|  |
| --- |
| An investigation into the use of Procedural Level Generation to create more replayable 2D platformers  Callum Myers  Computer Games Application Development, 2025 |

|  |
| --- |
| School of Design and Informatics  Abertay University |

Table of Contents

[Table of Figures ii](#_Toc92987347)

[Table of Tables iii](#_Toc92987348)

[Acknowledgements iv](#_Toc92987349)

[Abstract v](#_Toc92987350)

[Abbreviations, Symbols and Notation vi](#_Toc92987351)

[Chapter 1 Introduction 1](#_Toc92987352)

[Chapter 2 Literature Review 2](#_Toc92987353)

[Chapter 3 Methodology 3](#_Toc92987354)

[Chapter 4 Results 4](#_Toc92987355)

[Chapter 5 Discussion 5](#_Toc92987356)

[Chapter 6 Conclusion and Future Work 6](#_Toc92987357)

[List of References 7](#_Toc92987358)

[Bibliography 8](#_Toc92987359)

[Appendices 9](#_Toc92987360)

# Table of Figures

# Table of Tables

# Acknowledgements

# Abstract

# Abbreviations, Symbols and Notation

If required

# Chapter 1 Introduction

With a significant influx of new games hitting the market every single day, there is pressure piling on game developers to make their project stand out from the crowd. This often means having to make vast amounts of content for the game to give players the best value for money possible, and this may cause longer development times as well as huge costs for the studios to keep enough staff on the team to manage the increase in workload.

One way this workload could be reduced is using Procedural Content Generation, or PCG. PCG is defined as “the algorithmic creation of game content with limited or indirect user input” (Shaker, N., Togelius, J. and Nelson, M.J., 2016, pp. 30). Using a variety of algorithms and techniques, developers can create a model which can help them to generate entire levels all at once, without the need for level designers to draw and implement each one by hand, significantly cutting the workload down to be more manageable. Creating these algorithms may require some more time dedicated from the programmers, however this could be seen as a longer-term investment, since an appropriate model could be adapted for use in many different games by the team. By simplifying the level generation process, the team will have more time to improve other aspects of the game, such as mechanics or graphics, to ensure the player has an engaging experience throughout their playthroughs.

Another benefit PCG can offer is creating a new, unique experience for the player each time they play. A strong example of this is seen in the 2D platformer; Spelunky [Derek Yu, 2008]. Within Spelunky, levels are semi-procedural, meaning they are created dynamically, using presets which are made by level designers prior to the game’s release. This results in the game containing a massive number of possible level layouts, without requiring the level designers to implement them by hand. Because of this, players can play through the game many times, and the likelihood of experiencing the same levels multiple times is much lower, leading to fun, fresh and enticing gameplay each time.

While PCG is great at keeping content feeling unique and fresh every time, one challenge often encountered when using it in game development is the lack of emotion felt throughout the levels. Hollow Knight (Team Cherry, 2017) displays this, such as early in the game where the player can find remains of knights in The Forgotten Crossroads. This helps to tell a story, since they are placed strategically and expertly by the level design team and would not be as effective if they were randomly scattered using a PCG algorithm. This issue can be combated by using a variety of “online” and “offline” generation techniques. Online generation involves generating the content during gameplay, such as the way Minecraft (Mojang, 2009) handles it, where the world is created in small 16x16 sections, or “chunks”, as the player approaches. This method creates a much more randomised generation. Offline involves generating most or all the content all at once, for example when the player first opens the game or even before the game is released. This method could allow developers to create many levels all at once during the development with the algorithm, then allow level designers to go through and add in their own unique aspects, finding that balance between variety and offered by human touch and the unique feel PCG can offer.

**Research Question:**

*How can procedural level generation be used to lengthen a player’s playtime and improve player engagement in 2D platformer games?*

This project aims to answer this question by designing and developing a procedural level generation model in the Unity game engine which will create a series of levels for a 2D platformer game. This will be done using procedural techniques such as Noise Functions, particularly Perlin Noise, and a series of Markov Chains which will create new and unique levels on demand. By creating terrain generated using Perlin Noise, as well as populating the levels with objects using Markov Chain decisions, this will create unique and diverse levels, keeping players engaged and surprised throughout the levels.

**Aim and Objectives:**   
This project aims to investigate and analyse the benefits of procedural generation algorithms when creating levels for 2D platformers, then consider how this could be used to lengthen the time a player can stay engaged with the game. This will then be tested by several participants who will use the generation model and fill out a short questionnaire which will provide feedback into how they feel the level generation in the project compares to levels they have played in other, more traditional platformer games. The objectives of the project can be summarized as:

* To research how procedural level generation could be used in future 2D platformers to give the player a unique experience each time they start a new game.
* To evaluate the level and length of player engagement in a classic linear platformer when compared to one featuring procedural level generation techniques.
* To utilise programming techniques such as Perlin Noise and Markov Chains to implement a procedural level generator using the Unity game engine which effectively generates interesting and varied levels.
* To evaluate the feedback provided by participants to consider how procedural level generation compares to traditional level design in keeping players engaged for longer.
* To research current 2D platformers using procedural generation, for example Spelunky, and consider how this could be improved upon using my own methods and techniques.

# Chapter 2 Literature Review

**2.1 – Current Use of Procedural Generation in Games and Research**

**2.1.1 – Overview**

The first aim of this project was to research how procedurally generated content is currently being used successfully in games.

Using the Procedural Content Generation Wiki [Wikidot], two games were highlighted that could provide some valuable insight into the current state of PCG in games:

* Minecraft
* Spelunky

This section will cover each of these games in more detail, highlighting their success, as well as how procedural generation affects the game, and which techniques were used.

There was also a research paper which seemed particularly interesting in the context of the project. It used a combination of Markov Chains and machine learning to create a model which can procedurally generate levels in the style of Super Mario Bros [Nintendo, 1985].

**2.1.2 - Infinite Mario Bros**

Infinite Mario Bros was a research project taken on by Sam Snodgrass and Santiago Ontañón [Sam Snodgrass, 2014], where the aim was to create a procedural level generation model, using a combination of machine learning and Markov Chains, to dynamically create levels for an infinite version of Nintendo’s Super Mario Bros [Nintendo, 1985]. To achieve this, they first assign each type of tile a letter, for example G for ground, P for pipes, etc., then feed this data into a machine learning algorithm to evaluate where in the level they most often occur, and how common they are throughout the levels. From here, the levels are then built tile-by-tile, by feeding this data into a Markov Chain to decide which tile should be placed at each point.

A key point which can be taken from this paper’s conclusions, and applied to the current project, is that the Markov Chains created levels which were unique and varied. Whilst all levels were using the same set of available tiles, each one had objects placed in varying locations with different combinations, making each level feel new.  This is important for the project because it shows how player engagement could be improved using Markov Chains. Players can get used to the limited number of tile *types;* however, they may get surprised by the variation of how they are used and combined through the level.

**2.1.3 - Minecraft**

Minecraft [Mojang Studio, 2009] is an open-world sandbox game that has become the Guinness World Record holder for the best-selling video game of all time [Guiness World Records, 2023]. In 2023, it officially reached over 300 million copies sold, and as of January 2025, it still has over 50 million players logging in every day [Saisuman Revanker, 2025].

With such a vast number of live players almost 16 years after launch, Minecraft must have something to keep players constantly coming back and a huge part of this can be attributed to its virtually infinite combination of worlds. Each world in Minecraft is generated entirely using procedural generation, through a combination of Perlin Noise, Fractal Noise, and 3D noise [Dawnosaur, 2023]. On top of this, like how this project’s levels are generated, Minecraft does its world generation across three different passes [Alan Zucconi, 2022]. In the first pass, a “biome map” is generated, which dictates how each section of the world is generated. For example, mountain biomes will allow the world to be generated up to a higher altitude, whilst a desert or forest biome may keep the world at a lower altitude. The next pass then covers the biomes in an appropriate material, such as stone for the mountains and sand for the deserts, then grass for the forests. The third pass then goes back over the world and removes ground from specific areas to create caves or ravines, whilst the fourth and final pass decorates the world with structures.

Through these techniques, the game can generate approximately 18 quintillion unique world combinations [Minecraft Wiki]. Overall, Minecraft portrays a fitting example of how a game can use procedural generation to its advantage and allow players to replay the game multiple times over, causing it to retain popularity many years after release.

**2.1.4 – Spelunky**

Spelunky is a 2D platformer that uses procedural generation to define its’ level layouts. Unlike Minecraft [Mojang Studios, 2009] however, the game is not entirely random. Each level is a four-by-four grid, and each section of this grid contains a pre-made template, created by level designers. Since each section has around 8-16 templates [TinySubversions, 2013], this means there are at least 8^16 (280 trillion) level layouts, based on terrain alone. On top of this, the game then parses back over the level and checks for any “probabilistic” tiles in the room, which are tiles that can be replaced with several other blocks. This means there are even more than 280 trillion layouts, so it is entirely possible for players to play the game many times without repeating levels.

Whilst this generation may not provide the colossal number of layouts Minecraft can, it does give much more input to level designers. This is an excellent example of PCG not only because of its replayability but also because it helps to solve an issue mentioned in the introduction of this paper, where level designers are under more pressure to create a huge influx of levels to keep players engaged. By combining handmade level templates with the procedural generation which stitches them together randomly, the levels have a more personal touch, keep development time down by taking pressure off the level designers, and still give players the replayability value that can drastically increase the value-for-money aspect of the game.

**2.2. – Procedural Generation Techniques**

**2.2.1 – Overview**

After researching the many procedural generation techniques and algorithms available, Perlin Noise and Markov chains were deemed to be the best options in the context of the project.

One technique which was considered, but later removed from the project plan, was Wave Function Collapse (WFC). It was replaced with Markov Chains because the after research, it was found that Markov Chains are significantly more computationally efficient than WFC. WFC also relies on tighter constraints, whereas Markov Chains can return a wider range of outputs, making them more suitable for level variation.

Before starting the implementation of the level generator, some criteria were laid out to decide what would outline a successful level:

* The terrain of each level should be unique enough that a player could replay the game multiple times without encountering two levels that felt too similar. This can be achieved with the effective use of Perlin Noise.
* Each level must have enemies and collectibles placed randomly throughout to encourage the player to explore or find alternative routes. This can be achieved using Markov Chains.
* Each level must have at least one path which guarantees the player can reach the end goal and win the level. This can be achieved by setting rules and constraints for the level generator scripts.
* Players should have the ability to edit parameters that will customize how future levels are generated, allowing them to see how the procedural generation can lead to a variety of level types.

**2.2.2 – Perlin Noise**

Perlin Noise is a procedural generation technique that was first introduced and used by Ken Perlin in 1985 [[Ken Perlin](https://dl.acm.org/doi/pdf/10.1145/325165.325247), 1985]. The development of the noise function began because Perlin wanted to find a more efficient way of creating realistic textures to be applied to models in CGI. As Perlin mentioned in his research, many of the procedural generation techniques at the time required combining multiple individual functions, which required rewriting, recompiling, and rerunning the program until the desired effect was generated. Perlin found this to be “cumbersome” [Ken Perlin, 1985, Introduction], therefore he created his program, which would generate a black-and-white noise map and then manipulate the values at each point to correspond to different values on the final texture. Perlin outlines in his paper that he developed functions to convert these values to colours, normal, and heights.

Whilst the original intended use of Perlin Noise centred around graphical textures, game developers have since adapted many of these functions to create terrain in their video games. A prominent example of a game that brilliantly utilizes Perlin Noise is Minecraft [Mojang Studios, 2009], and it is fair to assume that much of the game’s success can be attributed to the infinite world generation created partly using this technique. Minecraft’s ground layer is largely generated from a Perlin Noise map, however due to the gradual gradients being one of the characteristics of a Perlin Noise map, the developers did not deem this to create unique enough terrain for an infinite world, therefore they layered multiple maps on top of each other which were combined to make one more significantly unique map [Dawnosaur, 2023]. The height of each block of the world is then scaled according to the grayscale value of the corresponding pixel on the map, like how Perlin created the “Stucco Donut” in his paper [Ken Perlin, 1985, A Pixel Stream Editing Language]. For example, a darker pixel would place blocks up to the upper limit of the world, whereas a white pixel would move towards the lower constraint.

Perlin Noise is a good technique to be used in this project as Perlin designed it to be as computationally efficient as possible. By mapping the height value of points throughout the level to the grayscale value of the noise map, the level generator can effectively adapt Minecraft’s generation technique to be used in the context of a 2D platformer.

A summary of the pros and cons of using Perlin Noise are as follows:

Pros:

* Perlin Noise was designed with efficiency and simplicity in mind, which will help to lower loading times when levels are being created, as well as allowing minor changes to be made which will help achieve desired effects on each level’s terrain.
* Perlin Noise generates softer, more gradual terrain which can be used to create smaller changes in height throughout the level to help prevent impassable walls or slowing down the pace of the level.’
* There has been a significant volume of research surrounding Perlin Noise, and therefore strong documentation, as well as the Unity Editor (being used for the project) including some functions dedicated to getting a Perlin Noise generator started quickly.

Cons:

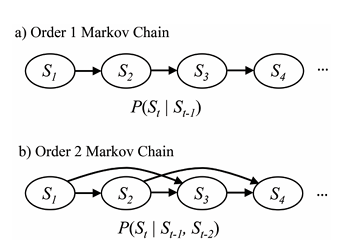
* Perlin Noise maps may sometimes “cluster,” creating areas of the same colour, which can result in extended areas of flat ground, however, this can be negated with secondary passes to add other objects to these areas.
* Perlin Noise maps may start to tile on larger maps, however since the levels in the project will be small, this is not expected to present any issues.

**2.2.3 – Finite Markov Chains**

Markov Chains were created by Andrey Markov during his research on the theories of probability in 1906 [Eugene Seneta, 2006]. The Markov Chain is a structure made of multiple states, where each state may have one or more transitions, but all transition probabilities must sum to one [J.R Norris, 1997]. In practice, this means the chain should never get stuck in one state and will always move forward, however, it also has the chance to have a different outcome each time.

This can be extremely useful in procedural generation since the transition chance of each state can be edited dynamically based on events in the world. This could, for example, mean as a world creates a larger ocean, the chance of creating an island will increase, meaning it is more likely to move from a “water” state to a “ground” state.

There is a key distinction to be made between the two different types of Markov Chains: a standard-order Markov Chain has no memory of its’ past states, meaning the decision of whether it should progress to the next state relies purely on parameters in the current state. A second-order Markov Chain, however, considers the current state as well as the state before, then uses a cumulative chance to choose its’ next state. This can improve the accuracy of the chain, but also increases the computational resources required as it must keep previous state results in memory.



[Sam Snodgrass, 2014].

In the project, standard order Markov Chains are used, since each should result in a true or false result, for example: deciding whether ground should be placed, whether ground should be destroyed to make a gap and deciding whether an enemy should be placed. A second-order Markov may have been more suitable if the project included biomes, for example it could have a state which checks the ground height, and a state which checks the current biome, then a combination of these chances could decide which biome it moves to next.

A summary of the pros and cons of using Markov Chains (standard-order) are as follows:

Pros:

* Markov Chains are very efficient since they do not need to retain any memory of past states, and each state is treated as its own decision. This stays in line with keeping the level generation fast to keep the pace of the game going and preventing players from being stuck on loading screens.
* Markov Chains allow for implementing rules and constraints, which allows the developers to decide how levels should look. This could, for example, prevent too many enemies from being spawned as the number of enemies could be passed as a parameter into the Markov Chain to decrease the chance based on how many are in the level.

Cons:

* Without appropriate parameter planning and restrictions, the Markov Chains will sometimes return the same result many times in a row, which can lead to long and repetitive sections in the level. This can be solved by thoroughly testing the generator and ensuring the levels are generated as intended.
* When there are too many states to be decided through, this can sometimes slow down and reduce the efficiency of Markov Chains, however, this will not affect the project since there are not too many decisions to be made.

**2.2.4 – Summary**

After careful consideration and extensive research into the many procedural generation techniques available, the techniques being used in the project will be:

* Perlin Noise.
* Standard-order Markov Chains.

The next section will go into detail about how these techniques were adapted for use in the project.

# Chapter 3 Methodology

**3.1. – Overview**

This section will give an in-depth explanation of the practical project. It will show the planning done prior to implementation within the game engine, as well as how each stage was implemented, and why certain technical choices were made.

To answer the research question, the project created a procedural level generator which could be used to make levels for a 2D platformer using the Unity game engine. The project’s primary aim was to create a tool which would make procedural levels, then build this into a very simple 2D-platformer, and send this to participants who would then partake in qualitative tests to evaluate how this would affect their willingness to replay a larger scale 2D platformer which created levels in a similar way to this model’s output.

To complete these tests, participants would play the game for a short period of time, then fill out a survey to give their feedback on how they felt the game’s levels compared to those in other similar titles which they have played in the past.

**3.2. –Research**

**3.2.1. – Overview**

Before practical implementation could begin, some crucial decisions had to be made. These decisions included which engine or framework was to be used, and which coding methods and algorithms would be primarily used within the project.

**3.2.2. – Game Engine**

Since the project was to be built under a relatively tight time-constraint, the most important consideration when choosing a game engine was prior experience, as well as the ease of setting up a basic project which could be built upon. For this reason, three game engines were considered, each with their pros and cons. These 3 engines were:

* Unreal Engine 4/5.
* GameMaker Studio 2.
* Unity.

Unreal Engine was eventually ruled out. Whilst it does offer some very powerful features, and C++ is generally considered as a very fast, efficient programming language, it is not as easy to use for 2D projects. Unreal is often utilized for 3D games, and as such it does not come with as many 2D-oriented features. Although there are some examples of successful 2D games made with Unreal, such as Yoshi’s Crafted World [Nintendo, 2019], it was found to be cumbersome and difficult to set up a basic version of this project.

Another consideration was YoYo Games’ 2D-oriented engine; GameMaker Studio 2. As an engine exclusively used for 2D games, it is extremely simple to use and set up simple projects quickly and efficiently. The problems with this engine however stemmed from delving deeper into the project. Since GameMaker uses its’ own proprietary language, GML, it was significantly more difficult to find references and documentation from other users who have pursued similar uses of procedural generation.

The final option was Unity. Due to an extensive quantity of prior experience, it was known that Unity makes it relatively fast and easy to get a project up and running, as well as having very high-quality documentation, alongside a large community plentiful in code solutions to most issues. It also has built-in Perlin Noise functions, which would help massively during implementation, therefore Unity was the engine chosen for the project.

**3.2.3. – Implementation Research**

The final stage of research was to find practical examples of how to implement the key areas of the project’s build: Markov Chains and Perlin Noise. Whilst Markov Chains was largely logic based, and therefore didn’t require any direct tutorials beyond the prior research needed to understand how they function, Perlin Noise required slightly more technical solutions. As such, there were three key references which proved to be extremely helpful:

* The first of these was a video made on Brackeys’ YouTube channel called “PERLIN NOISE IN UNITY – Procedural Generation Tutorial” [Brackey’s, 2017].
* The next was a tutorial created by diving\_squid entitled “2D PROCEDURAL GENERATION TUTORIAL UNITY - PERLIN NOISE” [diving\_squid, 2021].
* The final point of reference came from Unity themselves, through their Unity Documentation pages[Unity].

**3.3. – Design and Planning**

**3.3.1. – Overview**

It was vital to ensure that there was a solid plan of how the project should run, before any code should be added. Without this design phase, there was a good chance that many features would have to constantly be reworked to allow for future features, and this would severely hinder the project’s chances of completion.

**3.3.2. – Passes Design**

To prevent this, the project was split into five separate scripts, referred to as “passes”. These passes would be run sequentially, with the next one running once the prior one has completed its’ script, and by using this method it allowed for a streamlined development process as each feature would run almost independently of other features, allowing maximum control over editing their outcomes during development without requiring changes to the entire project. This section will outline how each pass was designed, and how it was intended to be implemented.

**3.3.2.1. – Perlin Noise**

Before starting the main passes, it was decided that the Perlin Noise texture itself should be generated from a separate script. To do this, there were four steps outlined in the design plan:

* Generate a random seed for the new level and use this seed to set a random X or Y offset to give a new Perlin Noise map every time.
* Set the pixel colour of every pixel within the map using the Perlin Noise algorithm.
* Store this new map to be used later.
* Begin to run the passes.

**3.3.2.2. – Pass One**

The first pass was to be used to generate a base ground terrain for the level, based on the Perlin Noise values. At this point, the ground would still be solid and would exclusively be used to create hills or dips throughout the level. The steps for this are as follows:

* Scale every point within the noise map to a height value based on the darkness of the pixel at that point.
* Go from left to right across the level and add ground blocks up to the correct point in that position’s Y axis based on the height value at that point in the map.
* Ensure all points fall within the correct constraints, which are:
  + The ground level falls between the maximum and minimum Y values possible for the level. This will be the highest or lowest that ground *can* spawn.
  + Make sure the ground does not have any immediate height changes which will make it impossible for the player to reach the next block.
* Run the next pass.

**3.3.2.3. – Pass Two**

The second pass oversaw adding gaps into the terrain and spawning in platforms for the player to use to beat these gaps. This pass was the beginning of Markov Chains usage, and as such a ruleset had to be created which would modify the passing chance at each state of the chain. The steps for this pass are as follows:

* Go over every point in the level.
* Run the Markov Chain at every position to test if a gap should be made based on the rules and parameters from the ruleset.
* If the Markov Chain passes then remove all the ground blocks at the current X position, otherwise move on to the next position and try again.
* Once all gap checks have been completed, repeat the process for spawning platforms, using the separate platforms Markov Chains.
* If any chains pass for the platforms, then spawn a floating ground block at that position, at an appropriate height above the ground to make sure the player can reach it.
* Once all gaps and platforms have been checked, the third pass should run.

**3.3.2.4. – Pass Three**

The third pass was created to add enemies throughout the level. These were added to improve user engagement, since it requires the user to slow down or speed up to get past the enemies without touching them and dying. This pass again used Markov Chains to decide where and when the enemies should spawn, following these steps for the script:

* Check each position in the level.
* Randomly select one of the available enemies to attempt to spawn.
* Run a Markov Chain to decide if it should spawn.
* If the chain passes then spawn the selected enemy, if not then repeat the process at the same position for all other remaining enemies.
* Once either one of the enemies have been spawned, or the chain has failed for them all, then the chain should move onto the next position.
* Run the final script.

**3.3.2.5. – Pass Four**

The fourth and final pass adds collectable coins to the level. These were added to give the player a reason to explore different paths, such as jumping on platforms to collect coins even if it may not be required to complete the level. The steps for this pass are:

* Check each position in the level.
* Run a Markov Chain to decide whether to spawn a coin.
* If the chain passes, create a coin, otherwise move onto the next position.
* End the level generation portion and spawn the player into the level, then begin the game.

**3.3.2.7. – Rulesets**

Each of the Markov Chains come with a ruleset, which contains either constraints or chance modifiers, each of which were created specifically for this project. The constraints must be passed otherwise the chain will fail, and chance modifiers simply increase or decrease the chance of the chain passing its’ current state. These are summarised for each chain in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Creating Gaps** | **Spawning Platforms** | **Spawning Enemies** | **Spawning Coins** |
| The gap must not surpass a certain length (constraint).  There must be a minimum number of ground tiles before a new gap can begin (constraint).  The chance of a gap is increased when the surrounding area is flat (modifier).  The chance of a new gap beginning will increase for every ground block without one (modifier). | The speculative platform block must be a certain distance away from any ground blocks (constraint).  The platform must not surpass a certain length (constraint).  The chance of a platform is increased if the area being checked is over a gap (modifier).  The chance of another platform block being added decreases as the platform gets longer (modifier). | Spider enemies must have solid ground to spawn on (constraint).  The chance of a bat spawning is decreased when checking a platform block (modifier).  The chance of any enemy spawning is increased the more time that passes since the last one was spawned (modifier). | Sets of coins must be larger than a set value, and less than another set value (constraint).  The chance of a coin spawning is increased as more time passes without one (modifier).  The chance of coins spawning is increased when checking a platform block (modifier). |

**3.3.3. – Level Design**

When considering how the game should function, there were multiple features and mechanics which were outlined, which would decide what is classified as a successful level. These rules are as follows:

* Each level must contain:
  + Platforms.
  + Gaps.
  + Enemies.
  + Collectables.
* It is worth noting that the above requirements apply to a standard generated level, as the player does have options in the parameter menu which allows them to toggle these to their liking.
* When a player falls through a gap in the level, the level should be reset, and the player should restart.
* If the player collides with any enemies, the level should be reset, and the player should restart.
* If the player successfully reaches the end goal, they should win the level and a UI should appear which will tell the player they have won.

**3.4 - Implementation**

**3.4.1. – Generating Perlin Noise**

To generate a Perlin Noise texture, the PerlinNoiseGeneratorScript.cs script is run as soon as a new level is being generated. First, in the start function, the script uses the C# function Random.Range() and generates a seed value between 0 and 9999. From here, an X and Y offset are created using the seed value, and this will change where to start sampling noise from within the texture, meaning a different noise map is given for each level, resulting in new terrain.

Next, the CreatePerlinNoise() function is run, which can be summarized with the following pseudocode:

*CREATE 2D texture AS noiseTexture*

*FOR EACH xPosition FROM 0 TO textureWidth*

*FOR EACH yPosition FROM 0 TO textureHeight*

*SET currentPos EQUALS ((xPosition / zoom) + xOffset), ((yPosition / zoom) + yOffset)*

*SET pixelColour AT currentPos TO Mathf.PerlinNoise()*

*RUN noiseTexture.SetPixel(pixelColour)*

*END FOR*

*END FOR*

*RUN noiseTexture.Apply()*

*SET BOOL runNextScript TO true*

**3.4.2. – Converting Noise to Height**

Before the terrain starts to be randomly generated, a short flat platform is created at the beginning of the level, to ensure the player will not spawn inside the ground. The same is also done for the end of the level, to ensure the win zone can generate correctly, and these platforms are created using the following simple method:

A screen shot of a computer program

AI-generated content may be incorrect.

This works by simply running an if statement which checks to see if the current X coordinate being checked is too close to the start or end of the level to be randomised, and if it is then it will manually set the height of the ground instead of sampling the Perlin Noise to calculate it procedurally. This will then cause ground blocks to only be placed up to this height, and if the X position is within the end zone range, then an end zone block will also be placed up to the maximum height of the level, allowing the player to jump into it.

Assuming the X coordinate being checked is a standard position, not in either of the set platforms, the direction that the ground should go at this point is calculated using the following code:

A screen shot of a computer

AI-generated content may be incorrect.

Breaking this down:

1. By dividing the current X position being checked by the width of the level, this normalizes the position within the level to be between 0 and 1. Multiplying this by the width of the Perlin Noise texture will then result in a float value which is the position of the level’s equivalent pixel in the texture. Mathf’s FloorToInt function converts this result to an integer, which is useful since each ground block has a width of one unit within the level, meaning when checking X positions the program always moves up in increments of one whole unit.
2. By using the grayscale function on this pixel’s colour, a float value between 0 and 1 is returned based on the pixel’s shade, with 0 being entirely white, and 1 being entirely black, then anything in between is getting gradually darker as the value gets nearer to 1.
3. The final line in this section then converts this noise value to a direction, which will decide whether the current ground block should move up, down, or stay at the same Y level as the previous block.

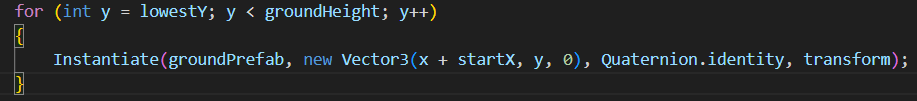
Now that the program has decided whether to go higher or lower, the new ground height is calculated using the following calculation:



* (highestY – lowestY) / 2 will return the half of the total possible range of Y positions, which results in the distance which the block can move from the middle of these values.
* Multiplying this by the ground direction decides where to move up or down to, based on the direction calculated before.
* (highest + lowestY) / 2 puts the newfound position central again in the range, giving the final Y position.
* The code uses FloorToInt because all ground blocks are one unit by one unit, therefore any positions should be converted to full integer values.

The final part of the calculation is as follows:  


This line is simply using the Mathf Clamp() function to ensure that the new ground height does not move up or down too much from the height of the previous block, which is how the model prevents impossible terrain that the player cannot jump over.



This final snippet then spawns ground tile objects from the bottom of the level up to the appropriate level using the values calculated above.

**3.4.3. – Using Markov Chains**

From the second part onwards, the scripts all use a very similar layout, where only the parameters change but the logic behind the functions remain the same. The Markov Chains are run using the following steps:

1. An enum is created which will store each possible state for the chain.
2. This state is used to start a switch statement for each block in the level.
3. During every state, the progression chance is updated using appropriate parameters for the current state.
4. This new progress chance is passed into a function called MoveStages(), which will generate a random float between 0 and 1 to compare the progression chance to, turning the progression chance into a percentage.
5. If the value generated in the function is lower than the progression chance, the current state then becomes the next state in the enum.
6. This process is repeated until the chain either passes all stages, or fails a check, at which point the switch statement is exited and the next block in the level will be checked.

The MoveStages() function is as follows:

A computer screen shot of text

AI-generated content may be incorrect.

This works by getting a float between 0 and 1, then storing this as the value to compare the chance parameter to.

An example of one stage of the Markov Chain can be found below. This example is checking how flat the ground is when choosing whether to create a gap:

A computer screen with text on it

AI-generated content may be incorrect.

To simplify this function, it will increase the pass chance for every block nearby which is at the same Y-level as the block being checked, then lower that chance for every block which is higher or lower, within the checking range. This new value is then passed into the MoveStages() function, and if it fails the comparison then the switch statement is exited, which will move to the next block. If it passes however, the current state changes, and the switch statement is run again, this time running as a different state, which will use similar logic to check how long it has been since a gap was last created.

**3.4.5 – Adding Parameters**

Players can edit the parameters of future levels using the following window, found in the pause menu:  
A screenshot of a video game

AI-generated content may be incorrect.

Each slider within the menu will return an appropriate value to the scripts, based on how far along the player has dragged the slider. Each slider uses the value as stated below:

* Level length will change the endX variable in the first pass, which will cause the for loop within this script to run for more iterations, meaning more ground blocks will be created at the end of the level.
* Flatness will edit the zoom variable when generating Perlin Noise. By zooming in, there is less room for the pixel colours to vary, resulting in less changes of height throughout the level. Zooming out will show a larger area of noise, which will result in more room for variation, therefore more variation in ground height throughout the level
* Gaps and platforms are changed by simply multiplying the progression chance in each stage of the Markov Chain when in the MoveStages() function. By multiplying the chance by a higher value, the chain is more likely to pass all stages and generate the gap or platform, whereas lowering this constant in the multiplication will have the opposite effect, causing less to be generated.
* The maximum enemies and coins work by adding a stage at the start of their respective Markov Chains which would check if generating a new object would surpass the limit for the level, and if it would then the chain would instantly fail, meaning nothing would be created.

**3.5. – Testing**

To test the effectiveness of the level generation model, multiple testers, recruited through the university or through friends and family, were given access to a build of the project, then asked to playtest as many levels as they required to get a good idea of the game, then leave feedback on how they felt about its’ output.

The survey was hosted using Microsoft Forms and included questions which focused around comparing the possible replayability to that of other games. It is important to note that due to the small scale of the project, it was not possible to directly ask players if they felt the game was replayable, since the game simply did not include enough features to make this viable. It is for this reason that the questions centered around the idea of the model being implemented into a larger scale game with a more complete feature set, and the testers were asked to evaluate the *level generation* as opposed to the game overall.

Referring to Appendix A, the first three questions were designed to get an idea of the tester’s background, to see how they felt about 2D platformers which are currently available. This would be helpful when considering answers given about the project, as it would show if the project changed their views on platformers in general, whether positively or negatively.

The next four questions were directed more towards the project specifically. They are designed to see how the tester felt about the project as a standalone model, which would help to judge the model’s success in creating levels which are playable for a standard player.

The final three questions focused on asking the player to compare the model’s levels to those of platformers which they have played in the past. This section was especially helpful in testing the success of the project. Question 9 is a good evaluation of how the player feels about engagement in a procedural level, which was outlined as a main objective within the project, and when combined with question 10, helps to answer both sections of the research question, focusing around using procedural generation to improve both playtime and player engagement in 2D platformers.

# Chapter 4 Results

**4.1 – Overview**

To measure how successful the project was at reaching its’ goals, the testing was carried out using a qualitative method. This was difficult, since it is hard to measure replayability over a short period of time. Therefore, testers were asked to talk in theory about how the project *could* improve replayability, to get an idea of how they felt the model’s structure could improve future gameplay.

Testers were given a link to a Microsoft Form to fill out, and all graphs are generated from Microsoft Forms, except for the sixth and seventh question since this was an open-ended answer as opposed to multiple choice, therefore the chart had to be created manually after analysing the answers. The results of all questions can be found below. Note, the first question in the form regarded consenting to the survey, therefore the “first” question is listed as 2. in the chart, etc.

**4.2 – Tester Past Experience Results**

The first section of the questionnaire was designed to get an idea of how much experience testers have playing platformers in the past, and how they felt about the levels within them. The results from the first question shows that all testers would agree that they have spent time in the past playing 2D platformers:

A graph with colorful lines

AI-generated content may be incorrect.

The second question showed more divided answers. Most testers said they did not regularly go back for a second playthrough of platformers after initially completing the game, whilst two felt agreed that they do:

A graph with purple and white lines

AI-generated content may be incorrect.

The third question showed that most players do feel 2D platformers generally have varied and unique levels, whilst one participant feels this was not the case:

A graph with colorful bars

AI-generated content may be incorrect.

**4.3 – Project Level Quality Results**

The next section of the questionnaire focused on asking players about their experience in the project.

The results of the fourth question showed that overall, players did feel that the level generation model was able to output unique and interesting levels:

A graph with multiple colored lines

AI-generated content may be incorrect.

The results from the fifth question showed that no players found any repeated levels:

A screenshot of a computer

AI-generated content may be incorrect.

The sixth and seventh questions showed that players played various numbers of levels, and most completed every level generated, whilst the others completed over half of the total levels generated. Note that this chart was made using Online Visual Paradigm [Visual Paradigm]:

A graph with numbers and a bar

AI-generated content may be incorrect.

**4.4 – Tester Comparison Results**

The final section of the questionnaire allowed testers to compare how they felt about the project with how they felt about other platformer games which they have played.

The eighth question showed that all testers did feel the project’s use of procedural level generation improved variation in the levels, when compared with the levels of other 2D platformers that they have played:  
**A screenshot of a graph

AI-generated content may be incorrect.**

The ninth question showed that testers felt the project’s levels improved their engagement within the game in comparison to the other 2D platformers they have played before:

**A screenshot of a graph

AI-generated content may be incorrect.**

The results of the final question shows that overall, the majority of testers feel they would be more likely to go back and play another playthrough of a 2D platformer if it used procedural generation. However, one tester felt that doing so would not have an effect on their original opinion of 2D platformers:  
A screenshot of a graph

AI-generated content may be incorrect.

# Chapter 5 Discussion

**5.1 – Overview**

This section will discuss and analyse how successful the project was in achieving its’ aims. The focus will be on the following sections:

- An analysis of how well the level generation model was in the proposed design stage.

- An analysis of the success of implementing this design into the final project.

- An analysis of the testing methods used, as well as how the feedback received by the testers proved the success or failures of the project.

The discussion of the design stage will be used to highlight how well the project was planned initially, including how strongly it related to the project’s aims and objectives, as well as how reasonable the scope of the project was for the timeframe provided.

The discussion of the implementation will highlight how well the final application matched the original design, as well as any issues or shortcomings found whilst implementing the program.

Finally, the discussion of the testing will highlight how effective and accurate the testing method was, including the quality of participants chosen and the effectiveness of the questions used in relation to the feedback expected. It will also discuss how the feedback from the participants was useful in evaluating the overall success and failures of the project in relation to aims, objectives and the research question outlined.

**5.2 – Design State Discussion**

As a whole, the application’s design was successful and carried the overall project well. Importantly, the scope was kept reasonable, and at each point in the project the features to be added were reconsidered.

By designing the application to use passes, which were effectively stages layering on top of one another, this ensured that the project was always in a playable state, which is an important step in ensuring the application could be tested at every stage, with iterations to previous stages being done as required.

Having the features laid out before starting implementation was also a key step, as this helped to prevent feature creep, which may have resulted in the project going outwith the scope, resulting in development not being completed by the deadline. It also ensured there was always a solid next step for development, which made sure each implementation session was efficient and had a clear goal.

The decision to use Markov Chains and Perlin Noise was good, as they were extensively researched prior to beginning the application, and this helped to give a reference to look back on when evaluating implementation methods to ensure the application output was matching what was expected from the design.

A key document which was created during the design stage, and turned out to be extremely helpful throughout development, was a Gantt Chart, shown below:

A screenshot of a video game

AI-generated content may be incorrect.

This helped to structure the order of development, and allowed for a visual representation of how well the project’s time management was being kept. The chart was reviewed at the beginning of each week, and a decision was made as to whether development could continue as planned, or if any non-essential features needed to be removed to ensure the project would be completed on time. As visible in the chart above, features were marked in green as they were completed, and features which were eventually removed were marked as red. The removed features will be discussed in more detail during the Future Work section of this paper.

**5.3 – Implementation Discussion**

Overall, the implementation of the project was a success. All features which were required to classify a successful level, as outlined in the Literature Review section, were successfully generated in the project, therefore the model can overall be classed as successful from an implementation view.

The implementation used an iterative coding method, where regular testing was carried out during the implementation, and would sometimes lead to reconsidering some design choices. For example, the original design included a fourth pass, which would have an AI player model run over the level before starting the game and would only allow the level to be used if it managed to complete it. After some time spent attempting to implement this however, it was found to be too complex for the project’s scope and time allowance, therefore it had to be cancelled. As a result of this, an iteration was required which would more tightly refine how certain aspects of the level was generated, such as adding a constraint to ground generation which only allowed the Y-level of the ground to change by one block at a time. This improved the chance of generated levels being possible, and therefore somewhat omitted the need for a fourth pass.

The biggest benefit of the iterative coding method was having the ability to constantly go back and edit past stages as required, since often adding a new stage would highlight issues in the previous stage.

Of the features outlined in the design stage, only three major additions were eventually omitted:

* The enemy pool was reduced from three to two.
* The fourth pass involving an AI tester was removed.
* The ability to zoom and move the camera around the level was removed.

The reasoning behind these features being removed will be outlined in the Future Work section.

**5.4 – Testing Stage Discussion**

Overall, the testing phase conducted was very effective in receiving useful feedback from the participants, in relation to the project’s overall objectives, specifically:

* To research how procedural level generation could be used in future 2D platformers to give the player a unique experience each time they start a new game.
* To evaluate the level and length of player engagement in a classic linear platformer when compared to one featuring procedural level generation techniques.
* To explore the links between new, unique levels and player engagement levels.

**5.4.1 – Participants**

Participants were found by requesting help through friends and family, as well as fellow students. This allowed the project author to consider who would be suitable to participate in the project as they were known prior to carrying out testing, which is shown from the results of question one of the surveys, where all participants outlined that they had prior experience in 2D platformer games. This was a benefit as the project required a comparison between the project itself and other 2D platformer games, and by ensuring all testers had prior experience, it would prevent outlying results where testers may not have had any other games which they could compare the project to.

This method of recruiting testers did come with some drawbacks however, primarily limiting the survey to a very small sample size. In the end, only five testers took part in the survey, and whilst this was enough to get a good idea of the success within the project’s scope, having more testers would have given a more reliable source of feedback. Another potential issue was that whilst all participants did have experience with 2D platformers, it was not known which specific games this included, and due to the wide range of games within the genre, the games may not have been as closely related to the project as expected. For example, a player who played Super Mario Bros [Nintendo] may not have experienced as many PCG levels as someone who played Spelunky [Derek Yu, 2008].

**5.4.2 – Survey Effectiveness**

In general, the survey was well received by participants, with each tester choosing to answer every question, despite none being mandatory. This can be attributed mainly to the careful consideration during the planning of the questions included, where there was a focus on making them as easy to answer as possible. The questions mainly required the tester to read a statement, then decide how strongly they agree or disagree with the statement.

By using Microsoft Forms, this also made it easy to send out links to the survey to participants, and setting the survey visibility to public meant the testers were not required to sign in prior to completion, making it easy again for testers to give their feedback. Forms also format answers automatically into bar charts, which allowed for an easy overview of feedback results for evaluation.

One issue with the question design was found during the evaluation of results. At the end of the second section, users were asked to compare how many levels they generated, and how many they completed. The issue came when two users said they left some levels incomplete. This caused a problem because there was no question which asked the testers why they failed to complete some levels, so it is unknown whether they generated impossible levels, or if there was another reason why they did not finish the level.

**5.4.3 – Feedback Evaluation**

**5.4.3.1 – Tester Past Experience**

As mentioned in 5.4.2, the first question showed that all users would agree that they have got experience in playing 2D platformers. This was the expected result, and it was important as it ensured testers would be able to make a reasonable comparison to these games to answer questions later in the survey.

The second question showed a more divided split, where three users said they do not often play platformers more than once, whilst two said that they do. This was a good outcome because it allowed for getting the point of view of two types of players; those who feel current platformers are not currently replayable, and those who feel some might have a good replayability aspect. This meant the project would be considered both as adding replayability to a genre which does not currently have it, as well as how it possibly improves and build upon features which already do.

The final question of this section showed that four of the five testers felt the games they played had varied and interesting levels, whilst one said they did not feel this way. Similar to the prior question, this was helpful as it allowed for a split between players considering if the project improves further upon this idea, or if they feel it fixes an issue found in current games.

**5.4.3.2 – Project Quality**

The first question of this section asked testers if they felt the model generated varied and interesting levels, and was met with an overwhelmingly positive response, with all users agreeing that it did. This shows that the project succeeded in the third objective outlined in the introduction, specifically “to utilise programming techniques such as Perlin Noise and Markov Chains to implement a procedural level generator using the Unity game engine which effectively generates interesting and varied levels”, as the feedback showed the project managed to achieve this very effectively.

The next question asked testers if they found the same level twice at any point, and all testers agreed they did not. This showed the first objective “To research how procedural level generation could be used in future 2D platformers to give the player a unique experience each time they start a new game” was a success, as the generator was able to generate a new level every time, so it can be presumed the player would not encounter the same level across multiple playthroughs.

The final two questions of this section asked users how many levels they generated, compared to how many were completed. Across all users, a total of forty-four levels were generated, forty of which were completed. Whilst this was an excellent result, as mentioned in 5.4.2, it is not confirmed why four were left completed. From these results, it can be concluded that players felt engaged enough to complete the vast majority of levels, however knowing why some were not completed would have helped with the strength of results evaluation. As a result of this oversight, it cannot be confirmed whether the project requirement “all levels should have a possible path to the goal zone” was achieved.

**5.4.3.3 – Project Comparison**

The first question of the final section asked testers to compare the project levels to other platformers. The response was entirely positive, which shows all users felt that the model did manage to make levels which were more unique than current platformer games. This shows that, as mentioned in the introduction under research question, “By creating terrain generated using Perlin Noise, as well as populating the levels with objects using Markov Chain decisions, this will create unique and diverse levels” was a success.

The next question asked if users felt the model improved their engagement compared to other 2D platformers, This was again met with positive responses, which shows the project did succeed in one of its’ primary objectives, which was “To evaluate the level and length of player engagement in a classic linear platformer when compared to one featuring procedural level generation techniques”, as it did improve the level of player engagement by procedurally generating levels.

The final question asked if players felt the model would encourage them to play through a platformer multiple times. This was met with primarily positive responses, with four of the testers agreeing, however one tester said they somewhat disagree. Due to the small sample size of testers, it is difficult to tell if this was a single outlying result, or if increasing the number of testers would show a more common trend of players disagreeing, but from the results provided it can be assumed that procedural levels would largely improve replayability. From here, it can be concluded that the project achieved the second part of the aim “consider how this could be used to lengthen the time a player can stay engaged with the game”, as it shows most players would be engaged for longer using this method of generation.

**5.4.3.4 – Summary**

In summary, from the feedback the research question “How can procedural level generation be used to lengthen players’ playtime and improve player engagement in 2D platformer games?” has been answered. The first section, considering player playtime, was answered in the survey’s final question, where the majority of testers agreed the project’s implementation would help to encourage them to replay 2D platformers. The second section is answered in the penultimate survey question, where testers all agreed that the procedural generation model did improve their engagement over their past experience in 2D platformers.

# Chapter 6 Conclusion and Future Work

**6.1 – Research Question and Aim**

In conclusion, the research question of the project was asked to discover if procedural level generation could assist in improving player engagement levels, and replayability aspects, across 2D platformer games. This question was decided upon as it would result in players of the genre having improved satisfaction whilst playing their games, and would also improve the amount of content which is available for players, without putting unreasonable workload onto members of the development team.

As a whole, the project was successful in showing that a procedural level generation model could be used to improve player engagement and game replayability, as shown through the user feedback given during playtesting.

Overall, the project also managed to achieve its’ primary aim. During the research phase, the benefits of PCG algorithms were thoroughly investigated and considered, then, during the testing phase, the results of implementing these algorithms in the context of a 2D platformer were analysed and showed largely positive results in relation to player engagement.

**6.2 – Future Work**

Whilst the project was an overall success, there is still plenty of room for future additions and improvements. These potential future work areas were found during all stages of development and include some features which had to be omitted from the original application design due to time constraints, as well as ideas which were identified during the feedback review and analysis stages.

**6.2.1 – Omitted Application Features**

This section will cover features specific to the application which were originally included in the project design, but never made it to the final application. The section will discuss what the feature was, why it was cancelled and why it would add to the application overall.

The first feature which was omitted was the idea of allowing players to scroll through the level and zoom in or out to see more or less of the level. This was intended to be the final feature added, however this came across as a feature which would be more suited for a final game, as opposed to a tool designed for game developers, and the application was intended to lean towards the latter. This feature would however be a useful addition if the model were to be applied in a full-scale game, as it would allow players to get a rough idea of what to expect in the level before starting to play.

The next feature which was omitted was adding a fourth pass into the model which would use an AI version of the player to test if the level was possible. A significant attempt was made during the implementation stage to add this feature, however it was quickly discovered that the complexity of adding this pass would far surpass the timeframe dedicated to the project overall. This feature would be a major improvement to the application in the future however, as it would add a method to ensure players would never encounter an impossible level, which was a requirement of the project’s success. With the current state of the application, some freedom had to be removed from the generator to ensure all levels generated would be possible, but by using a testing model, this would allow constraints and rules to be loosened slightly on other passes, allow the model to potentially create even more interesting and varied levels for the player.

**6.2.2 – Feedback Improvements**

The biggest oversight during the feedback stage was the lack of a question which asked players why they did not complete some levels. If this project were to be redone in the future, this would be an extremely useful addition to the questionnaire, as knowing the reasoning behind players generating more levels than they have completed would help to evaluate if the model was generating impossible levels, which would be an area that would need to be iterated on and fixed appropriately.

Another key stage of testing which would have been beneficial for the project would be a section of quantitative feedback. This could include asking the testers questions such as how long loading times were prior to levels being generated, and whether certain features worked as intended, such as the parameter menu. This was not added to the original project as the author felt the project had been self-tested extensively enough to be confident that loading times would not be an issue, and all features were found to work as intended. This may not be the case if the project were to be scaled up with the additional features listed about however, and in this case quantitative results may be crucial for user-testing.

**6.3 – Final Summary**

Overall, the project was a success. With an increased timeframe and team size the scope could be increased, which would allow for an even stronger output, however in relation to the resources and scope of the current project, the output worked as intended and originally designed.

In the future, it is hoped that the research and examples provided in this project can help developers and studios to apply these ideas into their own games. This may help them to improve the 2D platformer genre, by creating vast amounts of new and interesting content for their players which may not be possible without the use of procedural level generation.

# List of References

Wikidot.com. (2025). *Procedural Content Generation Wiki - Procedural Content Generation Wiki*. [online] Available at: <http://pcg.wikidot.com/> [Accessed 12 October, 2024].

Snodgrass, S. and Ontanon, S. (2014). Generating Maps Using Markov Chains. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, [online] 9(2), pp.25–28. Available at: <https://doi.org/10.1609/aiide.v9i2.12586> [Accessed 16 October, 2024].

Records, G.W. (2023). *Best-selling videogame*. [online] Guinness World Records. Available at: [https://www.guinnessworldrecords.com/world-records/best-selling-video-game?os=0&ref=app](https://www.guinnessworldrecords.com/world-records/best-selling-video-game?os=0&ref=app%20) [Accessed 11 March, 2025].

Saisuman Revankar (2025). *Minecraft Statistics By Player Demographics, Usage, Sales and Revenue*. [online] Sci-Tech Today. Available at: <https://www.sci-tech-today.com/stats/minecraft-statistics/> [Accessed 15 March, 2025].

Dawnosaur (2023). *How Minecraft generates Worlds you want to explore*. [online] Design with Dawnosaur. Available at: <https://dawnosaur.substack.com/p/how-minecraft-generates-worlds-you> [Accessed 16 March, 2025].

Zucconi, A. (2022). *The World Generation of Minecraft*. [online] Alan Zucconi. Available at: <https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/> [Accessed 16 March, 2025].

Minecraft Wiki. (Unkown). *World generation*. [online] Available at: <https://minecraft.wiki/w/World_generation> [Accessed 16 March, 2025].

Tinysubversions.com. (2013). *Spelunky Generator Lessons*. [online] Available at: <https://tinysubversions.com/spelunkyGen2/> [Accessed 2 February, 2025].

Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), pp.287–296. Available at: <https://doi.org/10.1145/325165.325247> [Accessed 17 March, 2025].

Seneta, E. (2006). *MARKOV AND THE CREATION OF MARKOV CHAINS*. [online] Available at: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4c6bb41637a6438f22eb190d014a01cd0b0a7162> [Accessed 17 March. 2025].

Norris, J.R. (1997). *Cambridge Series in Statistical and Probabilistic Mathematics*. [online] Google Books. pp.1-6. Available at: [https://books.google.co.uk/books?hl=en&lr=&id=qM65VRmOJZAC&oi=fnd&pg=PA3&dq=markov+chains+&ots=0KyNVsy-2b&sig=Z4XFCbMw1Q9CWlgaimZ7bbSTcTA&redir\_esc=y#v=onepage&q=markov%20chains&f=false](https://books.google.co.uk/books?hl=en&lr=&id=qM65VRmOJZAC&oi=fnd&pg=PA3&dq=markov+chains+&ots=0KyNVsy-2b&sig=Z4XFCbMw1Q9CWlgaimZ7bbSTcTA&redir_esc=y%23v=onepage&q=markov%20chains&f=false%20) [Accessed 17 March. 2025].

Brackeys (2017). *PERLIN NOISE in Unity - Procedural Generation Tutorial*. [online] YouTube. Available at: [https://www.youtube.com/watch?v=bG0uEXV6aHQ&list=PLS9pgTRs4XDvmyr0WYNx1b89fcbfhSyLO&index=1](https://www.youtube.com/watch?v=bG0uEXV6aHQ&list=PLS9pgTRs4XDvmyr0WYNx1b89fcbfhSyLO&index=1%20) [Accessed 31 September. 2024].

diving\_squid (2021). *2D PROCEDURAL GENERATION TUTORIAL UNITY - PERLIN NOISE*. [online] YouTube. Available at: [https://www.youtube.com/watch?v=puLpZIAGAcM](https://www.youtube.com/watch?v=puLpZIAGAcM%20) [Accessed 31 September. 2024].

Visual-paradigm.com. (Unkown). *Visual Paradigm Online*. [online] Available at: <https://online.visual-paradigm.com/drive/#proj=0&chart=list> [Accessed 14 April, 2025].

Unity (2025). *Unity Documentation*. [online] docs.unity.com. Available at: <https://docs.unity.com/> [Accessed 24 September, 2024]

# Bibliography

McDonald, N. (2019). *Markov Chains for Procedural Buildings – Nick’s Blog*. [online] Nickmcd.me. Available at: [https://nickmcd.me/2019/10/30/markov-chains-for-procedural-buildings/.](https://nickmcd.me/2019/10/30/markov-chains-for-procedural-buildings/)

Smith, G. (2014). Understanding procedural content generation. *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI ’14*. Available at: <https://doi.org/10.1145/2556288.2557341>.

Shaker, N., Togelius, J. and Nelson, M.J. (2016). *Procedural Content Generation in Games*. *Computational Synthesis and Creative Systems*. Cham: Springer International Publishing. Available at: <https://doi.org/10.1007/978-3-319-42716-4>.

Itch.io. (2024). *Spinning Pixel Art Coin*. [online] Available at: [https://thepeeps191.itch.io/spinning-coin](https://thepeeps191.itch.io/spinning-coin%20) [Accessed 30 September, 2024].

Itch.io. (2020). *2D Pixel Art Spider Sprite*. [online] Available at: [https://elthen.itch.io/2d-pixel-art-spider-sprites](https://elthen.itch.io/2d-pixel-art-spider-sprites%20) [Accessed 30 September, 2024].

itch.io. (2018)*Bat Sprite Pack by Elthen’s Pixel Art Shop*. [online] Available at: <https://elthen.itch.io/bat-sprite-pack> [Accessed 30 September, 2024]..

# Appendices