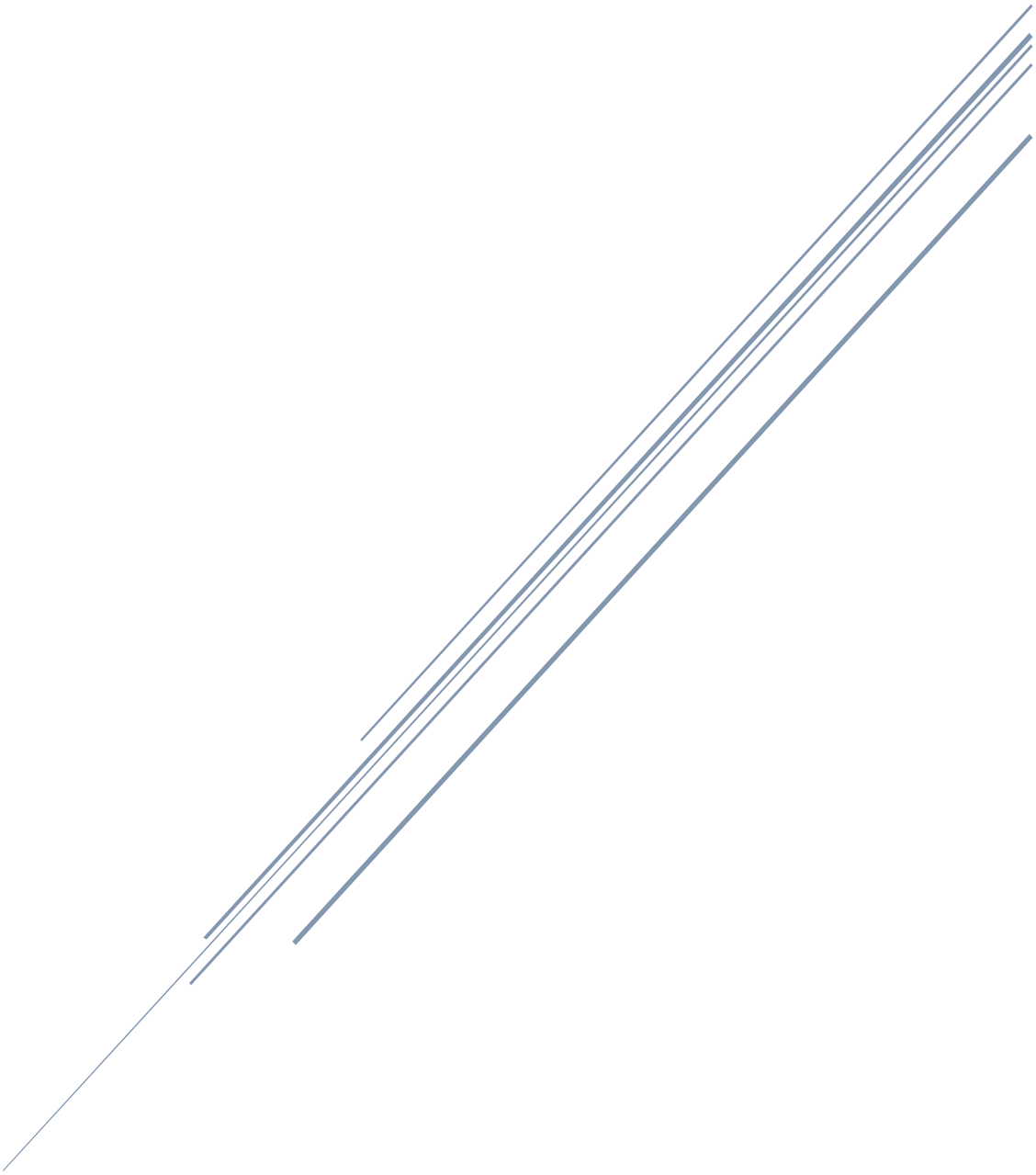
EXPANDING GENERATIVE ITERATIONS OF THE PROTOTYPE

Will Bennett, 20014262k,
B014262k@student.staffs.ac.uk, Shaun Reeves



Contents

Figure Table 2

Table Index 3

Glossary 3

Key words 3

1. Introduction 4

1.1 Aim 4

1.2 Objectives 4

2.Project Methodology 5

3. Background 5

4. Project Plan 5

5. Literature Review 6

Creating Space 6

Shape Grammar 6

Cellular Automata 7

Pre-authored Rooms 7

Environment Creation in Engine 10

Mesh Generator 10

Tilemaps 10

6. Design 11

Requirements 11

Level Generation 11

7. Implementation 13

Cellular Automata 13

Linking it to graph data 14

Pre-authored Rooms 17

Environment Creation 21

Chosen approach 21

Tile maps 22

8. Testing 24

Test Plan 24

Fine tuning adjustments **Error! Bookmark not defined.**

9. Results 30

10. Critical Evaluation 31

Bibliography 32

Figure Table

Figure 1:Example given showing "Generation of a shape using SG1" (Stiny & Gips, 1971) 6
Figure 2: Von Neumann and More neighbourhoods, 'r' is the neighbourhood collection size. (Macedo & Chaimowicz, 2017) 7
Figure 3: "Three versions of the same CA generated from different updating methods (from left to right: Synchronous; Asynchronous, with cells in random order; Asynchronous with cells updated sequentially ordered by grid position.") (Macedo & Chaimowicz, 2017) 7
Figure 4: Layout of an Enter the Gungeon dungeon (Boris the Brave, 2019) 8
Figure 5 Graph showing layout of a dungeon (Boris the Brave, 2019) 9
Figure 6 Composite version of dungeon layout (Boris the Brave, 2019) 9
Figure 7: Contour line lookup table (Ramalho, N.A.). 10
Figure 8: Screenshot from Ludomotion showcasing Cyclic dungeon generation (Ludomotion, 2017) 11
Figure 9: Screenshot of a method setting a value of neighbouring cells. 13
Figure 10: Screenshot showing CA application to limited areas. 13
Figure 11: Screenshot of a method for smoothing over a CA map. 14
Figure 12: A method for getting the surrounding number of neighbouring walls. 14
Figure 13: GenerateCave method that generates the map. 15
Figure 14: Applying EdgeData process within ApplyGraphData method. 16
Figure 15: Applying NodeData when creating pre-authored rooms and cave structures within ApplyGraphData. 16
Figure 16: Screenshot of method that creates definite wall zones. 16
Figure 17: Screenshot of "definite dead” cells blocking CA spread. 17
Figure 18: Example of a pre-authored room with gaps within all the doors. 17
Figure 19: Example of a door gap filler prefab used in project. 18
Figure 20: A clear example of pre-authored room application. 18
Figure 21: Scriptable Object "PreAuthoredRoomSO". 19
Figure 22: Screenshot of the scriptable object "PreAuthoredRoomSO" in Unity's inspector. 19
Figure 23: Code to create and fill a pre-authored room scriptable object in OnInspectorGUI. 20
Figure 24: Graph component script with buttrons. 20
Figure 25: Getting room data from a PreAuthoredRoomSO within SetRoomCells. 20
Figure 26: Applying data and clearing space within SetRoomCells. 21
Figure 27: SetLeftDoor which is an example of creating a blocker in a door. 21
Figure 28: Example of passing data to TileMap script in GenerateCave. 22
Figure 29: TileMap script variables as seen in Unity Editor. 22
Figure 30: SetTile method. 23
Figure 31: GetGameObject method. 23
Figure 32: Smallest limit of scale. 26
Figure 33: Largest limit of scale. 26
Figure 34: Scale set to 17 with a node range of 10-17. 27
Figure 35: Max node depth set below scale value. 27
Figure 36: Low random fill percent of 15. 27
Figure 37: High random fill percent of 75. 28
Figure 38: Low - middle random fill percent of 45. 28
Figure 39: High- middle random fill percent of 60%. 28
Figure 40: Low smooth iterations of 1. 29
Figure 41: High smooth iterations of 7. 29

Table Index

Table 1: Test plan. 24
Table 2: Adjustment tests 25

Glossary

PCG – Procedural content generation

CA – Cellular Automata

Key words

PCG, Graph rewriting, Game development, Cellular Automata, Pre-authored.

1. Introduction

Tool development is highly tied in with procedural content generation (PCG), especially in industry. Procedural content generation is a form of artificial intelligence that is often linked with replayability in games this is especially true within the roguelike genre. This project will explore procedural content generation that is able to create a physical environment, this environment will be based on generative graph grammar that was created in a previous project “Enhancing existing prototype” (Bennett, 2024). This will take the graph data and transform it into an environment through the use of cellular automata and pre-authored rooms.

1.1 Aim

Increase generative variations of an existing prototype so that more environments are available.

1.2 Objectives

- What ways of creating space suit graph grammar?
- How have other games used pre-authored rooms?
- How to transfer graph data to make physical space?
- Outline the limitations of the approach used in the framework.

2.Project Methodology

This project is managed with a supervisor, this allows for project's progress to be tracked and analysed. To aide in this, this management was done through the use of a Kanban board (Rehkopf, n.d.). This is due to larger tasks are easily broken down and assigned deadlines to keep the under a strict scope. The Kanban board is a type of agile methodology (Wrike, N.D), this allows for the project to be iteratively managed.

3. Background

Procedural content generation (PCG) is a hot technology within the games industry with many games boasting features created by PCG. The topic this paper explores is tool creation with a focus on graph rewriting. A previous project "Enhancing previous prototype" (Bennett, 2024) focused on creating a PCG tool using graph rewriting as its technique. The tool developed for that project was focusing on replicating Dorman's Ludoscope software (N.D) generative graph grammar. This was due to the tools versatility of allowing gameplay to influence the design and layout of a dungeon which is not common with PCG, Dormans coined this term of design Cyclic grammar (2017) . To further this Dormans' tool is not public and has no supporting documentation so creating a tool that can offer a similar process as Dormans' would be beneficial. After this project, the tool was able to create data representing the flow of an environment through the use of graph grammar. The aim of this project is to improve the tool by taking the graph data and allowing the user to transform it into a physical representation of a dungeon through additional PCG methods.

4. Project Plan

The aim of this project is to create a physical environment, this environment will be an expansion of a graph rewriting tool. The graph rewriting tool allows users to create generative grammars, which when applied to a graph allows the creation of mission data which represents the flow of a dungeon. To scope this project, the artifact will be able to create physical space with a focus on a cave like structure and pre-authored rooms. This will be done through exposing values of the tool to allow the user to set restrictions on the generation of the environment to meet expectations.

5. Literature Review

Creating Space

In the current project, generating a level stop at graph data. This data can represent a multitude of concepts, with Dormans (2010) using graph rewriting to represent mission data, or it can be interpreted as a direct layout of a dungeon.

Shape Grammar

Shape grammar was first discussed in 1971 by Stiny & Gips (1971), which they described it as being similar to phase structure grammars but differ due to being defined via an alphabet of shapes and are used to generate n-dimension of shapes. The application of shape grammar starts with an initial shape and recursively applies shape rules. This continues till no rules can be applied as seen in Figure 1.

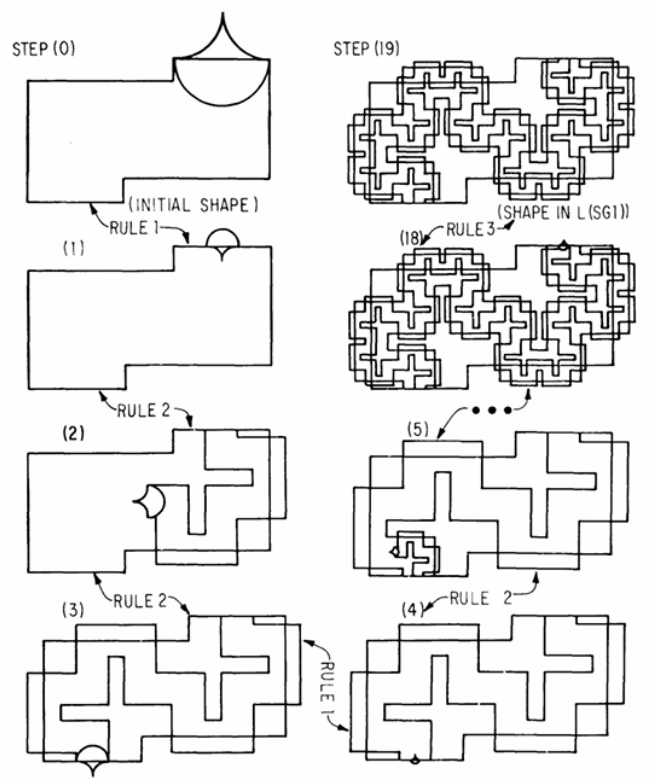


Figure 1:Example given showing "Generation of a shape using SG1" (Stiny & Gips, 1971)

While exploring strategies for generating levels for adventuring games, Dormans (2010) discussed a process of using shape grammar to generate space from a mission. Dormans details how associating terminal symbols of missions to rules of shape grammars. In doing so shape grammars can find applicable grammars and a suitable space. To avoid compromising missions such as putting keys behind locks, a reference is stored to allow for tight coupling. Combined with dynamic parameters, difficulty can be introduced by influencing rule selection. Once the mission is accounted for the shape grammar can continue in its default implementation to replace all non-terminal symbols and finalise the space. Dormans details how without these types of modifications then both grammars cannot be used to their strengths allowing for a coupled experience of environment reflecting story for users. This all depends on the quality of grammars created but can create a high variety of quality levels.

Cellular Automata

Cellular Automata (CA) is a spatial distribution process model that allows changes of “cells” in a grid according to defined rules driven by neighbouring cells (Awati, 2021). These cells usually represent a finite number of states, such as “0” or “1”. The state of each cell is determined by its neighbours, this can be done synchronously or asynchronously. This can lead to distinct patterns depending on the initial pattern. There is two common ways determining a cell’s neighbours which are Von Neumann and Moore Neighbourhood concepts.

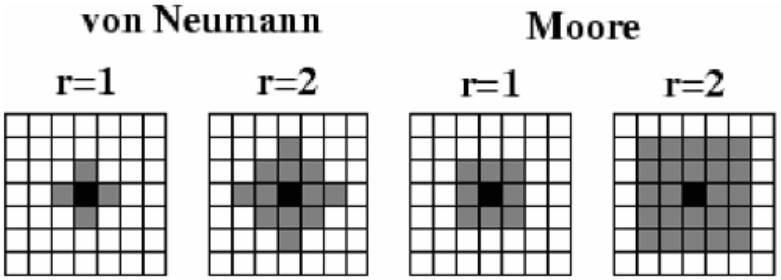


Figure 2: Von Neumann and More neighbourhoods, 'r' is the neighbourhood collection size. (Macedo & Chaimowicz, 2017)

Macedo & Chaimowicz (2017) explore the impact of Asynchronous and Synchronous updating which can be seen in Figure 3. Each approach yields distinct results with their project’s constraints causing significant reduction in the outcomes individuality. This showcases that each has their own purpose depending on the use within a project which will be explored later in the project.

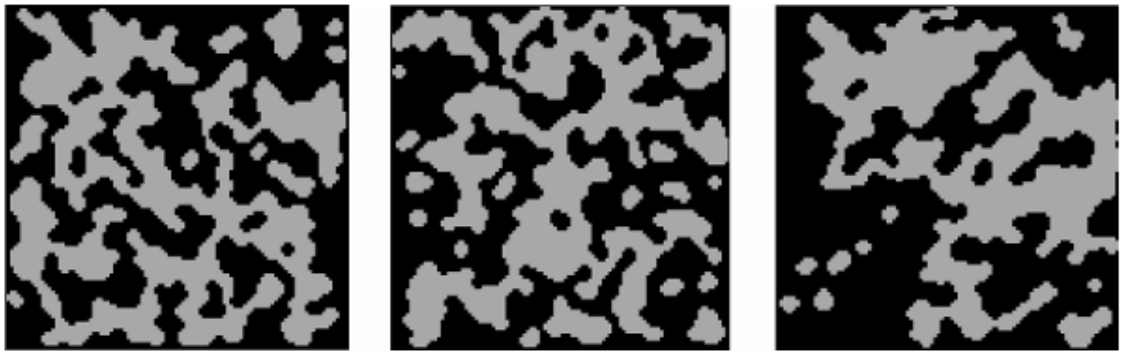


Figure 3: "Three versions of the same CA generated from different updating methods (from left to right: Synchronous; Asynchronous, with cells in random order; Asynchronous with cells updated sequentially ordered by grid position.") (Macedo & Chaimowicz, 2017)

Pre-authored Rooms

Another approach for creating environments is using pre-authored rooms. This approach does not a set way of achieving this, but they follow a concept of a generated path being filled with pre-authored rooms.

The Binding of Isaac

Boris (2020) has explored how The Binding of Isaac (McMillen & Himsl, 2011) generates its levels. The approach used is uniform rooms being built upon an empty grid. This begins from one room and spreads from there similar to a breadth first search algorithm. Conditions are applied to this search such as a room having a maximum of two neighbours to stop looping. The sprawl is made of blank rooms, to assign room types a set of rules is applied. Such as a room with only one connection is an

“end room” which can host a “boss”, “shops”. These can have more conditions applied such as a “boss” room being the furthest from the “start” room. An exception to all rooms being placed at the start is the “secret” room which is placed after generation in an empty cell that has at least three connections. The rest of the rooms are “normal” rooms and are designed to allow for a door in the middle of each wall, this allows for no considerations as each room can be accessed from four directions. These rooms have random layout and enemy types within. For difficulty scaling, the first level has a mix of easy and medium rooms while the second level has a mix of medium and hard.

Enter the Gungeon

Enter the Gungeon (Dodge Roll, 2016) is like The Binding of Isaac with its gameplay but it takes a different approach to its dungeon layout. The dungeon is made from pre-authored rooms, but what makes Enter the Gungeon special is the dungeon’s layout. This is achieved using pre-authored graphs. These rooms come in many different shapes and sizes but as seen in Figure 4, these seem to be locked into a grid with equal spacing between.

Boris (2019) explored this providing two figures to showcase how this works as seen in Figure 4 and Figure 5. Enter the Gungeons dungeon’s layout has both good spacing with filler rooms to ease difficulty and add pacing as well as looping. An example of this is the top of the map with the red line acting as a one directional hallway. This allows the player to explore for the cost of fighting more enemies, high risk high reward. While the level does have loops within there is still mechanics support quick movement around the dungeon such as teleporters. From the flow seen in Figure 5 to the actual layout in Figure 4 a technique called compositing is used to break down the map into manageable sections. Single loops, set of connections, and loopless rooms (tree) are broken from the main map this can be seen in Figure 6. These broken up rooms are pieced together from the start node onward. If a layout is not possible it will regenerate the layout. After this the separate layouts are then pieced together.

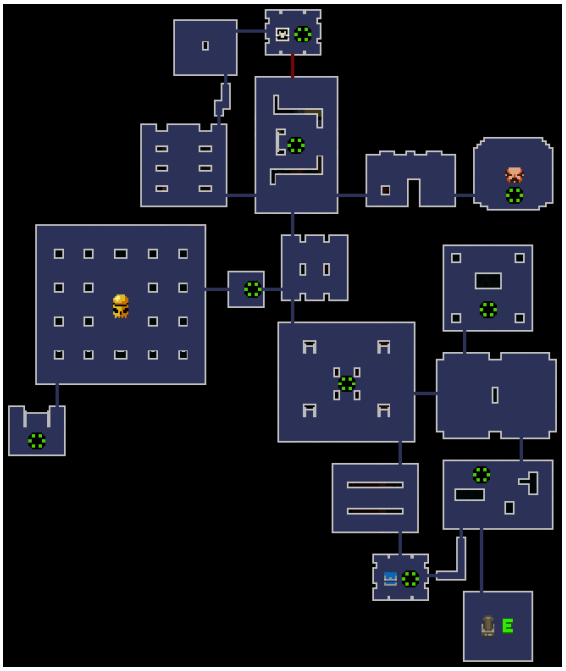


Figure 4: Layout of an Enter the Gungeon dungeon (Boris the Brave, 2019)

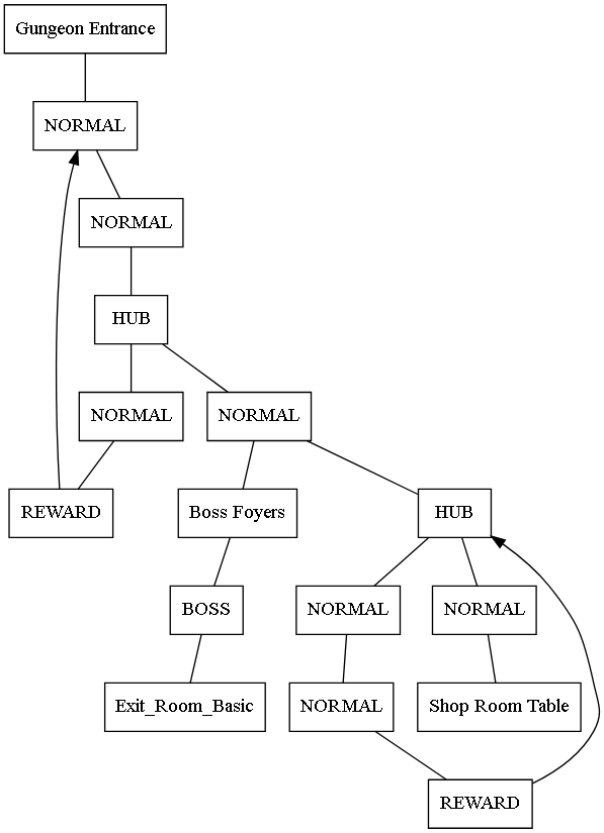


Figure 5 Graph showing layout of a dungeon (Boris the Brave, 2019)

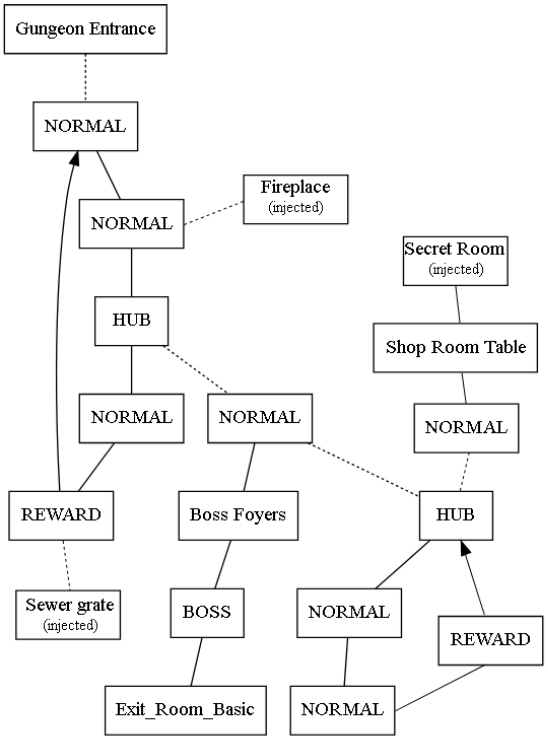


Figure 6 Composite version of dungeon layout (Boris the Brave, 2019)

Environment Creation in Engine

Mesh Generator

Unity (2023) offers a way to create custom meshes through code, this means data produced through the framework can then be applied to procedural mesh generator. While there are a multitude of ways of creating a mesh generator, most follow a similar idea. This idea is creating vertices, Flick (2021) details how triangles are the simplest surfaces that can be described in 3D, with each corner being a vertex of the mesh. By linking the vertices together through an integer array, a triangle is able to be bound together.

Taking this further, Lague (Lague, 2015) has a playlist detailing steps for creating a cave generator based off CA data. This approach would be ideal when creating a mesh for the cave in the framework. But creating a custom mesh generator might isolate users if they want to create a different style since user experience with coding mesh generators may differ greatly. This will have to be explored later in the project's development. One keynote that Lague mentions within his mesh generator is the concept of marching squares. These can be used in creating outside edges for cave like structures. Marching squares are an algorithm that is used for contouring, Ramalho (N.A.) states usually the algorithm has three steps, taking a gird of values to define a threshold and building a binary grid, traversing the new grid looking for 2x2 cells of values, and using a contour line lookup table to draw lines as seen in Figure 7. While these can be used to create 2D lines, it can be extrapolated to 3D.

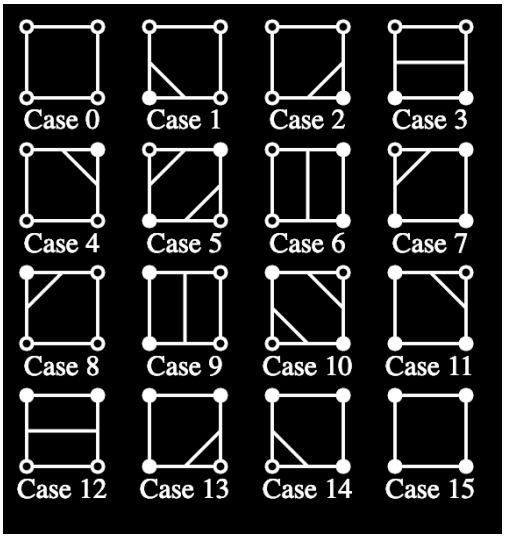


Figure 7: Contour line lookup table (Ramalho, N.A.).

Tilemaps

Unity's Tilemap system (2023) has a system that allows users to create tilemaps. This has two supported versions. The main use of Unity's tilemap system is to create 2D game worlds due to the tool allowing the attachment of sprites to the environment through the use of a tile palette. Although Unity does offer a package that can be downloaded through the package manager in engine that enables the attachment of GameObjects. This replaces the use of 2D sprites with 3D objects. These 3D objects can then be linked with the data created to represent physical space.

6. Design

Requirements

The generation framework should seamlessly link with the graph grammar system. This will happen through the data contained in the graph being displayed through the generation of the map. The graph should have data that relates to specifics of the generation such as type of technique used. This will allow the physical environment to properly display the data created by the user.

Requirement	Purpose of requirement
Ability to place pre-authored rooms.	This will create a baseline for the generator to build upon.
Ability to limit and shape CA paths.	Being able to separate and define paths will allow the use of CA without level layout compromise.
Ability to cohesively blend two generative techniques together.	The use of two techniques being used together makes for more variety.
Generation to reflect graph	The dungeon generated links back to graph data.

Level Generation

While each of these methods are valid for generating a level, especially with its connection to graph grammar. To provide an interesting level layout a mix of methodologies will be used. This will be a mix of pre-authored rooms and cellular automaton. CA will be used to generate natural looking cave paths/rooms. An example of this can be seen in Unexplored (Ludomotion, 2017) shown in Figure 8, CA is used to generate pathways as well as explorable areas. Additionally Unexplored generates its map based on graph grammar.

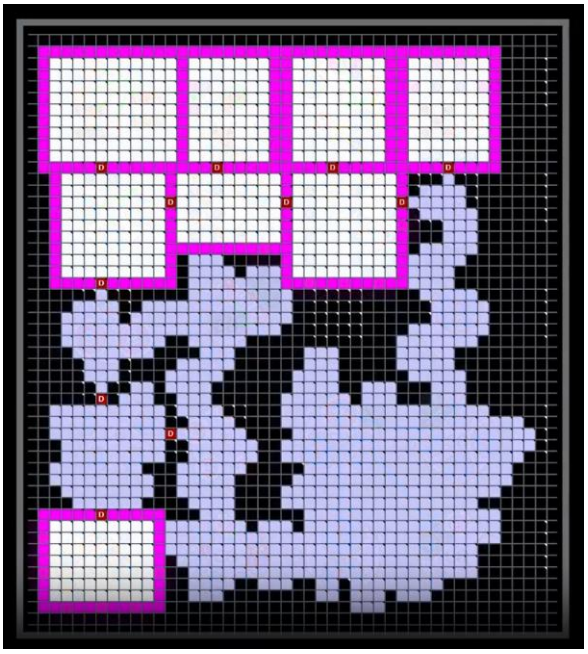


Figure 8: Screenshot from Ludomotion showcasing Cyclic dungeon generation (Ludomotion, 2017)

Cellular automata usually generates random layouts, to allow control over the generation process additional rules/constraints can be implemented. CA uses the states of its neighbours to determine its own state, this allows the switching between “true” and “false” states. This idea can be extended

to allow for “definite”. If a cell is labelled as “definitely true” it cannot be changed, this allows for set paths to be defined. Applying this to “definitely false” and set boundaries can be set to not allow for unwanted paths to be created.

7. Implementation

Cellular Automata

As mentioned in previous sections, CA is the chosen method for generating the cave like structures for this framework. This is implemented through the use of a two-dimensional integer array; this will allow for x and y coordinates and allow for data to be assigned to each grid coordinate. This data will represent the state of the cell allowing for rules to apply to influence the state of neighbouring cells. Within this project the cells can be four states; “0” and “1” which are default CA states that represent if a cell is a wall or not. In addition to this “2” and “-1” dictate that a cell cannot be changed.

These values are not definitive, as within this rule the values only need to be greater or smaller than 1/0 for the algorithm to not change its value. This allows for greater functionality which is used for pre-authored rooms which will be explained later.

The neighbourhood selection size is based off Moores’ neighbours design discussed by Macedo & Chaimowicz (2017). While the example they showcased can be seen in Figure 2, they use a r of one, meanwhile, to allow for more customisability the algorithm allows upward from one. This can be seen in Figure 9, specifically the integer variable called depth.

```
1 reference
void SetSurroundingCells(int gridX, int gridY, int depth, int value)
{
    for (int neighbourX = gridX - depth; neighbourX <= gridX + depth; neighbourX++) //LOOPS ROUND XY COORD
    {
        for (int neighbourY = gridY - depth; neighbourY <= gridY + depth; neighbourY++)
        {
            if (neighbourX >= 0 && neighbourX < m_width && neighbourY >= 0 && neighbourY < m_height) //within the grid
            {
                if(neighbourX != gridX || neighbourY != gridY) // not looking at self
                {
                    m_map[neighbourX, neighbourY] = value;
                }
            }
        }
    }
}
```

Figure 9: Screenshot of a method setting a value of neighbouring cells.

Most CA frameworks set the random pattern across the entirety of the environment, but due to this framework using it to make caves based on a set area designated from a graph this would introduce issues. One of these issues would be caves creating shortcuts from between nodes which could interrupt the flow of the dungeon. To reduce the risk of this happening, CA only affected cells a set distance from key girds which will later be the graph node positions. This can be seen in Figure 10, this method assigns a value of 0 or 1 based on if the random number generated is lower than the fill percent. Fill percent relates to coverage of walls.

```
void SetRandomSurroundingCells(int gridX, int gridY, int depth, System.Random rand)
{
    for (int neighbourX = gridX - depth; neighbourX <= gridX + depth; neighbourX++) //LOOPS ROUND XY COORD
    {
        for (int neighbourY = gridY - depth; neighbourY <= gridY + depth; neighbourY++)
        {
            if (neighbourX >= 0 && neighbourX < m_width && neighbourY >= 0 && neighbourY < m_height) //within the grid
            {
                ChangeMapValue(neighbourX, neighbourY, (rand.Next(0, 100) < m_randomFillPercent) ? 1 : 0);
            }
        }
    }
}
```

Figure 10: Screenshot showing CA application to limited areas.

To smooth over the data a method seen in Figure 11, gets the surrounding cell's total value and if it meets a threshold then the value of the original cell changes. This is done in an asynchronous method based on the cells grid position. Additionally, if the value is not 0 or 1 the value is not changed. GetSurroundingWallCount is seen in Figure 12, as seen before, if a value is not 0 or 1, they are not counted. This method adds the cells values together and allows SmoothMap to determine the value based on the total neighbours count. When checking the neighbours values Moores' neighbouring method is used with a range of 1.

```
void SmoothMap()
{
    for (int x = 0; x < m_width; x++)
    {
        for (int y = 0; y < m_height; y++)
        {
            if (m_map[x, y] < 0 || 1 < m_map[x,y])
            {
                continue;
            }
            int neighbourWallTiles = GetSurroundingWallCount(x, y);

            if (neighbourWallTiles > 4)
                m_map[x, y] = 1;
            else if (neighbourWallTiles < 4)
                m_map[x, y] = 0;

        }
    }
}
```

Figure 11: Screenshot of a method for smoothing over a CA map.

```
int GetSurroundingWallCount(int gridX, int gridY)
{
    int wallCount = 0;
    for(int neighbourX = gridX - 1; neighbourX <= gridX + 1; neighbourX++) //LOOPS ROUND XY COORD
    {
        for (int neighbourY = gridY - 1; neighbourY <= gridY + 1; neighbourY++)
        {
            if (neighbourX >= 0 && neighbourX < m_width && neighbourY >= 0 && neighbourY < m_height) //within the grid
            {
                if (neighbourX != gridX || neighbourY != gridY) //if not looking at self
                {
                    if (m_map[gridX, gridY] < 0 || 1 < m_map[gridX, gridY])
                        break;
                    wallCount += m_map[neighbourX, neighbourY]; //add its value to wallcount (0 is empty so it technically doesnt effect as its not a wall)
                }
            }
            else
            {
                wallCount++;
            }
        }
    }
    return wallCount;
}
```

Figure 12: A method for getting the surrounding number of neighbouring walls.

Linking it to graph data

Changes to NodeData

Data from the graph's nodes and edges must be passed and applied to the CA's map. The struct "NodeData" had to be adjusted to allow for storage of this data. The key changes are the addition of an integer "terrain" and bool "preAuthored". The variable "terrain" allows for the cave generator to check if it should create an empty space, while "preAuthored" allows for the assignment of a premade room on that node's coordinate. Additionally, two references to "Index2EdgeDataLinker" class for the above and right edge have been added. This optimises future searches for connected edges.

Initialising Data

When the ruleset is run, the script GraphComponent does two things. It initialises the graph, runs the ruleset, and sends the completed graph data to the CaveGenerator, this data includes the size of the map, the scale, and the edges and nodes. The reason the GraphComponent, does this instead of the Graph script, is because the GraphComponent stores most of this data to display to the user due to Graph being static which was explained in the previous project (Bennett, 2024). This data is then processed by the CaveGenerator script, from a method that can be seen in Figure 13. The process of smoothing has already been explained above, while transferring the data to the tile map will be explained later in the section on page 22.

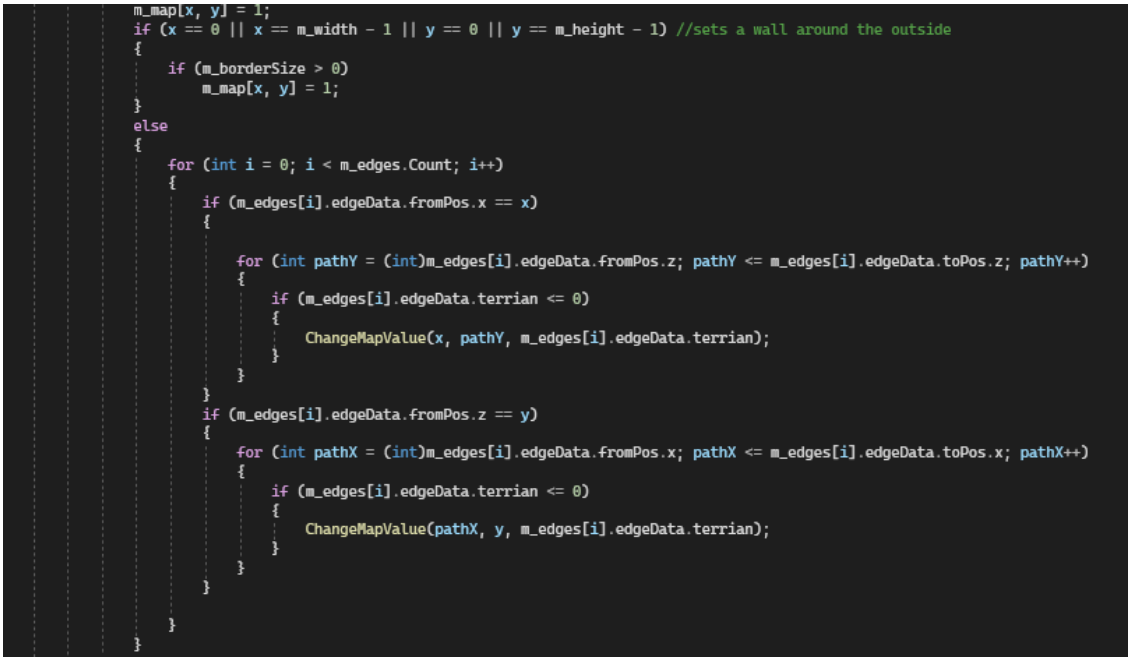
```
2 references
public void GenerateCave()
{
    m_map = new int[m_width, m_height];
    RandomFillMap();

    for (int i = 0; i < m_smoothIterations; i++)
    {
        SmoothMap();
    }

    TileMap tileMapGen = GetComponent<TileMap>();
    for (int x = 0; x < m_map.GetLength(0); x++)...
}
```

Figure 13: GenerateCave method that generates the map.

ApplyGraphData method

The ApplyGraphData does three main things, create paths based on edge data, create cave like structures, and place pre-authored rooms. In chronological order, edge data is applied first, then caves and pre-authored rooms.

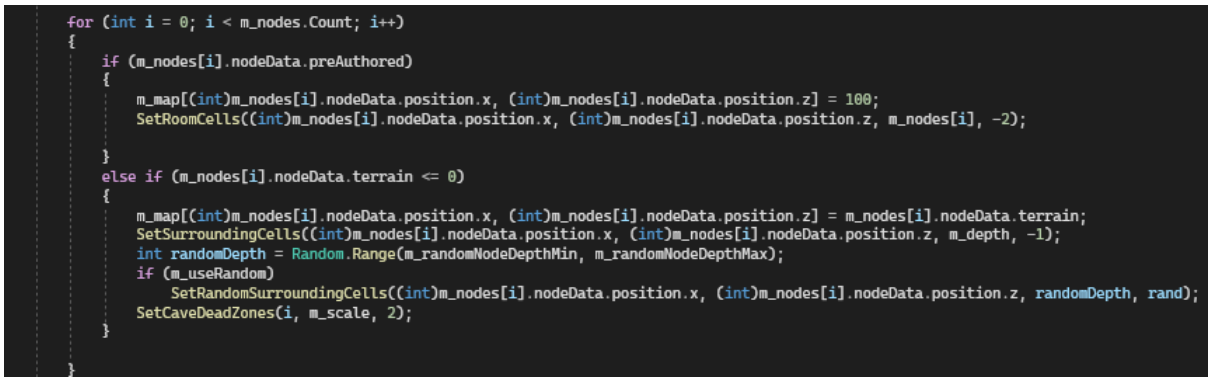
Edge data is applied in set conditions in a double for loop, it checks for matching positions with the edge's from position and a x and y coordinate (this would be the z coordinate from edge data since it is on a flat plane and y is up in Unity). Once this match is found a range coordinate have their data changed to the edges stored terrain. This can be seen in Figure 14, where a for loop is used to create a line from the "z" or "x" coordinates stored in "fromPos" and "toPos".



```
            m_map[x, y] = 1;
            if (x == 0 || x == m_width - 1 || y == 0 || y == m_height - 1) //sets a wall around the outside
            {
                if (m_borderSize > 0)
                    m_map[x, y] = 1;
            }
            else
            {
                for (int i = 0; i < m_edges.Count; i++)
                {
                    if (m_edges[i].edgeData.fromPos.x == x)
                    {

                        for (int pathY = (int)m_edges[i].edgeData.fromPos.z; pathY <= m_edges[i].edgeData.toPos.z; pathY++)
                        {
                            if (m_edges[i].edgeData.terrian <= 0)
                            {
                                ChangeMapValue(x, pathY, m_edges[i].edgeData.terrian);
                            }
                        }
                    }
                    if (m_edges[i].edgeData.fromPos.z == y)
                    {
                        for (int pathX = (int)m_edges[i].edgeData.fromPos.x; pathX <= m_edges[i].edgeData.toPos.x; pathX++)
                        {
                            if (m_edges[i].edgeData.terrian <= 0)
                            {
                                ChangeMapValue(pathX, y, m_edges[i].edgeData.terrian);
                            }
                        }
                    }

                }
            }
```

Figure 14: Applying EdgeData process within ApplyGraphData method.

Node data can be split into two sections, pre-authored rooms (which will be explained in Pre-authored Rooms) and cave structures which can be seen in Figure 15. Cave generation is mainly explained in beginning of Cellular Automata section. The one additional section is SetCaveDeadZones. In Figure 16, this method shows one of four if statements that creates a definite "dead zone". This stops the CA spread from creating shortcuts as mentioned previously. The two conditions it checks for are if the edge between directional or if it is a default name. With these no shortcuts can be created on key pathways. The outcome of this can be seen in Figure 17. Figure 16 is a key example of the optimisation saving processing time as the matching edges do not need to be searched for.



```
    for (int i = 0; i < m_nodes.Count; i++)
    {
        if (m_nodes[i].nodeData.preAuthored)
        {
            m_map[(int)m_nodes[i].nodeData.position.x, (int)m_nodes[i].nodeData.position.z] = 100;
            SetRoomCells((int)m_nodes[i].nodeData.position.x, (int)m_nodes[i].nodeData.position.z, m_nodes[i], -2);

        }
        else if (m_nodes[i].nodeData.terrain <= 0)
        {
            m_map[(int)m_nodes[i].nodeData.position.x, (int)m_nodes[i].nodeData.position.z] = m_nodes[i].nodeData.terrain;
            SetSurroundingCells((int)m_nodes[i].nodeData.position.x, (int)m_nodes[i].nodeData.position.z, m_depth, -1);
            int randomDepth = Random.Range(m_randomNodeDepthMin, m_randomNodeDepthMax);
            if (m_useRandom)
                SetRandomSurroundingCells((int)m_nodes[i].nodeData.position.x, (int)m_nodes[i].nodeData.position.z, randomDepth, rand);
            SetCaveDeadZones(i, m_scale, 2);
        }

    }
```

Figure 15: Applying NodeData when creating pre-authored rooms and cave structures within ApplyGraphData.

```
void SetCaveDeadZones(int nodeIndex, int depth, int value)
{
    if (m_nodes[nodeIndex].nodeData.upperEdge.index != -1
        && (m_nodes[nodeIndex].nodeData.upperEdge.edgeData.directional || m_nodes[nodeIndex].nodeData.upperEdge.edgeData.symbol == "edge"))
    {
        //create up blocker
        Vector3 posDifference = (m_nodes[nodeIndex].nodeData.upperEdge.edgeData.toPos + m_nodes[nodeIndex].nodeData.upperEdge.edgeData.fromPos) / 2;
        for (int posX = (int)posDifference.x - depth / 2; posX < posDifference.x + depth / 2; posX++)
        {
            ChangeMapValue(posX, (int)posDifference.z, value);
        }
    }
```

Figure 16: Screenshot of method that creates definite wall zones.

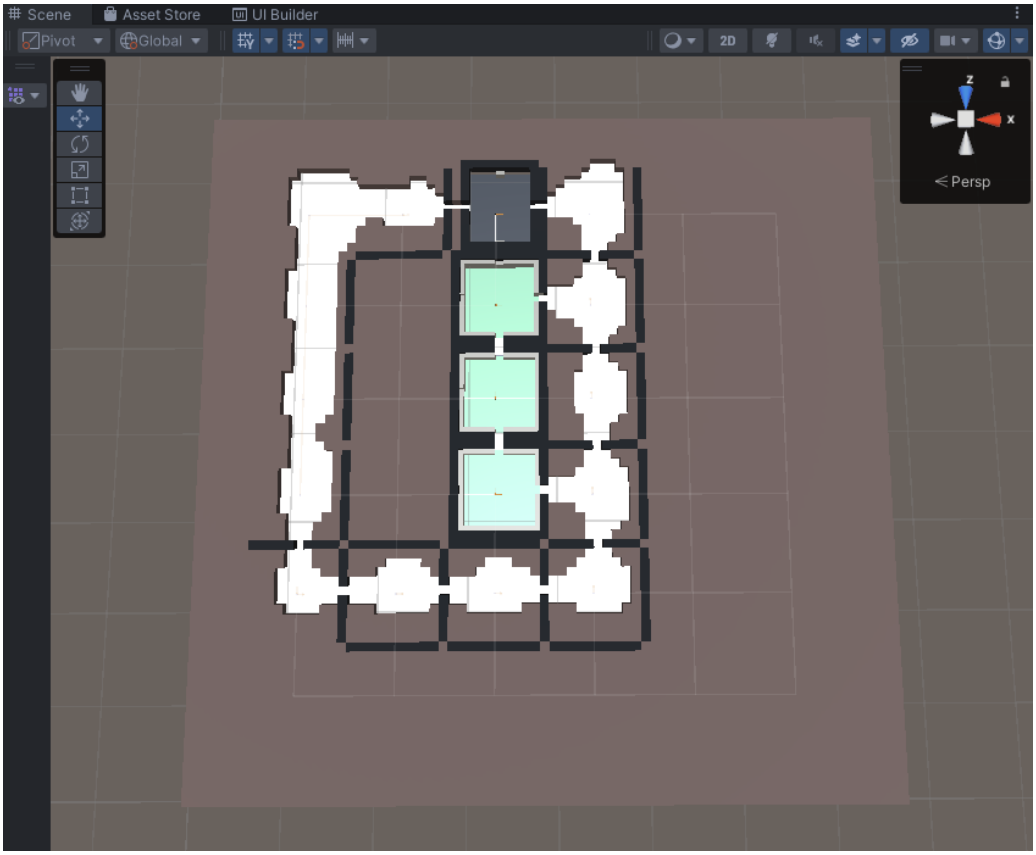


Figure 17: Screenshot of "definite dead" cells blocking CA spread.

Pre-authored Rooms

The approach used to place pre-authored is similar to "Binding of Isaac", the similarity is within the design of rooms that needs to be used. In Figure 18, an example of a pre-authored room is seen which has four gaps in each of the cardinal directions for doors. With this the room can have corridors attached in each direction based on the generation. To avoid these being kept open, these are then blocked with a prefab as seen in Figure 19 to create a closed-door affect which can be seen in Figure 20.

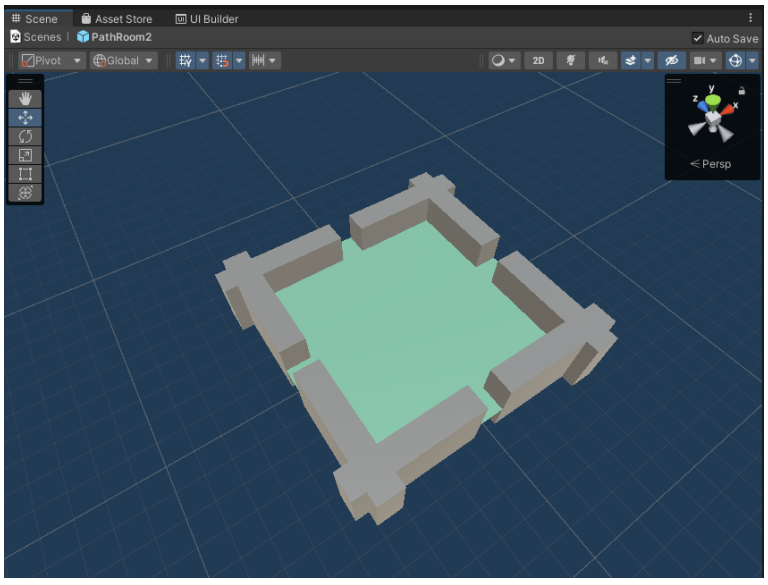


Figure 18: Example of a pre-authored room with gaps within all the doors.

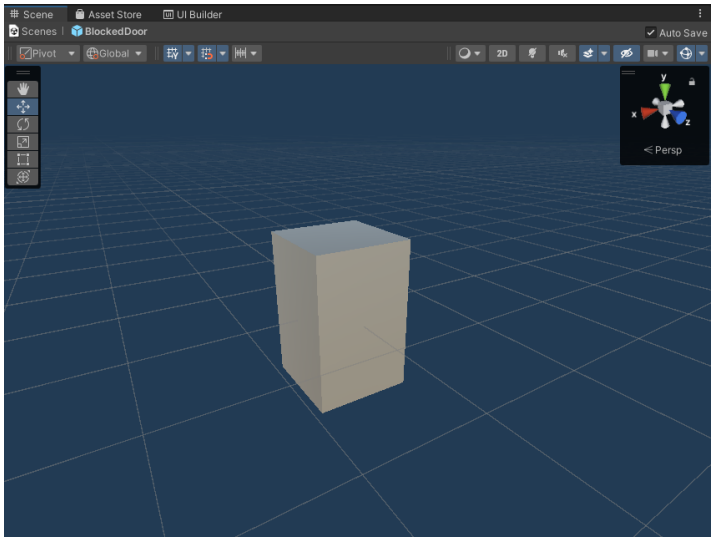


Figure 19: Example of a door gap filler prefab used in project.

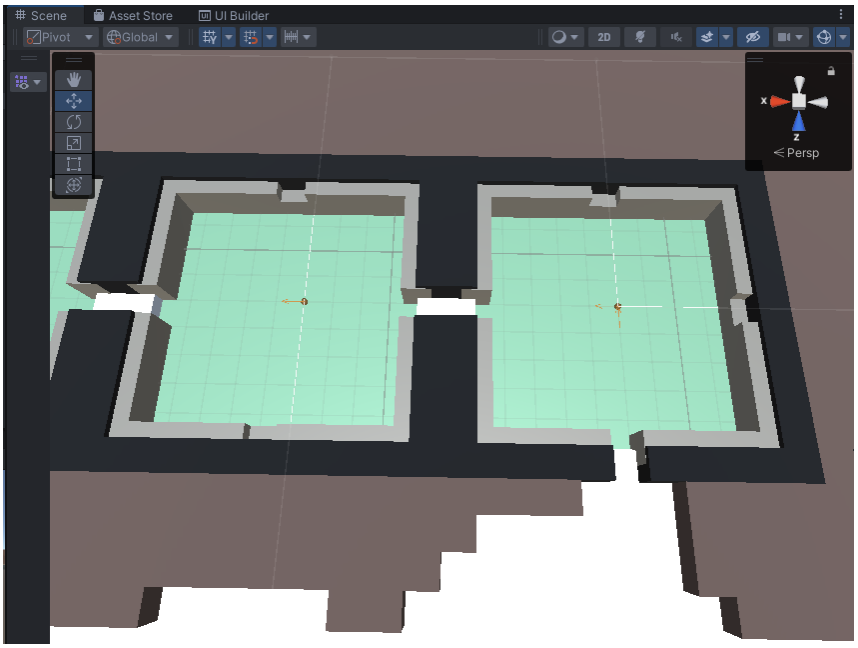


Figure 20: A clear example of pre-authored room application.

To allow for the storage of the pre-authored rooms, a scriptable object was created. This allows key data to be assigned to prefabs such as, the alphabet symbol/key that the rooms will be applied to, how likely the prefab is to be applied to the room, the size of the room, and the room itself with its blockers. All of this can be seen in Figure 21 and Figure 22. Allowing for each individual room to have different sizes enables more variety within the generation system. To allow for ease of use, a button was created that automatically creates the pre-authored room object filled with all alphabet keys. This was done to reduce the chance of user error in missing any keys they have added, this can be seen in Figure 23 and Figure 24.

```
Unity Script | 13 references
public class PreAuthoredRoomSO : ScriptableObject
{
    [Serializable]
    8 references
    public struct Room
    {
        [Tooltip("Always have a chance of 100 to allow at least 1 room to appear")]
        public int m_chanceOfAppearing;
        [Tooltip("This will dictate the size of blank space created")]
        public int m_roomWidth;
        public int m_roomHeight;
        public GameObject m_roomPrefab;
        public GameObject m_roomDoorBlockerPrefab;
    }

    [Serializable]
    4 references
    public struct Roomset
    {
        [Tooltip("This will be the key in the alphabet that will be associated with this room set")]
        public string m_alphabetKey;
        public List<Room> m_roomPrefab;
    }

    public List<Roomset> m_roomSets;


}
```

Figure 21: Scriptable Object "PreAuthoredRoomSO".

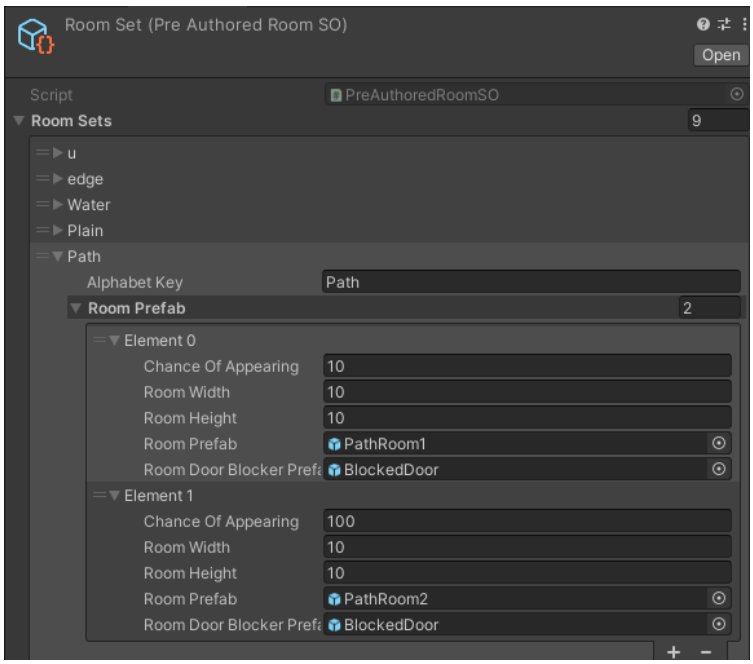


Figure 22: Screenshot of the scriptable object "PreAuthoredRoomSO" in Unity's inspector.

```
if (GUILayout.Button("Create Room Set SO"))
{
    PreAuthoredRoomSO instance = ScriptableObject.CreateInstance<PreAuthoredRoomSO>();
    AssetDatabase.CreateAsset(instance, "Assets/scripts/RoomSet.asset");
    instance.m_roomSets = new List<PreAuthoredRoomSO.Roomset>();
    foreach (Alphabet.AlphabetLinker key in graphComponent.m_alphabet.m_alphabet)
    {
        PreAuthoredRoomSO.Roomset roomSet = new PreAuthoredRoomSO.Roomset();
        roomSet.m_alphabetKey = key.m_symbol;
        instance.m_roomSets.Add(roomSet);
    }
    instance.name = "RoomSet";
    AssetDatabase.SaveAssets();
    EditorUtility.FocusProjectWindow();
    Selection.activeObject = instance;

    Debug.Log("Room set scriptable object outputted.");
}
```

Figure 23: Code to create and fill a pre-authored room scriptable object in OnInspectorGUI.

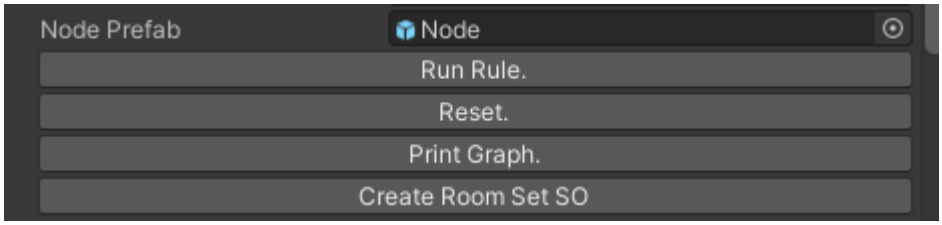


Figure 24: Graph component script with buttrons.

To get this data a foreach loop searches through all room sets stored in the prefab and then finds the list that match the alphabet key of the current node. Within this a random number is generated and compared to the chance of the room appearing. If this passes the data is set to the local variables as seen in Figure 25. Additionally, the room GameObject is added to a list which allows them to be deleted if the map is cleared.

```
int roomWidth=0, roomHeight =0;

GameObject roomPrefab;
PreAuthoredRoomSO.Room roomRef = new PreAuthoredRoomSO.Room();
System.Random rand = new System.Random();
//find correct wall
foreach (var roomSet in m_roomSets.m_roomSets)
{
    if (roomSet.m_alphabetKey == node.nodeData.symbol)
    {
        foreach (PreAuthoredRoomSO.Room room in roomSet.m_roomPrefab)
        {
            if (rand.Next(0, 100) <= room.m_chanceOfAppearing)
            {
                //get random room based on chance of appearing
                roomRef = room;
                roomWidth = room.m_roomWidth/2;
                roomHeight = room.m_roomHeight/2;
                roomPrefab = room.m_roomPrefab;
                GameObject roomObject = Instantiate(roomPrefab, new Vector3(gridX + 0.5f, 0, gridY + 0.5f), Quaternion.identity);
                m_createdRooms.Add(roomObject);
                break;

            }
        }
    }
}
```

Figure 25: Getting room data from a PreAuthoredRoomSO within SetRoomCells.

Now that a room has been chosen the data obtained is applied to the map. The two for loops get positions around the room defined by the width and height, when the outer edge is found and is less than 0 it is set to a value of 2 to not allow the border to be changed by CA. Then the door positions are obtained which is in the middle of each cardinal direction from the centre. Meanwhile if the

position is not on the edge of the room the cell's value is set to 100, this also stops any CA changes but also has an application for the physical rendering of the map which will be explain at Tile maps section.

```
//SETTING WALL BOUNDRY
for (int neighbourX = gridX - roomWidth; neighbourX <= gridX + roomWidth; neighbourX++) //LOOPS ROUND XY COORD
{
    for (int neighbourY = gridY - roomHeight; neighbourY <= gridY + roomHeight; neighbourY++)
    {
        if (neighbourX >= 0 && neighbourX < m_width && neighbourY >= 0 && neighbourY < m_height) //within the grid
        {
            //check for outer edge of depth set to extreme pos
            if(neighbourX == gridX - roomWidth || neighbourX == gridX + roomWidth || neighbourY == gridY - roomHeight || neighbourY == gridY + roomHeight)
            {
                if(0 < m_map[neighbourX, neighbourY])
                    m_map[neighbourX, neighbourY] = 2;
                if(m_map[neighbourX, neighbourY] != 100)
                {
                    if (gridX == neighbourX && gridY + roomHeight == neighbourY)
                        SetUpDoor(node, neighbourX, neighbourY, roomRef);
                    if (gridX + roomWidth == neighbourX && gridY == neighbourY)
                        SetRightDoor(node, neighbourX, neighbourY, roomRef);
                    if (gridX == neighbourX && gridY - roomHeight == neighbourY)
                        SetDownDoor(node, neighbourX, neighbourY, roomRef);
                    if (gridX - roomWidth == neighbourX && gridY == neighbourY)
                        SetLeftDoor(node, neighbourX, neighbourY, roomRef);
                }
            }
            else if (neighbourX != gridX || neighbourY != gridY) // not looking at self
            {
                if (0 < m_map[neighbourX, neighbourY] || m_map[neighbourX, neighbourY]==-1)
                    m_map[neighbourX, neighbourY] = 100;
            }
        }
    }
}
```

Figure 26: Applying data and clearing space within SetRoomCells.

Each direction sets up a direction-based door through the related method. This can be seen in Figure 27, which shows one of the four directions. The key if statement checks the corresponding edge's terrain type. If it is less than 1 then this will be kept open and be a doorway, otherwise it will have the blocker prefab spawned within that gap. This is inspired by McMillen and Himsl's approach used in "Binding of Isaac" (2011), since the rooms have four potential connected hallways, which if they lead to nothing become a blocked off wall. This logic has been applied but now works for graph data.

```
void SetLeftDoor(Index2NodeDataLinker node, int posX, int posY, PreAuthoredRoomSO.Room roomRef)
{
    if (0 < node.index - m_graphWidth)
    {
        if (m_nodes[node.index - m_graphWidth].nodeData.rightEdge.edgeData.terrian <= 0)
        {
            m_map[posX, posY] = -1;
        }
        else
        {
            GameObject blockerObj = Instantiate(roomRef.m_roomDoorBlockerPrefab, new Vector3(posX+1f, 0, posY+0.5f), Quaternion.identity);
            m_createdRooms.Add(blockerObj);
        }
    }
    else
    {
        GameObject blockerObj = Instantiate(roomRef.m_roomDoorBlockerPrefab, new Vector3(posX + 1f, 0, posY + 0.5f), Quaternion.identity);
        m_createdRooms.Add(blockerObj);
    }
}
```

Figure 27: SetLeftDoor which is an example of creating a blocker in a door.

Environment Creation

Chosen approach

Previously in the paper at Environment Creation in Engine, two different methods were discussed for creating a physical environment from the map data generated. After implementing a version of Lague's mesh generator (2015), it was deemed to be that it could be daunting for some users of the tool especially if users are wanting to implement custom textures. Meanwhile Unity tilemap system (2022) has pre-existing documentation attached to it, plus outside experience with Unity's system will enable some users to transfer that knowledge to the tool.

Tile maps

As hinted at in the Initialising Data in Figure 13, CaveGenerator script passes its map data over to the TileMap script which can be seen in Figure 28. The logic this process is the data gets passed by reference to a method that uses this data as a vector3. If a cell's value is 0 or passes the else, they get set on two different heights. This allows for pathways and walls to from. The cell value of 100 was seen in Figure 26 when setting inside of a pre-authored room since these areas need to be left empty. The exception to this is values of 2, which when "debug" is enabled will display definitely dead nodes.

```
TileMap tileMapGen = GetComponent<TileMap>();
for (int x = 0; x < m_map.GetLength(0); x++)
{
    for (int y = 0; y < m_map.GetLength(1); y++)
    {
        if (m_map[x, y] <= 0)
        {
            tileMapGen.SetTile(x, 0, y);
        }
        else if(m_map[x, y] == 2)
        {
            tileMapGen.SetTile(x, 5, y);
        }
        else if (m_map[x,y]==100)
        {
            tileMapGen.SetTile(x, 100, y);
        }
        else
        {
            tileMapGen.SetTile(x, 10, y);
        }
    }
}
```

Figure 28: Example of passing data to TileMap script in GenerateCave.

As seen in Figure 29, the TileMap script needs 4 key references, this is a base tilemap which acts as the walkable open areas and an upper tilemap which acts as the walls of the environment. The variable named tile is set up automatically for the user, while higher, lower, and limiter are prefabs used to create the environment with.

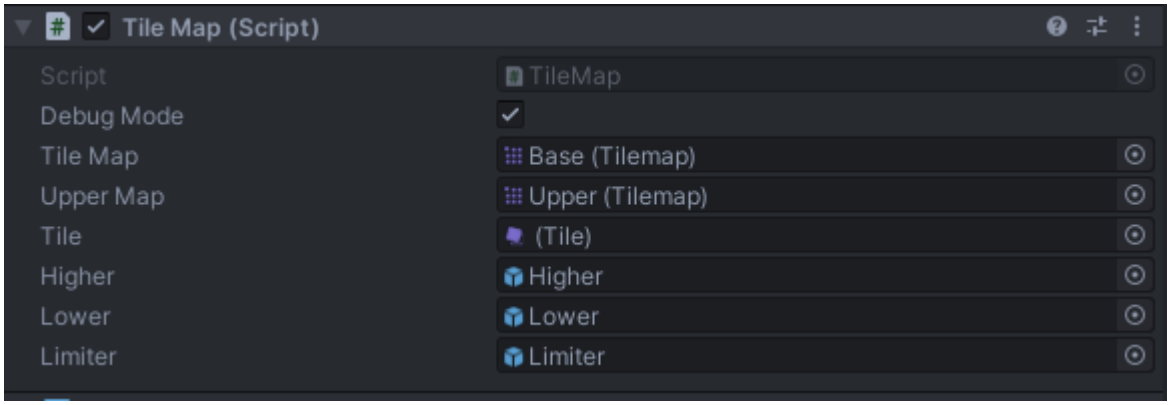


Figure 29: TileMap script variables as seen in Unity Editor.

The type of tilemap and GameObject applied to said tilemap is defined by the "yPos" passed in form the CaveGenerator script, which can be seen in Figure 30 and Figure 31.

```
public void SetTile(int xPos, int yPos, int zPos)
{
    if (yPos ==100)
    {
        return;
    }
    Vector3Int tilePos = new Vector3Int(xPos, yPos, zPos);
    Vector3Int currentCell;

    if (yPos <= 0)
    {
        currentCell = tileMap.WorldToCell(tilePos);
        m_tile.gameObject = GetGameObject(yPos);
        tileMap.SetTile(currentCell, m_tile);
    }
    else
    {
        currentCell = upperMap.WorldToCell(tilePos);
        m_tile.gameObject = GetGameObject(yPos);
        upperMap.SetTile(currentCell, m_tile);
    }

    if (m_debugMode)
    {
        if (yPos == 5)
        {

            currentCell = upperMap.WorldToCell(tilePos);
            m_tile.gameObject = GetGameObject(yPos);
            upperMap.SetTile(currentCell, m_tile);
        }
    }
}
```

Figure 30: SetTile method.

```
GameObject GetGameObject(int yPos)
{
    GameObject gameObject;
    if (yPos <= 0)
        gameObject = lower;
    else
        gameObject = higher;
    if (m_debugMode)
    {
        if (yPos == 5)
        {
            gameObject = limiter;
        }
    }

    return gameObject;
}
```

Figure 31: GetGameObject method.

8. Testing

Testing for this project will be split into two sections, one will follow a test plan, and the other will focus on parameter adjustments. Geeks for geeks (2023) outlines a test plan as a base document for testing activities, which provides benefits such as guidance in big projects, enables scope management, as well as possible problems and solutions to be outlined. The test plan outlined for this project was followed throughout the project and was adjusted alongside shifting elements of the framework. This allows for the scope of the project to be captured while keeping up to date on key areas to test throughout and at the end of the project. The key tests will allow for the analyse of the project by linking their successfulness at completing objectives. Meanwhile testing for parameter adjustments will allow for the analysis of the limitations of the tool while finding its "sweet spot" for generating dungeons. This can also be linked back to the objectives and allow for the analysis of the tool's success.

Test Plan

Table 1: Test plan.

Test NO.	Test description	Expected result	Actual result	Required actions
1	Graph data is accurately represented in the environment.	Graph data is correctly passed to map data.	Map data corrected stored passed data.	No actions are required.
2	Graph to map scale is correctly applied.	Graph scale is consistent between all applications.	Graph scale was inconsistent in creating pre-authored rooms.	Changing used variables to correct scale.
3	Caves have the correct sizing.	Caves have the correct sizing depending on depth stated.	Caves were correct but larger than expected.	Size of caves have been reduced to better suit the value stated.
4	Caves are varied.	Caves have a wide range of shapes.	A few caves had the same shape.	Caves have a range of sizes they can appear; this allows for more control and varied shapes.
5	Caves do not create short-cuts.	No paths break the pathing created by the graph.	Pathing was broken in corners of the maps.	Definite dead zones were created to isolate paths.
6	No detached caves occur.	No small unreachable caves are created.	Few unreachable caves were created.	CA now only occurs within the size of the cave. Localised CA reduces the risk of isolated caves.
7	Placed data does not get overwritten.	There is no unexpected formation.	Unexpected formations were caused from edge application.	Edges and Nodes cannot change cells that have been changed.

8	Smoothing of caves is applied correctly.	CA for caves is correctly applied based on value supplied.	CA for caves is correctly applied based on value supplied.	No actions are required.
9	Map data is created correctly into a tilemap.	Map data is correctly translated into physical space.	Map data was correctly translated into physical space.	No actions are required.
10	Room set can automatically be created with correct data.	Room set scriptable object is created with correct alphabet data.	Room set scriptable object was created but was saved with incorrect name.	Changed the saving process of the scriptable object.
11	Room set data is correctly applied to the environment.	The environment accurately represents the room set data.	The environment accurately represented the room set data.	No actions are required.
12	Pre-authored rooms have door blockers placed in correct positions.	Pre-authored rooms have door blockers placed in correct positions.	Door blockers were in incorrect positions.	Adjustments to prefab application to adjust for scale and offsets.
13	Previous environments are cleared before a new rule is run.	Environments are deleted when a new rule is run.	Created rooms persisted after a new rule was run.	Adding created rooms to a list to be cleared after a new rule is run.

Since pre-authored rooms are made with specific sizes in mind, the rooms will be removed from this testing. Although the rooms are not placed the pathing for them still should be placed, this allows for checking if the placement still works at each adjustment.

Table 2: Adjustment tests

Adjustment NO.	**Adjustment type**	**Value used**	**Notes**	**Figure number**
1	Smallest limit of scale.	4	The smallest limit for scale is 4, with node depth of 1-3.	Figure 32
2	Largest limit of scale.	30	The largest scale before performance drops is 30, with node depth set to 10-30.	Figure 33
3	Middle of scale limit.	17	The outcome highly depended on the range of depth. This had a node depth range of 10 - 17.	Figure 34
4	Upper limit of max node depth.	15	Max node depths rely on the scale used; best results often related to it being below the scale. This uses the same min and scale as adjustment 3 with max node depth being 15.	Figure 35
5	Low random fill.	15%	Random fill being set low causes overfill. This uses the same values as adjustment 4.	Figure 36

6	High random fill.	75%	Random fill being set to high causes narrow passageways. This uses the same value as adjustment 4.	Figure 37
7	Lower middle random fill.	45%	Random fill being set to a low middle number shows signs of overfill beginning to occur but is minimal. This uses the same values as adjustment 4.	Figure 38
8	Higher middle random fill	60%	Random fill being set to a high middle value shows signs of narrowing beginning to occur. This uses the same values as adjustment 4.	Figure 39
9	Low smooth iterations	1	Smooth iterations being this low leaves signs of the mixed cells and gives fewer smooth surroundings.	Figure 40
10	High smooth iterations	8	Smooth iterations being higher than this will cause more or lessen expansion.	Figure 41

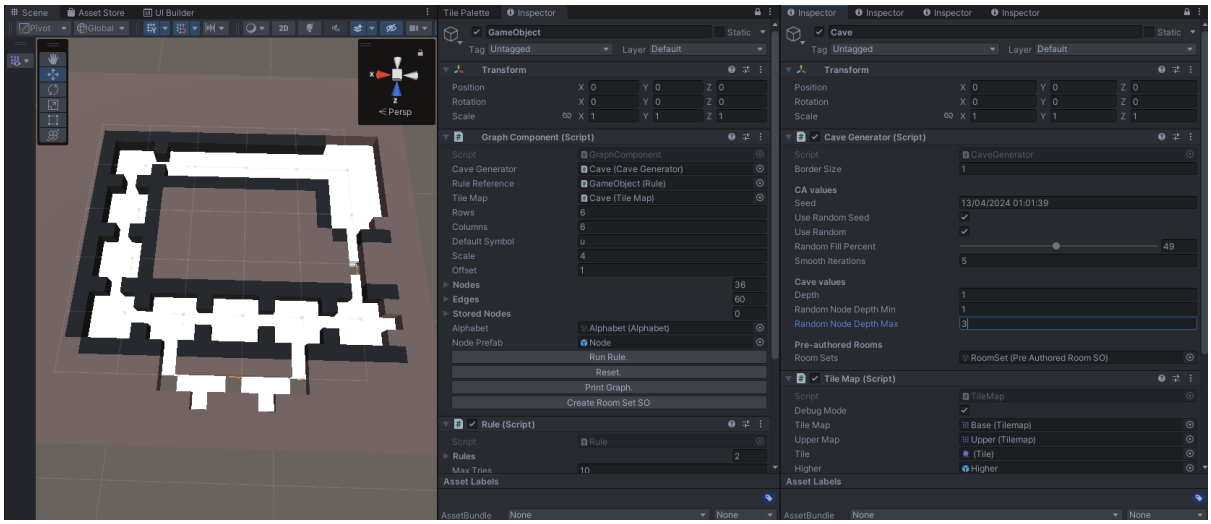


Figure 32: Smallest limit of scale.

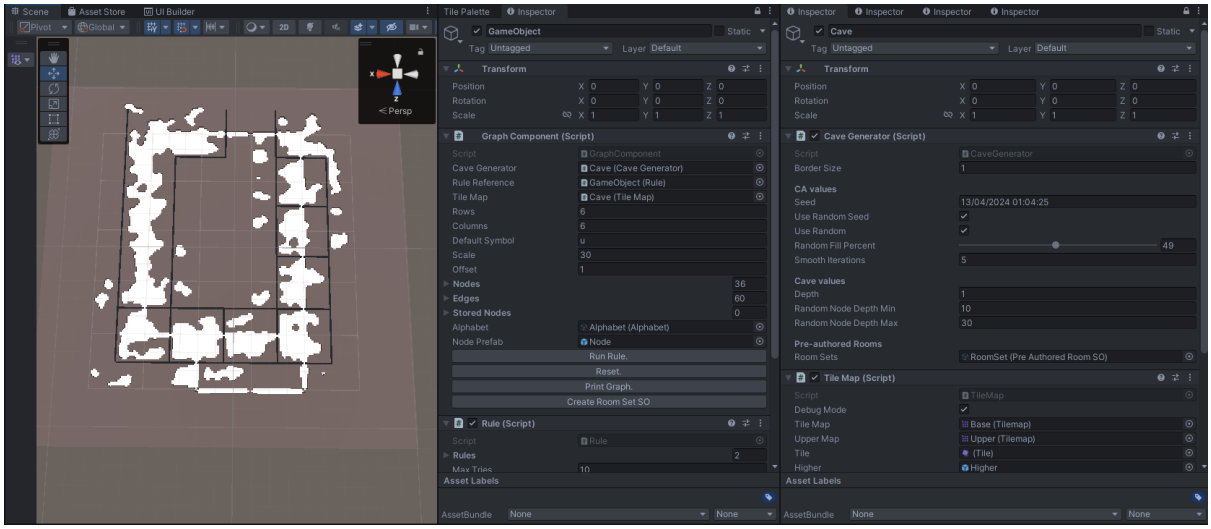


Figure 33: Largest limit of scale.

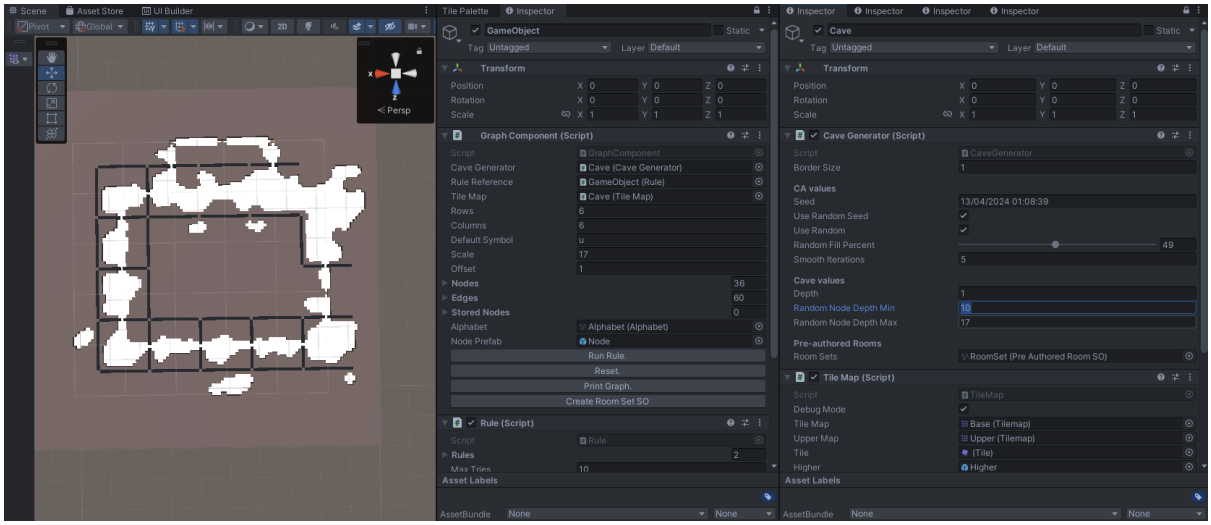


Figure 34: Scale set to 17 with a node range of 10-17.

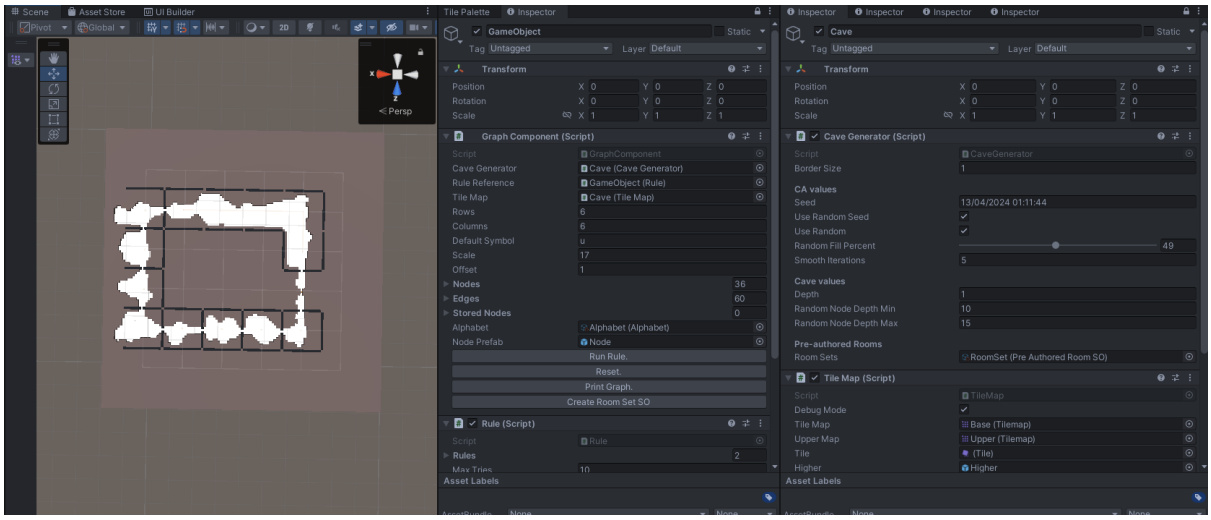


Figure 35: Max node depth set below scale value.

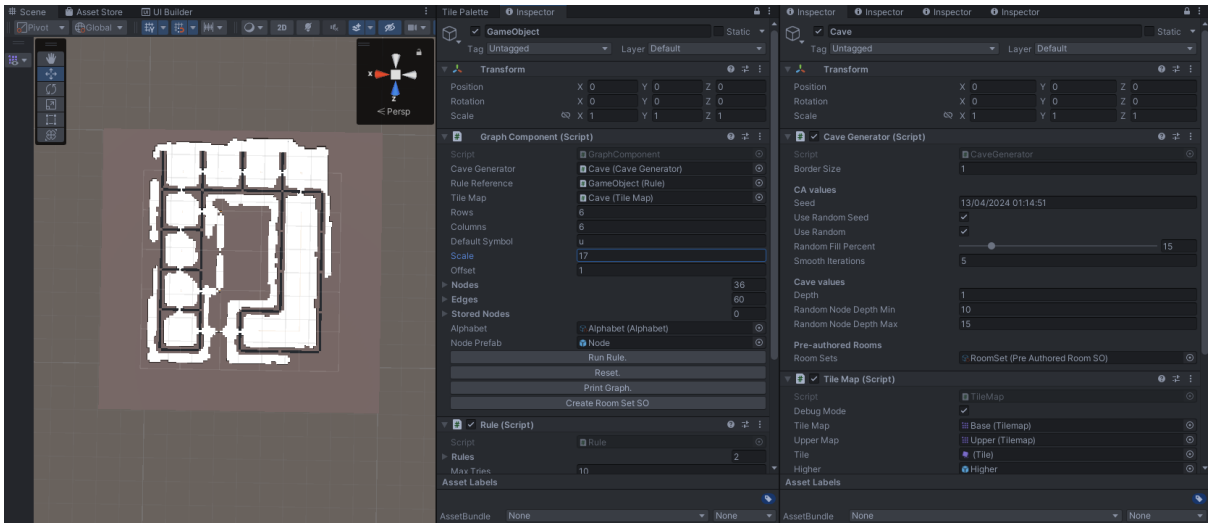


Figure 36: Low random fill percent of 15.

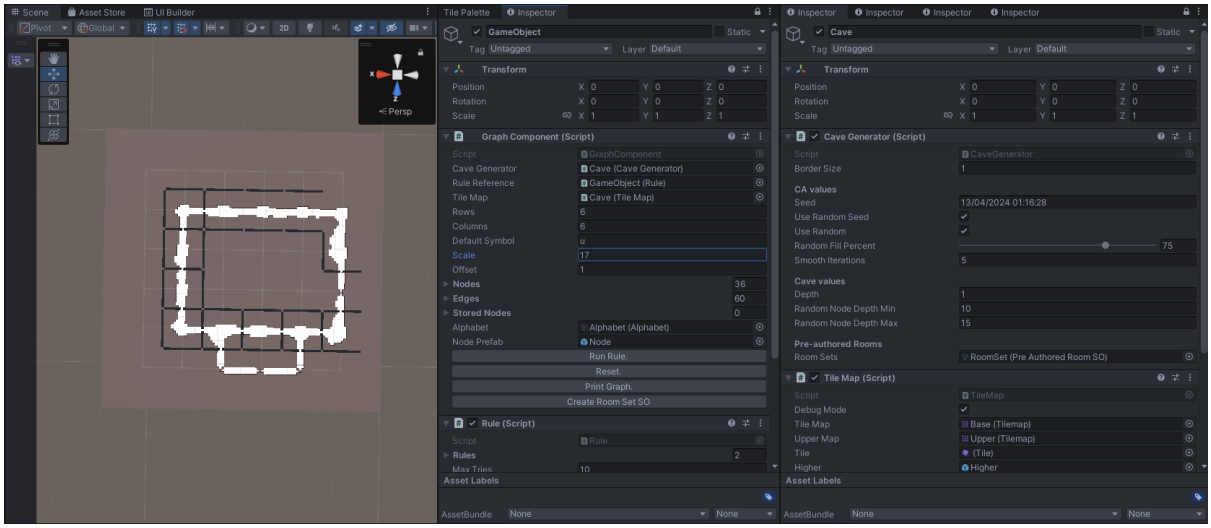


Figure 37: High random fill percent of 75.

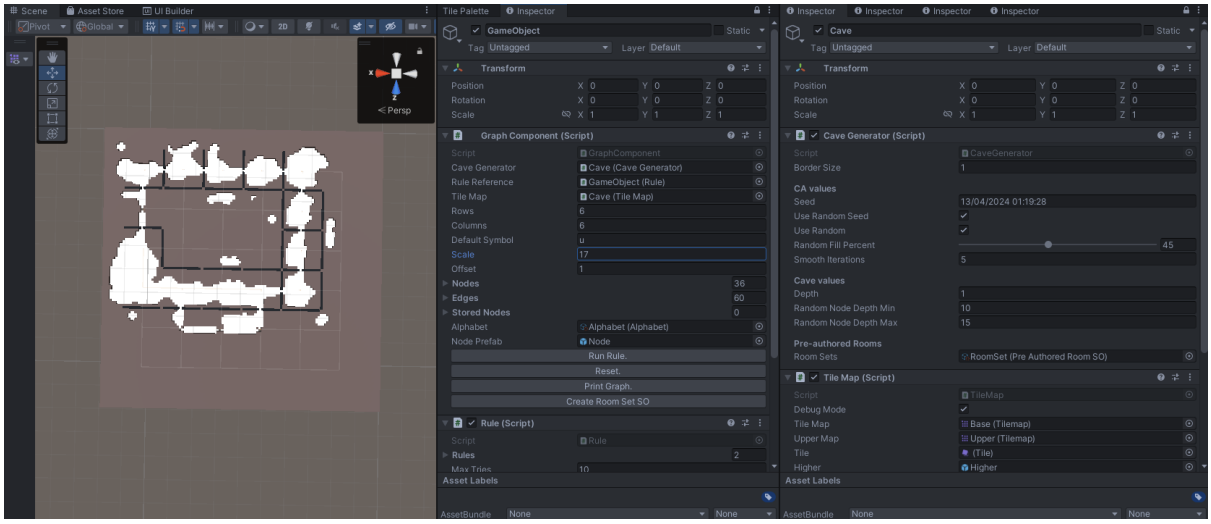


Figure 38: Low - middle random fill percent of 45.

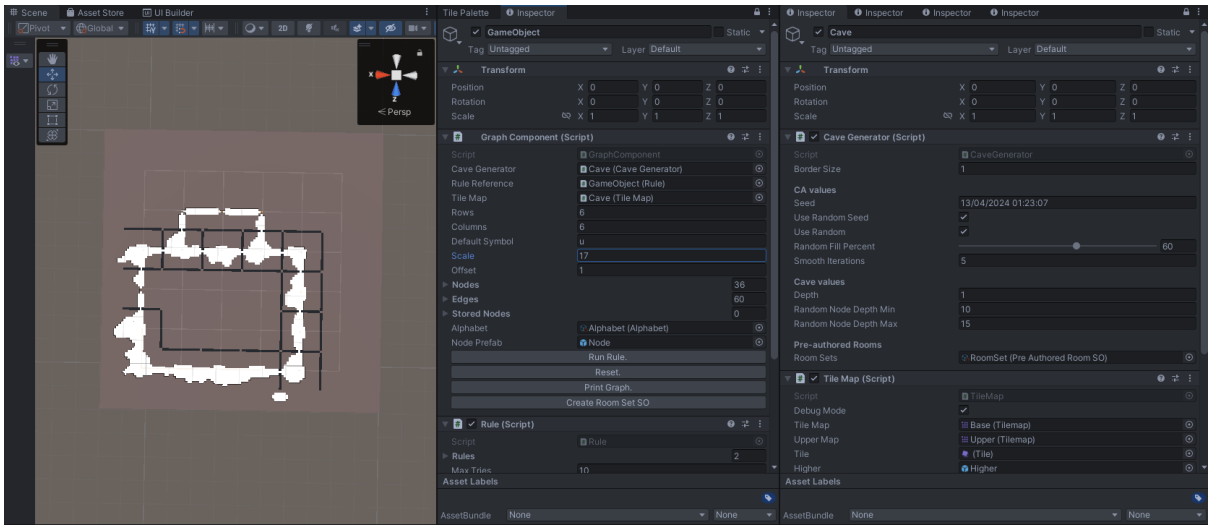


Figure 39: High- middle random fill percent of 60%.

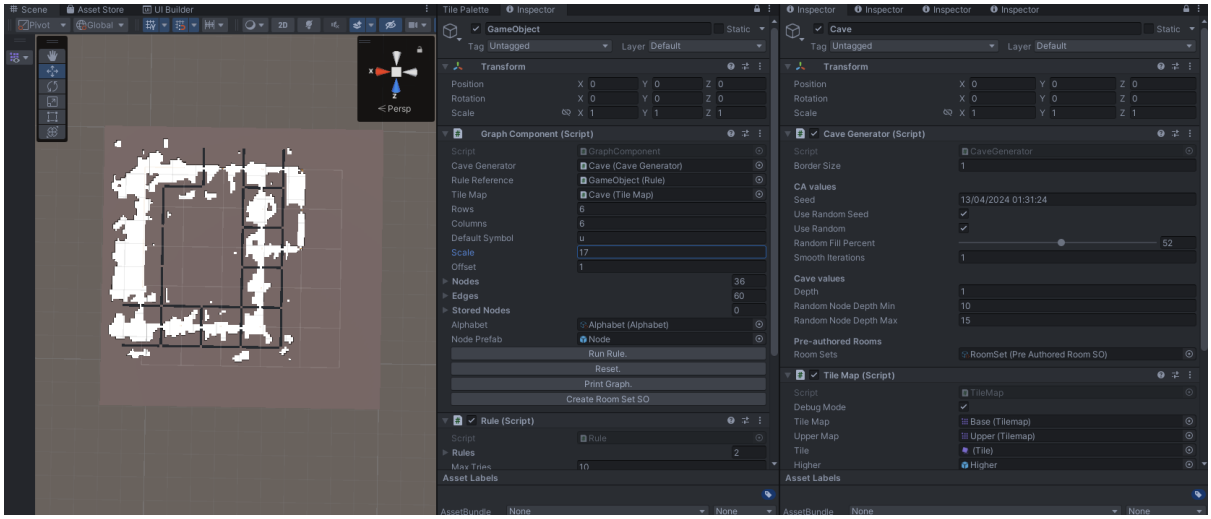


Figure 40: Low smooth iterations of 1.

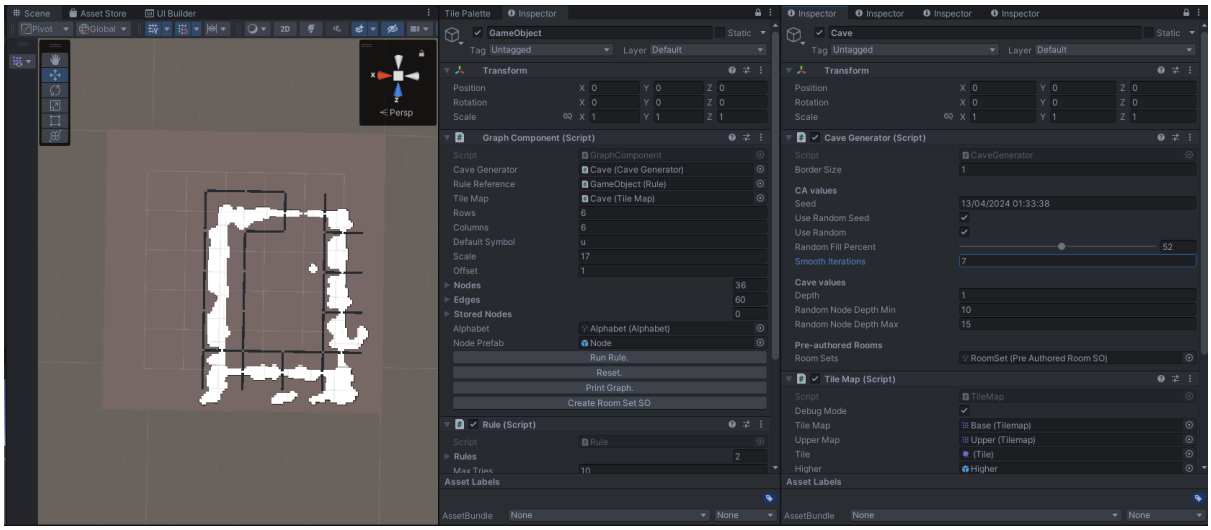


Figure 41: High smooth iterations of 7.

9. Results

Testing for this artifact was split into two, one focus was on testing alongside of development to allow for assessment of the successfulness of the framework, while the other test was to adjust key values of the tool to stress test its limits and find a range where the tool works best.

Testing during development allowed for flaws to be seen and adjusted to allow for the tool quality to increase. An example of this is test 4 in Table 1, having the CA caves be varied is a key part of the framework as if there was not enough variance then pre-made rooms would serve better at creating varied environments. Through the adjustments made from this test the tool can provide more varied generations, by giving the users more control. Another key test was tests 5 and 6, these tests focused on making sure features were working as expected. An example of this is dead zones not allowing short cuts to occur, this caused the introduction of a debug tool showing the dead zones which can help users understand the tool. Furthermore test 6 changed the application of CA randomness to the cells. This allowed for more concise CA generation and with the use of dead zones allowed the tool to successfully give users control over how neighbouring nodes influence each other. Tests 10, 11, 12 in Table 1 all show that the framework can add pre-authored rooms into the environment. Small changes were made to achieve this such as adjustments of the blocker's positions, but another change was added to improve the quality of life of the tool. This was the ability to create the scriptable object for room sets by pressing a button. This reduces the chance of error from occurring through user error. The feature of room sets working correctly is important as it builds upon the iterations the tool can provide by adding more depth into room types.

The fine-tuning adjustments tests explore the limits of the tool and how much impact the settings provided to the user affect the dungeon. Some settings explored in tests 1,2, 5, and 6 show that the dungeon can become very generic if extreme settings are used. This is no better seen in test 1 and 2 where any smaller or larger than the values used either causes the dungeon to break or cause performance issues. Meanwhile tests 5 and 6 show that while the dungeon may still be playable, it might not be enjoyable with generic layouts caused by extreme fill percent. Some settings have sweet spots as seen in tests 4,7 and 8 which depending on a user's needs can exceed these values, but the values explored showcase the best varied generations. This is one of the limitations of using CA as a generative tool as it is highly reliant on restrictions set by users.

Overall, these tests show that the tool can produce a varied dungeon environment. This indicates that the project was a success while providing users access to a customise the style of the dungeon. If not properly tested the user can end up breaking the dungeon if their settings are not finetuned.

10. Critical Evaluation

This project was a follow up on a previous masters project (Bennett, 2024) which explored the implementation of generative graph grammar, inspired from the tool called Ludoscope made by Dormans (N.D). This artifact takes this tool further by using the graph grammar to create physical environments through the use of CA and pre-authored rooms. The environments it creates are influenced by the roguelike genre and the dungeons that often feature within it.

The aim of the project was to research into ways of extending the variations of existing prototypes through the creation of a physical environment. This was achieved by implementing a CA algorithm to shape the environment based on user inputs. The CA system was able to extend the graph grammar system to create cave like formations and implement natural looking caves that mix well with the pre-authored rooms. Pre-authored rooms were the side of environment creation with the system successfully extending the iterations through the use of room sets. These allowed the option of full cave or full room dungeons or a mix of both allowing for higher variations. Despite this pre-authored rooms had some problems with them such as overlapping rooms, this issue was not solved due to the control of the room was given to the user. This could be explored in future projects to fully protect the user's generations from failing.

While this project did meet its original goal of increasing variations through introducing environments. It shares a common downside with most PCG tools which is control over the output. This control focuses on balancing values to make the best generation. This is further outlined in the fine-tuning adjustments tests which outlined the importance of testing values and their affects. Some values explored in the test shows that the range of effectiveness may be shallower than expected such as the random fill percentage. While this is something that could be adjusted to protect the user from using "bad" values, allowing the user to have full control over these values allows for more varied and enjoyable environments. Furthermore, user experience was not a key researched component, but taking the previous project's research into this allows for the usability of said tool to stay considered throughout the development process. This was reflected in the tests, with the optimisation of the use of the tool through the creation of room sets, further supporting the importance of exploring the limitations of tools to better understand how a user would use them.

Bibliography

Awati, R., 2021. *Cellular Automaton (CA).* [Online]
Available at: https://www.techtarget.com/searchenterprisedesktop/definition/cellular-automaton
[Accessed Feb 2024].

Bennett, W., 2024. *Enhancing Existing Prototype.,* s.l.: s.n.

Boris the Brave, 2019. *Dungeon Generation in Enter the Gungeon.* [Online]
Available at: https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/
[Accessed March 2023].

Boris, 2020. *Dungeon Generation in Binding of Isaac.* [Online]
Available at: https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/
[Accessed March 2024].

Dodge Roll, 2016. *Enter the Gungeon.* [Online]
Available at: https://store.steampowered.com/app/311690/Enter_the_Gungeon/
[Accessed March 2024].

Dormans, J., 2010. *Adventures in level design: Generating missions and spaces for action adventure games,* New York: Association for Computing Machinery.

Dormans, J., 2017. Cyclic Generation. In: T. A. Tanya X. Short, ed. *Procedural generation in games.* s.l.:CRC Press, pp. 83-95.

Dormans, J., N.D. *Ludoscope.* s.l.:s.n.

Flick, J., 2021. *Creating a Mesh.* [Online]
Available at: https://catlikecoding.com/unity/tutorials/procedural-meshes/creating-a-mesh/
[Accessed Febuary 2024].

GeeksForGeeks, 2023. *Test plan – Software Testing.* [Online]
Available at: https://www.geeksforgeeks.org/test-plan-software-testing/
[Accessed March 2024].

Lague, S., 2015. *Procedural Cave Generation.* [Online]
Available at: https://www.youtube.com/playlist?list=PLFt_AvWsXl0eZgMK_DT5_biRkWXftAOf9
[Accessed Febuary 2024].

Ludomotion, 2017. *Cyclic Dungeon Generation explained in 47 seconds.* [Online]
Available at: https://www.youtube.com/watch?v=_wvkTT-6P3Q&list=PLKrfA5RMeajfna6xcOe8WThxcDlxL58Xw
[Accessed March 2024].

Ludomotion, 2017. *Unexplored.* [Online]
Available at: https://store.steampowered.com/app/506870/Unexplored/
[Accessed December 2022].

Macedo, P. A. Y. & Chaimowicz, L., 2017. *Improving procedural 2D map Generation based on multi-layed cellular automata and Hilbert curves,* Curitiba: IEEE.

McMillen, E. & Himsl, F., 2011. *The Binding of Isaac.* [Online]
Available at: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/
[Accessed March 2024].

Ramalho, D., N.A.. *Marching Squares.* [Online]
Available at: https://ragingnexus.com/creative-code-lab/experiments/algorithms-marching-squares/#:~:text=Marching%20squares%20is%20a%20computer%20graphics%20algorithm%20that,scalar%20field%20%28rectangular%20array%20of%20individual%20numerical%20values%29.
[Accessed Feb 2024].

Rehkopf, M., n.d.. *What is a kanban board.* [Online]
Available at: https://www.atlassian.com/agile/kanban/boards
[Accessed December 2023].

Stiny, G. & Gips, J., 1971. *Shape Grammars and the Generative Specification of Painting and Sculpture,* s.l.: IFIP Congress.

Unity Technologies, 2022. *Tile component reference.* [Online]
Available at: https://docs.unity3d.com/Manual/class-Tilemap.html
[Accessed March 2024].

Unity Technologies, 2023. *Unity User Manual 2022.3.* [Online]
Available at: https://docs.unity3d.com/Manual/index.html
[Accessed Decemeber 2023].

Wrike, N.D. *What Is Agile Methodology in Project Management.* [Online]
Available at: https://www.wrike.com/project-management-guide/faq/what-is-agile-methodology-in-project-management/
[Accessed December 2023].