**Methodology Draft**

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.

**3.1. – Overview**

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](https://www.youtube.com/watch?v=bG0uEXV6aHQ&list=PLS9pgTRs4XDvmyr0WYNx1b89fcbfhSyLO&index=1)]. In this video, Brackeys goes over how to generate a Perlin Noise texture within Unity. Brackeys’ solution was slightly adapted then used in the final project.
* The next was a tutorial created by diving\_squid entitled “2D PROCEDURAL GENERATION TUTORIAL UNITY - PERLIN NOISE” [diving\_squid, 2020]. This tutorial was vastly useful as it explained how to take the generated Perlin Noise and utilize some of Unity’s Mathf functions to convert points on the Noise map to create a height map for the level. The examples used in this video inspired the script used for the first pass of the project.
* The final point of reference came from Unity themselves, through their Unity Documentation pages. Through these pages, a more in-depth explanation of their functions can be found, and this was later combined with the other two tutorials to help with adapting the examples to better fit the context of the project.

**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.

**3.3.2.6. – Pass Five**

During the design planning, a fifth pass was intended to be added. This pass would have generated an AI version of the player, which would attempt to complete the level to ensure it was possible. After attempting to implement this however, it was found to be too large for the scope of the project, as it would have required a large time-investment into researching AI pathfinding within Unity to teach the model how to complete levels, and it was not possible to complete this pass within the timeframe of the project.

**3.3.2.7. – Rulesets**

As mentioned, each of the Markov Chains come with a ruleset, which contains either constraints or chance modifiers. 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 outlines, 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 works using the following steps:

* Creates a new blank 2D texture which will store the generated map.
* Runs a for loop which runs over every X co-ordinate from 0 to the width of the texture, then within this is a nested for loop which runs over every Y co-ordinate up to the height of the texture.
* Gets the correct current position by dividing the X and Y values from the loop by the zoom value, then adding on the offsets created at the start to move to the correct pixel within the texture based on the offset.
* Using Unity’s built in Mathf.PerlinNoise() function, which returns a grayscale value based on the Perlin Noise algorithm, the program stores the value which is returns and passes this as a parameter to the texture’s SetPixel() function, which will now set the pixel colour of the current position in the texture to be the returned value.
* The texture is then updated using the Texture.Apply() function, and the Boolean which tells the next pass when to run is set 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. The next line then uses this returned value and passes it into the GetPixel() function, which will return the colour value at the pixel being checked in the format (Red, Green, Blue).
3. By using the grayscale function on this colour value, 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.
4. 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:

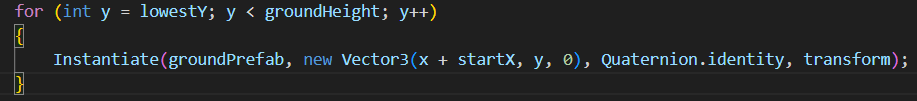


* (highestY – lowestY) / 2 will return the half of the total possible range of Y positions, which results in the distance which the block can move from the middle of these values.
* Multiplying this by the ground direction decides where to move up or down to, based on the direction calculated before.
* (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.

Now that the ground height at the current X position in the level has been calculated, the following function is run to place the ground into the level:



This function is running over every Y position from the lowest point in the level, and placing a ground object at that position, up to the ground height which was calculated above. This process is then repeated for all appropriate X positions in the level, then the first pass script will end and will update the pass two Boolean to allow it to run.

**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 using Unity’s Random.Range() function to get a float between 0 and 1, then storing this as the value to compare the chance parameter to. This converts the chance to a percentage because, if for example, the chance is passed as 0.8, the random generation value would have to generate between 0.81 and 1.0 to fail, meaning there is a 20% chance for it to fail, and an 80% chance to pass.

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 add an 8% chance for every block nearby which is at the same Y-level as the block being checked, then will subtract 10% from 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, increasing its’ length.
* 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 a lot more changes in height throughout the level, which will help to generate more hills.
* 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 were much simpler to implement. This required 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 were given access to a build of the project, then asked to playtest and 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.

The first three questions in the survey were as follows:

1. I have experience playing 2D platformers in the past.
2. I often go back to replay a 2D platformer after my first playthrough.
3. I feel levels in 2D platformers which I have played in the past contained unique and varied levels.

These 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 as follows:

1. I feel the levels in this project were varied and interesting.
2. Did you encounter the same level twice at any point whilst playing the project?
3. How many levels did you generate whilst testing?
4. Of these levels, how many did you successfully complete?

These 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 were:

1. I feel the project generated levels which were more varied and unique than the ones I have played in other 2D platformers in the past.
2. This style of procedural level generation would make me feel more engaged in the game than a traditional 2D platformer.
3. Overall, I feel this method of procedurally generating levels in a 2D platformer would encourage me to start a second playthrough after initially completing the game.

These final 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.