Implementing A Family Of Tower Defense Games

IMPLEMENTING A FAMILY OF TOWER DEFENSE GAMES

BY

YE LI, B.Eng.

A REPORT

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SOFTWARE ENGINEERING

Master of Software Engineering (2015) McMaster University

(Computing & Software Engineering) Hamilton, Ontario, Canada

TITLE: Implementing A Family Of Tower Defense Games

AUTHOR: Ye Li

B.Eng., Software Engineering

McMaster University, Hamilton, Ontario Canada

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: xi, 81

To my supervisor who gave me a chance to study at McMaster

To my parents who pay my tuition

To my fiancee who keeps company with me from a boy to a young man

Abstract

This project is about developing a family of Tower Defense games. It aims to implement a family of games that serves for studying the influence of different screen sizes and resolutions on players' performance. The games can fit various screen resolutions automatically, and the data of each experiment will be collected and saved to files. According to different selections of components in resource library, the system will generate different Tower Defense games. In other words, those games share the same mechanism but own different appearances.

The tool is developed with Unity. It adopts an efficient way in the period of implementation, including how to build Gird Map, how to spawn enemies by using XML files and Excel, and how to make the path for enemies to move along. It also solves some problems in game development like how to scale all elements and adjust them to different screen resolutions, and how to implement a generator to produce games.

Acknowledgements

I would like to thank my supervisor Dr. Jacques Carette for his expertise and guidance throughout the entire duration of this work. His knowledge and advice were essential in forming the motivation and the direction of this work. Our discussions with regards to this topic were invaluable. All of which, I am eternally thankful for.

I would also like to thank my family and friends for their encouragement and support. They continue to inspire me towards greater accomplishments.

Contents

List of Figures

Chapter 1

Objectives and Requirements

This project aims to develop a family of tower defense games. This report illustrates specifically how the project is designed and implemented. In addition, it demonstrates an efficient way in developing tower defense games by adopting software product engineering.

1.1 Objectives

This project is about developing an application with a family of tower defense games. Specifically, it requires all the games in this application share the same mechanism but own various appearances. The project is characterized with distinctive features: can be scaled to many resolutions, record players' performance and collect all related data.

The project has four main objectives, which are listed as follows:

- In the first place, the core goal of this project is to implement a family of tower defense games and turn them around. Each game is the fully functioning and

playable base for a tower defense game. It provides various types of towers and enemies with friendly graphics in order to increase the enjoyment of games. The games allow players to build, upgrade, and remove towers. The towers can attack enemies automatically and the players also can assign target for towers to fire.

- Secondly, we would like to create an experimental tool, which will help us to study and figure out how different sizes of screen scales influence players' performance. Recently, a variety of display scales and resolutions have been rapidly introduced, which leads game designers to rethink the design of user interface to suit different sizes of displays. Some games contain different sets of UI to fit various screen sizes, but they occupy too many system resources. While some games support a function that is able to scale automatically, but the effect on some displays may not let customers feel comfortable. Therefore, adapting the user interface in a flexible but efficient manner is a topic of great research [6]. Before finding the solutions, it is essential to know how different screen sizes influence on the performance of players.
- Thirdly, this project intends to be developed by adopting technologies of product family engineering. Product family engineering is a relatively new approach to the creation of software products which has achieved a broad recognition in the software industry. The main argument for introducing software product family engineering is to improve productivity by reusing components. However, there are not many cases of applying those techniques towards video games. A possible reason for this is due to the unique design of each interface for each game [5]. Tower defense games have some features that make it possible to

adopt product family engineering methods to make the development easier. For instance, each tower defense game is composed of relatively fixed elements such as towers, enemies, maps, game manager and other things. Although some elements like towers, enemies or maps have a variety of types, elements belonging to the same sort share a very similar mechanism, which provides an excellent environment to reuse the basic components.

- Finally, the project aims to offer a possibility to be a part of games in the future study of players' behaviours when they are playing strategy games. Thus far, a lot of work has been paid on intelligence video games, design and implement. Several techniques and methods of Artificial Intelligence and Data Mining have been applied to the analysis of players' behaviours analysis [4]. It is very interesting to find out how a player performs if he was under a nervous situation, and what strategy he would adopt.

1.2 Requirements

In order to achieve the objective listed in Chapter 1, the requirements of this project must be in keeping with main proposes. To ensure the output product of this project satisfies with what we expected, the requirements will be illustrated respectively.

Regarding as the general objectives, it requires developing a family of tower defense games, which has two meanings. On one hand, this project is not just implementing a TD game like those have already been released on AppStore, but creating many TD games. On the other hand, these TD games should be similar but own their variations as well.

In terms of gameplay, the games should have a very well defined and a small

set of tasks. This could bring fast pace of the game, giving players a hearty game experience. Moreover, in order to grab players' eyes at first glance, the style of user interface and graphics had better be simple and friendly. Also, like other successful TD games, the project comes with an extensive set of visual representations and offers several variants with various game elements like many tower types, different kinds of enemies, increasingly difficult maps and so on.

As for the aims of studying how different sizes of displays influence players' performance, all games can be scaled to fit various resolutions automatically. Since this project can be used as an experimental tool, data collection is an essential part. The data should include two parts: one saves the basic information such as the result of every level, the parameters of towers and enemies, and other elements; the other one stores the player's operation like building or removing a tower, targeting at an enemy. Apart from above, the spawning orders, grid nodes and paths should be prefixed, which ensure every tester receive the same difficult task.

A "game family" means the games of this project have different appearances but share the same mechanism. This provides conditions for the project to adopt software product family engineering technology. So the elements of a game can be reused in other games. To realize that, there requires a game generator to be implemented, which can generate TD games with the same mechanism. In this project, game elements like towers and enemies in each game not only have the same number of types but the character of each type in a game also exits in the other games. Besides, the difficulty of games also keeps the same, which means the parameters of game elements, spawning orders and the design of maps are the same.

For the future strategy research, the game balance should be set neither too hard

nor too easy. It is best to achieve the effect that players never stop interacting with the game until it is over, and it really has some opportunities to get through the game if players adopt some good strategies.

In conclusion, this application satisfies the following features:

- A game has a very well defined, and a small set of tasks.
- The project comes with an extensive set of visual representations and offers several variants with different number of UI elements, different placement of items, different renderings of the elements, etc.
- All elements can be scaled to fit different screen sizes automatically.
- The application can be experimented with, so it should have data collection and replay function.
- Implement a generator to produce TD games that look differently but have the same mechanism.
- The games are in a good balance, which are neither too easy nor too hard.

Chapter 2

Prerequisite Knowledge

2.1 Tower Defense Games

2.1.1 Concept

Tower Defense (TD) is a sort of Real-Time Strategy (RTS) game, which is focusing on placing static defensive units at strategic locations to stop enemies from moving across a map. Traditionally, a Tower Defense game includes a player purchasing and organizing defensive towers that fire on a troop of different types of enemies [16].

The strength of enemies grows with every wave so the difficulty is ever increasing. The player has several types of towers available, each with various attributes and characters. Every time the player defeats an enemy, the player gains money as reward. The player can use money to build other defenders or to upgrade exiting defenders in order to make them more powerful. A game normally has winning conditions, like surviving certain number of waves, or it may be endless and last until the player loses by letting enough enemies reach to the ending point [12].

Compared with other genres of RTS, TD games are more intermittent. A player's success is not just defined by the amount of actions taken per second or per minute (as seen on *Starcraft2* and *Warcraft3*), but depends heavily on the order of actions taken as well. Although the speed of actions may influence the result of a TD match, it is not the decisive factor of winning or losing.

Figure 2.1: Starcraft2

Mechanics

Generally, tower defense games share similar base mechanics, which is listed as follows:

- There is a map with at least one entrance and one exit or one target. Enemies will come in through the entrance and disappear through the exit.

- The goal of the enemies is to arrive at the exit, and the mission of the player is to stop enemies from reaching the exit.
- The player puts towers on the map to destroy the enemies.
- There should always exist at least one path that allows enemies to reach the exit.
- When an enemy reaches the exit the player loses one or more lives.
- The game is lost when the player has no lives left.
- Enemies usually come in turns or waves with a large troop.
- The game is won when there are no more waves.
- Towers can be upgraded and removed.
- Building towers and upgrading them costs money. Removing towers gets money back but less then the price purchased when they are built [3].

2.1.2 Existing Variations

The TD genre is a fairly recent one, gaining popularity particularly due to the games on mobile devices available for casual gamers [15]. There are a large number of popular TD games. In order to gain players' attention under a competitive market, game designers have created many variations on original games.

With the development of TD games, a set of features has been displayed. These features can be combined to make up the elements of a TD game. Through analysis, TD games can be categorized according to the major game elements.

1. Map: The map of TD games restricts how players allocate defenders. It can be split into two forms. The first one requires players to place towers in specific locations. The map is dominated by a linear path with surrounding area for

building towers. The path is fixed and remains unchanged in every level [19]. Commonly, different levels can display more challenging paths. One example of this is *Carrot Fantasy*, which is typical of one genre of TD games. The game itself is simple, maps are easy to understand, and the graphics are pleasing yet uncomplicated. The complexity of this game comes from completing the levels with increasingly harder deployment of enemies and unlocking new and more interesting towers.

Figure 2.2: Carrot Fantasy

One of the most popular multi-platform TD games is the *Plants vs. Zombies* created by Popcap Games. This game consists of several parallel lanes where different towers (plants) can be built to defend against the enemies (zombies).

The enemies also follow the same lanes, and the towers should be put in one of the many squares on the grid. Apart from being attacked, the enemies can destroy the towers to get to the end position.

Figure 2.3: Plants vs. Zombies

Other popular games follow this form of linear and branch paths including *Bloons TD*, *Gem Craft*, and many others. While many games include the map with linear and branching paths, other games are more flexible. The paths in those games are created by the placement of towers, and enemies have to go around them. The classic example of this genre is the *Fieldrunners*, where the entirety of the space is allotted for players to place towers. Enemies at any

instance attempt to select the shortest path towards the exit, as constrained by the towers. Lengthening the time it takes enemies to reach their destinations will increase the chance to eliminate enemies. However, that does not mean the longer path the better, because some enemies can fly or jump over towers to avoid being attacked.

Figure 2.4: Fieldrunners

2. Towers: Most games specialize in providing a variety of different types of Towers such as *Carrot Fantasy*, *Fieldrunners*, *Plants vs. Zombies* and many other games. Different towers have their special capabilities to deal with specific sorts of enemies. Normally, the strength of a tower depends on its power, range, and attack speed. The stronger a tower is the higher price it would cost.

For example, the simple towers fire a single linear attack at a moderate speed. The advanced towers have a larger range and higher power, but the cost of it is relatively expensive. Some towers attacks multiple creeps at once while some towers can both damage and slow enemies.

Many recent games allow players to upgrade towers. Upgrading a tower costs resources as well, which may cost more than build some low-level towers to reach the same effect. However, in some maps, there is not enough room to put as many towers as the player wants, so upgrading some towers at this situation is a wise choice. Both *Carrot Fantasy* and *Fieldrunners* allow players to upgrade defenders. Compared with two TD games mentioned above, *Plants vs. Zombies* does not provide the upgrading function in the match. But it provides much wider sorts of defenders for players to select. Before a match, the game asks players to choose certain number of defenders from a plant book. Some plants are strengthen version of simple plants and they cost more sunshine.

In addition to placing towers on the map at preset locations, other games like *Kingdom Rush* provide a hero where you can choose a particular type of movable unit that has varying capabilities. This unit works as a mobile defensive unit and requires players direct to a particular task, including directing them to attack a particular creep [3].

Figure 2.5: Kingdom Rush

3. Enemies: The types of towers influence the types of enemies directly. Enemies have different abilities usually consisting of speed, health. Slow creeps are normally harder to kill, while fast creeps have small health. As for traditional TD games, enemies are always bearing attacks from towers. Unlike these games, some games like *Plants vs. Zombies* allow Zombies to eat plants. In order to reach the house, zombies have to eat up all plants in front of them. Some zombies can even jump over plants.

4. Single or Multi-player: Most of the TD games are single player games, with a

human player competing against a static set of creeps. Recently, a new multiplayer genre of TD games called tower wars games is very heated on mobile platforms. These games require the player to send out enemies to their opponents' game boards respectively their controlled areas at a common game board. Among them, *Clash of Clans* is the most successful one. It is an online multiplayer game in which players build a community, train troops, and attack other players to earn gold and elixir, which can be used to build towers that protect the player from other players' attacks, and to train and upgrade troops [9]. It gives players the most freedom to determine the towers, enemies, path, and strategy.

Figure 2.6: Clash of Clans

2.1.3 Adopt TD Genres

TD games have proved to be a challenging, addictive and fun way to kill time. They provide a large user-base with a simple game mechanism, and a range of interesting potential research possibilities. The simplicity of gameplay makes TD games a great test bed for interaction research. Many popular RTS games are complex in nature and take a long time for players to get familiar with them, which often create a less enjoyable game experience. In terms of development, TD games are relatively easy to program and the requirement of graphics is simple.

This project chooses to develop a genre of TD games like *Fantasy Carrot* because

this sort provides an excellent field to study interaction research. Compared with other TD games, these TD games heavily depend on selection tasks. Regardless of the underlying device and its input modality, the game just requires players interacting with HUD by selecting elements. For example, the players need to click the positions to build towers, choose the types of tower, assign which tower to be upgraded, and many other click actions. Moreover, using predefined paths in TD games greatly improves performance. The testers need to place tower at specific grids, which is easy to test the time and accuracy they complete interaction tasks. In addition, adopting the fixed enemy spawning order keeps the experimental results stable. This eliminates the influence of fortune from testers and help researchers to get more precise data.

2.2 Software Product Families

Software product families have been widely recognised in the software industry. Some organizations have adopted this technology and got satisfied achievements. Some companies are considering to adopt it. It focuses on the development, evolution and reuse of components. Basically, product family engineering is all about reusing components and structures as much as possible. When a component is completed, it can be used in several products or systems, which is one of the main benefits to be achieved [2].

Before a product family can be successfully established, an extensive process has to be followed. This process is known as product family engineering [10]. Product family engineering (PFE) is a synonym for "domain engineering", which can be defined as a method that creates an underlying architecture of an organization's product platform. It provides an architecture based on commonality and planned variabilities

[7]. Different product variants can be generated from the basic product family making it possible to reuse and differentiate on products of the family.

The main proposes for adopting software product family engineering are to: increase productivity, improve predictability, decrease for time to market, and reduce labour costs [21]. A fundamental reason for using technologies of product families is to minimize the proportion of application engineering. As Figure 2.7 shows, with the requirement of technology becomes more and more sophisticated, adopting domain engineering (the same with PFE) has more significant benefit to reduce the efforts of application engineering [18].

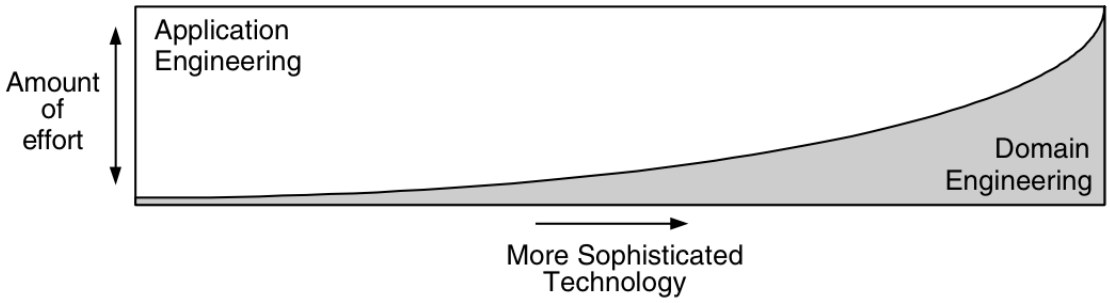


Figure 2.7: Domain vs. application engineering

The process of product family engineering includes three main phases: product management, domain engineering, and product engineering.

Product management is the starting up of the whole process. In this process, some concepts and terms are defined with economic aspects. In other words, this phase is responsible for making market strategies and defining a scope.

The goal of domain engineering phase is to establish a reusable platform. When this phrase is over, it provides a set of common and variable requirements for all

products.

Product engineering is the last phase, which a product X is being engineered. The product X is being derived from the platform established in the domain engineering phase. It satisfies all common requirements and owns its unique variable characters. After being fully tested and approved, the product can be delivered [14].

Chapter 3

Software Design

This chapter is focusing on using the methods of software product engineering to design this project, which includes designing architecture, use cases and components.

3.1 Project Architecture

The architecture of software product families has significant differences from architecture in single application development [13]. Detecting, designing and modelling the variable and common parts are important in software family engineering [1].

Just like other Unity projects, which are composed of scenes, this project consists of five scenes, "MainMenu", "Settings", "LevelSelect", "Replay", and several "Level" scenes. Figure 3.1 shows the workflow and relationship of the scenes mentioned above.

The application starts from "MainMenu" scene which offers options for users. "Settings" button allows users to set resolution, music and sound. After confirmation made, the application will return to "MainMenu" scene. Before playing, the users

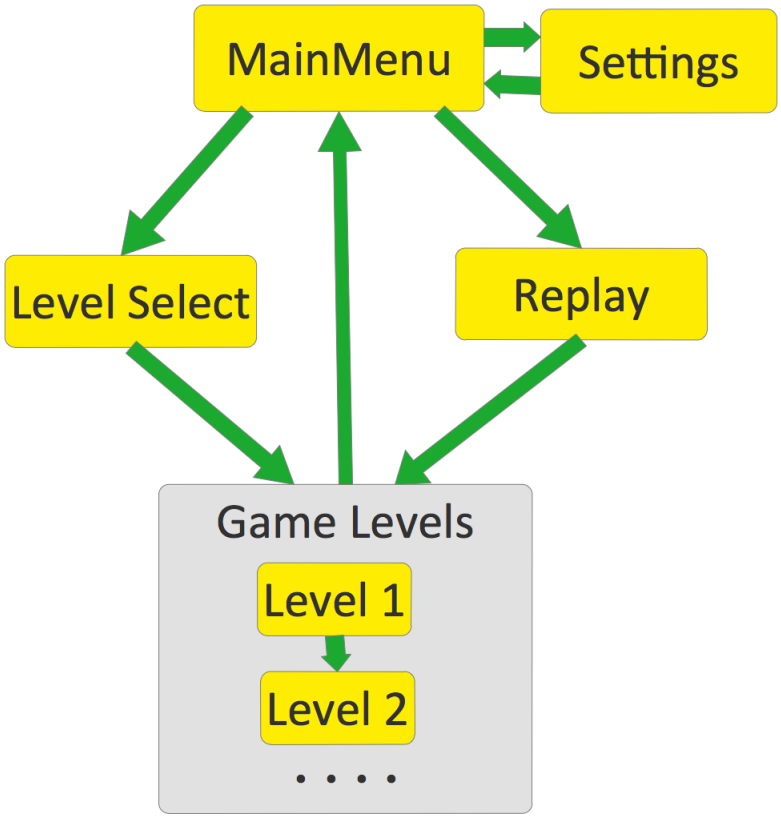


Figure 3.1: The main composition of project

have to assign towers, enemies and themes to create a game at "MainMenu" scene. Then click "Play" button, and the application will jump to "LevelSelect" scene where users could choose which level to play. After selecting a level, the user will enter into that level scene. These level scenes are the core parts of this application where players play tower defense games. When players defeat a level, they could choose to play the next level or return to "MainMenu". Also, this project provides a service to watch former play records. Users could do this by simply clicking a record at "Replay" scene. After the chosen record finished, players will return to "MainMenu" scene.

3.2 User Operations

Users can interact with scenes and take several different kinds of operations. There are three sets of users in this project: Designers of levels, Players, and Experimenters. These three sets of users have different responsibilities and take different operations. The use cases of each user set will be illustrated respectively.

3.2.1 Designers' Use Cases:

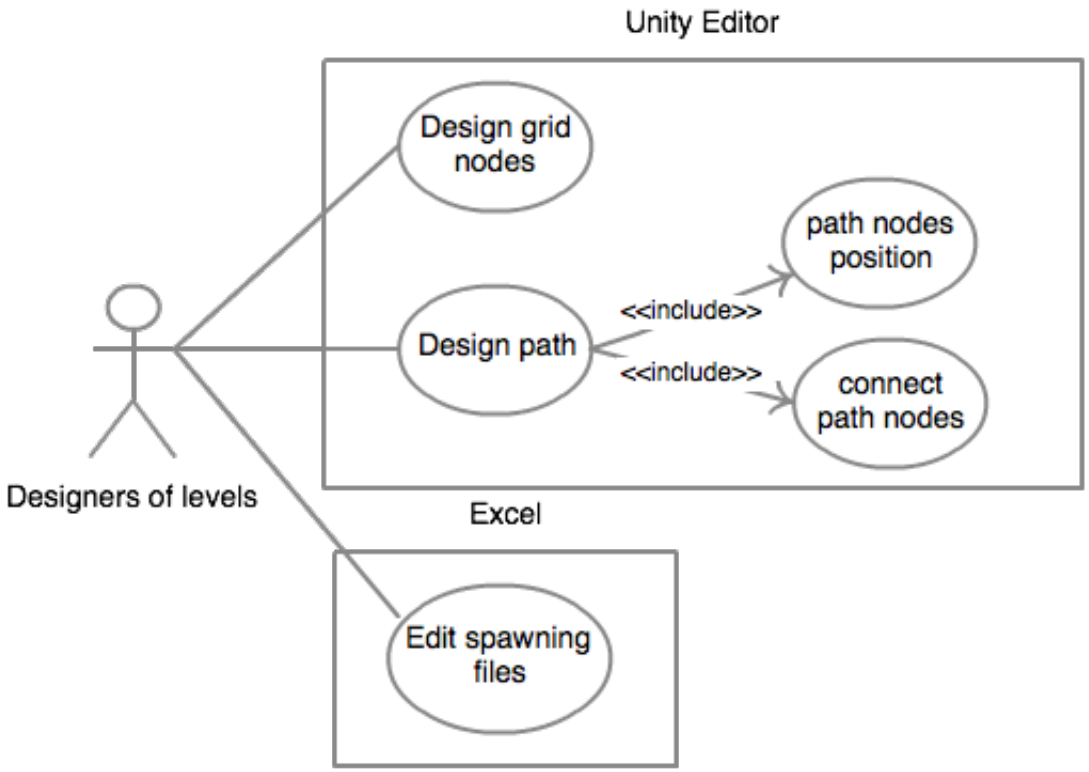


Figure 3.2: Designers' Use Cases

As shown in the figure above, a designer has to design a level in Unity Editor and Excel software. In process of designing grid nodes, the designer first create a grid node and set its attribute. The node attribute identify whether players could place towers at its position. Then the designer drags the node to the grid. Designers also can design a path, which includes placing path nodes at desired positions and connecting them in order. Enemies are spawned according to spawning files. Designers edit these files to determine what kind of enemy to spawn, how many of them will appear in a wave and how often an enemy in spawned. Since the data in each file is relatively big, it is convenient for designs to edit these files through Excel.

3.2.2 Players' Use Cases:

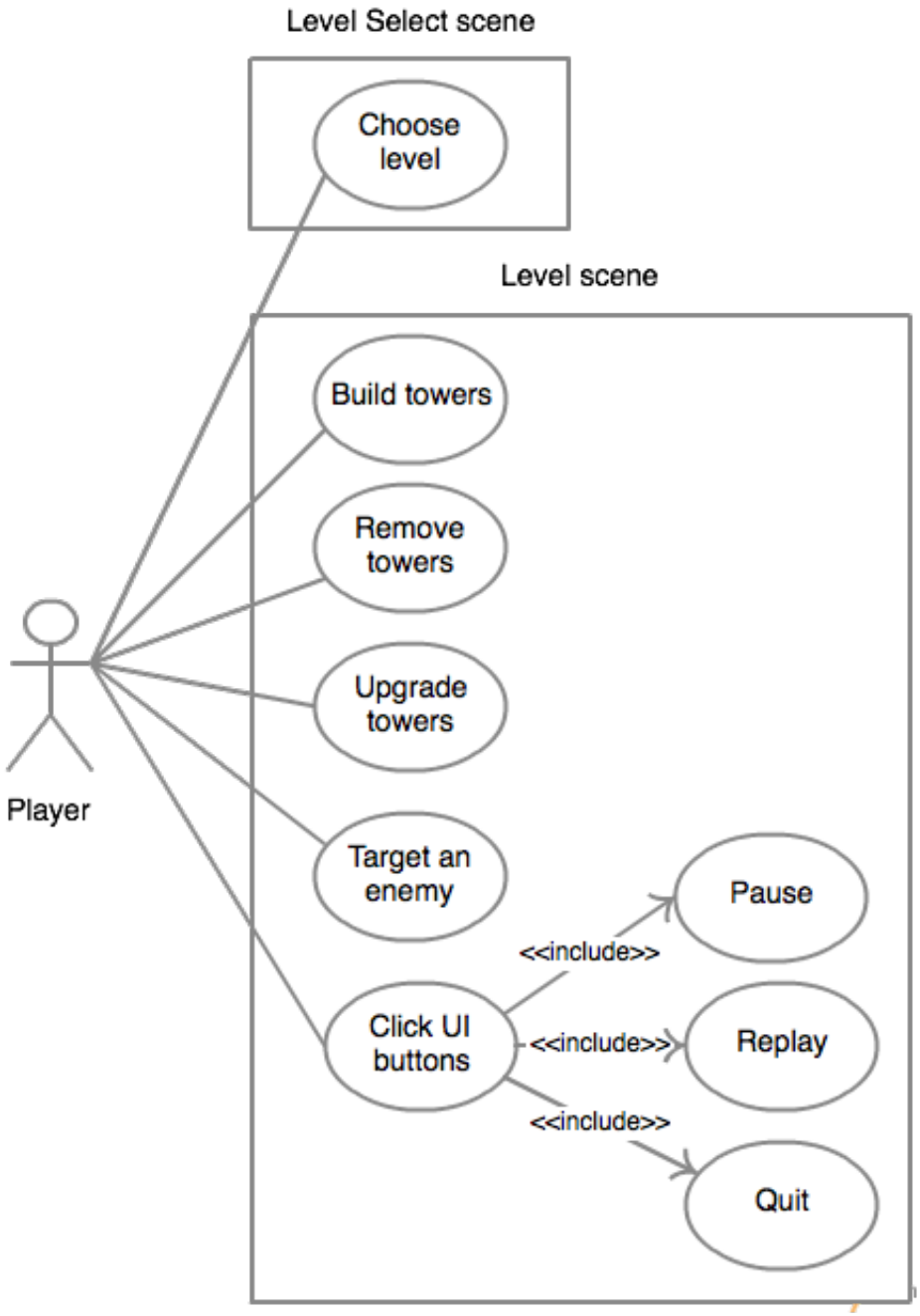


Figure 3.3: Players' Use Cases

Players choose a game level in “LevelSelect” scene and then the system jump into that level scene. In level scenes, players have many interacting operations with the game. A set of tower buttons show up when players clicks an empty grid on map. Then they could determine which tower to build at that position by clicking the according tower button. When a tower is selected, players will see its operations. Then they could upgrade or remove this tower. Players can also let defenders fire on an enemy by just click on it. The game provides some buttons for players to control the game process. Clicking the relative buttons, players can pause the game, play from the beginning, or quit to “MainMenu” scene.

3.2.3 Experimenters' Use Cases:

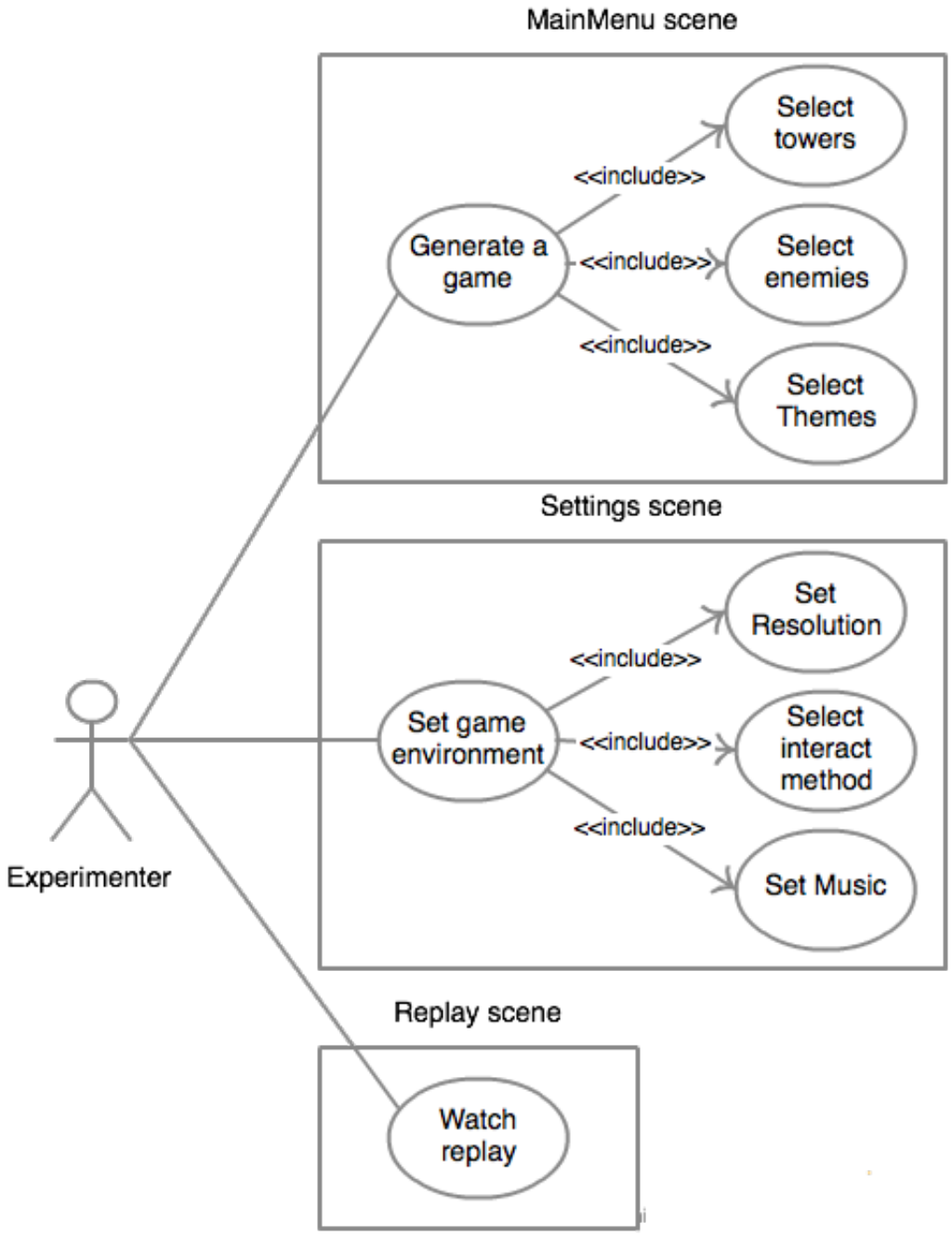


Figure 3.4: Experimenters' User Cases

Before conducting an experiment, experimenters usually need to set game environment and select game elements to generate a TD game. To set game environment, experimenters click settings button in "MainMenu" to enter into the "Settings" scene. In there, experimenters can set game resolution by choosing one from the list. For example, an experimenter select a resolution of 1024×768, the project will scale to that size. Experimenters also can select interact method and the state of music just by selecting the corresponding buttons. To generate a TD game, experimenters have to select game elements in "MainMenu". For example, Experimenter A determines to select the first tower set, the second enemy set, and the third theme set. After clicking the play button, the system generates a TD game according to his choice. If researchers would like to analyse players' performance, they could enter into Replay scene where all replays are listed. Choosing one from the list and then pressing the play button, then experimenters will watch how a player was playing at that time.

3.3 Components of Game

Level scenes are the core of this project. Each level scene includes the following components: Settings, Recorder, Game Manager, Loading Game, Enemy Spawner, Grid Map, Path, Background Controller, Theme, and UIs [17]. The relationships between these components will be illustrated in 3.5.

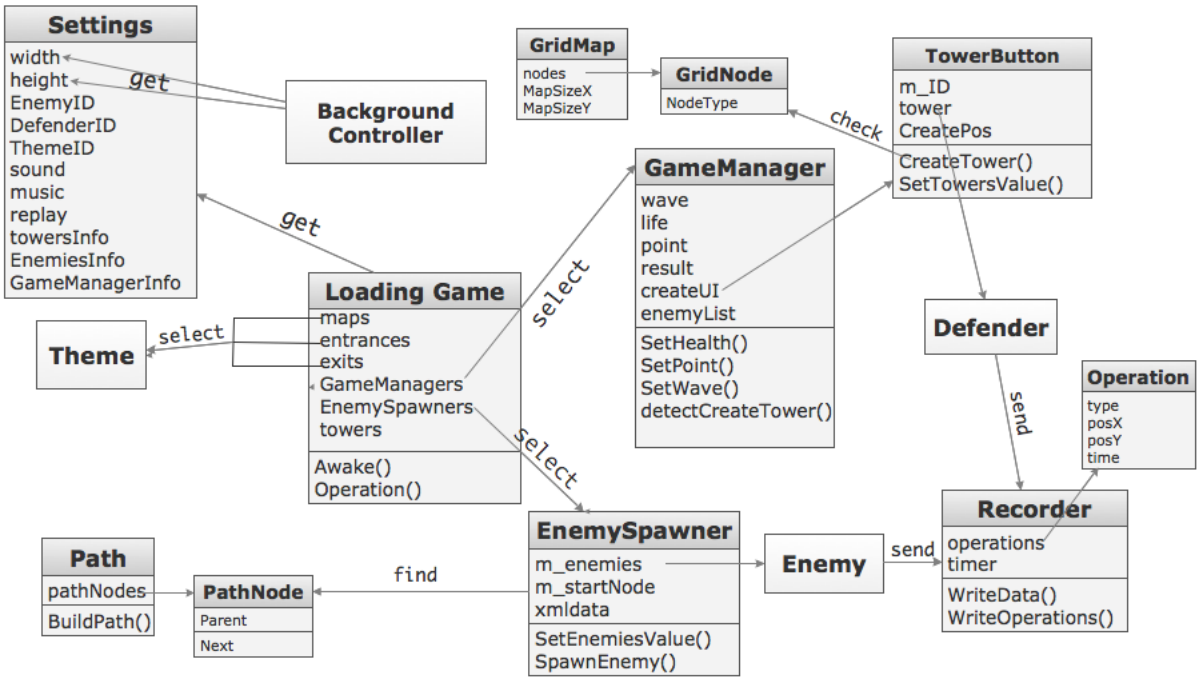


Figure 3.5: Game Components

• Settings: It stores the basic information of the game. When the application is started, "Settings" component loads "sound" and "music" values from the preference list and gets the parameters of towers, enemies and game manager from a text file named "Settings.txt". Other variables record system's current status.

• Background Controller: It gets the screen width and height from "Settings" component and scales all elements in order to adjust to the current screen resolution.

• Loading Game: Before a game starts, the system should have already prepared all resources for the game. In "Awake()" function, "LoadingGame" component gets data from "Settings". According to variable "Theme ID", "LoadingGame" component selects theme prefabs like entrance, exit, and map. Similarly, it chooses "GameManager" and "EnemySpawner" components according to variables "DefenderID"

and “EnemyID” in “Settings” component. Apart from these, “LoadingGame” gets the value of the boolean variable “replay” to judge whether it needs to load and run a record’s operations.

• Game Manager: It saves the latest data of the game such as the wave number, life, point(also known as money), and result(the final score). Moreover, “GameManager” also has the responsibility to judge whether the game is over depending on variables “life” and “enemyList”. Additionally, “GameManager” keeps on checking creating tower operations and instantiates tower buttons if the player selects a proper place to build a tower.

• Tower Button: It allows players to build wanted towers by clicking the corresponding button. Before creating a tower, its script first sets the values for this tower according to “towersInfo” in “Settings” Component. The tower will be constructed at the position where users click on the ground. When the tower is created, a message will be sent to “Recorder”.

• Enemy Spawner: It is in charge of spawning enemies according to an XML file. The array “m_enemies” stores all enemy prefabs. When “EnemySpawner” is created, it will find start node in “Path” component and assign the node to the enemies as starting path node.

• Path: This component consists of several “PathNodes”, which are well-arranged to define the path of enemies. Therefore, an enemy runs towards one path node to another and finally reaches the end. Each path node stores its parent and next node.

• Grid Map: Like “Path” component, “GridMap” comes along with dozens of “GridNodes”, which are used to mark the status of each small square area. That is to judge whether a tower could be built on this cell or not. Some certain area on the

road cannot have any towers standing while some cells that have already been placed a tower cannot be placed another tower, either.

• Recorder: "Recorder" is used to collect a game's data and players' operations. It gets a game's information from "GameManager" and "EnemySpawner". "Recorder" also receives messages when operations are taken and saves them into an array named "operations". "Operations" stores all actions in a game like building, selling, and upgrading a tower.

Chapter 4

Game Design

This chapter is to talk about the design of a single tower defense game. As mentioned before, a TD game is composed of several game elements such towers, enemies, game manager, and others. The main purpose of game design is to make the games more interesting, so game balance is also an inconvenient factor to be well defined.

4.1 Tower Design

Considering the playability of games and referring to some heated tower defense games like TNT (Towers N' Trolls), Ace Defense, and Toy Defense, the application offers five unique tower types to construct. In the process of designing towers, we need to be aware of the following points.

Firstly, these five sorts of towers must own their unique characters. For example, there may be a sort of tower whose attack speed is slow but its power is high, or a type of tower that can slow down enemies. Moreover, these tower types should not have any possibilities to replace each other. In other words, the game require players

to make fully use of all type of towers rather than just adopt only one type to get through.

Secondly, all towers, regardless of their types, share the same module. Specifically, every tower has five attributes: type, power, attack time, attack area, and cost. "Type" determines the unique character of towers. Attributes "power", "attackTime", "attackArea" determines the efficiency of a tower. To keep different tower types have a similar cost performance, the price of a tower depends on its strength.

Thirdly, a tower could be upgraded and sold. It has four levels: low, medium, high, and super. Upgrading a tower will cost more money and selling a tower will get the money less than its original price. The purpose of making towers can be upgraded is to improve the game's strategy. The cost of upgrading a tower is much higher than that of creating a few low-level towers to reach the same effect. However, in many situations, building a large number of low-level towers is impractical because there is not enough room to do that. So players have to upgrade some towers to win the game.

Here is the description of five tower types:

• Light Tower: It is a basic defender in the game, with lower power, medium attack area, and medium attack speed. Light Tower has the highest cost-performance.

• Heavy Tower: Its power is the strongest, but its attack speed and attack area are little. Heavy Tower is more efficient to deal with hard enemies.

• Ice Tower: Ice tower can slow down enemies in a short period, but its power is the lowest of all. Combining it with other towers may cause surprising effects.

• Poison Tower: Poison Tower hurts enemies every second during a short period. Although the damage in every second is small, the total harm is larger than Light

Tower's. Poison Towers would be more efficient if they were placed near the beginning point.

• Electric Tower: It is the strongest and the most expensive tower. The tower attacks enemies constantly and its power is lower than Heavy Tower but higher than Poison Tower. The attack area of this type tower is the largest of all.

4.2 Enemy Design

In order to make game's difficulty increase gradually, various types of enemies are necessary to appear. We should be aware that there are three factors influencing the strength of every enemy: health, speed, and harm. In addition, if an enemy is killed, the player will get some money as a reward. The amount of money depends on the degree of enemy's strength. Referring to those famous tower defense games, this project divided enemies into 11 types.

• Easiest: This kind of enemies are the easiest to kill and have a decent reward when killed by defenders. They are designed as a welfare for players to add flexibilities of the game.

• Easier: Travel in large packs, low health, and medium speed.

• Medium: Travel in medium packs, slower speed, and moderate health.

• Fast: Travel fast in large packs, moderate health.

• Faster: Compared with fast enemies, faster enemies move with a higher speed.

• Fastest: Swift moving, the fastest of all enemy types, low health.

• Harder: Travel with small packs, high health, and medium speed.

• Large Health: A real formidable foe, extremely high health.

• Hardest: A terrible enemy that not only stronger than "Large Health" but moves

faster.

• Revive: Travel in a frightful army, can revive when it is killed at first time, higher health and medium speed.

• Boss: A boss is the strongest enemy in the game. It has the greatest health, moves faster, and causes the biggest damage on player's life when it arrives at the ending.

4.3 Game Manager Design

Game Manager controls a player's status including life, money, score and wave. When a game begins, initially, the player will get some money. Game manager is in charging of updating the status of life, money, score and wave. For instance, if an enemy reaches the end, game manager will reduce the player's life; if an enemy is killed, game manager will update money and score; if the player has built or sold a tower, game manager will update money.

4.4 Game Balance

Game balance is key to a game's quality. A game with a good balance will make players addicted into playing it. Otherwise, players would lose their interests because they think the game is too easy, or too hard, or too dull. In this project, the parameters of towers, enemies, and game manager influence the game balance.

4.4.1 Build A Module

First of all, it is necessary to make it clear what determines the strength of a tower and an enemy. For a tower, power, attack area, and attack speed are the main factors. For an enemy, health and moving speed influence its strength.

Then, in an ideal situation, suppose an enemy is moving under attack by some towers. In order to kill the enemy, we get formula 4.1:

$$HP = \frac{ATK1 \times FRQ1 \times ELP1}{Speed} + \frac{ATK2 \times FRQ2 \times ELP2}{Speed} + \cdots \tag{4.1}$$

HP stands for the health of that enemy, ATK stands for a tower's power, FRQ stands for a tower's attack frequency, and ELP stands for the effective length of the road. From formula 1, it is easy to transform and get:

$$HP \times Speed = FRQ1 \times ATK1 \times ELP1 + FRQ2 \times ATK2 \times ELP2 + \cdots \tag{4.2}$$

Introducing a variable SE to stand for the strength of an enemy and a variable ST to stands for the strength of a tower, we get:

For an enemy:

$$SE = HP \times Speed \tag{4.3}$$

For a tower:

$$ST = FRQ \times ATK \times ELP \approx FRQ \times ATK \times Radius \times \sqrt{2} \tag{4.4}$$

Assuming a large number of enemies are spawned at the same time, if they belong

to the same type, then the difficulty of the game can be expressed as:

$$Difficulty = SE \times (Num - \frac{Life}{Harm}) \tag{4.5}$$

Therefore, in order to survive, a player should be given a chance to satisfy:

$$Difficulty < ST1 + ST2 + ST3 + \cdots \tag{4.6}$$

$$Cost1 + Cost2 + Cost3 + \cdots \leq Money \tag{4.7}$$

4.4.2 Set Parameters

Parameters of enemies need to be set first. Then, build on this, set parameters for towers. Setting parameters for towers is more complicated than others. During this process, we should pay attention to two things.

As different tower types own their unique characters, the strength of each tower needs to be calculated separately according to its character. For example, ice towers can slow down enemies, so this feature should be taken into consideration as well.

In addition, towers with different types should have a similar cost-performance. That means the designer should avoid the situation that one type is so useless that can be replaced by using other types entirely.

4.4.3 Adjust game balance

Making a game in an excellent balance is a hard and complicated work. It is impossible to avoid errors when facing with various situations. In order to make up potential errors, we could increase initial money for players properly. Also, some values need

to be modified during practical test period.

Chapter 5

Implementation

This chapter will talk about the implementation via Unity, which will be mainly divided into four parts: a tower defense game, game generator, data collecting, and replay.

5.1 Development Environment

This project is developed with Unity4.5 and the scripts are mainly written in C# with an IDE called MonoDevelop. Unity is a game engine that is fully integrated with a complete set of intuitive tools and rapid work flows to create interactive 3D and 2D content like images, sounds, and physical effects. Since first released in June 2005, Unity has developed into a powerful game development ecosystem. It smashes the time and cost barriers for independent developers and studios. Now Unity is easy multiplatform publishing and widely used in creating mobile and web applications [20].

This project chooses to use Unity because it offers a convenient 2D workflow,

which makes importing sprites as simple as dragging and dropping them into the relevant folder. Moreover, Unity can slice sprite sheet automatically, and manual slicing is easy by just clicking and dragging the mouse over the desired area.

5.2 Develop A Tower Defense Game

5.2.1 Background Controller

Background Controller is used to scale all game elements to fit current resolution. It is realized as a C# script and added as a component of "Game" object. In every level scene, we put all other objects under "Game" object, so it becomes the governor of sizes. That is to say whenever its size changes, Background Controller will scale its children to fit its new size.

In Unity, sprites will be automatically scaled when resolution changes. However, the ratio between width and height is never changed. In other words, when length-width ratio of resolution changed, the game elements would not be scaled properly. As Figure 5.1 shows, the size of the blue rectangle is the default size of this project, and the size of the yellow rectangle is current screen size. However, the size of the green one is the actual size after scaling.

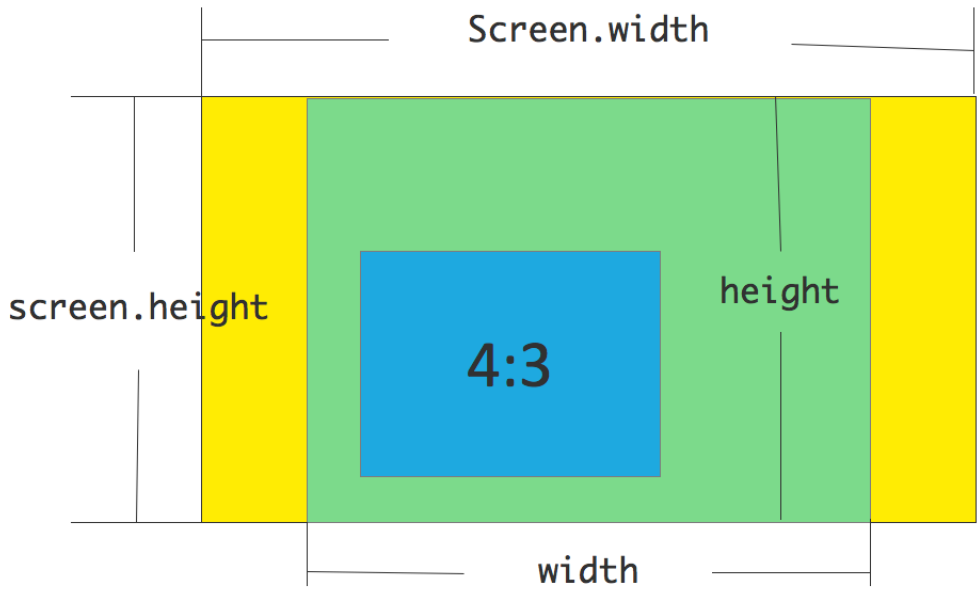


Figure 5.1: The new size after being scaled

Obviously, the width of the green rectangle needs to be stretched. To do this, we shall first get the rate between screen's and game's width:

$$rate = \frac{screen.width}{width} \tag{5.1}$$

Since $\frac{width}{height} = \frac{4}{3}$ and $height = screen.height$ We get:

$$width = \frac{4}{3} \times height == \frac{4}{3} \times screen.height \tag{5.2}$$

Then the rate can be calculated from the value of resolution:

$$rate = \frac{screen.width}{screen.height} \times \frac{3}{4} \tag{5.3}$$

Let the property "transform.localscale.x" of "Game" object multiply with value "rate", then all its children are scaled to fit the new resolution perfectly. In this way, it reduces a lot of work on considering how to scale every single element separately.

5.2.2 Grid Map

Grid Map, like a checkerboard, can be placed defenders and a road for enemies. It is composed of two-dimensional cells. Each cell may have different usages. Some are used for putting defenders; some are used for enemies to pass; others neither allows players to build towers nor enemies to pass.

A tool to build grid map

We are engaged to find a tool which makes it possible for developers to build grid map in Editor. A tool with graphic interface brings not a few benefits. On one hand, it transfers the code into visible images, which make it possible for developers to see their thoughts directly and concentrate on designing works without considering how to writing commands. On the other hand, it is convenient for modifying their design in the future instead of changing the code. As a whole, using a tool in developing games improves efficiency and productivity.

In this project, Grid Map is implemented by using a two-dimensional array to represent it. Every element of the array represents a cell named Grid Node. According to real situation, each Grid Node only has two conditions, can put a defender or can not put a defender.

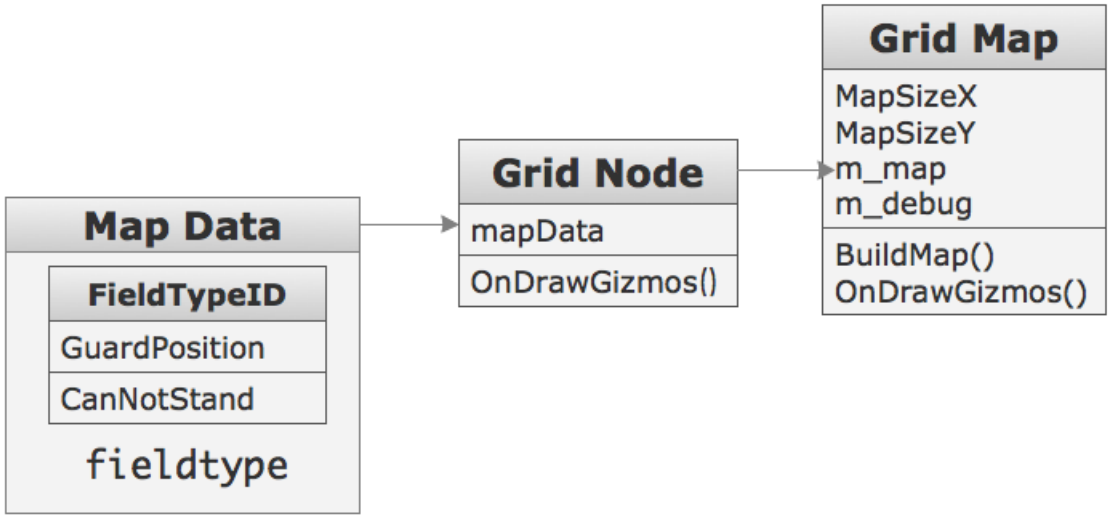


Figure 5.2: GridMap class and GrideNode class

As Figure5.2 shows, in class "MapData" the script defines an attribute named "field type", which is used to mark the status of cells. "FieldTypeID" is an enumeration offering two conditions of a cell. "GuardPosition" means can put a defender and "CanNotStand" means cannot put a defender.

Class "GridNode" only has one "MapData" attribute to save its status and use OnDrawGizmos() to draw its icon in Editor. In class "GridMap", "m_map" is a two-dimensional array storing the map's information and "m_debug" controls showing information in Editor. In BuildMap() function, it first initializes "m_map" and sets all cells' state as "GuardPosition", then the function finds all grid nodes with status "CanNotStand" and finally changes the value of corresponding cells in m_map. OnDrawGizmos() draws cells with blue lines and paints those cells whose status is "CanNotStand" with red color.[8]

Build grid map in Editor

First, we create a game object named "GridMap" in Hierarchy tab, and then drag "GridMap.cs" as a script component on it. Click on debug attribute and run BuildMap() function, we will see blue lines divide the map into many cells.

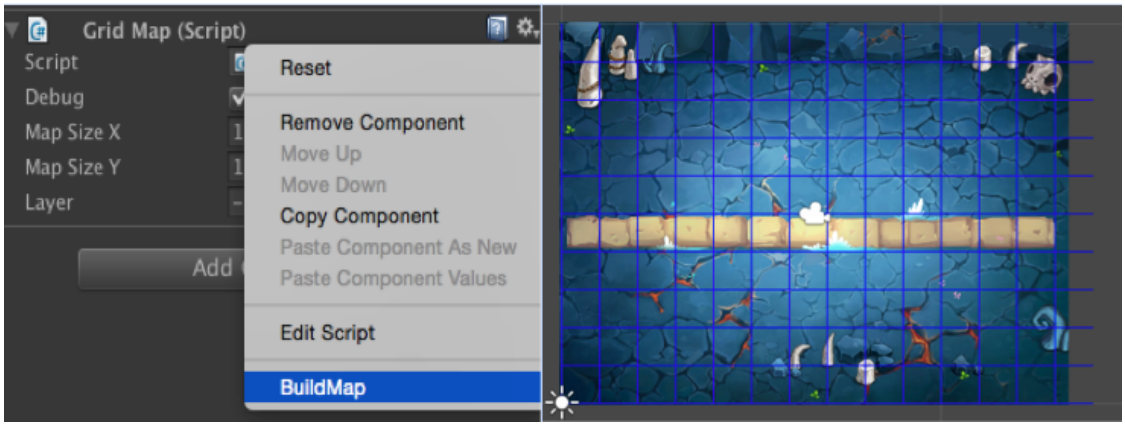


Figure 5.3: Initialize grid map

Second, we design the grid map by creating some game objects named "gridnode", setting their status as "CanNotStand", and moving them on some cells.

Next, run BuildMap() function again in order to update the map information into array "m_map". The cells that cannot be placed a defender will turn red.

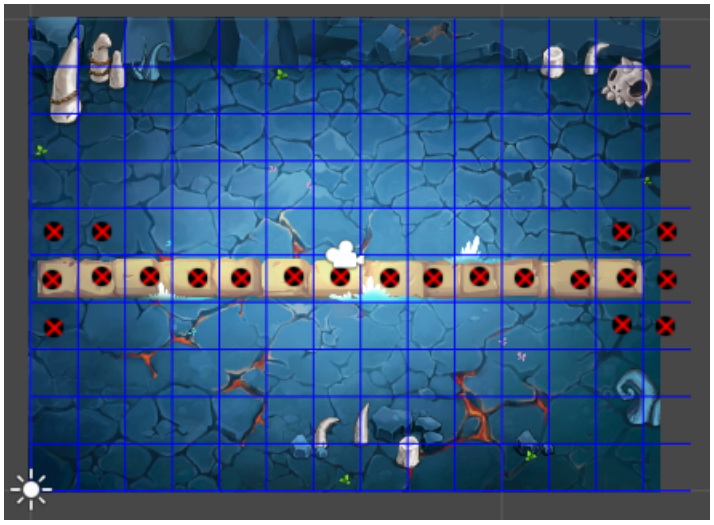


Figure 5.4: Assign unavailable nodes

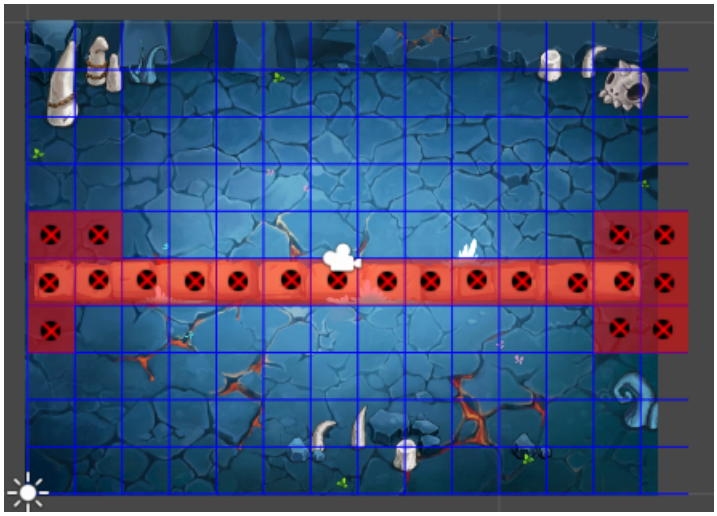


Figure 5.5: Build grid map

5.2.3 Path

In tower defense games, enemies run along with a preset road. A path is made up with many path nodes. An enemy runs from one path node to another path node until it arrives at the ending.

Make a path tool

Like the procedure in implementing Grid Map, we also would like to use a tool to build a path for enemies.

First, we define a "PathNode" class. Each path node has two attributes "parent" and "next". OnDrawGizmos() function shows its icon in Editor.

Next, we write a class named "PathTool" to assign values for every path node. There are two functions in PathTool, SetParent() and SetNextChild(). The two functions connect path nodes as

Finally, we define a class named "Path" and write BuildPath() and OnDrawGizmos() for it. BuildPath() function puts path nodes into the array "pathnodes" in order, and OnDrawGizmos() function draws lines to connect all path nodes together.

Build Path

To begin with, create path nodes and adding "PathNode.cs" as script component. Then, put script "PathTool.cs" into folder "Editor", which allows us to run the script in Editor and it creates a menu on Toolbar. The next work is to set "parent" and "next" attributes for each path node. At last, run BuildPath() function to store all path nodes into an array and draw the final effect. [8]

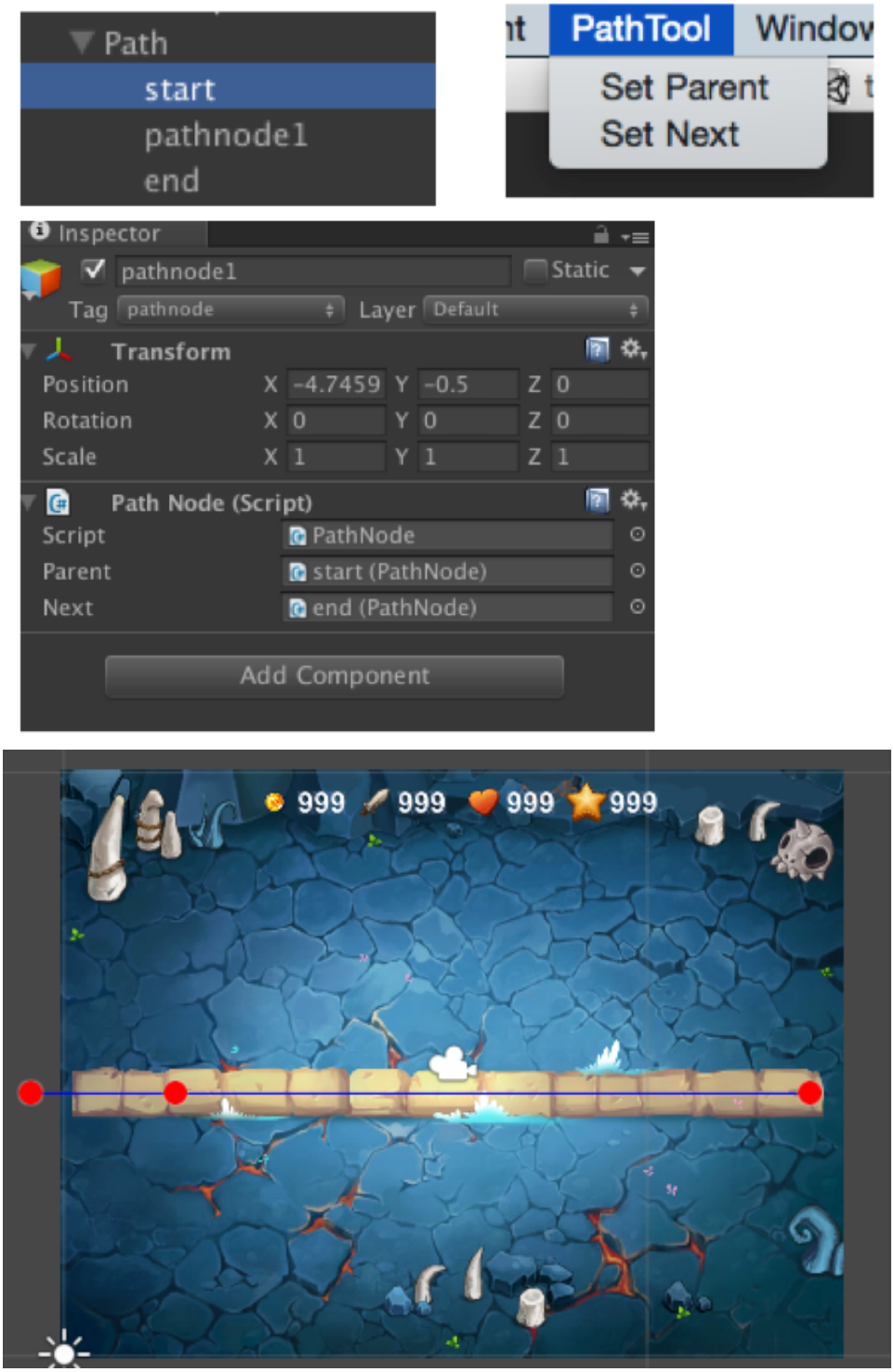


Figure 5.6: Build path

5.2.4 Game Manager

The next we will implement a game manager. This component is in charge of updating the latest game status and showing it on screen. Also, it has the responsibilities to detect the player's operation like building tower.

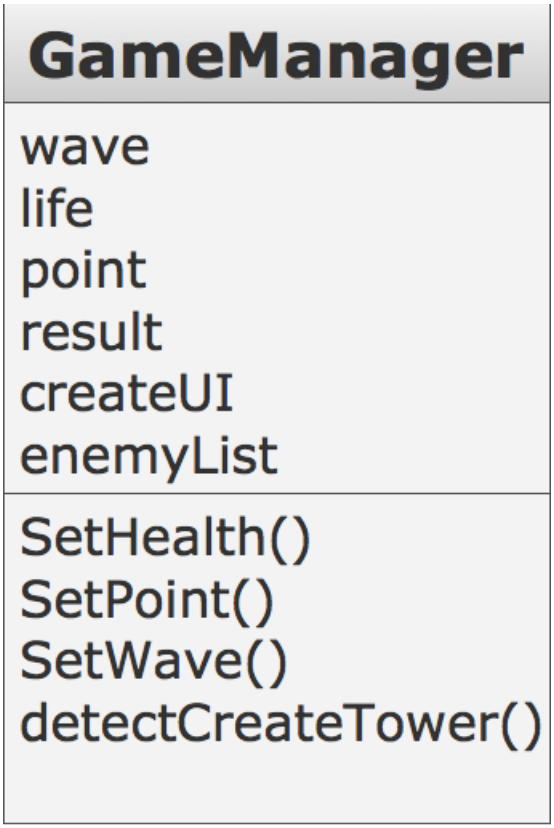


Figure 5.7: Build grid map

In Figure 5.7,"CreateUI" is a prefab instantiated when a player selects an available position on Grid Map. It is a group of tower buttons for players to choose. "EnemyList" is an array that saves alive enemies. Functions SetHealth(), SetPoint(),

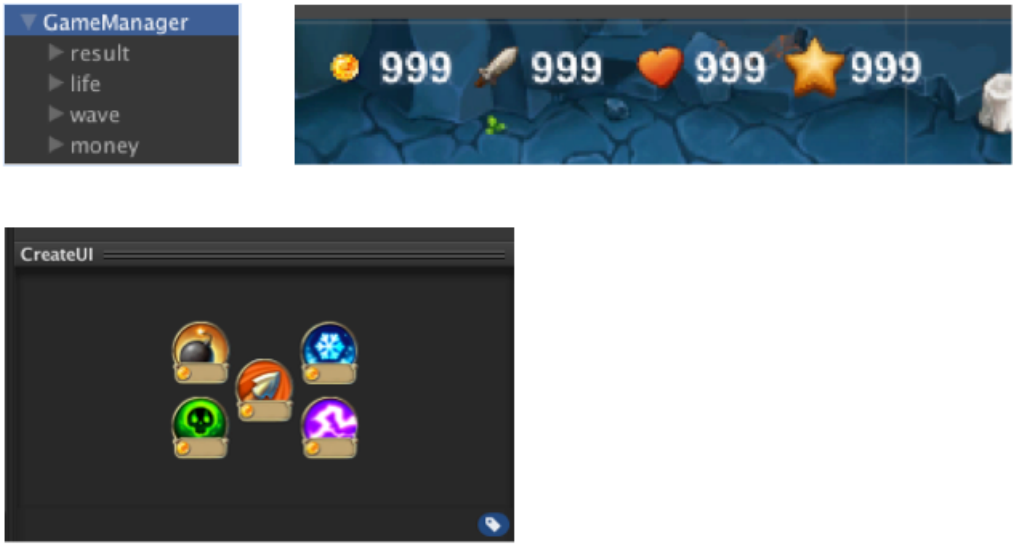


Figure 5.8: The shortcut of object "GameManager"

SetWave() will be called by other game objects when game status changes. DetectCreateTower() runs every fixed frame. If that kind operation were detected, game manager would instantiate a "createUI" at the selected place. Click one button of "createUI", the system will build a tower of the selected type.

5.2.5 Enemies

There are 11 types of enemies in a game. All these enemies are controlled by the same code, and they are made up of the same components. Therefore, we just need to take one enemy as an example to illustrate here.

An enemy runs along a pre-set path and bears attack from defenders. It has the following components: Animator, EnemyController, and Collider. Animator controls all animations of an enemy. EnmeyController is a script that controls an enemy's all

activities. Collider detects the collision between enemies and projectiles.

Make animations

Frist, we need to use sprite editor to split an enemy's sprite sheet.

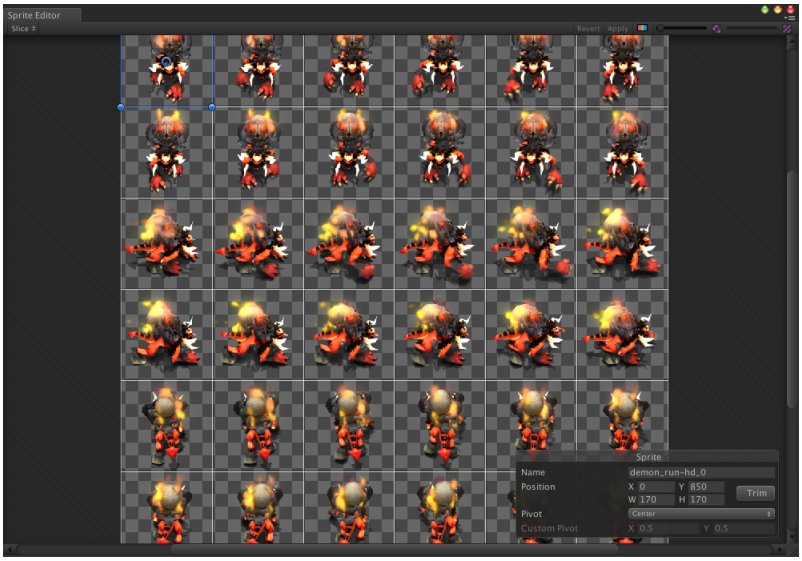


Figure 5.9: Sprite an enemy's sprite sheet

Then, make all animation clips one by one.

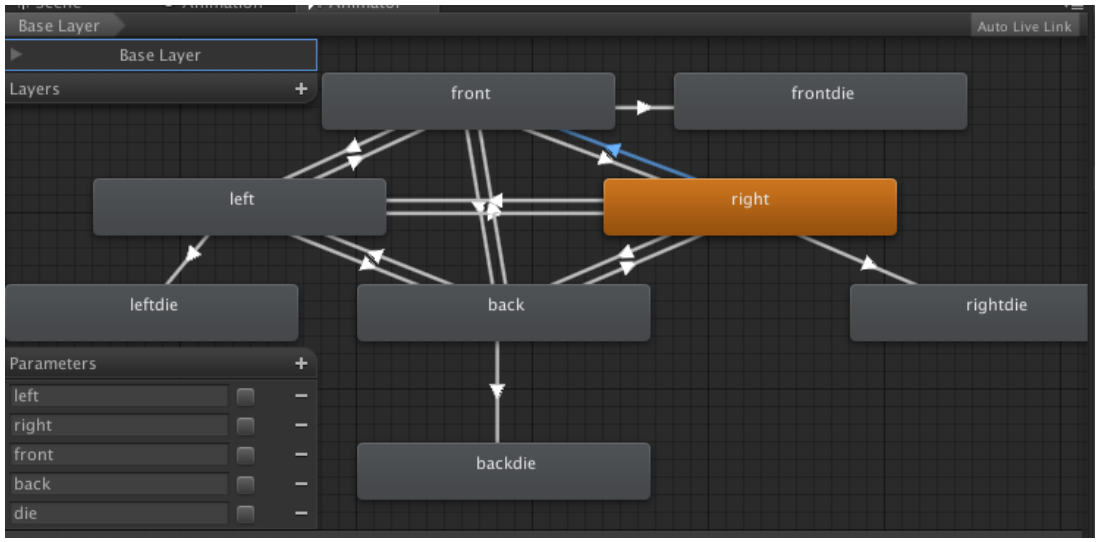


Figure 5.11: Animator

Figure 5.10: Make an animation clip

Finally, add transitions between these animations and set a parameter for each transition. When EnemyController triggers a parameter in Animator, the enemy's animation will transit to the corresponding one.

An enemy has four animations in running which correspond to four directions.

Also, its dead animations vary according to its last heading direction. These dead animations will call Destroy() function after finish playing.[11]

EnemyController

EnemyController is in charge of controlling an enemy running along the road, updating its health when the enemy is attacked by defenders, and checking whether the enemy is targeted. The next picture shows the class diagram of EnemyController.

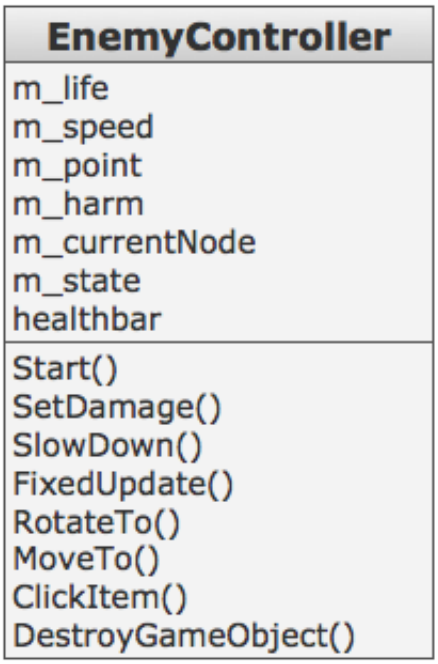


Figure 5.12: Animator

In Figure 5.12, attribute "m_currentNode" stores the nearest path node that the enemy is moving to; "m_state" saves the current animation state; "health bar" is a prefab that should be added to an enemy in Start() function. SetDamage() and SlowDown() are called when a projectile has hit this enemy.

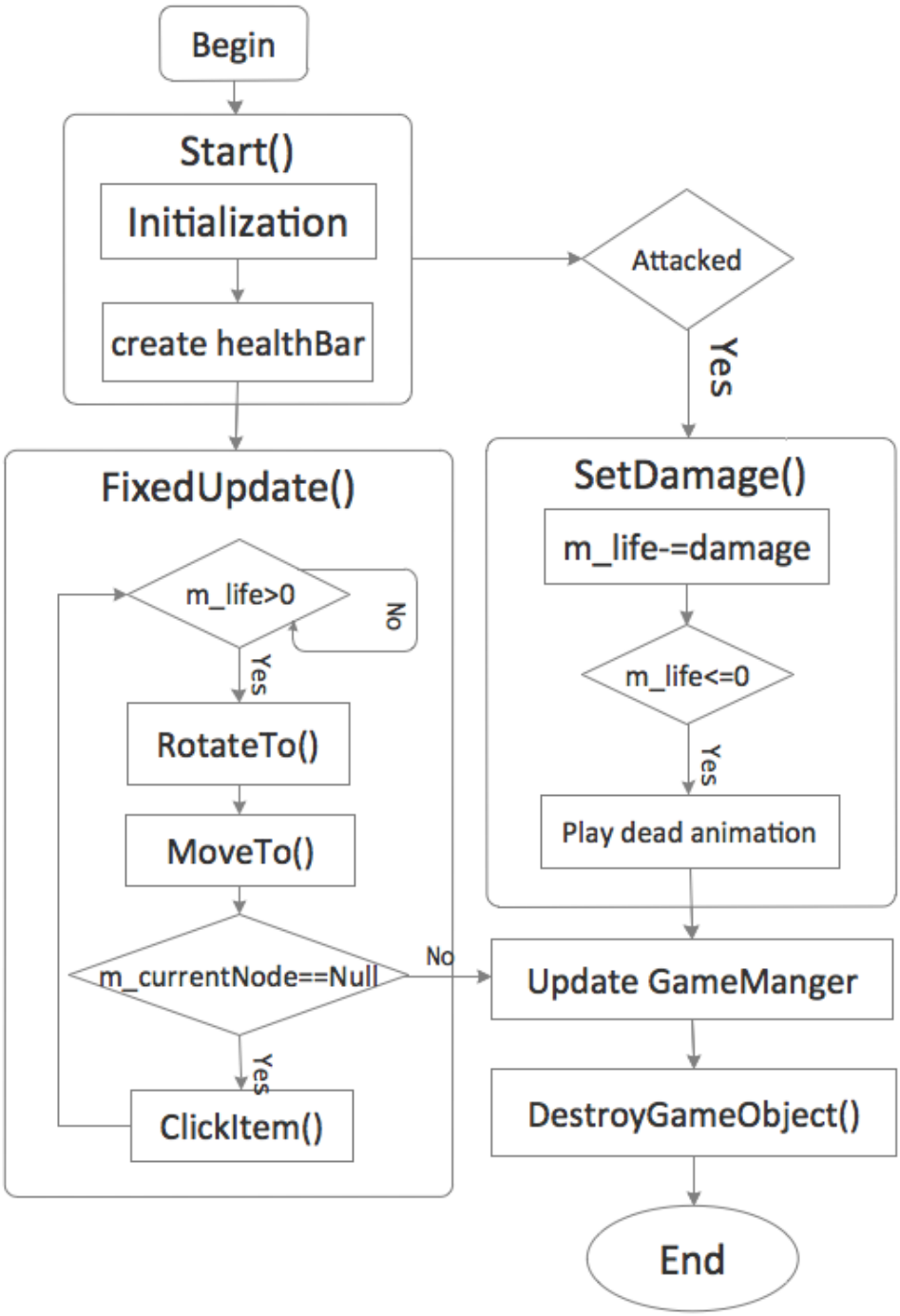


Figure 5.13: Animator

Figure 5.13 shows the workflow between the operations of EnemyController. After an enemy is created, the script first operate Start() to initialize the variables and instantiate a health bar for this enemy.

Then FixedUpdate() will be executed every 0.02 second until the enemy is destroyed. In FixedUpdate(), the script controls alive enemies running to its next path node. RotateTo() function turns the enemys heading direction to "m_currentNode" and adopts the proper animation. MoveTo() function moves the enemy towards "m_currentNode" with a speed of "m_speed". ClickItem() judges whether this enemy is marked. If the enemy is marked, the system will add an icon on it. If an enemy has arrived at the ending, the system will call SetLife() function in GameManager and destroy this enemy.

At the same time, the system continues checking if this enemy is attacked. When the enemy is hit, SetDamage() will be called, and the enemy's health will be reduced. If the value of "m_life" is less than zero, the script will trigger the dead animation to destroy this enemy and update the information in GameManager.

5.2.6 Enemy Spawner

Usually, enemies in tower defense games appear in packs wave after wave. Since the quantity of enemies is vast, the game needs a spawner to spawn enemies as pre-set order.

Set Spawning Order

In order to improve productivity, we will set enemies' order with Excel then save it as an XML file. Enemy Spawner loads this file and spawns enemies.

To begin with, we create an XML template to define the format of data source.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
    <!-- content -->
    <table wave="" enemyname="" level="" wait=""/>
    <table wave="" enemyname="" level="" wait=""/>
</ROOT>
```

Figure 5.14: Data template file

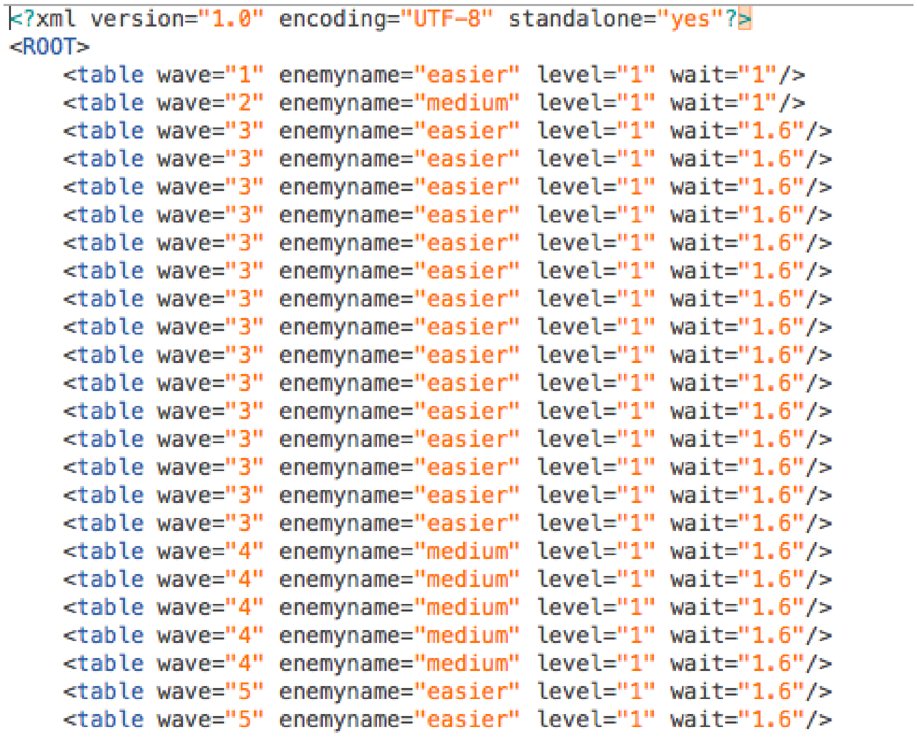
Then, map this template file to an Excel sheet and set spawning orders.

	A	B	C	D
1	wave	enemyname	level	wait
2	1	easier	1	1
3	2	medium	1	1
4	3	easier	1	1.6
5	3	easier	1	1.6
6	3	easier	1	1.6
7	3	easier	1	1.6
8	3	easier	1	1.6
9	3	easier	1	1.6
10	3	easier	1	1.6
11	3	easier	1	1.6
12	3	easier	1	1.6
13	3	easier	1	1.6
14	3	easier	1	1.6
15	3	easier	1	1.6
16	3	easier	1	1.6
17	3	easier	1	1.6
18	3	easier	1	1.6
19	4	medium	1	1.6
20	4	medium	1	1.6

Sheet1 Sheet2 Sheet3

Figure 5.15: Design in Excel

Finally, export the sheet as an XML file.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ROOT>
    <table wave="1" enemyname="easier" level="1" wait="1"/>
    <table wave="2" enemyname="medium" level="1" wait="1"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="3" enemyname="easier" level="1" wait="1.6"/>
    <table wave="4" enemyname="medium" level="1" wait="1.6"/>
    <table wave="4" enemyname="medium" level="1" wait="1.6"/>
    <table wave="4" enemyname="medium" level="1" wait="1.6"/>
    <table wave="4" enemyname="medium" level="1" wait="1.6"/>
    <table wave="4" enemyname="medium" level="1" wait="1.6"/>
    <table wave="5" enemyname="easier" level="1" wait="1.6"/>
    <table wave="5" enemyname="easier" level="1" wait="1.6"/>
```

Figure 5.16: XML File

Implement Enemy Spawner

Enemy Spawner can be divided into two parts according to its functions. The one is about loading data from assigned XML file to an array. The other one is about spawning enemies from that array. In class “EnemySpawner”, we define two classes “EnemyTable” and “SpawnData” to help store the data of enemies.

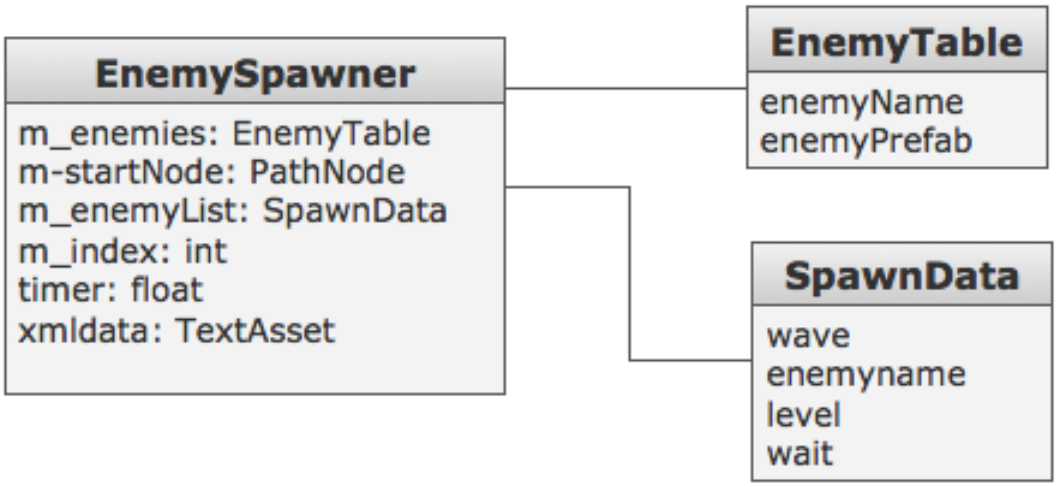


Figure 5.17: Class EnemySpawner

The array "m_enemies" saves a series of enemy prefabs, "m_startNode" refers to the beginning path node, "xmldata" points to the XML file and the data after analyzing XML file will be stored in "m_enemyList". Variable "m_index" record the number of lines to be executed, and "timer" is used to count waiting time. The process how EnemySpawner works is shown in Figure5.20.

At beginning, the script gets enemies' parameters from "Settings.txt" and set these parameters to the prefabs in "m_enemies". Then, EnemySpawner reads XML file and save data into "m_enemyList". After preparations, the game begins to spawn enemy. The system first checks whether the spawning wok has finished or not. If the value of "m_index" still less than the number of lines of that XML file, EnemySpawner has to continue spawning enemies. If the next enemy is in a new wave, it cannot be generated until its former wave is finished, and its waiting time has arrived. When all enemies have already been spawned, the script will check alive enemies in GameManager. If a player's life is beyond zero and no enemies are left, then the player wins.

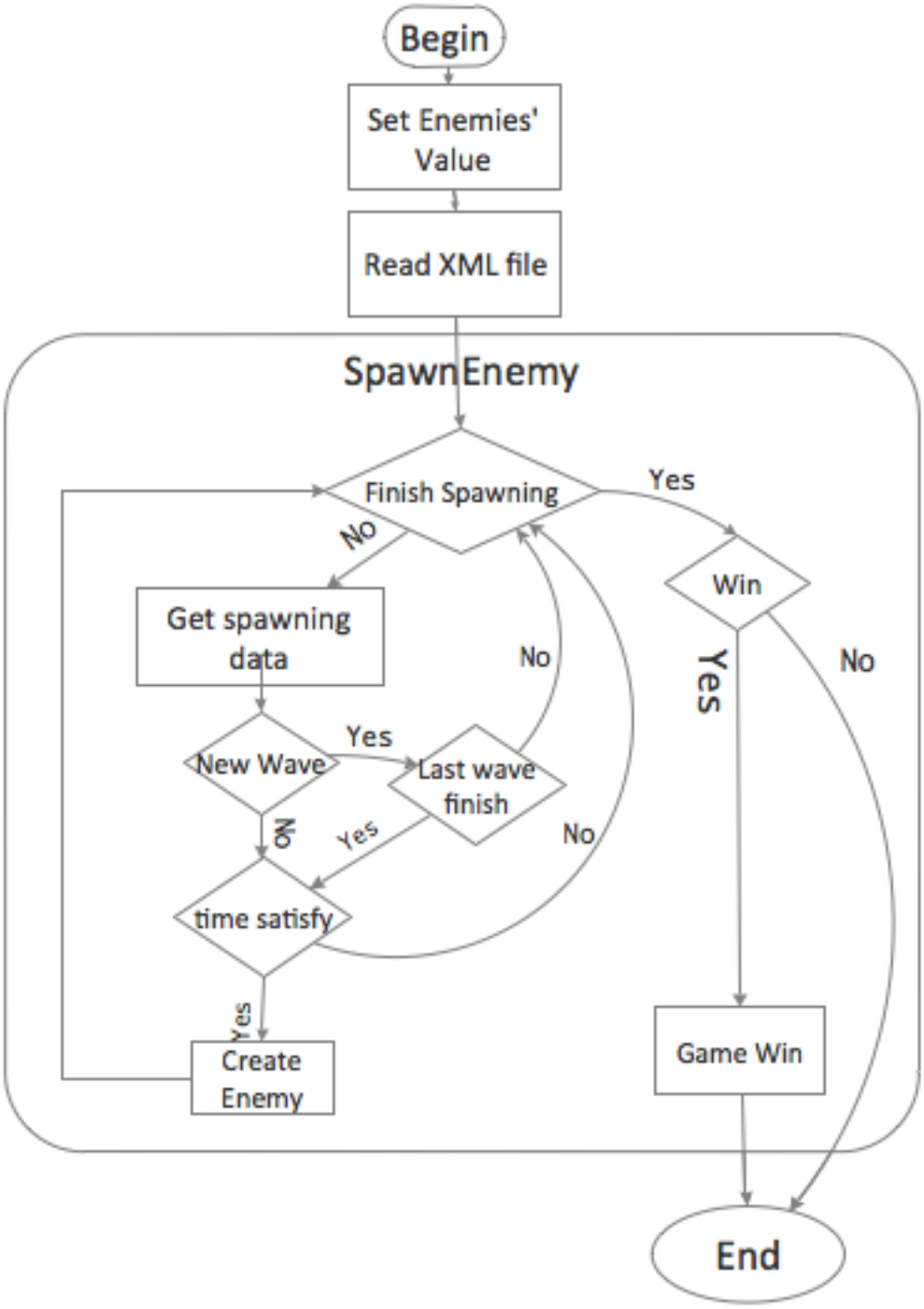


Figure 5.18: The workflow of EnemySpawner

Figure 5.19: Spawn Enemies

Figure 5.20: EnemySpawner judges game wins

5.2.7 Defender

Defenders forbid enemies arriving at the ending. There are five tower types, and each type has its unique character. A defender includes several components: “Defender Controller”, “attack”, “attack area”, “upgrade button” and “sell button”. “Defender Controller” controls a tower attacking an enemy and operations like upgrading a tower or selling it. The object “attack” is used to mark the position to instantiate projectiles and “attack” rotates the tower’s shooting direction to target at enemies. Components “attack area” and “upgrade button” are shown when a tower is selected. Figure5.21 displays the elements of a tower..

Figure 5.21: Components of a tower

Implement Defender Controller

As mentioned before, a tower’s strength mainly depends on four parameters: power, attack area, attack speed and cost, so the class “DefenderController” should contain

these four attributes. A tower can be upgraded three times, which means it has four level status and needs a variable to save its current level. What's more, it is necessary to mark the type of towers as well.

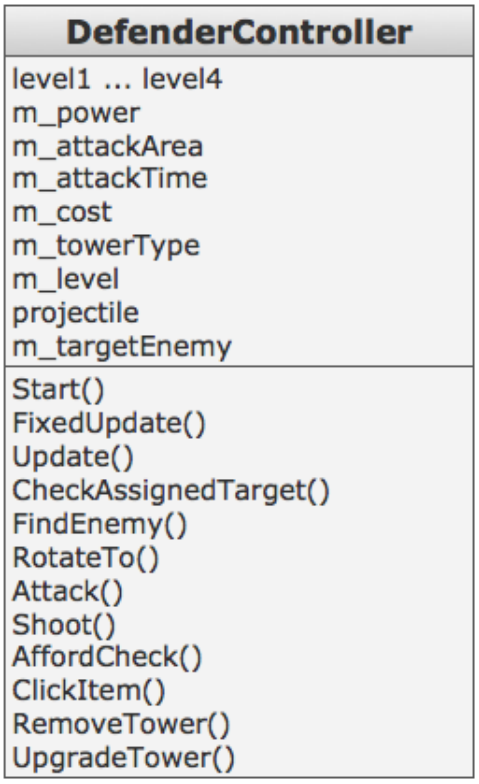


Figure 5.22: DefenderController

In Fugire5.22, attributes "level1" to "level4" are arrays storing the sprites of a tower. "m_targetEnemy" saves the enemy that the tower has to attack. If "m_targetEnemy" is null, the tower will keep on finding enemies. When a tower finds an enemy can be attacked, it first rotates to it. As for rotation, only the object "attack" turns to the enemy, and the tower does not rotate at all but changes its corresponding sprites to make the tower seem like rotating. The workflow of a defender is shown in Figure5.23.

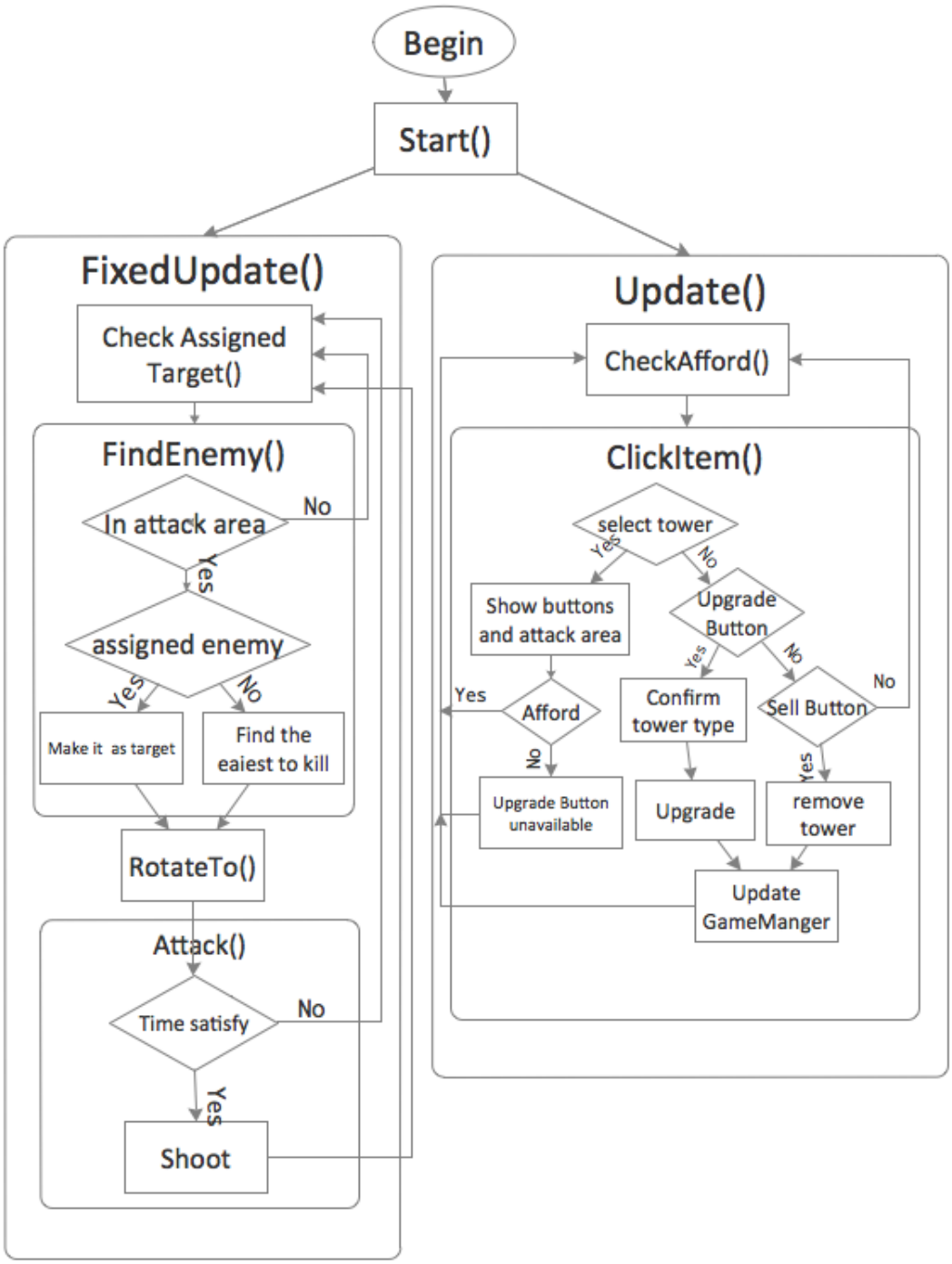


Figure 5.23: The workflow of DefenderController

After initialization, the tower begins to check users' operation and find enemies to attack at the same time. In the procedure of attack, the tower first searches all living enemies to check if there is a marked enemy. If the marked enemy is in the shooting area of this tower, the tower will select it as attacking target. Otherwise, the tower will find enemies that are in the shooting area and choose the weakest one to kill. Next, the tower rotates to the target, waits for a moment and shoots a projectile. During the whole life of a tower, the system continues checking whether current money could afford the next upgrade cost. When the player clicks on a tower, its buttons and attack area will show up. The player clicks buttons to do upgrading or selling operations. If the player could not afford the cost, the color of update button would turn gray. When the player clicks on other places rather than this tower itself, its attack area and buttons will disappear.

Projectile and Explosion

Defenders shoot projectiles to attack enemies. When a projectile hits an enemy, an explosion will be created. Both projectiles and explosions also have five types corresponding to the types of defenders. These projectiles and explosions bring various kinds of damage to enemies. For example, poison explosion reduces an enemy's health every second and the ice shell slows down an enemy.

Projectiles are controlled by script "ProjectileController". After being instantiated, a missile flies directly towards the target. OnTriggerEnter() function is called when the collision happens. After a projectile crashes on an enemy, "ProjectileController" instantiates an explosion, which causes damage on that enemy and destroys itself. The harm depends on the projectile's type.

Similarly, explosions are controlled by script "ExplosionContoller". It also has an animation component. Some explosions stay on enemies' body for a while and harm them constantly such as poison and ice.

Build towers

Each tower takes an area of a cell, and it will be placed at a cell's center. If a tower is constructed on a cell, the cell cannot be put another tower unless the tower has already been removed. Players can build a tower by clicking a tower button on "CreateTowerUI" that is mentioned in Game Manger part.

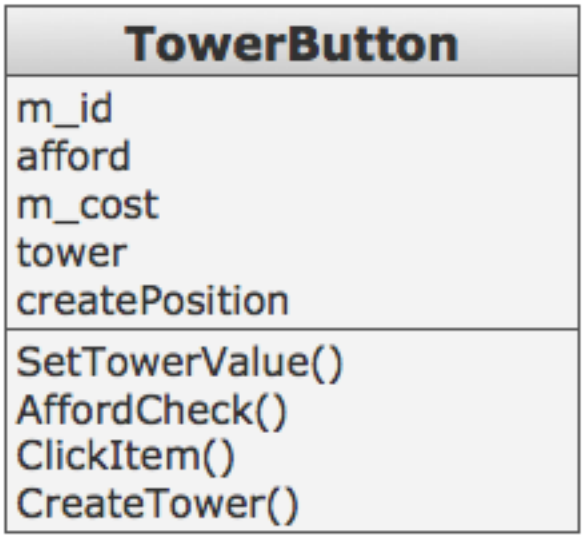


Figure 5.24: Class TowerButton

"TowerButton.cs" controls all tower buttons. As the figure shows, attribute "m_id" gives every tower button an ID to identify what type of tower they should create. If the variable "afford" equals false, this tower button will be unavailable, and

its color will turn gray. Attribute "tower" stores a tower prefab that will be built by this button. "createPosition" is the center of the cell that the player selected to build a tower.

At beginning, the script loads information from "Settings.txt" and calls SetTowerValue() to assign values for tower prefab. During the game, AffordCheck() function is being excited to update button status in real time. ClickItem() function listens to the player's action and checks whether players click on itself. If a button is clicked, it will run CreateTower() to build a tower at "createPosition".

5.2.8 Widgets

The application still needs some buttons to realize functions of suspend, play, replay, quit, open music and so on. All these buttons are under the control of "playButtonController". The script identifies takes different actions when a button is clicked by identifying its "buttonID". In addition, some notice boards are necessary to be implemented as well.

5.2.9 Music and Sound Effect

In order to make games more interesting, we add background music for each game. The project also provides some sound effects for game events as follows:

- Enemy: yelling sound, step sound and dead sound;
- Tower: build sound, upgrade sound, and remove sound;
- Projectile: hit sound;
- Game:beginning sound, celebrate sound, and lose sound.

Figure 5.25: A game's screen shot

5.3 Game Generator

As has been mentioned before, a family of tower defense games means these games share the same mechanism but look like entirely different in terms of their appearances. In other words, these games are made up of similar components which are controlled by the same code, but look like different tower defense games.

Developing so many games needs a large number of resources and takes a lot of time to repeat the similar work again and again. However, using a generator to produce these games improves productivity. Before implementing a generator, it is necessary to know what makes these games look differently.

Through analysis, we find that towers, enemies, and theme elements(including

entrance, exit, background) determine a game's appearance. Therefore, if we change one of these three parts of a game, the game will look like a new one. In this project, we offer three sets of prefabs for towers, enemies and themes respectively, which means the project could at most generate as many as 27 games. Figure 5.26 shows three groups of towers as an example.

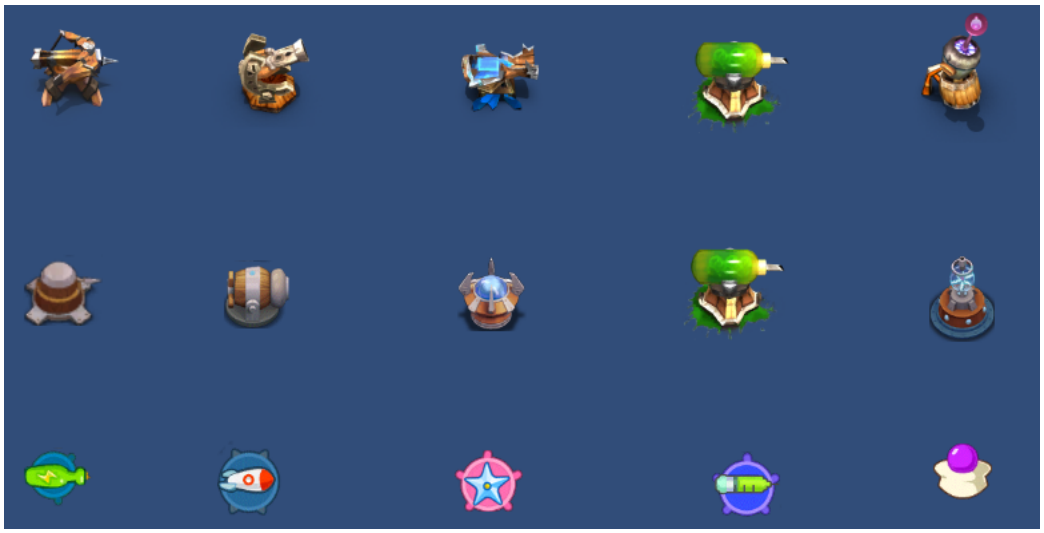


Figure 5.26: Three tower groups

Before generating a game, it is necessary to build a resource library for the generator to select. As tower prefabs are stored in tower buttons and tower buttons are stored in a Game Manager, the generator loads different tower groups by selecting different Game Managers in resource library. Similarly, the generator chooses an Enemy Spawner to load a group of enemies since a group of enemies is stored in an array of an Enemy Spawner. Besides, the library also includes various groups of theme elements like entrances, exits, and background maps. The resource library is created in script "LoadingGame.cs" shown in Figure5.27.

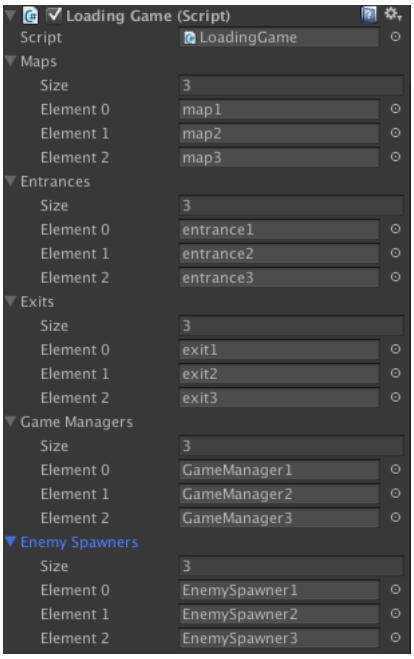


Figure 5.27: Resource library

The procedure of generating a new game is illustrated in Figure 5.28. In Main Menu, a player selects a group of towers, enemies, and themes respectively. The player's selection will be preserved in attributes "TowerID", "EnemyID", and "ThemeID" of Settings. Before enters into a level, the system creates a game generator to load resources. Initially, the game generator gets loading information from Settings component and selects prefabs from resource library. Next, it instantiates selected prefabs one by one and places them as children under "Game" object. Finally, the generator sets some initial values for Game Manager and Enemy Spawner.

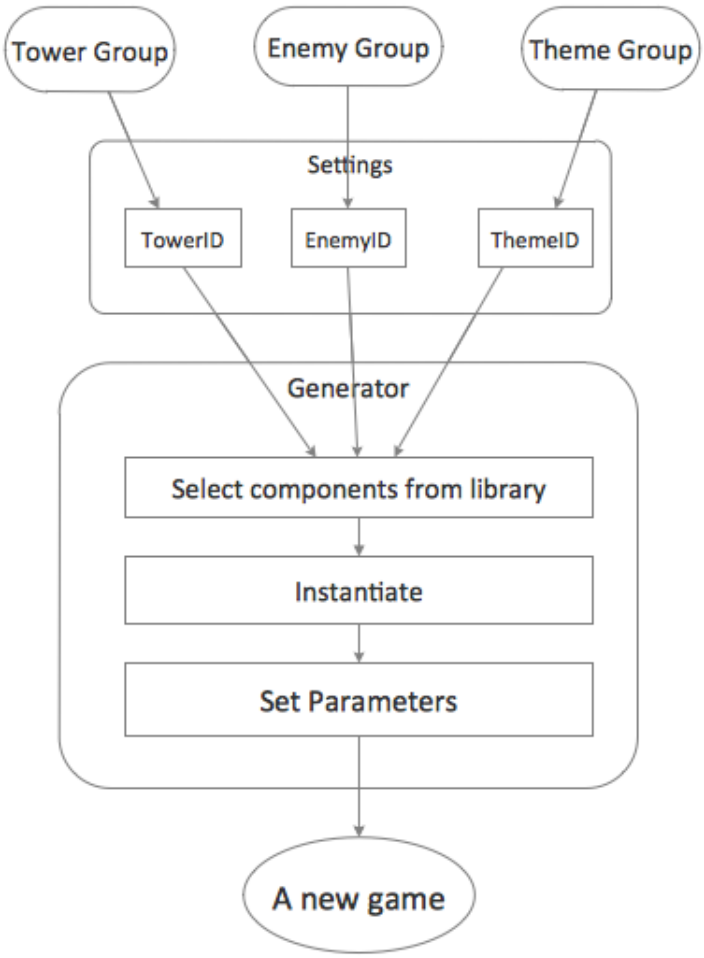


Figure 5.28: The workflow of generator

The screenshot of generating a game is shown bellow:

Figure 5.29: MainMenu

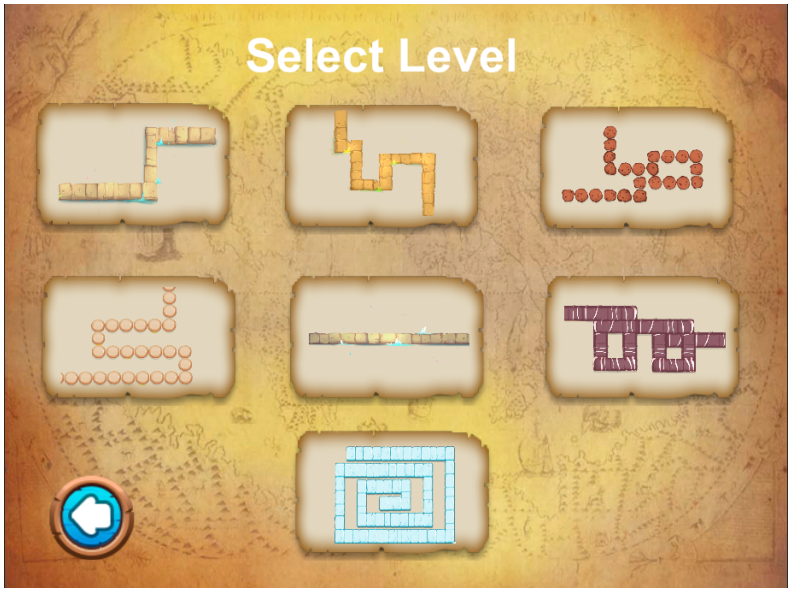


Figure 5.30: LevelSelect

Figure 5.31: Two games played in the level

5.4 Data Collection

As an experimental tool, data collection is essential to research players' performance under different screen sizes. When one level is over, the system will save data into a file. The data can be divided into two parts: one contains all parameters of game settings and records final result; the other one keeps operations of players.

We create a component "Recorder" to listen events and collect data. "Recorder" is controlled by script "RecordController.cs". "RecordController" defines a class to stipulate the format of an operation. An Operation instance saves what type it is, when it happens, and where. Array list "ops" stores all operation records and WriteOperations() saves these recorders into a file.

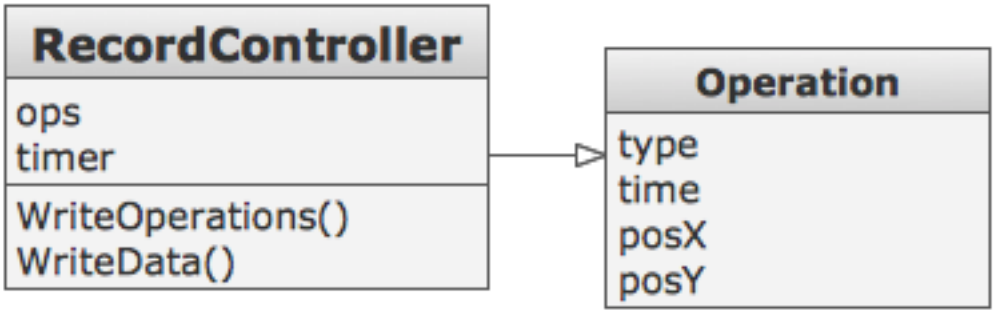


Figure 5.32: RecordController

"RecorderController" calls function WriteData() to keep all information of a game, including the loaded level, screen resolution, selected Ids like tower group, enemy group and theme group, the status of music and sound, all parameters of towers, enemies and game manager, and the final result such as score, life, wave, and money.

Operation20150105181734.txt

```
LoadedLevel:level2
Settings:
width:916,height:687,EnemyID:2,TowerID:1,ThemeID:0,music:False,sound:False
Operations:
Type:LightTower,PosX:5.5,PosY:4.5,Time:3.999997
Type:IceTower,PosX:5.5,PosY:5.5,Time:6.890063
Type:ElecTower,PosX:7.5,PosY:4.5,Time:9.40012
Type:LightTower,PosX:3.5,PosY:5.5,Time:14.65024
Type:Sell,PosX:3.5,PosY:5.5,Time:18.14032
Type:PoisonTower,PosX:3.5,PosY:5.5,Time:20.01036
Type:HeavyTower,PosX:8.5,PosY:4.5,Time:24.12046
Type:Upgrade,PosX:5.5,PosY:5.5,Time:42.65885
Type:Upgrade,PosX:5.5,PosY:4.5,Time:55.95662
Type:Upgrade,PosX:3.5,PosY:5.5,Time:76.65797
Type:Upgrade,PosX:5.5,PosY:5.5,Time:102.4835
Type:Upgrade,PosX:7.5,PosY:4.5,Time:111.7855
Type:Enemy,PosX:6.827109,PosY:5.106156,Time:111.7855
Type:Enemy,PosX:7.089687,PosY:5.411834,Time:113.6259
Type:Upgrade,PosX:8.5,PosY:4.5,Time:123.1479
Type:Upgrade,PosX:5.5,PosY:5.5,Time:162.0602
Type:Enemy,PosX:4.776755,PosY:5.760098,Time:162.0602
Type:Sell,PosX:3.5,PosY:5.5,Time:176.6422
Type:Upgrade,PosX:7.5,PosY:4.5,Time:194.6623
```

20150105181734.txt

```
LoadedLevel:level2
Settings:
width:916,height:687,EnemyID:2,TowerID:1,ThemeID:0,music:False,sound:False
GameManager:
Wave:10/10,Life:9/10,Money:107/60,Result:1104,Win
Enemy Parameters:
Name:faster,Life:72,Speed:2,Reward:4,Harm:1
Name:boss,Life:1500,Speed:1,Reward:140,Harm:4
Name:hardest,Life:960,Speed:1,Reward:80,Harm:2
Name:largehealth,Life:720,Speed:0.8,Reward:32,Harm:2
Name:harder,Life:480,Speed:1,Reward:40,Harm:2
Name:fastest,Life:54,Speed:3,Reward:5,Harm:1
Name:easier,Life:36,Speed:1,Reward:2,Harm:1
Name:easiest,Life:8,Speed:0.8,Reward:15,Harm:0
Name:revive,Life:144,Speed:1,Reward:8,Harm:1
Name:fast,Life:72,Speed:1.5,Reward:6,Harm:1
Name:medium,Life:72,Speed:0.8,Reward:4,Harm:1
Tower Parameters:
Type:LightTower,Power:8 12 18 27,AttackArea:1.7 2 2.3 2.6,AttackTime:0.9 0.8 0.75
0.7,Cost:12 24 48 96
Type:HeavyTower,Power:14 21 32 48,AttackArea:1.6 1.8 2 2.2,AttackTime:1.2 1.05 0.9
0.8,Cost:18 36 72 120
Type:PoisonTower,Power:5 8 11 15,AttackArea:1.7 2 2.3 2.6,AttackTime:0.9 0.8 0.7 0.6,Cost:
15 30 60 110
Type:IceTower,Power:4 5 7 9,AttackArea:1.6 1.8 2 2.2,AttackTime:1 1 1 1,Cost:10 20 40 80
Type:ElecTower,Power:11 17 27 40,AttackArea:2 2.3 2.6 2.9,AttackTime:0.1 0.1 0.1 0.1,Cost:
18 36 72 120
```

Figure 5.33: Data Files

5.5 Replay Record

In order to analyse a player's performance more precisely, researchers need to watch the whole process sometimes. Therefore, the project offers a function to reappear the scene when a player was playing. The way to realize it is to redo all operations under the same situation by analysing an operation record file. The function is implemented in "LoadingGame.cs".

When a user chooses a record to replay, the system first read the related file and then store all lines into an array list. Then replay begins, and the system executes lines one by one until all operations have been executed. There is a timer calculating time through whole process. When time equals the trigger time of an operation, the operation will be performed, and then the index points to the next line.

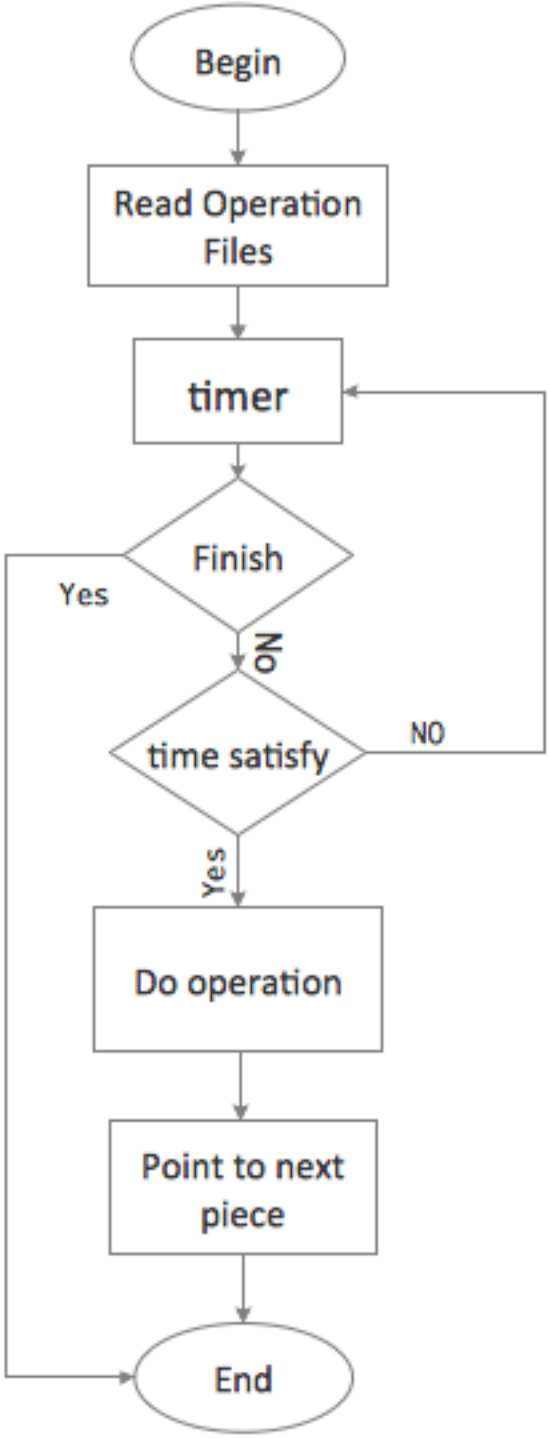


Figure 5.34: The workflow of Replay function

Chapter 6

Future Enhancements

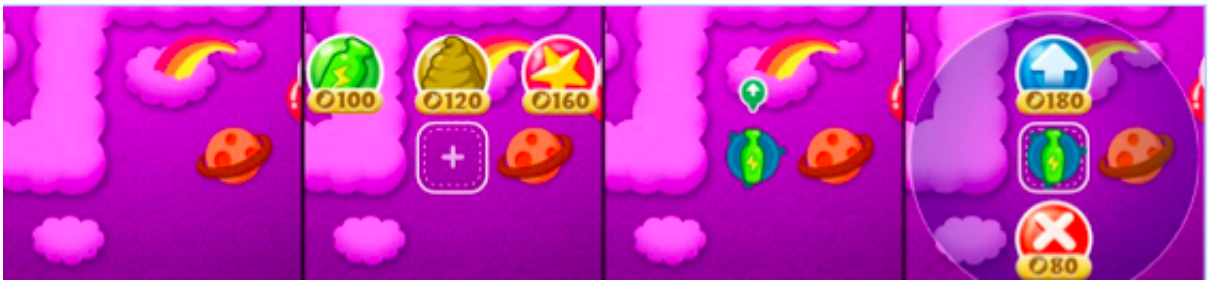
After the project was completed, we showed it on a digital media show. Generally, the feedback from players was great, but they also raised a problem. The problem was that some players got confused about how to interact with the games at the beginning. This was because the way of interaction of this project was not designed as adaptable as their habits, which takes them a short time to learn how to play the games.

In order to improve the interactive experience, many efforts have been paid on researching other TD games. We found that there are mainly two styles of interactive design on current market:

One provides a menu with several types of towers on the map. The player selects a type of tower in this menu and places it on his desired grid. He can click on an existing tower to active its options as well, and then click on the upgrade button to upgrade the tower or tap the remove button to remove the tower. *Fieldrunners* and *Desktop TD* are designed like this.

Figure 6.1: Fieldrunners Interaction Design

The other one is designed like: In order to build a tower, the player must click on a grid to get a menu showing all types of towers. Then he clicks a tower button from the menu, and then the game will build a basic level tower. Each tower has some levels and, after building the tower, the player could active tower options by clicking on it again. *Carrot Fantasy*, *Elf Defense*, *Kingdom Rush* and *Pirate Legends TD* adopt this style [3].



(a) Carrot Fantasy

(b) Elf Defense

(c) Kingdom Rush

(d) Pirate Legends TD

Figure 6.2: The Second Type of Interaction Design

At last, the project decides to provide both the two ways for players. We suppose there may be other better solutions of this issue, which is waiting for designers to invent. However, at the moment, two interactive methods are enough for players to take part in experiments.

(a) Solution1

(b) Solution2

Figure 6.3: Two Solutions of Interactive Problem

Chapter 7

Conclusion

Overall this project was a success. It can produce interesting games, which have received a lot of praises. At the same time, during the process of development, we figure out an efficient way to improve productivity by introducing product family engineering (PFE) technologies into this project. This work has proved PFE is also suitable for game development, especially for the games that requires an extensive of variations. Apart from above achievements, this project also invented a technique that makes all graphic sprites in Unity are able to adapt different scale sizes.

Bibliography

[1] Bosch, J. (2000). Bosch, j., 2000. design and use of software architectures: adopting and evolving a product line approach. *Pearson Education (Addison- Wesley and ACM Press), ISBN 0-201-67494-7.*

[2] Bosch, J. (2004). On the development of software product family components.

[3] Fara, N. (2014). Deconstructing the tower defense game giving options to the players.

[4] Fernando Palero, Antonio Gonzalez-Pardo, D. C. (2015). Simple gamer interaction analysis through tower defence games. *New Trends in Computational Collective*, pages 184–185.

[5] Furtado, A. W. and Santos, A. L. (2006). Using domain-specific modeling towards computer games development industrialization. *In The 6th OOPSLA Workshop on Domain-Specific Modeling.*

[6] Gajos K. Z., W. D. S. and O., W. J. (2010). Automatically generating personalized user interfaces with supple. *Artificial Intelligence.*

[7] Institute, C. M. S. E. (2006). Software product family engineering.

[8] Jin, X. (2013). *Unity3D Mobile Development*. Qing Hua.

[9] Johnson, L. (2012). Clash of clans review.

[10] Neighbors, J. (1980). Software product lines.

[11] Pang, M. (2014). *Unity4.3 Development*. Qing Hua.

[12] Phillipa Avery, Julian Togelius, E. A. (2012). Computational intelligence and tower defence games.

[13] Pierre America, Henk Obbink, R. v. O. (2000). A component-oriented platform architecting method family for product family engineering. *Proceedings SPLC-1*, (167-168).

[14] Pohl K., Bockle G., . L. F. v. d. (2005). Software product line engineering. *Springer-Verlag*.

[15] Rummell, P. A. (2013). Adaptive ai to play tower defense game.

[16] Steven K.C., H.-C. K. and Chang, C.-M. (2013). A multi-agents coordination mechanism to improving real-time strategy on tower defense game.

[17] Sun Xiaopeng, Feng Junjie, S. Y. and Cuifang, L. (2010). Using uml in game design. *CNKI*.

[18] Sybren Deelstra, M. S. (2004). Roduct derivation in software product families: a case study. *The Journal of Systems and Software*.

[19] THEVATHASAN, M. (2014). Modeling hud layouts for selection based video games. page 27.

[20] Unity (2014). Introduction of unity. http://unity3d.com/unity.

[21] van der Linden, F., Bosch, J., and Erik Kamsties, Kari Känsälä, H. O. (2012). Software product family evaluation.