Department of Electrical and Computer Engineering

North South University

Senior Design Project

Tower Defense

MD Ahad Hasan: 1211028040

Samiul Ahmed Sami: 1511510040

Faculty Advisor: Dr. Shazzad Hossain

Professor

ECE Department

Agreement Form

We take great pleasure in submitting our senior design project report on “Tower Defense”. This report is prepared as a requirement of the Capstone Design Project CSE/EEE/ETE 499 A & B which is a two-semester-long senior design course. This course involves multidisciplinary teams of students who build and test custom-designed systems, components, or engineering processes. We would like to request you to accept this report as partial fulfillment of the Bachelor of Science degree from the Electrical and Computer Engineering Department of North South University.

Declared By:

.....

Name: MD Ahad Hasan

ID: 1211028040

.....

Name: Samiul Ahmed Sami

ID: 1511510040

Approved By:

.....

Supervisor

Dr. Shazzad Hossain

Professor, Department of Electrical and Computer Engineering

North South University, Dhaka, Bangladesh

.....

Dr. Rajesh Palit

Chairman, Department of Electrical and Computer Engineering

North South University, Dhaka, Bangladesh

Acknowledgment

We would like to express our heartfelt gratitude to our supervisor Dr. Shazzad Hossain as well as our chairman Dr. Rajesh Palit who gave us the golden opportunity to do this wonderful project on the topic of simulation games, which also helped me in doing a lot of Research and we came to know about so many new things. We are really thankful to them. Secondly, we would also like to thank the friends who helped me a lot in finalizing this project within the limited time frame.

Table of Contents

Abstract

This paper describes an attempt at making a classic genre of game based on java. Games are a very important source of entertainment for millions around the world, the worldwide gaming industry being worth around $188 billion around the world in 2022. Steps taken to make the game initially has been described. The project was an attempt at understanding the most basic steps, without using game engines to assist, to make the game. The game itself has commercial value if completed and released on the market. Tower defense is a subgenre of strategy games where the goal is to defend a player's territories or possessions by obstructing the enemy attackers or by stopping enemies from reaching the exits, usually achieved by placing defensive structures on or along their path of attack. This typically means building a variety of different structures that serve to block, impede, attack, or destroy enemies automatically. Tower defense is seen as a subgenre of real-time strategy video games, due to its real-time origins, even though many modern tower defense games include aspects of turn-based strategy. Strategic choice and positioning of defensive elements is an essential strategy of the genre.

Chapter 1: Overview

Chapter 1: Overview

1.1 Introduction

The game we made is a tower defense game. The first of its kind was released around 1990 by Atari Games called Rampart for arcade machines. The first one released for pc was around 2007 called Desktop TD, made with flash. Games of this genre combine the real-time strategy and shooter genre in a sense to form a new genre. The tower defense genre has captured the fascination of many around the world because of its diverse nature and its learning curve where the initial concept is easy but game completion is usually difficult and requires a mix of well-thought-out tower placement and strategy.

Fig: Castle Defense

Fig: Bloons TD

1.2 Motivation

The idea was to try making a strategy game from scratch using java which is an object-oriented programming language. There are quite a few things that we had to familiarize ourselves with which are different from how usual programs are made and run. Learning these things to later be able to make more games was the motivation behind this project.

1.3 Goals

The main target was to make a demo that could show the functionality of the game and do all the basics necessary for the game to run in java. The final goal was to produce a fully functional game with adjustable settings and multiple levels with interesting combat mechanics and enemies.

1.4 Summary

The project aims to make a tower defense game demo in java to show a glimpse of what the final product will look like while going through the learning processes involved in making the game from scratch.

Chapter 2: Previous work

Chapter 2: Previous work

The tower defense genre is by no means new, with the first release being in 1990 and the first pc release in 2007, a full 15 years of pc releases means that this genre is well developed and has many unique titles under its belt. One of the most unique that comes to mind is monkeys defending their homes from balloons (called Bloons TD), a hat tip to balloons getting stuck in trees. Predictably, this means that there already exists a large number of varied games that fall under this genre. There are many more examples of such games, where you as a human defend yourself against various forces and factions. Even as a non-human there are many options to pick from where you play with elves and faeries to defend together with humans. In recent years a sort of variant of tower defense has come up where you play as a dungeon keeper or overlord and you have to defend your dungeon, which is your home, from humans who invade it in the hopes of finding treasure. This similar but different flavor of the game caught on quite quickly and there are now two series that play into a similar genre though slightly different at its core. Another popular mix is the zombie and tower defense genres where you defend against hordes of zombies using towers. One of the more popular games in recent times is called they are billions, a difficult game where one zombie that got through your defense can turn your entire colony into Zombieland.

Chapter 3: Producing a game

Chapter 3: Producing a game

3.1 Update and frame rates

To produce a game of this genre the actors are the player, who places towers, the towers who attack enemies, and enemies who path toward a target to reduce lives. To make everything run there needs to be a constant visual update and a constant action update. This is handled via two different tickers - one that controls how often the game takes an action and one that controls how often said action is updated visually. For both of these, the time is taken from the system in nanoseconds and then checked with a previously setup variable to approximately update the game's actions 60 times a second (henceforth the update rate) and to visually update the game 120 times a second (henceforth the frame rate).

In some older games, there was an issue where the frame rate and update rate were not separated so the game moved faster instead of updating faster visually. Some games were made with no limiter at all and the run speed of the game simply depended on the performance of the pc and so when you ran said games on a modern pc they would update very fast and you could not react at all to the game being run.

The tickers were checked against the internal system time in nanoseconds. The rate is still not 100% accurate, however, and there is an extra update once every now and then.

```
private final double FPS_SET = 120.0;
private final double UPS_SET = 60.0;
```

Table 1.

```
public void run() {
    double timePerFrame = 1000000000.0 / FPS_SET;
    double timePerUpdate = 1000000000.0 / UPS_SET;
```

```
    long lastFrame = System.nanoTime();
    long lastUpdate = System.nanoTime();
    long lastTimeCheck = System.currentTimeMillis();
    int frames = 0;
    int updates = 0;

    long now;

    while (true) {
        now = System.nanoTime();

        // Render
        if (now - lastFrame >= timePerFrame) {
            repaint();
            lastFrame = now;
            frames++;
        }

        // Update
        if (now - lastUpdate >= timePerUpdate) {
            updateGame();
            lastUpdate = now;
            updates++;
        }

        if (System.currentTimeMillis() - lastTimeCheck >= 1000) {
            System.out.println("FPS: " + frames + " | UPS: " + updates);
            frames = 0;
            updates = 0;
            lastTimeCheck = System.currentTimeMillis();
        }
    }
}
```

Table 2.

```
private void updateGame() {
    switch (GameStates.gameState) {
        case EDIT:
            editing.update();
            break;
        case MENU:
```

```
            break;
        case PLAYING:
            playing.update();
            break;
        case SETTINGS:
            break;
        default:
            break;
    }
}
```

Table 3.

3.2 Visual resources

The visuals used were 30, 32x32 pixel images (10 in a row - 3 columns). 4 images were used to produce a water animation by changing the image once every few frame changes (which is determined by the frame rate ticker previously described). Images had transparent pixels and were rotated to produce a total of 16 images, from a starting 7, with some water on them all of which were animated. The rest of the images were path tiles, enemies, towers, projectiles shot by towers, frozen effects, explosion animations, and start and end users only in the editor.

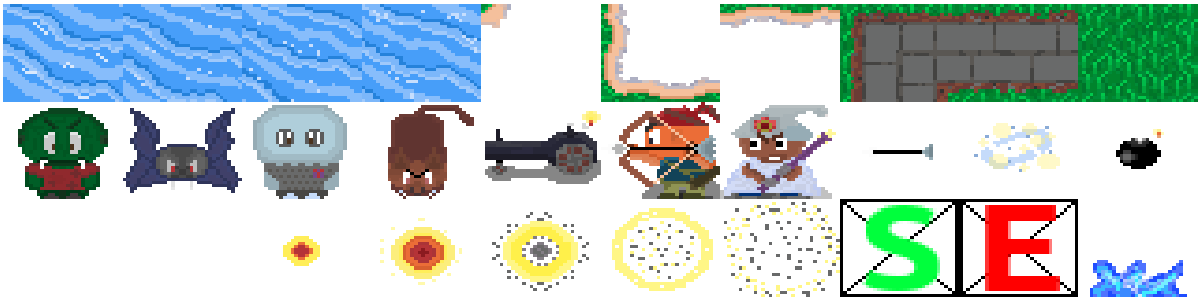


Figure 1.

3.3 Towers

There are three towers in the game. The basic tower is the archer, used as the baseline. It has average projectile damage and no specialties. The cannon tower fires bombs at enemies which explode with a splash effect dealing good damage to multiple targets. The primary target also takes double damage so it is a strong, but expensive, tower. The mage tower fires low-damage projectiles that slow down enemies, which results in two effects. The enemies

are slower so they can be damaged longer before they take a life. The enemies get bunched up together so a cannon tower becomes more effective against them.

3.4 Projectiles

The projectiles were an interesting portion of the code. The image of the projectile had to be rotated based on the tower location and the current enemy location and had to be adjusted to appear from the center of towers and target the center of enemies which required pixel adjustments in the code. The active projectiles are stored in a list of projectiles along with their type, and one of them, the cannonball, produced explosions that used a separate list with active and inactive explosions. The active explosions played 7 images before becoming inactive.

```
public void newProjectile(Tower t, Enemy e) {
    int type = getProjType(t);

    int xDist = (int)(t.getX() - e.getX());
    int yDist = (int)(t.getY() - e.getY());
    int totDist = Math.abs(xDist) + Math.abs(yDist);

    float xPer = (float) Math.abs(xDist) / totDist;

    float xSpeed = xPer *
helpz.Constants.Projectiles.GetSpeed(type);
    float ySpeed = helpz.Constants.Projectiles.GetSpeed(type) -
xSpeed;

    if (t.getX() > e.getX())
        xSpeed *= -1;
    if (t.getY() > e.getY())
        ySpeed *= -1;
```

```
    float rotate = 0;

    if (type == ARROW) {
        float arcValue = (float) Math.atan(yDist / (float) xDist);
        rotate = (float) Math.toDegrees(arcValue);

        if (xDist < 0)
            rotate += 180;
    }

    for (Projectile p: projectiles)
        if (!p.isActive())
            if (p.getProjectileType() == type) {
                p.reuse(t.getX() + 16, t.getY() + 16, xSpeed,
ySpeed, t.getDmg(), rotate);
                return;
            }

    projectiles.add(new Projectile(t.getX() + 16, t.getY() + 16,
xSpeed, ySpeed, t.getDmg(), rotate, proj_id++, type));
}
```

Table 4.

3.5 Waves

The enemies in such games appear one after the other with some delay in between, waves are coded based on the update rate, rather than the frame rate (which would be adjustable by the player based on what their system can handle). At the end of every wave all lists are cleared (enemy/projectile/explosion) so that a memory leak does not occur. Two classes were used to simulate waves. The first class handles the enemy list. The second class manages all the waves for the duration of the game and spawns enemies during waves.

```
private void createWaves() {
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 0, 0))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 0, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 1, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 1, 1, 1, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 0, 0, 1, 1, 1, 1, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 0, 1, 1, 1, 1, 1, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1))));
    waves.add(new Wave(new ArrayList < Integer > (Arrays.asList(0,
0, 0, 0, 1, 1, 1, 1, 2))));
}
```

Table 5.

3.6 Enemies

The game has 4 enemies - the orc, wolf, bat, and knight. The orc is your basic enemy used as basic fodder. The wolves are fast but have lesser hitpoints. The bat was intended to be a flying enemy type that only archer towers would target, but this was not coded in yet. The knight is your tank - high hp and slow-moving, able to take a lot of punishment. Each of these enemies has its own class which handles storing the data about each individual enemy. An enemy manager handles all the enemies present in a wave.

```
private void attackEnemyIfClose(Tower t) {
    for (Enemy e: playing.getEnemyManger().getEnemies()) {
        if (e.isAlive())
            if (isEnemyInRange(t, e)) {
                if (t.isCooldownOver()) {
                    playing.shootEnemy(t, e);
                    t.resetCooldown();
                }
            }
    }
}
```

Table 6.

3.7 Lives

For a game to be interesting it needs a win condition and a loss condition. You start with a set number of lives and if enemies reach the end of the path you lose a life. The win condition is killing all enemies before losing all lives. The loss condition is losing all lives before killing all enemies. All this is handled in the bottomBar class in the UI package.

3.8 Load/Save

The loading and saving of levels are handled with a file that stores the data saved with the level editor. Currently, only one file is saved so multiple levels cannot be saved. The data is saved as a single number per line but when loaded into the game it is converted into a 2d array. This 2d array is used to draw the level.

3.9 Pathing

Enemies had to figure out a path to the end because the path needed to be edited to produce levels instead of predetermined. To that end, some code was used to detect road tiles to determine where to move next. Some adjustments needed to be made to determine if a straight path had ended because the pixel to check the current tile wouldn't be exactly reached because they had fractions of pixels as a movement per update.

```
private void updateEnemyMove(Enemy e) {
    PathPoint currTile = getEnemyTile(e);
    int dir =
roadDirArr[currTile.getyCord()][currTile.getxCord()];

    e.move(GetSpeed(e.getEnemyType()), dir);

    PathPoint newTile = getEnemyTile(e);

    if (!isTilesTheSame(currTile, newTile)) {
        if (isTilesTheSame(newTile, end)) {
            e.kill();
            playing.removeOneLife();
            return;
        }
        int newDir =
```

```
roadDirArr[newTile.getyCord()][newTile.getxCord()];
        if (newDir != dir) {
            e.setPos(newTile.getxCord() * 32, newTile.getyCord() *
32);
            e.setLastDir(newDir);
        }
    }
}
```

Table 7.

```
private void setDirectionAndMove(Enemy e) {
    int dir = e.getLastDir();

    int xCord = (int)(e.getX() / 32);
    int yCord = (int)(e.getY() / 32);

    fixEnemyOffsetTile(e, dir, xCord, yCord);

    if (isAtEnd(e))
        return;

    if (dir == LEFT || dir == RIGHT) {
        int newY = (int)(e.getY() + getSpeedAndHeight(UP,
e.getEnemyType()));
        if (getTileType((int) e.getX(), newY) == ROAD_TILE)
            e.move(GetSpeed(e.getEnemyType()), UP);
        else
            e.move(GetSpeed(e.getEnemyType()), DOWN);
    } else {
        int newX = (int)(e.getX() + getSpeedAndWidth(RIGHT,
e.getEnemyType()));
        if (getTileType(newX, (int) e.getY()) == ROAD_TILE)
            e.move(GetSpeed(e.getEnemyType()), RIGHT);
        else
            e.move(GetSpeed(e.getEnemyType()), LEFT);
    }
}
```

Table 8.

3.10 Buttons

The jButton class produces static buttons, but the window required different buttons based on what game state was currently active, this means that a custom button class was necessary. The button class handles make the buttons however the display and functionality of each button were handled in each scene separately. Additional functionality was handled in the UI package.

3.11 Constants

To make game balancing easier all the values that adjusted gameplay elements such as tower damage/range enemy health/movement and such were placed in a class of their own called constants. Values from this class are loaded in various parts of the game to make it functional. Other things were also stored to make it easier to remember what we were coding. So instead of typing in 0, 1, and 2 to specify which projectile we just imported the projectiles class from constants and typed in arrow/magic/bomb instead. This improved the readability of the code by leaps and bounds.

```
public class Constants {
    public static class Projectiles {
        public static final int ARROW = 0;
        public static final int CHAINS = 1;
        public static final int BOMB = 2;

        public static float GetSpeed(int type) {
            return switch (type) {
                case ARROW - > 8 f;
                case BOMB - > 4 f;
                case CHAINS - > 6 f;
                default - > 0 f;
```

```
        };
    }
}

public static class Towers {
    public static final int CANNON = 0;
    public static final int ARCHER = 1;
    public static final int WIZARD = 2;

    public static int GetTowerCost(int towerType) {
        return switch (towerType) {
            case CANNON - > 65;
            case ARCHER - > 35;
            case WIZARD - > 50;
            default - > 0;
        };
    }

    public static String GetName(int towerType) {
        return switch (towerType) {
            case CANNON - > "Cannon";
            case ARCHER - > "Archer";
            case WIZARD - > "Wizard";
            default - > "";
        };
    }

    public static int GetStartDmg(int towerType) {
        return switch (towerType) {
            case CANNON - > 15;
            case ARCHER - > 5;
            case WIZARD - > 0;
            default - > 0;
        };
    }

    public static float GetDefaultRange(int towerType) {
        return switch (towerType) {
            case CANNON - > 75;
            case ARCHER - > 120;
            case WIZARD - > 100;
            default - > 0;
        };
    }

    public static float GetDefaultCooldown(int towerType) {
```

```
            return switch (towerType) {
                case CANNON - > 120;
                case ARCHER - > 35;
                case WIZARD - > 50;
                default - > 0;
            };
        }
    }

    public static class Direction {
        public static final int LEFT = 0;
        public static final int UP = 1;
        public static final int RIGHT = 2;
        public static final int DOWN = 3;
    }

    public static class Enemies {
        public static final int ORC = 0;
        public static final int BAT = 1;
        public static final int KNIGHT = 2;
        public static final int WOLF = 3;

        public static int GetReward(int enemyType) {
            return switch (enemyType) {
                case ORC, BAT - > 5;
                case KNIGHT - > 25;
                case WOLF - > 10;
                default - > 0;
            };
        }

        public static float GetSpeed(int enemyType) {
            return switch (enemyType) {
                case ORC - > 0.5 f;
                case BAT - > 0.7 f;
                case KNIGHT - > 0.45 f;
                case WOLF - > 0.85 f;
                default - > 0;
            };
        }

        public static int GetStartHealth(int enemyType) {
            return switch (enemyType) {
                case ORC - > 85;
                case BAT - > 100;
                case KNIGHT - > 400;
```

```
                case WOLF - > 125;
                default - > 0;
            };
        }
    }

    public static class Tiles {
        public static final int WATER_TILE = 0;
        public static final int GRASS_TILE = 1;
        public static final int ROAD_TILE = 2;
    }
}
```

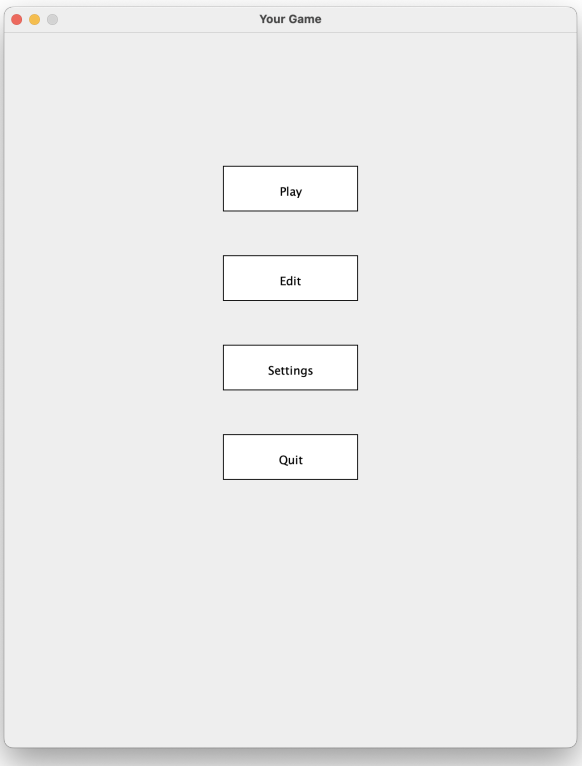
Table 9.

3.12: Economy

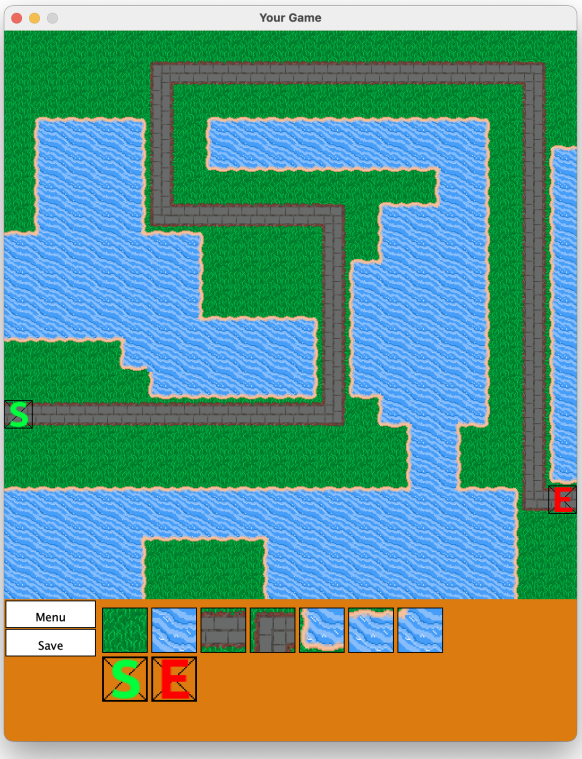
One of the Main elements in a game of tower defense, enemies reward gold, and towers cost gold to place. This requires careful balancing to allow a margin of error initially and later on be quite unforgiving. Gold was handled in the UI for now, though planned to have a class of its own later on.

3.13 GameStates

When run the application opens in the menu state and then there are 4 more states present. The playing state is where all the action happens - the enemies spawn in waves, the players place towers, and the economy flourishes as more and more enemies die. When the playing state has reached a conclusion (win or lose) you reach the game-over state where you can go back to the menu or replay the level. The editing state lets you edit the level you will play on. It has a number of land tiles and water tiles, most of which can be rotated (except for only water and only grass tiles). The settings state is incomplete but was intended to be able to change resolution and the refresh rate. A custom mouse listener was used and passed onto each of these states to make functional buttons as the jButton class could not be used.

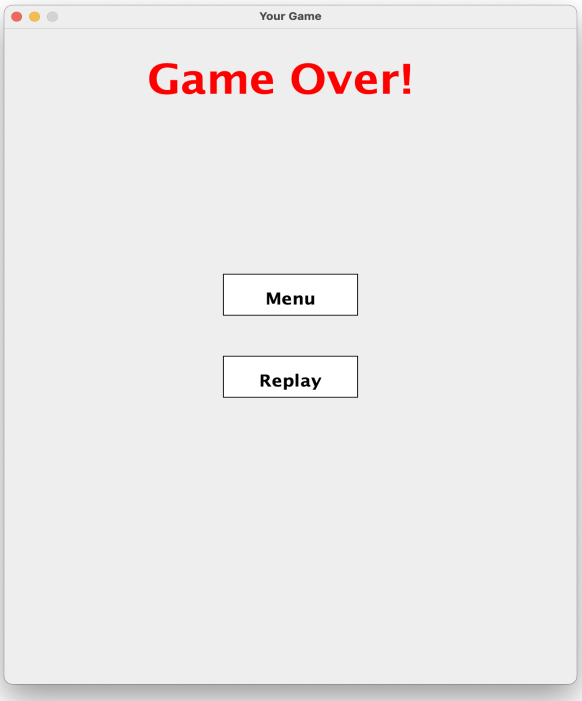


This menu represents the main menu of the game.
The players can select from the options in the game menu.



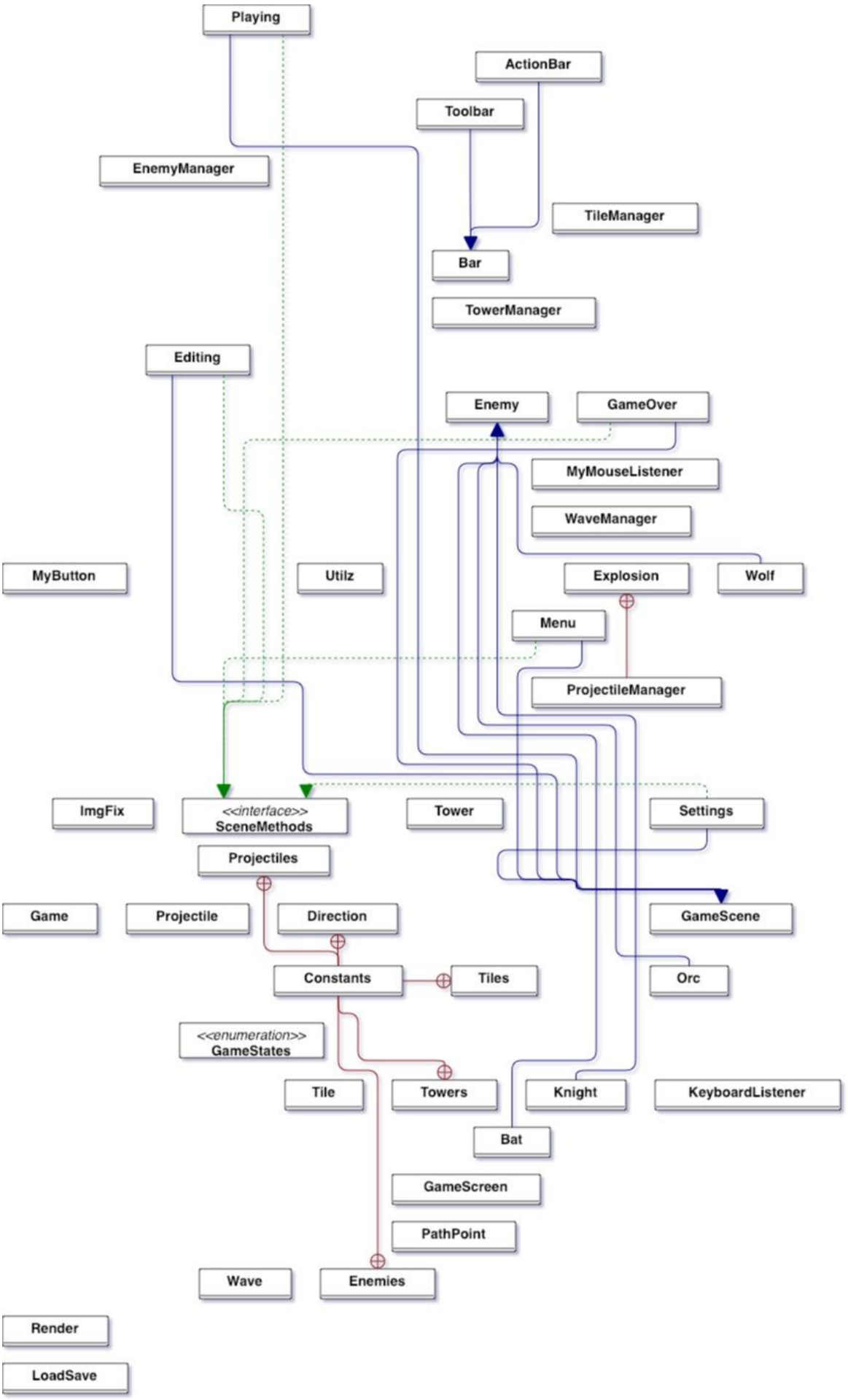
This screen shows the level editor where the designers can design a level. This is also the place to explore your creativity on how to use the game platform.

This is the gameplay screen. Players can place their towers and defend against enemy attacks. They can also upgrade their weapons with gold. Players also have the option to pause and resume the game in this version.



This screen shows the end game section. Players can choose to start a new game or go to the main menu section.

3.14 Class Diagram



3.15 Conclusion

For now, we have a working demo with combat, placeable towers, a working economy, some animations, and more. Due to time constraints, the game could not be improved further. But as it stands, this is an excellent playable and enjoyable game. We have tested this out with our friends and family. Everyone really liked the game and encouraged us to make it bigger. The internal game engine that we developed has been working great for us for the time being. But a real game engine can enhance the experience ten-folds.

Chapter 4: Future Goals

Chapter 4: Future Goals

Additional plans were to finish the settings adjustments players could make so that they can change the resolution of the game to fit their screens. Images would need to be resized accordingly and this also requires additional time spent coding or making more detailed sprite images because the ones currently used are 32x32 pixels. Audio tracks would also be added later on in the production cycle because for a good game a catchy audio track is extremely important. Additionally, soundtracks for the different enemies, towers, projectiles, and hits are required, because without sound the game is very lackluster. More enemies and towers with unique interactions between them would also be very important because for a good game the initial levels should be easy to complete but to get through harder levels and higher difficulties the towers should have interactions the player can learn and exploit to pass them. A campaign with multiple pre-built levels and difficulty adjustments would be necessary in the long run as well for a complete game. When all this is complete the game requires balancing because an interesting game rewards thoughtful interactions with the game. It should not be so difficult that no mistakes may be made but not so easy either that any number of mistakes can be made. One very important aspect that exists in many games today is the ability to play with friends. One of the largest requirements for many people is that they can play games with family, a loved one, or friends. Social interaction now rarely occurs away from your mobile devices and thus there tends to be some craving for it everywhere. A game which caters to this social interaction need is therefore much more likely to succeed then a game without. Further time spent on this should accomplish many of these goals.

Chapter 5: Impact & Sustainability

Chapter 5: Impact & Sustainability

5.1 Impact

A game such as this has the potential to make money on an app store such as steam or google play so long as it attracts customers because of interesting mechanics and visuals. On the whole, the impact of such a game is not major as it is simply a source of entertainment. Playing such games does however require a unique set of skills - an understanding of the economy and strategy elements and can be educational in teaching such elements. Studies have shown that playing games can yield more brain development in youth. Additionally games of this genre create a feeling of solidarity to some extent as the main goal in most games of this type is to defend homes against invaders. This can be seen as breeding a patriotic tendency, whose desirability is not the subject of this topic.

5.2 Sustainability

Games have a life cycle of their own and this cycle can be extended as long as features - such as new levels/enemies/towers/terrain is added in regular intervals. Such games are usually popular for a few months at their peak before dwindling to nothingness. More popular ones can sustain public interest for a much longer time. For this game in particular because it targets a wave of nostalgia from the 2005s the target audience would be people aged around 16-24 at that time because they would be the ones to have played such games the most. In our current society to stay relevant in games means to be able to put in the effort and put out updates nearly every week and so there is some difficulty but with some effort, it should be

possible to extend the life cycle of our game. Furthermore, the bigger our game grows with the effort the more people it may draw in, and the more people that find interest in this game the more likely they will recommend it to others and then play the game as well. All this works like a positive feedback loop that will continuously increase the player base in an ideal scenario. When the player base grows sufficiently adding in some multiplayer features will likely extend its life cycle to years instead of months.