Interactions Between Game Design and Procedural Level Generation

Yuheng Cai
Game Science and Design
Northeastern University
cai.yuh@husky.neu.edu

ABSTRACT

“Endless tower” is an interesting single player strategy RPG that implements a procedural map generation system. In this game, players use their skills to beat enemies and explore the map, and after they are killed by enemies, they are able to create a new world and have different experiences. This paper illustrates how the author developed the level generation system while designing the game, and also provided new ideas for procedural level generation.

Table of Contents

ABSTRACT 2
1 INTRODUCTION 4
2 BACKGROUND 4
2.1 Binary Space Partitioning 4
2.2 Agent-Based Growing 5
2.4 Cellular Automata 6
2.5 Separation 7
2.6 Grammar-based 8
2.7 Evaluation 8
3 APPROACH/METHODOLOGY 10
3.1 Generation work flow analysis 10
3.2 Generated result analysis 10
3.3 Idea forming 11
3.4 Game Design 11
3.4.1 Game Introduction 11
3.4.2 Game rules 12
3.5 Developing walk through 13
3.5.2 Pre-Populate 13
3.5.3 Abstraction 14
3.5.4 Construction 16
3.5.5 Post-Populate 19
3.5.6 Final Image 28
3.5.7 Map Converting 29
4 Results 29
4.1 Overall looking 29
4.2 Evaluation 30
5 CONCLUSION 31
6 REFERENCES 31
Appendix 1: System architecture 33
Appendix 2: Game loop 35
Appendix 3: Design of Merchant 36
Appendix 4: Enemy setting 38

1 INTRODUCTION

There are many ways of implementing 2d procedural map generation systems these days. Well-described approaches in the literature include grammar-based systems and cellular automata, but also more recent methods that use machine learning to generate a level based on previously seen map data. All of these methods can create a playable world, but they all have their own problems and none of them are perfect. Because level generation is tied to the specific game it is used for, it's hard to evaluate a level generation process by itself only – it is necessary to also take into account the game's design. In order to investigate this relationship, I first analyze six popular dungeon generation methods and form my own map generation idea. In order to bring my idea alive, I designed a new game on top of this map generation idea and keeps grinding it according to my game design and tests. Through this process, I investigate how my level generation algorithm influences my game design, and keep iterating it to see how my design choices can influence my level generation methods, in order to understand the interplay between the two. Thus, my thesis represents an inquiry into designing with and for procedural level generation.

2 BACKGROUND

There are many types of level generation methods that are mature enough to produce a game, but they vary, and e.g. methods for terrain and the dungeon generation are quite different. In this section, I will focus on typical dungeon generation methods in games, and shows what I want to accomplish by analyzing those methods that I am designing as part of this thesis. Dungeon generation is typically considered the problem of creating and linking rooms as playable space, typically rooms and hallways [1], sometimes only rooms, and in those spaces, there are keys, doors, items, and enemies. Six influential methods in the literature on dungeon generation are Binary Space Partitioning (BSP), Agent-Based Growing, Connection, Cellular automata (CA), Separation, and Grammar-based. Below, I briefly outline each method.

2.1 Binary Space Partitioning

Binary Space Partitioning was developed by Schumacker et al.[2] in 1969. It was built for 3D object rendering but later, in 1993, Doom used this concept for dungeon generation. notable games that have used variations of BSP include Quake and Path of Exile. The method works by recursively dividing a surface into two for a number of iterations, and then connecting them into a whole. The algorithm works in the following manner:

1. Choose a direction, vertical or horizontal.
2. Choose a random start position.
3. Split the dungeon into two sub-dungeons A and B.
4. Go to step 1 to divide sub-dungeons until they are in desired size.
5. For all tree's leafs, connect them with their sister from the center of the room.
6. Repeat the process for the parent sub-regions, until room A and B are connected.

The final result looks like the example shown in Figure 1:



Figure 1. Result from BSP map generation

As Shaker and Liapis [3] and Pedersen [4] categorized, it is a relatively easy approach to create groups of rooms with no overlapping in this way .The main drawbacks of BSP are that the room generation provided limited control over the generated rooms and the generated dungeon seems too organized. As a consequence, the first created level could be boring because it doesn't contain any play experience design [5]. A way of addressing this is to modify the dungeon later in the process. An example is Path Of Exile (Grinding Gear Games, 2013), where the developers used a weighted based room selection method to create a meaningful path, thus improved the player experience and making the maps more lifelike.

2.2 Agent-Based Growing

Agent-Based Growing is based on Random Walk algorithm, which was first introduced by Karl Pearson in 1905 [6]. Many games use this approach as their level generation method, such as Slappy and Nuclear Throne. The concept of this method is using a single agent to dig tunnels and placing rooms for levels. The algorithm works in the following manner:

1. Place an agent at a dungeon tile.
2. Place a room in the current location.
3. Randomly choose a direction for it to move.
4. Move the agent by the chosen direction.
5. Go back to step 2 until you get enough rooms.

Results:

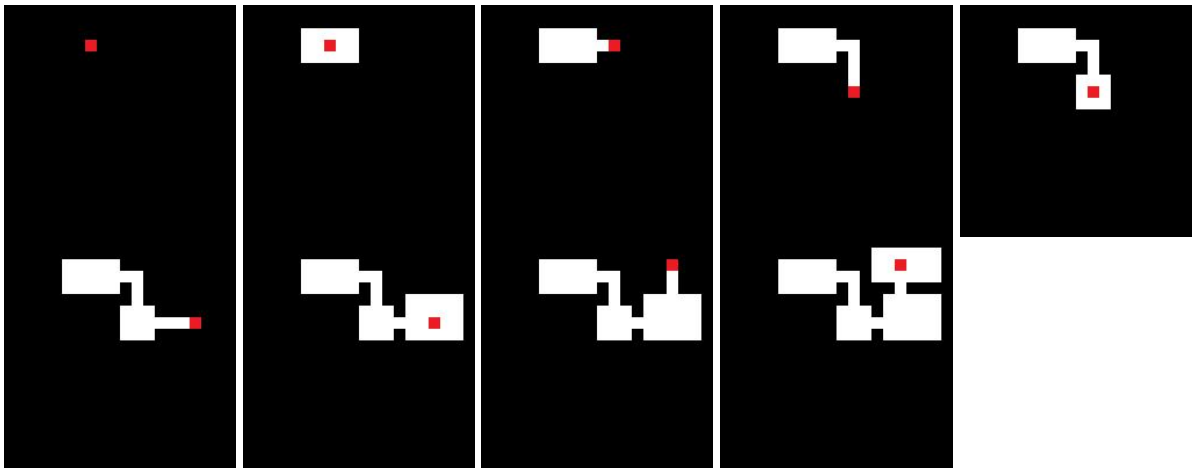


Figure 2. Result from Agent based map generation

The advantage of this algorithm is it is more likely to create organic levels, and easier to apply linear play through design because the map is created by a walking agent. A major

drawback is that the final shape largely depends on the behavior of the agent, thus the map could end up with a in chaotic structure, which could make it unplayable or not entertaining.

2.3 Connection

Connection algorithm, which is the standard dungeon generation method first used in Rogue and later in famous games like Brogue, Spelunky and Dead Cells, also uses this concept as the base of their map generation method. This concept requires preset and premade rooms, and also specific doors for these rooms. The main idea is it connect preset rooms along their doors in order to make a whole. This method can also be applied in a hierarchical manner. A simple layer connection method is directly connecting those rooms together like the games Brogue, Spelunky [7] and Dead Cells did, start with a room, choose a direction, add another room to it, repeat until the level is full [8]. And hierarchical usage of it can generate small rooms in a large region created by the previous generation. The algorithm works in the following manner:

1. Start with current room.
2. Select a door of current room.
3. Select a random room which its door can be connected to the current room.
4. If there is no overlapping between rooms, go ahead, otherwise return to step 3.
5. Place new room according to door positions, and connecting them.
6. Assign the new room as the current room, go back to step 1.

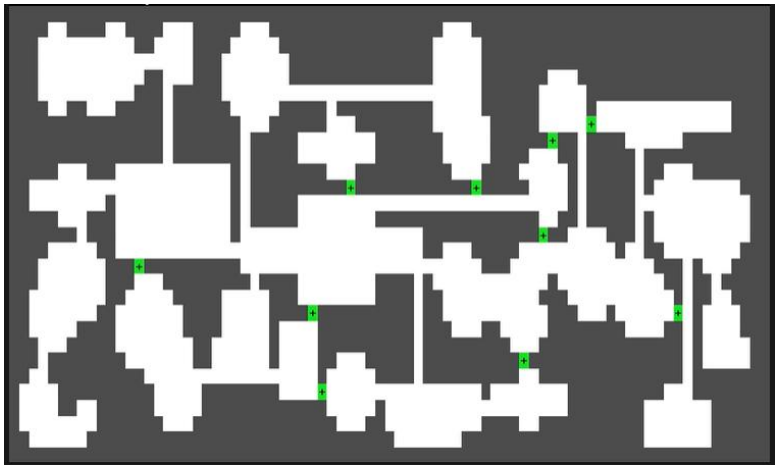


Figure 3. Result from Connection map generation

This method is popular not only because it's simple, but also because it can easily adapt a play through experience because rooms are generated in sequence. It does, however, have the disadvantage that there might be lots of backtracking while playing. This was addressed by the developer of Brogue (2018), who proposed that one possible solution is to connect adjacent rooms, and another solution is to create portals that can transfer players around the map, a technique used in the game Dead Cells (2018).

2.4 Cellular Automata

Cellular Automata was developed by Ulam and von Neumann in 1950s, It was used for calculating liquid motion, but it has the potential to be used as a map generation method too. Later games achieve this by changing the rules and parameters of the algorithm. Modern games like Terraia (2011) and Oxygen Not Included (2017) adapted this method for cave generation and liquid behavior simulation. The concept of it was to consider a liquid as a group of discrete units and calculate the motion of each based on its neighbor's behaviors [9].In the domain of procedural map generation, it is usually used to create organic shapes, most common ones are caves and terrains.

The algorithm works in the following manner:
1. If a living cell has less than two living neighbors, it dies.
2. If a living cell has two or three living neighbors, it stays alive.
3. If a living cell has more than three living neighbors, it dies.
4. If a dead cell has exactly three living neighbors, it becomes alive.

An example result is shown in Figure 4:

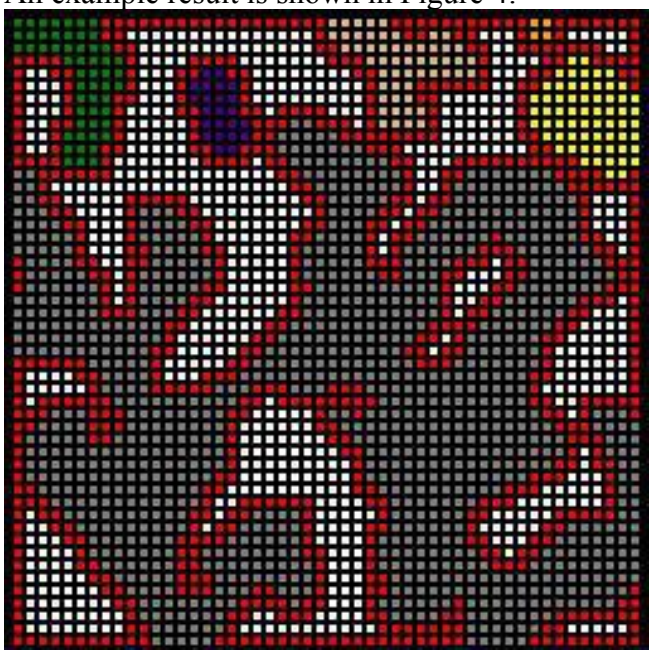


Figure 4. Result from CA map generation.

The outcome is diverse and looks natural, you can also play with the values to perform very different outcomes. This makes the algorithm very versatile, but the disadvantages are there might be isolated cave sections that aren't connected to other sections. These could block players from completing the game. To solve this problem, we have to tune the method or ensure the connection of the rooms as a post processing step. Another disadvantage compared to other generation methods is that it is hard to abstract the method to generate a tree or a linear structure.

2.5 Separation

This method was first introduced by TinyKeepDev (2013), they use this technique as their map generation method in TinyKeep. The concept of is to first spawn all rooms in the center of the map, and then separate them all to form the layout. After that, the shapes are connected to form a tree structure by tunnels, after which the whole map is formed.

The algorithm works in the following way:
1. Distribute all selected rooms randomly within a shape.
2. Separate them to prevent overlapping.
3. Link them together using Delaunay Triangulation.
4. Prune using Minimum Spanning Tree.
5. Add some rooms back.
6. Make tunnels where connections are missing.

An example result is displayed in Figure 5:

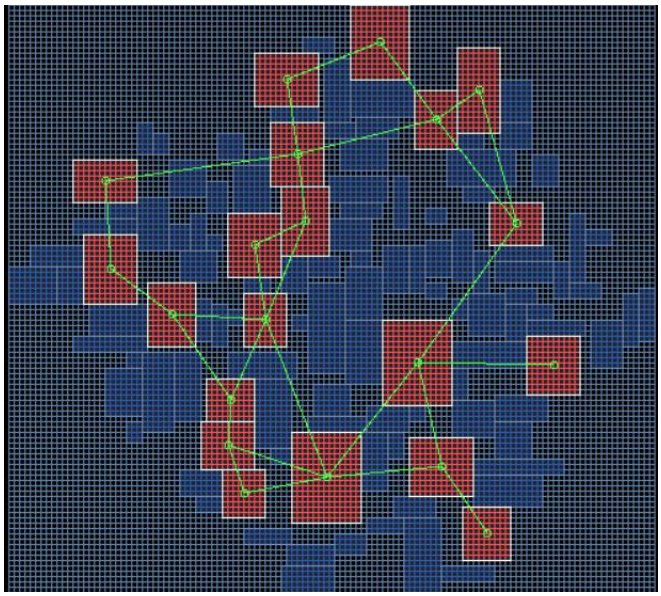


Figure 5. Result from separation map generation

The advantage of this method is that the positions of rooms is not restricted to the doors' directions since this algorithm dynamically create tunnels and doors, thus the outcome could be more diverse. The main drawback of this method is that the execution speed is slow when the map is large, because the iteration of the separation takes most of the time and scales with the size of the map and the number of rooms.

2.6 Grammar-based

Graph grammar based models [10] was first introduced in 1997. The approach uses a set of recursive rules, placed as individual nodes, to represent a graph. Later, Adams[11] , Dormans[12] introduced this idea to level generation, inspired by the work of Dormans. Van der Linden et al.[13] demonstrated a grammar-based dungeon generator by implementing it in the game Dwarf Quest[14]. The concept of this method is to first let game designers describe a graph of gameplay, then the program will use this graph to generate the level.

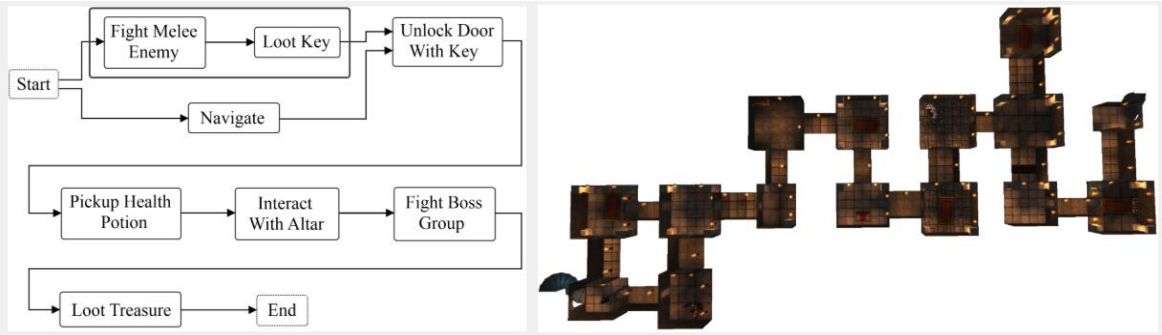


Figure 6. Grammar-based level generation - Source(van der Linden et al.[13])

The advantage of this method is that the generated map is predictable in terms of gameplay, thus it's controllable for progress design. Another advantage is that it is easy for designers to alter the map by changing the grammar. The disadvantage of this method is that the generated result feels less natural, and the diversity is also limited because outcomes are largely the same.

2.7 Evaluation

To evaluate these six methods in terms of their ability to support game design, and to compare them, I must first define what a good dungeon is and what features it should contain and implement. To do this, I Togelius et al. (2015a) [15] provide a number of criteria that can be used for evaluation. The dimensions and criteria are listed in Figure 7.

Speed	Reliability	Controllability	Expressivity and diversity	Creativity and believability
having content in time	having content in a desired quality	human can specify certain aspects	diverse set of content	content should not look 'generated'

Figure 7. Five desirable properties of level generation (Niemann[16]).

Combining the criteria of Figure 8 and the descriptions presented in Section, we arrive at the comparison matrix of the different methods presented in Figure 8.

	Speed	Reliability	Controllability	Expressivity and diversity	Creativity and believability
BSP	Fast	Medium	Medium	Medium	Low
Agent-Based	Medium	Medium	Low	High	Medium
Connection	Medium	High	Medium	Medium	Medium
CA	Medium	Low	Low	High	High
Separation	Slow	Medium	Medium	Medium	Medium
Grammar-Based	Medium	High	High	Low	Medium

Figure 8. The result of evaluation

Generally, this form of evaluation must be made by comparison. It is difficult to specify absolute values for each algorithm, but we can define a high performance or a low performance by comparing different methods and ranking them relative to one another. Still, we need to define some method and indicators for this rnaking process. The answer is to rank the algorithms and methods by "key features".

In the following section, I show how these criteria can be used to score the algorithms in terms of Low and High performance along each dimension.

Speed: BSP is fast because the amount of time it takes to generate a large map is very short. The reason is the number of rooms increases exponentially while other dimensions increase linearly. Separation is slow is because it additionally contains a try and fail loop for collision detection, which will highly increase the time when generating a large map.
Reliability: Connection and Grammar-Based are high because they have a high degree of control over either the room's connection or the map's layout. CA is low because it might generate unexpected situations like an unplayable level .
Controllability: Agent-Based, CA are low because it is hard to specify a tree or graph structure into them, thus they lack the ability to support secondary content creation. Grammar-based is high is because it takes control over the data structure being generated.
Expressivity and diversity: Agent-Based and CA are high because the resulting map's form is highly random, it's not limited to a certain shape. Grammar-based is low because abstraction of the result is the same in every generation, which means low diversity.
Creativity and believability: BSP is low because without further modifications, the result is flat and the map feels generated. CA is high because it performs best in generating organic shapes, which looks natural and highly believable.

All medium ones are considered not having big drawbacks or advantages in a certain property. I use this table only to describe the advantages and disadvantages of each method, but not to specified how these methods are ranked. As the result shows, the weakness and strength of each method is different, but can we find a different approach that cover the strengths and eliminate the weakness? In next section, I will show how I conducted my research on this.

3 APPROACH/METHODOLOGY

3.1 Generation work flow analysis

According to the overview of level generation algorithms presented in Figure 8, their common weakness is to not have enough control over map, and because the game's progress is not predictable, the reliability tends to be medium. But there is an exception, the Grammar-Based method. To find out why this method is unique, we need to first layout the map generation methods more structurally.

To do this, I divided the room generation process into three main phases: *Construct,Abstract, and Populate.* The phases are are defined as described below:

Construct: The construction of the map, thus the period when the layout is generated.
Abstract: Abstracting the layout into a data structure, like a linear structure or a tree.
Pre-Populate: Determining the contents of the map, i.e. positions of enemies, items, etc.
Post-Populate: Modify or add-on any content.

The control flow of each method is described in Figure 9, going from step 1 to 4:

	1	2	3	4
BSP	Pre-Populate	Construct	Abstract	Post-Populate
Agent-Based	Pre-Populate	Construct	Abstract	Post-Populate
Connection	Pre-Populate	Construct	Abstract	Post-Populate
CA	Construct	Abstract	Populate	
Separation	Pre-Populate	Construct	Abstract	Post-Populate
Grammar-Based	Pre-Populate	Abstract	Construct	Post-Populate

Figure 9.Control flow table of six methods

From Figure 9, it is visible that the main difference between Grammar-Based and other methods is that the Grammar-Based approach reverses the sequence of the Abstract and Construct phases, which makes it easier to control in the map generation process.

3.2 Generated result analysis

Another perspective on each method is the result data structure of each method. Figure 10 presents the resulting data structure of each approach:

	Data structure
BSP	Tree
Agent-Based	Linear
Connection	Tree
CA	Random
Separation	Tree
Grammar-Based	Graph

Figure 10. result data structure

It is noticeable that the data structure of Grammar-Based is different from all other methods. The defining characteristic of the Grammar-Based approach is that it is a graph. Crucially gprahs can be mapped onto and used to control gameplay. So what is the relation between graph and low diversity? The answer is that the graph is made by a designer and as such is fixed. A fixed layout will have less diversity, compared to variable ones, which are the data structures generated by the rest of the algorithms.

3.3 Idea forming

Could this collection of dungeon generation methods somehow be improved, taking the best from several algorithms? My hyopethesis is yes, and my initial idea can be contained in one sentence: Learn from the Control Flow of Grammar-Based method, adding Diversity to Abstraction phase and adding Controllability to Populate phase.

To do this, I adopted these two approaches:

1.Instead of making the abstraction by hand, we generate a tree structure instead.

2.Not only control the gameplay based on abstraction, but also control it in the populate phase.

Map generation is not solely based on the spatial layout, but is constrained by game itself and its rules. In the next section, I will describe how I build the game on top of this idea and how I keep improving this idea by iterating the game design, developing the map generation approach in tandem with the game's design.

3.4 Game Design

In this section, I describe how the process of fitting a procedural level generation method to a particular game design was investigated. In order to do this, I designed and implemented a top-down view turn based RPG called "Endless Tower". The game includes map exploring, field of view, battling, item collecting and upgrading status and skills. I first designed my game and implemented it in Unity3D, then I kept improving the map generation method and summarized where I did right and where I did wrong during this process, creating an account of how the procedural level design methods evolved with the game's design.

3.4.1 Game Introduction

This section describes the design of the game. The player will be spawned in a dungeonlike environment, and there are plenty of rooms waiting for player to explore, there are items waiting to be taken, dangerous enemies waiting to kill the player, but if player wants to explore the whole map, players have to collect keys to open the doors of every rooms. During the journey, players will beat enemies, drink potions, level up, learn new skills, find new

weapons, and die a lot, but each time player is dead, they gain skills to beat the enemies and knowledge of control the map.

3.4.2 Game rules

1. The game is turn based.
2. Each time the player presses arrow keys, the game advances one turn.
3. Player can only see a part of map, because there is a field of view setting.
4. Enemies that are in player's view are considered as active, otherwise they can't move.
5. Each time the player moves, all the enemies which can see the player move for one turn.
6. Player can attack enemies by moving into the enemy for one turn while standing next to the enemy.
7. Player and enemies have health, if they go down to zero, they die.
8. If player is hit by enemies, the health of the player will be reduced. The damage is calculated by the attack of the attacker and the defense of the player, vice versa.
9. If player is dead, the player may choose to continue the game or start a new game. If the player chooses continue, the level will be the same, and if you choose new game, the level will be different.
10. If the player is dead, they lose all weapons, items, status, etc.
11. Players will explore the map by collecting keys to open doors, and only if the colors of a key and a door match, it will be opened.
12. Players can collect items for gaining power, using these to beat dangerous enemies.
13. There will be shops in game world where players can level up in exchange for money.

Screen shots form the game are displayed in Figures 11 and 12.:

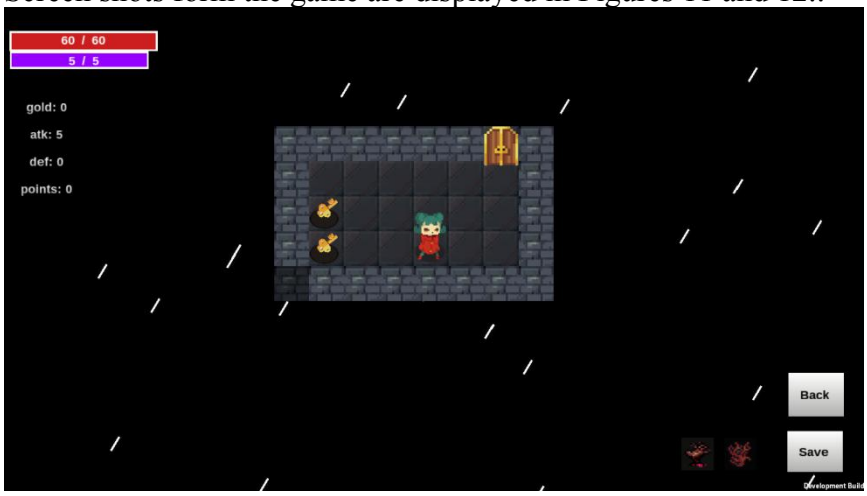


Figure 11. Player spawned in the start room.

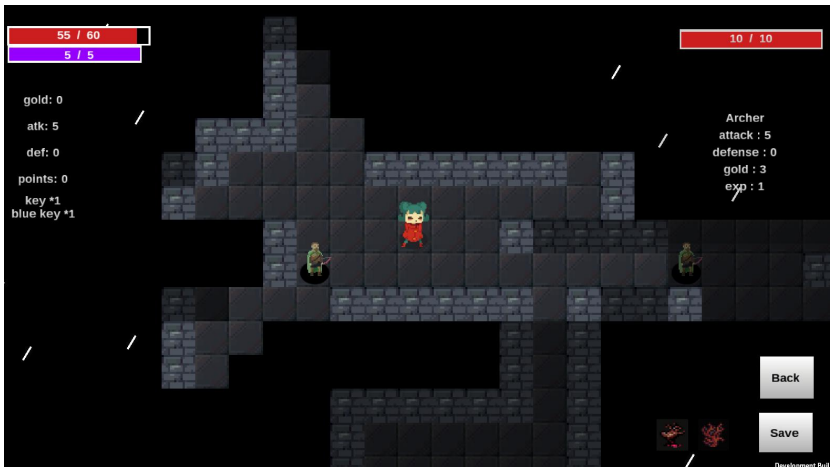


Figure 12. Player's first encounter with archers.

3.5 Developing walk through

3.5.1 Overall work flow design

After considering the ideas presented above, I developed my own map generation process to suit the design of this particular game. The map generation process is, like most of the processes described above, based on four steps:

1. *Pre-Populate:* Figure out the room's configuration.
2. *Abstraction:* Generate a tree structure as the blue print (as Idea 1 discussed above).
3. *Construction:* Layout the in-game map by the blue print.
4. *Post-Populate:* Adding and modifying game elements in the map (as Idea 2 discussed above).

The key features of this method are:

1. Not only take control over resulting data structure, but also not lacking of randomness.
2. Enabling content changes before and after layout generation, this allows us to change the content according to the layout we generated, e.g. cross room puzzles.

I will illustrate how I developed these four phases in the following sections.

3.5.2 Pre-Populate

This part is lead by two questions: What should be in our room? Is it also randomly generated too?

The answer to the first question is rooms a 2d string array, each string carries the position and type information of the tile that should be spawned in the game world. The first version of our room is a 11*11 square, but after a several generation, I found it is very limited in terms of playing space, so I changed it into 19*19. The data structure of the room is in text format.

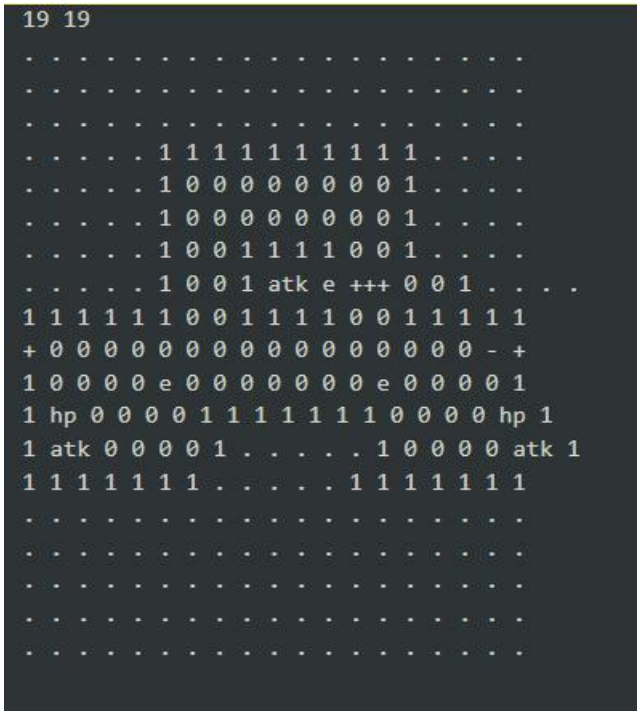


Figure 13. One room's configuration file

Here is a template of my room configuration. The first line represent the height and the width, the rest is the symbols represent different tiles in the actual game. "." means empty and "0" means floor, we will talk about all tiles in the map convert chapter.
The answer to the second question is: our room generation is in a hybrid form, which means we handmade the template before the construction phase and modified it by our algorithm in the populating phase.

The reason for starting with the handmade version:
1. Puzzles are easier made by hand.
2. Can make map generation speed faster.

And why it need to be modified later:
1. Can creates more variety.
2. Allows enemies and reward curve.
3. Allows cross room puzzles.

On the program side, our room configuration files are saved in the "Rooms" folder, once the game is start, our MapDB class read in all those room configurations and hold them in memory for later usage, and the list of these room templates are called "room pool".

3.5.3 Abstraction
Going back to the map generation analysis part, four of them are first generate the map, then abstract the data structure, that lead to a problem that the tree structure is derived by how the map is randomly generated, it's hard to alter the data structure directly in this way. And if we want to develop a narrative game, which is randomly map generated, we are more likely to design the tree structure first then apply it to a map. To make my map generation system more universally compatible with different genres, I generate the tree structure first, then I generate the map according to it.

This approach has certain advantages:
1. It provides total control of the tree structure.

2. Easily adapted to different games.
3. More friendly to design process.
4. Clear to see the higher level structure before the map is generated.
5. Changing the tree structure won't affect the later process.

Iteration #1

The first version of tree generation method is simple, we start from a beginning node and generate two children for it, and children for the generated children too, until we reach a certain depth.
But then, I found that the nodes could have a maximum of 3 children, because the game has 4 directions, and one node can connected to 3 children and one parent at once, and also, the number of child could also be random, so, in the second iteration, I changed the algorithm into:

Iteration #2

1. *Start from layer 0.*
2. *Generate the root node.*
3. *Generate children for it, by taking a random test to confirm the number, chances are equal for child count, 0,1,2,3.*
4. *Generate children.*
5. *If there is no node generated in this layer, but haven't reach the depth, generate a children for a random node for current layer, and move to next layer.*
6. *Generate children for all nodes in current layer according to rule 3.*
7. *Go back to step 5 until the depth is equal to the input.*

Iteration #3

During the later iterations, we found that three child branches are interesting, but relatively hard to deal with in terms of gameplay, but one child branches looks too straight. So I decided to add more two children nodes than one and three ones.
So we changed the probability for different count of children:
0 children: 0.1
1 children: 0.2
2 children: 0.5
3 children: 0.2

Iteration #4

The third version went well for few iterations, but I found a problem:
There might be too many branches at last, we usually want player to follow one path to the end, with those branches, players might get lost in a big map.So in the third version of tree generation, we alter the number 3 rules to : Generate children for it, by taking a random test to confirm the number, while the depth goes higher, the random child number tends to be reduced. We did this by adding a balance factor :
*(Depth-current)*2/Depth*
After introducing it to the random test, we can ensure that the deeper our current level is, the fewer chance it will get higher children, and vice versa.

Final result

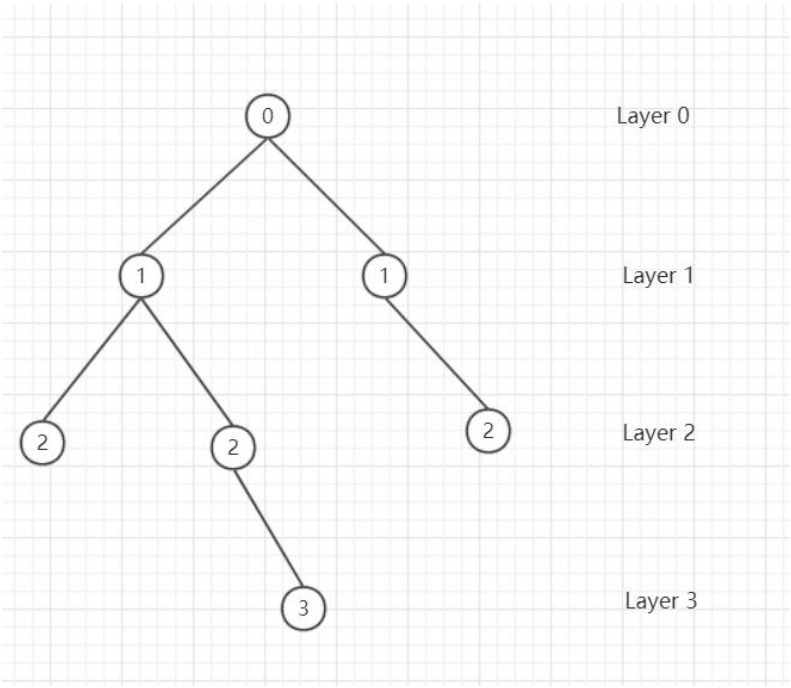


Figure 14. result after abstraction phase

3.5.4 Construction

For all advantages we have in this new work flow, it also brings a problem that we need to solve, which is, how to convert the tree structure to the actual map?
The answer is specific to games themselves, if your game is a 3D modeled game which terrains, you probably want to combine with your terrain generation method to apply this tree, and in my case, my game is in 2 dimension and only allow 4 direction movement,so I decided to generate it according to grids.
Before think about how we could convert a tree structure into a map, we should first consider how the final map should be like:

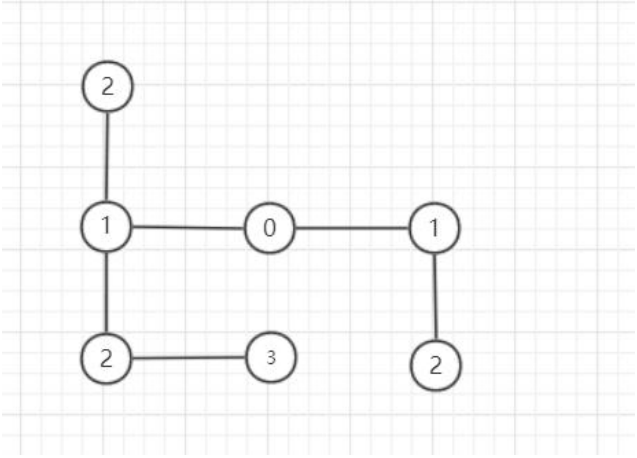


Figure 15. node representation of final map

3.5.4.1 Node formation

My first approach of forming this kind of map is by starting from the start node and randomly choose one direction for each child node. But After implement it, I found a new problem: what if a node's child can't find its place because 4 directions are all occupied?

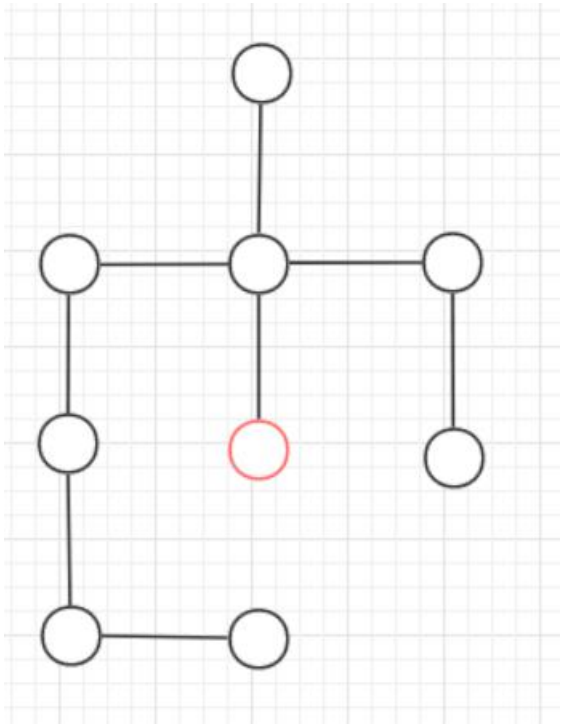


Figure 16. error node representation

The red node's 4 directions are all occupied,it is acceptable to some degree that the child to be spawned is just a branch room, but it will make things worse if the child is a important room,like a boss room, and this will prevent players from completing the game.
To solve this problem, I developed a two step generation method, first we generate the main path by a two dimensional Random Walk algorithm[7], which means it contains all necessary rooms player need to explore, and in this process, we not only solve the problem by ensuring that a child node will always have a possible position, but also gain the diversity property from Agent-Based Algorithm.

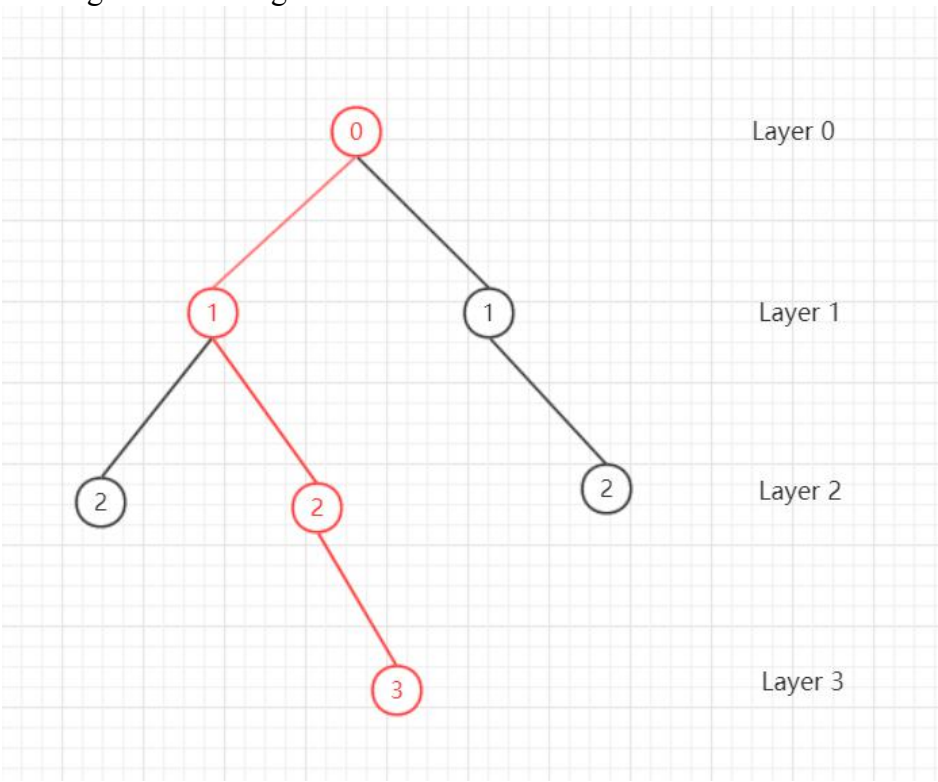


Figure 17. result after confirming main path

After ensuring the main path is constructed, our next step is simply pruning the branches which having issue in children positioning, then our tree is perfect for us to generate the rooms.

3.5.4.2 Room selection

Room selection are based on the direction of doors it holds, each room have certain amount of doors on border, and that the tool to connect them together, this concept is adopted from Connection method described in previous section, in this way of room selection, we can take the high reliability from it, and this strength not only comes from Pre-Populate phase's room presets, but also fixed doors positions that preventing our rooms from performing unnaturally.

Iteration #1

The algorithm is written as *FindRoom(int[4])*, the parameter is an integer array which contains the doors information, for an instance, if room has two doors that are up and left, the rule for door list sorting is the up direction is the first element and others are clockwise arranged, in this case the door array will be {1,0,0,1}, then we select a room from our room pool that has the same door array as the result.

Iteration #2

In the later iterations, we found that the start room and ending room's configuration should be special, start rooms contains an "@" which represent the player, end rooms contains a "B" which means boss, so the selection method was increased to three: *FindStartRoom(int[4]), FindEndRoom(int[4]), FindMiddleRoom(int[4])*, for start rooms, our searching pool is limited to rooms which has an "@", and for end room, we limit our searching pool to rooms which has a "B", and then apply the door list for final filtering.

Result of Room selection

After the room selection process, the final result should looks like this:

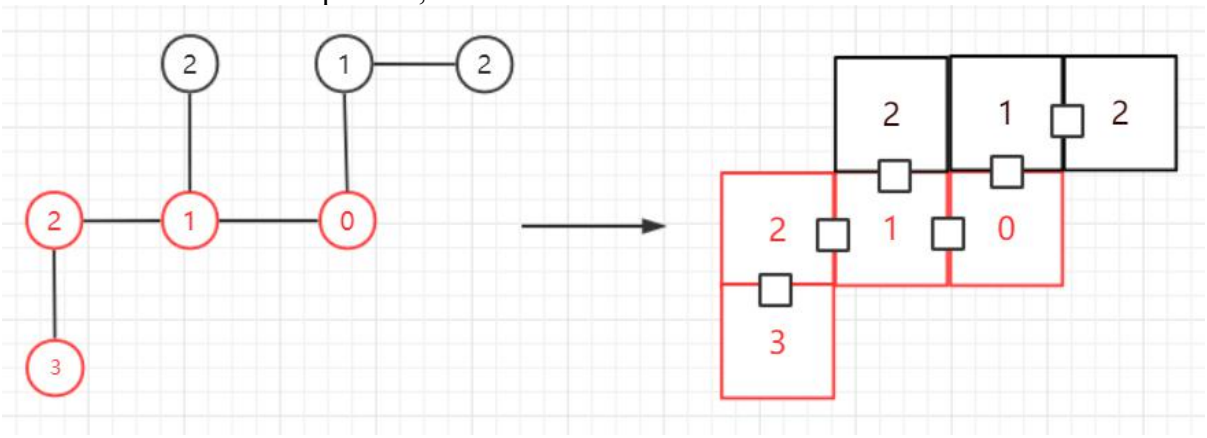


Figure 18. transfer from nodes representation to game map

The small squares are door positions.

Result of previous work

And for now, our map construction phase is done, the overall structure of the process is :

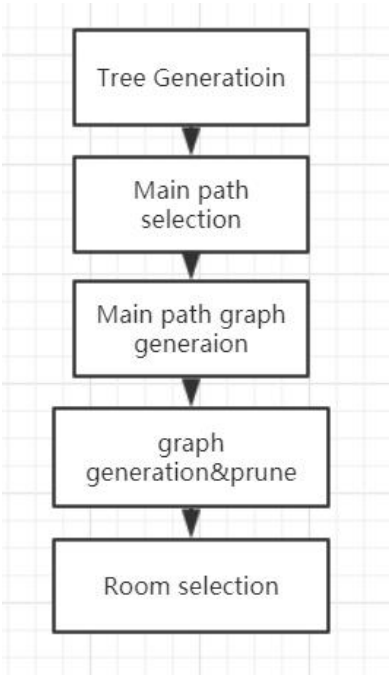


Figure 19. Processes in Abstraction and Construction

This graph showed universal ideas of map generation, most important ones are the idea to use tree structure as the blue print and the process of tree structure's adaption to a 2d map.These ideas are not specific to my own game, they have potential to be used in other ones.

3.5.5 Post-Populate

In the populating phase, the mission is to modify and add elements into our map, the final elements I am going to implement in my game are: *Cross room puzzles, Entity positioning ,Enemies,Difficultly curve, Special enemies, Supplies, Items,Shops,* but they are not done at once, but in few iterations, let's begin from our first iteration.

Iteration #1: Red doors, enemies, merchant and supplies

3.5.5.1 Red doors

Red doors are representations of different stages, after a red door, you can access stronger enemies and different items, The idea of it's generation is to put it every two layers.
Red doors can only be opened by red keys,the idea of key generation is to put it before the current door but after the previous door. For an instance:

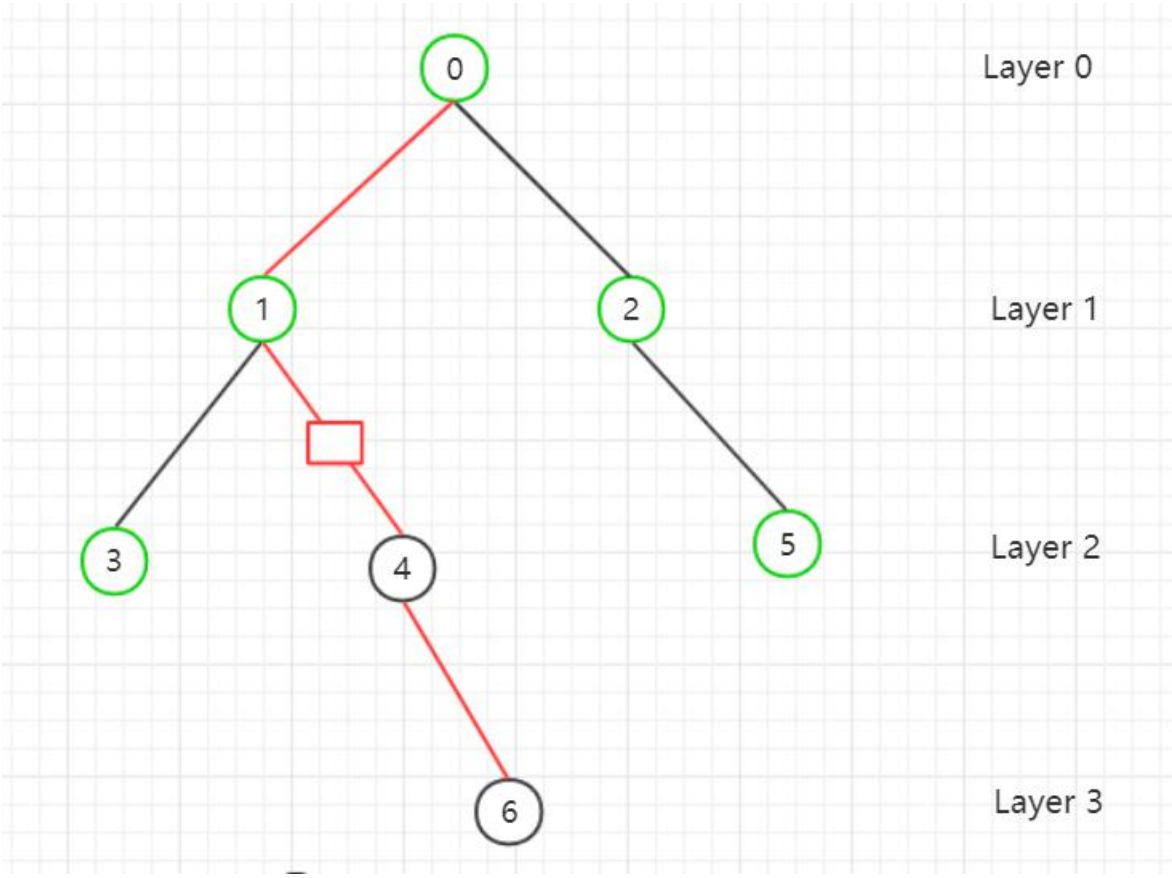


Figure 20. Possible locations for generate red keys

The red door on graph is between the nodes 1 and 4, the potential position of red key is not limited to nodes 0 and 1, it also includes branch nodes which are 2,3 and 5.The advantage of this approach is it can encourage the player to visit all the rooms, and it also can make the playing experience closer to a handmade level experience.
For the program part, the algorithm is *GetNodesChildFromStart(Node,Node)*, it takes two parameters as input, the start node and the end node, and the process is add all nodes derived from start node but not from end node, after this process, you can get the ideal positions result.

The **door/key** puzzles are just one representation of cross room puzzle, it can be **switch/doors, pedal/traps**, the core of this back tracking ability is *"things in a room can have effect on another room/rooms"*, the usage of this is not limited to generating key/doors puzzles, for an instance, events. You trigger a pedal in room 3 and suddenly, a firewall is spreading in your way back. You triggered a switch in room 2 and then two new rooms appears, allows you to leveling up your hero and discovering more items. You pressed a button in room 7,suddenly the screen is shaking ,and there is a prompt comes along "a wall is cracking", now it's time for you to find the secret room.My usage of this concept is limited but the potential of it is huge and also fun to discover.

3.5.5.2 Enemies


Figure 21. The enemy

Enemies are very important roles in my game.
For the big picture, it can:

1.Create senses of Exploration, Challenging, and Accomplishment.
2.Introduce more mechanisms.
3.Allow players making decisions.
4.Allow the sense of progression.
5.Create safe/dangerous atmosphere.

Talking specifically,first, battling highly required right positioning, because my game is turn based, it allows more time for players to decide which tile show he/she moves to, second, killing enemies can gain golds and experience, these introducing shopping and leveling up, which gives players more decisions to make and increase the variety of tasks. Third,when encounter with enemies, there is a sound effect indicate that an enemy has found you, different enemies have different sounds, the more dangerous one have more scary ones, thus create the atmosphere of dangerous, and in contrast, safe.
Dying also plays an important role, it allows players to remaking decisions like not entering the room that he/she died from in the last run and challenging easier enemies, opening different doors this time,this allows player getting more and more familiar with the map and eventually can beat the boss and end the game.

Enemy iteration #1

Enemies keeps chasing player endlessly.
Enemies' status is fixed.

In this version, I found that the battling is too intense, and decisions player have are limited while battling, also, in this version, we have to determine enemies' status by hand.

Enemy iteration #2

Enemies chasing player with a gap of 1 turn.
Enemies are curved according to room's layer level.

This time , the battling feels far more interesting, because the player's moving decision can avoid being hit by enemies. For enemy curving, we need to assign a power index to each room, the initial value of it is the current layer of a room, then the algorithm for curving is, if the room is in the main path, divided the power index by 2 and get the largest integer less than the result, if not, after the previous calculation, adding 1 to the result.This means the difficulty in the branches are harder than the main path.After this step, you will get the final result of symbols like e_0, e_2 in the room configuration file.This approach makes it more flexible in difficulty setting.But the current version of combat is also boring because after a few encounter with enemies, you can find the moving pattern and the combat feels not excited anymore.

Enemy iteraion #3

Divide types of enemies(Enemy mutation).
Merage with curving system.
Adding new enemies.

In the final version, we defined two primitive types of enemies, the first one is called guard, the mission of it is guarding one spot on the map, and once player getting close into its walking range, it begins to chase the player, but if it gets too far from the guarding spot, it will return to the origin position,Second one is the killer, once it find the player, it keeps chasing until losing the sight or die.These are two primitive types of enemies, which is represent by 0 and 1. Meraging with the curving attribute, the result should be like e_0_1 as a level 1 guard and e_1_2 is as a level 2 killer.
And our mission also include deciding which enemy on the map should be a guard and which one should be a killer, because of that enemies positions are already decided in room configuration files, so my algorithm only have to assign a type symbol to current enemy symbol in configuration file. The algorithm is when a cell is an enemy, and it can find any kinds of door,key,or items adjacently, it turns into a guard. This approach means to block the player if he is not meant to enter a certain level's room, and players have to kill them to get certain rewards.The rest of the enemies will turn into killers if they neighbor enemies are less or equal to three, this prevent multiple killers happens in one area preventing players from completing the level. The final enemy setting is in appendix.

3.5.5.3 Merchant



Figure 22. The merchant

Merchant is a character exists in the map, by spending one move turn, we can trade him with golds for attack, defense points, or restore certain amount of health.It is a indirect way of gaining status, create variety of play and fun of exploration.There is a graph shows in detail how it influence the game.
After talking about the usage of merchant, let's talk about the generation method.

Merchant iteration #1

Generate the merchant in separate layers' corner position.
Layer finding : *Every two layers counted from layer one.*

This step means to limit the frequency of shop, thus allow the scenario of shop finding.Without this step, merchant will be meaningless because its everywhere, if that happens, why not bring it directly into a shopping UI button.
But there appears a problem that if a room doesn't have a corner which means haven't get a position with 3 adjacent cells are walls, that will lead to a generation false.

Merchant iteration #2

Put the merchant on a edge position.Edge means a position with at least 1 wall as neighbor.

This time, the possible positions are increased, and the merchant will always get a position because there should be no room with no walls.

But after some gameplay, a new problem appears, sometimes the merchant will block the player's move.

Merchant iteration #3

Put the merchant if surround wall count is odd.

Because of the merchant is not able to removed from the map, it might blocks players path, there are all situations that might happen.



Figure 23. possible locations to get stuck

There are more than one algorithm to prevents those from happening, and because in all those situations, counts of wall neighbors are all even, I just choose odd count situations, this is fast but also eliminate some accessible situation from even ones.Notice this process is not limited to merchant itself, it universally refer to things that are permanent in map, which can also used in wall generation.
For now, the positioning of merchant is complete, but to encourage more decision making and adding more variety of game play, I decided to add curving method to merchant.

Merchant iteration #4
Curve each merchant by its index.

This concept of leveling merchant is adding the strategy of saving and spending. Merchant with higher level have a better cost performance than lower level ones. There is a table shows the price calculation in appendix. Also to let he player more directly recognize the level of merchant, we paint the merchant in different color according to its level, our current solution is by increasing the blue channel.For now, the merchant generation is complete, it has a potential to create an infinite playing loop with further modification and improvement.

3.5.5.4 Supplies



Figure 24. The health potion

Currently there is only one type of supplies which is health potion.
The main propose of this potion is to increase the player's health and prevent players from dying..
The generation method of supplies are hybrid, their positions are first determined by designer and then modified by the algorithm, the reason that why it is not completely generated by algorithm is that hand made is more easier in terms of purpose based positioning.

First, let's see the basic rules of placing supplies.

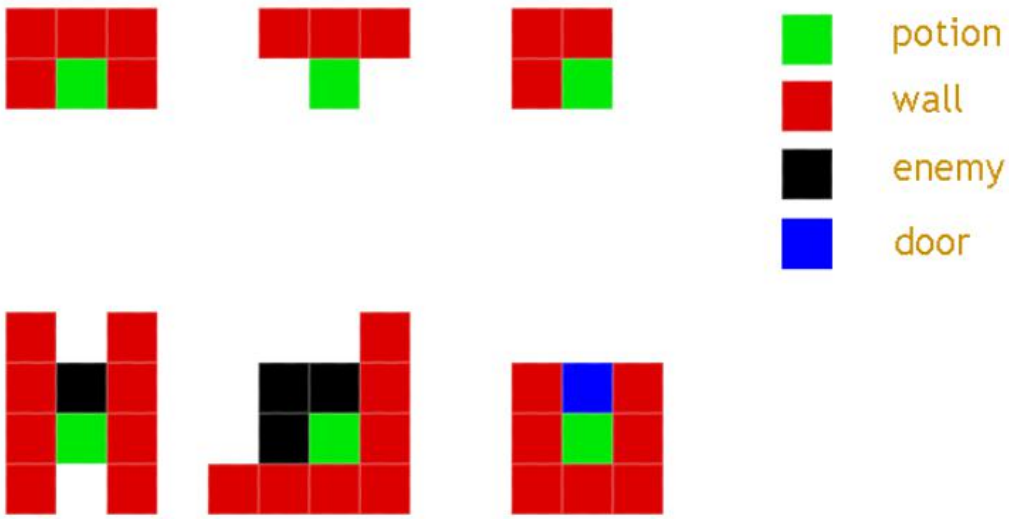


Figure 25. combination positions of potions, walls, enemies, and doors

These are common positions of supplies, first they should be placed against the wall to be considered natural, second, they have to be placed based on purpose of reward or discovered. The reward is represented by number 4,5 , which you have to defeat the enemy to get it, the discovered is represented by number 6 in graph, it means you have to open a door to get it.The actual purposed based positioning is not limited to last 3 forms, it is varied in designer's hand.After our program having these preset locations, they are all considered as health potions, and now the positioning of supplies is complete.

The game feels playable and interesting for now, but I want to add more content that related to cross map puzzles to make it more interesting, and for the enemy part, because of all enemies' positions are determined by hand, it is less flexible, so I also want instances that could be complete generated by algorithm.And I also want to add more items into the game world to allow more decision making process.

Iteration #2: More cross room puzzles,Special encountings and rewards.

3.5.5.5 Blue Keys

Another usage of this cross room puzzle concept in this game is generate blue keys, there are certain amount blue doors in the map, they are usually guarding valuable treasures like gems or scrolls, and the location of blue doors are stored in the room configuration file, but the location for blue keys are not, and because of blue doors are not in a single path like red doors were, instead they are scattered around, so I developed a different algorithm for its generation.

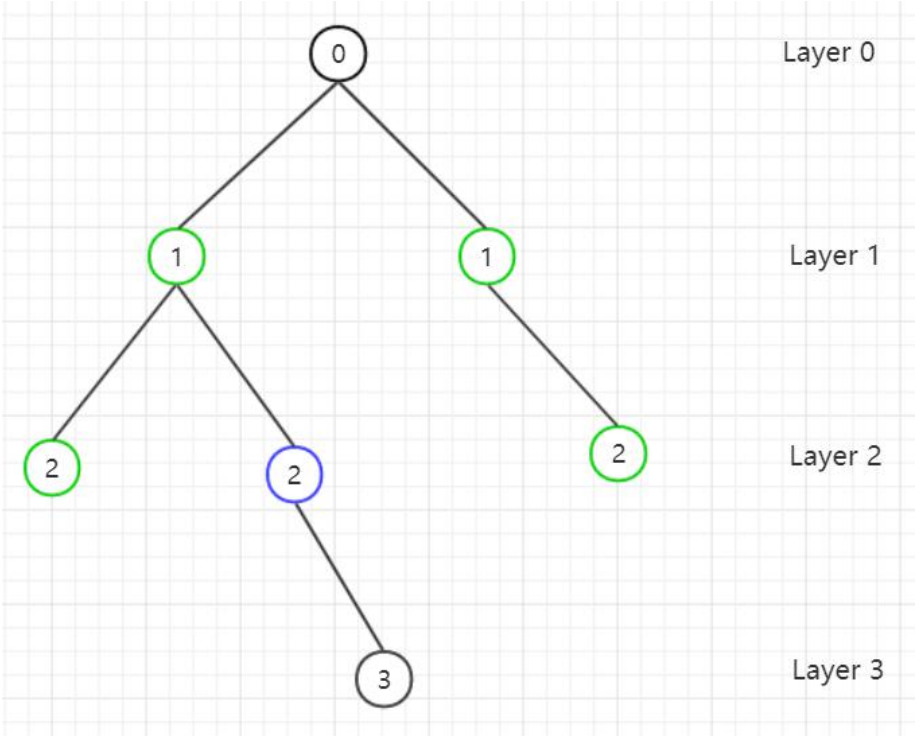


Figure 26. possible locations for blue keys

This time, we only allows previous one layer plus current layer to be the target space, the reason are

1.Limit the distance of blue key/door pair.

2.Prevent clustering of blue keys.

3.Teach player that blue doors should be opened by the key in a similar difficulty room.

The algorithm goes like *GetNodesChildFromLayer(int,Node)*, it takes two parameters, first is the start layer, and the second is the end node, the process is we first add all nodes from start layer to the current layer if the node is not the end node, doing this means keep the blue keys from doors in a distance at least 1, this means to prevent these two cases from happening.

1.A blue key is generated near a blue door.

2.A blue key is generated inside the room guarded by a blue door.

3.5.5.6 Special enemies

To achieve the goal of completely generate enemies by algorithm, I decided to add special enemies into game.These special enemies are considered as special events, they can gives player a sense of surprise. The current approach of doing special is by adding status to the enemy, currently there is one special enemy in game, it is overpowered in terms of room level.



Figure 27. The special enemy

Player can choose to challenge it immediately after encounter or ignore it and focus on main path's enemies.We generate it in the branches of layer 3, placing it while the neighbor wall

tile's count is greater than one, the generation of it after is outside of mutation and curving loop.

3.5.5.7 Gems



Figure 28. two gems

Gems are consumables that can permanently increase status of player, there are two types of gems, ruby and sapphire.Ruby can increase player's attack and sapphire can increase player's defense, which are two most important attributes in my game.
Gem's placement pattern is same as supplies, but they are rarer, usually behind blue doors or surrounded by 3-4 enemies.Because of defense is more valuable than attack, the number of sapphire should be smaller than ruby, to achieve this goal, in our configuration file, there are less sapphire than ruby.
During the earliest tests, we found that gems are too powerful for start games, so we introduced a limitation function that restrict the number of gems for each room according to its layer.

Ruby: *layer*2+1*
Sapphire:*layer+1*

The exceeded ones are not deleted, but turned into health potions, this can prevent player's frustration after opening a door and gets nothing.

In this version, the game feels more compelling, but I found that while playing, the long term goal is to complete the map, the short term goal is to get items , but there is a lacking of medium term goal, so I decided to add more content into the game.

Iteration #3: Magic potion, skill scroll, and weapons.

3.5.5.8 Magic potion

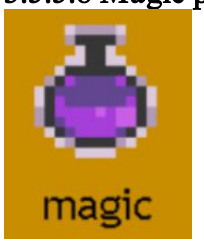


Figure 29. The magic potion

To add variety to our supplies, and allow players to restore magic after combat, I decided to add magic potions as new supplies.
It's main function is adding magic to player thus add the frequency of player casting spells.
The Generation of it is based on supply generation talked above. After getting all supplies' positions, we will do a random number test to determine whether it should be a magic potion.

Possibility of magic potion:

oddFactor(wallNeighborCountIsOdd?1:0)*+originFactor*

Wall neighbor count means the count of wall tiles in the around 8 cells, and if that is odd, there will be higher probability to be magic potion, this approach means to capture most of

magic potion in the corner.And after this calculation, all locations of magic potions are found, and to merge with health potion generation, we generate this first, and let the rest supplies positions to be the health potion after, now the whole stage of supplies generation is named “Supplies mutation”.

3.5.5.9 Skill scroll



Figure 30. The skill scroll

Skill plays an important role in this game, reasons are

1.It can add more depth to the hero.
2.It allows more decision making process.
3.It forms different strategy of playing.

And the way of level up skills, instead of leveling up our hero,we upgrade them by collecting skill scrolls, this design choice can

1.Increase player’s desire of exploring the map.
2.Separate player’s exploration experience from player’s combat experience.
3.Adding variety to item type.

The generation of skill scroll is based on algorithm.First we confirm the room is on a suitable layer, then take a random number test, if it passed, we then find a right place and spawn it.

Layer finding:*larger than depth/2/-1*
Positioning:*surrounding wall tiles is greater than 1.*

3.5.5.10 Equipment

Equipment and skills can both adding depth to a character, but it also have its unique function,it serves as a different way to upgrade player’s status, and you can only equip certain amount of them, this allows decision making.
The generation of equipment is same as skill scrolls, i.e. each possible position for a skill scroll, it is a possible position for the equipment to spawn.

3.5.5.11 Summary of Post-Populate phase

After these three iterations, our Populate phase is complete.
The overall diagram of Populating is :

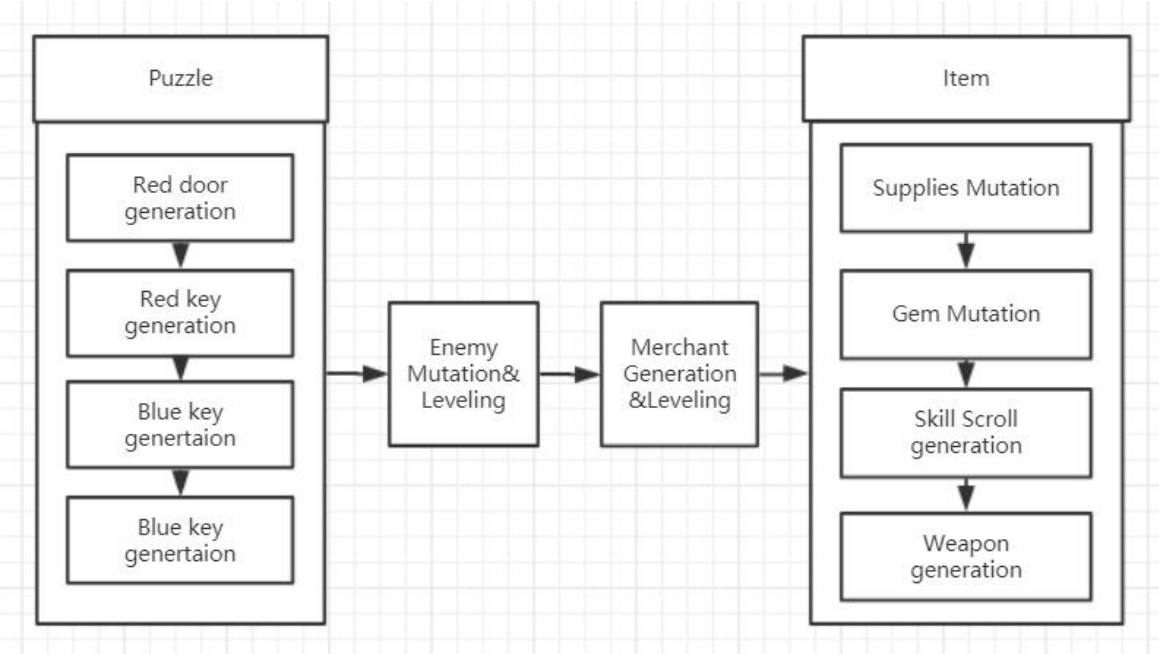


Figure 31. Overview of post-populate phase

3.5.6 Final Image

The result of abstraction, construction and population:

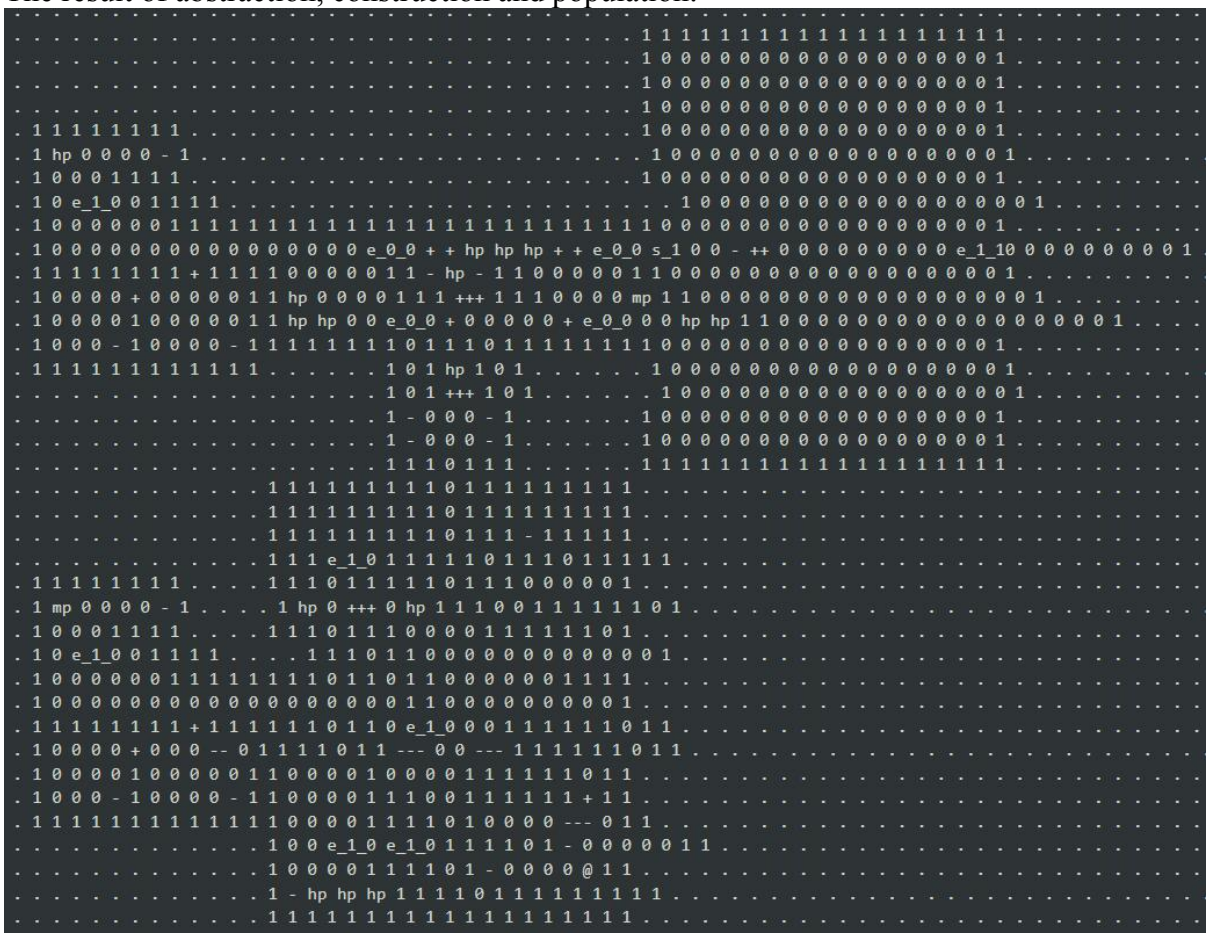


Figure 32. Result after Abstraction, Construction and Populate

Here is a **depth=3** map, which means there are 3 layers of rooms, and the reason that the shape of the text are not rectangular is the length of different varies, some strings like e_0_1

will take extra space then a single "@", our parsing function is to divide text by space, so the out looking won't effect the final result.
After the construction and the populating phase, our next mission is to convert string into actual object , so player can interact with them in the game world.

3.5.7 Map Converting

The convert phase's mission is to bring those strings into the game world, to do this, it has to interactive with two classes, a parser and the MapConvertor class.
The parser class first convert the file into a map, which in our program, a string[,] array, and then,our MapConvertor class has three main missions.

1. Find the tile for each string.
2. Spawn the tile in right location.
3. Level up enemies.

For the first mission, the MapConvertor class owns a list that linking a string to a GameObject, here is an table how strings are linked to objects:

1	0	floor
2	1	wall
3	@	player
4	e	enemy
5	e_1	killer enemy
6	-	yellow key
7	+	yellow door
8	--	red key
9	++	red door
10	---	blue key
11	+++	blue door
12	s	merchant
13	hp	hp bottle
14	atk	attack gem
15	def	defense gem
16	mp	mp bottle
17	wp_0	sword
18	wp_1	shield
19	a	ability scroll
20	g	gold
21	1_0	normal wall
22	1_1	blue wall
23	1_3	red wall

Figure 33. String and Object pairs

For the second mission, we read our string[,] from left to right and from top to bottom, in this case the down direction is the positive y direction, but in Unity3D, the down direction is the negative direction, so during we spawn our object per unit in x direction, we have to spawn them per -1 unit in y direction.
For the third mission, we parse the "e_x_x" string and get the last index, then pass it to the levelUp function that combined with the enemy's GameObject to level up the current enemy.
After this stage, the game's environment is complete.

4 Results

4.1 Overall looking

Here is a overall screenshot of the final game looking:



Figure 34. Final image

4.2 Evaluation

I evaluated my result map generation method in the same way of other map generation methods.The five desirable propeties described above.

Speed: The speed of my map generation method is medium, although the generation phases are relatively linear, it also has time consuming functions, like position finding, room finding, and file reading/writing, all those three are have same issue as others, that's when it comes to large maps, it will highly increase the processing time.
*Reliability:*The reliability of this method is high,The quality of the outcome map is good, every generation cover the whole play through content, and players will can always complete the map with the right strategy.
*Controllability:*The controllability is high, this is achieved by our hybird generation structure, thus in our Pre-Populating phase, we can specify the room's elements and also in our Post-Populating phase, we can override them all.
Expressivity and diversity: We have a high diversity , the reason is every generation's structure, layout, and room's content is different, but if we can't get enough preset rooms, the outlooking of individual rooms will be similar, thus our diversity will reduce, to overcome this problem, so we have to add enough rooms from the beginning.
Creativity and believability: The preset rooms are hand made, this can increase the believability, and rooms are connected by algorithm, this increase the creativity, thus both the overall looking and the individual rooms looks natural. But it still can't get as high creativity as cellular automata, thus still remain a medium rank.
Compare table:

	Speed	Reliability	Controllability	Expressivity and diversity	Creativity and believability
BSP	Fast	Medium	Medium	Medium	Low
Agent-Based	Medium	Medium	Low	High	Medium
Connection	Medium	High	Medium	Medium	Medium
CA	Medium	Low	Low	High	High
Separation	Slow	Medium	Medium	Medium	Medium
Grammar-Based	Medium	High	High	Low	Medium
Tree-Based	Medium	High	High	High	Medium

Figure 35. Our results compared with other map generation methods

5 CONCLUSION

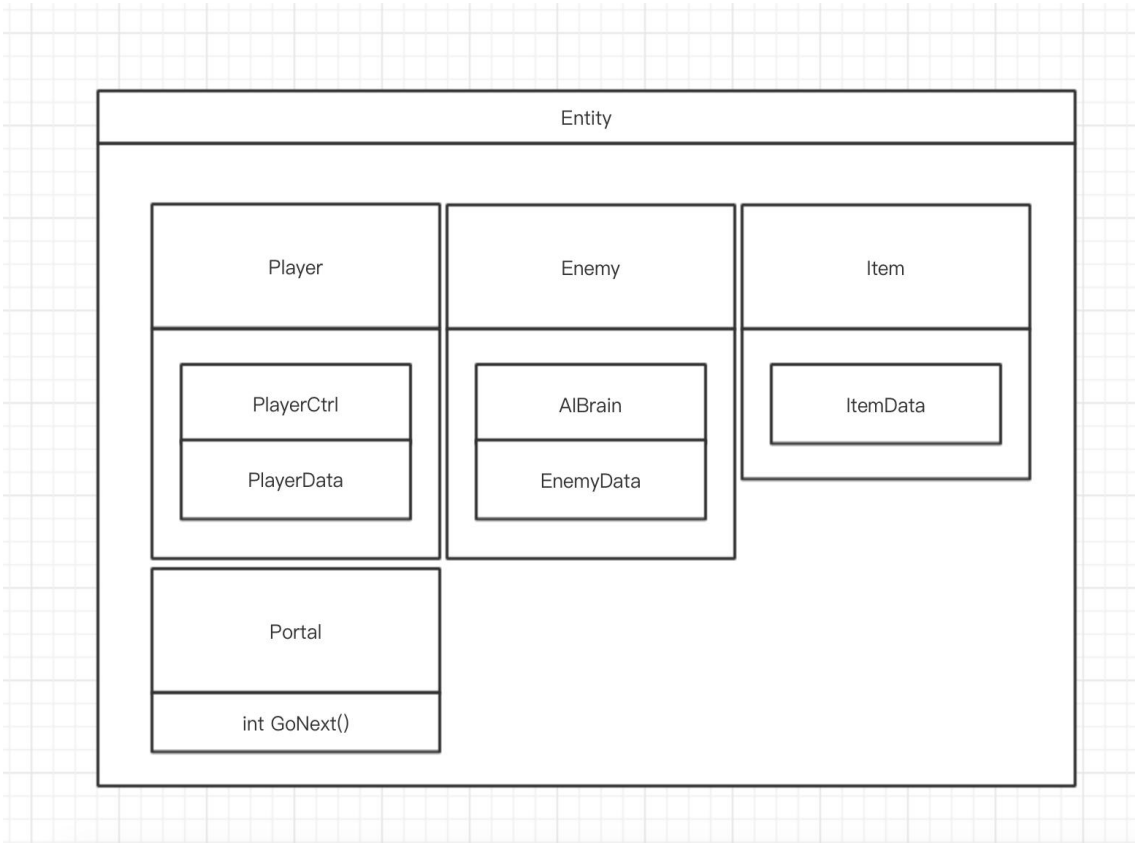
The initial goal of the thesis is to make improve on current map generation methods, and to do this, I created a game digital game called “Endless tower”, it is a single player turn-based strategy game, it took ideas from other classic RPGs, like Rogue, Brogue and Tower of sorcerers. During the game, players will control the character to explore the map, defeat enemies, find items and finally, fight the boss and complete the game.
Frist I researched for most popular 2d map generation methods, and after collecting them, I deep dived in their procedure and performance, during this period, the idea of my own map generation method is formed.My map genaration method took the high level structure of Grammar-Based generation method, and took ideas of Agent-Based algorithm and Connection algorithm for map construction and room configuration. With the game’s own emeies and items, after several iteraions, finally formed a complete map generation system.
In this search, analyze, explore, and develop process, it expand my knowledge in procedural map generation, and let me have a deeper understanding of game maps.
Further work will be keep adding content into the game, introduce new cross room puzzles, enemy AI, new weapons and skills to make the game more interesting.

6 REFERENCES

[1] Short, T., & Adams, T. (2017). *Procedural generation in game design*. Boca Raton: CRC Press, Taylor & Francis Group.
[2] Hiwiller, Z. (2015). Players making decisions: Game design essentials and the art of understanding your players. New Riders.
[3] Pedersen, Kim (2014a). Procedural Level Generation in Games. url:http://www.raywenderlich.com/wpcontent/uploads/2014/05/rw_procedural_level_generation_final.pdf. Preuß, Mike (2015). Search-Based PCG.
[4] Shaker, Noor and Antonios Liapis (2013). Lecture 3: Constructive Generation Methods for Dungeons and Levels. url: https://blog.itu.dk/MPGG-E2013/files/2013/09/3-dungeons1.pdf.
[5] Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural content generation in games*. Switzerland: Springer International Publishing.
[6] *Pearson, K. (1905). "The Problem of the Random Walk". Nature. 72 (1865): 294.*
[7] Stammer, D., Gunther, T., & Preuss, M. (2015). Player-adaptive Spelunky level generation. *2015 IEEE Conference on Computational Intelligence and Games (CIG),* 130-137.

[8] Dahlskog, S. (2016). Patterns and procedural content generation in digital games:: Automatic level generation for digital games using game design patterns. Studies in Computer Science, Studies in Computer Science, 2016.
[9] Frank, J. (1984). Cellular automata.
[10] Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific (1997).
[11] Adams, D.: Automatic generation of dungeons for computer games (2002). B.Sc. thesis, University of Sheffield, UK.
[12] Dormans, J.: Adventures in level design: generating missions and spaces for action adventure games. In: PCG'10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games. ACM (2010).
[13] Van der Linden, R., Lopes, R., Bidarra, R.: Designing procedurally generated levels. In: Proceedings of the the 2nd AIIDE Workshop on Artificial Intelligence in the Game Design Process, pp. 41–47 (2013).
[14] Wild Card: (2013). Dwarf Quest.
[15] Togelius, Julian, Noor Shaker, and Mark J Nelson (2015a). "Introduction". In: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Ed. by Julian Togelius, Noor Shaker, and Mark J Nelson. Springer, pp. 1–15. url: http://pcgbook.com/wp-content/ uploads/chapter01.pdf. – (2015b). Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer.
[16] Niemann, M., & Preuß, M. (2015). Constructive Generation Methods for Dungeons. In Seminar-Thesis in Procedural Content Generation for Games. Westfalische Wilhelms Universitat Munster.

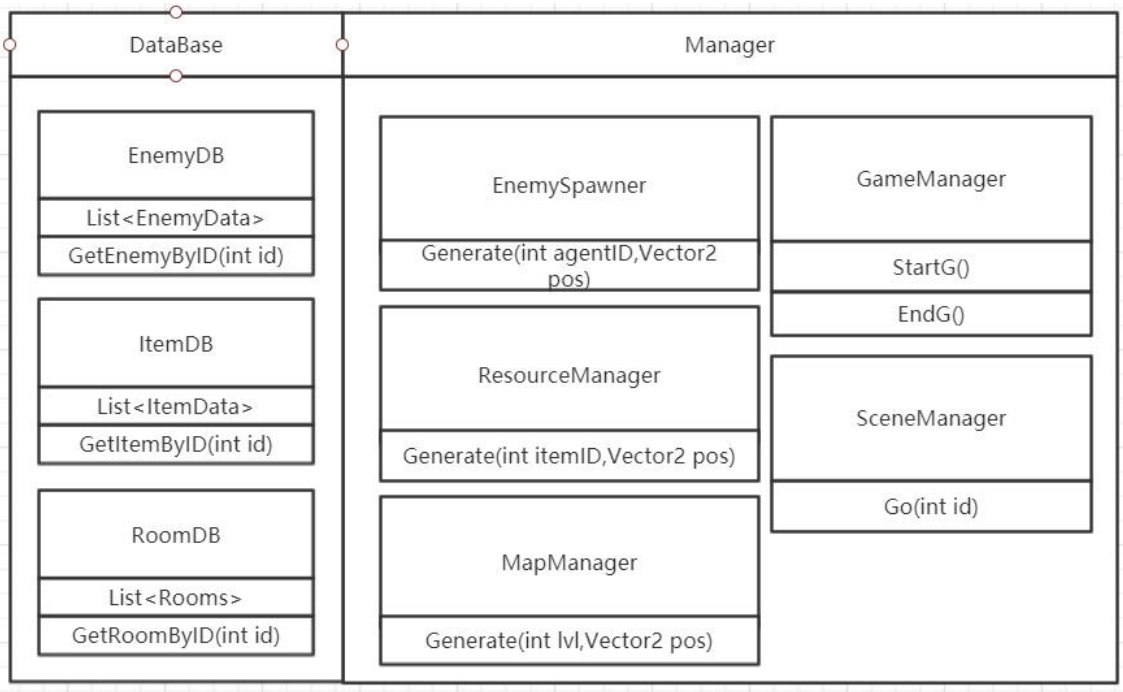
Appendix 1: System architecture



The figure above shows all the in-game objects which are a player, enemy, item, and portal, the player control script contains how to process the player's input and how the agent could interactive with the world like get an item or get hit and update PlayerData.The AIBrain of Enemy class instructs the behavior of enemies and upstate EnemyData.Item only contains ItemData which is used when player getting an item and add it into PlayerData, and the Portal class let the player jump into next level with a new map.

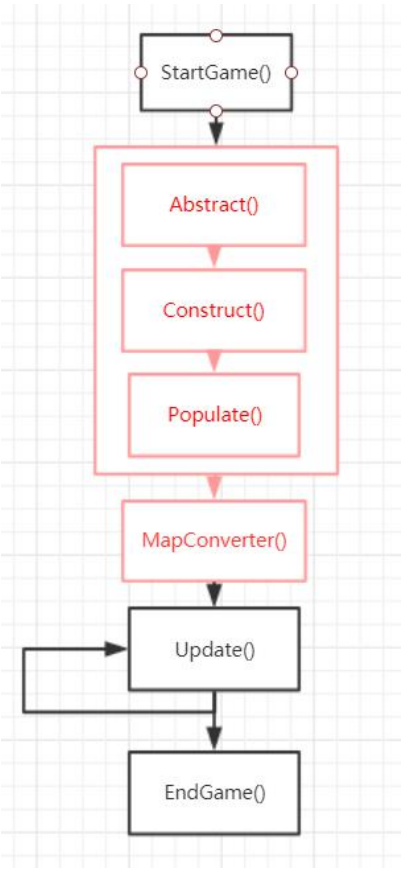
PlayerData	EnemyData	ItemData
hp	hp	ItemID
gold	gold(toDrop)	Use()
List<itemId>	List<itemId>(toDrop)	UISprite
exp	exp(toGive)	DropSprite
lvl	lvl	
atk	atk	
def	def	

The figure above shows the basic properties of PlayerData, EnemyData, ItemData class in the first iteration.



The figure above shows manager classes of the game, those are all helper functions which stays at a high level that allows me to create my map more effectively.

Appendix 2: Game loop



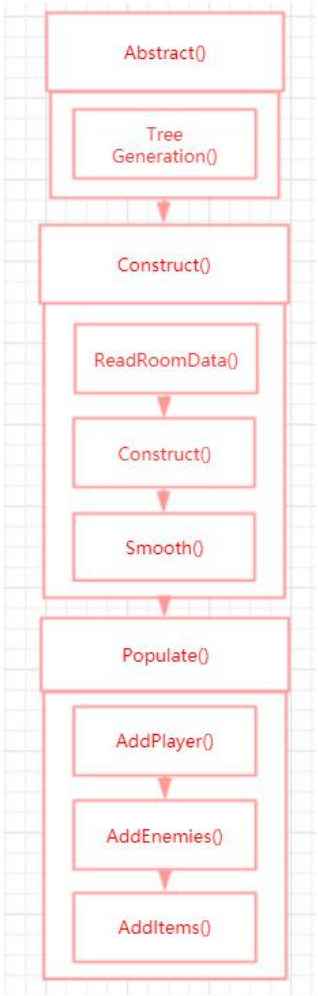
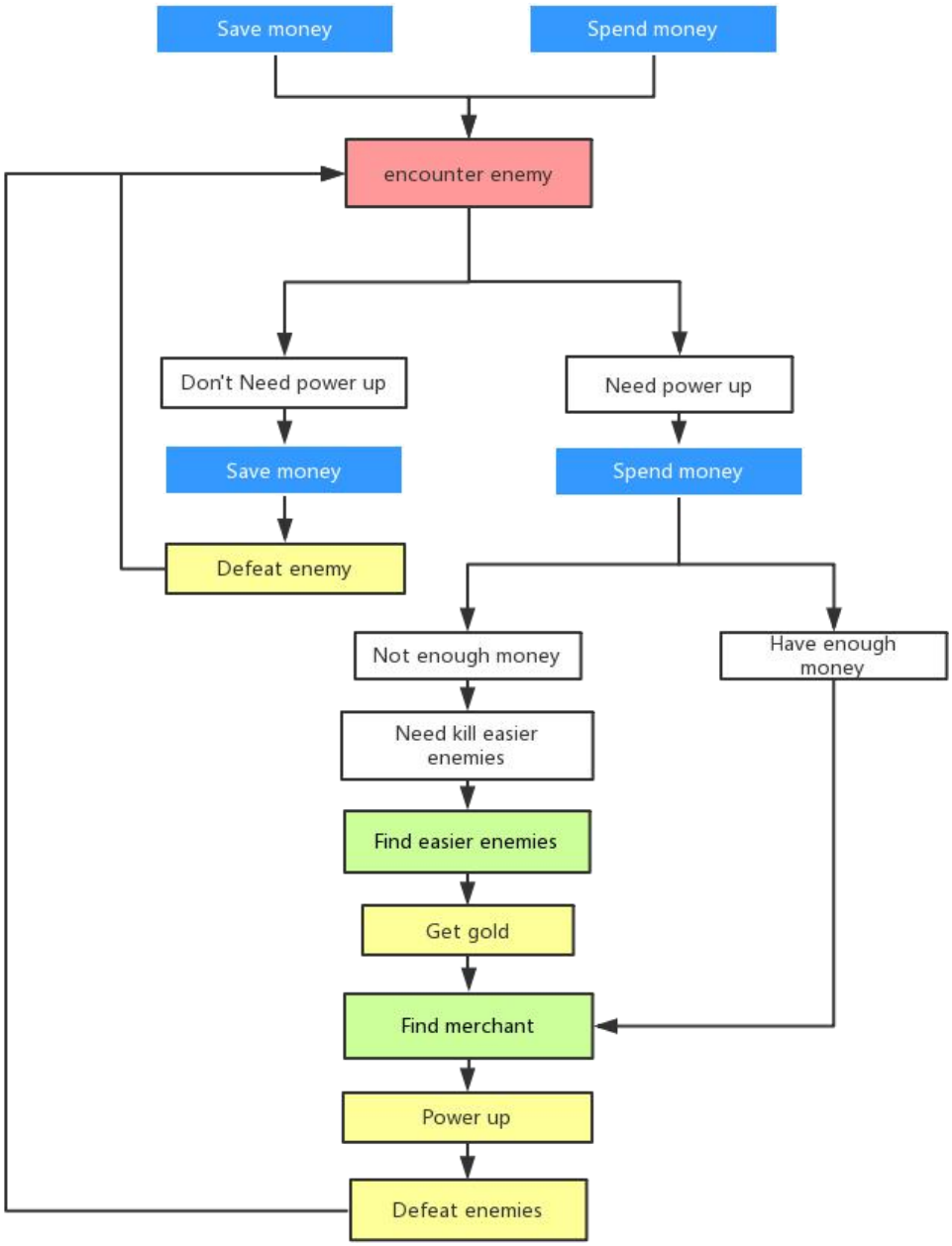


Figure on the left shows the main game loop, red ones are the part that related to map generation, Abstract Method output a tree structure that represent the layout of the whole map, Construct method take the tree as the input and positioning the nodes to output a text file for Populate method to use, and Populate method decorate the text file and output it for MapConverter method to turn it into the in-game map.
Figure on the right shows with more detail functions of how this process works.

Appendix 3: Design of Merchant



Merchant's influence in game

Colors in blue are strategies, white are thinking process of player, yellow are the source of sense of accomplishment, greens are explore behaviors that are good to provoke, red is the event that is the initiator. With the initiator, our strategy can alter from spend money to save money depending on the judgement of player himself/herself, and with the choice player made, it create large amount of chance to keep player immerse in the game loop.

Attribute	level formula/points	cost formula/gold
health	10*level+10	8*level+12
attack	1*level+1	8*level+12
defense	1*level+1	8*level+12

Formula for calculating price

Appendix 4: Enemy setting

Guards:



Killers:



Different shapes of enemies have different behaviors, different colors indicate different levels,notice that there are two e_1_2 in the second row, when this happens, it means these monster's are about the same level of dangerous to the player, so the system will randomly pick one according to a random seed given.Each types of enemy has different beginning status, that because different enemies have different behavior, some are melee and some are range ones, and some can move every turn and some have a moving gap.

1		maxHp	gold	exp	attack	defense	evade	target
2	Priest	10+12	1+2	1+3	10+8	0+4	10+2	80+2
3	Archer	10+12	3+2	1+3	5+8	0+4	10+2	80+2
4	Statue	8+12	1+2	1+3	2+8	0+4	5+2	80+2

Beginning status and growth of enemies