TECHNICAL DESIGN DOCUMENT

PCG Cave Generation Tool

Jack Slaski
22013671

Contents

Glossary

GPU – Graphics Processing Unit

PCG – Procedural Content Generation

UE5 – Unreal Engine 5

CA – Cellular Automata

1. Research

A realistic cave can be generated by using Perlin noise-based algorithms. Perlin noise is a gradient noise function that generates natural, flowing features that are useful for creating terrains, textures, and, in the case of this project, cave formations. This could work by layering different frequencies of Perlin noise to create natural look cave structures. According to Perlin (1985), "Noise() is a scalar valued function which takes a three dimensional vector as its argument" making it ideal for creating large scale and complex patterns with a more natural look such as a cave formation. Another approach to cave generation could be using the Drunkard's Walk algorithm. This algorithm creates natural-looking passageways and chambers by carving out the cave in complete random directions. The randomness can be altered by adding rules and constraints such as avoiding dead ends or ensuring connections between the chambers. This results in another organic structure that is a lot more linear and winding than outputs from the other two methods. The final way this project has explored to generate a realistic cave structure is Cellular Automata. Cellular Automata is a "discrete model of computation" designed to cause a grid of cells to evolve over multiple iterations, mimicking the layout of a cave or dungeon by having each cell exist "in one of two states: empty or rock." (Yannakakis & Togelius, 2018). The evolution of the grid is based on rules that the user defines. In 'Conway's Game of Life', the first rule dictates that if a cell is alive and has 2 or 3 live neighbours, it stays alive, otherwise, it dies. The other rule is that if a cell is dead and has exactly 3 live neighbours, it becomes alive. Using this method to generate a cave could give an output similar to this picture from Kyzrati (2014):

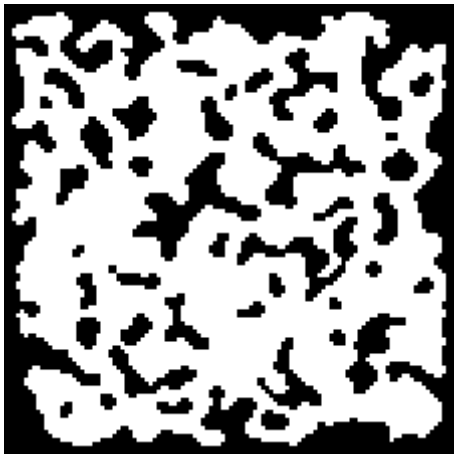

Figure 1. Cellular Automata Map

(from Mapgen: Cellular Automata - Cogmind / Grid Sage Games)

2. Introduction

2.1 Purpose

The purpose of this tool is to solve the problem of creating a cave like structure and environment in Unreal Engine 5. This tool will automate the process of creation of a cave structures by using procedural generation whilst also allowing for other features such as a structure seed, random corridor and room generation, whilst also having a detailed layout of custom settings to allow users to customise the output of this tool to their exact liking.

2.2 Scope

The PCG tool will focus on generating the following types of content:

- **Organic Cave Structure**: Generation of a cave structure with an array of vast and winding open areas.
- **Corridors and Rooms**: Creation of completely customisable interconnected corridors and open room chambers within the cave.
- **Loot**: Placement of loot actors throughout the cave.

2.3 Goals

The goals of this PCG tool include:

- **Increase Development Efficiency:** Provide a highly intuitive tool that efficiently procedurally generate realistic caves to result in a reduction in manual work that creating a cave structure would require.
- **High Level of Customisability:** Offer multiple adjustable settings such as structure size and scaling, structure seed, room and corridor properties, to allow for complete customisability of the cave's environment.
- **Modularity:** A high range of modularity within the project allowing for range of different outputs of the structure.

2.4 Elements

2.4.1 Must Have

This project must have an output that generates an accurate cave like structure containing a contrast of vast open areas as well as thin and winding chambers and corridors. This project also must have a detailed settings panel allowing for complete control over the generated structure.

2.4.2 Should Have

This system should have rooms and corridors throughout to ensure there isn't a lack of content and allow for connections between different parts of the cave structure. This project should also have loot placed inside of different parts of the cave structure.

2.4.3 Could Have

This project could have a seed-based generation system that allows for users to return to a specific generation of the structure. This project could also have some sort of pathfinding throughout to ensure that certain chambers and pathways are connected.

3. System Architecture

3.1 Procedural Generation Core

3.1.1 Cellular Automata

This system uses cellular automata rules for generating the base structure of the cave. The structure of the cellular automata follows similar principles to Conway's Game of Life. The cave generation starts by creating a grid of cells where each cell is either filled (true) or empty (false). The initial state of each cell is determined based on a fill probability influenced by a random or chosen seed. Then, cellular automata rules are applied for a set number of iterations to generate more natural cave formations. The default CA rules in this system are that 'If a cell has less than 4 neighbouring cells that are alive, it becomes empty' and 'If a cell has 5 or more neighbouring cells that are alive, it becomes alive'. These rules are then applied over 6 iterations to ensure the most accurate cave generation. Below is a small

diagram I have created to give some visual representation of the systems default cellular automata rules and how they work.

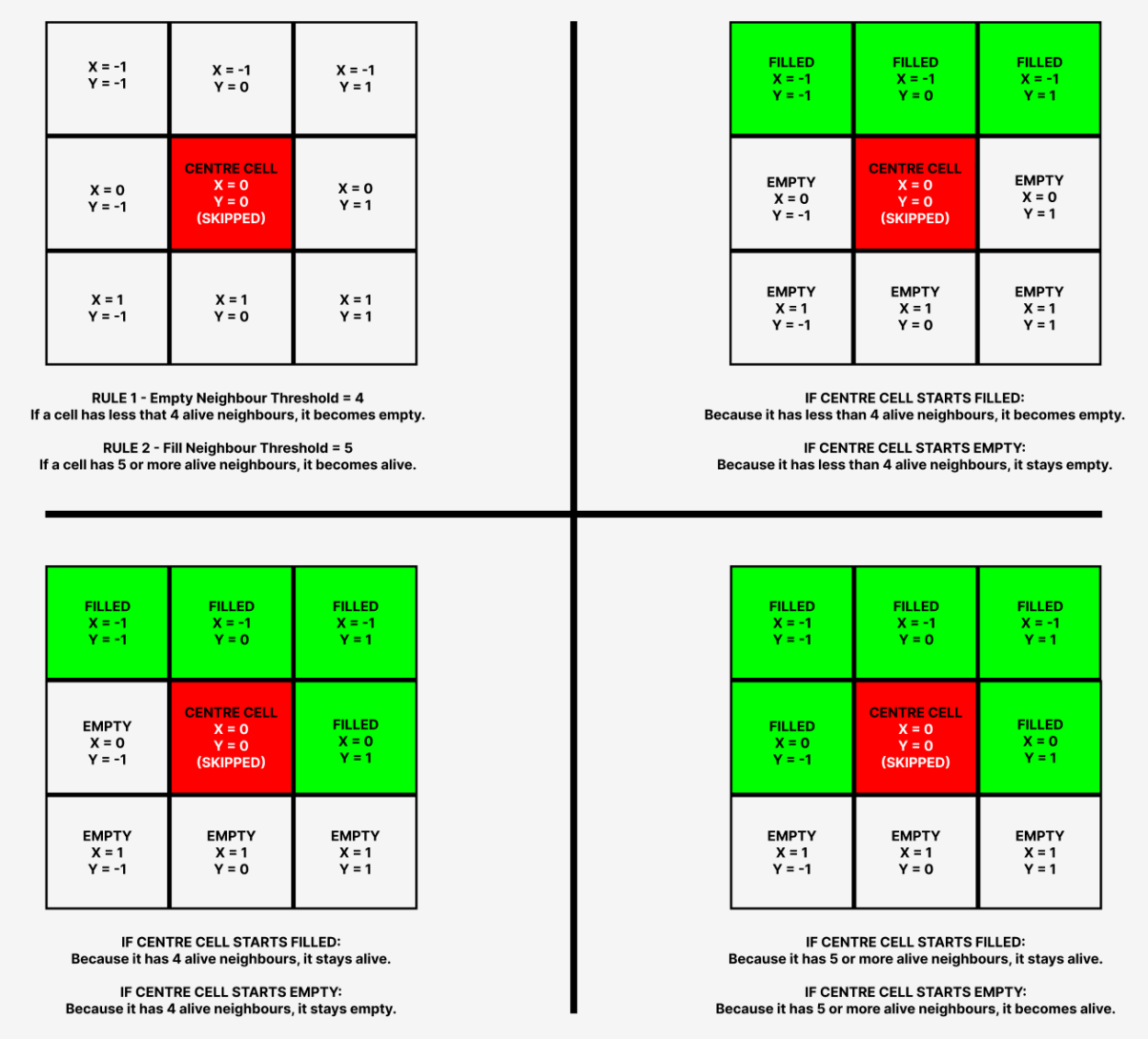

Figure 2. Cellular Automata Rules

3.1.2 Random Corridor Generation

Randomly placed corridors are added to allow for connections and pathways to open up within different areas of the cave. This is done randomly using random corridor lengths and directions, influenced by the tools detailed settings to ensure a high level of customisability. These settings also include options for the corridors thickness, type, and amount.

3.1.3 Random Room Generation

This tool can create randomly placed rooms of randomly chosen sizes within the cave system. These rooms have the option to contain a blueprint actor of the user's choice making it useful for loot, objective or start and exit rooms. This part of the system is also highly customisable with room size (including minimum and maximum values), loot height, and number of rooms.

3.1.4 Seed-Based Random Generation

The generation process begins by initialising a seed value, either provided by the user or generated randomly. The seed is then used to initialise Unreal Engine's random number

generator (FMath::RandInit), which guarantees repeatable cave layouts when the same seed is used.

3.2 Unreal Engine Integration

3.2.1 Blueprint Integration

The vertical prototype of this system is built in Unreal Engine 5's blueprint system. However, the final polished prototype will be converted to C++ so that the system will be much more efficient. Therefore, the blueprint integration of the system is the use of a blueprint actor derived from a C++ class.

3.2.2 C++ API

- **Classes and Functions**: The core cave generation logic is implemented in C++ using Unreal Engine's APIs. This allows fine control over the cave structure and the random generation process.
- **Mesh Components**: Static mesh components are added for filled grid cells, representing the cave's geometry.
- **FMath**: This project uses FMath functions throughout including RandRange, RandInit and more.

3.2.3 Data Exchange

The generated cave elements, such as room loot and corridor structures, are dynamically generated during runtime and are attached to the scene component to make sure there is a much easier cleanup process.

3.3 Data Storage

3.3.1 Configuration Files

This project uses a large amount of configuration parameters such as grid size, corridor count, room count, fill probability, and seed. These are stored as configurable properties, allowing users to customise their cave generation settings.

4. Functionality and Features

4.1 Cave Generation

4.1.1 Cellular Automata

The cellular automata rules and functionality are highly customisable in this system with an in-depth category of settings that allow for changes in the number of iterations, fill probability and rules. These settings allow for a completely customisable CA experience.

4.1.2 Corridor and Rooms

The system also generates corridors and rooms:

- **Corridors** are randomly generated between different parts of the cave, creating pathways that link otherwise isolated areas.
- **Rooms** are added randomly within the cave structure, providing open spaces that may contain loot or serve as objective locations.

4.2 Loot Placement

Loot is spawned in the middle of each generated room if room loot is enabled. The placement of loot is cantered in the rooms and then influenced by the configuration

parameters for room size and spacing. Loot placement can be customised with settings allowing for loot height, and spawning to be changed as well as the type of actor that the loot is.

4.3 Customisation and Control

This system has large amount of customisation to allow for users to refine the structure to however they please. The customisation means that the structure of the system can be completely changed. The settings that can be edited are as follows:

Grid Settings

- Grid Seed: Specifies the seed value for generating the grid. If set to 0, a random seed will be generated.
- Seed Lock: Determines whether the seed is locked or not. If true, the generator will use the provided seed value, if false a random seed will be generated.
- **Grid Size Settings**
    - Grid Size X: Defines the number of cells along the X-axis in the grid.
    - Grid Size Y: Defines the number of cells along the Y-axis in the grid.
    - **Grid Scale**
        - Grid Scale Z: Sets the scale for the grid along the Z-axis, determining the height of the generated grid.
    - **Cell Spacing**
        - Cell Spacing X: Sets the spacing between cells along the X-axis.
        - Cell Spacing Y: Sets the spacing between cells along the Y-axis.
- **Grid Generation**
    - **Cellular Automata**
        - Fill Probability: Determines the initial fill probability of each cell being filled (alive).
        - Number of Iterations: Specifies the number of times the cellular automata rules are applied to modify the grid.
        - Empty Neighbour Threshold: If a cell has fewer than this number of neighbours that are alive, it becomes empty.
        - Fill Neighbour Threshold: If an empty cell has this number or more neighbours that are alive, it becomes alive.
    - **Corridors**
        - Number of Corridors: Specifies the total number of corridors that will be generated.
        - Corridor Length Min: Sets the minimum length of the generated corridors.

        - Corridor Length Max: Sets the maximum length of the generated corridors.
        - Corridor Thickness: Determines the thickness of the corridors. For diagonal corridors, the thickness will be increased by 1.
        - Allow Straight Corridors: Allows the generation of straight corridors in the grid.
        - Allow Diagonal Corridors: Allows the generation of diagonal corridors in the grid.
    - **Rooms**
        - Allow Room Spawning: Determines whether rooms should be generated or not.
        - Number of Rooms: Specifies the number of rooms to be generated.
        - Room Size Minimum: Defines the minimum size of each generated room.
        - Room Size Maximum: Defines the maximum size of each generated room.
        - Allow Room Loot: Determines whether loot should be spawned inside the rooms.
        - Loot Height: Specifies the height at which loot should be spawned inside the rooms.
        - Room Loot: Defines the type of object to be spawned inside the rooms.
- **Grid Mesh**
    - Grid Mesh: Specifies the static mesh that is used for each grid cell.
    - Grid Material: Sets the material to be applied to each cell in the grid.

5. Implementation Details

5.1 Programming Language

The tool is implemented using C++ to take full advantage of Unreal Engine's native performance, as well as its advanced procedural generation capabilities.

5.2 Data Structures

- **TArray**: Used for storing the grid's cell states, room loot actors, and mesh components.
- **Static Mesh Components**: Each filled cell in the cave is represented using a static mesh component to visually display the generated structure.

5.3 Optimisation Techniques

- **Conversion To Code:** This system was originally created using UE5's blueprint graph. However, due to poor performance of this system in the blueprint graph this system was then converted to C++ code in unreal engine.
- **Seed Initialisation**: The use of seed-based initialisation allows repeatable generation, which aids in debugging and testing.
- **Component Pooling**: Mesh components and actors are destroyed and cleared between generations to optimise memory usage and avoid accumulation of unused objects.

6. Testing and Evaluation

6.1 Unit Testing

Due to how in depth this projects settings and customisability is, individual unit testing is a relatively smooth operation. Within this system, there are a couple of ways to test the grid generation. One way (although it being slightly more technical), is that the cell states bool array can be made public and "VisibleAnywhere" or "EditAnywhere". This makes it so that when the grid has been generated, the table of bools for the cell states array is visible so each cell state can be tested, checked, and then verified. Another way to test the grid generation is to simply change and update the settings and cellular automata options and view the updated output. By spawning two grids next to each other with different settings they can be compared and contrasted to test the different settings and ensure that they are working as intended. Testing both the room and corridor generation logic is also simple. Firstly, set the grid fill probability to "1.0", this will ensure that the entire grid fills out with cubes which in turn causes the cellular automata rules to not be applied. Then, by having a full grid, both corridor and room generation features can be tested as they will become obvious as the only holes in the grid will be from the corridors and rooms. This then means that they can be easily checked and verified to ensure that they are correct.

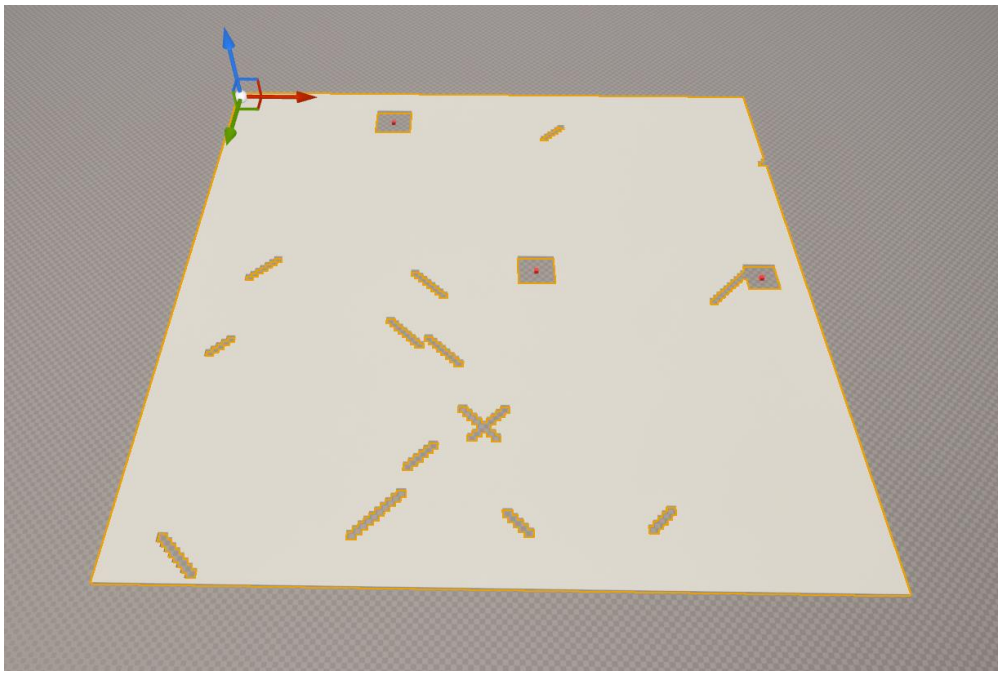

Figure 3. Room and Corridor Testing

6.2 Performance Testing

For performance, the system works well and as intended. The main performance issues this system has is the loading and buffering the system has when generating the grid or changing the grids variables. When changing values in the system there is a slight buffering time when whilst the values are updating. Furthermore, depending on the size of the grid, using the generate button can cause more buffering. For smaller grids such as 50x50 grids or 150x150 grids there is around a 1-2 second buffering time, however for larger grids this time increases. However, this buffering was way worse in the blueprint version of the system which is why the conversion to code improved the systems efficiency.

6.3 Usability Testing

The tool itself does not use a custom user interface or editor utility widget as it largely does not need to. As there is a vast amount of options and settings that can be changed, with the majority of the values being bools, floats, and integers that do not need maximum values. The UE5 details panel provides everything that this system needs. It allows for solid categorization between the different aspects of the system as well as good intuitive design. In terms of other usability, the system works well and as intended.

7. User Guide

There are two ways in which this system can be used including by importing the class itself into another project, or by opening the provided project. To import the class to another project, see ‘7.1 Class Import and Setup’. If using the system in the project provided, see ‘7.2 Project Setup’. For instructions on how to use the blueprint, see ‘7.3 Blueprint Use’.

7.1 Class Import and Setup

For this system to be used on another project, a user must first import the C++ class titled “PCGCaveGenerator” to that project. Once imported, find the C++ class located in the files, right click on it and then choose “Create a Blueprint class based on PCGCaveGenerator”. This creates a blueprint of the class so that it can be used and edited in the engine, this also means means that no additional plugins, materials or models are required for this system.

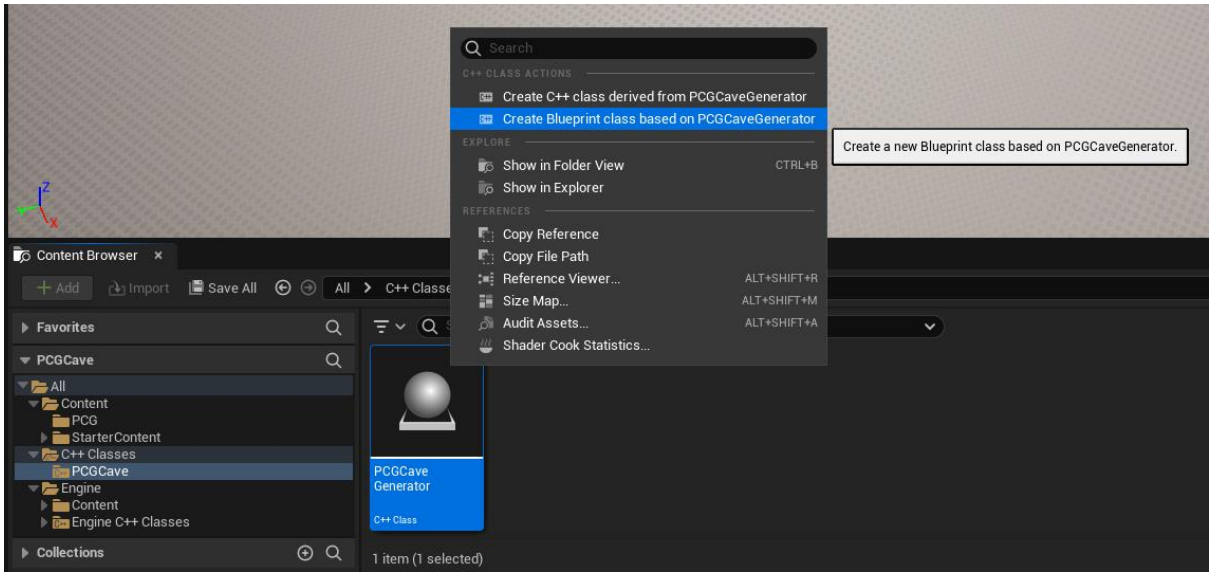

Figure 4. Creating Blueprint Class

7.2 Project Setup

To use the provided project, either create a new level, or open ‘Content/PCG/PCGBlankWorld’. Once in a level, drag the blueprint titled ‘BP_NewPCGCave’, located “Content/PCG/…’, and position it using the translation gizmo. The gizmo will be located at the top right of the structure. Once positioned and setup, the blueprint is ready for use.

7.3 Blueprint Use

Use of this blueprint system is very simple. Once having the blueprint in a level, highlight it and go to the details panel to see the controls and settings for this system under the category of “Grid Settings”.

Figure 5. Grid Blueprint Settings

This provides all of the controls needed to use this system and as well as all of the detailed settings (A description of each setting is in section ‘3.3 Customisation and Control’ in this document).

To generate the grid, simply click the “Generate Grid” button and by default a 150x150 grid will be generated with a random seed. Using the default settings will result in an accurate cave generation, however the settings can all be changed and edited to give a different or more refined output.

To remove the grid, just click the “Remove Grid” button. This should also be done before deleting the blueprint in order to make sure that all parts of the system are deleted and removed from memory.

Bibliography

Perlin, K., 1985. An image synthesizer. *ACM Siggraph Computer Graphics*, *19*(3), pp.287-296.

Yannakakis and Togelius (2018) 'Artificial Intelligence and Games' available at: https://gameaibook.org/book.pdf (Accessed 10/25/2024)

Kyzrati (2014) ‘Mapgen: Cellular Automata’ available at: Mapgen: Cellular Automata - Cogmind / Grid Sage Games (Accessed 10/25/2024)