| Unity Game with procedural level generation using a custom Perlin noise algorithm

# Introduction

In order to showcase an AI system for this project, a relatively simple game was chosen with easy-to-understand controls and mechanics. The game is made in the style of an endless runner, in which a ship is controlled either left or right by the player to dodge obstacles on an infinitely generated level. The AI system chosen to mirror this effect and produce a similar but unique result was procedural generation with Perlin noise.

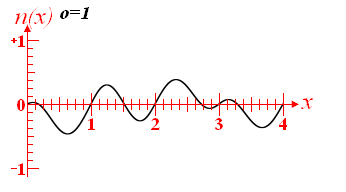
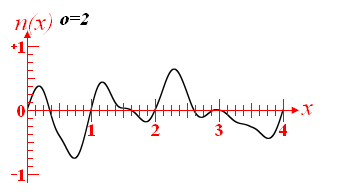
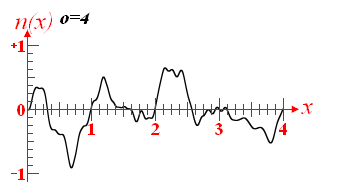
The goals set in place for this AI system was to have a randomly generated level in the form of a mesh, with a height map, that would have drastic ‘spikes’ in the terrain which would pose as the obstacles for the player to dodge. This level would be infinite using a chunk-based system that only generates parts of a level that are in the player’s line of sight, deleting other parts of a level the player cannot see in order to keep the computational requirement low and increase efficiency. The level should generate in a way that is realistically completable and avoid creating paths that are impossible to manoeuvre through. If it cannot do this, there may be points in the game that randomly become impassable and be frustrating to try and play through as well as making the AI system seem less complex and not well made. However, in the limited time given to create this game, not all these features were fully implemented, although the core aspect of the AI system is present and working.

The way this game was developed means that the AI system needs to have several requirements in order to work fully and provide the best gameplay experience, such as customisation and unpredictability. In this project that was developed, the system is also controllable in the Unity editor.

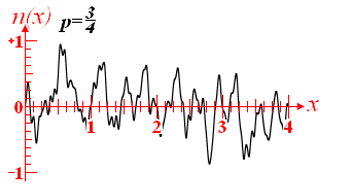
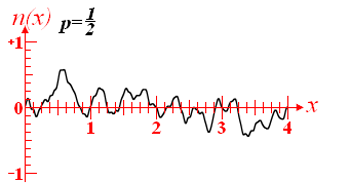
The system chosen for this game was Perlin noise. Perlin noise works by producing a smooth sequence of pseudo-random numbers – it does this because truly random numbers would produce erratic effects that would not look smooth or ‘natural’. In Unity, this is created by assigning a pattern of float values from 0.0 to 1.0 across a 2D plane, which is then assigned a colour from black to white. Instead of just being a random value between these points, it produces a ‘wave’ that gradually increases and decreases the value, making the noise more coherent.

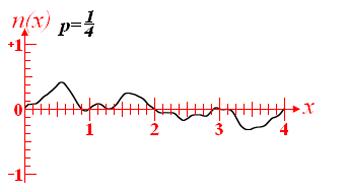
This method was chosen for this project because it is a very widely used method of procedural generation and with the way the game works, this method is the most effective at generating the intended effect. As well as this, the time taken to implement this method would be faster than other methods due to its understanding and support. This method used in conjunction with the game is not computationally expensive, and as such the runtime requirements can be kept low to suit many system types.

# Implementation

The implementation of Perlin noise in unity can take many forms. The Unity scripting API has a built-in library which was used, but a lot more work around this was done to produce the unique effect needed for the game. Since Perlin noise produces a ‘wave’, it can be controlled on the X and Y axis by increasing or decreasing the amplitude and the frequency. In order to create Perlin noise that has more detail and coherence, multiple levels of noise can be layered several times, known as *octaves,* which when added together can create noise that is moresuited to terrain generation. To increase the frequency of noise of these octaves layered together, a multiplier is used called *lacunarity.* As this value increases, the amplitude and frequency of each octave will rapidly increase. To create more natural and smooth generation, a value called *persistence* is used to control the decrease in the amplitude of the octaves.

*Figure 1 (above): Graphs showing how increasing the lacunarity affects the production of noise. From* [*http://libnoise.sourceforge.net/glossary/index.html*](http://libnoise.sourceforge.net/glossary/index.html)

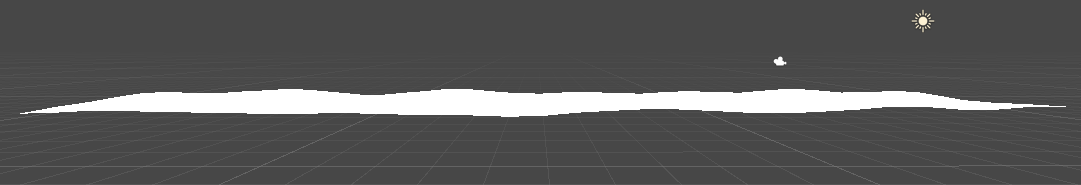




*Figure 2 (above): Graphs showing how the persistence value affects the production of the noise. From* [*http://libnoise.sourceforge.net/glossary/index.html*](http://libnoise.sourceforge.net/glossary/index.html)

The above diagrams illustrate how these variables all contribute to generating a terrain map that is natural and smooth while still being randomly generated. These values also allow for the aspect of customisation to make this system unique for this game.

Mentioned before, Unity’s built-in API for Perlin noise was used (*Mathf.PerlinNoise)* with the system developed for this game, since a lot of the fundamental mathematics behind Perlin noise is used in that function and as such it is much more time effective and easy to include it. In this project, it should be noted that the output of this function was edited to further increase the ‘realistic’ nature of the noise generated – this was done by multiplying the value by 2 and decreasing it by 1. This lets the Perlin noise function output a value from -1 to 1, instead of 0 to 1. The amplitude, frequency and lacunarity are then added onto this.

The standard implementation of Perlin noise from Unity proved to be very simple and produced a minor effect on the surface of the level.

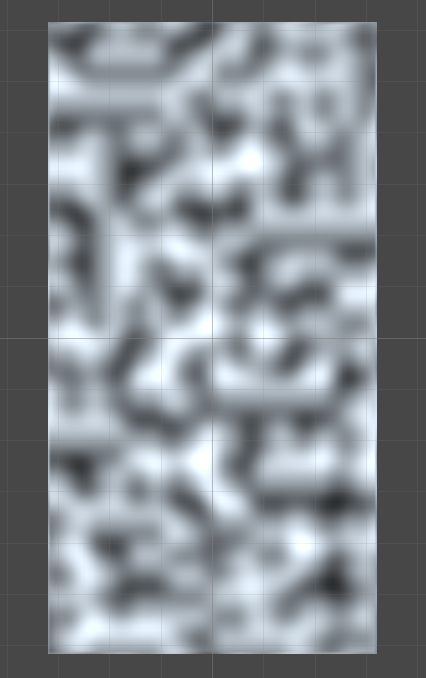
*Figure 3 (above): Mesh with built-in Perlin noise from Unity*



*Figure 4 (above): Simple version of Unity’s built-in code API for Perlin noise, producing the mesh seen in figure 3*

**

*Figure 5 (above): More complex usage of the API in conjunction with the rest of the system*

**

*Figure 6 (above): Noise map generated by the code. This is customisable in the Unity editor.*

In order to achieve the desired effect for the game, a deeper level of implementation was required. Changing to the method described above (using different techniques and values) created a much more in-depth system that allowed for customisation, true random generation and an editor that allowed the original level idea to be created.

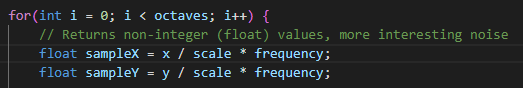
Both methods do not require a lot of computational power to run, however the more complex system will still be very performant since coding practices have been followed that allow for smooth running and error handling.

The code uses a 2D float array for the values of noise, as these values are in a range from 1 to -1. The method used for generating the noise map has all of the different parameters passed into it, such as the width, height, the noise scale, and the persistence, lacunarity and amplitude. A for loop is used to instantiate through the current size of the map and calculate what each of the different affecting values will have on the generated noise.



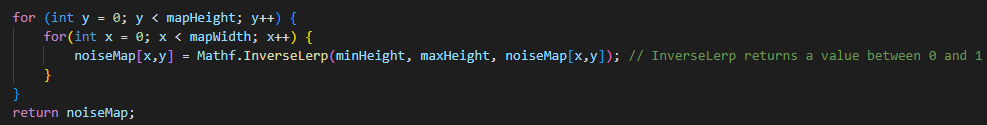
*Figure 7: Noise generation code*

In the loop that holds the calculations for the number of octaves, a float is created that represents a sample of the noise, which is then divided by the scale of the noise and multiplied by the frequency. This, along with the float variables, created non-integer values for more interesting noise.



*Figure 8 (above): Octave calculation loop with noise sample*

The built-in *InverseLerp* method is used with the map width and height in order to produce values between 0 and 1, and values in-between this.



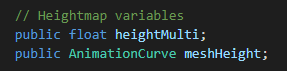
*Figure 9: Map width and height calculations with inverse lerp.*

Custom tools and configuration boxes were created for the Unity editor, which allows for deep customisation as well as improving the error handling – for example, only letting certain parameters have a certain range via a slider rather than an input field *(See figure 11 & below).*

**

**

**

**

*Figure 10: Editor customisation options, in order: Range slider, dropdown list, checkbox, animation curve*

In order to evaluate performance, an FPS counter was added to the game (*code from* [*https://sharpcoderblog.com/blog/unity-fps-counter*](https://sharpcoderblog.com/blog/unity-fps-counter)*).*  The resulting FPS was very high, given the low run time of the algorithm and the simple graphics of the game.Note that the FPS is not locked to any value, so it will run as high as it will possibly go, and there could be inconsistencies in the counter code as well as the system running this game being very powerful.



*Figure 11 (above): FPS counter.*

The fast performance can also be seen as the game does not take very long to load when play mode is entered.

# Discussion

The system implemented for this game works by using normal Perlin noise and then making it more complex by layering several levels of noise on top of one another, then using values such as octaves, lacunarity, persistence, the scale of the noise and a height map to create a randomly generated level with unique features. This implementation of the Perlin noise algorithm was successful at creating all of this in order to produce the gameplay experience that was originally intended.

A picture containing text

Description automatically generated

Graphical user interface

Description automatically generated*Figure 12: Example mesh generated with more complex Perlin system*

*Figure 13 (above): Mesh generation settings in the inspector*

*Chart

Description automatically generated*

*Figure 14 (above): Animation curve tool used to achieve the desired effect.*

The size of the level generated may play a part in a performance detriment, although there is no notable framerate detriment due to the simple nature of the geometry. The initial idea for this algorithm was to have a chunk-based system that generates new terrain in the player’s line of sight and deletes terrain the player cannot see. This would have created an infinite level with little to no performance deficit on systems with less computing power. The main goal for this project was achieved, however there were features that had to be restricted. One idea was to include an agent that would navigate the level, using a physics system that mirrored the player and controls, in order to create a path through the level to ensure that each level was passable. This may have increased the power needed to run and train, but it was not implemented due to time constraints and a lack of technical knowledge. The method that was implemented was suited for and built around the Unity engine, therefore there were no compatibility barriers imposed in development.

There are many other methods that could have been combined to create a very strong and intelligent system. For example, as discussed above, an AI agent using a simple or complex pathfinding algorithm, such as A\*, could have been assigned to navigate a level and clear a path to ensure the level is completable. However, this could increase the time it takes to generate a level and increase the system requirements. Another way this can be verified is by hand-authorising the levels to make sure they are passed, but this would not be feasible for any random level and would therefore eliminate the procedural generation.

Machine learning is also an algorithm technique that could have been used. For example, methods of using deep learning to generate terrain do exist as well as methods that use real-world terrain data to train a model to produce lifelike digital terrain. In the scope of this project, this method of generation would have taken a lot of time and skill to produce as well as drastically increase the power needed and run time.

As discussed before, Perlin noise is only one of many ways to generate terrain randomly. Other noise algorithms could have been used, such as Simplex noise, which was created by the same person that created Perlin noise. This form of noise is not as widely used as Perlin noise, therefore available resources and documentation may be limited making the implementation in Unity much more time consuming and challenging. Along with this, there are not too many differences in these algorithms, and Simplex noise can pose to be much harder to understand fully.