Developing a General Procedural Generation Tool for Grid-based Dungeons

Olin Gao
Supervised by Penny Sweester as part of SCNC3101 (6 units)
The Australian National University

May 31, 2022

Abstract

The goal of this project is to develop a tool to procedurally generate a general family of 2D dungeon geometries, termed "grid-based dungeons". This tool gives designers a significant level of control over the properties of the output. The family of dungeons under our scope will be rigorously defined, and is largely inspired by the dungeon geometries observed in the 2D dungeon crawler *The Binding of Isaac: Rebirth*. In this report, we review popular procedural generation techniques such as cellular automata and random walks. Then we generalise the notion of a 2D grid-based dungeon and identify possible parameters which may govern observable properties in the geometry of such dungeons, such as the tendency for rooms to cluster, or the degree of branching. We adapt the aforementioned techniques to design an algorithm that specialises in generating such dungeons, and analyse how *The Binding of Isaac: Rebirth* fits under our model. The intent is for this algorithm to be applied by developers to assist in the development of a wide variety of 2D room-based dungeon crawler games.

1 Introduction

Procedural generation involves creating data algorithmically and nondeterministically. The output can be influenced by predefined parameters which govern random number generation and how randomly generated numbers are interpreted. In the context of video games, this data is often used to construct game geometries such as the terrain in *Terraria*[2] or the hexagonal tile grid in Sid Meier's *Civilisation* series[3].

In this project, we will focus on the procedural generation of 2D grid-based dungeon geometries. Specifically, these are dungeons which can be represented as a collection of rooms of varying sizes on a 2D grid; this is a generalisation of dungeons in *The Binding of Isaac: Rebirth* and are rigorously defined in section 4. To this end, we created an algorithm that is based on both an adaptation of known procedural generation techniques, and an intuitive reverse engineering of patterns observed in *The Binding of Isaac: Rebirth*. The associated program is intended to be used as a tool for designers, and will be produced as a standalone package for Unity. The design of the algorithm was independent from the actual dungeon generation algorithm used in *The Binding of Isaac: Rebirth*, which is publically available[4]. The range of geometries we are able to produce should somehow encapsulate the spirit of 2D dungeon crawlers, and as a sanity check, should be a superset of the set of possible dungeon geometries that can be found in *The Binding of Isaac: Rebirth*. We will survey various known procedural generation algorithms and adapt appropriate techniques for our needs.

The core of the algorithm involves multiple applications of a sophisticated adaptation of random walks on a two dimensional grid[7]. Essentially, we will hold objects in a queue which contain data about room shapes and the circumstances under which they are to be placed. Then in each step of the random walk, we will ask the both the current object and the previously placed object for a table of probabilities telling us if we should place the current object, backtrack or give up. We will see that, besides the shapes of the rooms, the most important parameter that dictates the kind of geometry we end up with is the relationship between the number of neighbours a room has and the probabilities involved in the random walk.

2 Statement of Authorship

This report and the associated program were created by me as part of a project supervised by Penny Kyburz at the Australian National University. One hundred percent of the code is my own. The project is inspired by procedural generation observed in *The Binding of Isaac: Rebirth*, a 2D dungeon crawler game developed by Nicalis and designed by Edmund McMillen.

3 A Survey of Procedural Generation Techniques

We will begin by reviewing some common techniques used in procedural dungeon generation. Many of the ideas we will review are adapted for use in our own algorithm.

R. van der Linden and R. Bidarra[6] have stated that in practice, dungeon generation consists of three elements:

1. A representational model: a definition of an object that holds data about the final dungeon structure.
2. A method for constructing that representational model.
3. A method for generating the geometry of a dungeon based on that model.

For our purposes, (1) essentially involves implementing the appropriate data structures to accommodate the definitions provided in section 3.1, and (3) is a way to visualise that data structure. Below we will briefly discuss common methods to procedurally construct representational models (2).

3.1 Cellular Automata

A **grid** $\mathcal{C}$ is a collection of cells $\{C_{ij} : i, j \in \mathbb{Z}\}$ with each C_{ij} having a set of predefined relations to other cells called **neighbours**. Each cell carries a **state**. A **cellular automata** is a sequence of grids $\mathcal{C}^0 = \{C^0_{ij} : i, j \in \mathbb{Z}\}, \mathcal{C}^1 = \{C^1_{ij} : i, j \in \mathbb{Z}\}, \mathcal{C}^2 = \{C^2_{ij} : i, j \in \mathbb{Z}\}$.... The state of each C^{t+1}_{ij} is determined by a (possibly non-deterministic) **generation function** that depends on C^t_{ij} and its neighbours.

Intuitively, if we start with a 2D grid containing a peculiar formation of rock cells (either chosen or randomly generated) on a background of air cells, a cellular automata will evolve that grid by first considering the neighbours of each rock to generate a new grid. This new grid might contain rocks, air, water and grass cells. The cellular automata will then apply the same generation rules based on the neighbours of each cell, with no memory of the initial peculiar formation of rocks, stopping at a predefined time when the grid of cells passes as a section of geometry in the game it is used in.

Perhaps the most well known example of a cellular automata is Conway's Game of Life[5]. In this example, each cell has one of two states: living and dead. Each cell is defined to have eight neighbours: the four orthogonally adjacent and the four diagonally adjacent.

The generation function of this cellular automata has the following rules: in the $\mathcal{C}^t$ step of the automata,

1. A living cell with two or three neighbouring living cells is living in the following step $\mathcal{C}^{t+1}$.

2. A living cell with any other number of neighbouring living cells is dead in $\mathcal{C}^{t+1}$.

3. A dead cell with exactly three neighbouring living cells is living in $\mathcal{C}^{t+1}$.

4. A dead cell with any other number of neighbouring living cells remains dead in $\mathcal{C}^{t+1}$.

Figure 5 is an example of a "periodic structure", so-called because the exact formation shown will repeat in the area shown after some number of steps.

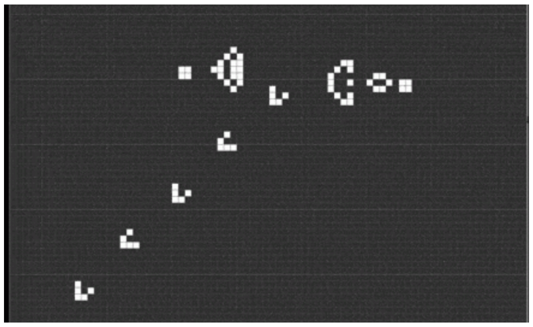


Figure 1: A periodic grid in Conway's Game of Life

In the context of video games, the 2D survival game *Terraria* partially utilises a cellular automata system to generate its terrain[8]. The primary weakness of cellular automata is its aforementioned lack of "memory". In particular, most generation functions will create rather uniform outputs, and it is difficult to generate predefined structures on a large-scale or have more fine-tuned control over the output of your algorithm. Due to this, *Terraria* requires other algorithms in tandem to correctly construct predefined structures, or to create distinct biomes.

3.2 Random Walks

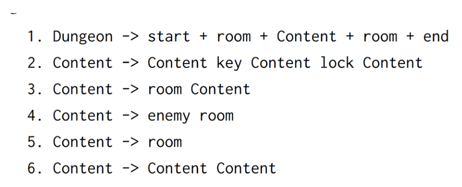
Traditionally, a walk on the 2D integer lattice is defined as a sequence $x_1, x_2, x_3, ...$ such that each x_i is orthogonally adjacent to x_{i+1}. A random walk is then a way to randomly generate such a sequence given a starting point x_1[7]. Typically neighbours are assigned the same probability to be chosen.

This concept can be generalised for application to procedural dungeon generation. To use our own dungeon model as an example, we can define two tiles as adjacent if their respective rooms are adjacent. This allows us to, at each step in the sequence, travel to a tile that is connected to the room we came from and place a new room there. We essentially create a linked chain of rooms this way, with each new room connected to the last in a random location.

3.3 Generative Grammars

A **context-free grammar** consists of a set V of non-terminal symbols and Σ of terminal symbols. A string is an ordered collection of symbols. Viewed as a sequence of strings, at each step a symbol $v \in V$ produces zero or more symbols in the subsequent step based on pre-defined rules. Each symbol may have multiple allowed rules. The result of a context-free grammar is a string that contains only terminal symbols.

A. Gellel and P. Sweetser[9] have developed a dungeon generation algorithm which makes heavy use of context-free grammars. In summary, a context-free grammar is applied to an initial symbol `Dungeon`, which eventually yields a string containing terminal symbols such as `room`, `end` or `lock`. These symbols are then placed into a queue and a random walk process on a 2D grid is run, with symbols in the queue acting as instructions for the process. For example, `room` may indicate to choose the next tile in the random walk, `end` may indicate to terminate the process, and `lock` may indicate to place a locked objective in the current room.



```
1. Dungeon -> start + room + Content + room + end
2. Content -> Content key Content lock Content
3. Content -> room Content
4. Content -> enemy room
5. Content -> room
6. Content -> Content Content
```

Figure 2: Examples of generation rules in Gellel's algorithm.

3.4 Adapting Techniques

Roughly speaking, our own dungeon generation algorithm will largely use the language and conventions of cellular automata, but the geometry itself will be generated using a sophisticated adaptation of the random walk as previously mentioned. This approach is similar to that used in *Terraria*[10]. We also take inspiration from Gellel's approach by storing a queue of all the objects we wish to place in order, though we do not use generative grammars to obtain this queue. These ideas will be made precise in the following sections.

4 The Grid-Based Dungeon

We will now introduce the specific type of object which we seek to procedurally generate in this project.

4.1 Definition

A **floor** or **grid-based dungeon** is essentially a grid that is partially filled with non-overlapping and connected rooms of varying sizes. More precisely, it is an integer lattice **tiles** $\mathcal{T} = \{T_{ij} : i, j \in \mathbb{Z}\}$ with the following properties:

1. Each tile is either **empty** or **traversable**.
2. $T_{a_1 a_2} \in \mathcal{T}$ and $T_{b_1 b_2} \in \mathcal{T}$ are **adjacent** if they are orthogonally adjacent, that is, if $|a_1 - b_1| = 1$ and $a_2 = b_2$, or if $a_1 = b_1$ and $|a_2 - b_2| = 1$.
3. If $T_{a_1 a_2}$ and $T_{b_1 b_2}$ are traversible, then there must exist a path of tiles $T_{a_1 a_2} = t_0, t_1, t_2, ..., t_n = T_{b_1 b_2}$ with t_i adjacent to t_{i+1} for all $0 \leq i \leq n-1$. In other words, the adjacency graph of traversible tiles is connected.
4. The set of traversible tiles is partitioned into connected subsets called **rooms** $\mathcal{R} \subset 2^{\mathcal{T}}$.
5. $|\mathcal{R}| > 0$.
6. Each room is assigned one or more predefined **tags** which hold unique **priorities**. `start` and `end` must be two such available tags. Exactly one room must be assigned `start` and exactly one room must be assigned `end`.
7. Rooms $R_1 \in \mathcal{R}$ and $R_2 \in \mathcal{R}$ are adjacent if a tile in R_1 is adjacent to a tile in R_2. To each pair of such tiles, we assign a **door** which connects R_1 and R_2.
8. Each door has one or more tags, and one of the tags must be the same as that of the room it connects to that has the highest priority.

A full **dungeon** is a non-empty connected directed graph of labelled floors. One of these floors is called the **starting floor** represented by a **starting vertex**. There also exist zero or more floors called **exit floors** represented by **exit vertices**. Each directed edge is assigned a set of **conditions** $\mathcal{C}$, and from each vertex that isn't an exit vertex there must exist at least one edge with $\mathcal{C} = \varnothing$. The topology of the graph is called the **dungeon plan**.

4.2 Gameplay Notes and Intuition

Adjacent rooms *must* contain a theoretical door between them. The non-existence of such a door can be represented as a door with a certain tag, however this is discouraged and against the spirit of this dungeon model. Typically, rooms tagged `end` contain a passage to the `start` room of a further floor. Games should ensure that the there is a path between a `start` and `end` room that consists of doors whose traversibility can never be permanently blocked, as this will cause a softlock situation.

4.3 The Grid-based Dungeon in *The Binding of Isaac: Rebirth*

Our definition for a dungeon is motivated by the dungeon geometry observed in *The Binding of Isaac: Rebirth*. In this game, each playthrough (or run) of a dungeon may either end in the player's death or the player defeating a final boss in the end room of an exit floor. The former is counted is a loss while the latter is counted as a victory.

Below is an in-game minimap showing the floor plan of a relatively larger floor.

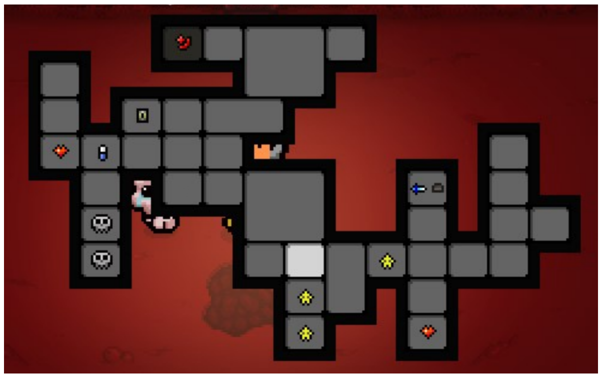


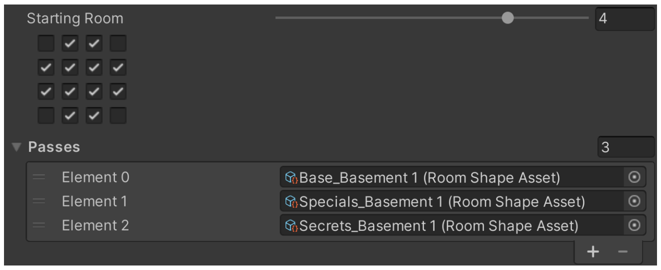
Figure 3: An in-game minimap of a floor

A more substantive case study of *The Binding of Isaac: Rebirth* is presented in section 6.

5 Algorithm Design

Our algorithm adapts the aforementioned techniques to generalise the dungeon generation algorithm found in *The Binding of Isaac: Rebirth*. The program was created as a Unity package and is intended to be used by designers to create their own family of dungeon geometries.

In this algorithm, a grid-based dungeon is generated in multiple **passes**. A pass is a set of parameters which dictate the placement of new rooms given an existing arrangement of rooms. We start with a pre-defined one-room dungeon which acts as the starting room, then these passes are applied consecutively.



```
Starting Room                                   4
  ☐ ✓ ✓ ☐
  ✓ ✓ ✓ ✓
  ✓ ✓ ✓ ✓
  ☐ ✓ ✓ ☐
Passes                                          3
  Element 0   Base_Basement 1 (Room Shape Asset)
  Element 1   Specials_Basement 1 (Room Shape Asset)
  Element 2   Secrets_Basement 1 (Room Shape Asset)
```

Figure 4: Our algorithm takes in a starting room and a list of passes as input.

5.1 The Room Queue

Each pass contains a set of room types along with a **size range**, and a default **neighbour ruleset** (defined in 6.3). Room types are selected from the set one by one to be placed into a **room queue**. The size range dictates how many *tiles* should be placed in this pass. Each room holds the following data:

1. **Shape** (`2D boolean array`): The shape of the room. The number of tiles which the shape occupies is called the **size** of the room. Shapes are equivalent up to rotation (that is, all rotations of the specified shape will be considered when the room is to be placed). Shapes which are not connected will be ignored.

2. **Connections** (`integer` ≥ 0): The number of traversible tiles that can be adjacent to this room.

3. **Minimum Count** (`integer` ≥ 0): The number of this room type which the room queue begins with.

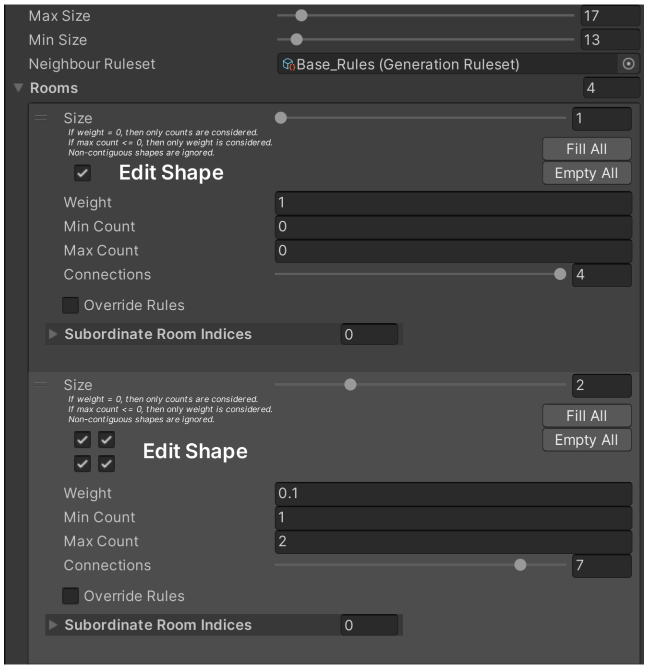
4. **Maximum Count** (`integer` ≥ 0): The maximum number of this room type which may be placed in the room queue. If this number is 0 or less than the minimum count, then this constraint is ignored.

5. **Weight** (`float` ≥ 0): For each choice, the probability to choose this room is equal to the weight of this room divided by the sum of weights of all shapes. If this number is 0, then a random number between minimum and maximum is chosen as the number of times this room type appears in the room count.

6. **Subordinate Rooms** (`List of other room types`): This room type contributes to the minimum and maximum count of all rooms in this list. This is useful for flagging certain room types as being of equivalent "rarity", or to denote certain room types as special cases of other room types.

7. **Override Neighbour Ruleset** (`Neighbour Ruleset`): This room will not abide by the default placement rules and instead will implement its own placement rules for this pass.

The room queue is populated so that the sum of the sizes of rooms in the queue is within the size range of the current pass. The room queue is shuffled so that the shapes which had non-zero minimum counts are not guaranteed to be the first placed. Rooms also contain a property called `tag` along with a tag priority; tags essentially store gameplay related data about the room.


Max Size 17
Min Size 13
Neighbour Ruleset Base_Rules (Generation Ruleset)
Rooms 4

Size 1
If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.
✓ Edit Shape
Fill All
Empty All
Weight 1
Min Count 0
Max Count 0
Connections 4
Override Rules
Subordinate Room Indices 0

Size 2
If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.
✓ ✓
✓ ✓ Edit Shape
Fill All
Empty All
Weight 0.1
Min Count 1
Max Count 2
Connections 7
Override Rules
Subordinate Room Indices 0

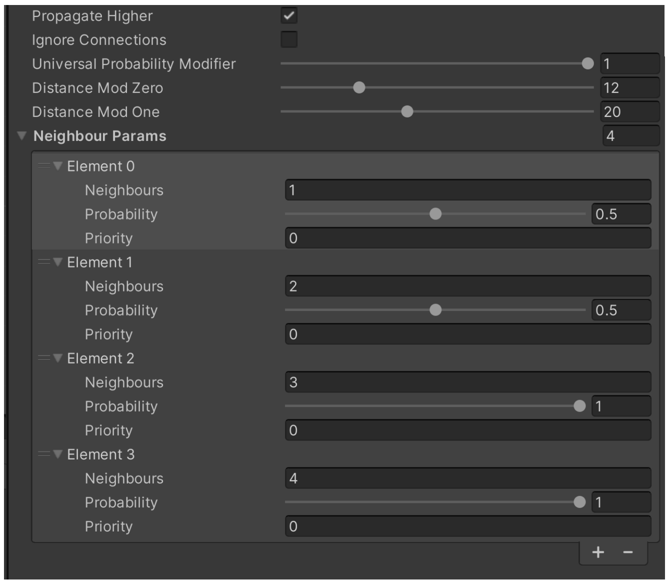
Figure 5: A pass contains size constraints along with a list of room types and parameters defining placement rules.

5.2 Placement and Neighbour Rulesets

Rooms in the queue are placed via a random walk subject to given placement rules (i.e. neighbour rulesets), in a similar fashion as described in subsection 5.3. In each step, a tile adjacent to the previously placed room is chosen and we **attempt** to place the next room in the room queue there. An attempt succeeds if a placement is geometrically possible, if all rooms would not end up with more adjacent tiles than connections, and if it passes probability checks defined by the neighbour ruleset of that room type.

A neighbour ruleset is a property of a pass that can be overridden by individual rooms. A neighbour ruleset contains the following elements:

1. **Neighbour Parameters**: Parameters governing success and order in attempts to place a room based on how many adjacent tiles that room would have if placed.
   (a) **Neighbour count** (`integer`): How many neighbours a prospective room placement would have.
   (b) **Probability** (`float` $0 \leq x \leq 1$): A placement attempt with the aforementioned prospective number of adjacent tiles will have this probability of not failing.
   (c) **Priority** (`integer`): Neighbour counts that are assigned higher priorities will all be searched through first before neighbour counts with lower priorities are even considered. Even if probability is not set to 1, placement of higher priority neighbour counts is guaranteed if such a placement is possible.
2. **Propagate Higher** (`boolean`): Neighbour counts not listed will inherit the highest neighbour count that is listed.
3. **Ignore Connections** (`boolean`): Connections of already-placed rooms are ignored when placing this room.
4. **Universal Probability Modifier** (`float` $0 \leq x \leq 1$): Multiply all probabilities by this number.
5. **Distance modifiers** d_0 **and** d_1 (`float` ≥ 0): With t being the distance between this room when placed and the starting room, the probability of placement is multiplied by $\frac{t-d_0}{d_1-d_0}$. That is, d_0 is the distance where the placement becomes impossible and d_1 is the distance where the probability of placement is unaffected by this step. If both values are set to 0, then this parameter is ignored.



Propagate Higher ✓
Ignore Connections
Universal Probability Modifier 1
Distance Mod Zero 12
Distance Mod One 20
Neighbour Params 4
Element 0
Neighbours 1
Probability 0.5
Priority 0
Element 1
Neighbours 2
Probability 0.5
Priority 0
Element 2
Neighbours 3
Probability 1
Priority 0
Element 3
Neighbours 4
Probability 1
Priority 0

Figure 6: This particular neighbour ruleset for single-tile rooms favours *clustering* and demands that placed rooms be relatively far from the starting room.

In summary, for each room in the room queue, we initiate a search starting from unoccupied tiles adjacent to the last placed room. Initially we are only considering the neighbour counts with the highest priority. A placement attempt will be rejected if:

- Placing the room here would be impossible without overlapping other rooms.
- Placing the room here would result in rooms having more neighbours than it has connections. (Unless ignore connections is true).
- We are not currently considering the neighbour count of this attempt.
- The probability check fails. Probability is defined on a neighbour count basis and can also be affected by distance from the starting room.

If a placement attempt is rejected, we move on to either a room adjacent to the room we are attaching to, or another tile adjacent to the room we are attaching to. And if no placements of the neighbour counts we are considering can be found, then we will consider neighbour counts with lower priorities and repeat this process. Of course, doors are generated between adjacent tiles belonging to different rooms whenever a room is placed. These doors inherit the tag of the neighbouring room with the higher priority tag.

6 Case Study: The Binding of Isaac

This section will analyse *The Binding of Isaac: Rebirth* in depth, with reference to the model we have developed for grid-based dungeons.

Below is the directed graph representing a simplified version of the dungeon plan found in the game. This is how floors are connected in the game, and each floor is a grid-based dungeon.

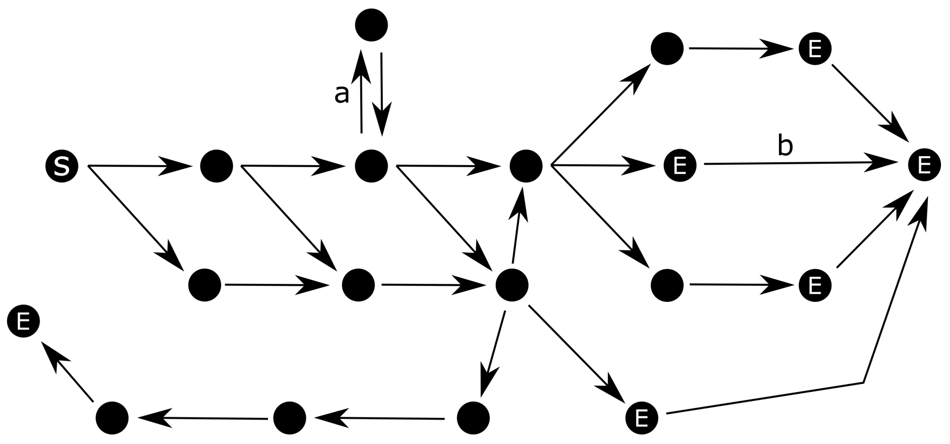


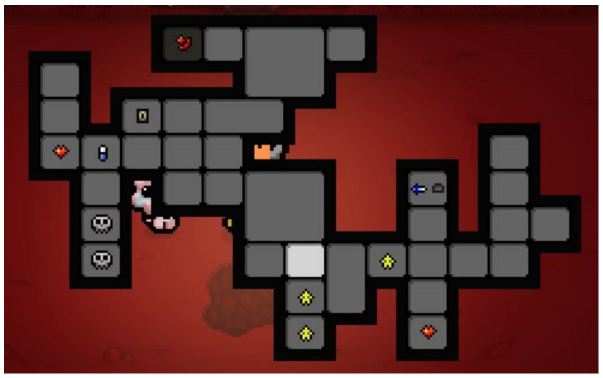
Figure 7: Simplified dungeon plan of the main floors in *The Binding of Isaac: Rebirth*

The vertex labelled S is the starting floor, and vertices labelled E are the exit floors where the player may exit the current run and claim victory. Access between some floors is subject to certain in-game conditions. For example, the edge labelled a can only be accessed if the end room of its source floor is reached within 20 minutes of the start of the current run, and the edge labelled b is made available based on random chance.

Floors typically vary in size from 10 to 60 traversible tiles (or just "tiles"), with floor size generally increasing with distance from the starting floor. Rooms themselves can only consist of up to 4 tiles. A typical room will consist of enemies which much be defeated before the doors of that room are unlocked, allowing the player to progress into adjacent rooms. The goal is to reach the end room and defeat a powerful boss, upon which passage to one or more other floors will be made available. Certain rooms may be designated as **special rooms**, which usually have doors requiring resources to unlock and contain items which contribute to the player's power progression.

Recall the in-game minimap of a typical larger floor from Figure 2:

Here, the highlighted room is the start room, and rooms marked with skulls contain are **boss rooms** with the bottom one being the end room. The spe-



An in-game minimap of a floor

cial room labelled with a red star and crescent is a curse room whose door requires health to traverse. The special room labelled with a blue sword is a challenge room whose door can only be traversed if the player has sufficient health. Other labels such as stars, hearts or pills convey information about resources in that room, and belong to non-special rooms.

On every floor there exist two **secret rooms** whose icons are not visible on the map, and whose doors are initially hidden during gameplay. These doors can be traversed only after they have been affected by an in-game explosion.

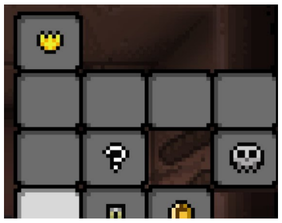


Figure 8: A secret room marked by ‘?’

6.1 Observed Generation Rules

We will discuss observations regarding the apparent properties of each floor in *The Binding of Isaac: Rebirth*.

The vast majority of rooms in each floor consist of single tiles, with a guaranteed but limited number of tiles that are 2, 3 or 4 tiles in size. Note that connected 2-sized rooms, 3-sized rooms and 4-sized rooms are all unique up to rotation.

The fundamental idea that differentiates the floor geometry in *The Binding of Isaac: Rebirth* from a completely random generation of a valid grid-based dungeon is how it treats the **clustering** of rooms and tiles. Intuitively, we can define clustering as the tendency for rooms to "clump up" with nearby tiles. More precisely, the reader may convince themselves that this notion can be captured as such: Consider a particular room R and its set of adjacent tiles $R_A = \{T_1, ..., T_n\}$. The set of tiles which are adjacent to two or more tiles in R_A is the **clustering set** R_C of R. The number of elements in this set is called the **clustering number** of R.

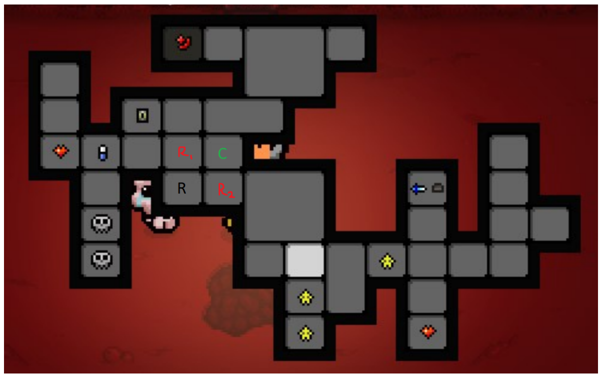


Figure 9: Indicating clustering on Figure 3

For example, in the above figure, the room labelled `R` has a clustering number of 1, with its clustering set containing just one room $R_C = \{C\}$ that is adjacent to both single tile rooms R_1 and R_2. In general, the maximum clustering number for a single tile room is 4.

In *The Binding of Isaac: Rebirth*, we observe that rooms tend to disfavour clustering, and when it does occur, clustering numbers are typically only as large as 1. In contrast, secret rooms are typically known to seek as much clustering as possible, for example as seen in Figure 8 with a clustering number of 2. Since floors are connected, this is equivalent to secret rooms seeking the location with the highest number of adjacent tiles after the rest of the floor geometry has been generated.

Special rooms and so-called "super secret rooms" may only be adjacent to one other tile as a strict rule, and the end room must be at least a certain distance away from the start room.

6.2 Replicating the Dungeon Geometry

The exact parameters I used to imitate *The Binding of Isaac: Rebirth*'s basic dungeon generation algorithm are given in the appendix.

In summary, *The Binding of Isaac: Rebirth*'s algorithm consists of three passes: one for the core geometry, one to place its special rooms, and one to place its secret rooms. The only room that makes use of distance modifiers is the end room.

In the first pass, we have all possible (connected) room shapes up to size 4. Single-tile rooms are the overwhelming majority of rooms in our queue, but larger rooms are guaranteed to be chosen at a certain rate, which involves setting minimum and maximum counts for them. Rooms generally tend to avoid clustering as discussed earlier. From observation, even clustering numbers of 1 are rather rare for single-tiled rooms. To this end, only a neighbour count of 1 is considered with any significant probability. There is also a moderate amount of branching, so overall probabilities are set at around 0.5.

Subsequent passes only contain single-tile rooms. In the second pass, we will only consider neighbour counts of 1. This entails disabling `propagate higher`, and assigning a non-one probability to the room shape so that special rooms are not guaranteed to be spawned close together.

In the third pass, we have two distinct types of rooms. For the secret room, we enable `ignore connections` and seek high neighbour counts with priority as opposed to probability. This means that, for example, if a placement with 4 neighbours exists, then it is guaranteed to be chosen. For the super secret room, we do the exact opposite - we only seek neighbour counts of 1.

7 The Program

We have implemented the aforementioned algorithm in C# as a Unity package. The algorithm, with the exception of the initialisation process, is extremely fast and it doesn't noticeably bottleneck the main thread under almost any circumstances. For visualising the output, we store a large grid of non-overlapping images, with each image corresponding to a tile. Then we create boundaries between the squares, corresponding to doors. Images are made visible or invisible depending on if a corresponding room or door is placed there.

7.1 Installation

To open the dungeon generation tool, it is recommended to use Unity version 2020.3.26f1. However, the tool should be compatible with later versions of Unity.

Once you have installed the Unity package, create a new 2D unity project. Then go to into Assets → Import Package → Custom Package...

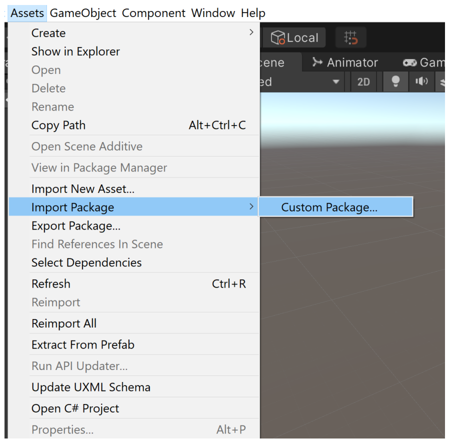


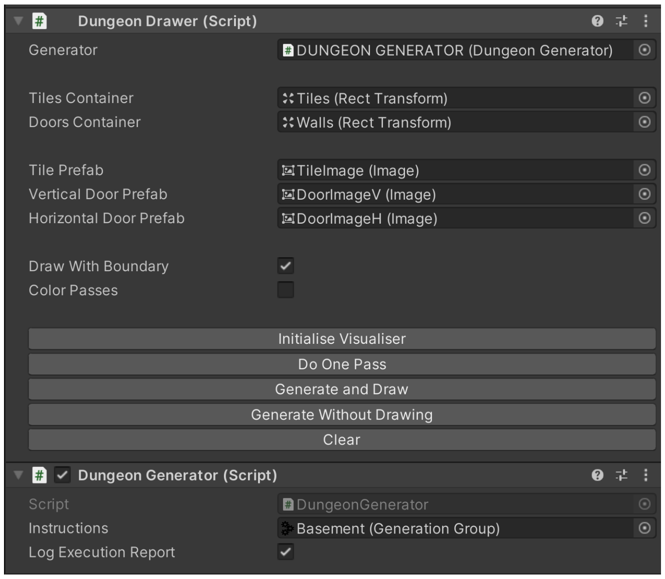
Figure 10: Assets → Import Package → Custom Package...

Select the installed package and import all files. Then go into the **Dungeon_Scenes** folder and open the **DungeonDemo** scene. You are now ready to use the dungeon generation tool.

7.2 How to Use the Tool

In short, to see the algorithm in action, open the **DungeonDemo** scene and select the **Dungeon Generator** game object. Check **Color Passes**, then click **Initialise Visualiser** → **Do One Pass** → **Do One Pass** → **Do One Pass**.

In more detail, when you have imported the package into Unity, you will see a scene called **DungeonDemo** in the Scenes folder. In this scene, the object named **Dungeon Generator** takes in a set of generation instructions and visualises the output.



Dungeon Drawer (Script)

Generator	DUNGEON GENERATOR (Dungeon Generator)
Tiles Container	Tiles (Rect Transform)
Doors Container	Walls (Rect Transform)
Tile Prefab	TileImage (Image)
Vertical Door Prefab	DoorImageV (Image)
Horizontal Door Prefab	DoorImageH (Image)
Draw With Boundary	✓
Color Passes	☐

Dungeon Generator (Script)

Script	DungeonGenerator
Instructions	Basement (Generation Group)
Log Execution Report	✓

Figure 11: The inspector window for the dungeon generator

We see that the generator comes loaded with a set of predefined instructions **Basement (Generation Group)** in the instructions field. Click **Initialise Visualiser** to associate the images in the visualisation tool with the data we will generate. This may take a couple of seconds. Then click **Generate and Draw**, which should be instant. You will see a visualisation of the output of the algorithm (if you can't find it, simply double click the **Canvas** object in the hierarchy). Click **Clear** to reset the dungeon data and visualisation (the tool does not need to be reinitialised).

To see each pass of the algorithm step by step, keep clicking **Do One Pass** and observe how each step of the dungeon geometry is created. You can enable the property **Color passes** to see which tiles each pass has placed.

7.2.1 Generation Parameters

The Rulesets/Isaac folder contains the parameters I used to replicate *The Binding of Isaac: Rebirth*'s dungeon generation.

A generation group is represented by the icon with three cogs; a pass is represented by the icon with one cog; a ruleset is represented by the paper icon.

To create these objects, simply right click in the space where these icons are located and hover over **Create**.

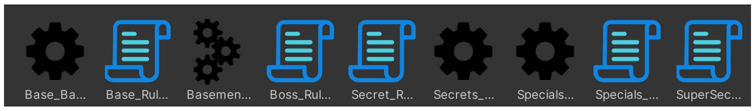
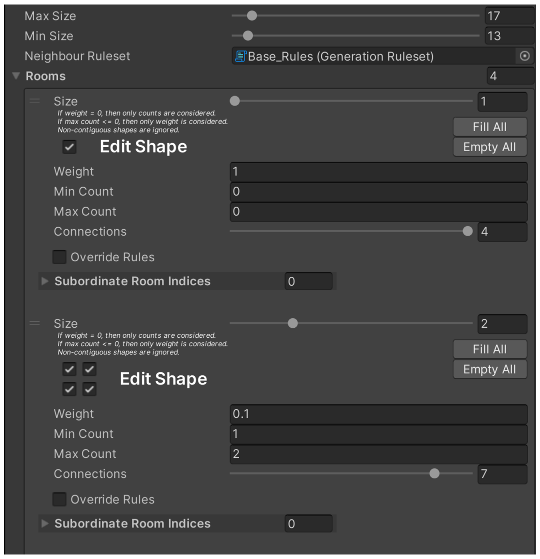


Figure 12: Groups (three cogs), passes (one cog) and rules (paper)

7.2.2 Configuring a Pass

If you click on the icon labelled **Base_Basement** you will find the first pass of the algorithm you had just visualised. This pass generates the core geometry of the dungeon and its output should have been coloured white if you had enabled **Color passes**.



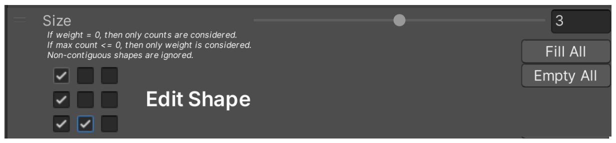
```
Max Size                 17
Min Size                 13
Neighbour Ruleset        Base_Rules (Generation Ruleset)
Rooms                    4
  Size                   1
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
  [✓] Edit Shape                               Fill All
                                               Empty All
  Weight                 1
  Min Count              0
  Max Count              0
  Connections            4
  [ ] Override Rules
  Subordinate Room Indices   0

  Size                   2
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
  [✓][✓]                                       Fill All
  [✓][✓]  Edit Shape                           Empty All
  Weight                 0.1
  Min Count              1
  Max Count              2
  Connections            7
  [ ] Override Rules
  Subordinate Room Indices   0
```

Figure 13: Pass configuration inspector

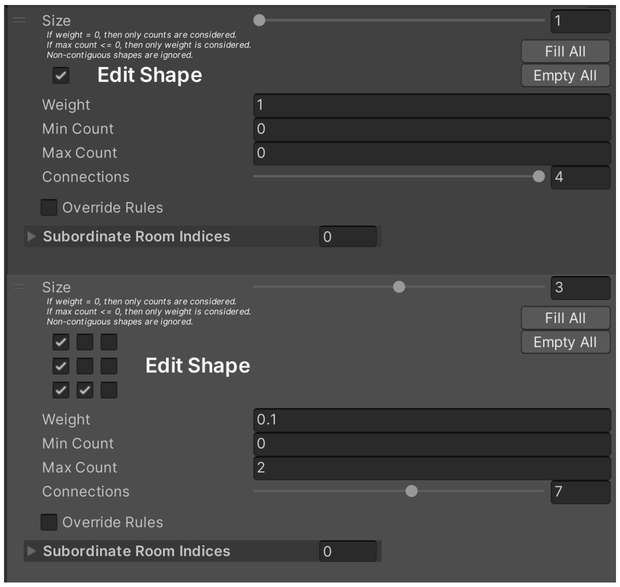
In this window, you can specify a list of room shapes along with the conditions for their placement into the room queue. These properties were explained in the previous section. To create a room shape, use the slider labelled **Size** to control the dimensions of the room, and the 2D array of boolean boxes defines the shape of the room. For example, to create an L-shaped room with four active tiles, we need a 3×3 grid. So set size to 3 and enable the boxes as such:

And in the below configuration, we have that this L-shape will be about ten times rarer than a standard single-tile room, and the L-shaped room will be placed at most two times in this pass:


Size 3

If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.

Figure 14: An L-shaped room definition


Size 1

If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.

Weight 1
Min Count 0
Max Count 0
Connections 4
Override Rules
Subordinate Room Indices 0

Size 3

If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.

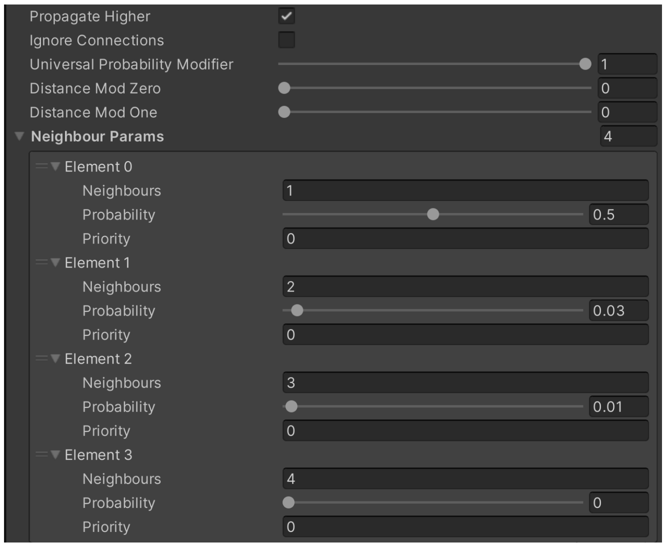
Weight 0.1
Min Count 0
Max Count 2
Connections 7
Override Rules
Subordinate Room Indices 0

Figure 15: Two room definitions in a pass

7.2.3 Configuring Rulesets

In the pass configuration, there were two places to put neighbour rulesets. The pass itself has a default ruleset in the field **Neighbour Ruleset**, and each room has an option to override this default ruleset.

If you click on the paper icon labelled **Base_Rules**, you will see the following window in the inspector tab:



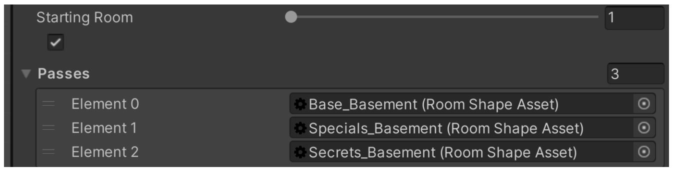
Propagate Higher	✓
Ignore Connections	
Universal Probability Modifier	1
Distance Mod Zero	0
Distance Mod One	0
Neighbour Params	4
Element 0	
Neighbours	1
Probability	0.5
Priority	0
Element 1	
Neighbours	2
Probability	0.03
Priority	0
Element 2	
Neighbours	3
Probability	0.01
Priority	0
Element 3	
Neighbours	4
Probability	0
Priority	0

Figure 16: The ruleset window

Here, we note that **Distance Mod Zero** and **Distance Mod One** correspond to d_0 and d_1 as defined previously.

7.2.4 Putting it Together

Now, if you click on the icon with three cogs labelled **Basement**, you will see the full definition for one iteration of the dungeon generation algorithm. This consists of a starting room and a list of ordered passes. In our tool, we call these a **generation group** or a set of **instructions**.



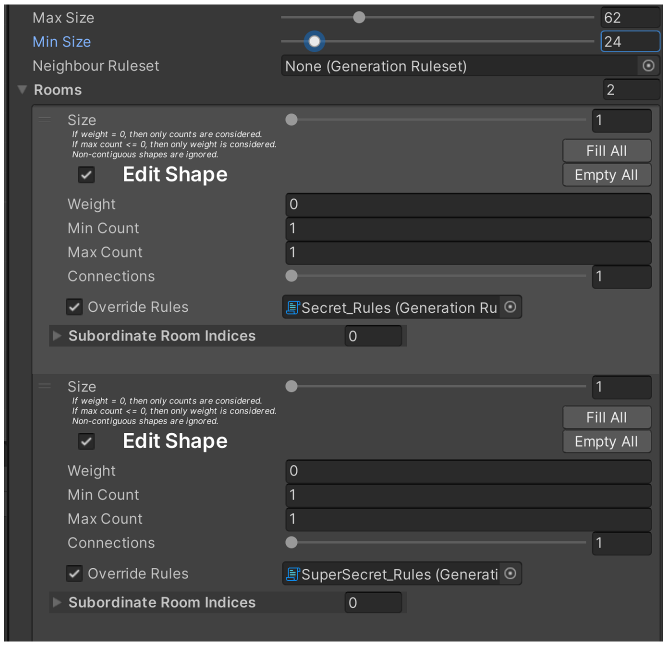
Starting Room	1
✓	
Passes	3
Element 0	Base_Basement (Room Shape Asset)
Element 1	Specials_Basement (Room Shape Asset)
Element 2	Secrets_Basement (Room Shape Asset)

Figure 17: The generation group window

To apply instructions to the visualisation tool, assign it to the **instructions** field on the dungeon generator game object. As we previously showed, the **Basement** instructions are assigned by default.

7.3 Caveats

In the pass window, make sure that the maximum count of the rooms is consistent with the minimum size of the pass. An error will be thrown to notify you if you have violated this consistency.



```
Max Size                62
Min Size                24
Neighbour Ruleset       None (Generation Ruleset)
Rooms                   2
  Size                  1
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
  ✓ Edit Shape          Fill All  Empty All
  Weight                0
  Min Count             1
  Max Count             1
  Connections           1
  ✓ Override Rules      Secret_Rules (Generation Ru
  Subordinate Room Indices  0

  Size                  1
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
  ✓ Edit Shape          Fill All  Empty All
  Weight                0
  Min Count             1
  Max Count             1
  Connections           1
  ✓ Override Rules      SuperSecret_Rules (Generati
  Subordinate Room Indices  0
```

Figure 18: An invalid configuration. Note that the minimum pass size of 24 is impossible to fulfill since the max count of the rooms suggests a maximum pass size of only 2.

Furthermore, distance modifiers are not allowed on the first pass. If you attempt this, the distance modifiers are ignored and a warning will be shown.

8 Discussion and Conclusion

In this project, we have reviewed a series of procedural generation methods and adapted ideas from random walks and generative grammars to create a procedural generation tool for a precise family of objects we call grid-based dungeons. These dungeons were intended to be an extension and generalisation of the data representing the dungeon geometry in *The Binding of Isaac: Rebirth*.

8.1 Designer's Intuition

The overall geometry of a grid-based dungeon is governed primarily by the shapes of rooms and their clustering tendencies. In dungeons with a clear goal, lower clustering is preferred, while in more open ended dungeons, higher clustering is preferred. To this end, low clustering is associated with high probabilities in low neighbour counts while high clustering is associated with high probabilities in high neighbour counts.

Due to the nature of the random walk, the universal probability modifier dictates the degree to which our dungeon "branches". If neighbour count 1 had probability 1, then a universal probability modifier of 1 will mean that we will complete a perfect random walk (provided that we don't encounter geometry issues), resulting in a perfectly linear non-branching dungeon. A lower universal probability modifier will conversely create more branches since there is a greater chance that the next step in our random walk on the current room we are considering will be arbitrarily rejected.

Different passes of the algorithm correspond to sets of fundamentally different room types. For example, to produce single-tile rooms (perhaps containing important objectives) which only connect on one side, it is most intuitive to create a pass on top of a pre-generated dungeon base specifically for such rooms. However, the design is flexible enough that one could theoretically do everything in one pass, though it would be a messy ordeal for the user.

8.2 Limitations

The entire project notably only has a single significant case study, namely *The Binding of Isaac: Rebirth*. This is of course a result of us basing our definition of a grid-based dungeon solely on observations in this particular game, as opposed to starting with a general model and justifying how many known examples exist.

The definition of a grid-based dungeon could have been topology-agnostic or dimension-agnostic. However, this is perhaps too wacky and complex for a one-semester project, and it is more practical to split these into cases (e.g. 2D hexagonal grids, 3D cubic grids). It would certainly be impractical (though admittedly cool) to make an algorithm that can accommodate 4D cubic grids or 3D triangular grids. This version of the algorithm is as general as possible without needing to delve into such fundamentally differing cases without being ridiculous.

8.3 Future Ambitions

The tool that we have developed is, although perfectly functional, not entirely polished. This was in the interest of completing this project within the span of one university semester. For a public release of this dungeon generation tool, I would certainly like for the visualisation component to be more informative. Most pertinently, the room tag feature is not yet implemented, since although the data is useful for application in a real game, it has no effect on the actual geometry of the output. For the visualisation tool, rooms should be associated with icons corresponding to their tags.

Some parameters such as `Universal Probability Modifier` may be mathematically descriptive, but on their own provide no intuition as to how they affect the dungeon geometry. For designers without technical backgrounds, I would like to create a *facade object* which holds intuitive-sounding properties such as `clustering` or `snaking` and sets the actual parameters of the algorithm based on these values.

8.4 Conclusion

We conclude that we have discovered an accurate model for generalising the dungeon geometry found in *The Binding of Isaac: Rebirth*. Under our model, we have also identified parameters responsible for phenoma such as the tendency for rooms to cluster and branch. Using these, we are not only able to imitate the kinds of dungeons we observe in *The Binding of Isaac: Rebirth*, but we have also been able to create a tool for creating two dimensional grid-based dungeons with reasonable generality.

References

[1] *The Binding of Isaac: Rebirth.* PC version, Nicalis, 2014.

[2] *Terraria.* PC version, Re-Logic, 2011.

[3] *Sid Meier's Civilisation Series*, Sid Meier, 1991.

[4] BorisTheBrave.Com. 2020. Dungeon Generation in Binding of Isaac. https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/ (Accessed 27 May 2022).

[5] Gardner, M., 'The fantastic combintions of John Conway's new solitaire game "life"', *Scientific American* (1970).

[6] Van der Linen R., Lopes R. and Rafael B., 'Procedural generation of dungeons', *Transactions on Computational Intelligence and AI in Games*, 6 (1), 78-89 (2014).

[7] Breno M. F. and Selan R. dos Santos, 'A Survey of Procedural Dungeon Generation', *Proceedings of SBGames* (2019).

[8] *Fundamentals of Programming and Computer Science* Lecture notes, p. 15-112, https://www.cs.cmu.edu/ 112/notes/student-tp-guides/Terrain.pdf, (accessed 27 May 2022).

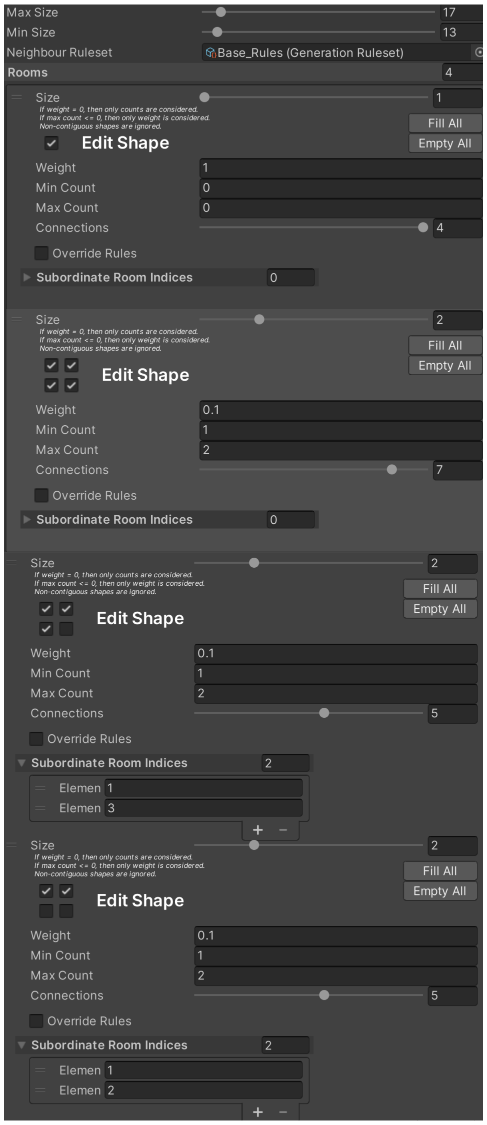
[9] Alexander Gellel, Penny Sweetser. 2020. A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels. In *International Conference on the Foundations of Digital Games (FDG '20), September 15–18, 2020, Bugibba, Malta.* ACM, New York, NY,USA, 18 pages.

[10] Jason Weimann, Yorai "Yoraiz0r" Omer. (2022, May 30). *Game Dev Show #64* [Video]. YouTube. https://youtu.be/Vi6UcHyO3RM?t=3385.

9 Appendix

9.1 Parameters to Imitate *The Binding of Isaac: Rebirth*'s Algorithm

Based on rough experimentation, *The Binding of Isaac: Rebirth*'s dungeon generation algorithm for *Basement* 1 can be analysed as follows:



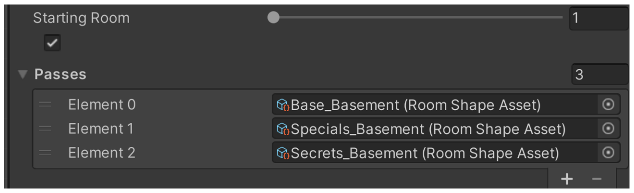
```
Max Size                 17
Min Size                 13
Neighbour Ruleset        Base_Rules (Generation Ruleset)
Rooms                    4

  Size                   1
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
                                             Fill All
    ✓         Edit Shape                     Empty All
  Weight                 1
  Min Count              0
  Max Count              0
  Connections            4
  ☐ Override Rules
  ▸ Subordinate Room Indices   0

  Size                   2
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
                                             Fill All
    ✓ ✓                                      Empty All
    ✓ ✓       Edit Shape
  Weight                 0.1
  Min Count              1
  Max Count              2
  Connections            7
  ☐ Override Rules
  ▸ Subordinate Room Indices   0

  Size                   2
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
                                             Fill All
    ✓ ✓                                      Empty All
    ✓ ☐       Edit Shape
  Weight                 0.1
  Min Count              1
  Max Count              2
  Connections            5
  ☐ Override Rules
  ▾ Subordinate Room Indices   2
      Elemen 1
      Elemen 3
                              + −

  Size                   2
    If weight = 0, then only counts are considered.
    If max count <= 0, then only weight is considered.
    Non-contiguous shapes are ignored.
                                             Fill All
    ✓ ✓                                      Empty All
    ☐ ☐       Edit Shape
  Weight                 0.1
  Min Count              1
  Max Count              2
  Connections            5
  ☐ Override Rules
  ▾ Subordinate Room Indices   2
      Elemen 1
      Elemen 2
                              + −
```

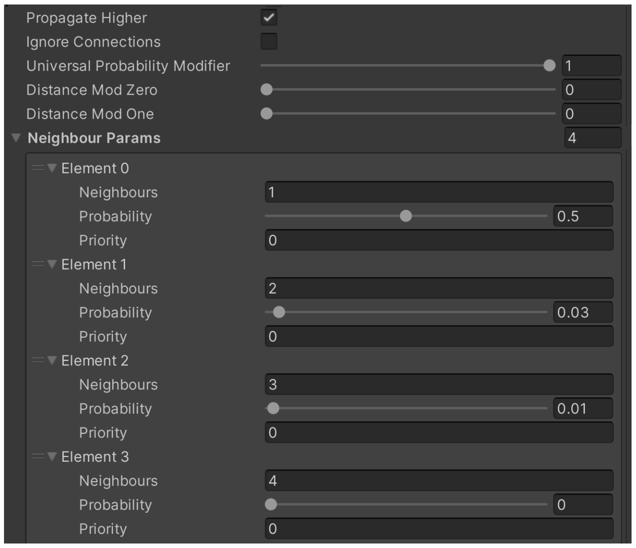


Starting Room 1

✓

Passes 3

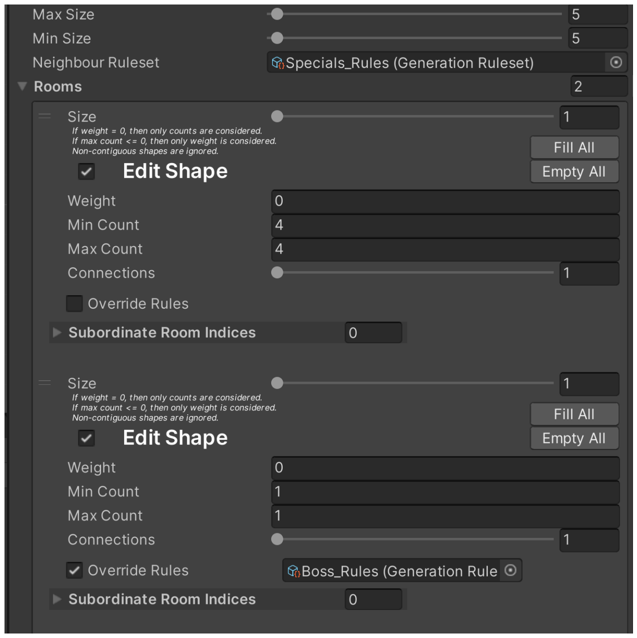
Element 0	Base_Basement (Room Shape Asset)
Element 1	Specials_Basement (Room Shape Asset)
Element 2	Secrets_Basement (Room Shape Asset)



Propagate Higher	✓
Ignore Connections	
Universal Probability Modifier	1
Distance Mod Zero	0
Distance Mod One	0

Neighbour Params 4

Element 0	
Neighbours	1
Probability	0.5
Priority	0
Element 1	
Neighbours	2
Probability	0.03
Priority	0
Element 2	
Neighbours	3
Probability	0.01
Priority	0
Element 3	
Neighbours	4
Probability	0
Priority	0



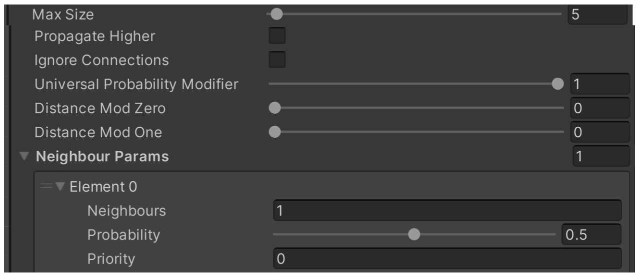
Max Size 5
Min Size 5
Neighbour Ruleset Specials_Rules (Generation Ruleset)
Rooms 2

Size 1
If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.
☑ **Edit Shape**

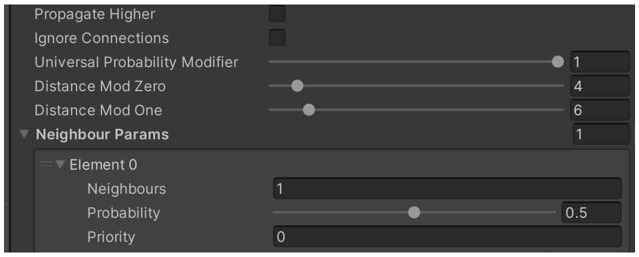
Weight 0
Min Count 4
Max Count 4
Connections 1
☐ Override Rules
Subordinate Room Indices 0

Size 1
If weight = 0, then only counts are considered.
If max count <= 0, then only weight is considered.
Non-contiguous shapes are ignored.
☑ **Edit Shape**

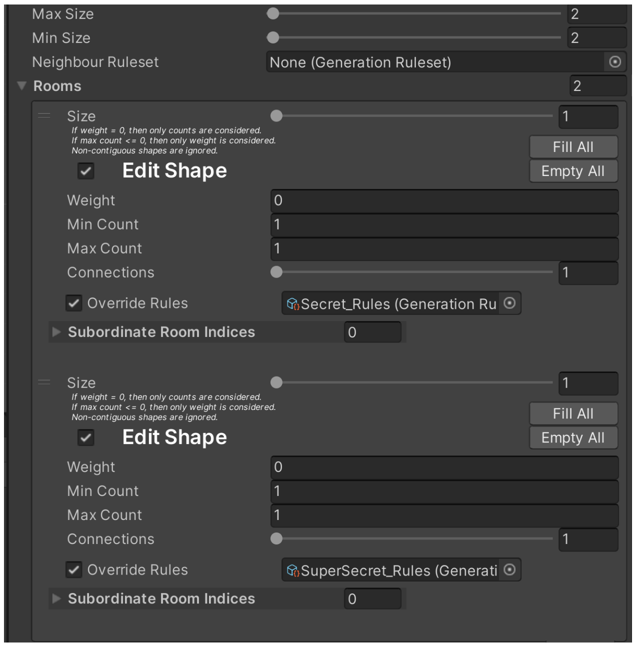
Weight 0
Min Count 1
Max Count 1
Connections 1
☑ Override Rules Boss_Rules (Generation Rule
Subordinate Room Indices 0



Max Size 5
Propagate Higher ☐
Ignore Connections ☐
Universal Probability Modifier 1
Distance Mod Zero 0
Distance Mod One 0
Neighbour Params 1
Element 0
Neighbours 1
Probability 0.5
Priority 0

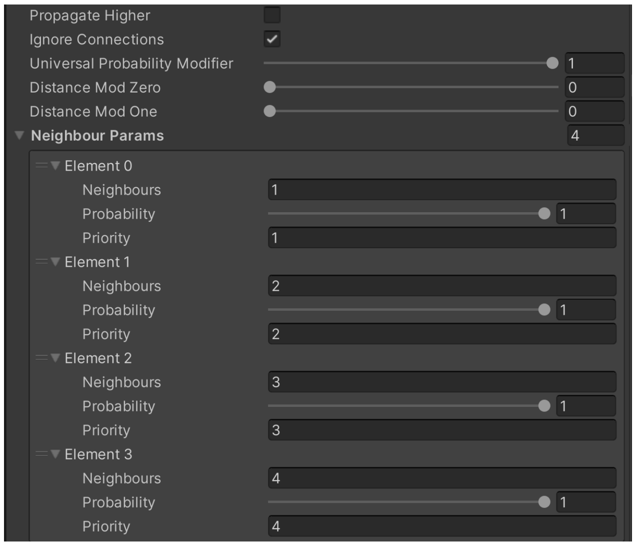


Propagate Higher ☐
Ignore Connections ☐
Universal Probability Modifier 1
Distance Mod Zero 4
Distance Mod One 6
Neighbour Params 1
Element 0
Neighbours 1
Probability 0.5
Priority 0

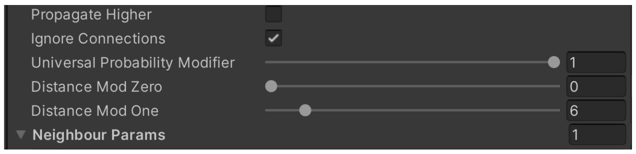


```
Max Size                        2
Min Size                        2
Neighbour Ruleset               None (Generation Ruleset)
Rooms                           2
    Size                        1
      If weight = 0, then only counts are considered.
      If max count <= 0, then only weight is considered.
      Non-contiguous shapes are ignored.
                                                  Fill All
      ✓  Edit Shape                               Empty All
    Weight                      0
    Min Count                   1
    Max Count                   1
    Connections                 1
    ✓ Override Rules            Secret_Rules (Generation Ru
    Subordinate Room Indices    0

    Size                        1
      If weight = 0, then only counts are considered.
      If max count <= 0, then only weight is considered.
      Non-contiguous shapes are ignored.
                                                  Fill All
      ✓  Edit Shape                               Empty All
    Weight                      0
    Min Count                   1
    Max Count                   1
    Connections                 1
    ✓ Override Rules            SuperSecret_Rules (Generati
    Subordinate Room Indices    0
```



```
Propagate Higher                  
Ignore Connections                ✓
Universal Probability Modifier    1
Distance Mod Zero                 0
Distance Mod One                  0
Neighbour Params                  4
  Element 0
      Neighbours                  1
      Probability                 1
      Priority                    1
  Element 1
      Neighbours                  2
      Probability                 1
      Priority                    2
  Element 2
      Neighbours                  3
      Probability                 1
      Priority                    3
  Element 3
      Neighbours                  4
      Probability                 1
      Priority                    4
```



Propagate Higher ☐
Ignore Connections ✓
Universal Probability Modifier 1
Distance Mod Zero 0
Distance Mod One 6
Neighbour Params 1