

TOWER DEFENSE GAME

*A project report submitted in partial fulfillment of the requirements for the
degree
of*

**Bachelor of Computer Applications (BCA)
OF
North Eastern Hill University (NEHU)
2017**



Submitted by

Name: Micheal Lyting

Roll No.:P1400027

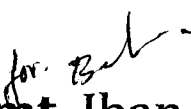
Reg. No.:9505 of 2012 - 2013




**DEPARTMENT OF COMPUTER SCIENCE & APPLICATIONS
SHILLONG COLLEGE
SHILLONG 793003
MEGHALAYA.**

CERTIFICATE

This is to certify that the Project work titled **TOWER DEFENSE GAME** is a bonafide work done by **Micheal Lyting**, University Roll No.: **P1400027**, Registration No.: **9505 of 2012-13**, under my guidance during the final year of **Bachelor of Computer Applications**.


Smt. Iban Sunn
Supervisor
Dept. Of Comp. Sci.&App.
Shillong College

Project seminar was held on 17/4/2017 at
Shillong College, Shillong.


Smt. Aiom Mitri
HOD
Dept. Of Comp. Sci.&App.
Shillong College
Shillong.


External examiner

Dept. Of Comp. Sc.&App., Shillong College, Shillong.



Acknowledgements

I would like to thank my supervisor Smti.Iban Sunn for ~~her~~ expertise and guidance throughout the entire duration of this work. ~~Her~~ 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.

Contents

1 Objective and Requirements

1.1 Objectives

1.2 Requirements

2 Prerequisite Knowledge

2.1 Tower Defense Games

2.1.1 Concept

2.1.2 Existing Variations

2.2 Software Product Families

3 Software Design

3.1 Project Architecture

3.2 User Operations

3.2.1 Players' Use Cases

3.2.2 Experimenters' Use Cases

3.3 Analysis Models

3.3.1 Object Model

3.3.2 Domain Lexicon

3.4 Dynamic Models

3.4.1 State Charts

3.4.2 Flow chart

4 Game Designs

4.1 Tower Design

4.2 Enemy Design

4.3 Game Manager Design

4.4 Game Balance

4.4.1 Build a Module

4.4.2 Set Parameters

4.4.3 Adjust game balance

5 Implementation

5.1 Development Environment

5.2 Development a tower defense Game

5.2.1 Background Controller

5.2.2 Grid Map

5.2.3 Path

5.2.4 Game Manager

5.2.5 Enemies

5.2.7 Defender

6 A* (A-star)

6.1 Motivation

6.2 What is A* Search Algorithm?

6.3 Why A* Search Algorithm?

6.4 Explanation

6.5 Introduction

6.5.1 The Search Area

6.5.2 Starting the Search

6.5.3 Path Scoring

6.5.4 Continuing the Search

6.6 Algorithm

7 Conclusion

7.1 References

7.2 Bibliography

Chapter 1

Objectives and Requirements

This project aims to develop a tower defense games. This report illustrates specifically how the project is designed and implemented.

Objectives

This project is about developing an application 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:

1. In the first place, the core goal of this project is to implement a 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 sell the towers.
2. 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 re-think 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. Before finding the solutions, it is essential to know how different screen sizes influence on the performance of players.

3. 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. 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.

4. Finally, the project aims to offer a possibility to be a part of games in the future study of player's 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 player's behaviours analysis. It is very interesting to find out how a player performs if he was under a nervous situation, and what strategy he would adopt.

1.1 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 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 game play, 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 waves and so on.

As for the aims of studying how different sizes of display influence player's 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 wave, 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 "product 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 exists 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:

1. A game has a very well defined, and a small set of tasks.

2. 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.
3. All elements can be scaled to fit different screen sizes automatically.
4. Implement a generator to produce TD games that look differently but have the same mechanism.
5. 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 are on a troop of different types of enemies. 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 existing 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. 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.

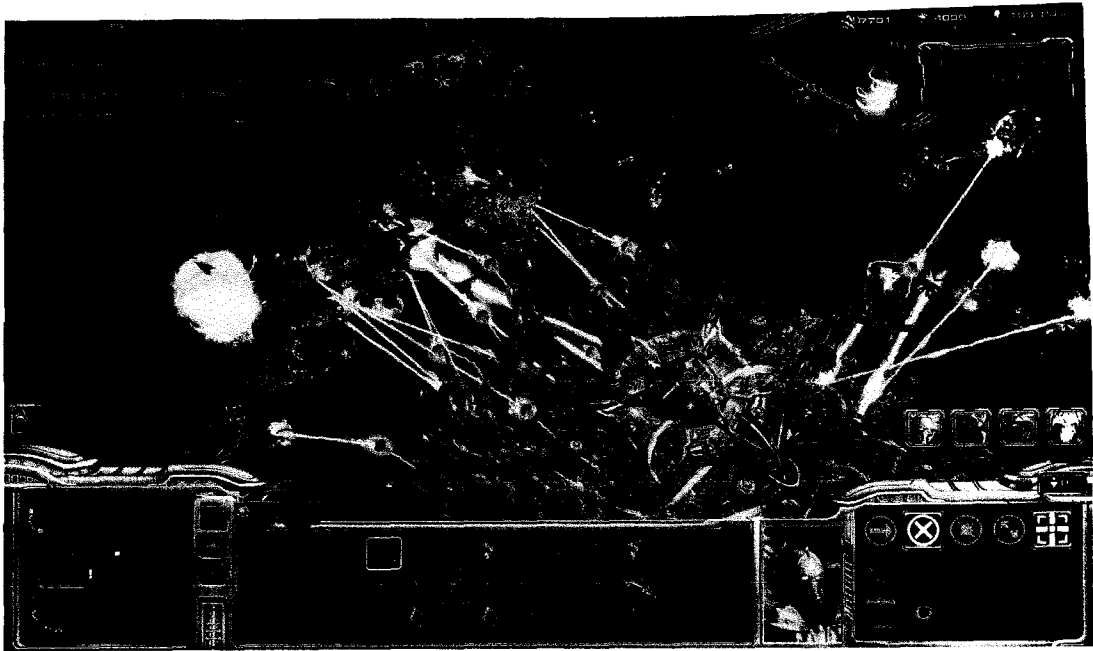


Figure 2.1: Starcraft2

Mechanics

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

1. 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.
2. 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.
3. The player puts towers on the map to destroy the enemies.
4. There should always exist at least one path that allows enemies to reach the exit.
5. When an enemy reaches the exit the player loses one or more lives.
6. The game is lost when the player has no lives left.
7. Enemies usually come in turns or waves with a large troop.
8. The game is won when there are no more waves.
9. Towers can be upgraded and removed.

1. Building towers and upgrading them costs money. Removing towers gets money back but less than the price purchased when they are built.

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 generated in next wave. 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.

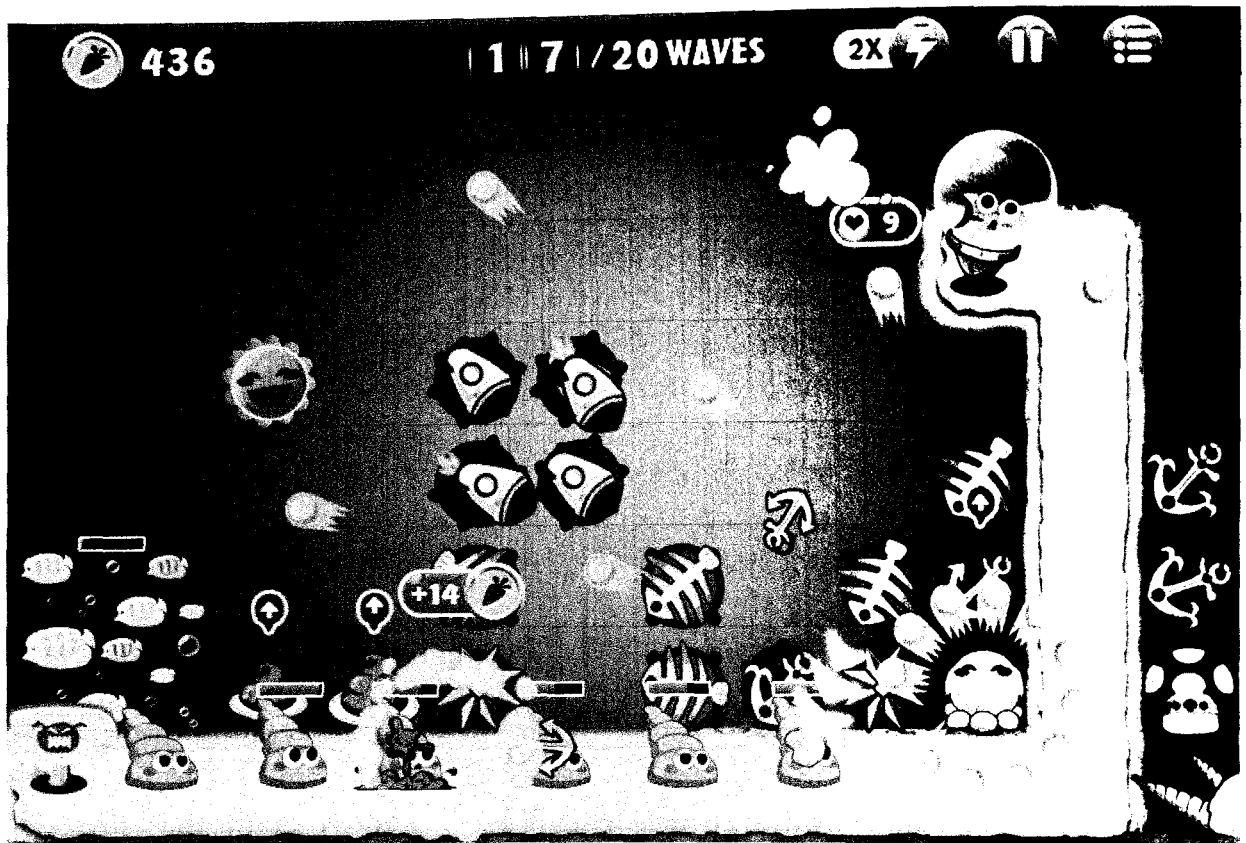


Figure 2.2: Carrot Fantasy

One of the most popular multi-platform TD games is the Plants vs. Zombies created by Pop cap 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

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.

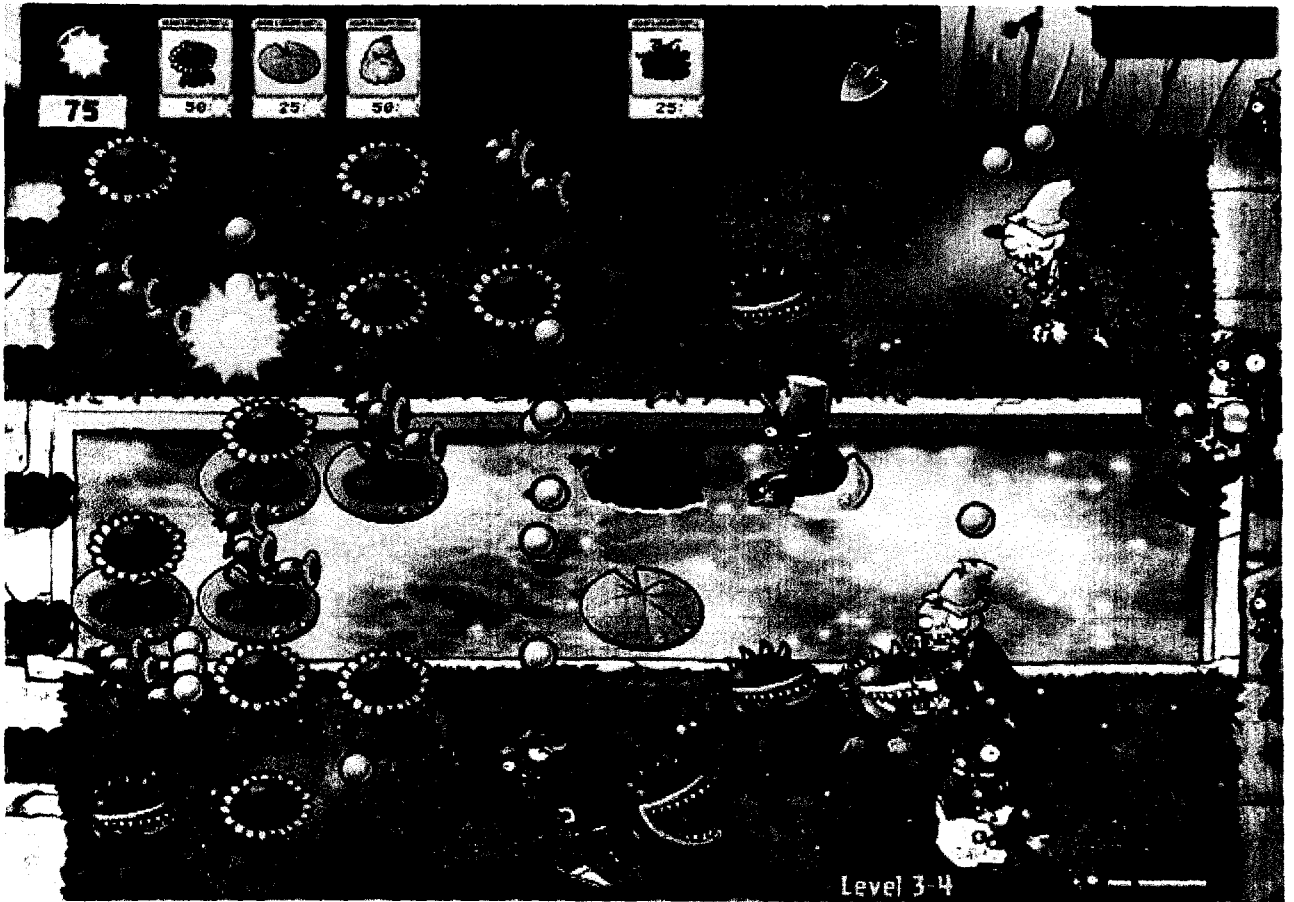


Figure 2.3: Plants vs. Zombies

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 y or jump over towers to avoid being attacked.

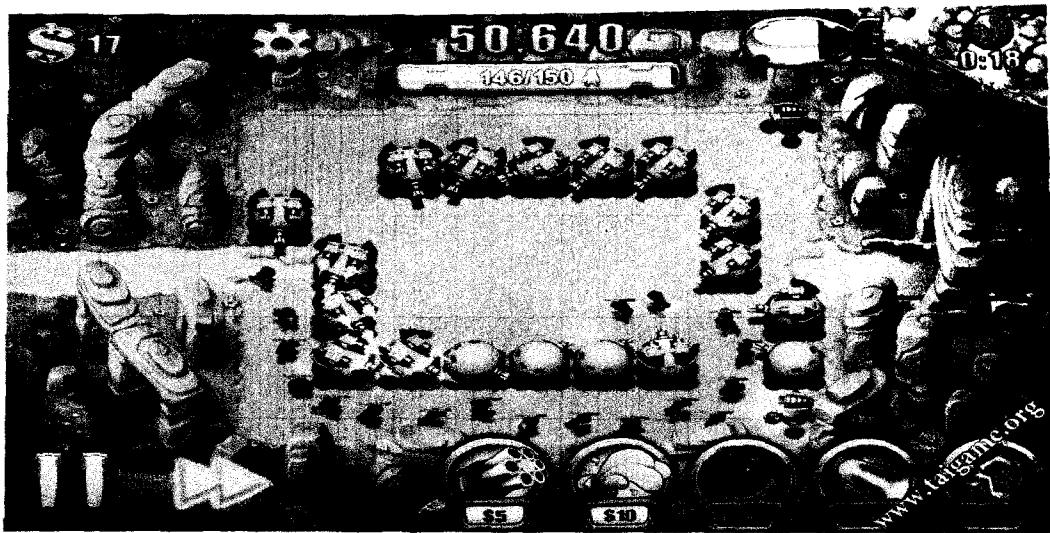


Figure 2.4: Fieldrunners

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 are 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 .



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.

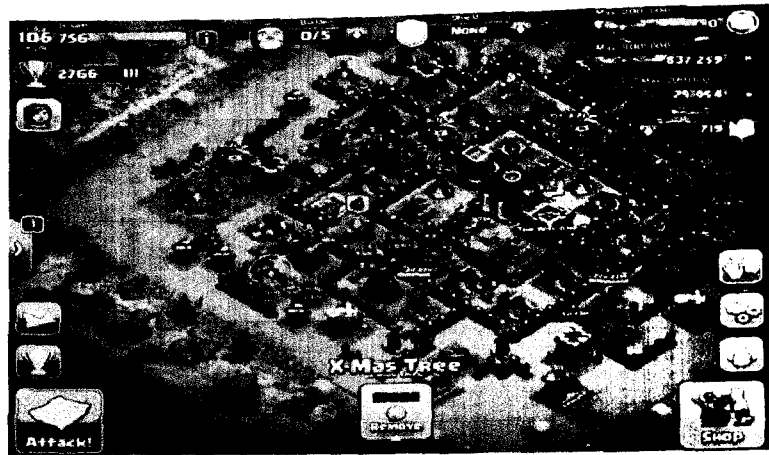


Figure 2.6: Clash of Clans

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 recognized in the software industry. Some organizations have adopted this technology and got satisfy achievements. Some companies are considering adopting 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 are one of the main benefits to be, achieved. Before a product family can be successfully established, an extensive process has to be followed. This process is known as product family engineering. 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 architecture based on commonality and planned variability's.

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 labor costs. 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.

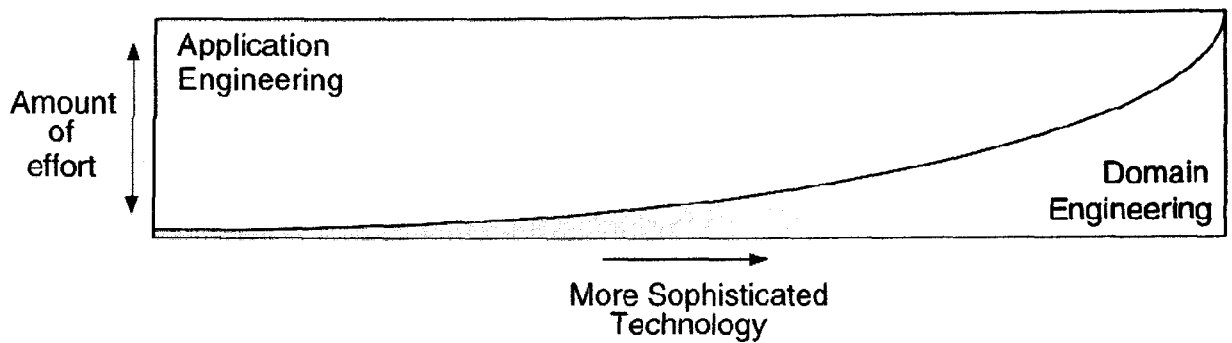


Figure 2.7: Domain vs. application engineering

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 phase 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.

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 modeling the variable and common parts are important in software family engineering .

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

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

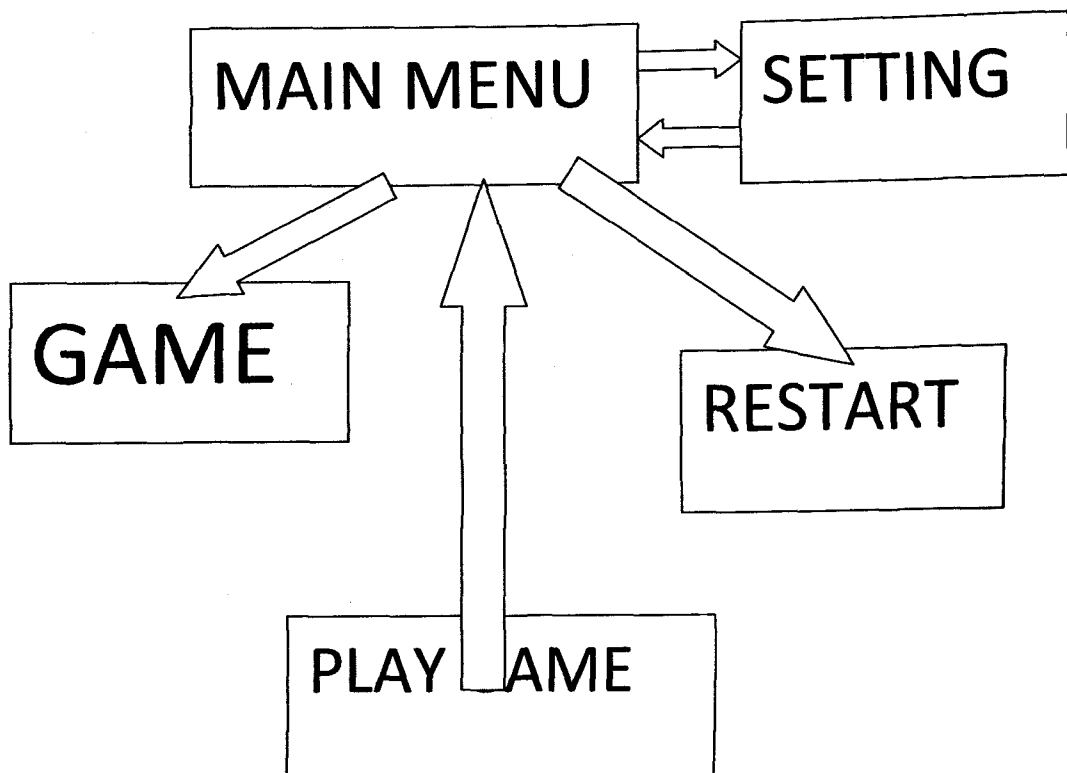


Figure 3.1: The main composition of project

have to assign towers, enemies and themes to create a game at "Main Menu" scene. Then click "Play" button and the application will jump to "Play" 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 "Main Menu". 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 "Main Menu" 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 Players' Use Cases:

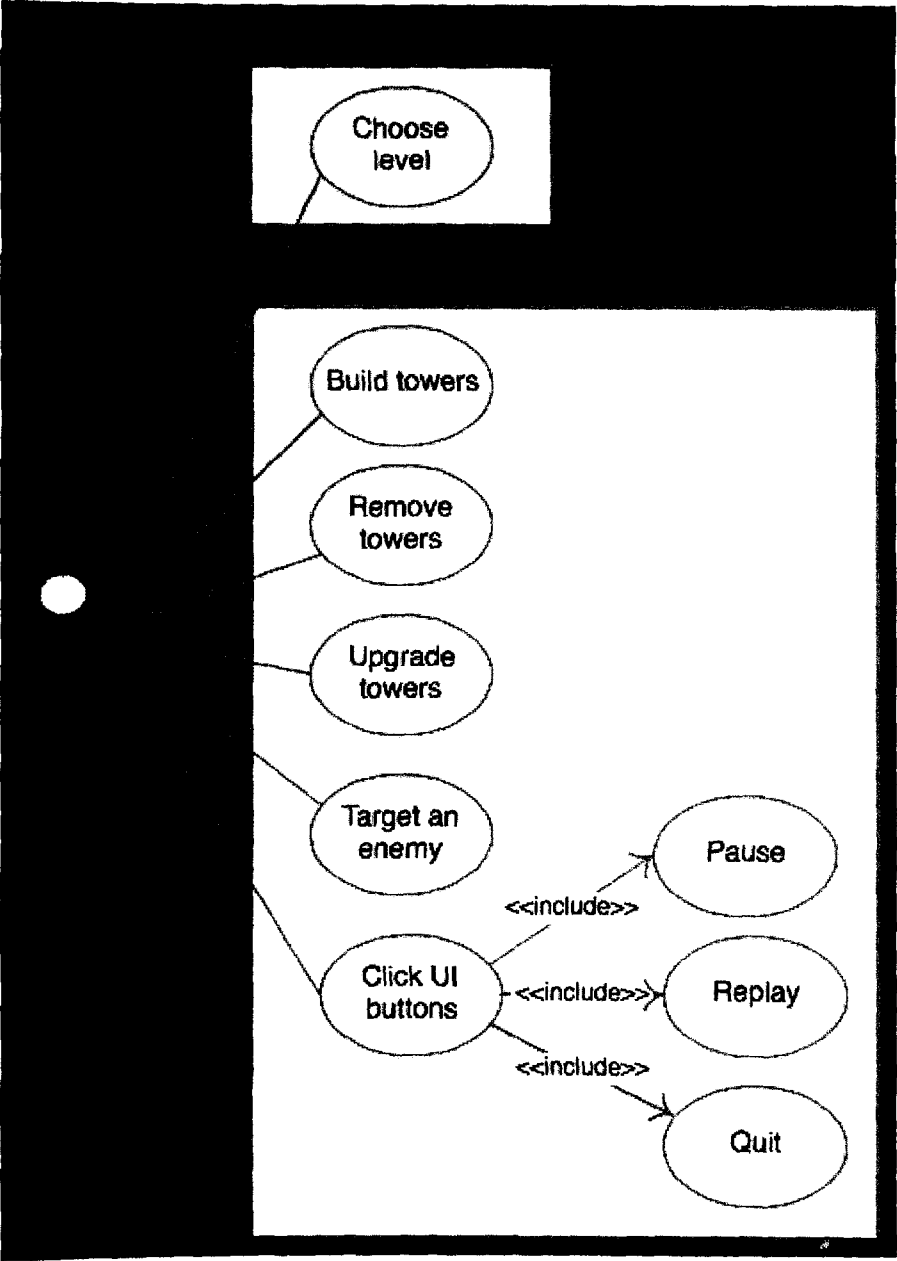


Figure 3.3: Players' Use Cases

Players choose a game level in "Level Select" 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 _re 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 "Main Menu" scene.

3.2.2 Experimenters' Use Cases:

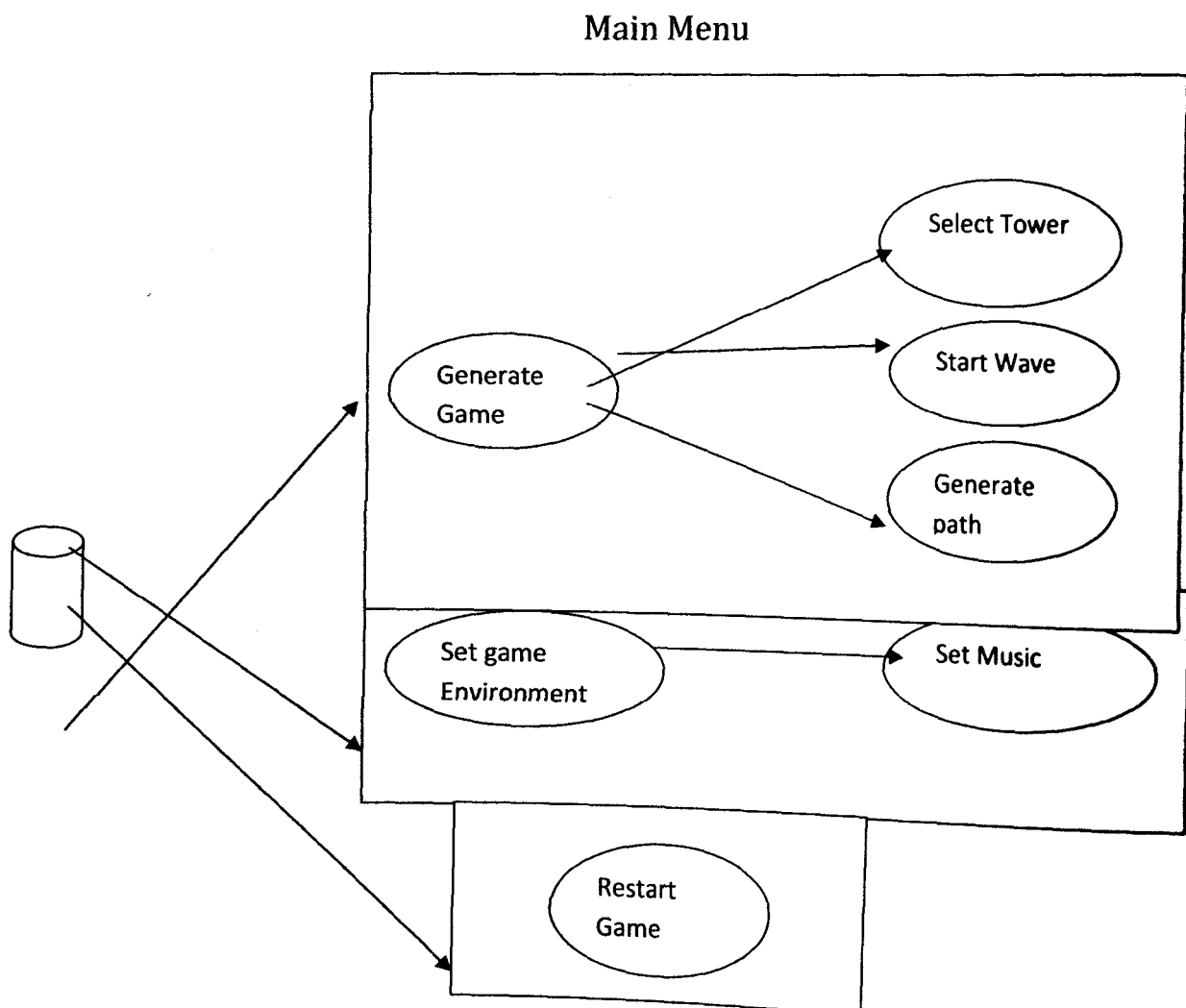


Figure 3.4: Experimenters' User Cases

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 "Main Menu" to enter into the "Settings" scene.

In there, 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 "Main Menu". For example, Experimenter A determines to select the first tower set, the second start wave, and the third generate path. After clicking the play button, the system generates a TD game according to his choice. If player lose a game he or she can restart the game

3.3 Analysis Models

3.3.1 Object Model

3.3.2 Domain Lexicon

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 "Map" component and scales all elements in order to adjust to the current screen resolution.

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, "Game Manager" also has the responsibility to judge whether the game is over depending on variables "life" and "enemy List". Additionally, "Game Manager" 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 "towers Info" 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 A star algorithm. The array "enemies" stores all enemy prefabs. When "Enemy Spawner" is created, it will find start node in "Path" component and assign the node to the enemies as starting path node.

Grid Map: Like "Path" component, "Grid Map" comes along with dozens of "Grid Nodes", which are used to mark the status of each small square area. That is to judge whether a Monster could be walk 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.

During the designing of a tower defense game, more than one comprehensive domain had to be examined: gaming, game development, computer graphics

game engines, user interface, content and logic. These examinations had been done both by doing research on existing systems and techniques, by doing some questionnaire, or from our general knowledge. Most of the resulting terms are familiar to gamers, but they still should be reported.

A Game is the program to be developed, all of the system, is called a game. It is the resulting artifact user should

Play/uses, and Player refers to the targeted user of the game, the live person itself.

Game Engine: It is the part of the system which does not interact with the user. However, it is the core of the all system. Does all needed calculations, maintenance of graphical, audible and interactive control objects and ultimately changes in other objects. Ideally, it is completely separated from game logic and game content.

Animation: Animations are the rapid display of a sequence of images of two-dimensional or three dimensional artwork or model positions in order to illustrate an effect.

Collision: A major term in game development, collision refers both the notion of “colliding” agents in the game, and the logical aspect of such collisions. Although it is a major concern of game engines, in tower defense games it is often trivial

Draw: Rendering all drawable objects by the game engine, it allows user to see actually all graphical aspects of the game.

Game Time: Elapsed time in the game logic. System time is not included; it refers the time considered to be spent in the game, not the real time.

Grid: In most systems, not only in games, a screen is separated to little sub-screens, to the smallest units which a user can take action on. It is usually a square, a hexagon or an octagon.

Light: A computer graphics term, light refers both the lightning effect on three dimensional objects and the source of light itself. Light allows user to see rendered objects in the three dimensional space.

Real Time: A genre of strategy games. In such games, all players (either artificial intelligence or human) play the game at the same time, decide and take action simultaneously.

Rotation: A computer graphics term, rotation specifically refers to the action of rotating an image according to a specified origin, to some extent. It is commonly used to illustrate stationary movements in two-dimensional games.

Scaling: A computer graphics term, scaling refers re-sizing an image on the screen.

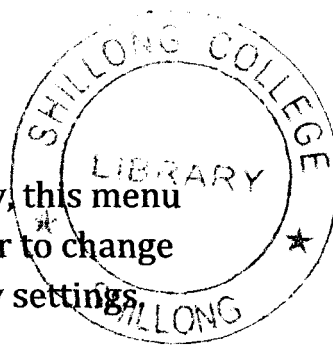
Shadow: A computer graphics term, a shadow is partly illuminated or un-illuminated areas in the three dimensional space. The renderer dynamically creates them.

Sprite: A computer graphics term, sprites are two-dimensional or three-dimensional images or animations that are integrated into a larger screen. Usually, they consist of a number of images or models which are displayed one after another and create an animated visualization.

System Time: Elapsed time in system creates the real time with game time and other environmental variables. System time increases as the calculations of the game increase and force the hardware.

Texture: May also considered as a game content term, textures are two dimensional image files that cover certain areas, such as a background or a model.

Update: A game development term, update refers to applying all changes in the game logic to the game objects, usually dictated by the state of a game.



Options: Reached either from main menu or after starting a play, this menu allows user to change settings of the game. It usually allows user to change graphics settings, sound settings, control settings and game-play settings.

Preparation Screen: A state of a tower defense game which is just before game-play settings. In this state, enemies do not attack, and there is no movement. User prepares his actions, and usually can do what s/he can do during game-play screen.

Tower Info Menu: Visible and reachable on game-play and preparation screen, tower info menu is visible only when a built tower is selected. It gives information about towers status, values, provides a button to sell it or upgrade it.

Game Logic: With game engine, game content, and game control creates the sum of all the game. Game logic is the model of the game, and defines what game should do, and how it should interact with the user. Do calculations and changes that are related with the rules of the game, however is not interested how in the end they will be presented or implemented in the system. Ideally, game logic is completely separated from game content and game engine.

Agent: An agent refers to an object in the game that is capable to take an action, by examining other agents and the environment.

Attack: A common term in gaming, attack refers any type of action which aims to do any kind of damage or hindrance. In tower defense games, attacking refers the whole action of aiming a creature in range, shooting a bullet to it, hit or miss action of a bullet, applying damage and damage type to creatures related.

Attack Range: Distance, which a tower can aim and shoot to, starts from its center.

Attack Speed: Attack speed is a value of an attacking agent, which decides the minimum time between two following attacks

X Width: A graphical term, defines the X coordinate of a drawable object in the space.

Y Length: A graphical term, defines the Y coordinate of a drawable object in the space.

Z Depth: A graphical term, defines the Z coordinate of a drawable object in the space.

Effect: Effects are all kind of experience enrichment content, either graphical or audible.

Game Control: This part of the system deals with user interaction and its effects on both game engine and game logic

Building Menu: Visible and reachable on game-play and preparation screen on a different panel; building menu allows player to browse towers (s) he can build and choose a tower to build.

Click Area: A user interface term, click areas are defined to specific buttons and correspond to a certain area, which allows system to decide when a button is clicked.

Game-play Screen: A state of a tower defense game where game runs and enemies attack to the player's character. The most dynamic of a game, this part can be sums up the conventional meaning of playing with the "Preparation Screen".

Main Menu: The first user interactive state of a game. Unlike other computer programs, nearly all computer games start with a main menu and allows user to navigate through states by using large buttons.

Map Editor: A sub-system in the game, level editors allows an interface to users add contents (specifically, a map) to an existing game.

Environment: A term related to game logic and artificial intelligence, environment is the sum of all objects that are not capable to take an action and is in game logic itself.

Game Artificial Intelligence: A sub discipline of artificial intelligence, game AI is very different from academic discipline of artificial intelligence. Even to some, it is wrong to state that artificial intelligence is used in games. Artificial intelligence in games aims to achieve an enjoyable and competitive agent behavior in the game.

Game Level: In tower defense games, a game level starts when player finishes preparation screen and ends when game-play screen finishes either with success, or failures. A resulting success allows user to pass a level and start a new one. Usually, all game levels consist of a certain type of enemy.

Ground: Remaining parts of the map, excluding paths, home base and enemy base. Consists the majority of the map, and user is allowed to build on such areas.

Gold: Also referred as “coin” or “money”, it stands for the resource itself that is used in the game. It comes from the notion of playing a game in the old ages.

Hit: A common term in gaming, a hit means a successful attack. In tower defense games, it is one of possible ending states of a sent bullet; a hit means a bullet has successfully met with its target.

Hit Damage: A damage type which is common in tower defense games, “hit damage” type attacks only affect the targeted creature and decrease its hit point.

Hit Point: A common term in gaming, hit point is a value that decides how much damage an agent can take before being eliminated. In tower defense games, only enemy creatures have hit points.

Home Base: A tower defense game term, referring to the target point of enemy creatures, which is the “home” of the player.

Building: The most common and needed player action in tower defense games meaning creating fortifications (usually towers) is that preventing enemy.

Bullet: Although word of bullet mean a single bullet in general gaming usage, tower defense games refers bullets as agents that are sent by the towers to the enemy creature.

Bullet Speed: Bullet speed is a value of an attacking agent, and the bullet agent, which decides how much space a sent bullet will parse in a unit time.

Continuous Damage: A damage type which is not common in tower defense games, "continuous damage" type attacks continue to decrease applied creatures' hit point even after it is applied to the target. Fire damages are an example to that.

Creature: Also referred as "monster" or "soldier", creatures are the tools of enemy to win the game in tower defense games. They move through the path, and wish to survive attacks from towers. They have no capacity to attack or to show any sentient resistance.

Creature Speed: Creature speed is a value of an enemy creature agent, which decides how fast it can move through the map.

Damage Type: Damage types are values that are stored in attacking towers and their bullets, to apply a certain type of damage. Damage types change the applied creatures, any possible hindrance to these creatures, damage that applies to these creatures.

Enemy: Opponent of the player. In gaming, it corresponds to all possible opponents and competitions, however tower defense games are single player, with very strict rules on player's and opponents role. Thus, enemy is the opponent of the player, which aims to finish the lives of the player.

Enemy Base: A tower defense game term, referring to the starting point of enemy creatures.

Miss: A common term in gaming, a hit means an unsuccessful attack. In tower defense games, it is one of possible ending states of a sent bullet. A miss means a bullet has not successfully met with its target.

Path finding: Algorithms and methods to define a path for game agents. Part of game artificial intelligence, path finding is a major concern in game development. In tower defense games, however, it is a trivial task.

Path: A static way for enemy creatures to move through screen, from enemy base to home base, defined by waypoints. User cannot build towers on paths.

Profile: Sometimes referred as “account”, a profile is recordings of a specific player. A player may have more than one profile, and can separate distinct plays from one another.

Slow Damage: A damage type which is common in tower defense games, “slow damage” type attacks decrease the speed of applied creatures. Cold damages are a common example to that.

Splash Damage: A damage type which is common in tower defense games, “splash damage” type attacks not only affects to aimed target, but other creatures in the range as well.

Splash Range: A term related to splash damage, splash range value is the distance that splashed bullet damage can reach at most.

Tower: Basic tool that player has, towers are the foundations of a tower defense game. They have a price (usually in terms of gold or coin), needs a clear area to build on, has a range, an attack type, attack speed and damage value. They can be upgraded to increase one or more of these attributes.

Tower Level: State of a built tower, starting from zero to either infinity or a magic number. As it increases, a tower becomes more effective. Tower levels are increased by upgrading.

Treasury: Also referred as “bank”, it stands for the resource itself that is used in the game. It comes from the notion of having gold as the resource.

Importer: In game systems, objects that are related to the game logic are subject to change, either during development or after release. It is also considered as a good practice of game development to separate game entities from the system as much as possible, allowing variations and rapid modification. An importer deals with such external files that keep needed parameters for game objects. They usually read a file, and allow game logic to create related objects accordingly.

Interest: Some tower defense games aim to reward users by not spending many resources and try to motivate others to challenge them to optimize resources. By having an interest rate, the gold player stores in her/his treasury will be on his interest.

Kill: A common term in gaming, killing means eliminating a creature of opponent, by decreasing its hit point to zero or below.

"Letter of Requisition": A term usually used interchangeably used with upgrade or research point in tower defense games. A letter of requisition is a point which plays gains after passing a number of levels/turns. S/he can use these points to

Life: Interchangeably used to define the notion of "life", or the number of "lives" a player has. A game starts with a number of lives, and ends when player lost all her/his lives. Extra life can be gained by spending gold or by having specific traits.

Manhattan Distance: A term related to game logic and artificial intelligence, Manhattan distance is the shortest path between two points by following only moving on one coordinate at a time.

Map: Sum of all grids, where all game runs and opponents compete with each other. A map consists of home base, enemy base and path. Towers and creatures are placed (and move) on the map; however is not part of the map.

Upgrade: A player action, which allows him to make more benefit of a tower by spending some resource (gold or coin). An upgraded tower has an increased level, and usually further upgrades are more expensive. By upgrading, tower may simply have increased attack range, speed or damage; or may evolve into something else completely.

Waypoint: Waypoints define a set of points which defines a path for the enemy creatures. It starts from enemy base, and is aimed to home base.

Sell: A common gaming term, sell refers to the action of sacrificing a possessed agent of the player to gain some resource on it. In tower defense games, selling refers removing a built tower from a map and gaining some degree of gold from this action.

Game Content: Part of a game system, which includes all non-implementation and non-logical (in the end, that is not related to the source code directly) features. Textures, sprites, animations, sounds, music themselves (not their implementation) are the most known examples of game content. In addition, aspects that are related with the game logic, but not the playability, are also fall into this category.

“Kill Streak”: Number of rapid kills that a player has reached without any miss, one after another.

Terrain: Terrain is the background image for the map, which is displayed on ground grids. It is read from external content files and displayed by the game engine

3.4 Dynamic Models

3.4.1 State Charts

Enemy State

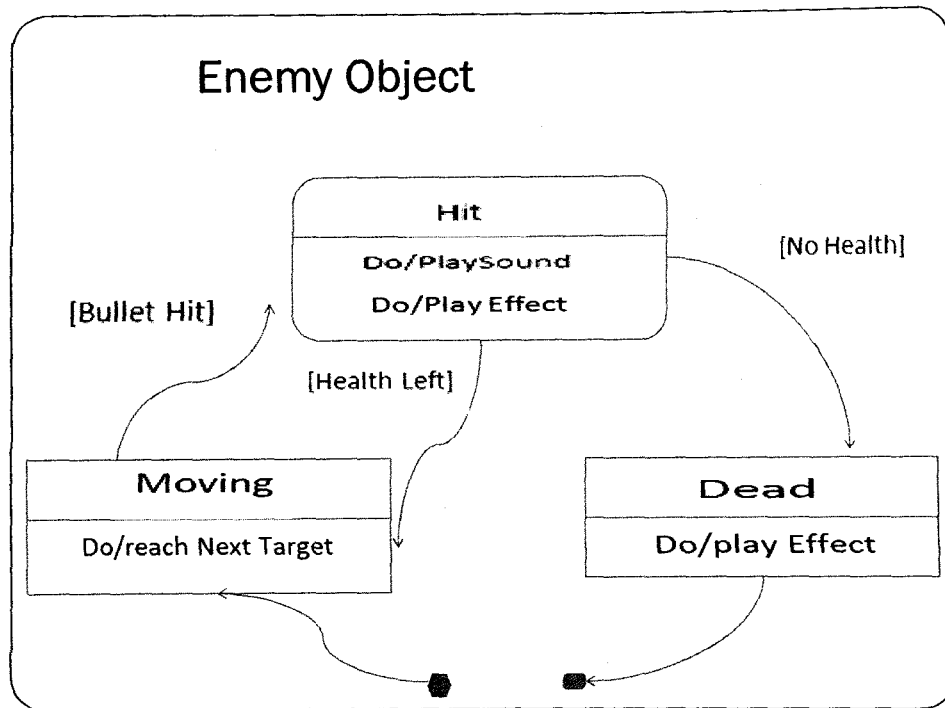


Figure 4.1

In figure 4.1, states of Enemy object are diagrammed. There are three states of Enemy object. When Enemy is in Moving state, it updates its position by trying to reaching next target, which is the next turning point in the path. If in Moving state, a bullet hits the Enemy, Enemy passes to Hit state, plays a sound and visual effect and checks if there is any health points left, if any left it returns to Moving state; if not it proceeds to Dead state. In Dead state, it plays an explosion effect and finishes the state machine.

Bullet State

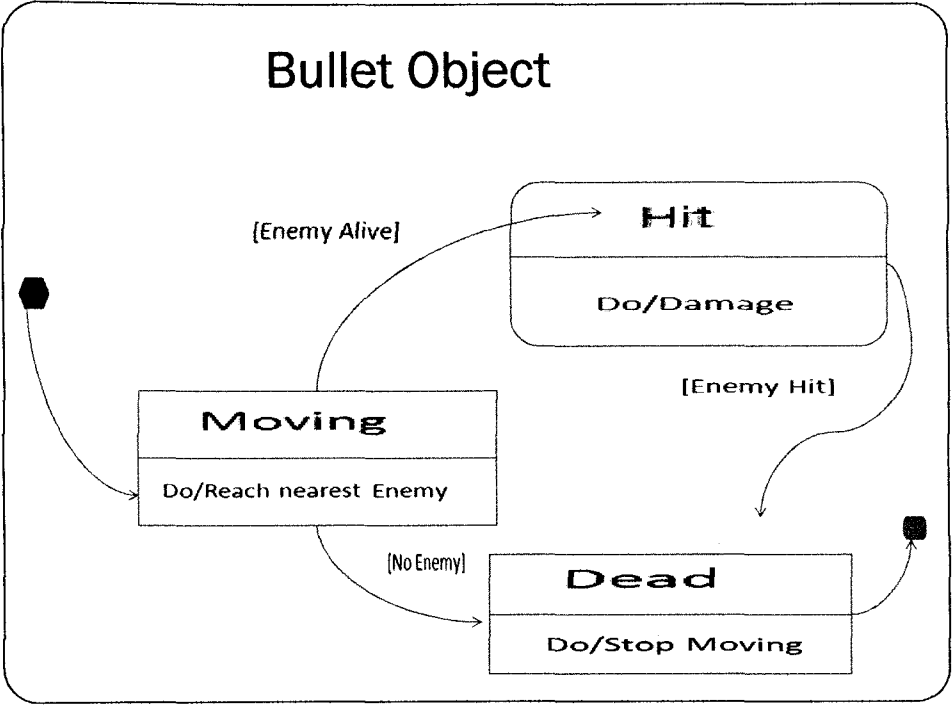


Figure 4.2

In figure 4.2, the states of Bullet object are diagrammed. Bullet starts in Moving state, it tries to reach the nearest enemy. When the enemy has health when Bullet reaches the position when enemy is in when the Bullet is fired; there are two conditions possible. If the Enemy is still alive, Bullet passes to Hit state and damages the Enemy, then proceeds to Dead state. If Enemy is not alive –killed by another Bullet-, Bullet jumps to Dead state. Dead state is the final state and Bullet finishes the state machine.

Tower State

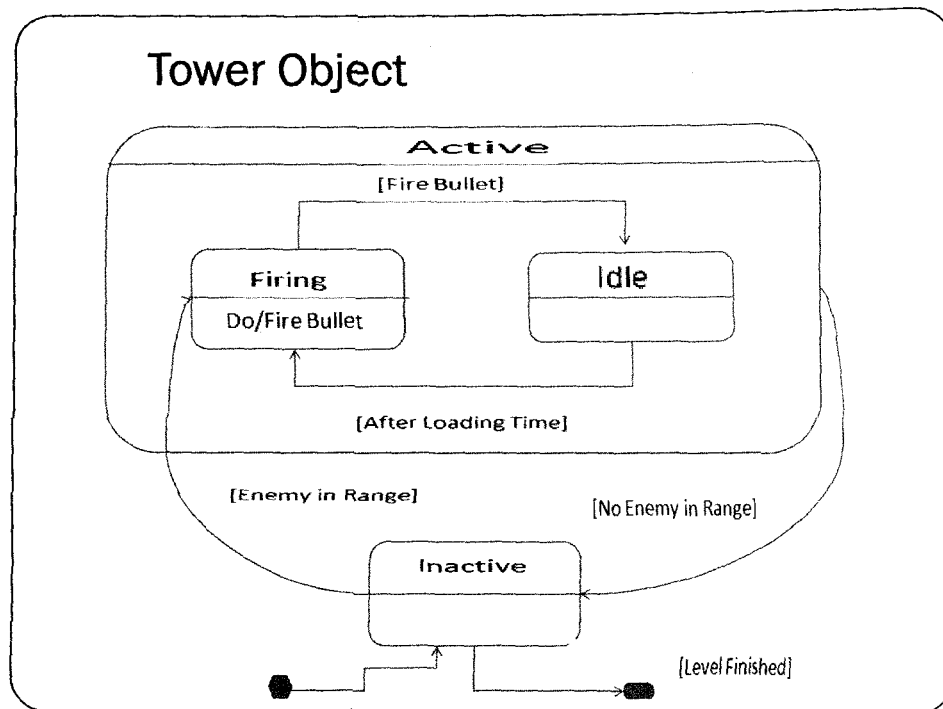


Figure 4.3, states of Tower object

In figure 4.3, states of Tower object have been diagrammed. There are two main states of Tower object; the Active and Inactive. When a level started, Tower starts in Inactive state. When there is an enemy in range it passes to the Firing state, which is a sub-state of Active state. In Firing state, Tower object calls fireBullet method and, whenever it calls fireBullet method, it passes to the Idle state. In Idle state, it waits for a “loading time”, which is predetermined by the properties of the Tower, and returns to the Firing state. Whenever in Firing or Idle state- i.e. in Active state- if there is no enemy in the range, Tower returns to the Inactive state. In Inactive state, when the level finishes, Tower exits state machine.

Game Object

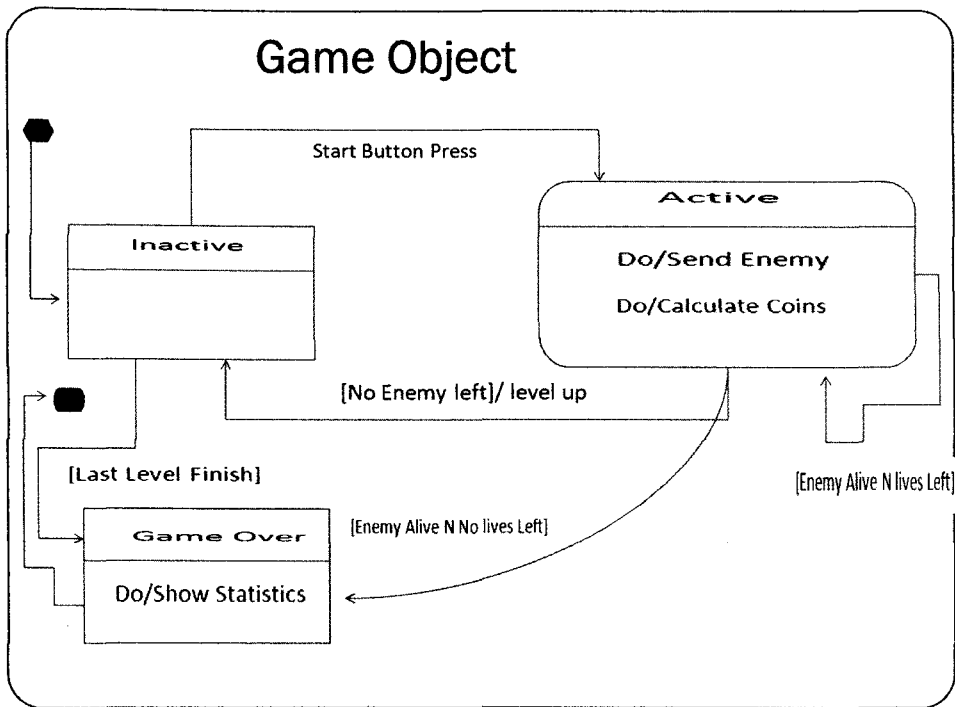
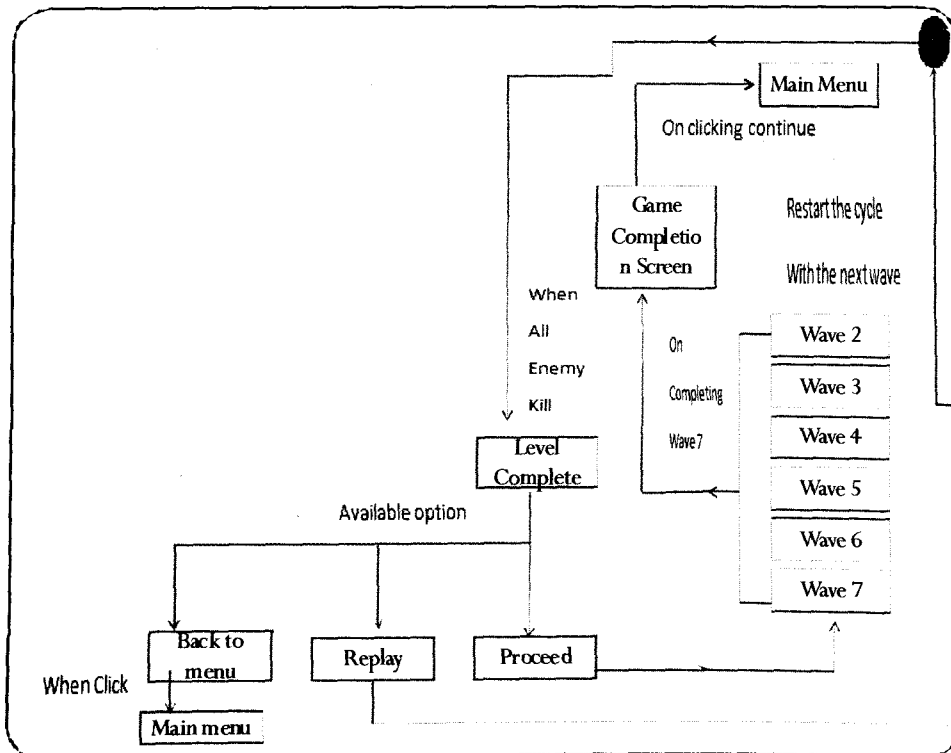
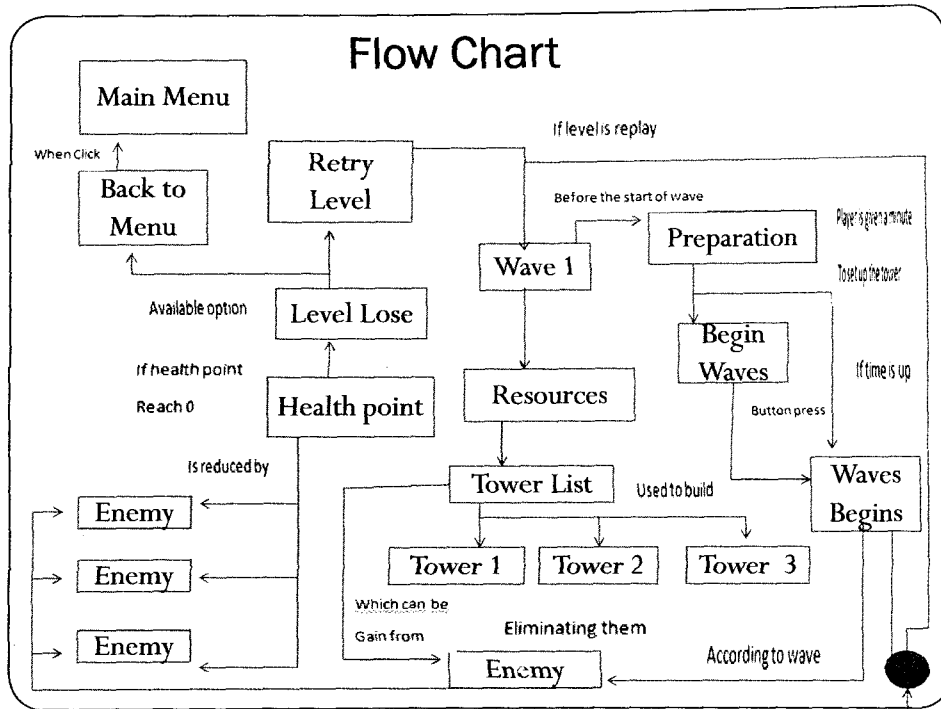


Figure 4.6, states of the Game object

In figure 4.6, states of the Game object are diagrammed. Game object starts in Inactive state. When start button is pressed the Game passes to Active state. In Active state, Game sends the enemies in pre-defined numbers according to level, calculates the coins and lives. If last enemy is dead and there are lives left in Active state, Game passes to Inactive state and increments level. If there are enemies alive and there are no lives left in Active state or last level is reached in Inactive state, Game passes to GameOver state. In GameOver state Game shows the statistics and finishes the state machine.

3.4.2 Flow chart

Flow Chart



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 four 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 four attributes: type, power, attack time, attack area, and cost. "Type" determines the unique character of towers. Attributes "power", "attack Time", "attack Area" 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 two levels: low and high. 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 four tower types:

Strom Tower: This tower fires thunderbolt projectiles, these projectiles have a chance to apply a storm debuff to the target. This debuff will stun the target for an amount of seconds.



Fire Tower: This tower shoots fire projectiles, these projectiles have a chance to apply a fire debuff. The fire debuff is a damage over time effect. When applied to a monster it deals damage every x amount of seconds, as long as the debuff is applied to a target.



Frost Tower: Frost 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 splash damage to enemies when they step in poison. 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.



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 4 types.

Red: This monster takes less damage from fire towers, and is immune to fire debuffs.

Blue: This monster takes less damage from frost towers, and is immune to frost debuffs.

Purple: This monster takes less damage from storm towers, and is immune to storm debuffs.

Green: This monster takes less damage from poison towers, and is immune to poison debuffs.

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:

$$H.P = \frac{ATK1*FRQ1*ELP1}{speed} + \frac{ATK2*FRQ2*ELP2}{speed} + \dots \quad 4.1$$

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:

$$HP * Speed = FRQ1_ATK1_ELP1 + FRQ2_ATK2_ELP2 + \dots \quad 4.2$$

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:

$$SE = HP * Speed \quad 4.3$$

For a tower:

$$ST = FRQ * ATK * ELP = FRQ * ATK * Radius * \sqrt{2}$$

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:

$$Difficult = SE * \left(Num \frac{Life}{Harm} \right)$$

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

$$Difficulty < ST1 + ST2 + ST3 + \dots \quad 4.6$$

$$Cost1 + Cost2 + Cost3 + \dots < Money \dots \quad 4.7$$

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 $_t$ 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.

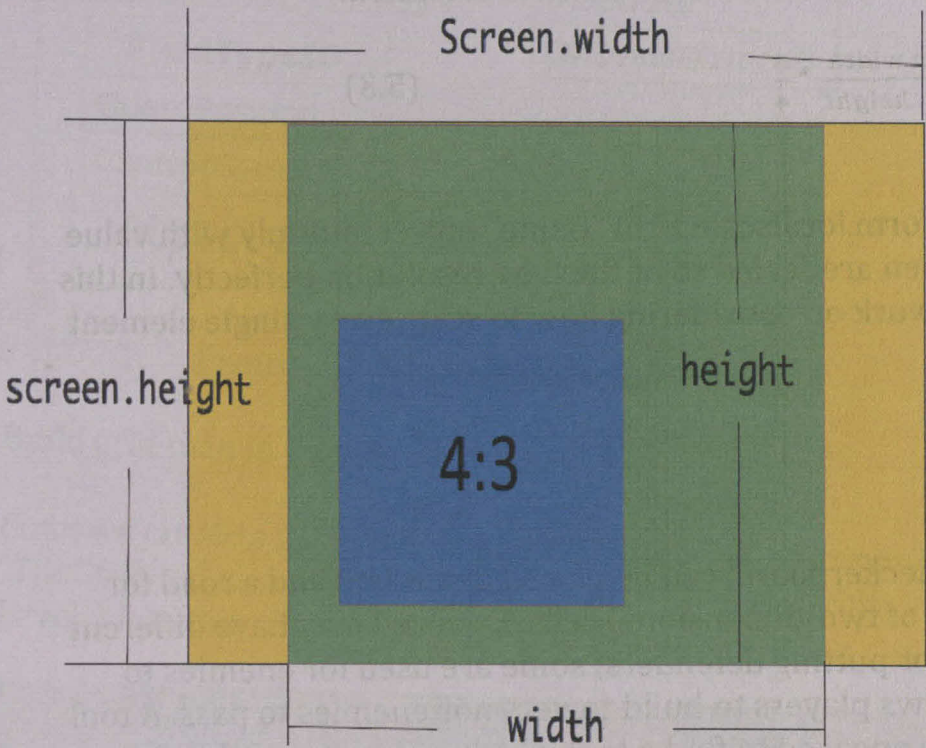


Figure 5.1: The new size after being scaled

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

$$\frac{\text{Width}}{\text{height}} = \frac{4}{3} \quad \text{And height} = \text{screen. Height We get:}$$

$$\text{width} = \frac{4}{3} * \text{height} = \frac{4}{3} * \text{screen. height} \quad (5.2)$$

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

$$\text{rate} = \frac{\text{screen.width}}{\text{screen.height}} * \frac{3}{4} \quad (5.3)$$

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.

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 Tile . According to real situation, each Tile only has two conditions, can put a defender or cannot put a defender.

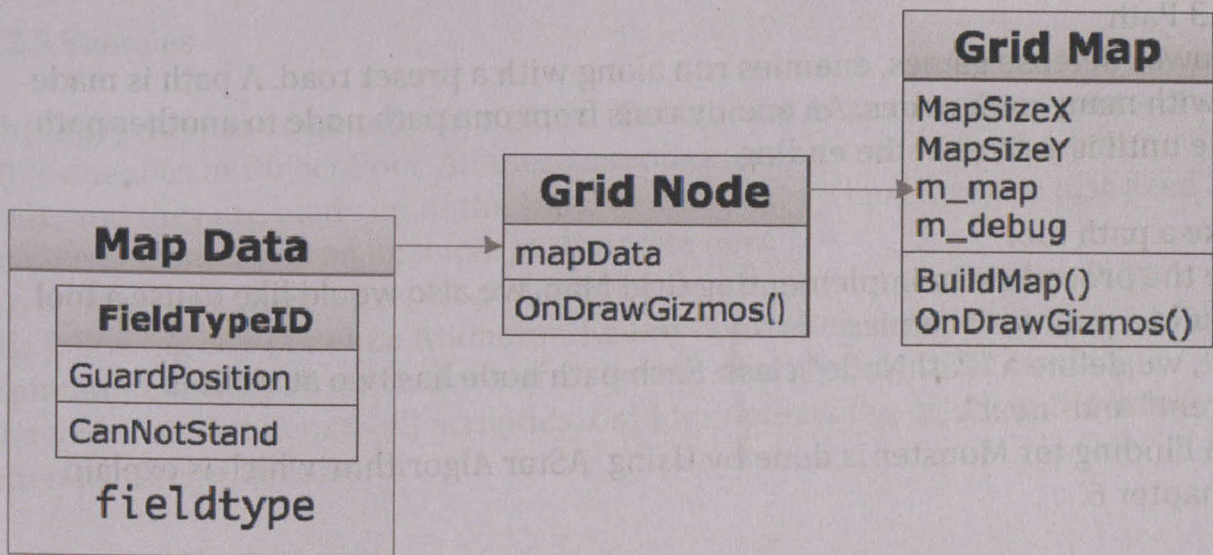


Figure 5.2: Tile Grid class and Tile class

Build grid map in Editor

First, we create a game object named "TILES" in Hierarchy tab, and then drag "TileGrids.cs" as a script component on it. Click on debug attribute and run `Level()` function, we will see our map into many cells.

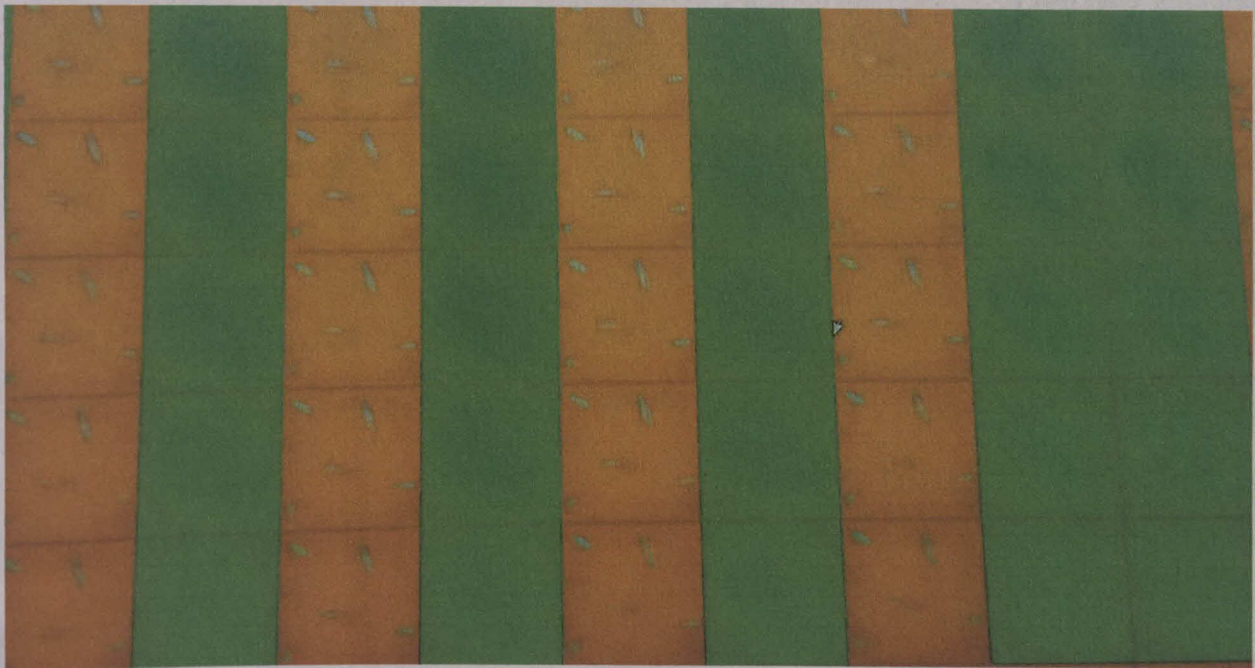


Figure 5.4: Available nodes

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".

Path Finding for Monster is done by Using AStar Algorithm which is explain in chapter 6.

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.

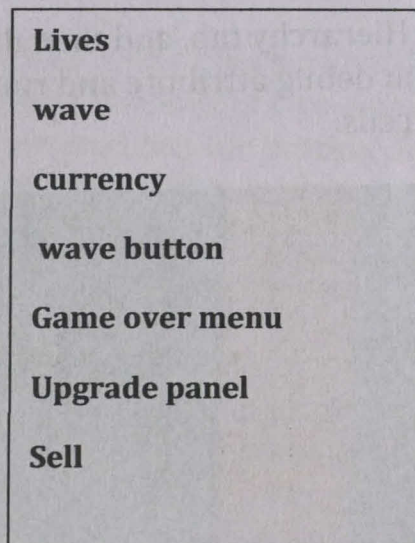


Figure 5.7: Build grid map

5.2.5 Enemies

There are 4 types of enemies in a game. Enemy List is an array that saves alive enemies in Object Pool .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

First, 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.

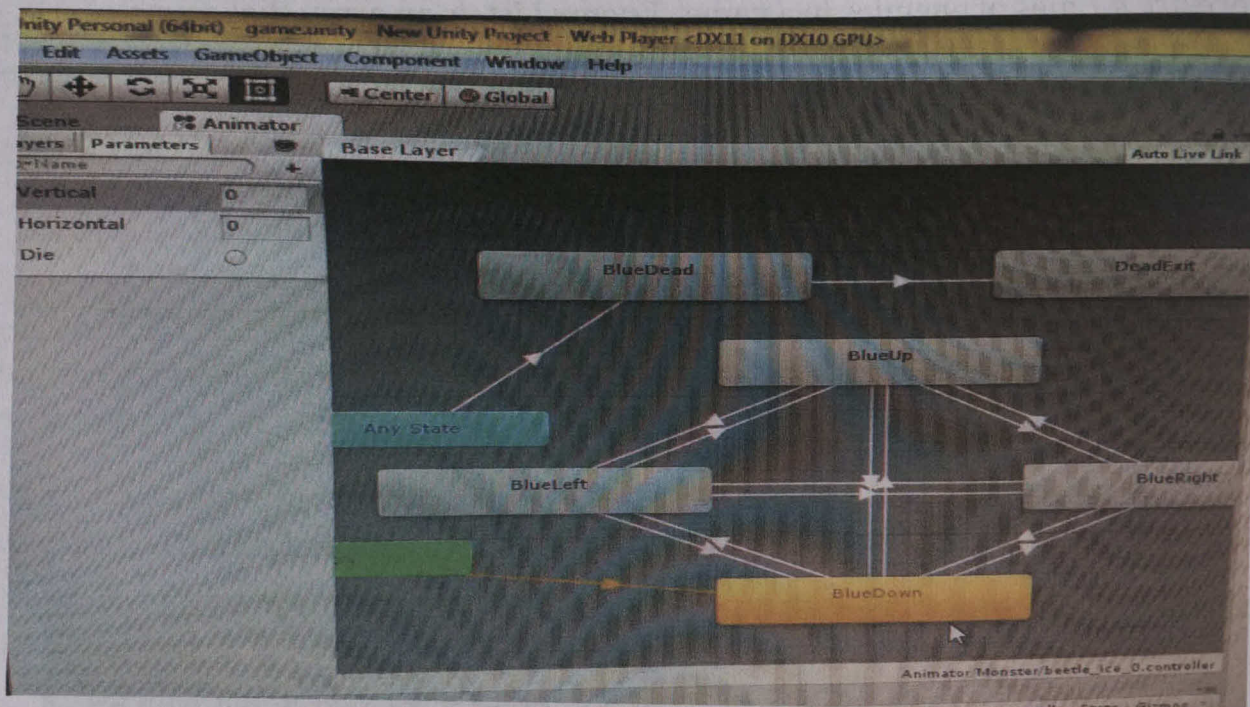
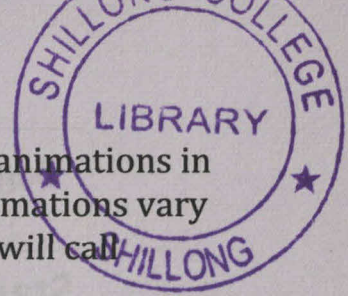


Figure 5.11: Animator



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.



Then, make all animation clips one by 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.

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.

EnemyController
m_life m_speed m_point m_harm m_currentNode m_state healthbar
Start() SetDamage() SlowDown() FixedUpdate() RotateTo() MoveTo() ClickItem() DestroyGameObject()

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.

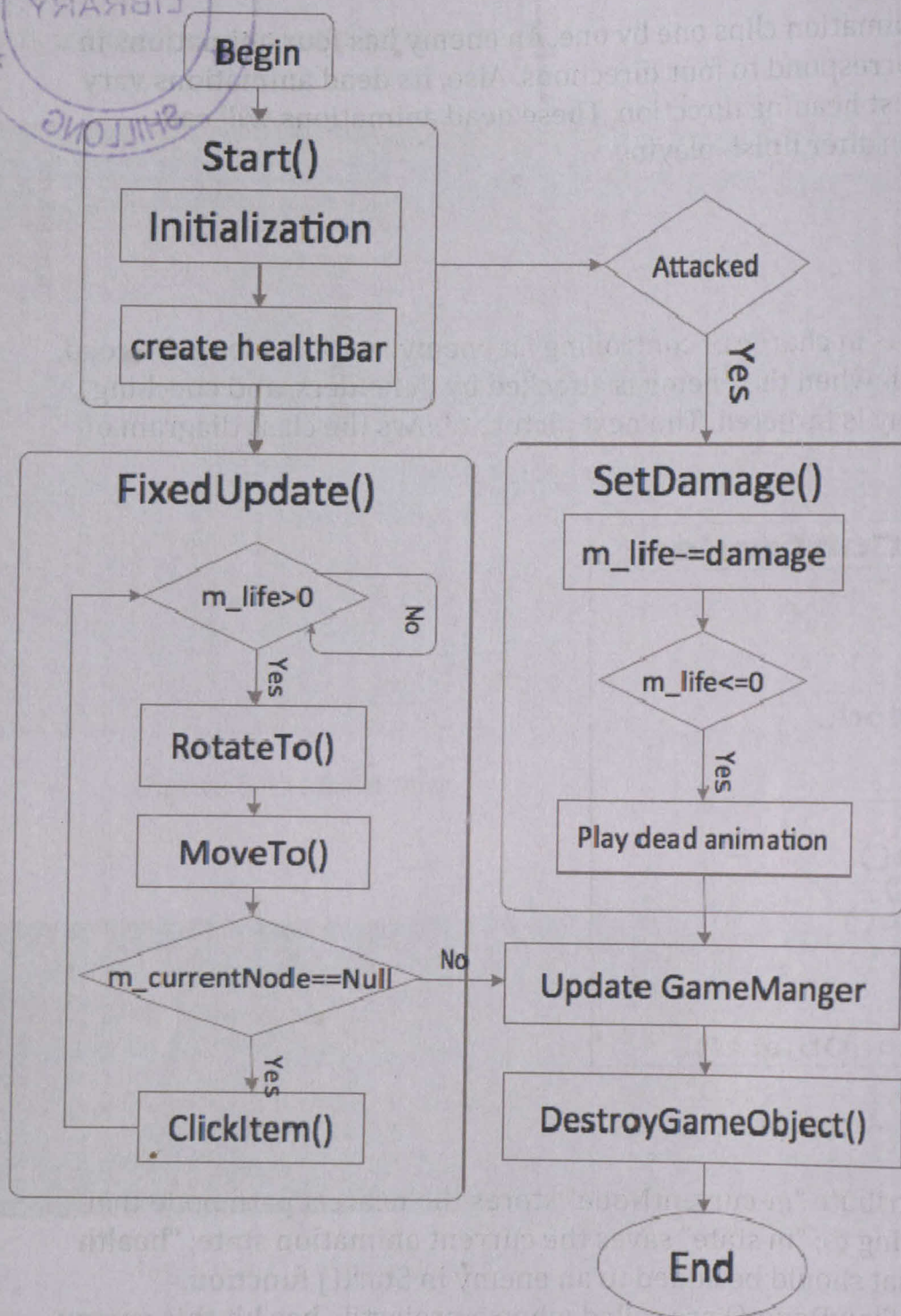


Figure 5.13: Animator

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: "DefenderController", "attack", "attack area", "upgrade button" and "sell button". "DefenderController" 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.

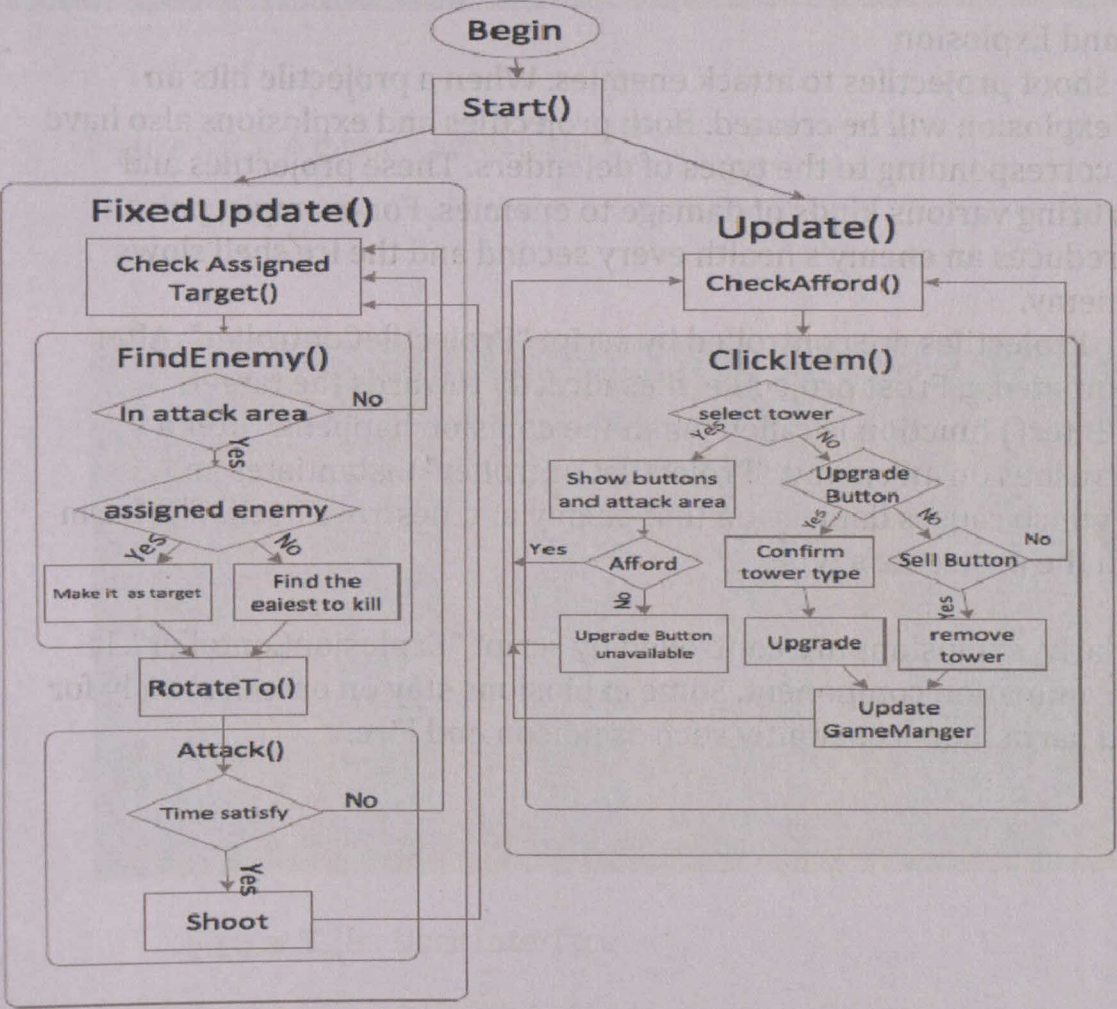


Figure 5.23: The workflow of DefenderController

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 four 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 Frost projectile 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 "ExplosionController". It also has an animation component. Some explosions stay on enemies' body for a while and harm them constantly such as poison and Fire.

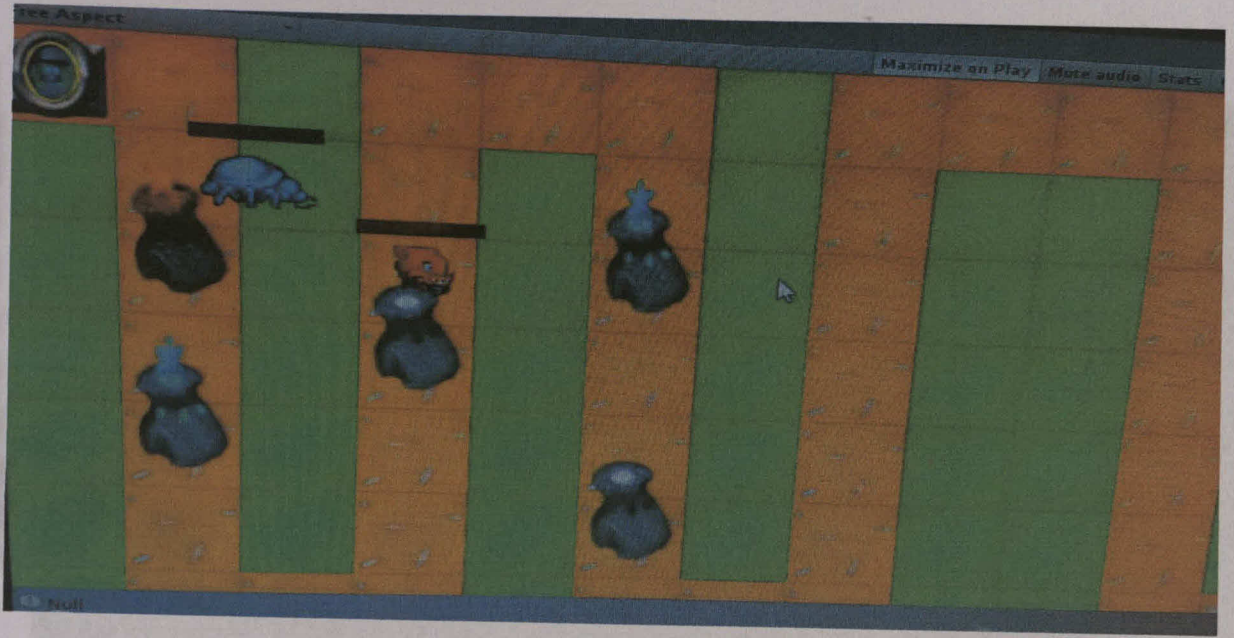


Figure 5.25: A game's screen shot

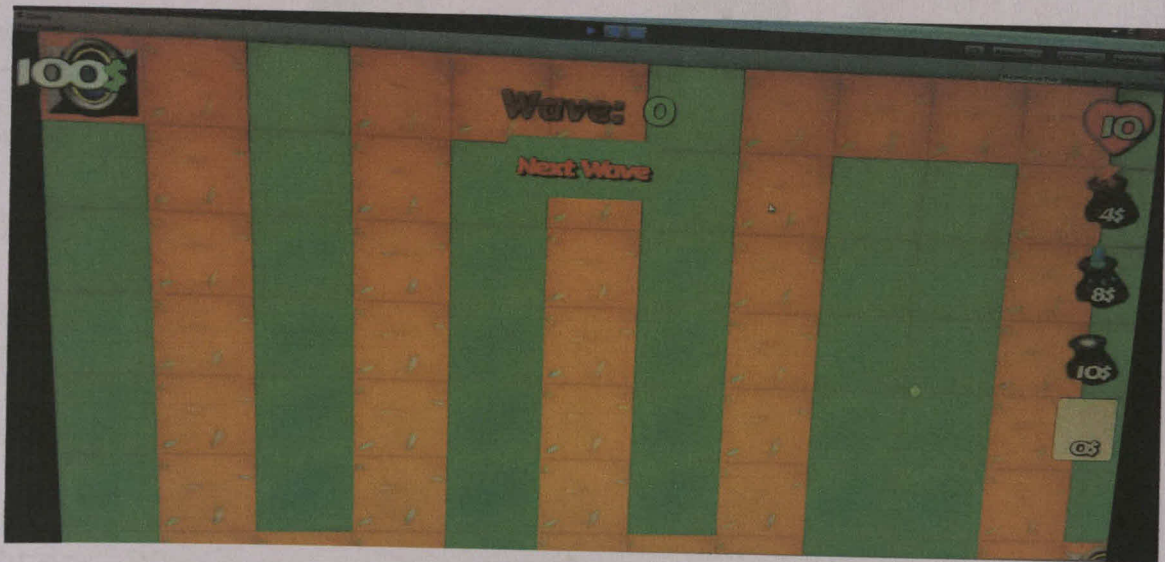
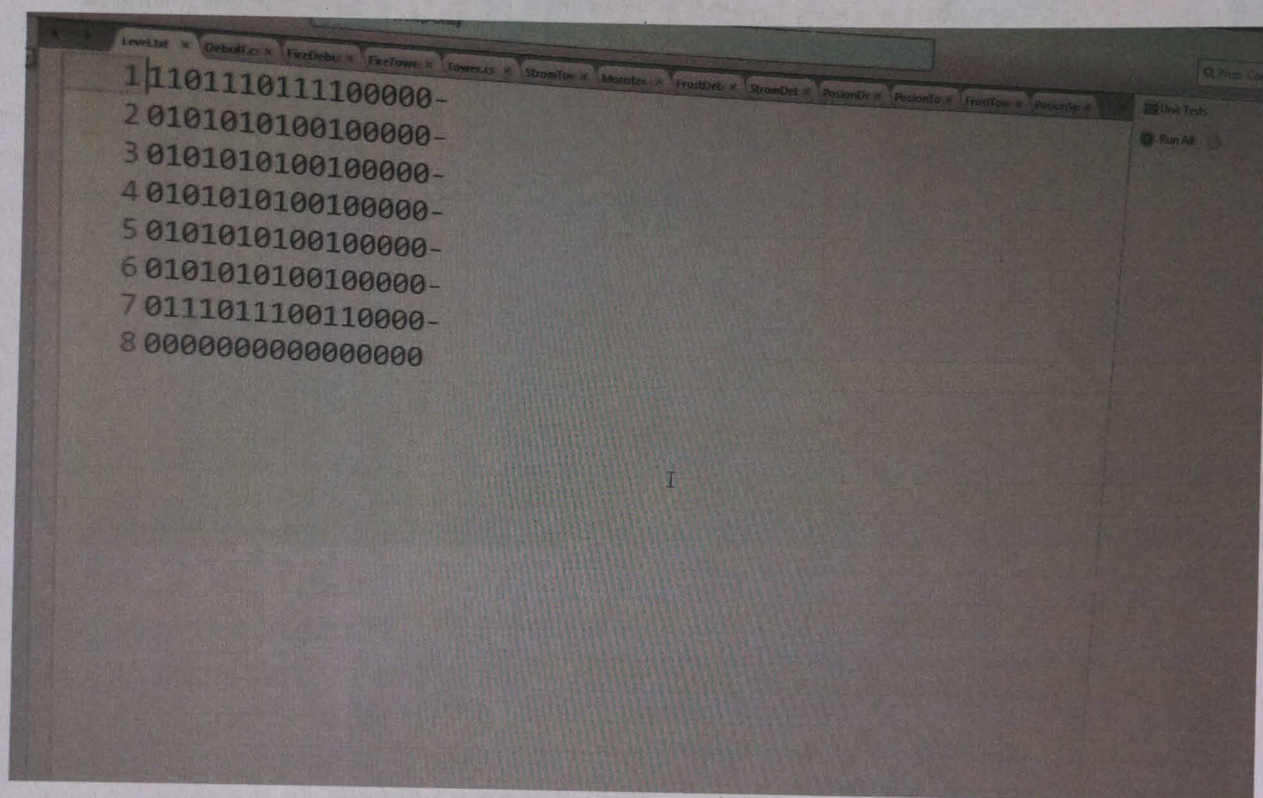


Figure 5.25 : User Interface

5.3 Level or Map



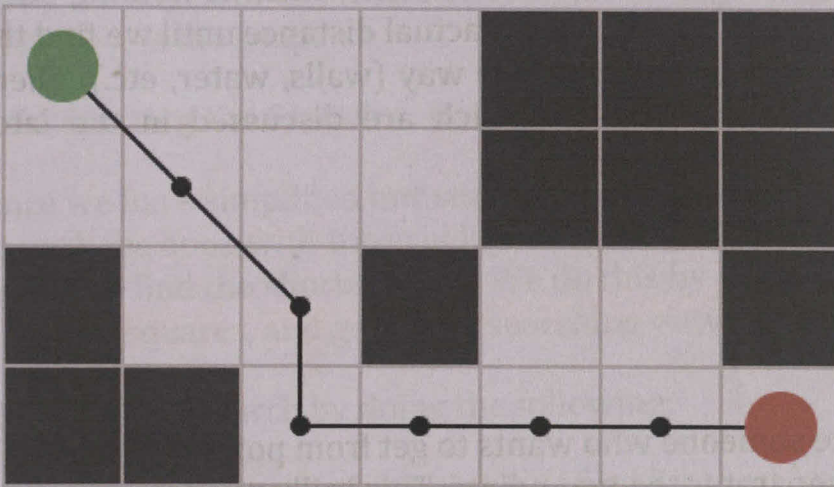
Chapter 6

A* (A-star)

6.1 Motivation

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.

We can consider a 2D Grid having several obstacles and we start from a source cell (coloured green below) to reach towards a goal cell (coloured red below)



6.2 What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals or Square box.

6.3 Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

6.4 Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value-' f ' which is a parameter equal to the sum of two other parameters - ' g ' and ' h '. At each step it picks the node/cell having the lowest ' f ', and processes that node cell. We define ' g ' and ' h ' as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ' h ' which are discussed in the later sections.

6.5 Introduction

6.5.1 The Search Area

Let's assume that we have someone who wants to get from point A to point B. Let's assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.

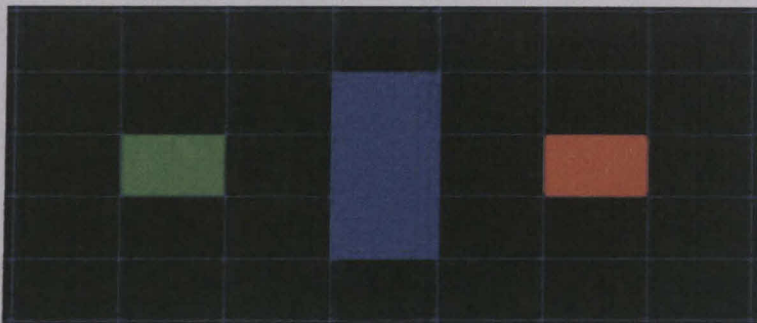


Figure 1

The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached. These center points are called "nodes". When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

6.5.2 Starting the Search

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

1. Begin at the starting point A and add it to an "open list" of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its "parent square". This parent square stuff is important when we want to trace our path. It will be explained more later.
3. Drop the starting square A from your open list, and add it to a "closed list" of squares that you don't need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.

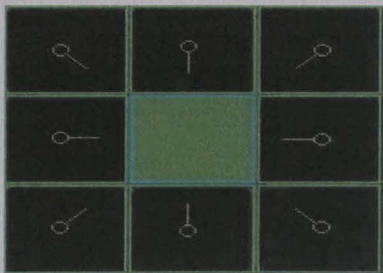


Figure 2

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

6.5.3 Path Scoring

The key to determining which squares to use when figuring out the path is the following equation:

$$F = G + H$$

Where

- G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
- H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let's look more closely at how we calculate the equation.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don't be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity's sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn't just because we are dumb and don't like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don't use short cuts like these.

Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can't cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic."

Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. Want to know more? You can find equations and additional notes on heuristics.

F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

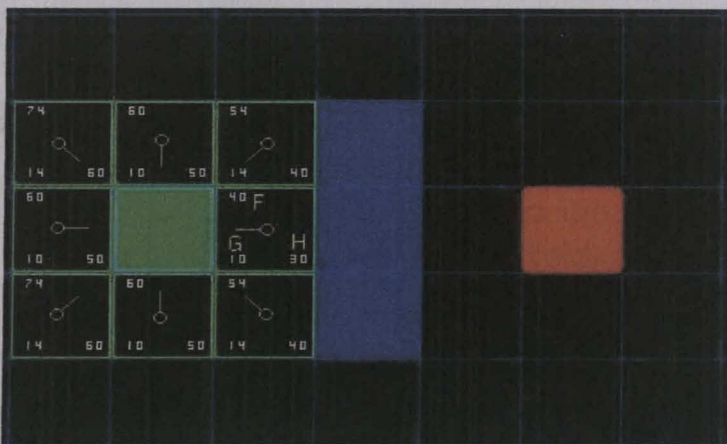


Figure 3

So let's look at some of these squares. In the square with the letters in it, $G = 10$. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

6.5.4 Continuing the Search

To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

- 4) Drop it from the open list and add it to the closed list.
- 5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the "parent" of the new squares.
- 6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don't do anything.

On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected

square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let's see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list. Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

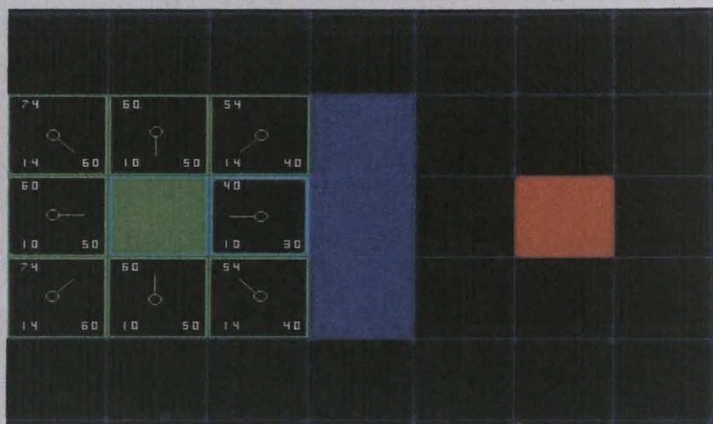


Figure 4

First, we drop it from our open list and add it to our closed list (that's why it's now highlighted in blue). Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That's on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let's look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram.

It's more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square.

When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don't change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn't really matter. For the purposes of speed, it can be faster to choose the last one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn't really matter. (Differing treatment of ties is why two versions of A* may find different paths of equal length.)

So let's choose the one just below, and to the right of the starting square, as is shown in the following illustration.

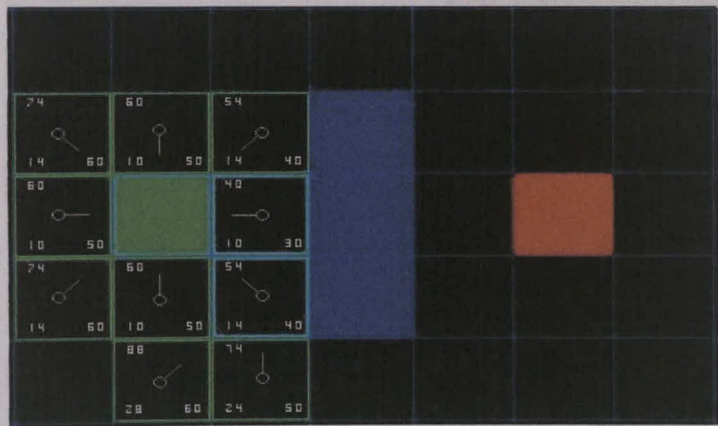


Figure 5

This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can't get to that square directly from the current square without cutting across the corner of the nearby wall.

You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.)

That leaves five other squares. The other two squares below the current square aren't already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we're done and ready to check the next square on our open list.

We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

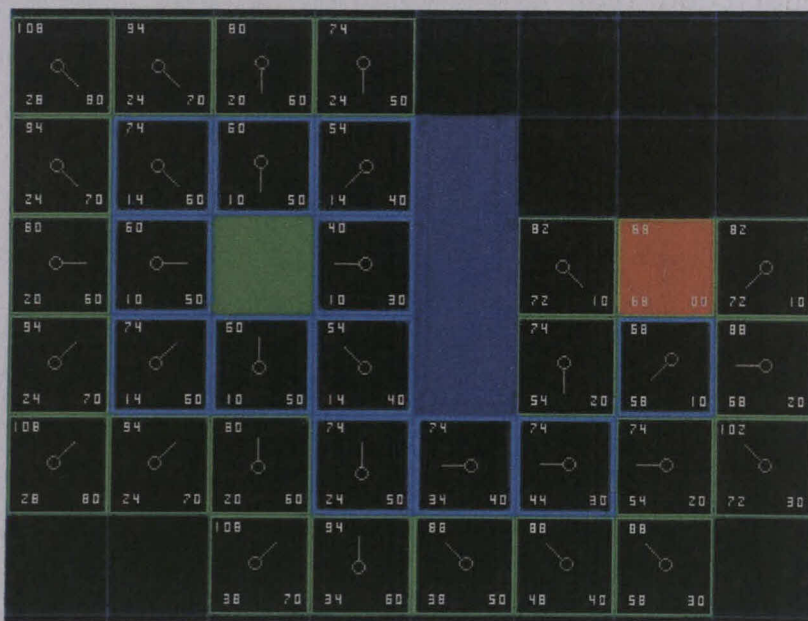
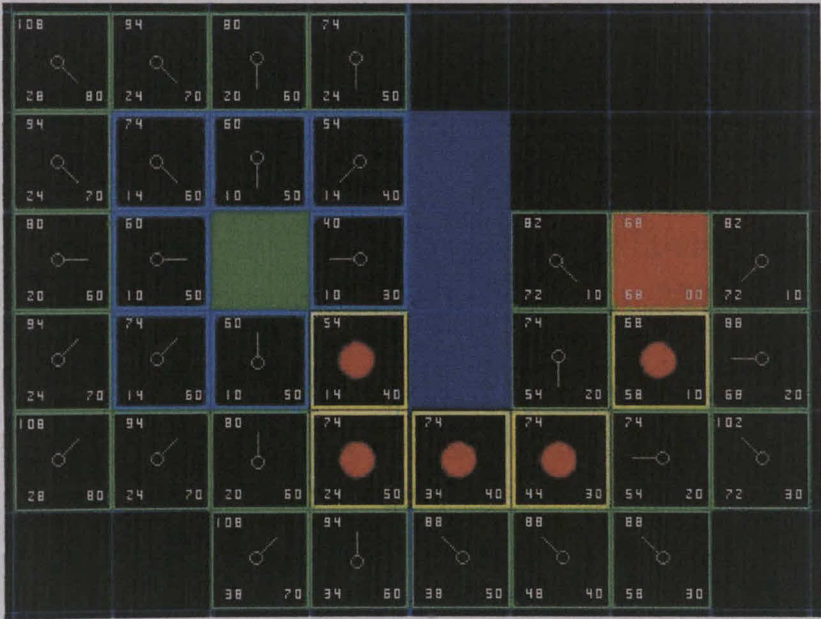


Figure 6

Note that the parent square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it.

This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the parent was switched and the G and F scores were recalculated. While this change doesn't seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to your target.

So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that's your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.



6.6 Algorithm

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square ...

- If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
- If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
- If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

- Add the target square to the closed list, in which case the path has been found or
- Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is the path.

Chapter 7

Conclusion

The project has chosen the platform as PCs for a better performance and improves graphics. Besides that, the design of towers, enemies and surroundings are studied in detail to catch an atmosphere that the gamer would like to return again. The algorithm of the game logic was hard to design since it must be a balance between hardcore and casual games. In this point, existing games are studied and their mechanisms are combined in a way that developer team believed that it is fun to play Tower Defense Game. Although it was a Greenfield engineering practice, many games had set some conventions and those must be followed without undergoing a total cloning operation.

The design process was an unfamiliar one, it was the first time practicing a strictly regulated design process for each. The programming experience was almost always about algorithms or learning key concepts. Because of these reasons, conducting a project from the scratch and following a development cycle were new experiences. The biggest challenge, in these terms, was to understand the concepts and applying them in implementation simultaneously, without any prior experience. It would be more convenient to exemplify the key concepts with real life project (coded and run).

The project is left incomplete in some parts, like the loading for Game Menu and Upgrading of tower and sound . Even though the infrastructure for map editor is ready, due to deadline issues, graphical interface (view part of that subsystem) is left to be finished later. By the same reasons, profile functionality is left unfinished. Because the project was done purely educational reasons, we believe these can be accomplished with a little more time.

7.1 References

- Bruegge, Bernd. Allen H. Dutoit. Object Oriented Software Engineering: Using UML, Patterns, and Java. Pearson Prentice Hall. NJ. 2004. A canon textbook of object oriented software engineering.
- <http://www.freewebarcade.com/tower-defense-games.php> A portal of free web based tower defense games

- www.gamedev.net the largest game development online community, both for beginners and seasoned programmers.
- Programming Manual by Microsoft
- <http://java.sun.com/blueprints/patterns/MVC.html> Application of Model View Controller pattern in Java

7.2 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,
- [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 suppl. Artificial Intelligence.
- [7] Jin, X. (2013). Unity3D Mobile Development. Qing Hua.
- [8] Johnson, L. (2012). Clash of clans review.
- [9] Neighbors, J. (1980). Software product lines.
- [10] Pang, M. (2014). Unity4.3 Development. Qing Hua.
- [11] Phillipa Avery, Julian Togelius, E. A. (2012). Computational intelligence and tower defence games.

- [12] Rummell, P. A. (2013). Adaptive ai to play tower defense game.
- [13] Unity (2013). Introduction of unity. <http://unity3d.com/unity>.
- [14] Unity (2016). Introduction of unity. <http://unity2d.com/unity>.



LevelManager.cs

using UnityEngine;

using System.Collections.Generic;

using System;

public class LevelManager : Singleton <LevelManager>

{

[SerializeField]

private GameObject[] tilePrebas;

[SerializeField]

private CameraMovement cameraMovement;

[SerializeField]

private Transform map;

private Point blueSpawn, redSpawn;

[SerializeField]

private GameObject bluePortalPrefab;

[SerializeField]

private GameObject redPortalPrefab;

public Portal BluePortal { get; set; }

private Point mapSize;

private Stack<Node> path;

public Stack<Node> Path

{

 get

 {

 if (path == null)

 {

 GeneratePath ();

 }

 return new Stack<Node> (new Stack<Node> (path));

 }

}

public Point BlueSpawn

{

 get

 {

 return blueSpawn;

 }

```

int mapSizeX = mapData[0].ToCharArray().Length;
int mapSizeY = mapData.Length;
Vector3 maxTile = Vector3.zero;
Vector3 worldStart =
Camera.main.ScreenToWorldPoint(new Vector3(0, Screen.height));
    for (int y = 0; y < mapSizeY; y++)
    {
        char[] newTile = mapData[y].ToCharArray();
        for (int x = 0; x < mapSizeX; x++)
        {
            PlaceTile (newTile[x].ToString(), x, y, worldStart);
        }
    }
maxTile =
Tiles [new Point (mapSizeX - 1, mapSizeY - 1)].transform.position;
cameraMovement.SetLimits (new Vector3 (maxTile.x + TileSize,
                                         maxTile.y - TileSize));

SpawnPortals ();
}

private void PlaceTile (string tileType, int x, int y, Vector3 worldStart)
{
    int tileIndex = int.Parse (tileType);

```

```
public Dictionary<Point, TileGrid> Tiles { get; set; }
```

```
public float TileSize
```

```
{
```

```
    get { return tilePrebas[0].GetComponent<SpriteRenderer>  
() .sprite.bounds.size.x; }
```

```
}
```

```
void Start ()
```

```
{
```

```
    Level ();
```

```
}
```

```
void Update ()
```

```
{
```

```
}
```

```
private void Level ()
```

```
{
```

```
    Tiles = new Dictionary<Point, TileGrid> ();
```

```
    string[] mapData = ReadLevel();
```

```
    mapSize = new Point (mapData [0].ToCharArray ().Length,  
    mapData.Length);
```

```

TileGrid newTile =
    Instantiate(tilePrebas[tileIndex]).GetComponent<TileGrid> ();
newTile.SetUp (new Point (x, y), new Vector3 (worldStart.x + (TileSize * x),
    worldStart.y - (TileSize * y), 0), map);
}

private string[] ReadLevel ()
{
    TextAsset binddata = Resources.Load ("Level") as TextAsset;
    string data = binddata.text.Replace (Environment.NewLine,
        string.Empty);

    return data.Split ('-');
}

private void SpawnPortals()
{
    //Spawns the blue Portal
    blueSpawn = new Point (0, 0);

    GameObject tmp = (GameObject)Instantiate (bluePortalPrefab,
        Tiles[blueSpawn].GetComponent<TileGrid>().worldPosition
        , Quaternion.identity);

    BluePortal = tmp.GetComponent<Portal> ();

    BluePortal.name = "BluePortal";

    //Spawns the blue Portal
    redSpawn = new Point (11, 6);

```

```

Instantiate (redPortalPrefab, Tiles
[redSpawn].GetComponent<TileGrid>().worldPosition ,
    Quaternion.identity);

    }

public bool InBouds(Point position)

    {

        return position.X >= 0 && position.Y >= 0 && position.X <
        mapSize.X &&    position.Y < mapSize.Y;

    }

public void GeneratePath()

    {

        path = AStar.GetPath (blueSpawn, redSpawn);

    }

}

```


Singleton.cs

```
using UnityEngine;

using System.Collections;

public abstract class Singleton <T>: MonoBehaviour where T : MonoBehaviour
{
    private static T instance;

    public static T Instance
    {
        get {
            if(instance == null)
            {
                instance = FindObjectOfType<T>();
            }

            return instance;
        }
    }
}
```

Point.cs

```
using UnityEngine;
using System.Collections;

public struct Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
    public static bool operator ==(Point first, Point second)
    {
        return first.X == second.X && first.Y == second.Y;
    }
    public static bool operator !=(Point first, Point second)
    {
        return first.X != second.X || first.Y != second.Y;
    }
    public static Point operator -(Point x, Point y)
    {
        return new Point (x.X - y.X, x.Y - y.Y);
    }
}
```

BarScript.cs

```
using UnityEngine;

using System.Collections;

using UnityEngine.UI;

public class BarScript : MonoBehaviour
{
    private float fillAmount;

    [SerializeField]

    private float lerSpeed;

    [SerializeField]

    private Image content;

    [SerializeField]

    private Text valueText;

    [SerializeField]

    private Color fullColor;

    [SerializeField]

    private Color lowColor;

    [SerializeField]

    private bool lerpColors;

    public float MaxValue { get; set; }

    public float Value
    {
        set
        {
            fillAmount = Map (value, 0, MaxValue, 0, 1);
        }
    }
}
```

```

}

// Use this for initialization
void Start ()
{
    if (lerpColors)
    {
        content.color = fullColor;
    }
}

// Update is called once per frame
void Update ()
{
    HandleBar ();
}

private void HandleBar()
{
    if (fillAmount != content.fillAmount)
    {
        content.fillAmount = Mathf.Lerp(content.fillAmount, fillAmount,
Time.deltaTime * lerSpeed);
    }
    if (lerpColors)
    {
        content.color = Color.Lerp (lowColor, fullColor, fillAmount);
    }
}

```



```
public void Reset()
```

```
{
```

```
    content.fillAmount = 1;
```

```
    Value = MaxValue;
```

```
}
```

```
private float Map (float value, float inMin, float inMax, float outMin, float outMax)
```

```
{
```

```
    return (value - inMin) * (outMax - outMin) / (inMax - inMin) + outMin;
```

```
    //(80 - 0) * (1 - 0) / (100 - 0) + 0;
```

```
    // 80 * 1 / 100 + 0 = 0.8
```

```
    //(78 - 0) * (1 - 0) / (230 - 0) + 0;
```

```
    // 78 * 1 / 230 + 0 = 0.339
```

```
}
```

```
}
```

CameraMovement.cs

```
using UnityEngine;
using System.Collections;

public class CameraMovement : MonoBehaviour
{
    [SerializeField]
    private float cameraSpeed = 0;

    private float xMax;
    private float yMin;

    private void Update ()
    {
        GetKey ();
    }

    private void GetKey ()
    {
        if (Input.GetKey(KeyCode.W)) {
            transform.Translate (Vector3.up * cameraSpeed * Time.deltaTime);
        }
        if (Input.GetKey(KeyCode.A)) {
            transform.Translate (Vector3.left * cameraSpeed * Time.deltaTime);
        }
        if (Input.GetKey (KeyCode.S)) {
            transform.Translate (Vector3.down * cameraSpeed * Time.deltaTime);
        }
        if (Input.GetKey(KeyCode.D)) {
            transform.Translate (Vector3.right * cameraSpeed * Time.deltaTime);
        }
    }
}
```

```
}
```

```
transform.position =
```

```
new Vector3 (Mathf.Clamp (transform.position.x, 0, xMax),  
Mathf.Clamp (transform.position.y, yMin , 0), -10);
```

```
}
```

```
public void SetLimits (Vector3 maxTile )
```

```
{
```

```
Vector3 wp = Camera.main.ViewportToWorldPoint(new Vector3(1, 0));
```

```
xMax = maxTile.x - wp.x;
```

```
yMin = maxTile.y - wp.y;
```

```
}
```

```
}
```

Portal.cs

```
using UnityEngine;

using System.Collections;

public class Portal : MonoBehaviour

{

}
```

ObjectPool.cs

```
using UnityEngine;

using System.Collections;

using System.Collections.Generic;

public class ObjectPool : MonoBehaviour

{

    [SerializeField]

    private GameObject[] objectPrefabs;

    private List<GameObject> pooledObjects = new List<GameObject> ();

    public GameObject GetObject(string type)

    {

        foreach (GameObject go in pooledObjects)

        {

            if (go.name == type && ! go.activeInHierarchy)

            {

                go.SetActive (true);

                return go;

            }

        }

    }

}
```



```
for (int i = 0; i < objectPrefabs.Length; i++)
{
    if (objectPrefabs [i].name == type)
    {
        GameObject newObject = Instantiate (objectPrefabs [i]);
        pooledObjects.Add (newObject);
        newObject.name = type;
        return newObject;
    }
}

return null;
}

public void ReleaseObject(GameObject gameObject)
{
    gameObject.SetActive (false);
}

}
```

GameManager.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class GameManager : Singleton<GameManager>
{
    /// <summary>
    /// a property for the towerBtn
    /// </summary>
    /// <value>The clicked button.</value>
    public TowerBtn ClickedBtn { get; set; }

    /// <summary>
    /// A reference The currency text.
    /// </summary>
    private int currency;

    private int wave = 0;

    private int lives;

    private bool gameOver = false;

    private int health = 15;

    [SerializeField]
    private Text livesTxt;

    [SerializeField]
    private Text waveTxt;
```



```
public int Currency
```

```
{
```

```
    get
```

```
    {
```

```
        return currency;
```

```
    }
```

```
    set
```

```
    {
```

```
        this.currency = value;
```

```
        this.currencyTxt.text = value.ToString () + "<color=lime>$</color>";
```

```
    }
```

```
}
```

```
public int Lives
```

```
{
```

```
    get
```

```
    {
```

```
        return lives;
```

```
    }
```

```
    set
```

```
    {
```

```
        this.lives = value;
```

```
        if (lives <= 0)
```

```
        {
```

```
            this.lives = 0;
```

```
            GameOver();
```

```
        }
```

```

[SerializeField]
private Text currencyTxt;

[SerializeField]
private GameObject waveBtn;

[SerializeField]
private GameObject gameOverMenu;

/// <summary>
/// The current selected tower.
/// </summary>
private Tower selectedTower;

private List<Monster> activeMonsters = new List<Monster> ();

/// <summary>
/// A property for the object pool
/// </summary>
public ObjectPool Pool { get; set; }

public bool WaveActive
{
    get
    {
        return activeMonsters.Count > 0;
    }
}

/// <summary>
/// Property for accessing the currency
/// </summary>

```

```

        livesTxt.text = lives.ToString ();
    }
}

private void Awake()
{
    Pool = GetComponent<ObjectPool> ();
}

void Start ()
{
    Lives = 10;
    Currency = 100;
}

void Update ()
{
    HandleEscape ();
}

/// <summary>
/// Pickks the tower then a buy button is presss
/// </summary>
/// <param name="towerBtn">The clicked button.</param>
public void PickkTower(TowerBtn towerBtn)
{
    if(Currency >= towerBtn.Price && !WaveActive )
    {
        //Stores the clicked button
        this.ClickedBtn = towerBtn;
    }
}

```



```

        //Activates the hover icon
        HoverTower.Instance.Activate (towerBtn.Sprite);
    }
}

public void BuyTower()
{
    if(Currency >= ClickedBtn.Price)
    {
        Currency -= ClickedBtn.Price;
        HoverTower.Instance.Deactivate ();
    }
}

public void SelectTower(Tower tower)
{
    if (selectedTower != null)
    {
        selectedTower.Select ();
    }
    selectedTower = tower;
    selectedTower.Select ();
}

public void DeselectTower()
{
    if(selectedTower != null)

```

```

{
    selectedTower.Select ();
}

    selectedTower = null;
}

/// <summary>
/// Handles escape presses.
/// </summary>
private void HandleEscape()
{
    if (Input.GetKeyDown (KeyCode.Escape))//if we press escape
    {
        //Deactivate the hover instance
        HoverTower.Instance.Deactivate ();
    }
}

public void StartWave()
{
    wave++;

    waveTxt.text = string.Format ("Wave: <color=lime>{0}</color>", wave);

    StartCoroutine (SpawnWave ());

    waveBtn.SetActive (false);
}

```

```

private IEnumerator SpawnWave()
{
    LevelManager.Instance.GeneratePath ();

    for (int i = 0; i < wave; i++)
    {
        int monsterIndex = 0;//Random.Range (0, 4);

        string type = string.Empty;

        switch (monsterIndex)
        {
            case 0:
                type = "Red";
                break;

            case 1:
                type = "Blue";
                break;

            case 2:
                type = "Purple";
                break;

            case 3:
                type = "Green";
                break;

            }

        //Request the monster from the pool

        Monster monster = Pool.GetObject (type).GetComponent<Monster>();

        monster.Spawn (health);
    }
}

```

```

        if (wave % 3 == 0)
        {
            health += 5;
        }

        //Adds the monster to the active Monster List
        activeMonsters.Add (monster);

        yield return new WaitForSeconds (2.5f);
    }
}

public void RemoveMonster(Monster monster)
{
    //Remove the monster from the active list
    activeMonsters.Remove (monster);

    //If we dnt have more active monsters and the game isn't over,then we need to
    show the way
    if (!WaveActive && !gameOver)
    {
        //Shows the wave button
        waveBtn.SetActive (true);
    }
}

public void GameOver()
{
    if (!gameOver)
    {
        gameOver = true;
    }
}

```

```
        gameOverMenu.SetActive (true);
    }
}

public void Restart()
{
    Time.timeScale = 1;

    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}

public void QuiteGame()
{
    Application.Quit ();
}
}
```


HoverTower.cs

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class HoverTower : Singleton<HoverTower>
```

```
{
```

```
    /// <summary>
```

```
    /// A reference to the icon's spriteRenderer
```

```
    /// </summary>
```

```
    private SpriteRenderer spriteRenderer;
```

```
    /// <summary>
```

```
    /// A references to the rangecheck on the tower
```

```
    /// </summary>
```

```
    private SpriteRenderer rangeSpriteRenderer;
```

```
    void Start ()
```

```
    {
```

```
        //Creates the references to the sprite rendere
```

```
        this.spriteRenderer = GetComponent<SpriteRenderer> ();
```

```
        this.rangeSpriteRenderer = transform.GetChild (0).GetComponent<SpriteRenderer>
```

```
    ();
```

```
    }
```

```

void Update ()
{
    FollowMouse ();
}

private void FollowMouse()
{
    if (spriteRenderer.enabled) {
        transform.position = Camera.main.ScreenToWorldPoint
(Input.mousePosition);
        transform.position = new Vector3 (transform.position.x,
transform.position.y, 0);
    }
}

```

```

/// <summary>

```

```

/// Activate the Hover icon

```

```

/// </summary>

```

```

/// <param name="sprite">Sprite.</param>

```

```

public void Activate(Sprite sprite)

```

```

{

```

```

    //Sets the correct sprite

```

```

    this.spriteRenderer.sprite = sprite;

```

```

    //Enables the renderer

```

```
spriteRenderer.enabled = true;

rangeSpriteRenderer.enabled = true;
```

```
}
```

```
/// <summary>
```

```
/// Deactivate this Hover icon.
```

```
/// </summary>
```

```
public void Deactivate()
```

```
{
```

```
    //Disables the renderer to that we can see it
```

```
    spriteRenderer.enabled = false;
```

```
    //Unclicks our button
```

```
    GameManager.Instance.ClickedBtn = null;
```

```
    rangeSpriteRenderer.enabled = false;
```

```
}
```

```
}
```

Monster.cs

```
using UnityEngine;

using System.Collections.Generic;

using System.Collections;

public class Monster : MonoBehaviour
{
    [SerializeField]
    private float speed;

    private Stack<Node> path;

    private List<Debuff> debuffs = new List<Debuff> ();

    [SerializeField]
    private Element elementType;

    private SpriteRenderer spriteRenderer;

    private int invulnerability = 2;

    private Animator myAnimator;

    [SerializeField]
    private Stat health;

    public bool Alive
    {
        get { return health.CurrentVal > 0; }
    }

    public Point GridPosition { get; set; }

    private Vector3 destination;

    public bool IsActive { get; set; }
```

```

//sets the monster path
    SetPath (LevelManager.Instance.Path);
}

/// <summary>
/// Scale a monster up or down
/// </summary>
/// <param name="from">start scale.</param>
/// <param name="to">end scale.</param>
/// <return><c>/returns>
public IEnumerator Scale(Vector3 from, Vector3 to, bool remove)
{
    //IsActive = false;

    //The scaling progress
    float progress = 0;

    //As long as the progress is less than 1, then we need to keep scaling
    while (progress <= 1)
    {
        //Scales the monster
        transform.localScale = Vector3.Lerp (from, to, progress);
        progress += Time.deltaTime;
        yield return null;
    }

    //Make sure that is has the correct scale after scaling
    transform.localScale = to;

    IsActive = true;
}

```



```

public Element ElementType
{
    get
    {
        return elementType;
    }
}

private void Awake()
{
    spriteRenderer = GetComponent<SpriteRenderer> ();
    health.Initialize ();
}

private void Update()
{
    HandleDebuffs ();
    Move ();
}

public void Spawn(int health)
{
    transform.position = LevelManager.Instance.BluePortal.transform.position;
    this.health.Bar.Reset ();
    this.health.MaxVal = health;
    this.health.CurrentVal = this.health.MaxVal;
    myAnimator = GetComponent<Animator> ();
    //Starts to scale the Monster
    StartCoroutine (Scale (new Vector3 (0.1f, 0.1f), new Vector3 (1, 1), false));
}

```

```

if (remove)
{
    Release ();
}
}

/// <summary>
/// Move this instance move along the given path.
/// </summary>
private void Move()
{
    if(!IsActive)
    {
        transform.position = Vector3.MoveTowards (transform.position,
            destination, speed * Time.deltaTime);

        if (transform.position == destination)
        {
            if (path != null && path.Count > 0)
            {
                Animate (GridPosition, path.Peek ().GridPosition);

                GridPosition = path.Peek ().GridPosition;

                destination = path.Pop ().WorldPosition;
            }
        }
    }
}
}

```

```

private void SetPath(Stack<Node> newPath)
{
    if (newPath != null)
    {
        this.path = newPath;

        Animate (GridPosition, path.Peek ().GridPosition);

        GridPosition = path.Peek ().GridPosition;

        // Sets a new destination
        destination = path.Pop ().WorldPosition;
    }
}

private void Animate(Point currentPos, Point newPos)
{
    if (currentPos.Y > newPos.Y)
    {
        //We are moving down
        myAnimator.SetInteger ("Horizontal", 0);
        myAnimator.SetInteger ("Vertical", 1);
    }
    else if (currentPos.Y < newPos.Y)
    {
        myAnimator.SetInteger ("Horizontal", 0);
        myAnimator.SetInteger ("Vertical", -1);
    }
}

```

```

if (currentPos.Y == newPos.Y)
{
    if (currentPos.X > newPos.X)
    {
        //Then we are moving
        myAnimator.SetInteger ("Horizontal", -1);
        myAnimator.SetInteger ("Vertical", 0);
    }
    if (currentPos.X < newPos.X)
    {
        myAnimator.SetInteger ("Horizontal", 1);
        myAnimator.SetInteger ("Vertical", 0);
    }
}

private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "RedPortal") //if we collide with the red portal
    {
        //Start Scaling the monster down
        StartCoroutine(Scale(new Vector3(1,1), new Vector3(0.1f,0.1f), true));
        GameManager.Instance.Lives--;
    }
}

```

```

if (other.tag == " Tile")
{
    spriteRenderer.sortingOrder = other.GetComponent<TileGrid>
().GridPosition.Y;
}
}

public void Release()
{
    IsActive = false;

    GridPosition = LevelManager.Instance.BlueSpawn;

    GameManager.Instance.Pool.ReleaseObject (gameObject);

    GameManager.Instance.RemoveMonster (this);
}

public void TakeDamage(int damage, Element dmgSource)
{
    if (IsActive)
    {
        if (dmgSource == elementType)
        {
            damage = damage / invulnerability;

            invulnerability++;
        }

        health.CurrentVal -= damage;

        if (health.CurrentVal <= 0)
        {

```

```

        GameManager.Instance.Currency += 2;

        myAnimator.SetTrigger ("Die");

        IsActive = false;

        GetComponent<SpriteRenderer> ().sortingOrder--;
    }

}

public void AddDebuff(Debuff debuff)
{
    if(!debuffs.Exists(x => x.GetType() == debuffs.GetType()))
    {
        debuffs.Add (debuff);
    }
}

public void RemoveDebuff(Debuff debuff)
{
    debuffs.Remove (debuff);
}

private void HandleDebuffs()
{
    foreach(Debuff debuff in debuffs)
    {
        debuff.Update();
    }
}
}

```


Projectile.cs

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour
{
    private Monster target;

    private Tower parent;

    private Element elementType;

    //private Animator myAnimator;

    // Use this for initialization

    void Start ()
    {
        //myAnimator = GetComponent<Animator> ();
    }

    // Update is called once per frame
    void Update ()
    {
        MoveToTarget ();
    }

    public void Initialize(Tower parent)
    {
        this.target = parent.Target;

        this.parent = parent;

        this.elementType = parent.ElementType;
    }
}
```

```

private void MoveToTarget()
{
    if (target != null && target.IsActive)
    {
        transform.position =
Vector3.MoveTowards(target.transform.position,transform.position,Time.deltaTime *
parent.ProjectileSpeed);

        Vector2 dir = target.transform.position - transform.position;

        float angle = Mathf.Atan2 (dir.y, dir.x) * Mathf.Rad2Deg;

        transform.rotation = Quaternion.AngleAxis (angle, Vector3.forward);
    }
    else if(!target.IsActive)
    {
        GameManager.Instance.Pool.ReleaseObject (gameObject);
    }
}

private void ApplyDebuff()
{
    if(target.ElementType != elementType)
    {
        float roll = Random.Range (0, 100);

        if (roll <= parent.Proc)
        {
            target.AddDebuff (parent.GetDebuff ());
        }
    }
}
}

```

```

private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Monster")
    {
        if (target.gameObject == other.gameObject)
        {
            target.TakeDamage (parent.Damage, elementType);
            GameManager.Instance.Pool.ReleaseObject (gameObject);
            ApplyDebuff ();
        }
    }
}
}

```

State.cs

```

using UnityEngine;
using System.Collections;
using System;
[Serializable]
public class Stat
{
    [SerializeField]
    private BarScript bar;
    [SerializeField]
    private float maxVal;
}

```

[SerializeField]

private float currentVal;

public float CurrentVal

```
{  
    get  
    {  
        return currentVal;  
    }  
    set  
    {  
        this.currentVal = Mathf.Clamp(value,0,MaxVal);  
        bar.Value = currentVal;  
    }  
}
```

public float MaxVal

```
{  
    get  
    {  
        return maxVal;  
    }  
    set  
    {  
        this.maxVal = value;  
        bar.MaxValue = maxVal;  
    }  
}
```

```

public BarScript Bar
{
    get
    {
        return bar;
    }
}

public void Initialize()
{
    this.MaxVal = maxVal;
    this.CurrentVal = currentVal;
}
}

```

TileGrid.cs

```

using UnityEngine;
using System.Collections;
using UnityEngine.EventSystems;
public class TileGrid : MonoBehaviour
{
    /// <summary>
    /// Gets the grid position.
    /// </summary>
    public Point GridPosition { get; private set; }
    public bool IsEmpty{ get; private set; }
}

```

```

public void SetUp(Point gridPos, Vector3 worldPos, Transform parent)
{
    WalkAble = true;
    IsEmpty = true;
    this.GridPosition = gridPos;
    transform.position = worldPos;
    transform.SetParent (parent);
    LevelManager.Instance.Tiles.Add (gridPos, this);
}

private void OnMouseOver()
{
    if (!EventSystem.current.IsPointerOverGameObject () &&
    GameManager.Instance.ClickedBtn != null)
    {
        if (IsEmpty && !Debugging)
        {
            ColorTile (emptyColor);
        }
        if (!IsEmpty && !Debugging)
        {
            ColorTile (fullcolor);
        } else if (Input.GetMouseButtonDown (0)) {
            PlaceTower ();
        }
    }
}

```



```

private Tower myTower;

/// <summary>
/// The color of the tile, when its full, this is used while hovering the tile with the mouse
/// </summary>
private Color32 fullcolor = new Color32 (225, 118, 118, 255);

/// <summary>
/// The color of the tile, when its empty, this is used while hovering the tile with the mouse
/// </summary>
private Color32 emptyColor = new Color32 (96, 255, 90, 255);

private SpriteRenderer spriteRenderer;

public bool WalkAble { get; set; }

public bool Debugging { get; set; }

public Vector2 worldPosition
{
    get
    {
        return new Vector2 (transform.position.x +
(GetComponent<SpriteRenderer> ().bounds.size.x / 2), transform.position.y -
(GetComponent<SpriteRenderer> ().bounds.size.y / 2));
    }

}

void Start ()
{
    spriteRenderer = GetComponent<SpriteRenderer> ();
}

void Update () {}

```



```
else if (!EventSystem.current.IsPointerOverGameObject () &&  
GameManager.Instance.ClickedBtn == null && Input.GetMouseButtonDown(0))
```

```
{  
  
    if (myTower != null)  
    {  
  
        GameManager.Instance.SelectTower (myTower);  
  
    }  
  
    else  
  
    {  
  
        GameManager.Instance.DeselectTower();  
  
    }  
  
}
```

```
}  
  
private void OnMouseExit()
```

```
{  
  
if (!Debugging)  
  
    {  
  
        ColorTile (Color.white);  
  
    }  
  
}
```

```
}  
  
private void PlaceTower()
```

```
{  
  
    //Creates the tower  
  
    GameObject tower = (GameObject)Instantiate  
(GameManager.Instance.ClickedBtn.TowerPrefab, transform.position, Quaternion.identity);
```

```
//Set the tile as transform parent to the tower
tower.GetComponent<SpriteRenderer> ().sortingOrder = GridPosition.Y;
tower.transform.SetParent (transform);
this.myTower = tower.transform.GetChild (0).GetComponent<Tower> ();

//Makes sure that it isn't empty
IsEmpty = false;

//Sets the color back to white
ColorTile (Color.white);

//Buys the tower
GameManager.Instance.BuyTower();
WalkAble = false;
}

private void ColorTile(Color newColor)
{
    spriteRenderer.color = newColor;
}
}
```

Astar

```
using System.Collections.Generic;
```

```
using UnityEngine;
```

```
using System;
```

```
using System.Linq;
```

```
public static class AStar
```

```
{
```

```
    private static Dictionary<Point, Node> nodes;
```

```
    private static void CreateNodes()
```

```
    {
```

```
        nodes = new Dictionary<Point,Node> ();
```

```
        foreach(TileGrid tile in LevelManager.Instance.Tiles.Values)
```

```
        {
```

```
            nodes.Add (tile.GridPosition, new Node (tile));
```

```
        }
```

```
    }
```

```

public static Stack<Node> GetPath(Point start, Point goal)
{
    if (nodes == null)
    {
        CreateNodes();
    }

    HashSet<Node> openList = new HashSet<Node> ();

    HashSet<Node> closedList = new HashSet<Node> ();

    Stack<Node> finalPath = new Stack<Node> ();

    Node currentNode = nodes [start];

    //STEP 1. Adds the start node to the openList
    openList.Add(currentNode);

    while (openList.Count > 0) // STEP 10
    {

```

```

nodes [neighbourPos]])

        if (!ConnectedDiagonally (currentNode,

            {

                continue;

            }

            gCost = 14;

        }

//STEP 3. Add the neighbor to the openList
Node neighbour = nodes [neighbourPos];

if (openList.Contains (neighbour))
{
    if (currentNode.G + gCost < neighbour.G)
    {

        neighbour.CalcValues (currentNode,

nodes[goal], gCost); //STEP 9.4

    }

```



```

//STEP 2. Runs through all neighbor
for(int x = -1; x <= 1; x++)
{
    for (int y = -1; y <= 1; y++)
    {
        //Sets the initial value of g to 0
        Point neighbourPos = new Point (currentNode.GridPosition.X
- x, currentNode.GridPosition.Y - y);

        if(LevelManager.Instance.InBouds(neighbourPos) &&
LevelManager.Instance.Tiles[neighbourPos].WalkAble && neighbourPos !=
currentNode.GridPosition)

        {
            int gCost = 0;

            if (Math.Abs(x - y) == 1) // check is we need score 10
            {
                gCost = 10;
            }

            else //Scores 14 if we are diadonal
            {

```

```
        //Sort the list by F value and select the first on th list  
        currentNode = openList.OrderBy (n => n.F).First ();  
    }
```

```
    if (currentNode == nodes [goal])  
    {
```

```
        while (currentNode.GridPosition != start)  
        {  
            finalPath.Push (currentNode);  
            currentNode = currentNode.Parent;  
        }  
        break;  
    }
```

```
}
```

```
return finalPath;
```

```
//have to remove later
```

```
//GameObject.Find("AStar").GetComponent<AStarDebug>().DebugPath(openList,  
closedList, finalPath);
```

```

    }
    else if(!closedList.Contains(neighbour)) //STEP 9.1
    {
        openList.Add (neighbour); // STEP 9.2
        neighbour.CalcValues(currentNode,
nodes[goal], gCost); //STEP 9.3
    }
}
}
}
}
}

```

```

// STEP 5 & 8. Moves the current node from the openlist to the closedList
openList.Remove (currentNode);
closedList.Add (currentNode);

if (openList.Count > 0) // STEP 7
{

```

```

}

private static bool ConnectedDiagonally(Node currentNode, Node neighbor)
{
    Point direction = neighbor.GridPosition - currentNode.GridPosition;

    Point first = new Point(currentNode.GridPosition.X + direction.X,
currentNode.GridPosition.Y);

    Point second = new Point (currentNode.GridPosition.X, currentNode.GridPosition.Y
+ direction.Y);

    if (LevelManager.Instance.InBouds(first) && !LevelManager.Instance.Tiles
[first].WalkAble)
    {

        return false;

    }

    if(LevelManager.Instance.InBouds(second) && !LevelManager.Instance.Tiles
[second].WalkAble)
    {

        return false;

    }

    return true;

}

}

```


FireTower.cs

```
using UnityEngine;
using System.Collections;

public class FireTower : Tower
{
    [SerializeField]
    private float tickTime;
    public float TickTime
    {
        get
        {
            return tickTime;
        }
    }
    [SerializeField]
    private float takeDamage;

    public float TakeDamage
    {
        get
        {
            return takeDamage;
        }
    }
}
```

FrostTower.cs

```
using UnityEngine;
using System.Collections;

public class FrostTower : Tower
{
}
```

```

[SerializeField]
private float slowingFactor;

private void Start()
{
    ElementType = Element.FROST;
}

public override Debuff GetDebuff ()
{
    return new FrostDebuff (slowingFactor, DebuffDuration,Target);
}
}

```

PosionTower.cs

```

using UnityEngine;
using System.Collections;

public class PosionTower : Tower
{
    [SerializeField]
    private float tickTime;

    public float TickTime
    {
        get
        {
            return tickTime;
        }
    }

    [SerializeField]

```



```
private PosionSplash splashPrefab;
```

```
[SerializeField]
```

```
private int splashDamage;
```

```
public float SplashDamage
```

```
{
```

```
    get
```

```
    {
```

```
        return splashDamage;
```

```
    }
```

```
}
```

```
private void Start()
```

```
{
```

```
    ElementType = Element.POISON;
```

```
}
```

```
public override Debuff GetDebuff ()
```

```
{
```

```
    return new PosionDebuff (splashDamage, tickTime, splashPrefab,  
DebuffDuration,Target);
```

```
}
```

```
}
```

```
StromTower.cs
```

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class StromTower : Tower
```

```
{
```

```

private void Start()
{
    ElementType = Element.STROM;
}

public override Debuff GetDebuff ()
{
    return new StromDebuff (Target, DebuffDuration);
}
}

```

Tower.cs

```

using UnityEngine;
using System.Collections.Generic;

public enum Element { STROM, FIRE, FROST, POISON, NONE }

public abstract class Tower : MonoBehaviour
{
    [SerializeField]
    private string projectileType;

    [SerializeField]
    private float projectileSpeed;

    public float ProjectileSpeed
    {
        get
        {
            return projectileSpeed;
        }
    }
}

```

```
[SerializeField]
```

```
private int damage;
```

```
public int Damage
```

```
{  
    get  
    {  
        return damage;  
    }  
}
```

```
[SerializeField]
```

```
private float debuffDuration;
```

```
public float DebuffDuration
```

```
{  
    get  
    {  
        return debuffDuration;  
    }  
    set  
    {  
        this.debuffDuration = value;  
    }  
}
```

```
[SerializeField]
```

```
private float proc;
```

```
public float Proc
```

```
{
```

```
        get
        {
            return proc;
        }
        set
        {
            this.proc = value;
        }
    }
}
```

```
public Element ElementType { get; protected set; }
```

```
public int Price { get; set; }
```

```
private SpriteRenderer mySpriteRenderer;
```

```
private Monster target;
```

```
public Monster Target
{
    get
    {
        return target;
    }
}
```

```
private Queue<Monster> monsters = new Queue<Monster> ();
```

```
private bool canAttack = true;
```

```
private float attackTimer;
```

```
[SerializeField]
```

```
private float attackCoolDown;
```

```
// Use this for initialization
```

```
void Start ()
```

```
{
```

```
    mySpriteRenderer = GetComponent<SpriteRenderer> ();
```

```
}
```

```
// Update is called once per frame
```

```
void Update ()
```

```
{
```

```
    Attack ();
```

```
    Debug.Log (target);
```

```
}
```

```
public void Select()
```

```
{
```

```
    mySpriteRenderer.enabled = !mySpriteRenderer.enabled;
```

```
}
```

```
public void Attack()
```

```
{
```

```
    if (!canAttack)
```

```
    {
```

```
        attackTimer += Time.deltaTime;
```

```
        if (attackTimer >= attackCoolDown)
```

```

        {
            canAttack = true;
            attackTimer = 0;
        }
    }

    if (target == null && monsters.Count > 0)
    {
        target = monsters.Dequeue ();
    }
    if (target != null && target.IsActive)
    {
        if (canAttack)
        {
            Shoot ();

            canAttack = false;
        }
    }

    else if (monsters.Count > 0)
    {
        target = monsters.Dequeue ();
    }
    if (target != null && !target.Alive)
    {
        target = null;
    }
}

```

```

private void Shoot()

```

```

{
    //Gets a projectile from the object pool
    Projectile projectile =
GameManager.Instance.Pool.GetObject(projectileType).GetComponent<Projectile>();

    //Makes sure that the projectile is instantiated on the towwr position
    projectile.transform.position = transform.position;

    projectile.Initialize (this);
}

public void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Monster") //Adds new monsters to the queu when they enter
the range
    {
        monsters.Enqueue(other.GetComponent<Monster>());
    }
}

public abstract Debuff GetDebuff ();

public void OnTriggerExit2D(Collider2D other)
{
    if (other.tag == "Monster") //Adds new monsters to the queu when they enter
the range
    {
        target = null;
    }
}
}

```


Debuff.cs

```
using UnityEngine;
using System.Collections;

public abstract class Debuff
{
    protected Monster target;

    private float duration;

    private float elapsed;

    public Debuff(Monster target, float duration)
    {
        this.target = target;
        this.duration = duration;
    }

    public virtual void Update()
    {
        elapsed += Time.deltaTime;

        if (elapsed >= duration)
        {
            Remove ();
        }
    }

    public virtual void Remove()
    {
        if(target != null)
        {
```

```

        target.RemoveDebuff (this);
    }
}

```

FireDebuff.cs

```

using UnityEngine;
using System.Collections;

public class FireDebuff : Debuff
{
    private float tickTime;
    private float timeSinceTick;
    private float tickDamage;

    public FireDebuff(float tickDamage, float tickTime, float duration, Monster target) :
    base(target, duration)
    {
        this.tickDamage = tickDamage;
        this.tickTime = tickTime;
    }

    public override void Update ()
    {
        if (target != null)
        {
            timeSinceTick += Time.deltaTime;

            if(timeSinceTick >= tickTime)
            {
                timeSinceTick = 0;
            }
        }
    }
}

```

```

        target.TakeDamage (tickDamage, Element.FIRE);
    }
}

    base.Update ();
}
}

```

FrostDebuff

```

using UnityEngine;
using System.Collections;

public class FrostDebuff : Debuff
{
    private float slowingFactor;
    private bool applied;
    public FrostDebuff(float slowingFactor, float duration, Monster target) : base(target,
duration)
    {
        this.slowingFactor = slowingFactor;
    }

    public override void Update()
    {
        if(target != null)
        {
            if (!applied)
            {
                applied = true;
                target.Speed -= (target.MaxSpeed * slowingFactor) / 100;
            }
        }
    }
}

```

```

    }

    base.Update ();

}

public override void Remove()
{
    target.Speed = target.MaxSpeed;

    base.Remove ();
}
}

```

PosionDebuff.cs

```

using UnityEngine;
using System.Collections;

public class PosionDebuff : Debuff
{
    private float tickTime;
    private float timeSinceTick;
    private PosionSplash splashPrefab;
    private int splashDamage;

    public PosionDebuff(int splashDamage, float tickTime, PosionSplash splashPrefab, float
duration ,Monster target) : base(target, duration)
    {
        this.splashDamage = splashDamage;
        this.tickTime = tickTime;
        this.splashPrefab = splashPrefab;
    }
}

```

```

public override void Update()
{
    if (target != null)
    {
        timeSinceTick += Time.deltaTime;

        if (timeSinceTick >= tickTime)
        {
            timeSinceTick = 0;
            Splash ();
        }
    }

    base.Update ();
}

private void Splash ()
{
}
}

```

PosionSplash.cs

```

using UnityEngine;
using System.Collections;

public class PosionSplash : MonoBehaviour
{
}

```



StromDebuff.cs

```
using UnityEngine;  
using System.Collections;
```

```
public class StromDebuff : Debuff  
{
```

```
    public StromDebuff( Monster target, float duration) : base(target, duration)  
    {  
        if (target != null)  
        {  
            target.Speed = 0;  
        }  
    }
```

```
    public override void Remove()  
    {  
        if (target != null)  
        {  
            target.Speed = target.MaxSpeed;  
            base.Remove ();  
        }  
    }  
}
```

TowerBtn.cs

```
using UnityEngine;  
using System.Collections;  
using UnityEngine.UI;
```

```

public class TowerBtn : MonoBehaviour {

    [SerializeField]
    private GameObject towerPrefab;

    [SerializeField]
    private Sprite sprite;

    [SerializeField]
    private int price;

    [SerializeField]
    private Text priceTxt;

    public GameObject TowerPrefab {
        get
        {
            return towerPrefab;
        }
    }

    public Sprite Sprite
    {
        get
        {
            return sprite;
        }
    }

    public int Price
    {
        get

```



```
    {  
        return price;  
    }  
}
```

```
private void Start()  
{  
    priceTxt.text = price + "$";  
}
```

```
}
```