GENERATING GAME DATA TO SPACE

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

DTA
Masters of Negotiated Study

Contents

Figure Table

Table Index

Glossary

PCG – Procedural content generation

CA – Cellular Automata

PDS – Poisson disk sampling

BFS – Breadth first search

SO – Scriptable object

Key words
PCG, Graph rewriting, Game development

1. Introduction

Procedural content generation (PCG) is highly tied with the games industry due to its ability to speed up development. This is because its highly tied with tool development, which allows for countless generations to be produced by adjusting key values within tools. This is a key aspect that led to PCG to be used within the roguelike genre, as the genre highly relies on the replayability that PCG offers developers. This project follows up on a previous project "Expanding generative iterations of the prototype" (Bennett, 2024), and aims to implement Poisson disk sampling and jittered grid method to spawn entities within procedural rooms as well as apply a difficulty curve to the dungeon linked with progression based on user values.

1.1 Aim.

The prototype will be visualised with core assets and environments, alongside with a number of iterations for quality testing.

1.2 Objectives

- How Procedural environments are populated with entities.
- How entities are placed with awareness.
- Outline the different types of difficulty curves and their effects.
- Define the limitations of framework.

2. Project Methodology

The project was managed through the use of Agile (Wrike, n.d.). Agile allows for an iterative approach to project management, of which has different ways of being applied. This project uses a Kanban board (Rehkopf, n.d.). This allows for reflection on the project's progression and allowing the project to be scoped through smaller tasks. These smaller tasks are broken up from larger key goals of the project, with each of them having a key component which is a deadline. These smaller tasks can then be shared with a supervisor, which allows for easy visualisation of the projects progress since Kanban boards is a visual management tool.

3. Background

Procedural content generation (PCG) is an ever-evolving tool for the games industry. This paper follows on from two paper's developed tool. This tool began in "Enhancing previous prototype" (Bennett, 2024), which focused on creating a PCG tool using generative graph rewriting which was inspired from Dorman's Ludoscope software (N.D). The second paper "Expanding generative iterations of the prototype" (Bennett, 2024), focused on taking the graph created and creating an accurate representation using cellular automata and pre-authored rooms. This was inspired from roguelike games; particularly "The binding of Isaac" (McMillen & Himsl, 2011) and "Unexplored" (Ludomotion, 2017). After these projects, the framework as able to create a physical environment that reflects the generated graph.

4. Project Plan

With PCG being a rising technology in the games industry, it is important to be able to build other game systems around it. The aim of this project is to enhance the framework to be able to fill an environment with entities and apply a difficulty curve. This aims to support the downsides that PCG has which is offering linking game data to a procedural environment. While this is was also explored in previous projects, it was done through constrictive methods such as graph grammar (Bennett, 2023) and generating physical environment from said graph (Bennett, 2024) of which allows for gameplay considerations to be implemented into level design procedurally. This project will focus on additive methods to give more control to users to customise the output. This will be done through expanding on stored nodes and the layout the of environment itself.

5. Literature Review

Spawning data into space

Once an environment has been created through PCG, the next step is filling the environment with entities. There is a wide range of ways to approach this with approaches relying on the type of PCG algorithm used. One approach is to use pre-authored rooms that contain all data premade, this allows for developers to control the user's experience directly. Another way is through algorithms, these allow for entities to be placed inside designated areas randomly with constraints being applied to them to suit the environment.

Jittered-gird methods

Jittered grid is variation of the original jittering method, Kensler (2013) modified the method for the correlation of jittering of rows and columns. Red Blob Games (2018) discusses how jittered grid can be applied by modifying the x and y values with random values. They further state how this causes clumps and gaps in the generation, but it has high speed. This statement is backed by Baerentzen (2023) as they state it is efficient and simple formulation, but this was related to generating procedural textures.

Poisson disk sampling

Poisson disk sampling (PDS) is a sampling strategy that ensures each sample will keep a certain minimum distance from each other while being independent with a random distribution (2021, pp. 2-8). PDS has a wide array of applications in the games industry, this paper will focus on its use in random object placement, an example of this can be seen in Figure 1.

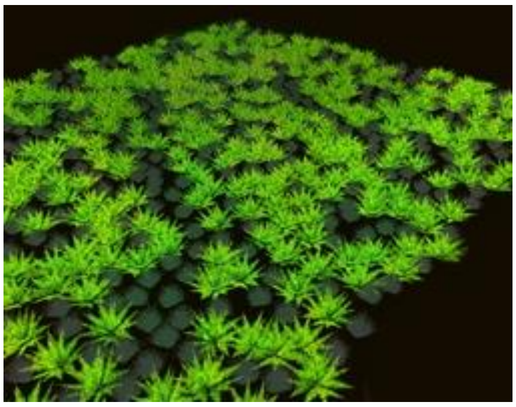


Figure 1: Example implementation of PDS placing shrubs (Tulleken, 2009).

The application of this algorithm allows for spaced entities to be added to a map, Yahya & et al. (2021) explored how PDS could be applied in conjunction with cellular automata (CA). Their findings showcase how it can be a powerful tool but must be constrained to not block player progression depending on what type of entity it is used to populate. Meanwhile Tulleken (2009), explored adding variation within PDS. This was achieved through the use of Perlin noise which drove the minimum distance. This caused clusters of objects, which he concluded could be a useful application to generating vegetation which can be seen in Figure 2.

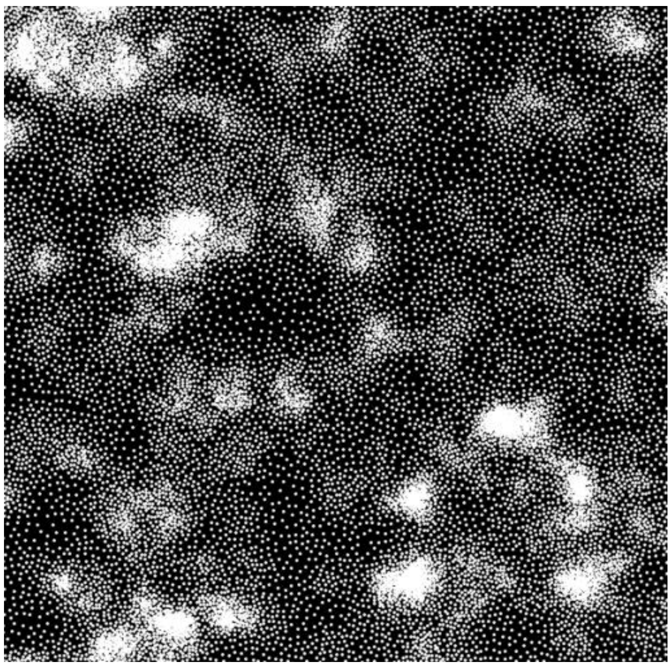


Figure 2: Poisson disk sample with minimum distance is driven by Perlin noise. (Tulleken, 2009)

Difficulty curve based on progress

Games have a wide array of difficulty, with each genre taking their own approach from the casual experience of Stardew Valley to the punishing enemies in the souls-like genre. One thing these games have in common is trying to achieve a flow state. Csikszentmihalyi (1990) explains an optimum state for the player to be fully immersed, full involvement, and enjoyment. A diagram showcasing this theory can be seen in Figure 3. While this is important to consider when designing games, it would be impossible to implement a solution that fits all games into a tool. That being said, giving developers the ability to attach their tools from the framework will be able to ease development with this framework.

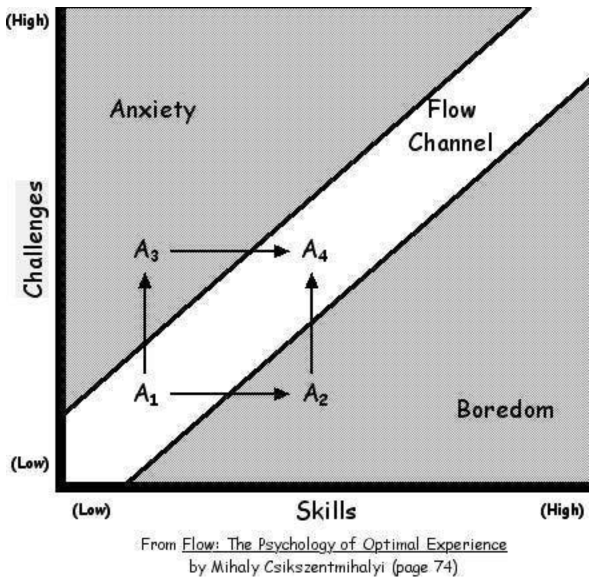

(High)
Anxiety
Flow Channel
Challenges
A3
A4
A1
A2
Boredom
(Low)
(Low)
Skills
(High)
From Flow: The Psychology of Optimal Experience
by Mihaly Csikszentmihalyi (page 74)

Figure 3: Diagram of Csikszentmihalyi flow channel (Icodewithben, 2023).

One common design of games is difficulty curves. These curves often allow designers to assign changing difficulties throughout the game, this allows the Csikszentmihalyi's flow channel to be achieved. These can be tightly linked with resource and enemy appearances, with changes in the curve directly impacting the spawn rates of these entities (Agiv, 2024). This is especially important with roguelikes as without the finetuned experience, the player may become bored or frustrated causing the replayability mechanic of the genre to be a hindrance.

Larsen (2010) discusses different ways difficulty curves can take shape, while he briefly describes some bad examples such as linear curves, he later moves onto curves that focus on varying the player's experience. In Figure 4, the difficulty will only progress upwards, this may cause the player to feel frustrated if it peaks over their skill ceiling, but it may also become predictable. To fix predictability using intervals of difficulty can avoid this as seen in Figure 5, but one downside of this is it may lead to an easier experience if the intervals are not carefully tested. Figure 6, showcases the best of both worlds with the experience having an upward curve to challenge the player while not being able to easily be predicted.

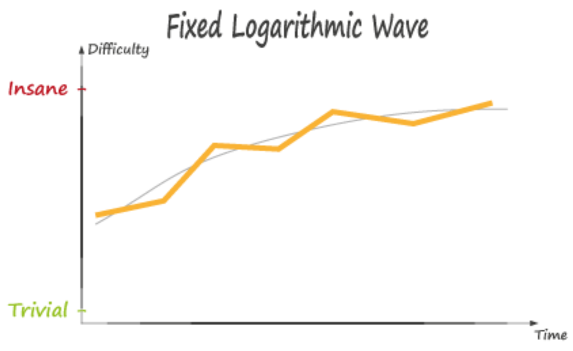


Figure 4: Fixed Logarithmic wave (Larsen, 2010).

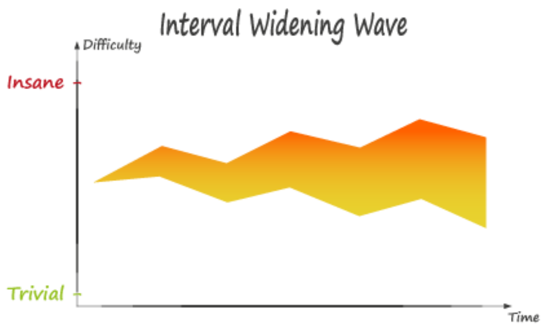


Figure 5: Interval widening wave (Larsen, 2010).

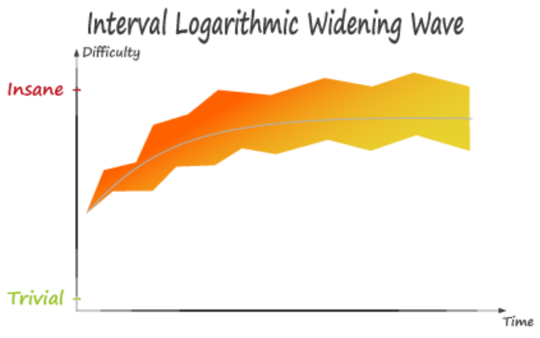


Figure 6: Interval Logarithmic Widening Wave (Larsen, 2010).

While Larsen explored the use of predetermined difficulty curves, Biemer (2023) explored the use of dynamic difficulty through the use of directors. Through their article they explore the concept of Markov Decision Process can be used to assemble player tailored levels. While this may not be applicable to this tool as the developer has a high amount of control over the generation of the level, it will be important when considering how developers approach dynamic difficulty.

Applying difficulty curves

Difficulty curves can be applied in a multitude of ways, one way is through progression. As the environment is procedural no path is certain, this is where pathfinding can be a crucial tool. The pathfinding algorithm will have to be adjusted since most algorithms find the shortest path. There are two algorithms that have to adjust the least: breadth first search (BFS) and depth first search (DFS) (GeeksForGeeks, 2024). Both offer a similar method of finding paths with BFS searching each cell's neighbouring cells. These neighbouring cells are added to a queue which are then later searched. DFS is similar in it searches the cells neighbours but instead of using a queue it uses a stack to traverse cells to the furthest point and then retraces back to the cell that has different neighbours. GeekForGeeks states that while both are traversal pathfinding algorithms, DFS is faster but is better suited for acyclic graphs. This is an issue due to the graph representing the map may be a cyclic map. This shows that BFS may be a better suited choice despite its slower performance due to it having to be altered less.

6. Design

Requirements

The framework should be able to support the addition of applying entities within the environment. This should make sense within the concept of the framework thus allowing for a seamless link to the graph grammar tool.

Requirement	Purpose of requirement
Stored nodes can represent entities being placed in the environment.	This will allow entity spawning to cohesively mix in with the existing framework thus giving a consistent user experience.
Able to spawn entities through Poisson or Jitter	Being able to choose between two options will allow users to pick the algorithm that suits their game best.
Ability to place entities within clear space of the environment.	This will ensure that the techniques used are able to place entities in valid areas. This ensures the quality of the tool.
Being able to apply a difficulty curve.	The tool will be able to automatically apply difficulty based on progression or via rules to the graph.

Entity Spawning

Entity spawning will be linked with the physical space of environment as it will be able to pass on constraints for the placement of said entities. An example of this is placing entities in valid empty spaces, an empty cell was designated as 0 or less as defined in the previous project (Bennett, 2024). This idea will be combined with jittered grid and PDS to place entities within the environment. These will be tested against each other to see which method is best suited for the framework.

Difficulty curves

This pathfinding algorithm will be restricted to searching through the graph itself to increase efficiency of the search, when compared to searching through the tilemap environment. For this project breadth first search has been chosen due to the smaller size of the graph meaning performance of the search is not a large downside. Especially as both BFS and DFS have the same time complexity of O (V + E) (GeeksForGeeks, 2024). To allow users to enter data for the curves Unity's API called "AnimationCurve" (Unity Technologies, 2024) will be used which can be seen in Figure 7.

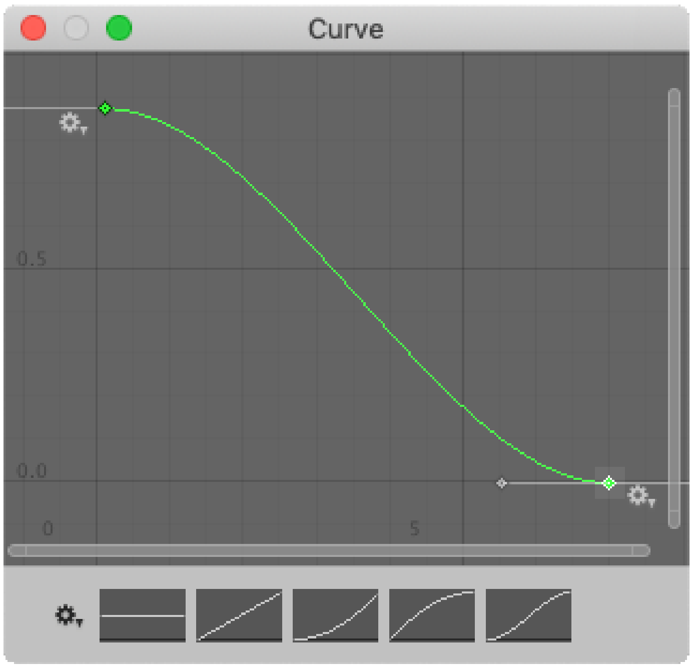


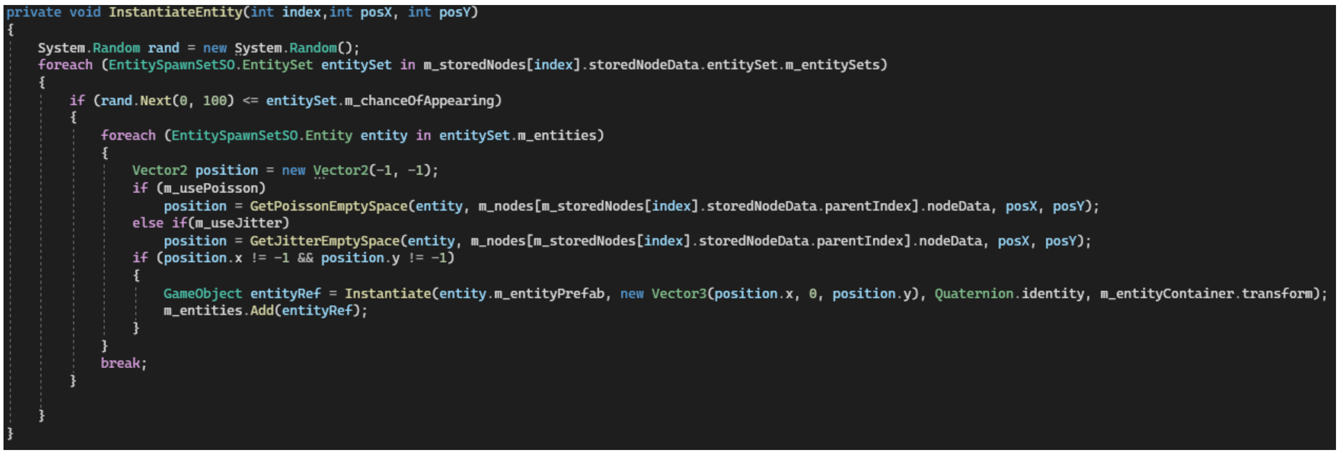
Figure 7: Example of an animation curve in Unity (Unity Technologies, 2019).

7. Implementation

Spawning Entities

Spawning entities within a procedural environment can pose more challenges than pre-authored environments due to the shape of the area is not predetermined. To do this the area needs to be surveyed to check for a valid position to spawn said entities. On top of this, to keep replayability high the placement of said entities should be randomised. This is where Poisson disk sampling and jittered grid method are able to provide solutions to said problems.

After the environment is generated, stored nodes are passed from the “GraphComponent” to the “EntitySpawner”. The method seen in Figure 8, is passed each stored node in the graph, with the scriptable object assigned to it, the data is passed to be instantiated into the scene. Depending on the option the user has set, this data is passed to “GetPoissonEmptySpace” or “GetJ”tterEmptySpace".

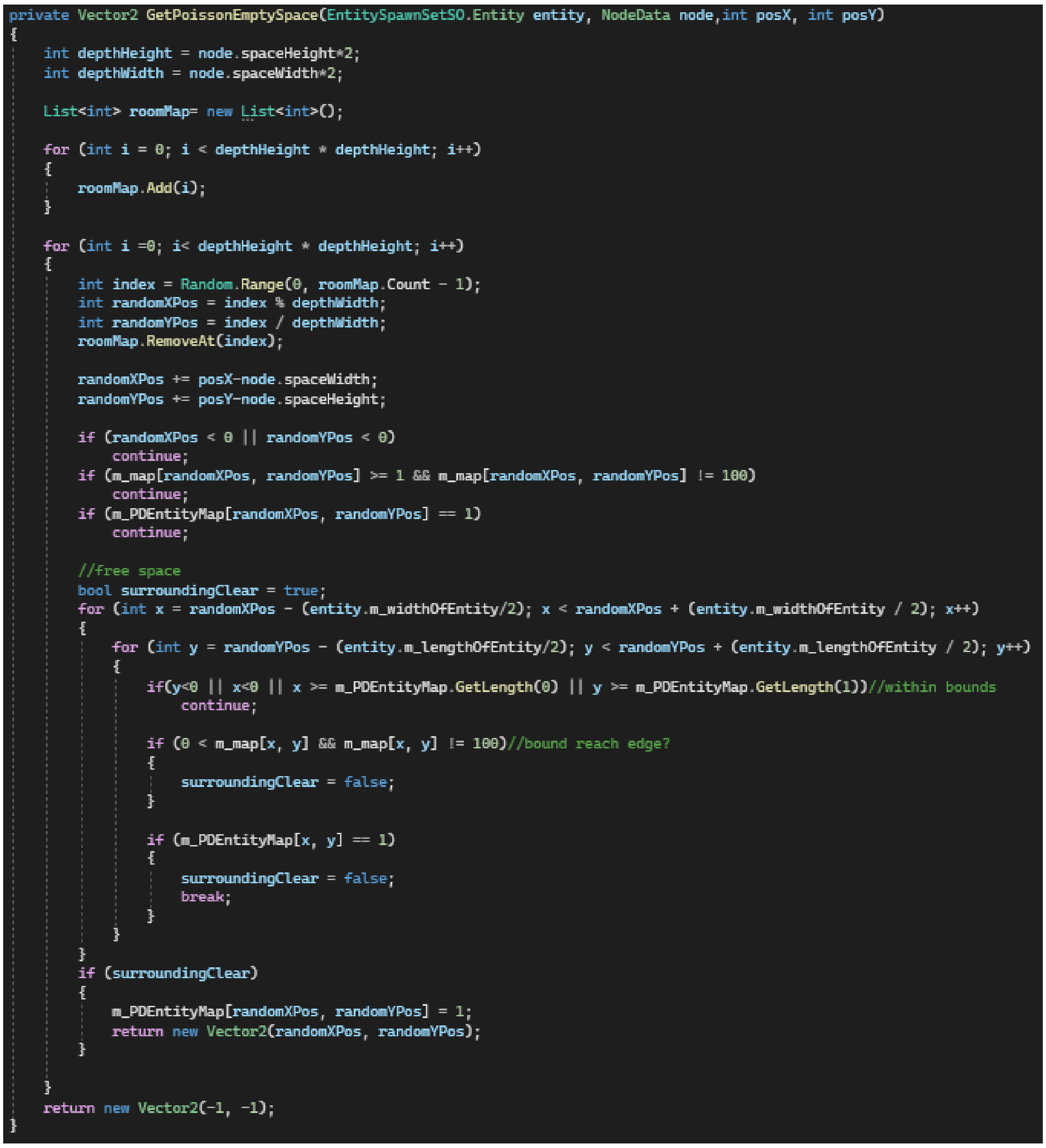


```
private void InstantiateEntity(int index,int posX, int posY)
{
    System.Random rand = new System.Random();
    foreach (EntitySpawnSetSO.EntitySet entitySet in m_storedNodes[index].storedNodeData.entitySet.m_entitySets)
    {
        if (rand.Next(0, 100) <= entitySet.m_chanceOfAppearing)
        {
            foreach (EntitySpawnSetSO.Entity entity in entitySet.m_entities)
            {
                Vector2 position = new Vector2(-1, -1);
                if (m_usePoisson)
                    position = GetPoissonEmptySpace(entity, m_nodes[m_storedNodes[index].storedNodeData.parentIndex].nodeData, posX, posY);
                else if(m_useJitter)
                    position = GetJitterEmptySpace(entity, m_nodes[m_storedNodes[index].storedNodeData.parentIndex].nodeData, posX, posY);
                if (position.x != -1 && position.y != -1)
                {
                    GameObject entityRef = Instantiate(entity.m_entityPrefab, new Vector3(position.x, 0, position.y), Quaternion.identity, m_entityContainer.transform);
                    m_entities.Add(entityRef);
                }
            }
            break;
        }

    }
}
```

Figure 8: InstantiateEntity method.

Poisson disk sampling

Poisson disk sampling solves an issue of spacing entities and placing them in random positions. This method creates a room map of all positions within the room area. This is then used as a counter, as when a position is checked and invalidated, it is removed from the map. This allows for random positions to be generated without double checking the same position. Once a valid open space is found the entity’s width and length is used to check if the area around it is clear of other entities which is stored in “m_entityMap”. This entity map uses a similar approach as CA used in “Expanding generative iterations of the prototype.” (Bennett, 2024), as valid spawns are represented as “0” while invalid positions are “1”.

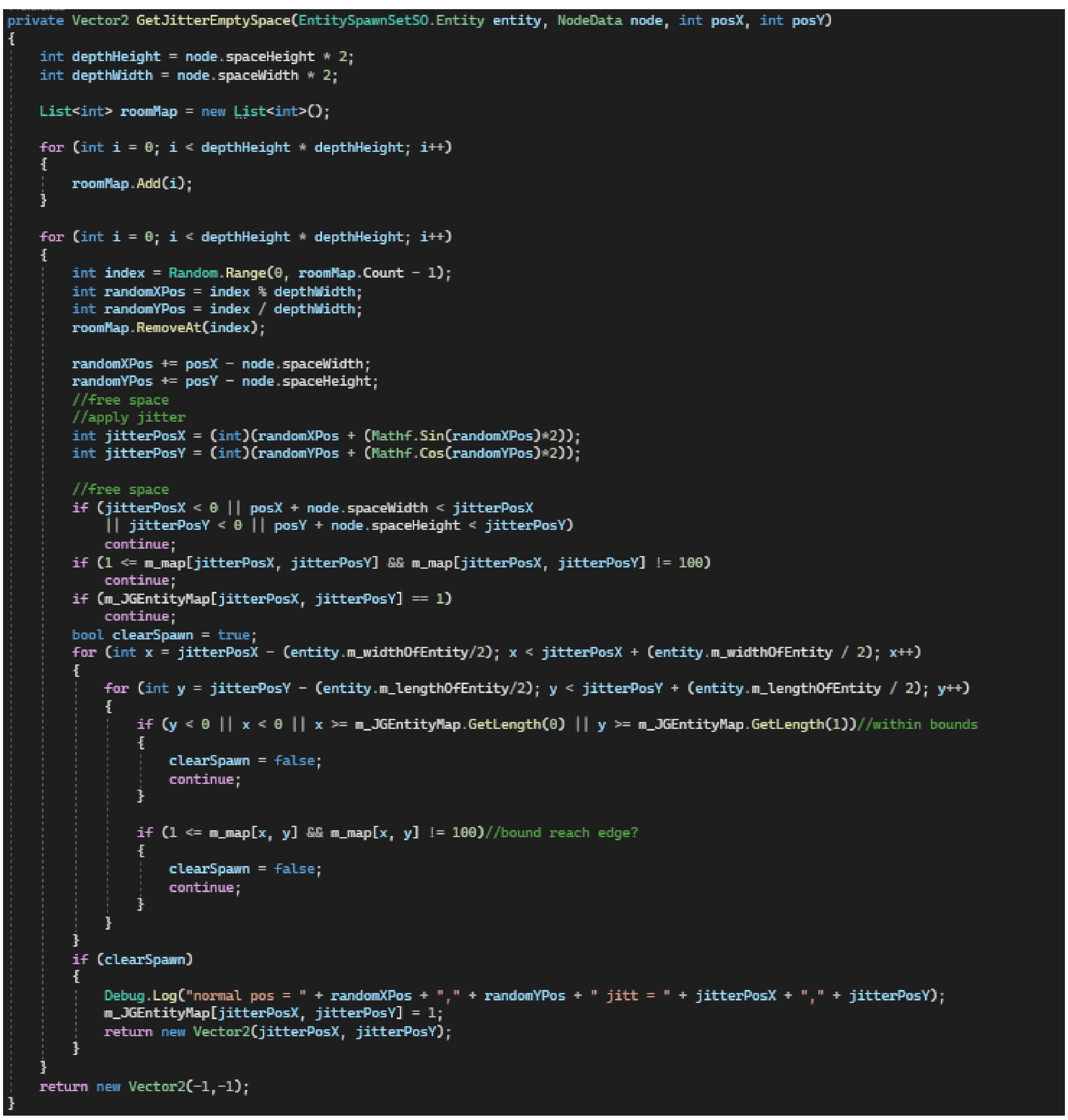


```
private Vector2 GetPoissonEmptySpace(EntitySpawnSetSO.Entity entity, NodeData node,int posX, int posY)
{
    int depthHeight = node.spaceHeight*2;
    int depthWidth = node.spaceWidth*2;

    List<int> roomMap= new List<int>();

    for (int i = 0; i < depthHeight * depthHeight; i++)
    {
        roomMap.Add(i);
    }

    for (int i =0; i< depthHeight * depthHeight; i++)
    {
        int index = Random.Range(0, roomMap.Count - 1);
        int randomXPos = index % depthWidth;
        int randomYPos = index / depthWidth;
        roomMap.RemoveAt(index);

        randomXPos += posX-node.spaceWidth;
        randomYPos += posY-node.spaceHeight;

        if (randomXPos < 0 || randomYPos < 0)
            continue;
        if (m_map[randomXPos, randomYPos] >= 1 && m_map[randomXPos, randomYPos] != 100)
            continue;
        if (m_PDEntityMap[randomXPos, randomYPos] == 1)
            continue;

        //free space
        bool surroundingClear = true;
        for (int x = randomXPos - (entity.m_widthOfEntity/2); x < randomXPos + (entity.m_widthOfEntity / 2); x++)
        {
            for (int y = randomYPos - (entity.m_lengthOfEntity/2); y < randomYPos + (entity.m_lengthOfEntity / 2); y++)
            {
                if(y<0 || x<0 || x >= m_PDEntityMap.GetLength(0) || y >= m_PDEntityMap.GetLength(1))//within bounds
                    continue;

                if (0 < m_map[x, y] && m_map[x, y] != 100)//bound reach edge?
                {
                    surroundingClear = false;
                }

                if (m_PDEntityMap[x, y] == 1)
                {
                    surroundingClear = false;
                    break;
                }
            }
        }
        if (surroundingClear)
        {
            m_PDEntityMap[randomXPos, randomYPos] = 1;
            return new Vector2(randomXPos, randomYPos);
        }

    }
    return new Vector2(-1, -1);
}
```

Figure 9: GetPoissonEmptySpace method.

Jittered-grid methods

As Baerentzen (2023) mentions, jittered-grid method allows for the quick application of entities within a scene. This is due to the random position that is used is randomised in a consistent way to further add variety. As seen in Figure 10, the jittering is done via the use of sine and cosine waves as they are continuous, causing the discontinuous matrix. Despite it being a quicker method, it needs the additional checks to assure the validity of the placement. Since the jittered x and y coordinates are used to check if a cell is valid, the jittered position needs to be within the maps bounds. Furthermore, the speed of the check is slowed by checking if any wall is within the entities area which assures quality.

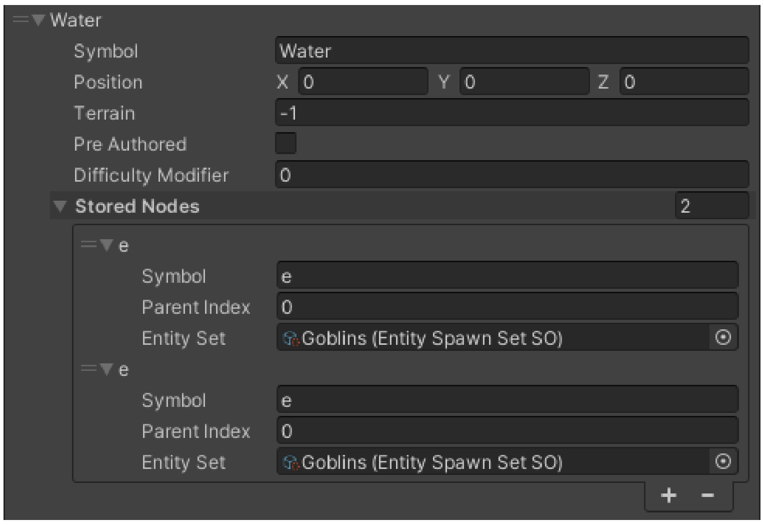


```
private Vector2 GetJitterEmptySpace(EntitySpawnSetSO.Entity entity, NodeData node, int posX, int posY)
{
    int depthHeight = node.spaceHeight * 2;
    int depthWidth = node.spaceWidth * 2;

    List<int> roomMap = new List<int>();

    for (int i = 0; i < depthHeight * depthHeight; i++)
    {
        roomMap.Add(i);
    }

    for (int i = 0; i < depthHeight * depthHeight; i++)
    {
        int index = Random.Range(0, roomMap.Count - 1);
        int randomXPos = index % depthWidth;
        int randomYPos = index / depthWidth;
        roomMap.RemoveAt(index);

        randomXPos += posX - node.spaceWidth;
        randomYPos += posY - node.spaceHeight;
        //free space
        //apply jitter
        int jitterPosX = (int)(randomXPos + (Mathf.Sin(randomXPos)*2));
        int jitterPosY = (int)(randomYPos + (Mathf.Cos(randomYPos)*2));

        //free space
        if (jitterPosX < 0 || posX + node.spaceWidth < jitterPosX
            || jitterPosY < 0 || posY + node.spaceHeight < jitterPosY)
            continue;
        if (1 <= m_map[jitterPosX, jitterPosY] && m_map[jitterPosX, jitterPosY] != 100)
            continue;
        if (m_JGEntityMap[jitterPosX, jitterPosY] == 1)
            continue;
        bool clearSpawn = true;
        for (int x = jitterPosX - (entity.m_widthOfEntity/2); x < jitterPosX + (entity.m_widthOfEntity / 2); x++)
        {
            for (int y = jitterPosY - (entity.m_lengthOfEntity/2); y < jitterPosY + (entity.m_lengthOfEntity / 2); y++)
            {
                if (y < 0 || x < 0 || x >= m_JGEntityMap.GetLength(0) || y >= m_JGEntityMap.GetLength(1))//within bounds
                {
                    clearSpawn = false;
                    continue;
                }

                if (1 <= m_map[x, y] && m_map[x, y] != 100)//bound reach edge?
                {
                    clearSpawn = false;
                    continue;
                }
            }
        }
        if (clearSpawn)
        {
            Debug.Log("normal pos = " + randomXPos + "," + randomYPos + " jitt = " + jitterPosX + "," + jitterPosY);
            m_JGEntityMap[jitterPosX, jitterPosY] = 1;
            return new Vector2(jitterPosX, jitterPosY);
        }
    }
    return new Vector2(-1,-1);
}
```

Figure 10: GetJitterEmptySpace method.

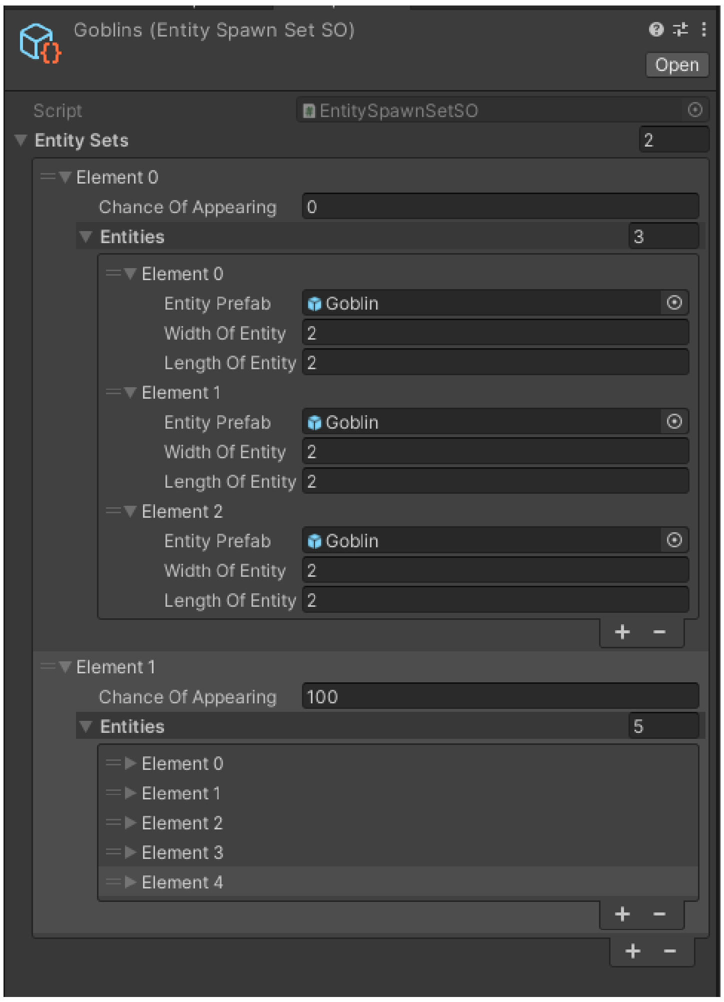
Attachment to stored nodes

To allow users to link which entities should spawn with the graph, stored nodes have been used due to their pre-existing link with the graph. As seen in Figure 11, the list of stored nodes attached to a node can be adjusted within the right hand of the rule. The scriptable object “Goblins” can be seen in Figure 12, this showcases how each set of entities have a chance of being selected to be applied, with the last index needing to be set to 100 to act as a default application. Within each set of entities is a list of entities that will be spawned. This links a prefab, width, and length, the prefab allows any object to be set from an enemy or a chest. Meanwhile the width and length describe the size of the entity and the space it will need to spawn, as seen in previous sections.



Water
Symbol: Water
Position: X 0 Y 0 Z 0
Terrain: -1
Pre Authored
Difficulty Modifier: 0
Stored Nodes: 2
e
Symbol: e
Parent Index: 0
Entity Set: Goblins (Entity Spawn Set SO)
e
Symbol: e
Parent Index: 0
Entity Set: Goblins (Entity Spawn Set SO)

Figure 11: Example of linking entity SO to stored nodes within rule right hand.

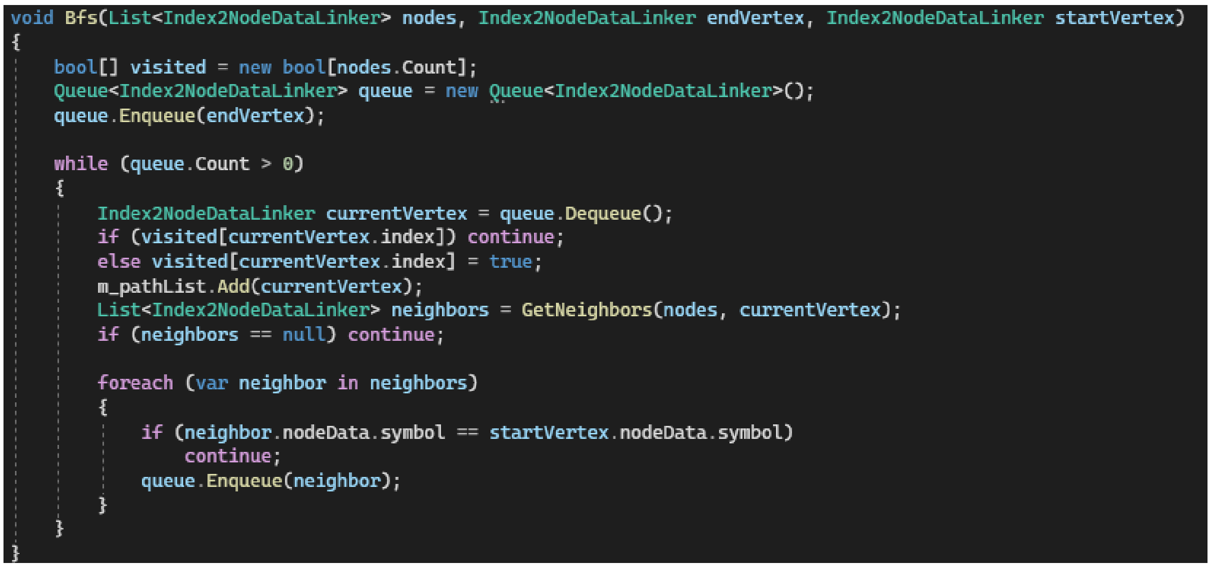


Goblins (Entity Spawn Set SO)
Open
Script: EntitySpawnSetSO
Entity Sets: 2
Element 0
Chance Of Appearing: 0
Entities: 3
Element 0
Entity Prefab: Goblin
Width Of Entity: 2
Length Of Entity: 2
Element 1
Entity Prefab: Goblin
Width Of Entity: 2
Length Of Entity: 2
Element 2
Entity Prefab: Goblin
Width Of Entity: 2
Length Of Entity: 2
Element 1
Chance Of Appearing: 100
Entities: 5
Element 0
Element 1
Element 2
Element 3
Element 4

Figure 12: Entity spawn set scriptable object.

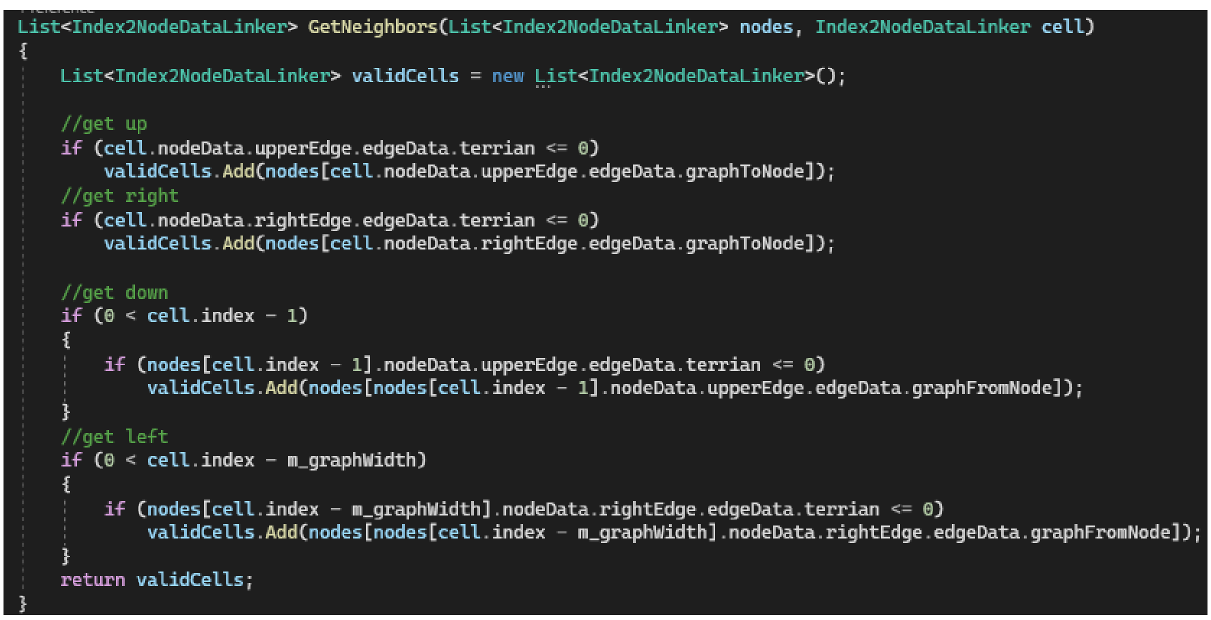
Difficulty Curve

Procedural environments have a key aspect which is ever changing layouts. This means no path is guaranteed, which makes applying difficulty based on progression a challenge. As mentioned in Applying difficulty curves, pathfinding offers a good solution to this problem. BFS is best suited since the map is small and needs to be adjusted the least to be able to handle the possibility of cyclic graphs. The BFS algorithm queues from the end node and finds its neighbours and queues them; ignoring the start node, this can be seen in Figure 13. To find the neighbours, the same method from "CaveGenerator" is used for checking if doorways need to be blocked which can be seen in Figure 14. When a neighbouring edge whose terrain is open is found, it gets added to a list to be checked. After all valid nodes have been found the path list is reversed and the start is then added.



```
void Bfs(List<Index2NodeDataLinker> nodes, Index2NodeDataLinker endVertex, Index2NodeDataLinker startVertex)
{
    bool[] visited = new bool[nodes.Count];
    Queue<Index2NodeDataLinker> queue = new Queue<Index2NodeDataLinker>();
    queue.Enqueue(endVertex);

    while (queue.Count > 0)
    {
        Index2NodeDataLinker currentVertex = queue.Dequeue();
        if (visited[currentVertex.index]) continue;
        else visited[currentVertex.index] = true;
        m_pathList.Add(currentVertex);
        List<Index2NodeDataLinker> neighbors = GetNeighbors(nodes, currentVertex);
        if (neighbors == null) continue;

        foreach (var neighbor in neighbors)
        {
            if (neighbor.nodeData.symbol == startVertex.nodeData.symbol)
                continue;
            queue.Enqueue(neighbor);
        }
    }
}
```

Figure 13: BFS method.



```
List<Index2NodeDataLinker> GetNeighbors(List<Index2NodeDataLinker> nodes, Index2NodeDataLinker cell)
{
    List<Index2NodeDataLinker> validCells = new List<Index2NodeDataLinker>();

    //get up
    if (cell.nodeData.upperEdge.edgeData.terrian <= 0)
        validCells.Add(nodes[cell.nodeData.upperEdge.edgeData.graphToNode]);
    //get right
    if (cell.nodeData.rightEdge.edgeData.terrian <= 0)
        validCells.Add(nodes[cell.nodeData.rightEdge.edgeData.graphToNode]);

    //get down
    if (0 < cell.index - 1)
    {
        if (nodes[cell.index - 1].nodeData.upperEdge.edgeData.terrian <= 0)
            validCells.Add(nodes[nodes[cell.index - 1].nodeData.upperEdge.edgeData.graphFromNode]);
    }
    //get left
    if (0 < cell.index - m_graphWidth)
    {
        if (nodes[cell.index - m_graphWidth].nodeData.rightEdge.edgeData.terrian <= 0)
            validCells.Add(nodes[nodes[cell.index - m_graphWidth].nodeData.rightEdge.edgeData.graphFromNode]);
    }
    return validCells;
}
```

Figure 14: GetNeighbours method.

After the path has been found the difficulty curve needs to be applied. An example of this can be seen in Figure 15, this allows the user to set how the difficulty progresses. This curve is inspired by Larsen's (2010) diagrams of logarithmic difficulty curves. The data within this curve is then applied to the path's nodes within the "nodeData". This data is applied by creating a fraction out of the node's index from the total amount of nodes. This fraction is then used as an index to evaluate the value on the graph at that value. An example of this would be at0.5 the value would be 10 therefore a node halfway through the path would have a difficulty rating of 10. Additionally, applying Larsen's logic of interval graph values, the user can set the interval values as seen in Figure 16, this is applied to the difficulty value to add some variety to the progression, with Figure 17 showing how these options can be toggled. On top of this users can artificially modify the value of the difficulty in the rules as seen in Figure 18. This allows for set nodes/rooms to have difficulty separated from the difficulty graph such as shops or challenge rooms. An additional option to not apply the graph difficulty allows the user to have more control over the difficulty through rules.

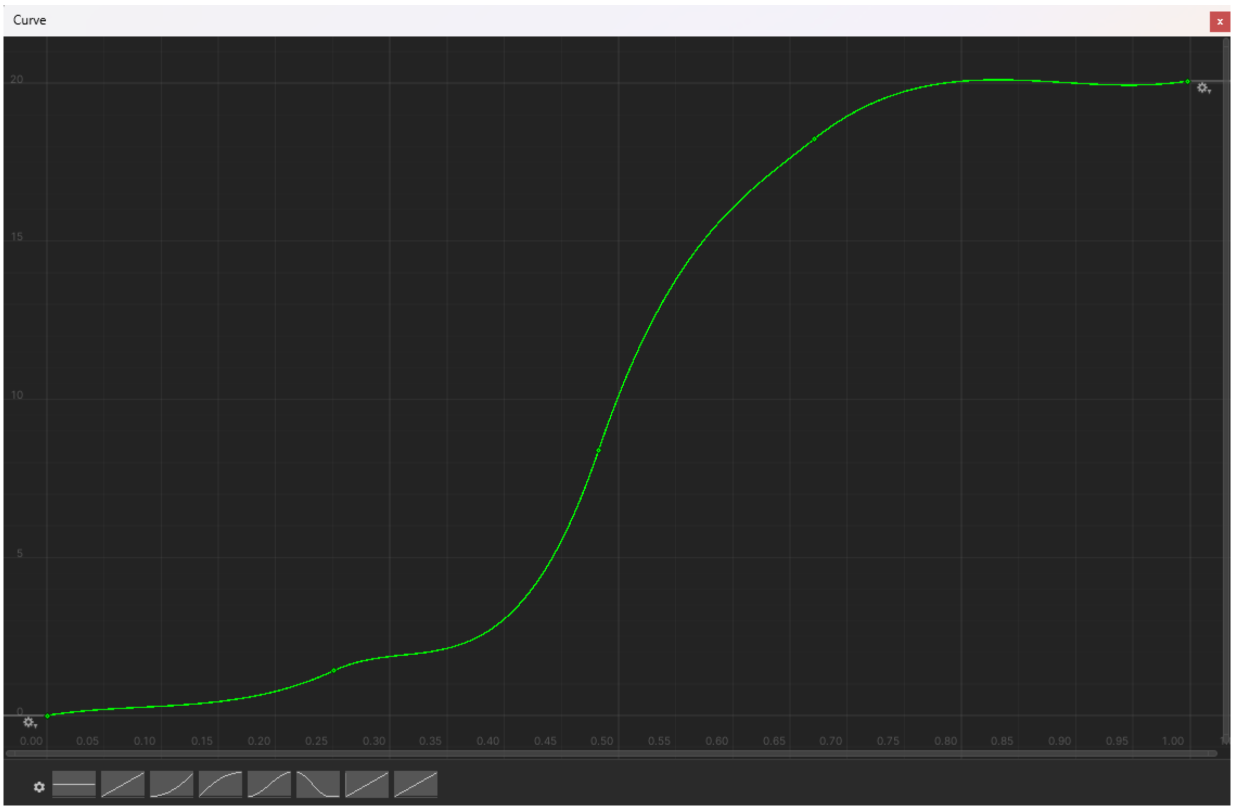
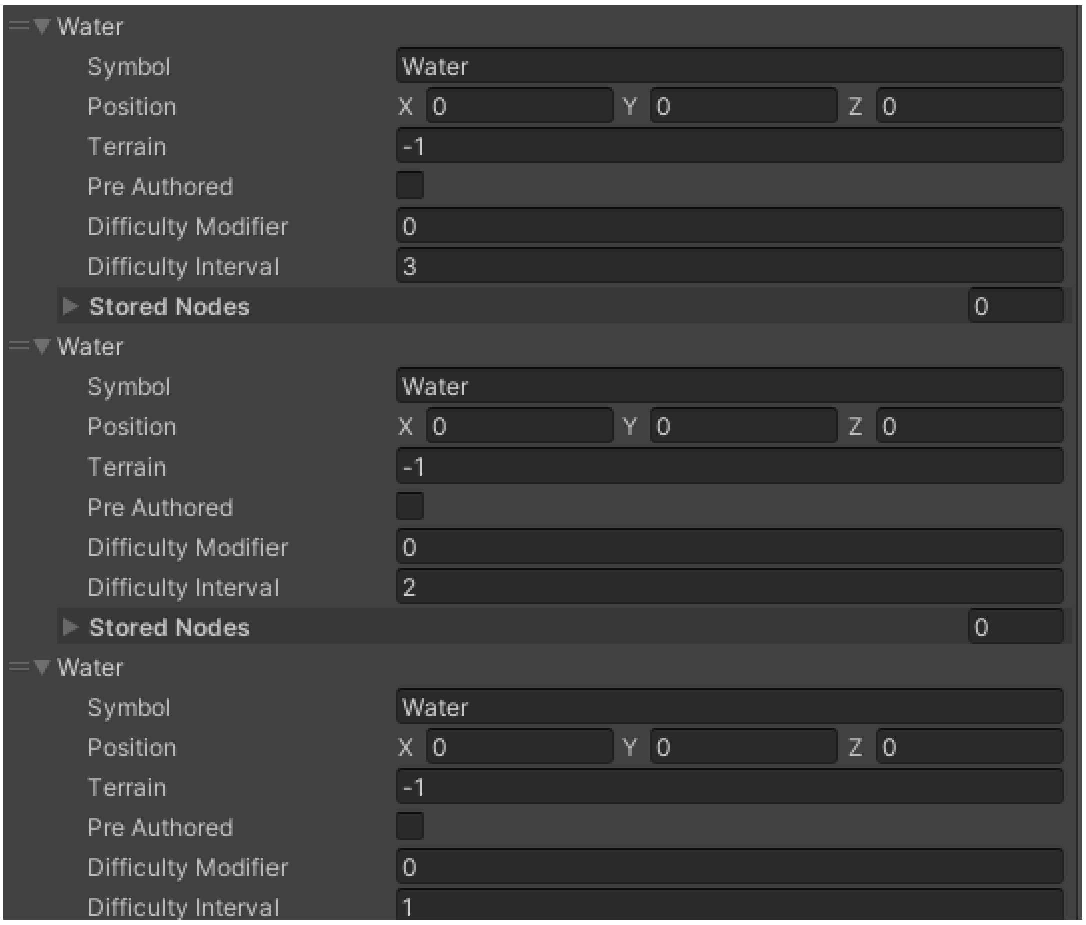


Figure 15: Example of a difficulty curve.

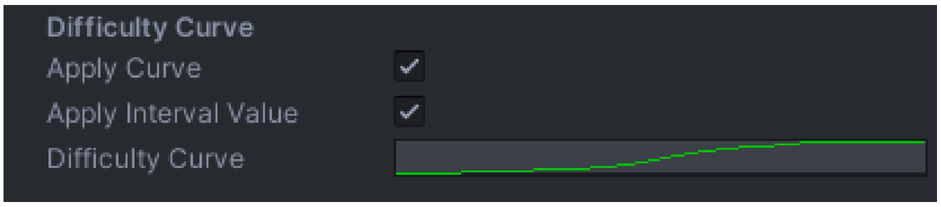


Water
Symbol: Water
Position: X 0 Y 0 Z 0
Terrain: -1
Pre Authored: ☐
Difficulty Modifier: 0
Difficulty Interval: 3
Stored Nodes: 0

Water
Symbol: Water
Position: X 0 Y 0 Z 0
Terrain: -1
Pre Authored: ☐
Difficulty Modifier: 0
Difficulty Interval: 2
Stored Nodes: 0

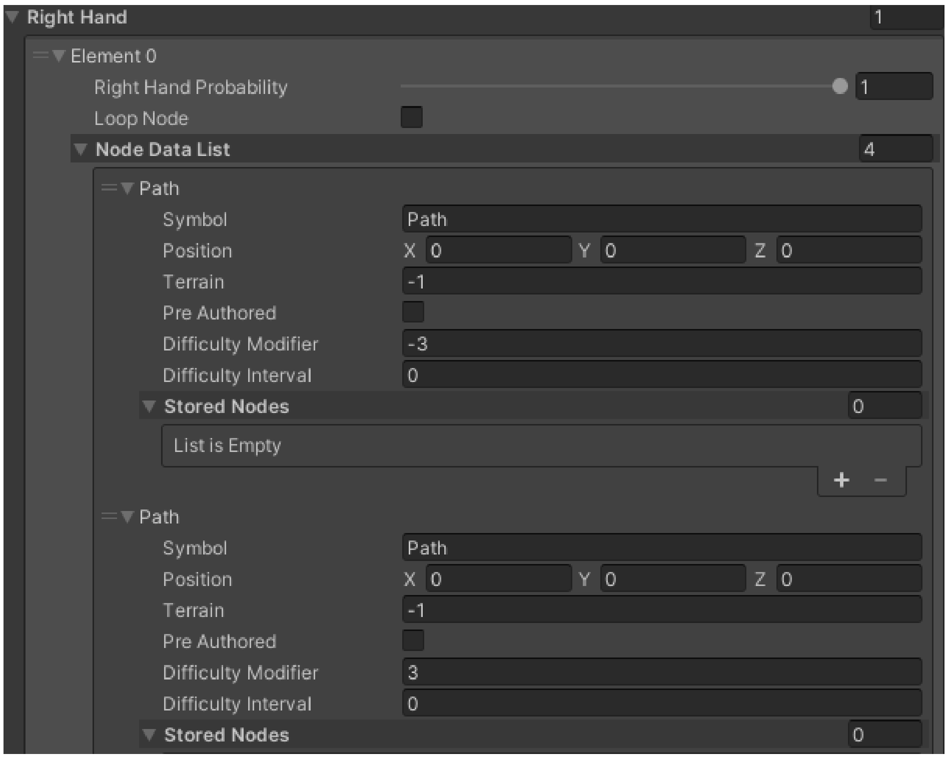
Water
Symbol: Water
Position: X 0 Y 0 Z 0
Terrain: -1
Pre Authored: ☐
Difficulty Modifier: 0
Difficulty Interval: 1

Figure 16:Screenshot of difficulty interval in CreatePath rule.



Difficulty Curve
Apply Curve: ✓
Apply Interval Value: ✓
Difficulty Curve

Figure 17: Screenshot of options for difficulty curve



Right Hand 1

Element 0

Right Hand Probability 1

Loop Node

Node Data List 4

Path

Symbol Path

Position X 0 Y 0 Z 0

Terrain -1

Pre Authored

Difficulty Modifier -3

Difficulty Interval 0

Stored Nodes 0

List is Empty

Path

Symbol Path

Position X 0 Y 0 Z 0

Terrain -1

Pre Authored

Difficulty Modifier 3

Difficulty Interval 0

Stored Nodes 0

Figure 18: Screenshot of difficulty modifier.

8. Testing

Testing for this project has been split into two segments, a test plan and testing the difference between entity spawning. The test plan outlines key tests that was done throughout the project, while being adapted with shifted elements as the project was developed. This allows for the project to be scoped, while including any issues found. The key tests will allow for the analysis of the project as they can be linked with project's aim and objectives. Testing the difference between entity spawning will allow to see the strengths and weaknesses between the two methods. This can be combined with fine-tuning adjustments to find the limitations of the methods. These can then be linked to the successfulness tool.

Test Plan

Table 1: Test plan

Test NO	Test description	Expected result	Actual result	Required actions
1	Entity data can be linked with stored nodes.	Entity set scriptable object is able to be linked correctly with stored nodes.	Entity set scriptable object is able to be linked with stored nodes.	No actions are required.
2	Entity data is accurately placed in environment.	Entities are placed randomly within the environment.	Entities are placed in environment but less are placed than expected.	Use of an array to store cells the entity has been placed to stop repeat tries.
3	Poisson disk sampling can place entities.	Poisson disk was able to place entities based on their size.	Poisson disk was able to place entities based on size constraints.	No actions are required.
4	Jittered grid can place entities.	Jittered grid can place entities.	Jittered grid can place entities, but they were clipping walls.	Add method to check around the entity for wall space.
5	BFS can find all valid nodes.	BFS can return all valid nodes within the graph.	BFS was able to return all valid nodes from the graph.	No actions are required.
6	Difficulty curve is able to adjust to the size of the path.	Difficulty curve is able to scale correctly based on the path given.	The curve was not able to adjust correctly.	Use fractions of the path progression to find the value from the curve.
7	Difficulty curve is accurately presented across the environment.	The curve created by user is applied to the graph based on the BFS paths.	The curve was able to be applied accurately.	No actions are required.
8	Difficulty intervals can be applied.	Intervals are used to get a random range to apply to the difficulty value.	The difficulty value is adjusted based on a random value from the interval.	No actions are required.

Fine Tuning Adjustments

Table 2: Fine tuning adjustments.

Adjustment NO.	**Adjustment type**	**Value used**	**Notes**	**Figure number**
1	Using Poisson disk sampling, with middle size.	Poisson true. Entity size 3 by 3. One room spawning 2 sets of entities.	Entities are spread away from each other and walls.	Figure 19
2	Using larger sized entities with Poisson.	Poisson true. Entity size 4 by 4. One room is spawning 2 sets of entities.	Entities are spread away from each other and walls but need a large room to be able to spawn all entities.	Figure 20
3	Using smaller sized entities with Poisson	Poisson true. Entity size 1 by 1. One room is spawning 2 sets of entities.	Entities are touching walls and each other.	Figure 21
4	Using medium sized entities with Jitter.	Jitter true. Entity size 3 by 3 One room is spawning 2 sets of entities	Entities are able to spawn and are grouped together.	Figure 22
5	Using large sized entities with Jitter	Jitter true. Entity size 5 by 5 One room is spawning 2 sets of entities	Some entities are able to spawn and are grouped.	Figure 23
6	Using smaller sized entities with Jitter.	Jitter true. Entity size 1 by 1 One room is spawning 2 sets of entities.	Entities grouped together and overlapping with the walls.	Figure 24
7	Only using the graph curve for difficulty values.	Seen in Figure 25.	Values are applied semi-correctly with a few nodes having lower than expected values.	Figure 26
8	Just using the interval values for difficulty values.	Low values are used at the star around 1, as the end gets closer values rise to a max of 20.	Just using random interval values leaves a spread of high and low values.	Figure 27
9	Using both difficulty values and interval values for difficulty values.	Nodes close to start have lower intervals of around 1, as the graph progresses to the end values rise to 5.	Using both adds a good amount of variety between neighbouring nodes.	Figure 28

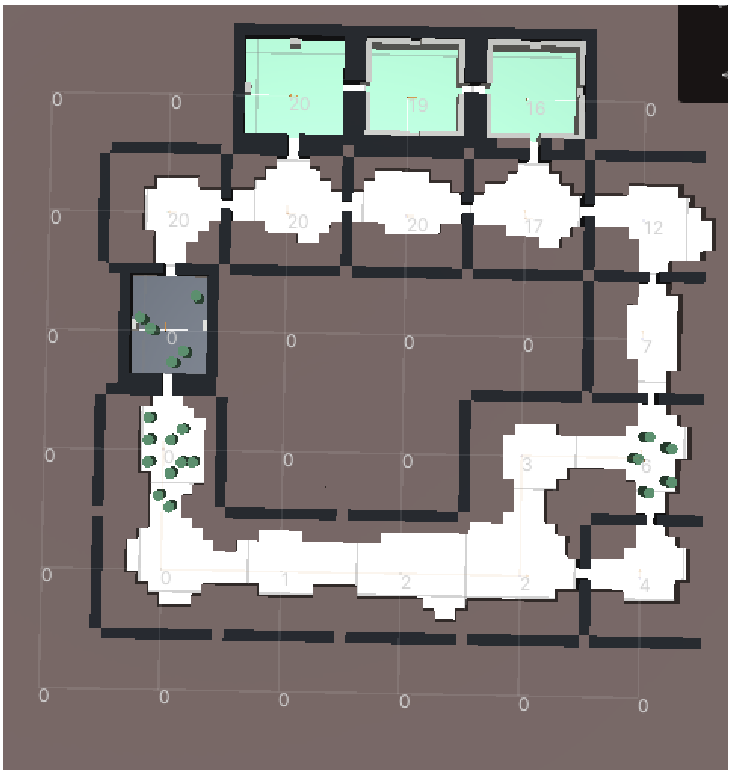


Figure 19: Entity size of 3 with 5 entities spawning.

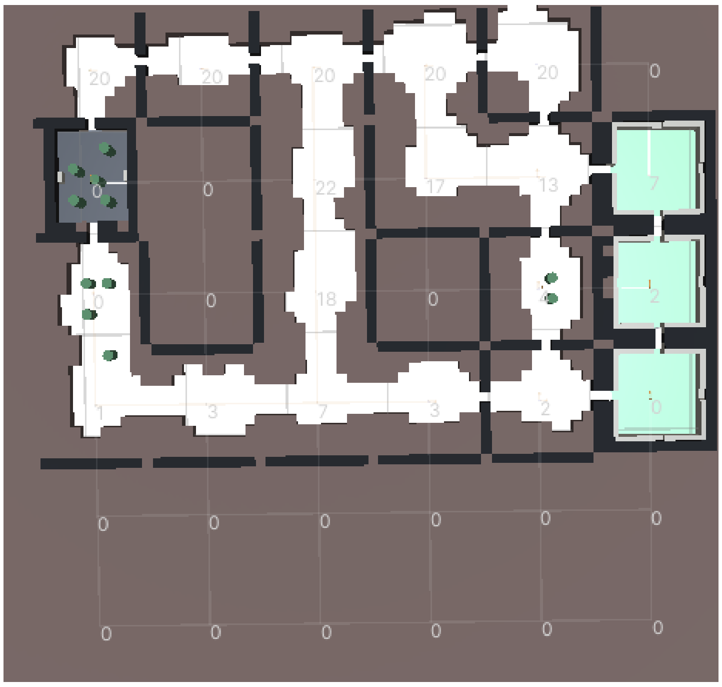


Figure 20: Increasing Poisson value to 4 with 5 entities spawning.

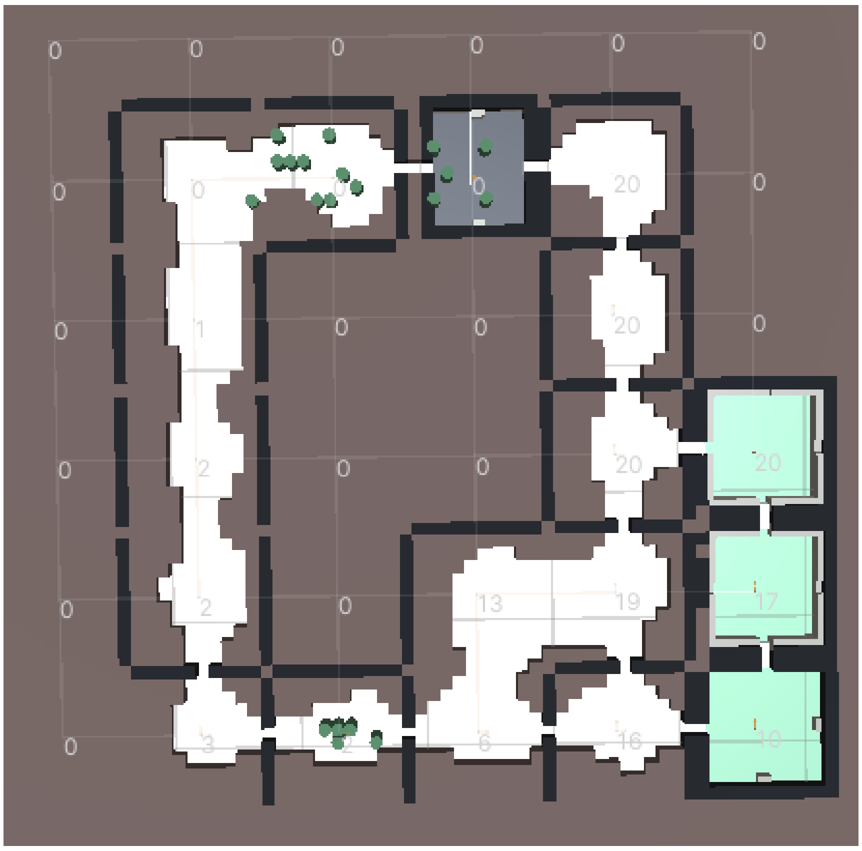


Figure 21: Small size of entities with 5 entities spawning.

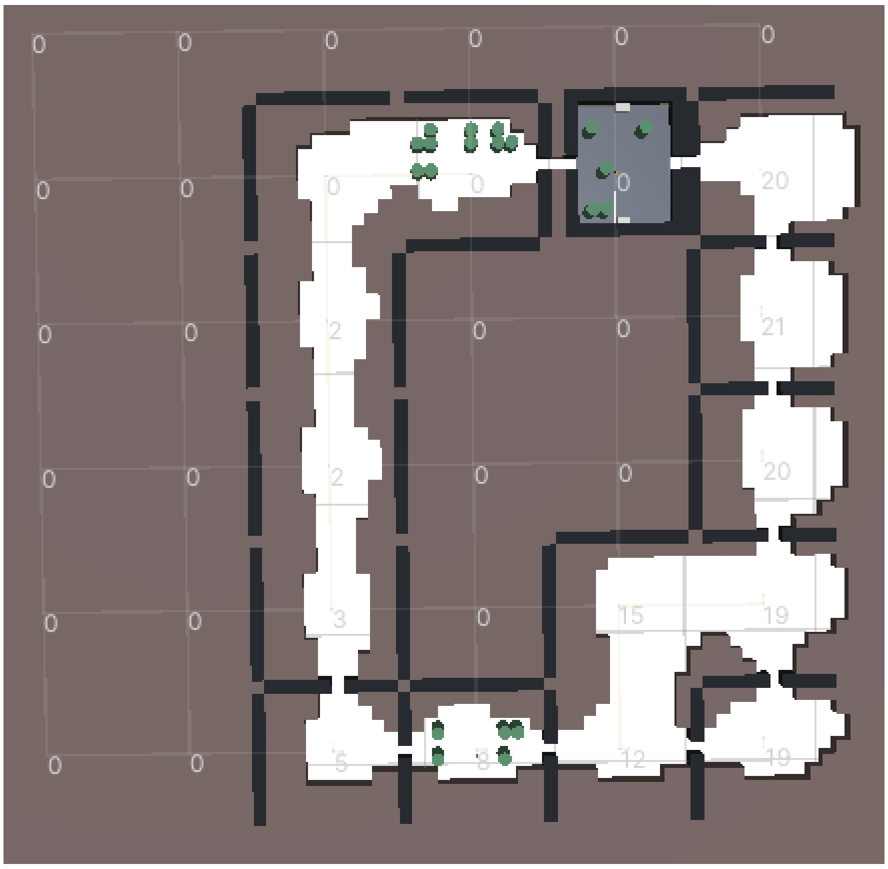


Figure 22: Medium sized entities spawned with Jitter.

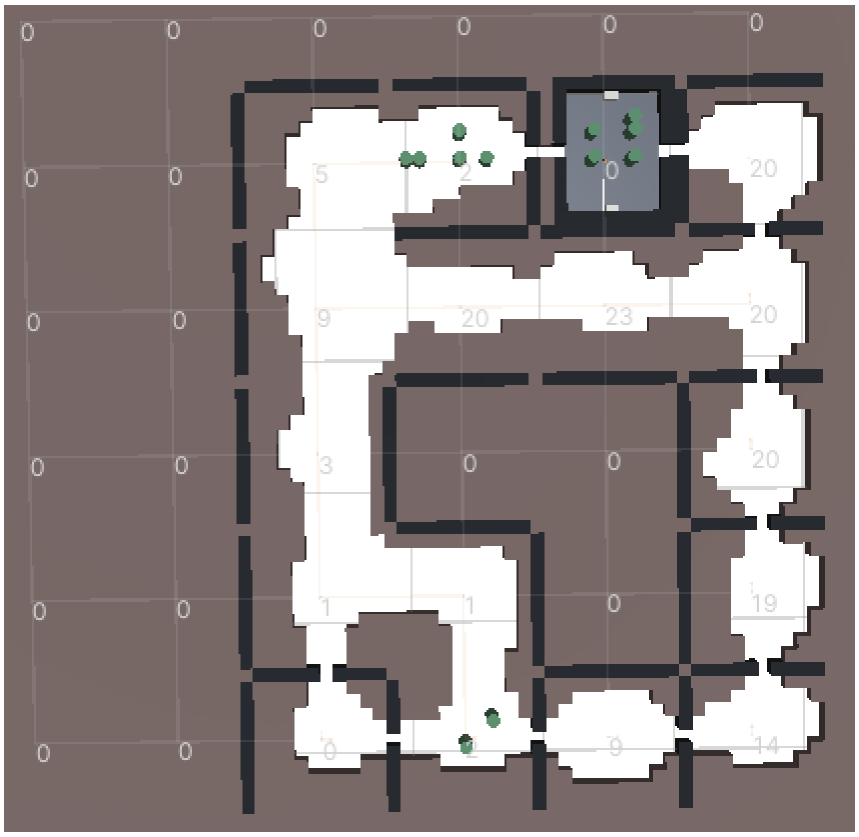


Figure 23: Large sized entities spawned with Jitter.

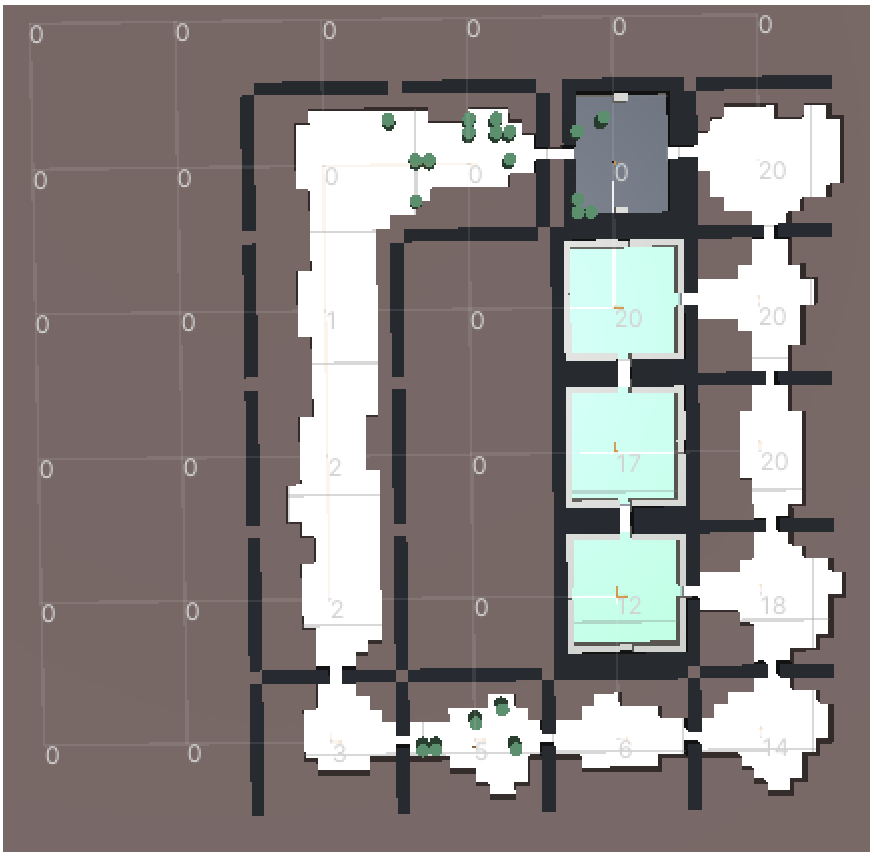


Figure 24: Small sized entities spawned with Jitter.

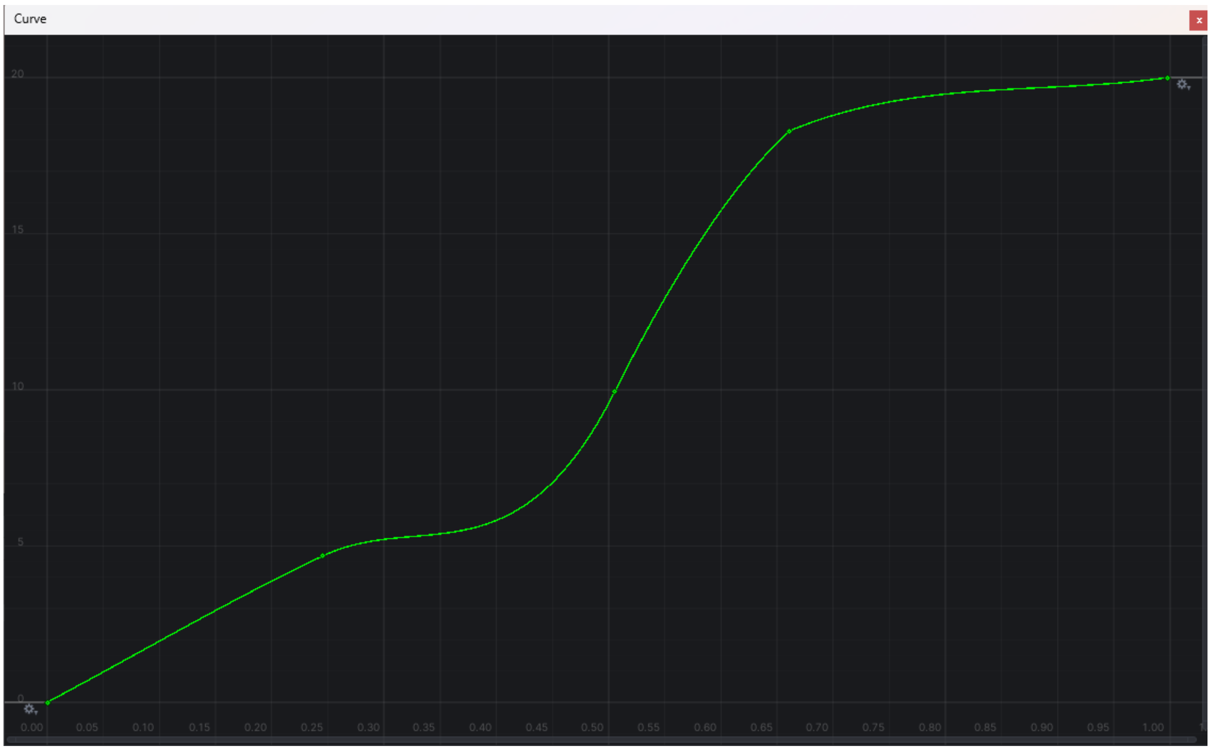


Figure 25: Difficulty Curve.

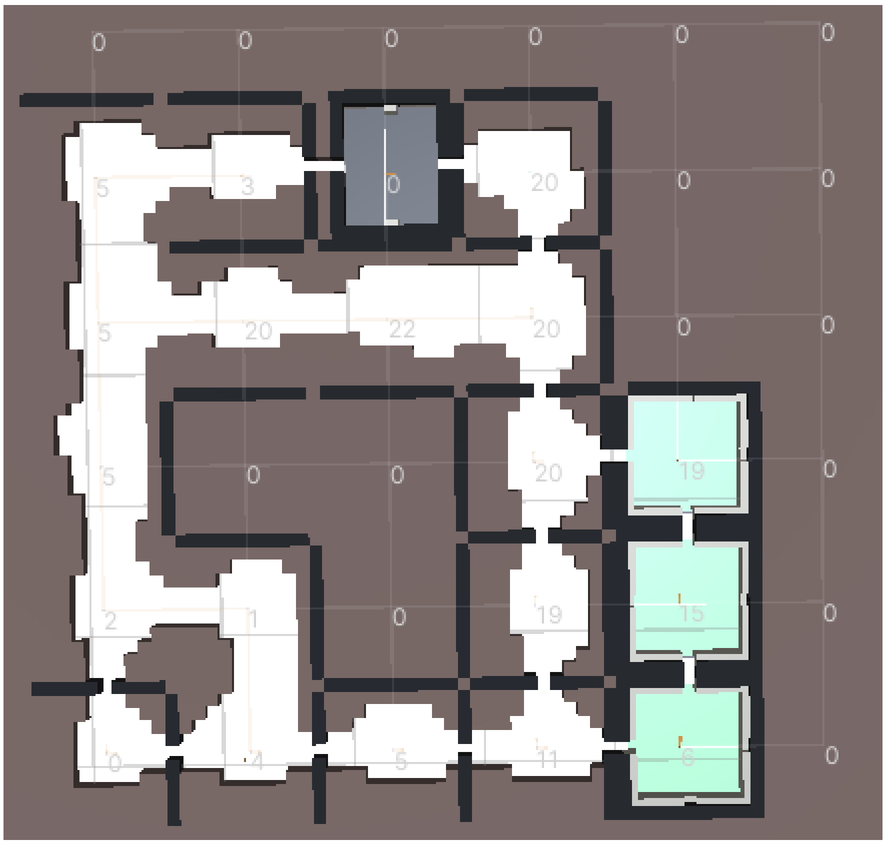


Figure 26: Only the graph curve is applied to difficulty values.

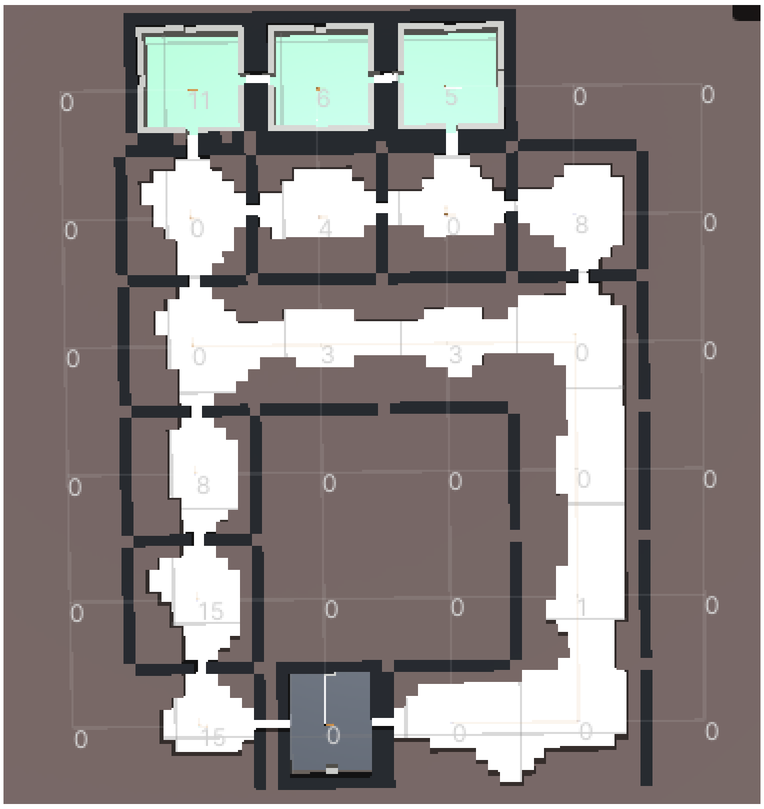


Figure 27: Just using random intervals for difficulty values.

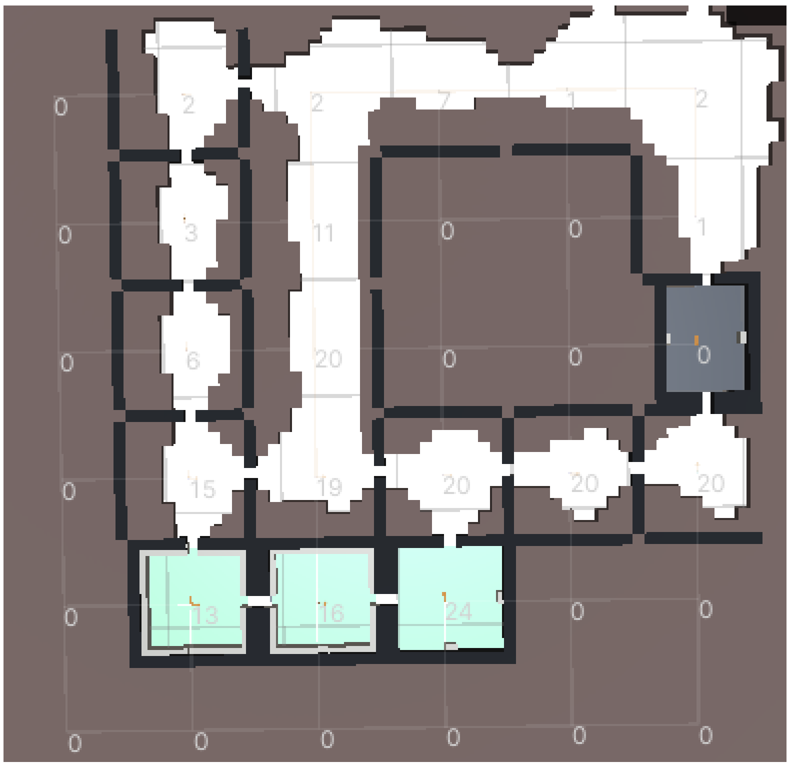


Figure 28: Using graph curve and interval values for difficulty values.

9. Results

Entity spawning was tested with two methods, during development and fine-tuning adjustment. The test plan was able to find oversights while implementing key components of the project, an example of this is test 2 in Table 1. This test showcased how entities were not being spawned correctly with cells that was already marked as invalid spawns still being checked. Without in-development testing to validate this, the fine-tuning tests would have been negatively impacted. Test 3 and 4, showcase the implementation of Poisson disk sampling and jittered grid work as expected with small logic errors with jittered grid. This was fixed by implementing the same method Poisson uses by checking its surroundings for walls, a downside of this is the method is slightly less efficient but still more efficient than Poisson disk method due to the additional checks the method needs. Within the fine-tuning adjustments for spawning entities, three different sized entities were used to test how well the methods are able to scale and what would happen at their limits. These tests outlined that the method's limits highly rely on the area of clear space. Poisson's ideal entity size for the tested environment was 3 by 3, depending on the user's expectations the size can be reduced to 1 by 1. Due to Poisson ensuring distance between different entities it is impacted more by smaller rooms than jittered grid. While jittered grid is able to fill smaller rooms with larger entities than Poisson, as seen in test 5 in Table 2, its main downside is the lack of consideration of other entities leading to grouping. While both methods are able to perform their purpose, each are limited by the size of the room that is being filled.

Offering a wide toolset of methods to control the difficulty in games is important, with each of the systems working well on their own and in cohesion. A key issue that was found during testing was the implementation of scaling the difficulty graph to match the length of the path found. This was fixed by making a fraction of each node's index out of the progression in path as explained in implementation. This fix was effective in scaling the difficulty progression and path length together. Test 7 in Table 2, showcases how only using the graph curve will cause some rooms to have lower than expected values, this is due to BFS reaching these rooms last. This is an issue with cyclic maps and pathfinding, as cyclic maps can offer infinite amounts of paths, which makes it hard to create logical paths from pathfinding. While interval values do not have issues with cyclic paths due it is focusing on a smaller scale, allowing for nodes values to be set beforehand. These values are not absolute, and it has an issue of values being low despite their distance through the dungeon as seen in test 8. Combining these two reduces this downside by adding variance throughout the dungeon. If a user wanted to add strict values to a room, then difficulty modifier value which is found in the right hand of rules would be a better option since they are not randomised. But these have a downside of the layout of the dungeon being procedural, which can cause a high difficulty rated rooms to be near the start if just modifiers are used. This is where test 9 showcases the benefit of using all the methods together as each method is able to cover for downsides.

Overall, these tests show that the tool is able to implement two viable methods for spawning entities, based on values provided by the user, within a procedural room and assign the environment a scaling difficulty values which can be used by developers to tune each room to their gameplay.

10. Critical Evaluation

This project was a follow up on a previous master's project (Bennett, 2024) which explored creating a physical environment from graph data through the use of cellular automata and pre-authored rooms. This artifact aimed to be able to apply methods for spawning entities within the empty environment and apply difficulty values to the rooms which can aide in the further development of the gameplay. The placement of entities is inspired by the roguelike genre and the dungeon design.

The aim of the project was to research into ways of visualising core assets into the environments. This was achieved by implementing two different methods: Poisson disk sampling and jittered grid. Both methods were implemented successfully, with the key difference between them being consideration of entities around itself leading to room size impacting Poisson more than jittered grid. But giving the user a choice allows for more variety when spawning entities within the room. Additional variety was added using scriptable objects, which allowed entity sets to be linked with stored nodes, these sets can hold a variety of entities with each having a different array of prefabbed entities to spawn to the allocated area. Difficulty curves are key within games to keep users engaged. This is true for roguelikes, as if the game does not challenge the user, no matter how enticing the environments are they will either become bored or frustrated as shown by Csikszentmihalyi's diagram (1990) in Figure 3. Thus, it is important to give user's a high level of control over the difficulty curve in games. This led to the implementation of multiple methods for the user to control the difficulty curve of the generated environment. The original idea was to implement a graph curve and pass the values from the graph to the node's data, based on the progression as inspired from the difficulty curves seen in Figure 6 from Larsen (2010), but issues with cyclic graphs and pathfinding algorithms led to the implementation of multiple methods such as difficulty modifiers and interval values being linked to specific nodes. This was successful at overcoming issues with pathfinding but also giving users more control over the outcome of the graph. These difficulty values can then later be used to scale enemies' levels, or increase the difficulty of puzzles, thus allowing for a more varied gameplay experience.

While the project did meet its original goal of visualising environments with core assets, it was made in consideration of the limitations of the tool. Fine-tuning testing allowed to find the limits of the toolset such as entity sizes being strongly related to room sizes or using all of difficulty toolset to properly adapt the node's difficulty values according to the progression to the path due to limitations that cyclic dungeon pose. While giving the user the freedom to adjust the tool however they like is key, it is important for the user to realise the limits of the tool and then use its limitations to their benefit. This further supports the use of test plans and fine-tuning adjustment tests, as these showed where the tool was the weakest and allowed for the analysis of if this weakness was from the implementation of method or a limitation of values provided.

The future of this project will be followed in the next project. This project will focus on creating a game from the features of the tool prototypes. This project will use the tool to create an environment with the tool, while trying to test the flexibility of the tool itself. The game will be within the roguelike genre and focus heavily on replayability.

Bibliography

Agiv, G., 2024. *Difficulty curves: how to get the right balance.* [Online]
Available at: https://www.gamedeveloper.com/design/difficulty-curves-how-to-get-the-right-balance-
[Accessed March 2024].

Baerentzen, J. A., Frisvad, J. & Martinez, J., 2023. *Curl Noise Jittering,* Sydney: SA Conference Papers.

Bennett, W., 2023. *An exploration of Cyclic procedural content generation,* s.l.: s.n.

Bennett, W., 2024. *Expanding generative iterations of the prototype.,* s.l.: s.n.

Biemer, C. F., 2023. *Dynamic Difficulty Adjustment via Procedural Level Generation Guided by aMarkov Decision Process for Platformers and Roguelikes,* s.l.: s.n.

Csikszentmihalyi, M., 1990. *Flow: The Psychology of Optimal Experience,* s.l.: Harper & Row.

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

GeeksForGeeks, 2024. *Difference between BFS and DFS.* [Online]
Available at: https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
[Accessed March 2024].

GeeksForGeeks, 2024. *Time and Space Complexity of DFS and BFS Algorithm.* [Online]
Available at: https://www.geeksforgeeks.org/time-and-space-complexity-of-dfs-and-bfs-algorithm/
[Accessed March 2024].

Guanghui, L., Lu, L., Chen, Z. & Yang, C., 2015. Poisson disk sampling through disk packing. *Computational Visual Media,* 1(1), pp. 17-26.

Icodewithben, 2023. *Mihaly Csikszentmihalyi's Flow theory — Game Design ideas.* [Online]
Available at: https://medium.com/@icodewithben/mihaly-csikszentmihalyis-flow-theory-game-design-ideas-9a06306b0fb8
[Accessed March 2024].

Kensler, A., 2013. *Correlated Multi-Jittered Sampling,* s.l.: s.n.

Larsen, J. M., 2010. *Difficulty Curves.* [Online]
Available at: https://www.gamedeveloper.com/design/difficulty-curves
[Accessed March 2024].

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

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].

Nguyen, W., 2022. *How to make superb survival games.* [Online]
Available at: https://www.gamedeveloper.com/game-platforms/how-to-make-superb-survival-games
[Accessed March 2024].

Red Blob Games, 2018. *Jittered grid vs Poisson disc.* [Online]
Available at: https://www.redblobgames.com/x/1830-jittered-grid/
[Accessed March 2024].

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

Tulleken, H., 2009. *Poisson Disk Sampling.* [Online]
Available at: http://devmag.org.za/2009/05/03/poisson-disk-sampling/
[Accessed March 2024].

Unity Technologies, 2019. *Editing Properties.* [Online]
Available at: https://docs.unity3d.com/2019.1/Documentation/Manual/EditingValueProperties.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].

Unity Technologies, 2024. *AnimationCurve.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/AnimationCurve.html
[Accessed March 2024].

Wang, T., 2021. Poisson-Disk Sampling: Theory and Applications. In: *Encyclopedia of Computer Graphics and Games.* s.l.:Cham Springer, pp. 2-8.

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 April 2023].

Yahya, N. M. H. H. et al., 2021. *Dungeon's Room Generation Using Cellular Automata and Poisson Disk Sampling in Roguelike Game,* Surabaya: IEEE.