Even Semester (2022)

BINUS UNIVERSITY

BINUS INTERNATIONAL

Final Project Cover Letter

(Individual Work)

Student Information:

Surname: Sutiono **Given Name:** Rafael **Student ID:** 2602174535

Course Code : COMP6699001 **Course Name :** Object-Oriented Programming

Class : L2CC **Lecturer :** Jude Joseph Lamug Martinez, MCS

Type of Assignment : Final Project Report

Submission Pattern

Due Date **:** 16 June 2023 **Submission Date :** 15 June 2023

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offences which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Rafael Sutiono

Table of Contents

A. Project Specification

My project, OverwatchTD, is a tower defense game that runs on the Java Swing GUI. The ‘Overwatch’ in the game is a reference to the sprites and tower attributes used in-game which are from the Overwatch lore. The game runs on a simple loop that uses threading in order to separate the frame and game event tick rate, set at 120 and 60 per second respectively.

The game has a total of 4 screens, which will be referred to as ‘Scenes’ for better understanding. Upon running the game, the player is introduced to the Menu scene, which displays the OverwatchTD logo and the Play, Edit, and Quit buttons below. First, clicking on the Edit button introduces the player to the Editing scene,

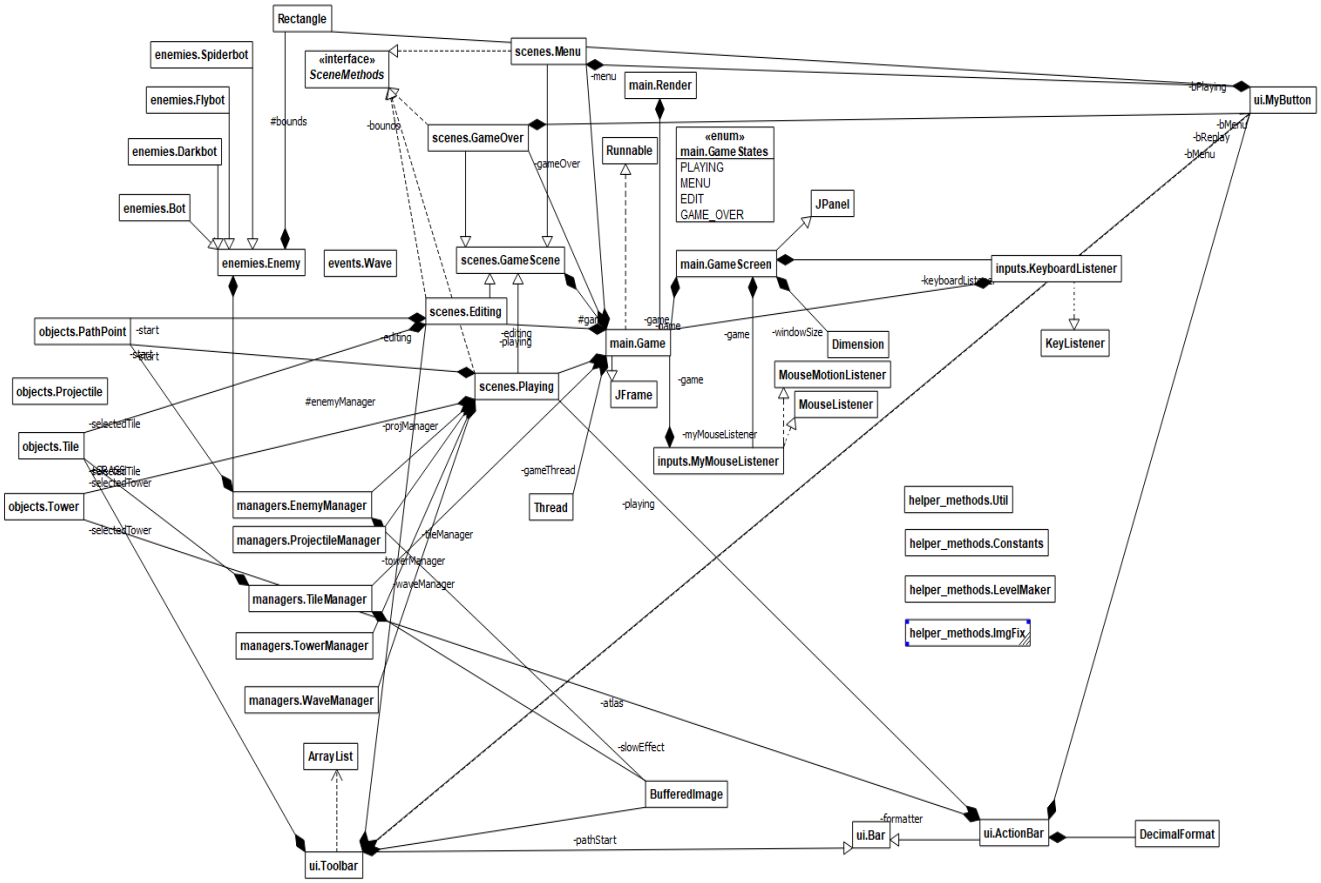
where players are able to create their own level layouts. Note that the player has to quit the game after editing a level before playing it in order for the enemy pathfinder to update. Second, clicking on the Play button takes the player to the main game, where they have to fend off enemies using towers in order to win the game. Last (and the least), clicking on the Quit button terminates the program.

To edit a level, the player is able to select the available sprites in the toolbar, drag the sprites on the level by holding down the mouse left-click, rotate images by pressing ‘R’ on the keyboard, and set the starting and ending points on an edge tile in the level.

The gameplay of OverwatchTD runs just like a standard tower defense game. Towers can be upgraded twice, allowing for longer range, increased attack power and faster fire rates. Enemies spawn in set wave layouts and they are spawned offset from one another using a set tick rate. The player has a set amount of lives (1 by default) and if the player loses the game, they are taken to the game over screen where they have the option to replay the game or go back to the menu scene.

B. Solution Design (Class Diagram)

Shown below is my class diagram for this entire project:



The classes used in the diagram are further explained below, in alphabetical order and including their dependencies (the referencing of instances aren't included). Note that java.util structures aren't included in the descriptions below and will be explained later on:

1. ActionBar: the game's action bar.

2. Bar: base class for a bar. It is the parent class for the ActionBar and Toolbar.

3. Bot: class for the Bot enemy.

4. Constants: class for storing constants.

5. Darkbot: class for the Darkbot enemy.

6. Editing: class for the Edit scene.

7. Enemy: base class for enemies. It is the parent class for all enemy types (Bot, Darkbot, Flybot, and Spiderbot). It makes use of the java.awt module for enemy hitboxes.

8. EnemyManager: manages the enemies in the game. It makes use of the java.awt module for storing and loading enemy sprites.

9. Flybot: class for the Flybot enemy.

10. Game: where the game loop runs. It makes use of the JFrame module and implements Runnable for running the game loop.

11. GameOver: class for the Game Over scene. It makes use of the java.awt module for drawing objects.

12. GameScene: base class for scenes. It is the parent class for all game scenes (Editing, GameOver, Menu, and Playing).

13. GameScreen: class for the game window. It makes use of the JPanel module for handling graphical calculations and the java.awt module to set the window dimensions.

14. GameStates: an enumeration representing different game states.

15. ImgFix: class for building and rotating sprites using the java.awt module.

16. KeyboardListener: listens to keyboard events using the necessary modules from java.awt.

17. LevelMaker: handles level creation in the game. It makes use of the java.io module for reading, saving, and writing level files and the java.awt module to return sprites from the spritesheet.

18. Menu: class for the Menu scene. It makes use of the java.awt module for drawing objects.

19. MyButton: base button class for text and sprite buttons. It makes use of the java.awt module for setting button attributes and bounds.

20. MyMouseListener: listens to mouse events using the necessary modules from java.awt.

21. PathPoint: class for path points.

22. Playing: class for the Playing scene. It makes use of the java.awt module for drawing objects.

23. Projectile: class for projectiles in the game. It makes use of the java.awt module for handling projectiles.

24. ProjectileManager: manages projectiles in the game. It makes use of the java.awt module for handling projectiles and recycling inactive ones.

25. Render: class for rendering the game using the java.awt.Graphics module.

26. SceneMethods: an interface that stores essential scene methods. It is implemented by all the game scene classes (Editing, GameOver, Menu, and Playing).

27. Spiderbot: class for the Spiderbot enemy.

28. Tile: class for static and animated tiles. It makes use of the java.awt module for initializing sprite images.

29. TileManager: manages tiles in the game. It makes use of the java.awt module for storing sprite data.

30. Toolbar: the toolbar in the Edit scene. It makes use of the java.awt module for drawing objects.

31. Tower: class for towers in the game.

32. TowerManager: manages the towers in the game. It makes use of the java.awt module for getting tower sprites.

33. Util: provides utility functions. Mostly contains the implementation of the enemy pathfinder.

34. Wave: class for enemy waves.

35. WaveManager: manages enemy waves.

C. Implementations and Essential Algorithms

Listed below are the most essential bits of the game code:

1. Game Loop

```
      public void run() {

double timePerFrame = 1000000000.0 / FPS_SET; // duration for each frame to be
displayed
// 1 billion nanoseconds = 1 second, calculation set to 120 FPS
double timePerUpdate = 1000000000.0 / UPS_SET; // duration for each event
update, set as 60 UPS

long lastFrame = System.nanoTime(); // duration the last frame was displayed
long lastUpdate = System.nanoTime(); // same but for update
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; // if current frame is being displayed longer or equal to the
set time per frame, a new frame will be painted and the current frame's time
is saved for the next frame to refer to
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();
```

```
// } // used to track FPS and UPS

}

}
```

The game loop is responsible for continuously updating and rendering the game state to provide a smooth and interactive experience for the player by using the Runnable module which allows for the usage of threads. I referred to this forum https://gamedev.stackexchange.com/questions/160329/java-game-loop-efficiency for creating my own.

First, the variables timePerFrame and timePerUpdate are calculated based on the desired frames per second (FPS_SET) and updates per second (UPS_SET) for the game. These variables determine the duration for each frame be displayed and each game update to occur. Second, the variables lastFrame and lastUpdate store the timestamps of the last frame and update, respectively. These timestamps are used to calculate the elapsed time since the previous frame and update. Third, the variables frames and updates keep track of the number of frames and updates that have occurred. Fourth, the while (true) loop represents the main game loop, which runs indefinitely until the game is terminated. Inside the loop, the current time (now) is obtained using System.nanoTime() to ensure high precision.

The code checks if the elapsed time since the last frame is greater than or equal to timePerFrame. If it is, the repaint() method is called to update the graphical representation of the game state. The lastFrame variable is updated with the current time, and the frames counter is incremented. Similarly, the code checks if the elapsed time since the last update is greater than or equal to timePerUpdate. If it is, the updateGame() method is called to update the logic and internal state of

the game. The lastUpdate variable is updated with the current time, and the updates counter is incremented. By continuously repeating said checks, the game loop ensures that the game is rendered at the desired frame rate and updated at the desired update rate.

2. Input Listeners

```
      public void initInputs() { // initiate input listeners
myMouseListener = new MyMouseListener(game);
keyboardListener = new KeyboardListener(game);

addMouseListener(myMouseListener);
addMouseMotionListener(myMouseListener);
addKeyListener(keyboardListener);

requestFocus(); // to have the focus of all inputs given
}
```

This method represents the initialization of input listeners for the game. Input listeners are used to detect and handle user input, such as mouse clicks, mouse movements, and keyboard inputs.

The method initInputs() is responsible for setting up the input listeners, and two input listener objects are created: myMouseListener and keyboardListener. These listener objects are instantiated with their respective input classes.

The addMouseListener() method is called to register the myMouseListener object as a listener for mouse events, allowing the game to receive and respond to mouse clicks, while the addMouseMotionListener() method is called to register the myMouseListener object as a listener for mouse motion events, allowing the game to detect and respond to mouse movements. The addKeyListener() method is also

called to register the keyboardListener object as a listener for keyboard events, allowing the game to capture and respond to keyboard inputs.

Last but not least, the requestFocus() method is called to ensure that the game can listen to and process input events correctly.

3. Image/Sprite Loader

```
public static BufferedImage getSpriteAtlas() { // return buffered img from
spriteatlas
BufferedImage img = null;
InputStream is =
LevelMaker.class.getClassLoader().getResourceAsStream("spriteatlas.png");

try {
img = ImageIO.read(is);
} catch (IOException e) {
e.printStackTrace();
}
return img;
}
```

The above method is what makes loading the sprites used in the game possible. It is declared as public static, indicating that it can be accessed from other classes without the need to create an instance of the class it belongs to. The paragraph below explains how this algorithm works.

Inside the method, a BufferedImage variable named img is declared and initialized to null. This variable will store the loaded sprite atlas image. Then, an InputStream named ‘is’ is created to read the sprite atlas image file, and the getResourceAsStream() method is called on the ClassLoader of the LevelMaker class to retrieve the image file as an input stream. The image file, "spriteatlas.png",

which should be there when the player downloads the .zip file from GitHub, is located in the classpath. The code then enters a try-catch block to handle any potential IOException that may occur when reading the image file, and within this try block, the ImageIO.read() method is called with the input stream ‘is’ as the argument. This method reads the image file from the input stream and returns a BufferedImage object representing the loaded image and so this image is assigned to the img variable.

If an IOException occurs during the image reading process, the catch block is executed. In this case, the exception is printed using the printStackTrace() method, which helps in identifying the cause and location of the exception. Finally, the method returns the img variable, which contains the loaded sprite atlas image.

4. Reading, Writing, and Saving Level Files

```
public static void CreateLevel(int[] idArr) { // creates a new level file if it
doesn't exist yet
if (lvlFile.exists()) {
System.out.println("File: " + lvlFile + " already exists!");
return;
} else {
try {
lvlFile.createNewFile();
} catch (IOException e) {
e.printStackTrace();
}

// add start & end points to the new level.
WriteToFile(idArr, new PathPoint(0, 0), new PathPoint(0, 0));
}

}

private static void WriteToFile(int[] idArr, PathPoint start, PathPoint end) {
try {
PrintWriter pw = new PrintWriter(lvlFile);
```

```
for (Integer i : idArr)
pw.println(i); // to write each element on a new line
// write x & y coordinates for start & end points:
pw.println(start.getxCord());
pw.println(start.getyCord());
pw.println(end.getxCord());
pw.println(end.getyCord());

pw.close(); // close printwriter and flush any remaining data into file
} catch (FileNotFoundException e) {
e.printStackTrace(); // error msg
}

}

public static void SaveLevel(int[][] idArr, PathPoint start, PathPoint end) {
// save level data, check if file exists before saving
if (lvlFile.exists()) {
WriteToFile(Util.TwoDToArrayList(idArr), start, end);
} else {
System.out.println("File: " + lvlFile + " does not exist! ");
return;
}
}

private static ArrayList<Integer> ReadFromFile() { // read level data and
return as arraylist
ArrayList<Integer> list = new ArrayList<>();

try {
Scanner sc = new Scanner(lvlFile);

while (sc.hasNextLine()) {
list.add(Integer.parseInt(sc.nextLine()));
}

sc.close();

} catch (FileNotFoundException e) {
e.printStackTrace();
}

return list;
}

public static ArrayList<PathPoint> GetLevelPathPoints() { // read path point
data and return as arraylist
if (lvlFile.exists()) {
ArrayList<Integer> list = ReadFromFile();
```

```
ArrayList<PathPoint> points = new ArrayList<>();
points.add(new PathPoint(list.get(400), list.get(401)));
points.add(new PathPoint(list.get(402), list.get(403)));

return points;

} else {
System.out.println("File: " + lvlFile + " does not exist! ");
return null;
}
}

public static int[][] GetLevelData() { // reads and converts level data into a
2d array
if (lvlFile.exists()) {
ArrayList<Integer> list = ReadFromFile();
return Util.ArrayListTo2D(list, 20, 20);

} else {
System.out.println("File: " + lvlFile + " does not exist! ");
return null;
}

}
```

This set of methods is what enables the game to read, write, and save level files so that the player is able to save their own level creations. Below are each of the methods explained:

1. `CreateLevel(int[] idArr)`: This method creates a new level file if it doesn't already exist. It checks if the level file exists using the exists() method. If the file exists, it prints a message and returns. Otherwise, it creates a new file using the createNewFile() method. After creating the file, it calls the WriteToFile() method to write the level data (specified by the idArr array) and the start and end points to the file.

2. `WriteToFile(int[] idArr, PathPoint start, PathPoint end)`: This method writes the level data, start point, and end point to the level file. It creates a

PrintWriter object (pw) to write data to the file. It iterates over the idArr array and writes each element on a new line using the println() method. Then, it writes the x and y coordinates of the start and end points on separate lines. Finally, it closes the PrintWriter to flush any remaining data into the file.

3. `SaveLevel(int[][] idArr, PathPoint start, PathPoint end)`: This method is responsible for saving the level data. It checks if the level file exists using the exists() method. If the file exists, it calls the WriteToFile() method to write the level data, start point, and end point to the file. Otherwise, it prints an error message and returns.

4. `ReadFromFile()`: This private method reads the level data from the level file and returns it as an array list containing int values. It creates an ArrayList<Integer> called list to store the data, uses a Scanner to read each line from the file, parse it as an Integer, and add it to the list. Finally, it closes the Scanner and returns the list.

5. `GetLevelPathPoints()`: This method reads the path point data from the level file and returns it as an ArrayList<PathPoint>. It first checks if the level file exists. If the file exists, it calls the ReadFromFile() method to retrieve the level data as an ArrayList<Integer>. It then extracts the x and y coordinates of the start and end points from the list and creates PathPoint objects. These PathPoint objects are added to an ArrayList<PathPoint> called points, which is returned. If the file does not exist, it prints an error message and returns null.

6. `GetLevelData()`: This method reads the level data from the level file and converts it into a 2D array (int[][]). It checks if the level file exists. If the file exists, it calls the ReadFromFile() method to retrieve the level data as an ArrayList<Integer>. It then uses a utility method (ArrayListTo2D()) to convert the ArrayList<Integer> into a 2D array with the specified dimensions (20x20). The resulting 2D array is returned. If the file does not exist, it prints an error message and returns null. The method responsible for enabling the retrieval of level data is explained next.

5. Array Lists to 2D Lists, and Vice Versa

```
public static int[][] ArrayListTo2D(ArrayList<Integer> list, int ySize, int xSize) { // takes in an array list (int type) with a set size (20x20) that converts it into a 2d array through the use of a nested loop
int[][] newArr = new int[ySize][xSize];

for (int j = 0; j < newArr.length; j++)
for (int i = 0; i < newArr[j].length; i++) {
int index = j * ySize + i;
newArr[j][i] = list.get(index);
}

return newArr;

}

public static int[] TwoDToArrayList(int[][] twoArr) { // essentially the reverse of the method above
int[] oneArr = new int[twoArr.length * twoArr[0].length];

for (int j = 0; j < twoArr.length; j++)
for (int i = 0; i < twoArr[j].length; i++) {
int index = j * twoArr.length + i;
```

```
oneArr[index] = twoArr[j][i];
}

return oneArr;
}
```

A convenient way to convert between a two-dimensional array and an array list of integers that are explained below.

The ArrayListTo2D method takes an ArrayList<Integer> called list and the desired dimensions of the resulting two-dimensional array (ySize and xSize) to create a new two-dimensional array with the specified dimensions and then use nested loops to iterate over the rows and columns of the array. Within the loops, it calculates the corresponding index in the list based on the current row and column. It retrieves the value from the list using list.get(index) and assigns it to the corresponding position in the two-dimensional array. Finally, it returns the resulting two-dimensional array.

On the other hand, the TwoDToArrayList method takes a two-dimensional array called twoArr and performs the reverse operation, calculating the size of the resulting one-dimensional array by multiplying the number of rows and columns of twoArr. It creates a new one-dimensional array with the calculated size and then uses nested loops to iterate over the rows and columns of twoArr. Within the loops, it calculates the corresponding index in the one-dimensional array based on the current row and column, assigning the value from twoArr at the current position to the corresponding index in the one-dimensional array. Finally, it returns the resulting one-dimensional array.

6. Rotating Images/Sprites

```
public static BufferedImage getRotImg(BufferedImage img, int rotAngle) {

int w = img.getWidth();
int h = img.getHeight();

BufferedImage newImg = new BufferedImage(w, h, img.getType());
Graphics2D g2d = newImg.createGraphics();

g2d.rotate(Math.toRadians(rotAngle), w / 2, h / 2); // rotate img on its center
-> w/2 and h/2
g2d.drawImage(img, 0, 0, null);
g2d.dispose();

return newImg;

}
```

This method allows the game to rotate a sprite by a specified angle and return its rotated version by utilizing the Graphics2D module to rotate an input image by the specified angle. It creates a new BufferedImage object with the same dimensions and image type as the input image, performs the rotation using the rotate() method, draws the input image onto the rotated image, and returns the resulting rotated image. Another version of this method exists in the same file to rotate a top-layer sprite without rotating the animated water tile below it, which follows this sequence instead:

```
// sequence: draw water tile -> rotate 2nd img -> draw new img -> dispose of
the previous one -> put new img in the array
g2d.drawImage(imgs[i], 0, 0, null);
g2d.rotate(Math.toRadians(rotAngle), w / 2, h / 2);
g2d.drawImage(secondImage, 0, 0, null);
```

```
g2d.dispose();
```

7. Projectile and its Trajectory

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

int xDist = (int) (t.getX() - e.getX()); // x distance = tower X - enemy X
int yDist = (int) (t.getY() - e.getY()); // y distance = tower Y - enemy Y
int totDist = Math.abs(xDist) + Math.abs(yDist);

// formula to calculate x & y speed using xPer, which is the x-distance over
the actual distance
float xPer = (float) Math.abs(xDist) / totDist;

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

if (t.getX() > e.getX())
xSpeed *= -1; // x gets translated to the left instead of the right if tower's
X is above the enemy X
if (t.getY() > e.getY())
ySpeed *= -1; // y gets translated downwards instead of upwards if tower's Y
is above the enemy Y

float rotate = 0;

if (type == ARROW) {
float arcValue = (float) Math.atan(yDist / (float) xDist); // formula to
calculate angle of rotation using arc-tan value
rotate = (float) Math.toDegrees(arcValue); // rotate according to arc-tan value
converted to deg

if (xDist < 0) // negative x-distance value -> + 180 deg
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));
```

```
}
```

This method is called when an enemy is within attacking range of a defense tower, creating a new projectile object and initializes its properties based on the tower and enemy involved. Personally, this is the algorithm I had the most fun with since it applies various mathematical formulae to determine the trajectory of a projectile and to rotate an arrow projectile relative to an enemy's position.

The method begins by determining the type of projectile based on the tower using the getProjType method and then calculating the x-distance and y-distance between the tower and the enemy. After that, we need to calculate the x-speed and y-speed of the projectile. The x-speed is determined by the ratio of the absolute x-distance to the total distance between the tower and the enemy and the y-speed is calculated as the difference between the projectile's maximum speed and the x-speed.

Based on the relative positions of the tower and the enemy, the x-speed and y-speed may be adjusted by multiplying them by -1 if the tower's x-coordinate is greater than the enemy's x-coordinate or if the tower's y-coordinate is greater than the enemy's y-coordinate (negative x-speed means a translation leftwards and the negative y-speed means a translation downwards).

If the projectile type is an arrow, the method calculates the rotation angle for the projectile. It uses the arc-tangent function (Math.atan) to determine the angle based on the y-distance divided by the x-distance. The angle is then converted to degrees using Math.toDegrees. If the x-distance is negative, indicating that the enemy is to the left of the tower, 180 degrees is added to the rotation angle.

The method then checks for any inactive projectiles of the same type in the projectiles list. If an inactive projectile is found, its properties are updated using the reuse method, including the position, speed, damage, and rotation angle, and the method returns. Conversely, if no inactive projectile of the same type is found, a new projectile is created using the Projectile constructor. The constructor takes the position of the tower with an offset of 16 pixels (the x*y center of the tower), the x-speed, y-speed, damage, rotation angle, projectile ID, and type. The newly created projectile is added to the projectiles list.

8. Changing a Tile in the Editing Scene

```
private void changeTile(int x, int y) {
if (selectedTile != null) { // if not null, tile is already selected

// calculate tile coordinates:
int tileX = x / 32;
int tileY = y / 32;

if (selectedTile.getId() >= 0) { // checks if selected tile has a
non-negative ID. If ID is non-negative, the selected tile is not a
special tile such as the start/end points
if (lastTileX == tileX && lastTileY == tileY && lastTileId ==
selectedTile.getId())
return; // checks if current tile coordinates and selected tile ID are
the same as previous tile coordinates and ID. Both are equal -> tile has
not changed -> no need to update it

lastTileX = tileX;
lastTileY = tileY;
lastTileId = selectedTile.getId(); // update previous tile coordinates
and ID with current tile coordinates and ID

lvl[tileY][tileX] = selectedTile.getId(); // update tile ID in 2D array
with specified coordinates, effectively changing the tile at a specific
position to the selected tile
} else {
int id = lvl[tileY][tileX];
```

```
if (game.getTileManager().getTile(id).getTileType() == ROAD_TILE /*
checks if said tile corresponds to a road tile */) {
if (selectedTile.getId() == -1) // ID == -1 -> start tile
start = new PathPoint(tileX, tileY);
else // ID == 0 -> end tile
end = new PathPoint(tileX, tileY);
}
}
}
}
```

The changeTile method in the Editing class is used to modify tiles at specific coordinates, and it is called when a user interacts with the level editor and selects a tile to place or change.

The method first checks if a tile is currently selected (selectedTile != null). If a tile is selected, the method proceeds to calculate the tile coordinates based on the provided x and y values. The coordinates are calculated by dividing the x and y values by the tile size (32 pixels).

Next, the method checks if the selected tile has a non-negative ID. If the ID is non-negative, it means that the selected tile is not a special tile. In this case, the method checks if the current tile coordinates and the ID of the selected tile are the same as the previous tile coordinates and ID (lastTileX, lastTileY, and lastTileId). If they are the same, it means that the tile has not changed, and there is no need to update it. In such a case, the method simply returns.

If the current tile coordinates and selected tile ID are different from the previous values, the method updates the lastTileX, lastTileY, and lastTileId variables with the current values. Then, it updates the tile ID in the 2D array lvl at

the specified coordinates (tileY and tileX) with the selected tile's ID. This effectively changes the tile at that position to the selected tile.

If the selected tile has a negative ID, it indicates that it is a special tile, i.e. the start or end points. In this case, the method retrieves the ID of the tile at the current coordinates from the lvl array. It checks if the tile corresponds to a road tile (getTileType() == ROAD_TILE). If it does, the method further checks the ID of the selected tile. If the ID is -1, it means the selected tile is the start tile, and a new PathPoint object is created with the current coordinates (tileX and tileY) and assigned to the start variable. If the ID is 0, it means the selected tile is the end tile, and a new PathPoint object is created and assigned to the end variable.

9. Enemy Pathfinder

```
public static int[][] GetRoadDirArr(int[][] lvlTypeArr, PathPoint start,
PathPoint end) { // enemy pathfinder
int[][] roadDirArr = new int[lvlTypeArr.length][lvlTypeArr[0].length]; // road
at y,x

PathPoint currTile = start; // main variable for pathfinding
int lastDir = -1; // 0 left, 1 up, 2 right, 3 down

while (!IsCurrSameAsEnd(currTile, end)) { // loops as long as currTile != end
PathPoint prevTile = currTile; // when loop starts, prevtile will become the
current tile
currTile = GetNextRoadTile(prevTile, lastDir, lvlTypeArr); // to get the
direction of the next tile
lastDir = GetDirFromPrevToCurr(prevTile, currTile); // to figure out the
direction from previous tile to the next tile
roadDirArr[prevTile.getyCord()][prevTile.getxCord()] = lastDir; // put prev
tile's coordinates in
```

```
}
roadDirArr[end.getyCord()][end.getxCord()] = lastDir; // executes when we get
the endpoint
return roadDirArr;
}

private static int GetDirFromPrevToCurr(PathPoint prevTile, PathPoint currTile)
{ // direction getter
// up or down:
if (prevTile.getxCord() == currTile.getxCord()) {
if (prevTile.getyCord() > currTile.getyCord())
return UP;
else
return DOWN;
} else {
// left or right:
if (prevTile.getxCord() > currTile.getxCord())
return LEFT;
else
return RIGHT;
}

}

private static PathPoint GetNextRoadTile(PathPoint prevTile, int lastDir,
int[][] lvlTypeArr) { // to find the next tile and try different directions
until it finds a direction that has a road

int testDir = lastDir;
PathPoint testTile = GetTileInDir(prevTile, testDir, lastDir);

while (!IsTileRoad(testTile, lvlTypeArr)) { // if it returns true, then a road
tile has been found
testDir++;
testDir %= 4; // go down to 0 again if it reaches 4
testTile = GetTileInDir(prevTile, testDir, lastDir);
}
```

```
return testTile;
}

private static boolean IsTileRoad(PathPoint testTile, int[][] lvlTypeArr) {
if (testTile != null)
// checks if it is within bounds:
if (testTile.getyCord() >= 0)
if (testTile.getyCord() < lvlTypeArr.length)
if (testTile.getxCord() >= 0)
if (testTile.getxCord() < lvlTypeArr[0].length)
if (lvlTypeArr[testTile.getyCord()][testTile.getxCord()] == ROAD_TILE) // is it
a road tile or no?
return true;

return false;
}

private static PathPoint GetTileInDir(PathPoint prevTile, int testDir, int
lastDir) { // to make sure tiles are coming from correct directions

switch (testDir) {
case LEFT:
if (lastDir != RIGHT)
return new PathPoint(prevTile.getxCord() - 1, prevTile.getyCord());
case UP:
if (lastDir != DOWN)
return new PathPoint(prevTile.getxCord(), prevTile.getyCord() - 1);
case RIGHT:
if (lastDir != LEFT)
return new PathPoint(prevTile.getxCord() + 1, prevTile.getyCord());
case DOWN:
if (lastDir != UP)
return new PathPoint(prevTile.getxCord(), prevTile.getyCord() + 1);
}

return null;
}
```

```
private static boolean IsCurrSameAsEnd(PathPoint currTile, PathPoint end) { //
to check if current tile isn't the end point
if (currTile.getxCord() == end.getxCord())
if (currTile.getyCord() == end.getyCord())
return true;
return false;
}
```

One of the most essential parts of the game is the enemy pathfinder, which generates a path for enemies by determining the directions from each road tile to the next one until reaching the end point. The pathfinding process starts by initializing the roadDirArr and setting the currTile to the start point.

- The lastDir variable is used to store the direction of the previous tile.
- The pathfinding loop continues until the current tile is the same as the end point.
- Inside the loop, the previous tile (prevTile) is assigned the value of the current tile, and the next road tile is obtained using the GetNextRoadTile method.
- The lastDir is updated with the direction from the previous tile to the current tile using the GetDirFromPrevToCurr method, and the direction information (lastDir) is then stored in the roadDirArr for the previous tile.
- After the loop, the direction information for the end point is assigned to the roadDirArr. Finally, the roadDirArr is returned.

Listed below are the supporting methods:

1. `GetDirFromPrevToCurr(PathPoint prevTile, PathPoint currTile)`: Determines the direction (UP, DOWN, LEFT, RIGHT) from the previous tile to the current tile based on their coordinates.

2. `GetNextRoadTile(PathPoint prevTile, int lastDir, int[][] lvlTypeArr)`: Responsible for finding the next road tile based on the previous tile, the last direction, and the level layout. It iterates through different directions until it finds a road tile.

3. `IsTileRoad(PathPoint testTile, int[][] lvlTypeArr)`: Checks if a given tile is a road tile by validating its coordinates and checking the corresponding value in the lvlTypeArr.

4. `GetTileInDir(PathPoint prevTile, int testDir, int lastDir)`: Returns the tile in a specific direction from the previous tile, excluding the opposite direction from the last direction.

5. `IsCurrSameAsEnd(PathPoint currTile, PathPoint end)`: Checks if the current tile is the same as the end point.

10. Data Structures

Data Structure	Description
Array List (1D and 2D)	An array list is essentially a resizable array. It is

	probably the most important data structure that gets this game running. It is used in: - setting the enemies to be spawned in a wave - storing level data, i.e. the tiles used in a level layout by ID - storing the enemy pathfinder layout - storing enemies, towers, projectiles, and explosion animations - storing map button images
Hashmap	A hashmap allows for the storing of key and value pairs, where keys should be unique. It is used in assigning different tiles (sprites) with unique IDs.

11. Modules

Module	Description
javax.swing	An API for providing a graphical user interface for Java programs. It is used in this project for rendering the game window.
java.awt	An API to develop Graphical User Interface (GUI) or Windows-based applications in Java. It is used in this project for:

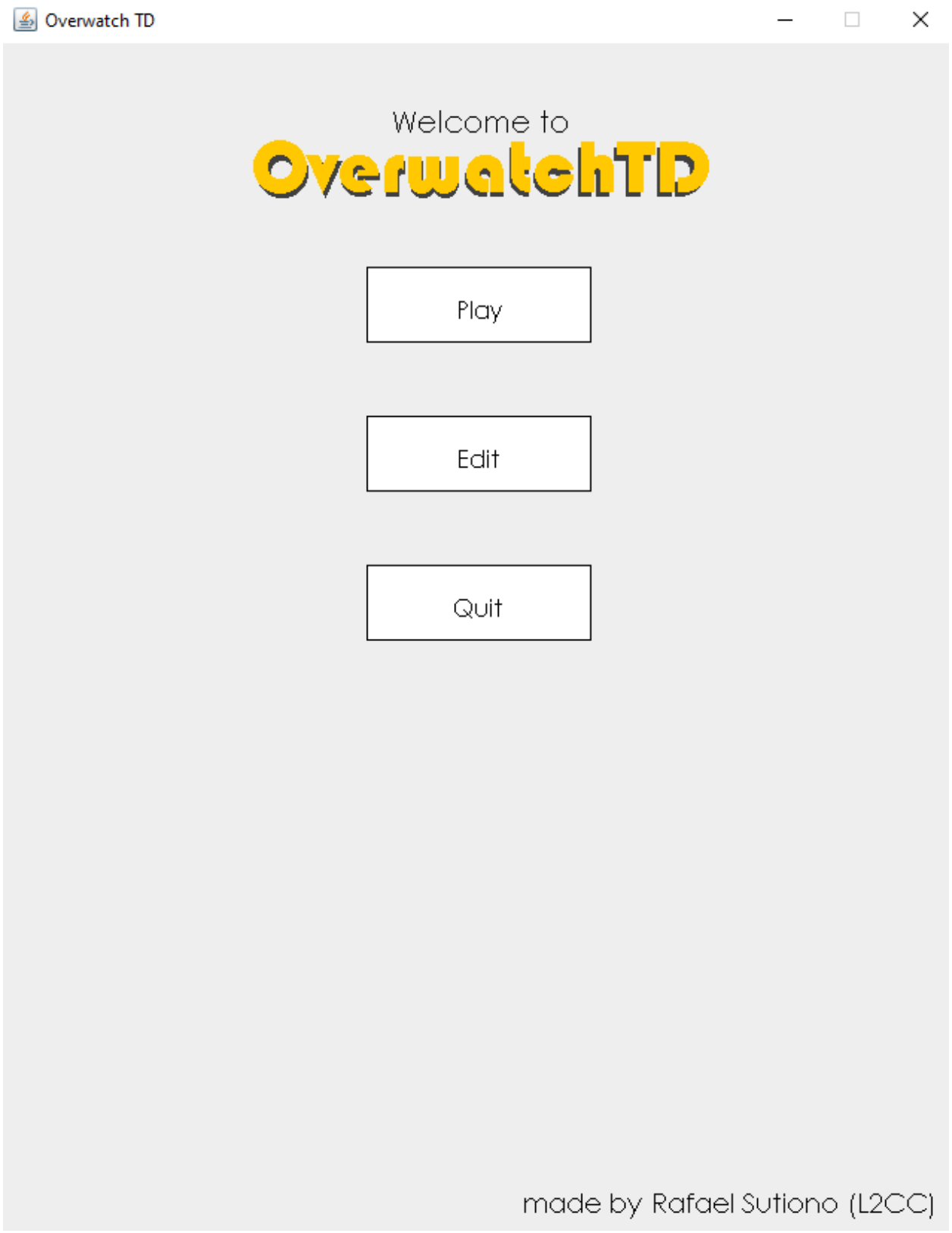
	- setting the window size rendering and rotating in-game images - displaying text - projectile movement - forming object bounds using the built-in Rectangle class - implementing input listeners for the game
java.io	A package for reading and writing data. It is used in this project to: - read, write, and save level files - return buffered images from the spritesheet
java.text	A package that provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages. It is used in the project to format a timer used in the game to 1 decimal place.

D. Evidence of Working Program

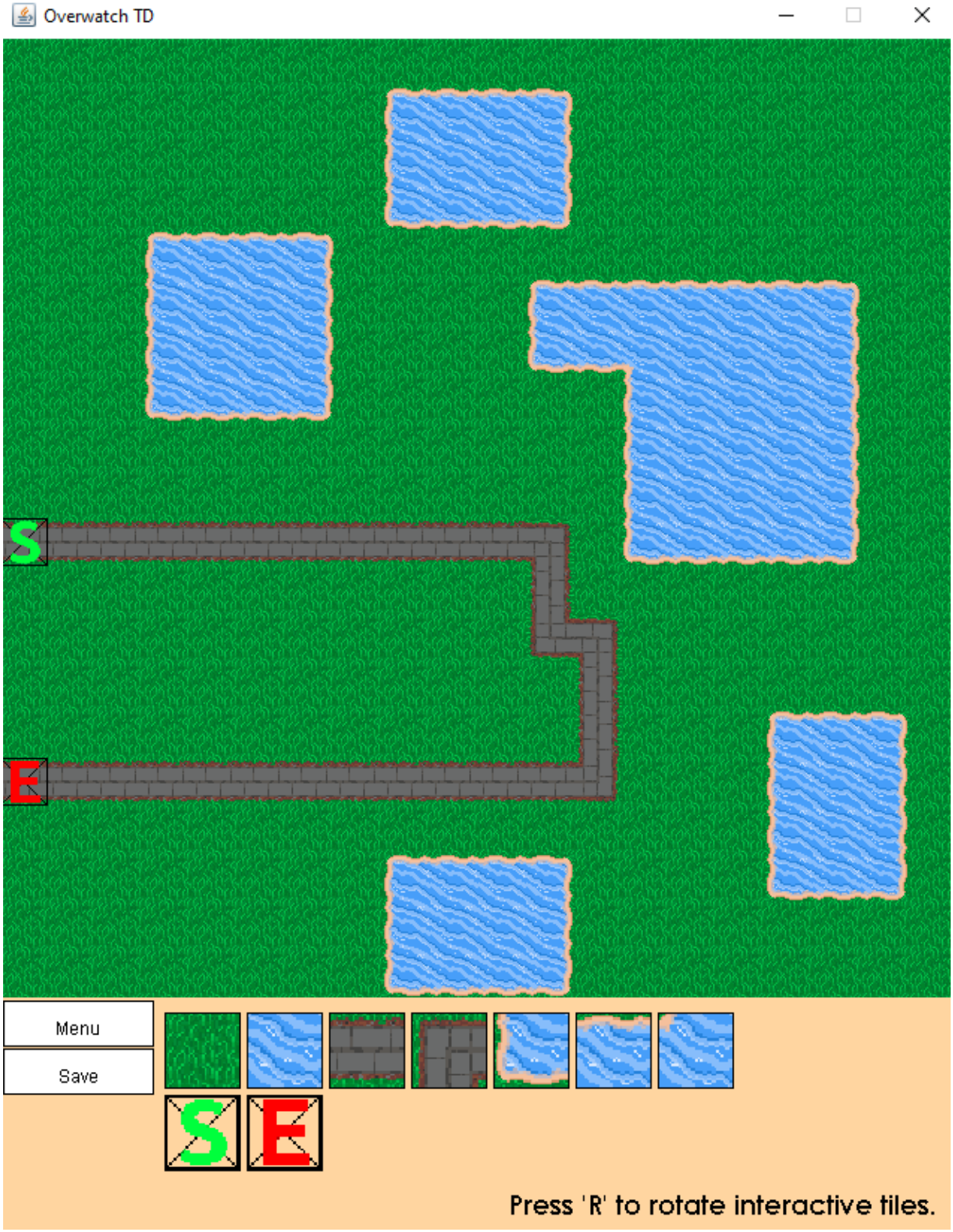
Listed below are the game's functions and the player's abilities to perform actions in-game:

- Player can access all 4 game scenes
- Player can place down and upgrade towers, on grass tiles only
- Player can earn gold
- Player cannot place down towers if gold isn't sufficient
- Enemies follow a fixed path from start to end point according to pathfinder
- Enemies are detectable and can be killed by towers if within range
- Playing scene switches to Game Over scene if player loses all lives
- Player loses a life if an enemy reaches the end point
- Player can pause the game
- Player can edit level layout and set start and end points
- Buttons work as intended

Here are in-game screenshots:

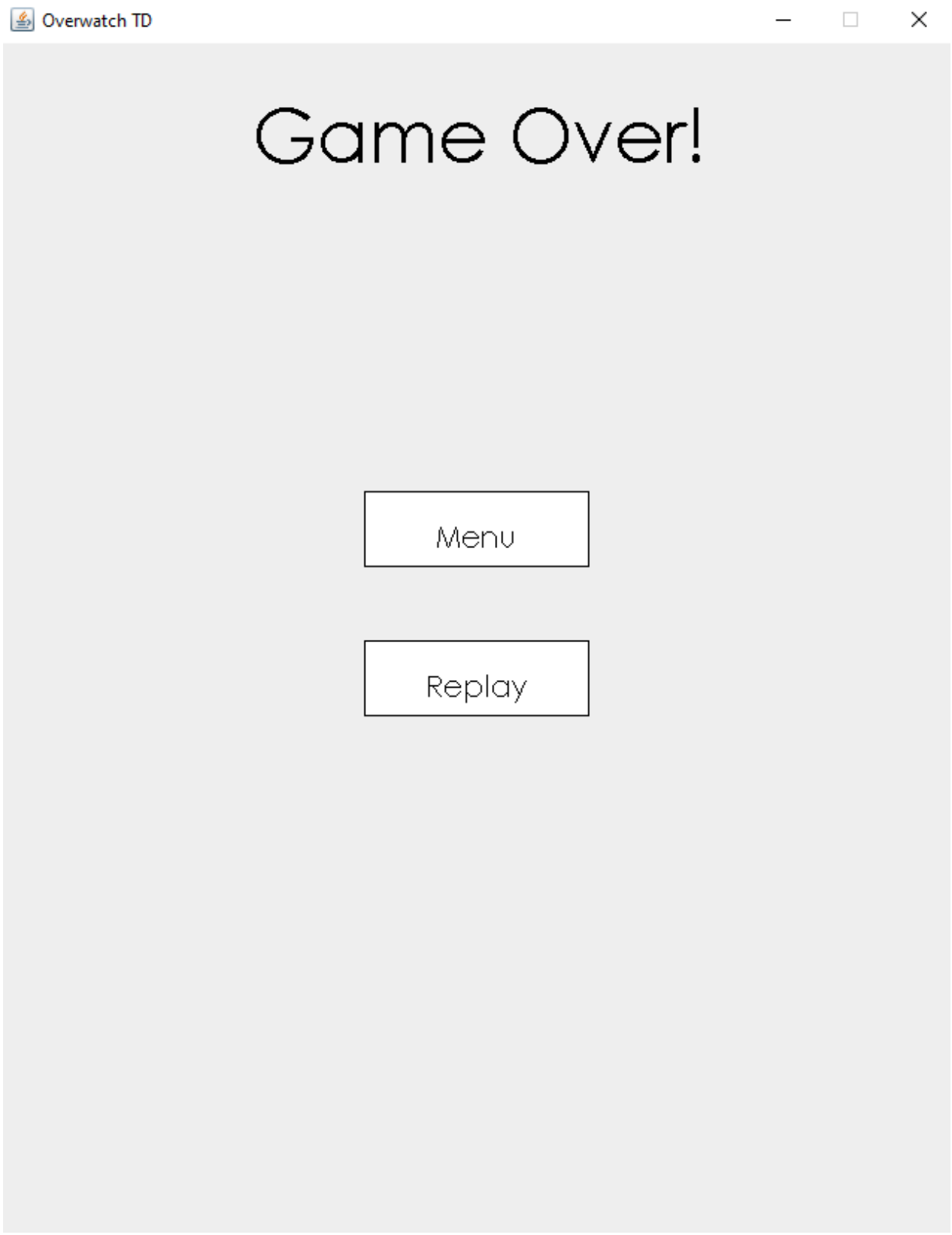

Overwatch TD
Welcome to
OverwatchTD
Play
Edit
Quit
made by Rafael Sutiono (L2CC)

*Menu scene



**Editing scene*

**Playing scene in paused state, with towers firing at enemies*



*Game Over scene

While the game works as intended, I did not have enough time to implement a win screen. As a result, the game sits idle after the final wave as shown in the video demo. I also did not upload a runnable jar for the player to download as I want the player to be able to set their own enemy waves, spawn tick rates, constants such as enemy hitpoints, tower fire rates, and firepower, and tower costs so they could balance the game better as I did not have enough time for that. It was a fun (gruelingly long though) project to make, but many more things could have been implemented.