, it could be more viable to combine our approach with another approach mentioned in **2.1**.

3 Chapter 3 | Methodology & Implementation

This chapter contains a detailed description and outline of the implementation process in creating an application that uses L-Systems to generate randomised dungeons.

3.1 Programming Environment

The main application was created and developed using version 2019.4.10f1 of the Unity cross-platform game engine. Although the project produced is a 2D application, the initial project was created using a basic empty 3D template provided by the Unity editor and was simply converted to run the application in a 2D space. The Unity engine was chosen due to the authors familiarity with the development environment and its useful debugging tools and existing libraries. Alongside Unity, all coding and scripting was written using the 2019 Community version of Visual Studio (Microsoft, 2019). Visual Studio was chosen as the IDE again due to the authors familiarity with the application, but also because of its wide amount of documentation and debugging tools. C# was chosen as the language of choice for all code to be written in (Microsoft, 2020). While Unity does cater for the use of other coding languages such as Java, Python or Lua, C# was chosen due to the authors existing knowledge of the language, the extensive amount of documentation and resources already available, and that C# is natively supported by Unity and therefore requires no plugins or third-party support. Source control was also used throughout the project to ensure that the project had a backup and that any state or mistake during production, could be rolled back if required. For this, a Github repository was created and updates were made in regular updates using the Github desktop application (Microsoft, 2020).

3.2 Unity Terminology

In this chapter, a lot of Unity specific terminology is used, therefore section is included to explain the terminology and can be referred to as required (Unity, 2020). At large, Unity is an *ECS* or *Entity Component System*. An *ECS* is an architectural design pattern that is mostly used in game engines and other aspects of game development. These systems prioritise composition over inheritance, meaning that

classes and objects should achieve polymorphism and code reuse through their composition, other than traditional inheritance from a single base or parent class. This means that objects can fit into two categories, *Entities* (referred to as game objects in Unity) or *components*. Each entity consists of one or more components and can be attached to other entities as either parents or children. A *tag* is a unique id that is used to group similar or separate important entities. These tags can be used to find game objects in data structures or in the Unity editor itself. *Components* are attachable systems or objects that when attached, describe how the game object will interact with the world. Unity contains hundreds of these component objects.

Scripts are components containing written code that can be attached and executed through a game object. *Scripts* can be thought of components that have been created by the developer to achieve a specific goal. Most of the Unity objects use inherit from the *Monobehaviour* class. Monobehaviour is the predefined base class from which every script derives (Unity Software Inc. 2020). This base class contains a wide variety of predefined functions to add to our scripts. The *Awake* function is called once when the script instance is being loaded and can act in a similar way to a constructor or initialisation function. Like the awake function, the *Start* function is called once on the first frame that a script in enabled before any other of its functions are called. There are two update functions, *Update* and *FixedUpdate*. The Update function is called once every frame while the FixedUpdate function is frame-rate independent and is called once every 0.02 seconds. The FixedUpdate function is mostly used for physics calculations due to its independent time step.

The Unity *scene hierarchy* refers to the visual area in the Unity editor and the data structure inside the game engine that contains each of the game objects added to the current scene. A *scene* can be described as an individual level within the application. This application contains a single scene. The *inspector* is a window within the Unity editor that can be used to view the GUI of components attached to an entity.

3.3 Map Representation

While Unity contains a game object implementation of a tile map object, it was decided that during the project, a separate tile map object would be created as the Unity implementation contained components and system that would not be required

for the project and could also affect the overall efficiency of the project, were they to be used. Maps of different dimensions were tested, and the final size used was 140x140 tiles as these dimensions allowed for decent sized maps to be generated and provided a minimum drop in efficiency. Other sizes tested were 100x100, 200x200 and 300x300.

3.3.1 Tile Map Data Structure

A tile map can be described as a two-dimensional grid of shapes. The data structure of a tile map can take many forms to fit the requirements of the application, however most implementations will have a common amount of data that is required to be held.

- **Tile size**: The size of each tile vertically and horizontally, usually recorded in pixels. The most common sizes are 16x16, 32x32 or 64x64 however any other sizes can be used and are usually closely linked to the artistic design of the application.
- **Image Atlas**: A reference to or location of the atlas or sprite sheet (Figure 9).
- **Map Dimensions**: The heigh and width of the map, normally recorded in the number of tiles across / tiles high.
- **Visual Grid**: Can be held in many ways but is usually stored as a 2D array of indices that describe the type of tile held at each location on the map.
- **Position**: The world position of the map, that allows all tile positions to be calculated relative to this information.

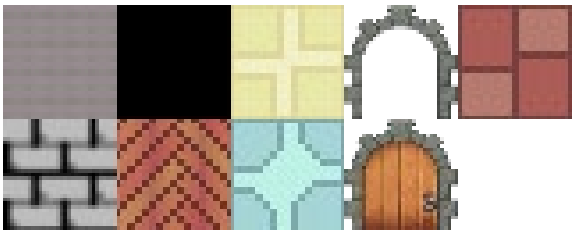


Figure 9: Tile Atlas used in this project

3.3.2 Tile Shape

Many different types of shapes have been used in creating tile maps. The shape chosen for this project was a simple square tile as it provides a level of simplicity, both in terms of artistic style, but also in terms of mathematics when it comes to coding. Another option could have been to create a tile map consisting of hexagonal shapes. A hexagonal tile map might have been more interesting to use with L-

Systems as it allows for easy diagonal movement but would have required additional programming as hexagonal maps are not naturally supported by Unity (Figure 10).

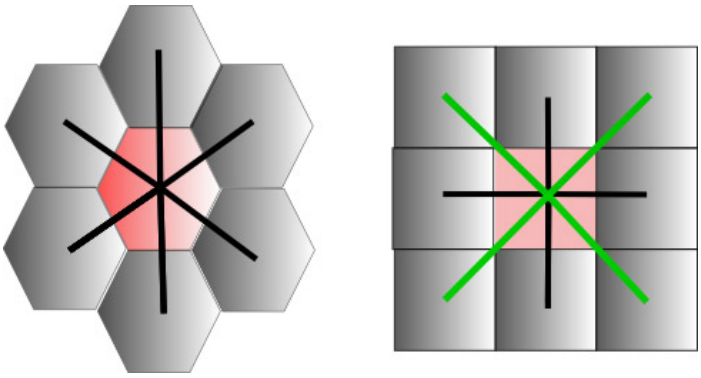


Figure 10: Hex tiles vs. Square tiles considering movement

3.3.3 Map View

The view used to display the tile map in the application was an orthographic plan view. This was chosen due to its simplicity and its ability to clearly display maps and other two-dimensional objects. Other view types were considered during the development of the project, such as Isometric or Oblique.

3.3.3.1 Orthographic

The orthographic view is a purely two-dimensional view and can be thought of as the camera looking down directly above a two-dimensional plane. This creates a purely two-dimensional world, often in which each object is the same size and fills a single tile space, both in terms of tile sprites and objects on top of the tile map. In this view type there is very little sense of depth.

3.3.3.2 Isometric

Also referred to as 2.5D, isometric references a style of view that creates the illusion of a three-dimensional world through camera viewpoint and art style.

3.3.3.3 Oblique

Oblique is a popular tile map view that looks like the orthographic view, however objects on the tile map are usually more than a single tile in height which can provide the illusion of depth for the user.

3.4 Game Objects

This section will discuss the individual object create to ensure that the application contained what was needed to achieve its goals.

3.4.1 Dungeon Generator

The dungeon generator was the first system created and the most important as it is the system that implements the L-System algorithm. This object is described in more detail in section 4.4.

3.4.2 Level

The level game object is the largest system created and the object that interprets the data passed from the dungeon generator to build the dungeon on the tile map. This object and the contents referring to the Level class are described in more detail in section 4.5.

3.4.3 State

The state class is a simple data structure object that holds data and information about the current pass through the L-System algorithm. This allows for the use of branching rules that give the L-System its iconic look. The class contains the variables listed below.

- Axiom: A string variable and holds the axiom at the current ‘state’ of the L-System.
- Position: A Vector3 variable and holds the current position of the ‘turtle’ or ‘cursor’ in the drawing of the L-System.
- Forward: A Vector3 variable and holds the current facing direction of the ‘turtle’ or ‘cursor’ at the current state of the L-System.
- Angle: A float variable that holds the angle of which the next room will be.
- CurrNode: A Node variable that holds reference to the current node held at the point the state was saved.

3.4.4 Node

Like the state class, the node class is a simple data structure object that holds information relating to the nodes created by the dungeon generator and the L-System algorithm. It contains a reference to the rule that created the node as a char variable called type which allows us to determine in the level object what kind of room the node should represent. The height and width of the represented room are also stored as variables respectively. The node object also stores the nodes initial position that is used as the starting point when rooms are generated on the tile map. Lastly, each node object contains a Node variable that holds a reference to the previous node object. This allows for the order of the nodes to be reconstructed in the level object

and is used for deciding the order in which the pathfinding algorithm is run between nodes.

3.4.5 Rule

The Rule class refers to the rules used about the L-System algorithm. The rule class was created to make the L-System algorithm more modular and allow for the algorithm to be easily added and extended upon in the future. Each rule object contains an array for six non-terminal characters and three non-branching characters in two separate string arrays (see section 4.4)

Each rule also contains a variable that holds the character that the rule represents and the replacement character that will be returned from its Replace method. The replace method randomly selects a non-terminal character to act as the replacement added to the axiom in the L-System algorithm. If the terminal character selected is open square bracket (‘[‘) then the Replace method will also decide on a random amount of characters to fill the brackets with from the non-branching array. Finally, the Replace method will also ensure that the final room, represented by the ‘B’ character will never be replaced. Finally, the rule class contains a Boolean variable named IsDrawable and a float variable called angle. Both variables are used by the dungeon generator object to decide in which method the rule should be acted upon and if a new node should be created.

3.4.6 Tile

For the tile object, a tile prefab was created so that each tile could be created equally, this allowed for less memory usage and faster instantiation by essentially creating clones of the initial tile object instead of creating separate objects for the 19000+ tile objects. Each tile game object also contained a Tile script that contained the necessary variables and methods required for the tiles to be used and accessed by the applications other systems.

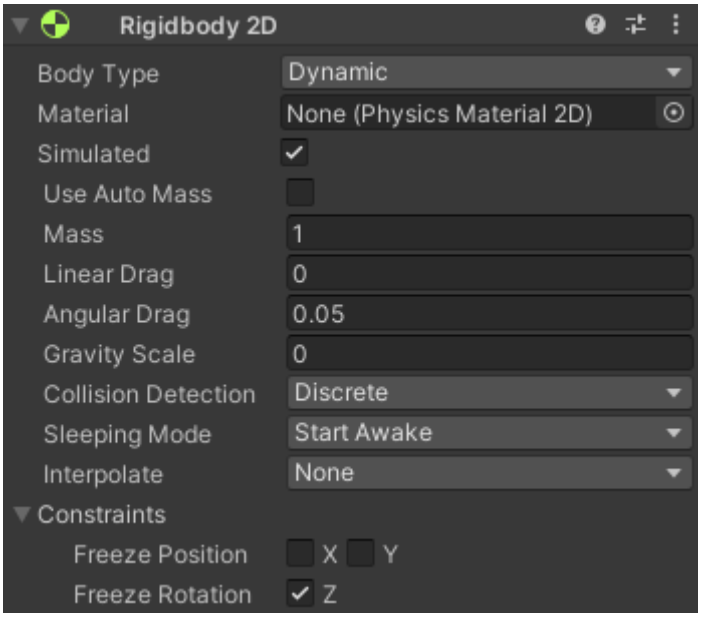
Each tile contained a sprite variable that allowed for the object to hold texture data in Unity’s sprite format. This data was used to visualise the tiles of each of the rooms generated. The single method held within the Tile script is the Setup method that accesses the Tile game objects Sprite Renderer component and sets its Sprite to the one held in the script. This allows for the texture to be changed at runtime, meaning

that maps can be repeatedly generated without restarting the application or changing the scene within the application.

During start-up of the application when the tile objects are instantiated, the tiles are passed a blank texture to act as the default sprite and are passed a position relative to the tiles index in order to generate a grid as mentioned in section 4.3.2. Lastly, each tile object contains a two-dimensional box collider that allows us to detect the if the position of a node is contained within the collider, thus giving us a starting point for the corresponding room.

3.4.7 Player

For the player, a simple player controller script was created so that some interactivity was added for participants during the testing phase of the project. The script made use of Unity's monobehaviour base class so that it would have access to the pre-defined functions provided by monobehaviour. The Start function was used to set up the player object and to set its initial positions to ensure that the object always started in the starting room of the current map. This was also done in the Update function to ensure that when the participant changed the map, the player objects position would be reset to the starting room to ensure that the object was not left in another room or even outside of the map. The update function also checked for any horizontal or vertical input using Unity's pre-defined input class that links the arrow keys or the WASD keys to denote vertical and horizontal movement. The FixedUpdate function was used to calculate the velocity of the rigidbody component attached to the player object. While a rigidbody component was not required, it allowed for collision between the player and wall tiles to be implemented in a convientient way that did not require any extra coding. Turning off the gravity and locking the rotation axis for the rigidbody component allowed for the sprite to act as expected when placed into the generated map. In Unity, a rigidbody component simple allows the game object to act under the effects of Unity's physics engine. (Figure 11).



$$rb.velocity = newVector2(horzontal \times runSpeed, vertical \times runSpeed)$$

Figure 11: Rigidbody component (top) & velocity formula (bottom)

3.4.8 Camera

During the implementation of early prototypes, a camera class was created to make the viewing of each L-System and generated dungeon easier to facilitate, especially for larger maps. The early version of the camera allows for camera movement using the WASD keys and for rotation controlled via the mouse. This was done through the creation of a simple camera script.

In the final application, the camera was changed to mimic that of a two-dimensional fixed camera. This was to ensure test participants could not see the map as a whole and had to manually traverse the map to complete it. The script attached to the camera object at this state used a single monobehaviour FixedUpdate method and contained a single Transform variable called target. The target was a simple reference to the player objects transform location and the single method ensured that the x and y coordinates of the camera's transform position, were set to the targets x and y coordinates at each frame.

3.5 L-System Implementation

This sub-section will describe and layout the steps taken to implement the L-System algorithm researched in chapter two, and the system that is the focus of this project. The implementation of the L-System algorithm is contained within the Dungeon Generator class and the dungeon generator object within the Unity hierarchy in the

final application. As part of the L-System implementation, additional data structure classes were created to assist in the implementation, mainly the State (4.3.3), Node (4.3.4) and Rule (4.3.5) classes.

3.5.1 Classic Implementation

This sub-section will look at the initial implementation of a basic L-System algorithm that was able to create structures like the ones described by Aristid Lindenmayer and the ones researched in section 2.2.

Initially, a test project was created to implement the L-System algorithm without being concerned with tile maps etc. The test application ran from a single class called Rule Builder which contained all logic pertaining to the L-System, as well as data structures for States, Replacement Rules and Draw Rules. To render the L-System's tree like structures, the class used a basic render function to draw vertex lines to the screen, following the L-System path. This was achieved using Unity's OpenGL API.

3.5.1.1 L-System Variables

To implement the L-System, the following variables were created in the Rule Builder class.

- **Iterations**: A integer variable to hold the amount of times the algorithm has passed through the axiom. The higher the iterations, the longer the axiom and more complex the system is generated.
- **Starting Axiom**: A string variable to hold the initial axiom before it is passed through the algorithm.
- **Current Axiom**: A string variable to hold the axiom of the current pass through the algorithm.
- **Forward Scale**: The scale at which the distance between nodes is calculated.
- **Angle**: The angle at which each rule is drawn.
- **States**: A stack containing state data structures. A state is created upon executing certain rules and is pushed onto the stack, holding all the data at the current place in the algorithm.
- **Replacement Rules**: A dictionary that holds all the given rules regarding replacement logic for each of the rules.
- **Draw Rules**: A dictionary that holds all the given rules regarding the draw logic for each rule.

3.5.1.2 Awake Method

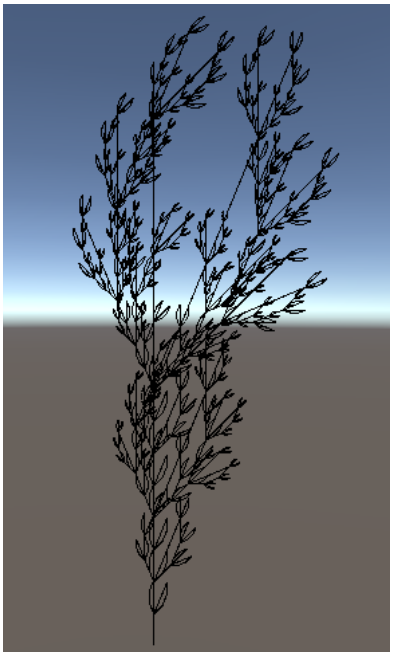
As there was no procedural generation in the basic implementation, all rules were added to the L-System manually inside the Awake method. The rules in the example were used to create the example L-System in the image below.

3.5.1.3 Axiom Generation

The first function used to implement the L-System algorithm is the Generate Axiom function. This function takes the current axiom and loops through the individual characters contained within the axiom. Inside the loop, each character is checked for a replacement rule containing the same character and each character is replaced with its replacement string.

3.5.1.4 Position Calculations

The second function used to implement the L-System algorithm is the Calculate Positions function. The function loops through the axiom in a similar way to the Generate Axiom function, but searches for draw rules containing each character in the axiom. It first checks if the current character is a bracket (‘[‘, ‘]’). If the character is an opening bracket, it creates a new state of all the algorithms current settings and pushes it to the stack. If the character is a closing bracket, the previous state is popped from the stack and is set as the current state in the algorithm. If the character is neither of the brackets, the rule pertaining to the current character is checked for an angle. If the angle equals zero, the function calculates the positional data of the next point in the structure and adds it to an array holding the positions. If the angle is not equal to zero the function then calculates the angle of the next line, using Unity’s quaternion class to calculate the angle and its axis (Figure 12).



$$F{:}FF-[-F+F+F]+[+F-F-F]$$
$$X{:}F[+X]F[-X]+X$$
$$I=5$$

Figure 12: Example of L-System generated using basic implementation.

3.5.2 Dungeon Implementation

This sub section will look at the implementation and development of the L-System from the previous section and how it was adapted to fit the requirements of the dungeon generation system. In the final application, a class named Dungeon Generator was created to implement the L-System algorithm. The class contained many similarities to the one described in 4.4.1 but was altered to make the system more modular and easier to edit. All visual elements, such as the OpenGL functions were removed as this was handled as part of a separate script.

3.5.2.1 Variable Changes

In the final version, separate rule objects were created that combined the replacement and draw rules into a single object. This meant that only a single dictionary structure was needed to keep track of the rules. The dictionary used each rule unique character to track and reference each rule. The stack data structure that contained the states of the L-System remained the same, however a new separate list was added to store the generated nodes. The only other additional variables were constant integers to store the maximum and minimum possible dimensions for the nodes, as this would be generated randomly.

3.5.2.2 Rule Setup

As the rules for the dungeon generation were to stay the same, a separate function was used to contain all the logic pertaining to the creation of rules, rather than doing so in the awake function. In addition, this function also created the initial state of the L-System and added it to the stack.

The rules used in the application are outlined below.

- S: This rule represents the starting room of the dungeon. Only a single S character may exist in an axiom at a time (normally at the beginning).
- B: This rule represents the end (Boss) room of the dungeon. Only a single B character may exist in an axiom at a time (normally at the beginning).
- C: This rule represents a challenge room inside the dungeon.
- L: This rule represents a loot room inside the dungeon.
- P: This rule represents a puzzle room inside the dungeon.
- +: This rule represents a rotation to the right for the placement of the next room. The angle at which the placement is rotated, is set in degrees, and is randomised upon the rule's creation, with a range of between 0 and 180.
- -: This rule represents a rotation to the left for the placement of the next room. The angle at which the placement is rotated, is set in degrees, and is randomised upon the rule's creation, with a range of between -180 and 0.
- [: This rule represents a branching path from the current room and is used to denote when a new state should be saved to the stack.
-]: This rule represents the end of a branching path and is used to denote when to return to the previous state.

Only rules referenced by a letter, instead of a symbol, were declared as drawable rules.

3.5.2.3 Updating Axiom

The Generate Axiom method described in 4.5.1.3 carried over the same loop but additional constraints were added to the function to deal the procedural nature of the replacements. If statements were put in place to ensure that every axiom generated contained a single instance of the S and B rules as well as checks to ensure that when adding '[' rules for branching, the brackets were paired to avoid any stack conflicts.

3.5.2.4 Updating Positions

Much like the axiom function, the function to calculate positions remained much the same and only had two additional sections added to it. The main feature that was added was to create a new node and add it to the list whenever the current character was found to be drawable. The node creation included calculating the position to line each node with the tile map, randomly generate the nodes width, height, and type, and set the nodes parent. The other sections added were a call to the Levels setup method and the Check Node Overlap method described in 4.7.

3.5.2.5 Controls

Controls were added to the update method to allow for generation of maps one after another in runtime. Each time a press was detected on either the left or right arrow keys, the function would perform a reset of all the data structures then decrease or increase the iteration count and run the algorithm again by calling the Generate and Calculate functions.

3.6 Level Setup

The level class is the object that links the dungeon generator and L-System to the Unity editor and the visual elements. This section will look at the main set up of the level object and the tile map in relation to the L-System. Inside the awake function a nested for loop was set up to create the tile map by initialising each tile, including their position in world space, and creating them in the Unity hierarchy. It also adds each tile to a two-dimensional array of game objects to store each tile entity created in the Unity editor.

$$xPos = x - (width / 2) - 0.5$$
$$yPos = y - (height / 2) - 0.5$$

The setup function was created to combine all other tile relevant functions and was used to reset all tiles prior to a new map being generated. The setup function takes the list of nodes created in the dungeon generator and uses the information stored in each node to determine each tile's settings. It does this by looping through each tile held within the array and then looping through each node. During the node loop, the algorithm checks to see if the tiles collider overlaps with the position held in the node. Upon finding a tile that overlaps with a nodes position, the algorithm will add a reference of that tile to a separate list that will be used as starting points for the path finding algorithm. After doing so, the algorithm will loop through the tiles in the vicinity of the current tile using the height and width attached to the current node object to

set up the rooms by changing the sprites of the surrounding tiles. To do this, the algorithm uses a switch to distinguish which sprite should be used by taking the current nodes type variable as the switch condition. The tile sprites are set by using Unity's component system and accessing the tile script component to access its local sprite variable (Figure 13).

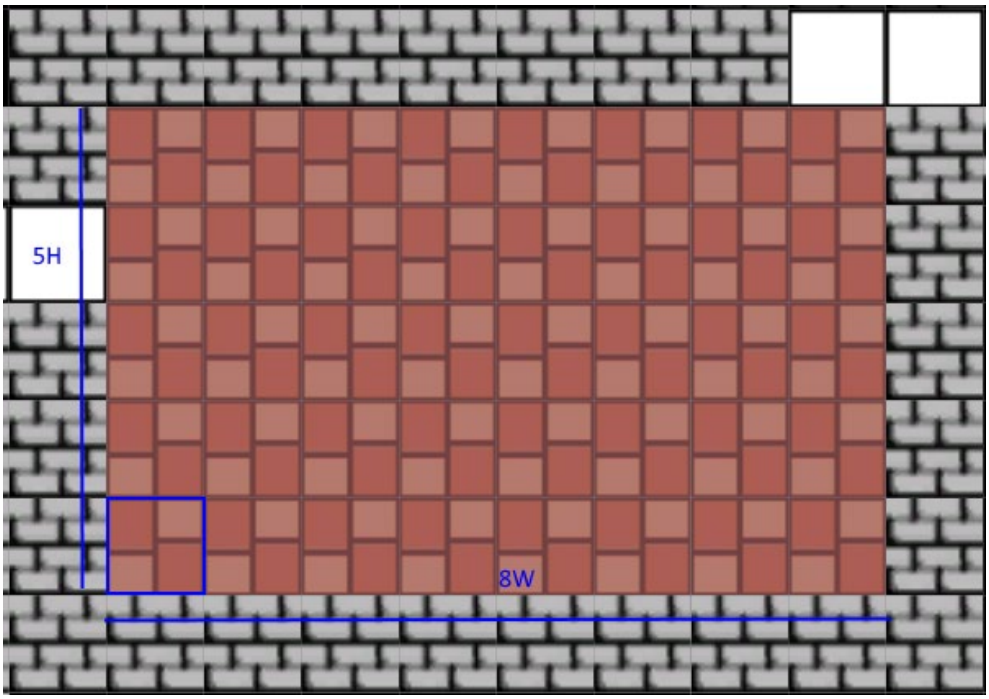


Figure 13: Room drawn from node height and width

3.7 Overlap Algorithm

During the initial generation and testing of the map generation, some rooms would generate in an overlapping manor, depending on their size and position. While this led to some interesting room combinations, it was not what the desired end goal of the map generation was, and it was felt that having distinctly separate rooms separated by corridors was more interesting, both from a visual and from a gameplay perspective. To ensure that rooms remained as separate entities and did not overlap, a method called CheckNodeOverlap was created to ensure that the nodes created at the L-System generation stage were evenly spread out before creating rooms on the tile map (Figure 14).

$$nodeI.position = Mathf.Max(0, node.position - node.width)$$
$$nodeJ.position = Mathf.Min(140 - nodeJ.width - 1, node.position + 1)$$

Figure 14: CheckNodeOverlap formula

The algorithm loops over each node and compares that node to each other in the list. For each node, the algorithm checks whether the node would overlap with the other by considering the nodes set position, width, and height. If it is decided that two nodes would overlap, the algorithm sets the x and y positions of each of the nodes by taking into consideration the total width and height of the tile map, as well as the positions and dimensions of each of the nodes. If a nodes position is changed in this way, the redo variable is set to true to ensure that the algorithm runs again to ensure that any movement done, has not caused another overlap. Only when no overlaps are detected, or the loop times out, does the algorithm complete. To check for overlaps, a separate method called IsOverlapping was created to be used in conjunction with the above algorithm. This method is the one that checks for overlaps between two nodes and is called during ever iteration of the main algorithm (Figure 15).

$$Vector2.Dist(n.pos, j.pos) < n.width + j.width + 1$$
$$Vector2.Dist(n.pos, j.pos) < n.height + j.height + 1$$

Figure 15: IsOverlapping formula, n refers to the current node while j refers to the next node being checked.

An extreme example of a simple map generated without implementing the CheckNodeOverlap method can be found below (Figure 16).

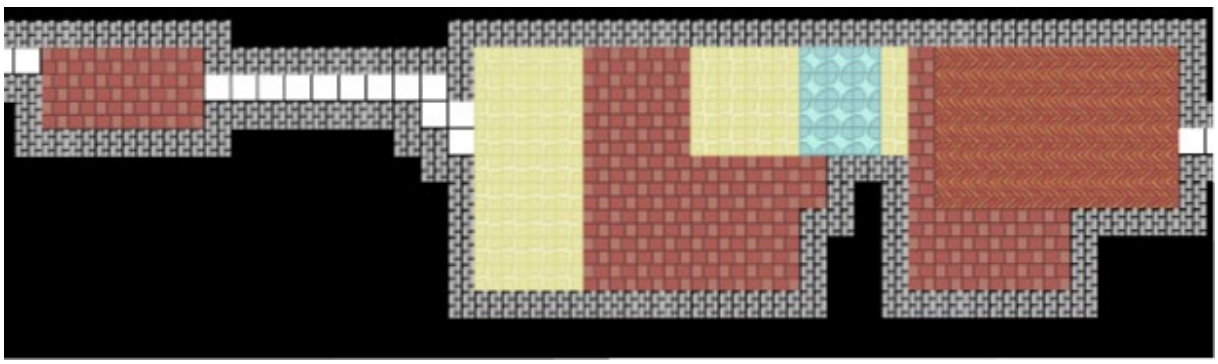


Figure 16: Map generated with no CheckNodeOverlap Method

3.8 Pathfinding

To connect the rooms generated from the L-System algorithms nodes, it was decided that using a pathfinding algorithm to create pathways between each of the rooms in the order that they appear in the L-Systems axiom, would be the most efficient way to

do so procedurally. The algorithm implemented was A* and was tested in a few different iterations throughout development. To fit with the A* algorithm, each tile object was given three variables: gScore, hScore and fScore. In accordance with the traditional A* algorithm (P. E. Hart, N. J. Nilsson, B.Raphael, 1968), the gScore variable was used to contain the cost of the path up to the point of that tile, whereas the hScore variable was used to contain the heuristic cost between the current tile and the desired end tile. The fScore variable was used to contain the sum of both the gScore and fScore and was the variable in which the algorithm used to decide which tile to inspect next.

$$i.fScore = i.gScore + i.hScore$$

The heuristic function used to determine the hScore of each tile was to calculate the distance between the current tile and the desired end tile. As this was a simple pathfinding algorithm to generate corridors in the L-System dungeon, no additional values for difficulty or tile type were added as the algorithm simply needed to find the most efficient route. During the implementation phase, two different heuristic functions were tested, Euclidean and Manhattan. The main function of the algorithm, other than to set and manage the gScore and hScore, is to loop through and check tiles using each tiles fScore to decide which tile is best to check next. Once a tile is checked, the algorithm will look at each neighbour of that tile and calculate the values for their scores while adding them to a list of potential candidates to check next. The algorithm will look for the candidate in the list with the lowest fScore and select that tile to be checked next. This will happen until the end tile has been found, or until all tiles have been checked (meaning that the end tile most likely does not exist). Each tile was also given a parent variable. This variable was used to store the previous tile and was used to reconstruct the path by backtracking through the parent values from the end tile to the start tile. This was run through the Reconstruct method. The Reconstruct method used a while loop to run until the original starting tile had been reached and collated each of the parent tiles indexes as point objects before adding each point to a list. The array of points returned from this method could then be used to identify each of the tiles within the original two-dimensional tile array. From the reconstructed path, a simple loop over each of the points contained within the list

could be used to access each of the corresponding tiles and change their sprite and ensure they were set up as walkable tiles.

3.8.1 Euclidean Distance

Euclidean distance was the initial formula used for calculating the heuristic value for each tile. This method worked but produced corridors thar were uncharacteristically diagonal and while they fitted well with the aesthetic of biology and the L-Systems traditional use, they did not fit well in terms of video game use and it was felt that the consistently diagonal corridors that the function produced would be found irritating by any potential user. This led to the function being swapped for an alternative. The Euclidean distance was calculated by using Unity's Distance method attached to the Vector2 object. The distance method returns a floating-point value which is calculated by applying the Euclidean distance formula (Figure 32).

$$Vector2.Distance(a,b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$$

3.8.2 Manhattan Distance

To produce the desired aesthetic look of the map's corridors, the Euclidean formula was replaced with the formula for the Manhattan distance. This allowed for the corridors to be right angled and while still adhering to the natural structure of the L-System. The Manhattan distance can be described as the distance between two points, measured along axes at right angles. The Manhattan distance also returns a floating-point value and is calculated by applying the following formula (Figure 34).

$$Vector2.Distance(a,b) = |a.x - b.x| + |a.y - b.y|$$

The Manhattan formula calculates the difference between each axis value and returns the sum of the absolute values of each axis, giving the total distance.

3.8.3 Euclidean Vs. Manhattan

While both distance calculations worked and provided maps that were completable, the below image shows the aesthetic difference between both versions when used to generate a map in the final application. The Euclidean distance produces jagged yet shorter pathways while the Manhattan produces longer but neater ones. While both

are equally usable, given the trend of four directional travel among games that may could make use of maps like the ones generated by this application, the neater more right angled corridors were felt to be better and more pleasing for the user (Figure 17).

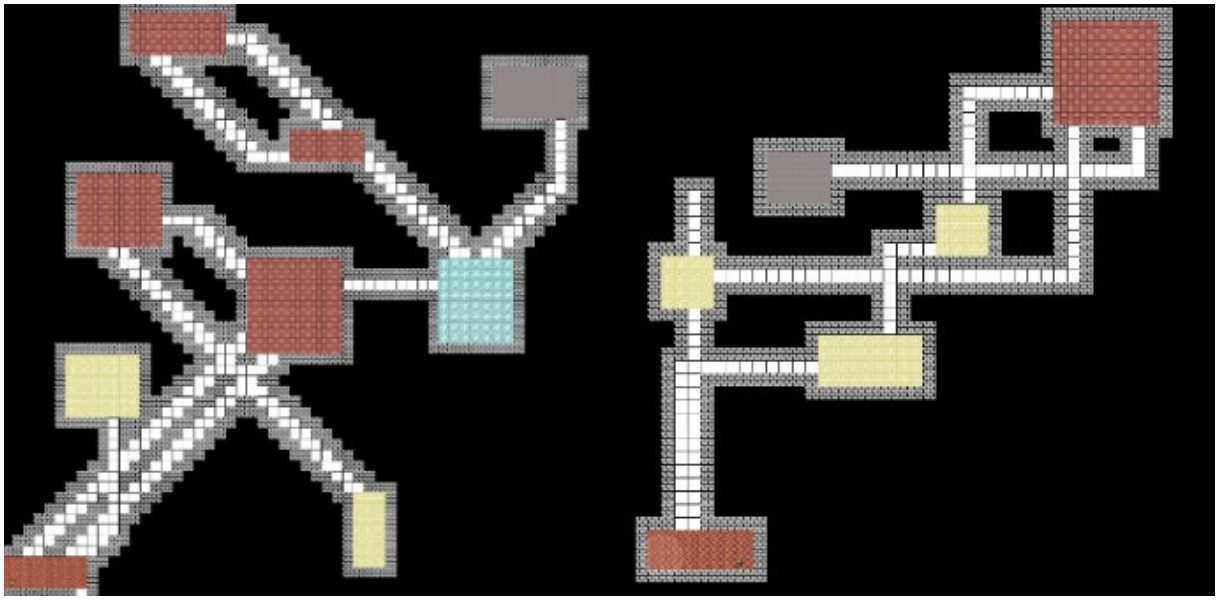


Figure 17: Comparison of Euclidean corridors (left) and Manhattan corridors (right)

3.9 Wall Generation

Due to the procedural nature of the application, it was decided to automate the approach to generating the walls around each of the rooms and corridors. To achieve this, a method called CalculateWalls was created. This method was called after the rooms and corridors were generated. The algorithm loops through the two-dimensional array of tile objects and checks if the current tile's initial sprite is not a floor sprite and also checks each of the tiles neighbours to see if any of which are considered to be floor tiles set by either room or corridor generation. If those conditions are true, the algorithm will set the tiles sprite to be a wall and will call the tiles setup method.

3.10 Door Generation

During the implementation phase, the idea of populating each dungeon with objects was looked at. While the generation of doors for each of the rooms is not active in the final application, the feature remains there. Door generation was done using a CalculateDoors method. Before generating, the method would gather any existing door objects and delete them to ensure no objects were left after generating a new map. The main part of the function would loop through the two-dimensional array of

tiles and in a similar fashion to the wall generation, check the surrounding condition of the current tile. If none of the surrounding tiles are blank, the current tile is a corridor tile and one of the adjacent tiles has a sprite belonging to the room tiles, then a door will be instantiated on top of the current tile.

4 Chapter 5 | Results

This chapter contains a detailed overview of the steps take to evaluate the application and a description of the further analysis of data gained from that evaluation.

4.1 Evaluation Strategy

Due to the application's procedural nature, it was important to investigate and seek feedback on the maps that the application could create. This would help provide insight into the reliability and quality of the maps generated from the application and its algorithms. To achieve this, testing packs containing a build of the application and a test survey were distributed.

4.1.1 Participant Test

Participants were given a testing kit that included a copy of the survey, a stand-alone build of the application and a consent form. Each participant was tasked with running through five different dungeons, starting from the start room, and ending in the boss room using the player character object. The camera used in the final build was zoomed in so that the participants could not see the entire dungeon layout at once. Each dungeon generated increased in iterations from 1 – 5 and participants were asked to answer the survey for a single dungeon at a time. Each participant was given the same instructions and were also asked to write down the axiom for each dungeon printed on screen.

4.1.2 Likert Scale

While designing the technique of acquiring feedback, it was felt that because the feedback given could be effected by the procedural nature of the test, and that the test could be effected by ones previous experience with similar systems, the survey should contain a Likert scale. The Likert scale is a five (or seven) point scale that is used to allow individuals to express how much they agree or disagree with a particular statement. The table below shows the scale used within the survey (Table 1).

Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
1	2	3	4	5

Table 1: Likert Scale

The Likert scale assumes that the strength/intensity of an attitude is linear and by limiting the responses, can allow for the data gathered to be clearly displayed. Likert scales have the advantage that they do not expect a basic yes or no answer from the participants but also allow for degrees of opinion and even no opinion at all. Therefore, the data obtained from the scale can be considered quantitative data (4.1.2). The Likert scale allows for responses to be anonymous which was important for this study to adhere to GDPR practices even though the data would not be considered sensitive. However, there are limitations to using a Likert scale. One limitation is that the validity of the attitude measurement can be compromised due to social desirability. This means that individuals may lie and record answers to either put themselves in a positive standing, or more likely in terms of this study, produce answers they think are positive or desired by the direction of the study (Likert Scale, 2019).

4.1.3 Quantitative Data

Quantitative data is described as statistical and is typically more structures than qualitative data. This means it is more rigid and defined. This data type is measured using numbers and values, making it a more suitable candidate for data analysis. The data takes the form of counts or numbers where each data set has an unique numerical value associated with it, like those that can be seen in a Likert scale, and can allow us to view not only the average answer but also how many participants answered each level of a statement. Quantitative data has the advantage of obtaining accurate results as the results obtained are typically objective. Another advantage of this data type is that it greatly reduces personal and research bias as the data is always clearly defined. The major limitation of this data type is that the data is not necessarily descriptive and can become difficult for researchers to make decisions based solely on the collected information.

4.1.4 Survey Statements

It was important that the statements given in the survey represented the goals of the overall project. This would ensure that participants payed attention to the correct area of the application and should make it clear what area of the application they were testing. The statements given are shown in the table below (Table 2). Participants were also given an extra box on each section of the survey to record the axiom, iteration count and to give any additional comments they had for each of the dungeons that they tested.

1	I find that the dungeon layout is confusing.
2	I find that there are too many corridors
3	I find the rooms to be of adequate scale.
4	I find the rooms are spaced out well.
5	I find that there is a good choice of exploration.
6	I find that the dungeon has a natural feel.

Table 2: Questionnaire Statements

4.1.5 Statement Results

This sub-section will look at and outline the results gained from each statement gathered from the returned surveys. Each section will look at a question asked to participants from the table above (Table 2).

4.1.5.1 Confusing Dungeon Layout

Statement one was designed to ask participants how confusing they found the dungeon layout over each map iteration. This statement allows us to understand whether the dungeons created may be viable for use outside of this study. It could be desirable in some games that the map be confusing or difficult but for most it is the contents and layout that could be deemed most important. Over the 20 responses received, we can see from the data shown in Figure 18, that there is a slight increase in responses that agree with statement which might suggest that dungeons with higher iterations are viewed as being more confusing or complex to traverse.

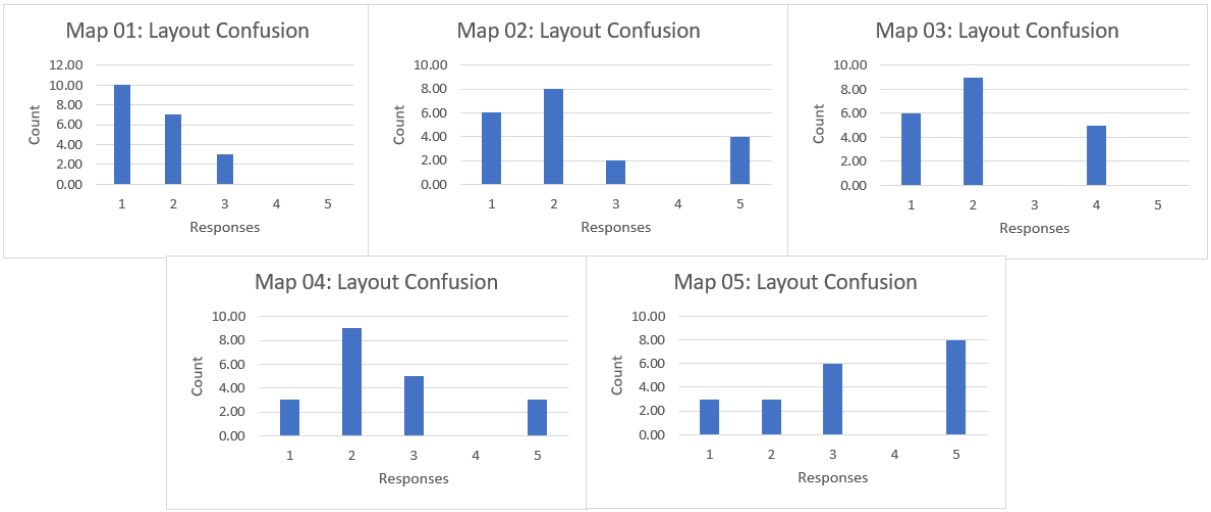


Figure 18: Counted Data from Statement One over all iterations

We can see this rise in responses again in Figure 19 by looking at the average level of responses given over the five iterations. However, the average only increases by 1.7 over all five iterations suggesting that may suggest the effects cause are minimal.

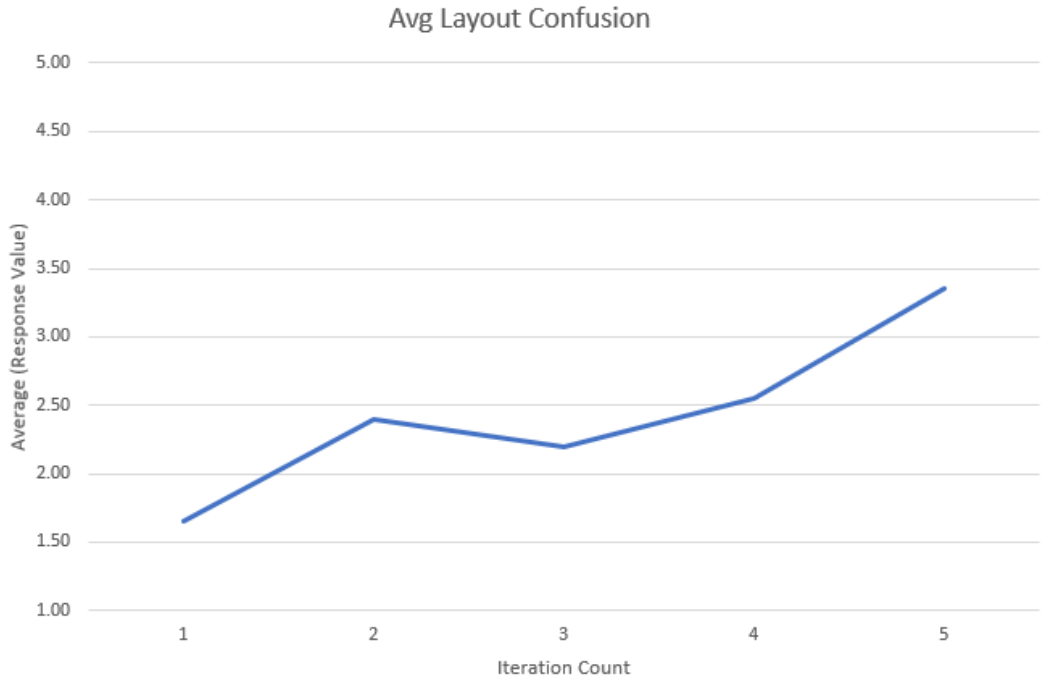


Figure 19: Average results over all five iterations from statement one

4.1.5.2 Corridor Amount

Statement two was designed to ascertain whether participants were happy with the number of corridors contained within each dungeon iteration. This statement allows us to understand if the use of the pathfinding algorithm with the L-System layout

works and if the combination could be a viable approach outside of this project. Shown in Figure 20 is the data collected from the responses regarding statement two. The data doesn't seem to hold much of a pattern, however as seen in Figure 21, the larger dungeons with a higher iteration count on average returned a lower level of satisfaction. Given that the lower iterations have a higher level of satisfaction, it would suggest that reducing the number of corridors and having some rooms join via single square doorways to reduce the number of corridors may be preferable for larger instances.

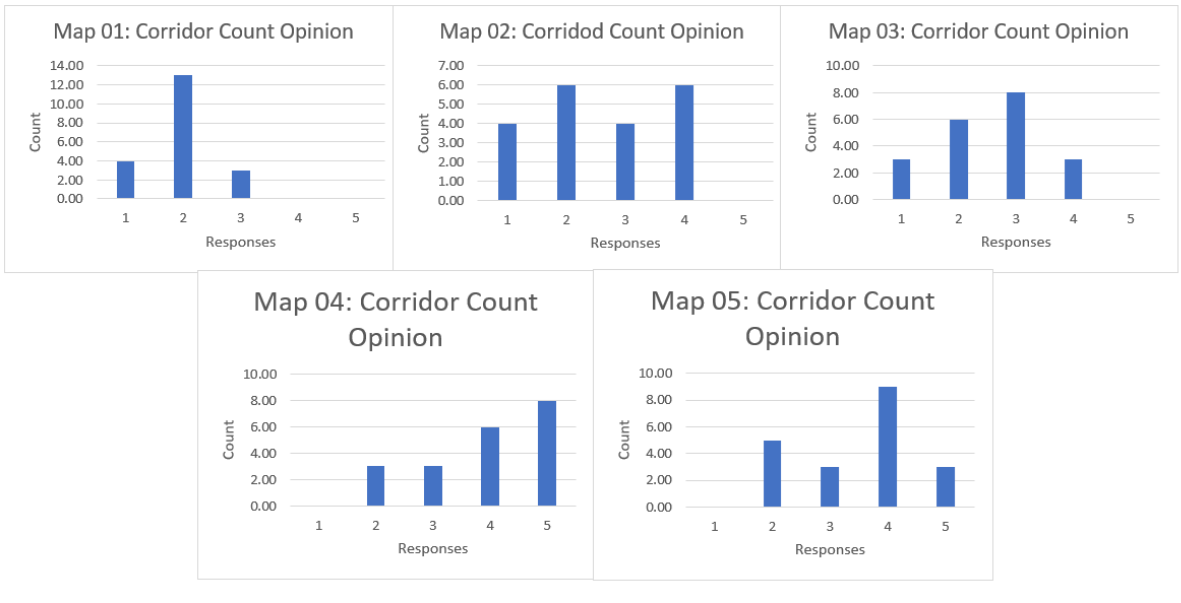


Figure 20: Counted Data over all iterations for statement two

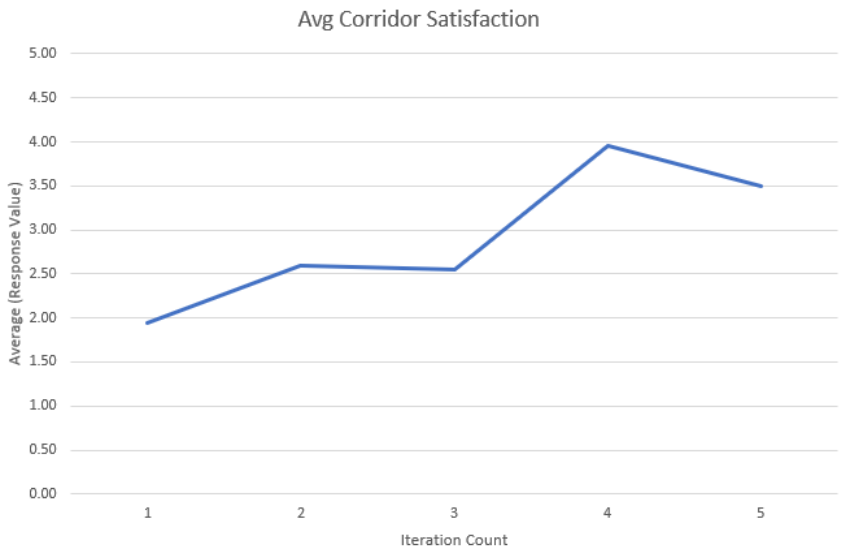


Figure 21: Average results over all five iterations from statement two

4.1.5.3 Room Size

Statement three was designed to ascertain whether participants were happy with size and scale of each room within each of the dungeon iterations. This statement allows us to understand how well the room sizes were randomly decided in node creation. Out of all of the statements covered in the survey, as we can see across both Figure 22 and Figure 23, we can neither draw a positive or negative opinion from the data. The responses gathered from the statement are split almost evenly across most iterations. Positively, this may be down to the room scales being of adequate size, but more likely is down to the data set and would require more testing to draw any conclusion from this statement.

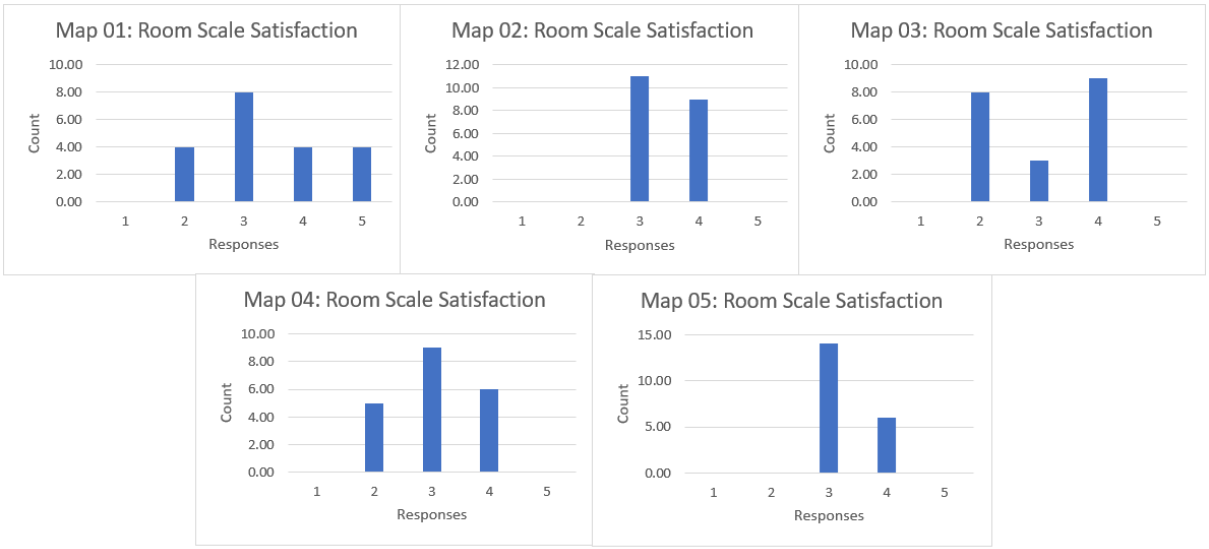


Figure 22: Counted data over all iterations from statement three

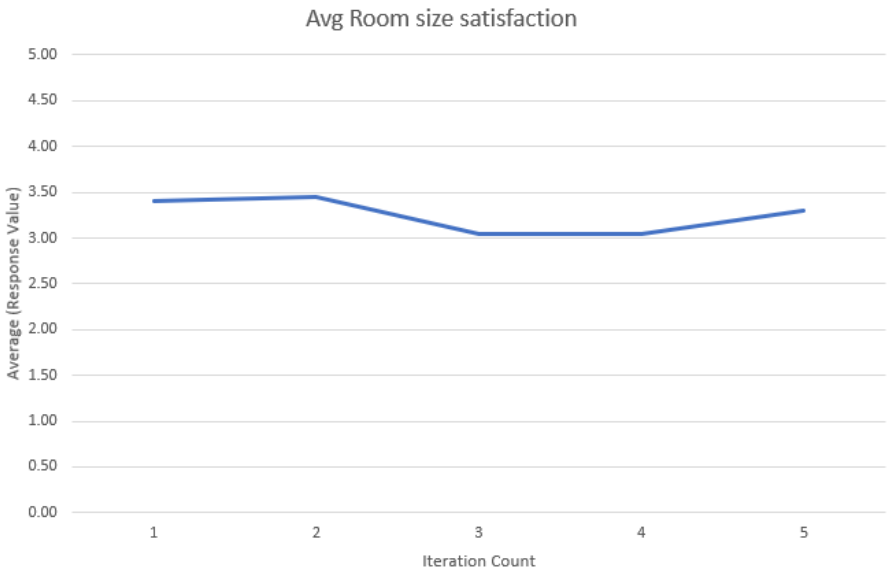


Figure 23: Average results over all five iterations from statement three

4.1.5.4 Room Layout

Statement four was designed to ascertain how happy participants were with the how well the rooms within each dungeon iteration were spaced out. The results from this statement can allow us to decide if the L-System positioning and the overlap algorithm working in tandem produces results that are desirable for this project. Looking at the overall responses given in Figure 24 we can see that the majority of participants agreed with the statement and were overall happy with the layout. We can also see from the average results in Figure 25 that the satisfaction generally increases with the overall increase in the number of iterations use to create the dungeon map.

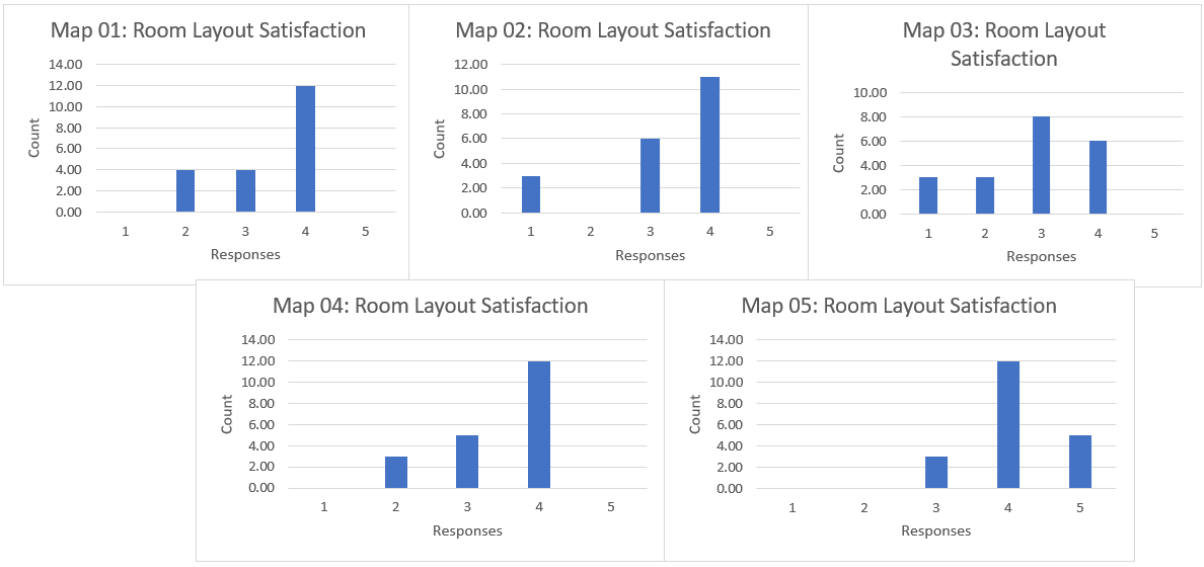


Figure 24: Counted data over all iterations from statement four

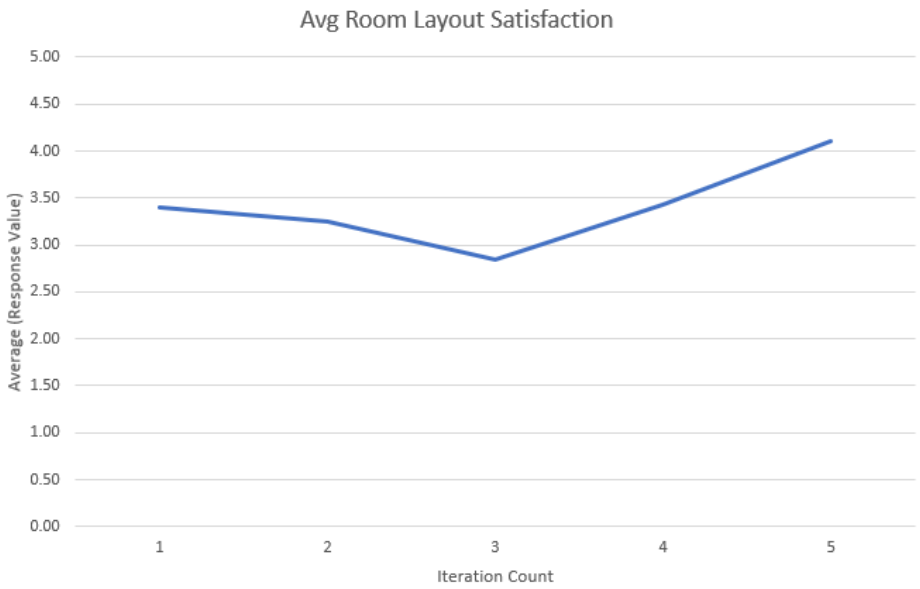


Figure 25: Average results over all five iterations from statement four

4.1.5.5 Exploration Choice

Statement five was included to ascertain how happy participants were with the choices given through each dungeon iteration of choices in exploration. The results from this statement can allow us to understand if the layout of dungeon can provide an objective amount of fun and interest from all the techniques used. As can be seen from Figure 26 and Figure 27, the responses are widely mixed and are unable to display any sort of recognisable pattern. This may be caused by the designed nature of the statement and the fact that the dungeons test were merely layouts and did not contain any further objects, causing participants to not find any interest in exploring the dungeons, therefore giving the response of having no strong opinion.

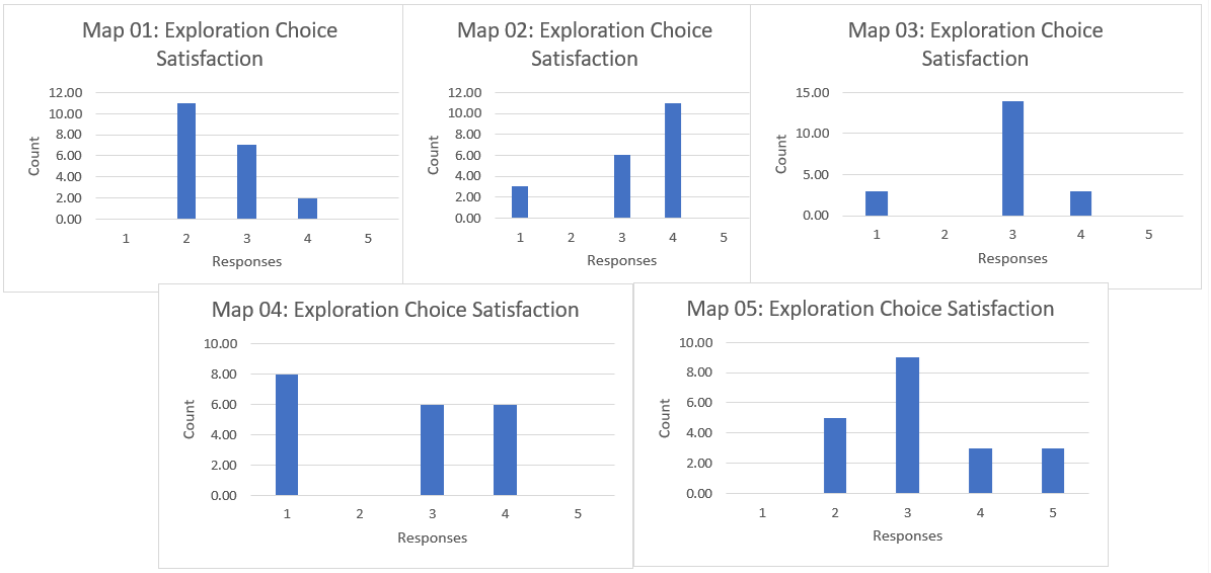


Figure 26: Counted data over all iterations from statement five

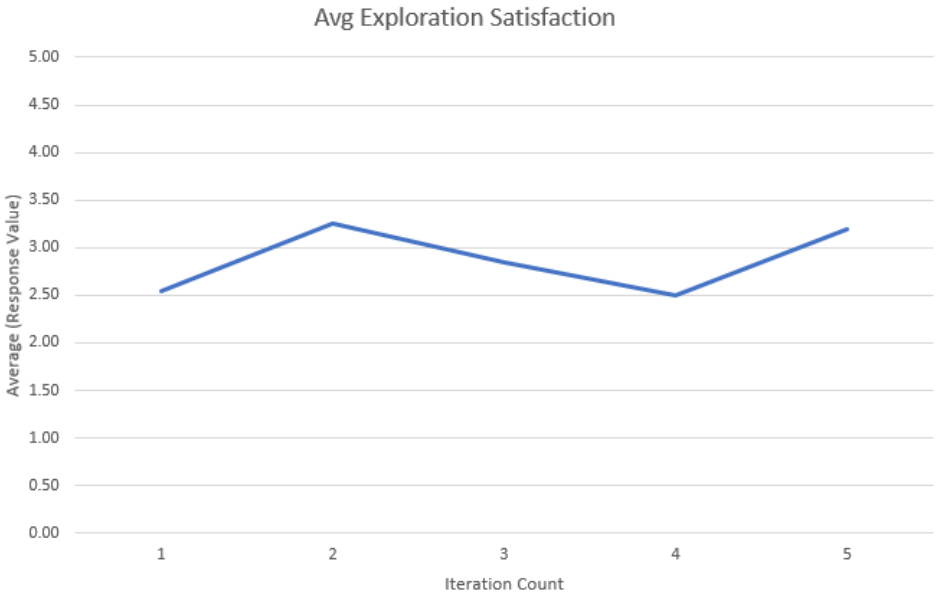


Figure 27: Average results over all five iterations from statement five

4.1.5.6 Natural Feeling

Statement six was included as an additional point to ascertain how strongly each participant thought the individual dungeons had a natural feel in terms of their layout. This was important to look at, given the original use for L-Systems in modelling plant growth and natural systems. A good result in this category would confirm that the dungeons, while using L-Systems, still retained some of the aesthetic flow and symmetry that is present in the L-Systems described by Lindenmayer. The responses gathered from this statement, shown in Figure 28 and Figure 29, show a clear rise in agreement with the statement as the iterations of the L-System increase. We can

draw from this that the dungeon layouts do still retain some of the elements native to L-Systems. The responses from this may have been improved more had the participants been able to see the entire map layout at once.

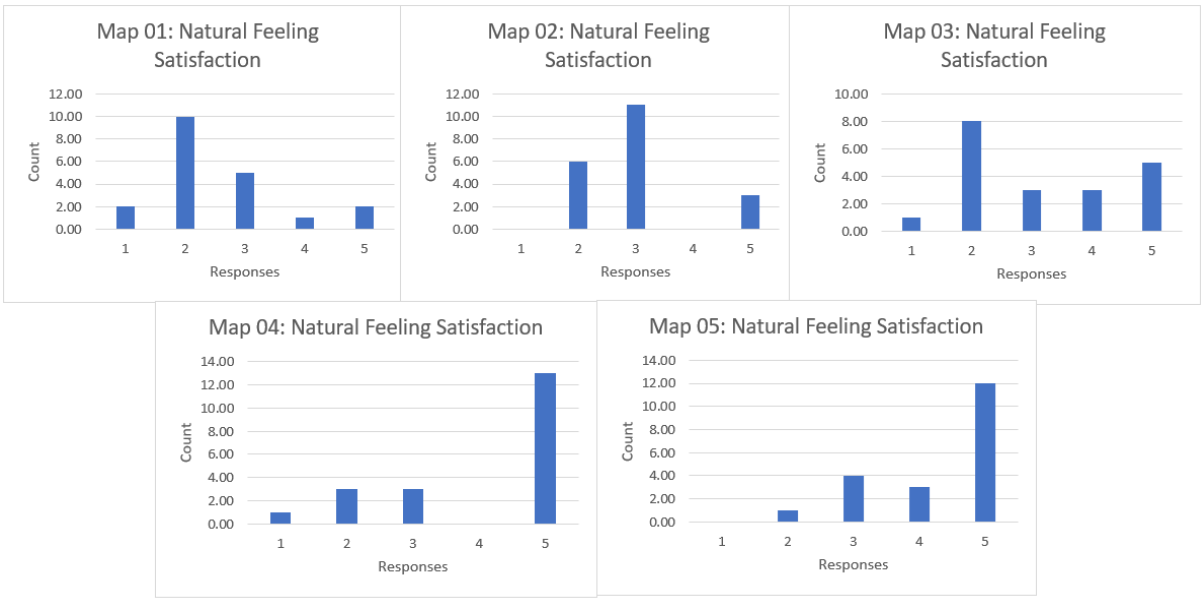


Figure 28: Counted data over all iterations from statement six

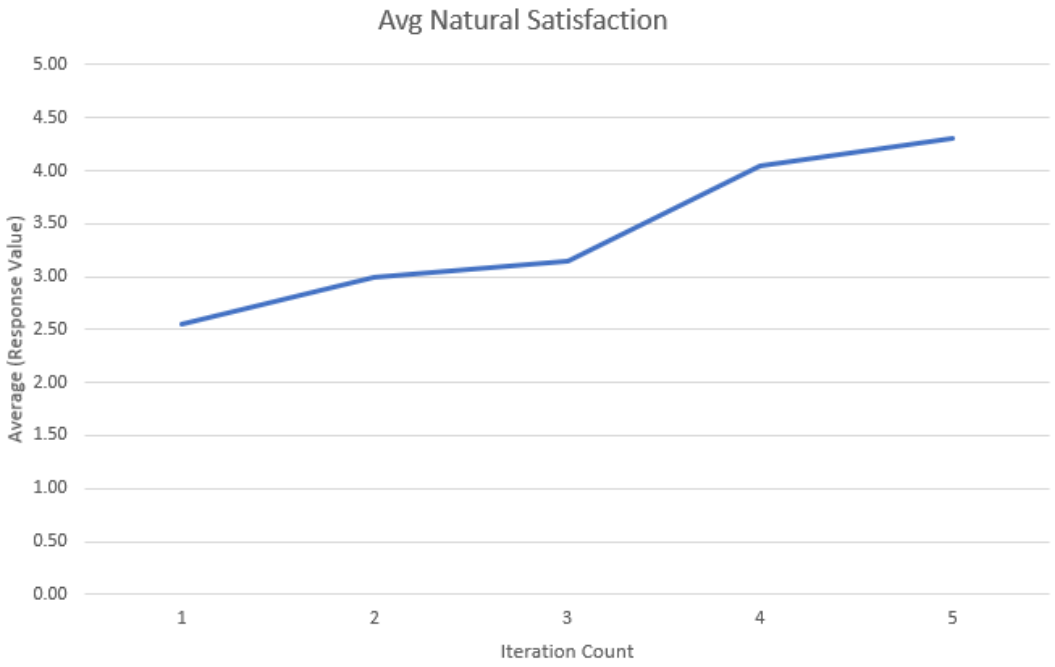


Figure 29: Average results over all five iterations from statement six

5 Chapter 5 | Conclusion

This final chapter contains a detailed overview and summary of the project, while also discussing and making recommendations for further studies and future work to be carried out regarding the use of L-Systems in procedural generation. A personal reflection will also be provided as the final subsection.

5.1 Objective Summary

This subsection will look back at the objectives outlined in the first chapter and provide a summary of those objectives in terms of the results and completed application.

5.1.1 Objective One

The first objective of this study was to identify whether the technique of L-Systems can be used in procedural generation to produce consistently random yet constrained results. This objective has been met as can be seen from the implementation and results sections, the maps generate provide a consistent amount of randomness with no two maps being the same. The maps are generated in a constrained fashion both in terms of:

- **Map size**

  The map size is constrained by limiting the maximum number of symbols present in the axiom, therefore limiting the overall size of the map. The room count within the dungeon can also be changed in this way.

- **Room size**

  While randomised, the room sizes have pre-defined maximum and minimum sizes that can be changed.

- **Room type**

  The application provides constrains in terms of ensuring that the map always contains a starting and ending room, while rooms in-between have no constraints on their types, they can easily be added.

- **Geographical spread**

  The application ensures that rooms are spread evenly in the direction of the L-System, providing constraints in terms of rooms being unable to overlap.

5.1.2 Objective Two

The second objective of the study was to look at how the algorithm and results obtained from the study compared in terms of performance and quality regarding other techniques. Looking at the results produced by the L-System technique, the potential for at least meeting the quality of other techniques such as Cellular Automata or Perlin Noise, however, a definitive statement would require the implementation of one or more of these alternative methods and would require qualitative data to be gathered on multiple implementations.

5.1.3 Objective Three

The third objective of the study was to look what other techniques could be used in conjunction with L-Systems. The use of the pathfinding within the L-System gives one example of using other techniques in conjunction with L-Systems.

5.2 Future Work

This subsection will layout and discuss recommendations of further work that could be taken in relation to this project. It is important to continue to work on exploring additional technique for procedural generation in video games and these recommendations should give some guidance for additional exploration in terms of L-Systems.

5.2.1 Gameplay Implementation

One thing left absent from this study was the implementation of gameplay in the final application that was tested by participants. It would be interesting to gather data from a similar study in which the application was fully fleshed out with game design and objectives. Giving participants goals to achieve within the application may lead to more favourable results being gathered using either the same survey or an extended one. This would realistically require a control group to ensure that participants were not responding exclusively to the gameplay and that concentration of the study remained on the procedural maps themselves.

5.2.2 3D Implementation

Already shown in Lindenmayer's work and in their use in creating three-dimensional models for plant structures in existing games, is the ability for L-Systems to do well in creating three-dimensional structures. This proves that L-Systems can work well with the additional dimension and repeating this study in a three-dimensional space is the logical next step. Doing so would allow for additional proof that the techniques covered in this study are viable and can be implemented in more than just two-dimensional worlds.

5.2.3 API Implementation

The third recommendation would be to create an API or standalone application that can produce two-dimensional or three-dimensional maps using L-Systems that can be exported and easily incorporated into third-party applications such as Unity or Unreal Engine. This would also include the implementation of something that was briefly considered during this study, the implementation of a GUI to change the constraints of each map at a user level. Things such as adding new rules or increasing the tile map size.

5.2.4 Improving Efficiency

The final recommendation would be to implement techniques to improve the efficiency of the dungeon generation, especially at higher levels of iterations as even the basic L-System can take an exponential amount of time to generate. Techniques such as multithreading and reducing the overhead or memory usage from other concurrent systems would be able to provide an increase in efficiency. Doing so would allow for larger, much more complex maps to be generated.

5.3 Personal Reflection

Overall, I feel the project was a success as it does prove that the technique can be used in the way the project intended, however, it would certainly require more work and testing to be proven as something that could be used in the industry at large. The testing and overall application would have been greatly improved by managing to implement gameplay features and object placement, possibly using one of the techniques included in 2.1, such as Cellular Automata.

If the project was to be carried out again, I would opt to use something other than Unity, forgoing the useful user interface that Unity provides, and instead use

something that allowed for a deeper control in programming, such as Monogame (a Microsoft Game Framework) or SFML (Super-Fast Media Library). This was due to most issues stemming from a base lack of understanding of the Unity Engine and the lack of first-party debugging tools, relating to code, that Unity provides.

The data collected from the testing phase was interesting, however having a better designed testing framework and a larger testing pool would have, I feel, improved the results. The testing also failed to question participants about other techniques which was unfortunate as the comparison would have been interesting, however it was felt that the application was not at a stage that it could stand against tried and tested techniques.

References

A. Lindenmayer, P. Prusinkiewicz (1990). *Algorithmic Beauty of Plants*. (4th ed.). Springer-Verlag.

D. Adams (2002). Automatic Generation of Dungeons for Computer Games. (1st ed.). Unknown.

E. W. Weisstein (2020). "Moore Neighborhood" from MathWorld. Retrieved from https://mathworld.wolfram.com/MooreNeighborhood.html.

I. Antoniuk, P. Hoser, D. Strzeciwilk (2019). L-System Application to Procedural Generation of Room Shapes for 3D Dungeon Creation in Computer Games. (1st ed.). Warsaw University of Life Sciences.

J. Suh, H. Zhang, Z. Wang (2018). Procedural Generating of Plants Models using L-Systems. (1st ed). University of Southern California.

J. Von Neumann, J. W. Burks, A. W. Burks (1966). Theory of self-reproducing automata. University of Illinois.

K. Pedersen (2014). Procedural Level Generation in Games using a Cellular Automaton. Retrieved from https://www.raywenderlich.com/2425-procedural-level-generation-in-games-using-a-cellular-automaton-part-1.

K. Perlin (2002). Improving Noise. (1st ed.). New York University.

M. Bauderon, H. Jacquet (1999). Node Rewriting in Hypergraphs. (1st ed.). University of Bordeaux.

M. Gardner (1970). Mathematical Games: The Fantastic combinations of John Conway's new solitaire game "Life". Scientific American, Vol. 223.

Microsoft. (2019). Visual Studio Documentation. Retrieved from https://docs.microsoft.com/en-us/visualstudio/windows/?view=vs-2019&preserve-view=true.

Microsoft. (2020). C# Documentation. Retrieved from https://docs.microsoft.com/en-us/dotnet/csharp/.

Microsoft Corporation (2020). Github Documentation. Retrieved from https://docs.github.com/en

N. A. Barriga (2019). A Short Introduction to Procedural Content Generation Algorithms for Videogames. (1st ed.). University of Talca.

P. E. Hart, N. J. Nilsson, B. Raphael (1968). A Form Basis for the Heuristic Determination of Minimum Cost Paths. SRI International.

R. L. Cook, T. DeRose (2005). Wavelet Noise. (1st ed.). Pixar Animation Studio.

R. van der Linden, R. Lopes, R. Bidarra (2014). Procedural Generation of Dungeons. (1st ed.). Delft University of Technology.

S. A. McLeod. (2019). *Likert scale*. Simply Psychology. Retrieved from https://www.simplypsychology.org/likert-scale.html.

S. Wolfram (1983). Cellular Automata. (1st ed.). Los Alamos Science.

V. Terrier (2002). Two-dimensional cellular automata and their neighborhoods. University of Caen.

Unknown (2021). Tiles and Tilemaps Overview. Retrieved from https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps. MDN Web Docs.

Unity Software Inc. (2020). Unity User Manual. Retrieved from https://docs.unity3d.com/Manual/index.html.

Unity Software Inc. (2020). Monobehaviour. Retrieved from https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Appendix 1 Project Overview

Initial Project Overview
SOC10101 Honours Project (40 Credits)
Title of Project: 2D Dungeon Generation Using L-Systems
Overview of Project Content and Milestones

This project is to research work done previously on the use of L-Systems in procedurally generating 2D dungeons/levels. The project will also be used to develop a software that successfully allows for the generation of procedural dungeon creation.

The Main Deliverable(s):

The deliverables for this project will be as follows.

- Details and a breakdown of the investigation done into the project and previous work done on L-Systems prior to the development phase
- A description of the steps taken to develop the software/algorithm.
- Testing and evaluation of the resulting software and methods used.
- An executable file with the standalone software.
- The project files and source code.

The Target Audience for the Deliverable(s):

For the target audience, there are a range of people who this project is aimed at. Primarily, this project is targeted at those in the game development who have an interest in procedural generation methods and those building 2D games with specific level types. Additionally, the target audience can be extended to those working in or involved in the tabletop games industry as this project could be used to build outlines for physical levels used in such games.

The Work to be Undertaken:

The work to be undertaken is as follows.

- To investigate previous work done on the subject of L-Systems used in procedural generation.
- To analyse other methods of 2D dungeon generation and evaluate their effectiveness compared to that of the methods used in the project. This will be done in the form of user testing and will allow users to take part in a short game like experience in a blind test using dungeons generated in the created software and comparing them to those generated using pre-existing software that use methods other than L-Systems.
- To develop a software and algorithm using L-Systems to generate 2D dungeons while adhering to constraints.

Additional Information / Knowledge Required:

Through this project I will be able to build upon my knowledge of software development and my knowledge of procedural generation and L-Systems. The final application will be developed in Unity.

Information Sources that Provide a Context for the Project:
Cellular Automata and Perlin Noise have been used a great deal in the past in order to provide procedurally generated levels and as such those techniques seem to have peeked with recent game releases that promise endless amounts of content and are able to produce something similar to what I hope to produce in this project.
Below is a list of references that outline similar approaches or work to this project.

Osama Alsalman. (2018). *Kastle: Dungeon Generation Using L-Systems.* Available: https://www.gamasutra.com/blogs/OsamaAlsalman/20180827/325319/Kastle_Dungeon_Generation_Using_LSystems.php.

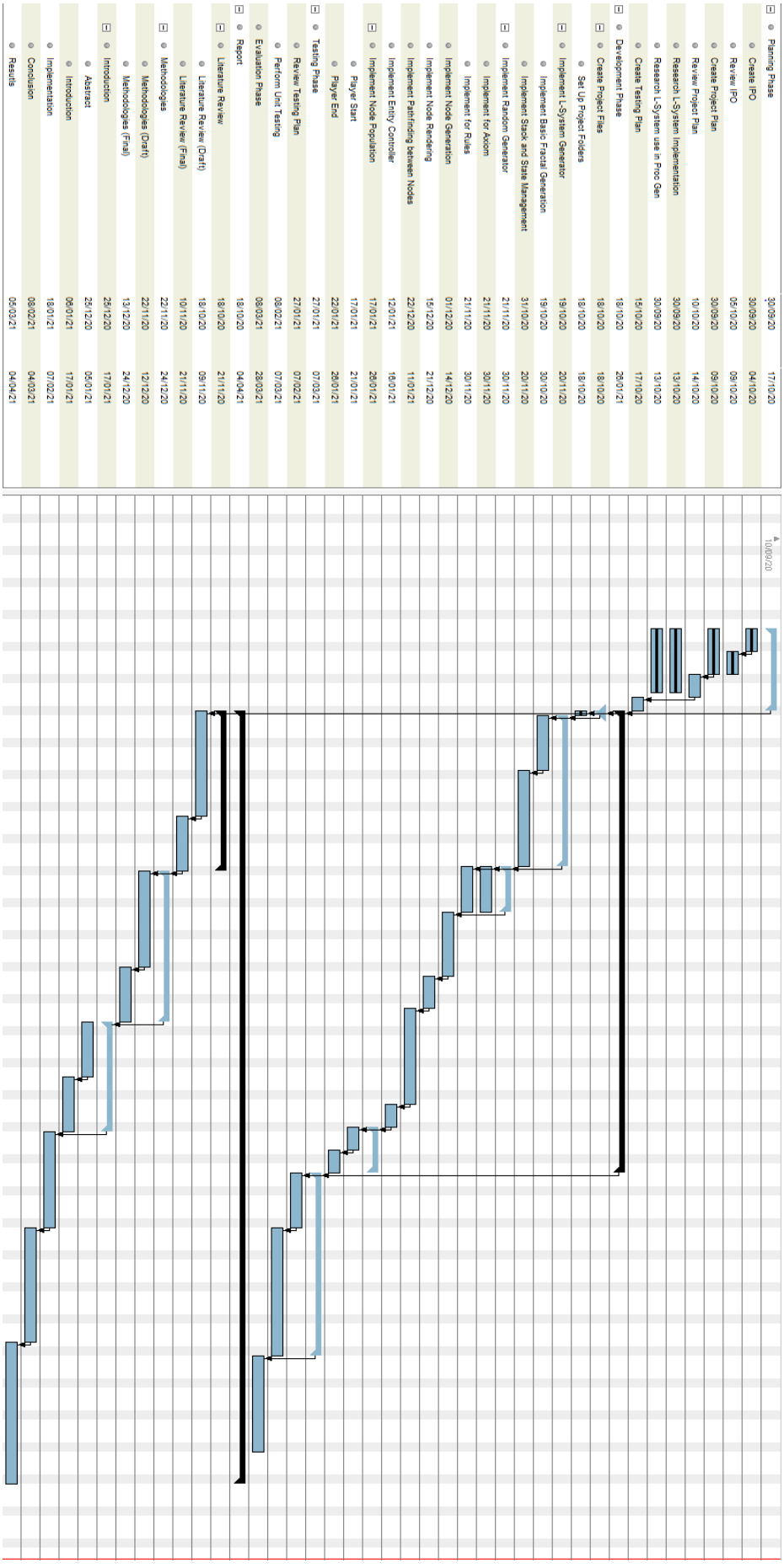
Paul Bourke. (1991). *L-System User Notes.* Available: http://paulbourke.net/fractals/lsys/.

The Importance of the Project:
Procedural generation is a topic that has been widely researched and used for a long time, as is the use of L-Systems, however expanding on what L-Systems can achieve and what other uses they have is not something that has been done to a great degree. 2D level generation specific to procedural dungeons is something that has been done in the past but from initial research, it is not something that has been combined with L-Systems to a great success.

The Key Challenge(s) to be Overcome:
The main challenge anticipated in this project is to implement a way in which the L-System can be used in conjuncture with constraints, ensuring that the dungeon can be populated with elements (this is in reference to game elements as well as rooms and corridors) in a way that takes into consideration room/level sizes and other placed elements within each room. A similar challenge would be to introduce a consistent random element into the fractal algorithm.

Another challenge to this project could be sourcing reliable material already created on this subject. There are a wealth of papers etc, outlining L-Systems themselves and their uses in botanical simulation, however material for their other uses is much lighter.

Project Plan



Planning Phase	30/09/20	17/10/20
Create IPO	30/09/20	04/10/20
Review IPO	05/10/20	09/10/20
Create Project Plan	30/09/20	09/10/20
Review Project Plan	10/10/20	14/10/20
Research L-System Implementation	30/09/20	13/10/20
Research L-System use in Proc Gen	30/09/20	13/10/20
Create Testing Plan	15/10/20	17/10/20
Development Phase	18/10/20	28/01/21
Create Project Files	18/10/20	18/10/20
Set Up Project Folders	18/10/20	18/10/20
Implement L-System Generator	19/10/20	20/11/20
Implement Basic Fractal Generation	19/10/20	30/10/20
Implement Stack and State Management	31/10/20	20/11/20
Implement Random Generator	21/11/20	30/11/20
Implement for Axiom	21/11/20	30/11/20
Implement for Rules	21/11/20	30/11/20
Implement Node Generation	01/12/20	14/12/20
Implement Node Rendering	15/12/20	21/12/20
Implement Pathfinding between Nodes	22/12/20	11/01/21
Implement Entity Controller	12/01/21	16/01/21
Implement Node Population	17/01/21	28/01/21
Player Start	17/01/21	21/01/21
Player End	22/01/21	28/01/21
Testing Phase	27/01/21	07/03/21
Review Testing Plan	27/01/21	07/02/21
Perform Unit Testing	08/02/21	07/03/21
Evaluation Phase	08/03/21	28/03/21
Report	18/10/20	04/04/21
Literature Review	18/10/20	21/11/20
Literature Review (Draft)	18/10/20	09/11/20
Literature Review (Final)	10/11/20	21/11/20
Methodologies	22/11/20	24/12/20
Methodologies (Draft)	22/11/20	12/12/20
Methodologies (Final)	13/12/20	24/12/20
Introduction	25/12/20	17/01/21
Abstract	25/12/20	05/01/21
Introduction	06/01/21	17/01/21
Implementation	18/01/21	07/02/21
Conclusion	08/02/21	04/03/21
Results	05/03/21	04/04/21

Appendix 2 Second Formal Review Output

Student Name: Calum Mathison 40406464
Supervisor: Babis Koniaris
Second Marker: Kevin Sim
Date of Meeting: 11 November 2020

Can the student provide evidence of attending supervision meetings by means of project diary sheets or other equivalent mechanism? **yes**

If not, please comment on any reasons presented

Please comment on the progress made so far

Literature review is generally good. Referencing style for in-text citations and the bibliography should be APA6. Some sections of the review could do with more explanation. The number of references is quite small.

Progress on implementation was not presented at the meeting but was discussed. The final application will be implemented in Unity. Currently a simple L-System generator has been implemented but needs better integration with Unity to render levels and add artifacts.

There was also discussion related to generating navigable maps and building these constraints into the generation algorithm.

Is the progress satisfactory? **yes**

Can the student articulate their aims and objectives? **yes**

If yes then please comment on them, otherwise write down your suggestions.

The student clearly described the aims of the project and exhibited a good understanding of the underlying research areas underpinning the project

Does the student have a plan of work? **yes** **no***
If yes then please comment on that plan otherwise write down your suggestions.

Not presented at meeting but the student stated they had a workplan and had given thought to the remaining sections of the report and suitable timescales.

Does the student know how they are going to evaluate their work? **yes** *
If yes then please comment otherwise write down your suggestions.

Unclear from report. During the meeting the student mentioned different ways of evaluating the work. Comparing levels generated by L-Systems with levels generated using cellular automata. Conducting a usability study to gauge the playability of generated levels.

Any other recommendations as to the future direction of the project

Comments have been added to the word document that was provided by the student before the meeting.

The literature review could also cover different technologies and justify why Unity was chosen.

Using a reference manager would help with formatting the bibliography correctly. Mendeley is popular, integrates with Word and is free to use https://www.mendeley.com/

Signatures: Supervisor Babis Koniaris Second Marker Kevin Sim

Student Calum Mathison

Appendix 3 Diary Sheets (or other project management evidence)

Project Diary 01

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 13/10/2020 **Last diary date: 13/10/2020**

Objectives:

Work towards a basic prototype and start working on literature review in preparation for Interim report. Complete Gantt chart

Progress:

Created the base project in Unity and implemented a simplified version of an L-System Generator that can produce certain fractals. Need to implement stack support and state control to allow for fully fledged use of L-System. Started taking notes from literature read during research phase in preparation for writing a literature review. Completed Gantt Chart up to Evaluation phase

Project Diary 02

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 28/10/2020 **Last diary date: 13/10/2020**

Objectives:

Work on completing literature review draft. Work on implementing stack support for L-System generation.

Progress:

Worked on adding parts to literature review. Plan to have literature review completed for Monday 02/11/20

Wrote pseudo code for stack implementation. Researched and tested adding stack implementation for L-Systems. Plan to have feature implemented for Friday 30/10/20.

Added iteration input. Allowing you to visualise the changes between each iteration.

Project Diary 03

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 18/11/2020 **Last diary date: 28/10/2020**

Objectives:

Finish stack and state implementation.

Start working on procedural generation for axiom and rules. Finish adding feedback to chapters.

Progress:

Finished adding stack and state support to L-System generator. L-Systems can now produce branching paths.

Project Diary 04

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 25/11/2020 **Last diary date: 18/11/2020**

Objectives:

Create and render basic rooms like objects on l-system

Progress:

Started working on room rules/objects

Project Diary 05

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 02/12/2020 **Last diary date: 25/11/2020**

Objectives:

Get rooms rendering through l-system

Progress:

Added simple way of rendering rooms using coloured 3d planes. Currently rendering in a straight line but able to represent the room type. In the middle of adding a way to add l-system angles back into rooms. Have a corridor object that needs to be added after that. Added the ability to randomise and increase iterations to generate more rooms although kept constraints on the first and last room.

Project Diary 06

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 09/12/2020 **Last diary date: 02/12/2020**

Objectives:

Add branches, collisions and corridors to the l-system

Progress:

Conducted some reading and research on topics discussed during last week's meeting.

Project Diary 07

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 16/12/2020 **Last diary date: 02/12/2020**

Objectives:

Fix Visual Studio.

Add branches to the l-system.

Fix corridors and add collision to l-system.

Progress:

Added branch object to allow for branching path. Works and looks good but does not work well with the way the system currently selects random characters. Need to add restraints to so that branch generation symbols are created in pairs as its currently creating errors with the data structure holding the states.

Began adding collision detection for room generation.

- Tried adding collision using Rigidbody physics. Technically worked but was hard to control and unpredictable. Accidentally created an interesting particle simulation.
- Tried adding collision using collider physics. Working, but needs testing and requires refactoring of the base room classes.
- Tried adding collision based on positions and removing rooms with the same positional data as others. Works, but requires the way that rooms are generated and stored to be changed.

Need to redo how rooms are stored before continuing with the collision stuff.

Added another symbol of ' ' to corridors to connect every room with a corridor even if there was no angle but ended up producing ridiculous amounts of corridors. Plan to take corridors out of the system as individual rooms and add them in after rooms have been generated.

A bit late but created github repo and added source control.

Need to reinstall Visual Studio or uninstall CUDA as still have a problem with CUDA and VS no longer registers errors or intellisense. Also have making unity duplicate objects when selecting multiple objects.

Going to redo how objects are created and make them prefabs instead as discovered objects don't update on the change of any connected elements.

Project Diary 08

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 26/01/21 **Last diary date: 16/12/2020**

Objectives:

Begin adding gameplay elements for testing.

Progress:

Implemented the node generation onto a tile map. Implemented A* to connect nodes and render hallways. Have rooms automatically generate with a random width and height. Constrained the maximum number of nodes. Implemented wall generation around rooms. Fixed errors coming from random generation of the axiom string.

Project Diary 09

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 09/02/2021 **Last diary date: 02/02/2021**

Objectives:

Fix Overlap, Finish pathfinding

Progress:

Implemented fix to overlapping rooms (was surprisingly simple). Tried a couple of other methods of implementing A* using online examples. Looked at adding dijkstra's algorithm as a simpler solution. Continuing to work on pathfinding. Looked at the option of using unity tile map object instead of creating one from scratch but would require refactoring of most code.

Project Diary 10

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 23/02/2021 **Last diary date: 16/02/2021**

Objectives:

Implemented A*. Added collision for wall tiles. Added doors to rooms. Add player controlled object. Removed diagonal pathing.

Progress:

Completed A* implementation. A* now draws corridor paths and works together with wall generation. Added collision detection for wall tiles so that the player cant pass through them. Added doors at the entrance of each corridor. Currently blocks player movement, need to add way of player opening them. Added simple player using a test sprite. Remove diagonal pathing by adding additional corner nodes to add 90 degree turns. Need to debug as it seems to have affected the way A* handles the branching paths.

Project Diary 11

Student: Calum Mathison **Supervisor: Babis Koniaris**

Date: 16/03/2021 **Last diary date: 23/02/2021**

Objectives:

Tweak L-System algorithm. Debug and fix errors. Come up with a survey for testing.

Progress:

Worked on modifying the L-System algorithm. Started debugging the method that separates the room objects. Still throws a stack overload exception. Started looking at GDPR and questions for testing.

Appendix 4

Procedural Dungeon Generation
Using Lindenmayer Systems Survey

Introduction

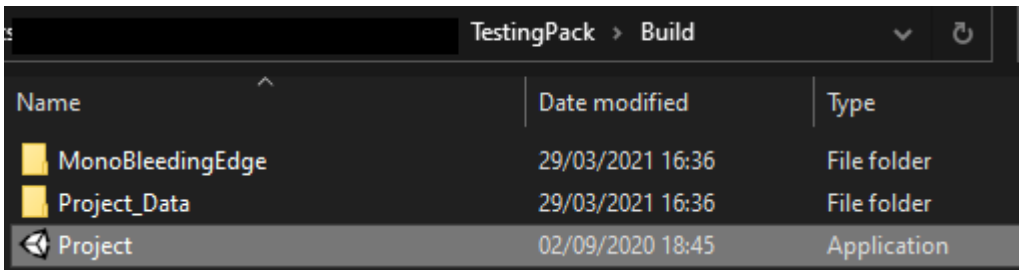
This exercise/questionnaire is part of a research study into the use of Lindenmayer Systems to procedurally generate dungeons and levels within video games.

What is a Lindenmayer System?

An L-System is a technique traditionally used to describe the behaviour of plant cells and to model the growth processes of plant development.

Installation

The software does not require installation. To open the software, click on the application names Project within the build folder.



Controls

WASD – Move character up, down, left, right

Right Arrow Key – Generate new map and increase iteration by 1

Left Arrow Key – Generate new map and decrease iteration by 1

If for some reason a map does not generate properly, simple pressing the right then left arrow keys will refresh the map at the current iteration count.

Survey

Please answer the following questions bellow using the corresponding scale for each of the five maps that you generate. **Please try to answer all the questions for a single map at a time as it is unlikely you will be able to view the same map twice.**

1. I find that the dungeon layout is confusing
Fully Disagree 1 2 3 4 5 Fully Agree

2. I find that there are too many corridors
Fully Disagree 1 2 3 4 5 Fully Agree

3. I find the rooms to be of adequate scale
Fully Disagree 1 2 3 4 5 Fully Agree

4. I find that the rooms are spaced out well
Fully Disagree 1 2 3 4 5 Fully Agree

5. I find that there is a good choice of exploration
Fully Disagree 1 2 3 4 5 Fully Agree

6. I find that the dungeon has a natural feel
Fully Disagree 1 2 3 4 5 Fully Agree

If you have any other comments, observations, or feedback, please add them in the extra box below each map along with the axiom and iteration number displayed on screen if possible.

Map 1

1.	
2.	
3.	
4.	
5.	
6.	
7.	Axiom: Iteration: Comment:

Map 2

1.	
2.	
3.	
4.	
5.	
6.	
7.	Axiom: Iteration: Comment:

Map 3

1.	
2.	
3.	
4.	
5.	
6.	
7.	Axiom: Iteration: Comment:

Map 4

1.	
2.	
3.	
4.	
5.	
6.	
7.	Axiom: Iteration: Comment:

Map 5

1.	
2.	
3.	
4.	
5.	
6.	
7.	Axiom: Iteration: Comment: