Table of Contents

1. Introduction

1.1. Project Overview:

My project is a typical Tank Wars game where two players exist that, each with the objective to win the game by firing bullets at the other player until the other player has no more lives. The game is Super Mario Bros. themed, and uses power ups to replenish players' health, lives, and give them bulletproof abilities. To realize this game Java Swing and other libraries to develop GUIs. The game allows two players, and the objective is to basically not die. The tanks are able to shoot one another and reduce the health count and life count of the other tank, and if they life count of one tank reaches zero, then that player loses. They play the game within the game panel, and also shoot breakable walls to create navigate more easily. Throughout the game a player can replenish itself with two kinds of power ups; one power up gives the player an extra life, another replenishes its health, and the other allows the player to be bullet proof for a certain amount of time. Each player starts out with 3 lives and is able to obtain a maximum of 5 lives; 1 life = 100 health counts, and each bullet fired at a tank reduces its health count by 10. Once the game is over, we are told which player won, and the player can exit.

1.2. Summary of Work Completed

For this project I was able to successfully complete all the requirements for the Tank Wars game. The game begins with a Start Screen which allows the player to press Start to begin the game, or exit. The game has 2 players aka tanks, that move forwards, backwards, rotate in all directions, and can shoot bullets. The game begins and has a mini-map in the bottom middle and a split-screen which follows each player. The game also has a stats section in the bottom that displays a players health with text and a health bar; it displays lives with green circles and text; it displays a photo of the tank depending on the player and text to differentiate Player 1 and 2; it also shows text that tells the player whether their tank is bulletproof. In the game, there exist 3 different power-ups which are dispersed. The game is surrounded by unbreakable walls and inside the game, a mixture of breakable and unbreakable walls. The tank can collide with either type of wall, power-ups, and the other tank. The bullets can collide with either type of wall, and the other tank; these bullets disappear after colliding with objects. Each player can successfully shoot another player and reduce the other players health count, as well as life count which is displayed in the stats section. When the game is over, because a player has lost all their lives, a Game Over screen appears, also displaying which player won the game, along with an exit button. Finally, the tank is built into a Jar.

2. Development Environment

Java Version - 16.0.1

IDE - IntelliJ

3. How to Build/Import your Project

3.1. Import

Once you copy the HTTP link from GitHub, from my repository, you can use your terminal to perform git clone and it will clone a local repository, typically on ones desktop, where the folder will then be . An easier way might be to go to GitHub and search for the GitHub repository link (https://github.com/csc413-su21/csc413-tankgame-adadallison). If you press code, now you can see 'Download ZIP files,' which will allow the folder to directly download to your computer. To import the project correctly onto an IDE, open your IDE and selected "import project," select the folder and then press Open. Once the file is open, the Launcher class can run the project, or running the jar file in the jar folder can also do that.

3.2. Build Jar

Once the project is open, the files should populate. To run the project with a built JAR, the first thing that should be done is making sure the Project Structure is accurate, to do this we go to File > Project Structure > Modules > and here we make sure the resources folder is marked as Resources, and the src folder is market as Sources. Then we go to Project Structure > Artifacts > '+' (Add) > JAR > "From module with dependencies…" > now select the main class > Apply. Then exit Files and go to Build > Build Artifacts… > Build. Now in the project folders if you go to out > artifacts > 'main-class-name'_jar > and here you should see your .jar file. Then you can hover over this .jar file and run it.

4. How to Run you Project

4.1. Build

Once the file is open on IntelliJ or another IDE, the Launcher class is where the main method is located, therefore this is where it can be ran. The jar file can also be selected to run the project. Once the project has successfully ran, the player begins the game by pressing start, or the space key. The game panel appears, and once a player loses, the end game panel appears telling the player that the game is over. Here the player must exit the game by pressing the exit button. If the player would like to play again, they can run the Launcher class by right clicking and pressing run, or running the jar the same way.

4.2. Game Controls and Rules

The two players use different keys to control their movements. Each player can move forwards, backwards, rotate, left or right, and shoot bullets. Player 1 uses the up, down, left, right, and enter arrows to navigate. Each key does that is implied, the enter key is for shooting bullets. Player 2 uses the W (up), S (down), A (rotate left), D (rotate right) and F (shoot bullets) keys. To begin the game once the start panel appears, the player can press the space key to begin playing instead of clicking Start.

The only rules are that after having 5 lives while having 100 health count, the mushrooms and coins (power-ups) are no longer effective. Players die when they have zero, lives and this will make them lose the game. They also have a bulletproof power-up, the other players bullets are not effective.

5. Assumption Made

Before coding the game assumption made were that the end panel, start panel, and launcher class worked accurately. Another assumption was that abstractions would be necessary, also that the Java Swing library would be use a lot to draw components. Also that these game objects would have to be divided into moveable and stationary categories. I did not make many assumptions. I assumed that tanks had to each have a certain amount of lives, a status bar to display their lives, and that the power-ups benefitted their livelihood and health.

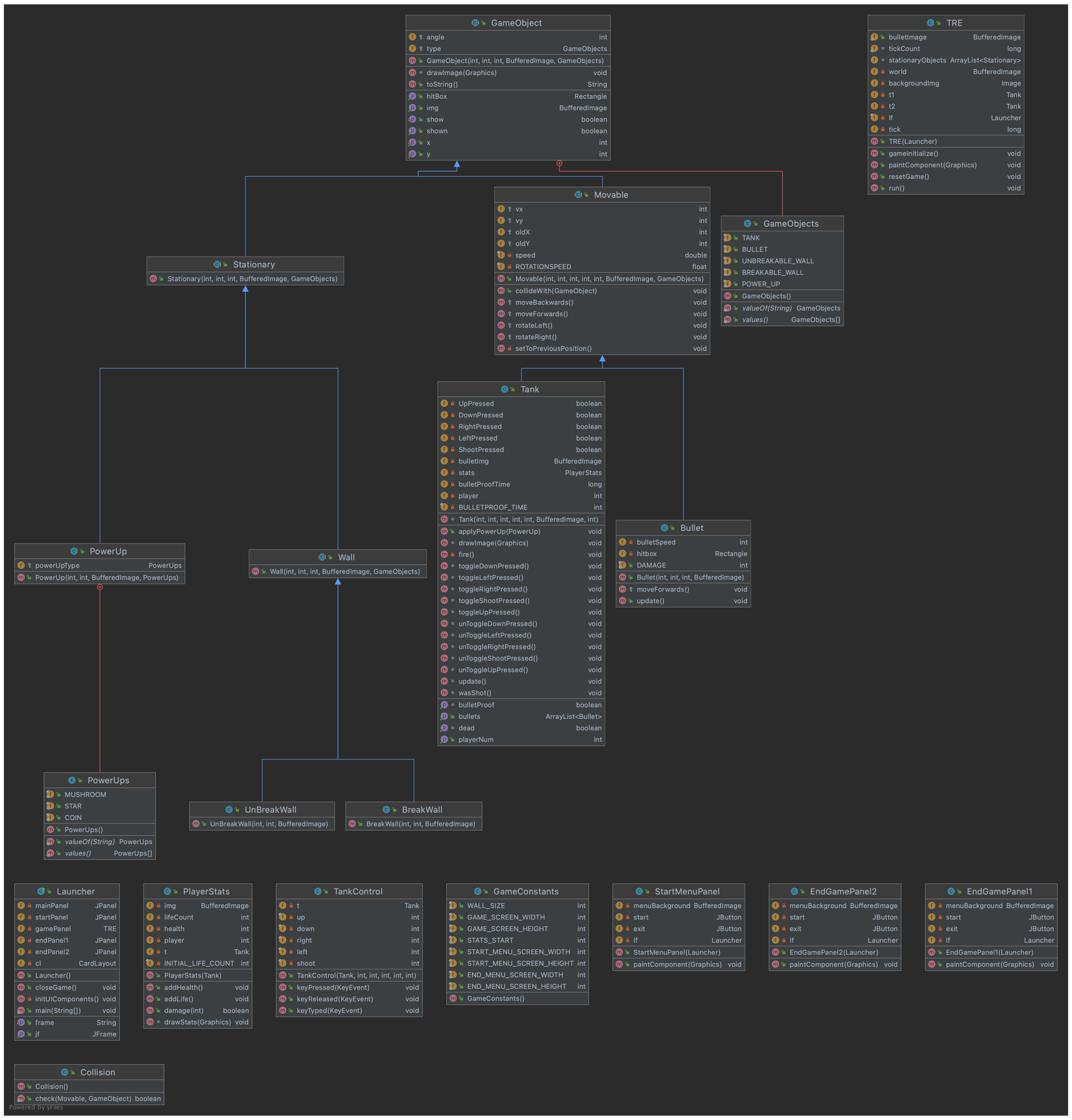
6. Implementation Discussion

6.1. Class Diagram

7. Technical Overview and Class Descriptions

Given the starter code (which includes the tank, tank control, end panel, and start panel classes, game constants, and TRE classes), the Launcher class is the class where the project is ran from and this class was the least changed. This class includes the instantiation of JPanels and a method that switches between the start, game, and end panels. The StartMenuPanel is the first panel that appears when the game is run from Launcher, this was given but I changed the title.png. I also split EndGamePanel into two different EndGamePanel classes (EndGamePanel1 & EndGamePanel2), depending on which player wins, the corresponding panel is called and also contains an exit button. These JPanels exist in the menus package. When the user selects “start” from the StartPanel, the gamePanel class, TRE, is called which includes the method gameInitialize() which sets all game objects to their initial states. Here we assign images (located in our resources package) to our game objects using the BufferedImage subclass, read the map text file to set up the games layout, create tank objects, and modify the keys that control them.

I approached the assignment by creating a UML document that used abstractions to simplify GameObjects, Moveable Objects, and Stationary Objects into corresponding classes. These three classes allows game objects that share certain characteristics to extend that Abstract class. The Moveable and Stationary extend the abstract GameObject class, and the GameObject



class has protected variables so that these attributes can be used by these classes that extend it. The Tank and Bullet extend the Moveable class, but the tank and bullet class also have attributes that are unique to the class. The Wall and PowerUp class extends the Stationary class. The BreakWall and UnBreakWall classes extend the Wall class; these two classes each have a GameObject type assigned to them which comes into play later. The PowerUp class also has a GameObject type assigned to it, and within the class is creates Enums to differentiate the different kids of power-ups. Enums are also used in the GameObject class to create a constructor that each child class uses to differentiate themselves as a specific type of GameObjects.

The Moveable class contains variables 'vx' and 'vy' which Is the speed applied to Moveable objects, such as the tank and bullets; as well as other position variables. Moveable class also contains rotations, methods for forward/backward movement (moveForwards() & moveBackwards()), collision scenarios that allow the tank and bullets to have motion. The Bullet class only uses moveForwards() since the bullets do not move backwards. The tank, however, is capable of moving backwards, forwards, rotating left & right. These movement classes in the Moveable class are later called in the Tank class to dictate what toggling certain keys does. The Tank Control class is in charge of listening to key events (using KeyListeners) so that the tank can move in specific directions according to the key pressed or unpressed. The Moveable class contains the collideWith() method which is crucial for differentiating the different types of collisions and their outcomes. The collideWith(GameObject other) is actually called in the Collision class, by the check() method, which takes in game object 1 (Movable g1) and game object 2 (GameObject g2) and uses an if statement to detect whether an intersection was detected. We are able to use this Rectangle (Rectangle is a java library) intersect method because our game objects all have Rectangle hit-boxes around them. Back to the Moveable class–with this collideWith(GameObject other) method uses a switch case to determine the different outcomes depending on which type of game object the parameter 'GameObject other' is, as well as determining whether 'Movable g1' is a Tank or a Bullet. Once it knows the object that is performing the collision to 'GameObject other,' then the outcome is different for each case. When bullets interact with breakable and unbreakable walls, or tanks, the bullet game object in the switch case calls the setShow() method and passes the false parameter, which allows for the bullet to not be visible. When the bullet interacts with a breakable wall, however, the bullet and the wall both become invisible. The Show boolean in used in the GameObject class in isShown() and setShow() methods. The actual drawing (or not drawing) of game objects is done in the GameObject class, inside the drawImage(Graphics g) method, depending on whether the boolean show = true. Back to the Moveable class– the encounters that Tanks have, is with walls, power-ups, and other tanks. Upon colliding with another tank or any wall, the setToPreviousPosition() method is called so that the tank does not run over the game object and sort of takes a step back. When tanks collide with power-ups the power-ups disappear.

The Bullet class does not contain much since it extends the Moveable class, but it does contains its *own* moveForwards() class and Speed variable because the speed must differ from the speed of the tank so that the bullet does not get stuck as the tank is moving forward at the same speed. It also contains a constant DAMAGE variable which defines how much damage is caused to a player when it is shot. The logic for a player being shot is in the Tank class, using the wasShot() method. This is then used inside our PlayerStats class that is in charge of tracking a players health, lifeCount, and drawing/displaying the stats for each player as they play the game. Lastly, it contains an update() method so that the bullets can continue to move forward when being shot.

The Tank class contains movement logic for when players press buttons which are accessed by the Tank Control class; a bullet ArrayList; a washShot() method that is called in Moveable when a collision between a bullet and a tank occur. The washShot(Bullet.*DAMAGE*) method then calls a method named damage() in the PlayerStats class. The PlayerStats class contains addLife(), addHealth(), and damage(). Damage() is used to reduce a tanks healthCount by increments of 10 per bullet shot at it, and to decrement the lifeCount by 1 per 100 health counts. So 1 life = 100 health counts, and 1 bullet = -10 health counts; the maximum lives a player can have is 5. Playerstats has a damage() method which returns a boolean determining if game is over, damage() takes in an 'int damageDone' which is equal to the amount of damage done aka how much value is reduced form healthCount or Life count. The boolean method damage() is called in the Tank class within another method called wasShot() which is also used in the collideWith method in Moveable. So if a bullet collides with a tank, that other tank is shot and this leads to the damage() method being called inside getShot(). The boolean method damage() contains the logic for the game to be over basically or for a player to win since the other player has no lives left. So if damage() returns TRUE, then the variable isDead = true (located in the Tank class). This then leads to TRE understanding TRE is true and calling game over. TRE does this by calling either EndGamePanel1 or EndGamePanel1, depending on which player isDead(). PlayerStats class is in charge of displaying, drawing, and calculating a player stats as the game progresses. The stats are drawn on the bottom right and left portions of the game panel. The washShot(Bullet.*DAMAGE*) takes in a parameter which is equal to the numerical damage done to the bullet that was shot, which is calculated in the stats class.

The applyPowerUp() method called in Moveable class when a collision between a tank and power-up occur, which also contains a switch case to differentiate the effects each power-up has on the tank; there is a switch case that determines which power-up is detected by using enums created in the PowerUp class. The Mushroom power up serves to increase the lifeCount by 1 (the Mushroom becomes ineffective to the tank once the player has reached 5 lives), and in this case the addLife() method is called in the PlayerStats class. The Star power-up serves to make the tank bulletproof for a set amount of time; the time is controlled by the bulletProofTime variable that decrements as game time passes by, and the Tank classes also draws a blue circle around the tank to demonstrate it is bulletproof for the time being. The Coin power-up increases the tanks health by 10 health counts (the Coin becomes ineffective to the tank once the player has reached 5 lives while having 100 health counts) by calling the addHealth() method in PlayerStats which calculates the health each time a tank is short or gets a power-up. Like the Bullet class, the Tank class has an update() method which updates the position and values of the tank when it moves, updates bullets being created that will moved forward. This class also has a drawImage(Graphics g) method that draws bullets as they are being updated and passes them to the Bullet class.

The TRE class uses the run() method from the Runnable Interface which is “designed to provide a common protocol for objects that wish to execute code while they are active” (Arthur van Hoff). In this method we update each tank and check for collisions. We check for collisions here because in TRE is where we initialize the tanks objects and the stationary objects ArrayList. The collision checking involves looping through the stationary objects and bullets to be able to accurately check if a Tank or Bullet has collided with them, using the check() method located in the Collision Class. TRE class checks all Tank collisions with Stationary Objects, all Bullet collisions with Stationary Objects, Bullet collision with Tanks, and tank-on-tank collision. TRE is also responsible for using the paintComponent to draw walls, power-ups, tanks, the split screen, the mini-map, and the background.

The way the Split Screen is drawn is using the BufferedImage subclass which draws the leftHalf and rightHalf of the screen. Its purpose is to be able to follow the tanks around wherever they go on the screen for each player and split the screen for each player. I did this by making the x-coordinate parameter be tailored for each scenario of the tanks portion on the game screen. The 1st scenario is when the tanks x coordinate is less than 1/4 of the GAME_SCREEN_WIDTH, then the split screen perspective begins at the top left corner of the screen = origin (0,0); The 2nd scenario is when the tanks x coordinate is greater than 3/4 of the GAME_SCREEN_WIDTH/4, then our perspectives of the screen begins at the middle of the screen = GameConstants.GAME_SCREEN_WIDTH /2; The 3rd scenario is when the tanks x coordinate is not less than 1/4 of the GAME_SCREEN_WIDTH or greater than 3/4 of the GAME_SCREEN_WIDTH/4, then the perspective is the value of the tanks x value minus GAME_SCREEN_WIDTH/4. The split screen only accounts for the x coordinate, the y coordinate remains 0. This is because my game screen is wide so it benefits the player to see entire length (y) of the screen while playing. Lastly, the TRE class holds the if statements that determine when the game is Over, by calling the isDead() method in the Tank class we discover which player has no lives left, and this if statement in the TRE class call the corresponding EndPanel by using the setFrame() method created in the Launcher class.

The Mini-Map was drawn using a text file, which was read in the TRE class with an InputStreamReader. I designed the game by assigning each kind power-up, and each kind of wall a specific number in the switch case and strategically designed the map with those numbers so the InputStreamReader could read it accurately.

The TRE class lastly draws the ArrayList of Stationary Objects, the mini-map, split-screen, the tanks, and the background, all using javas Graphics2D class. The GameConstants class is where we determine the screen measurements which remain the same and are used in the TRE class to draw with accurate measurements.

The process I took involved starting small, by only creating one tank first and making sure it was able to properly shoot, rotate, and move, I gradually started adding other game objects, like walls and bullets. Before even getting into collisions, I had to make sure the tank could shoot and move, so at this point the tank was not abiding by collisions and was running over walls. Once the collision class came along, I was able to create different scenarios for each type of collision, and learned how the TRE class would aid in these collisions. At this point, the abstractions were defined, it was a matter of putting variables and methods in their rightful place. Once the collisions began working, the game started feeling more real and this lead to the creation of PlayerStats which is what would officialize the game. The last thing done was edit the map layout so the game could feel challenging and authentic and change the EndGamePanel class, as well as the StartPanel class so it was Super Mario Bros. themed!

8. Project Reflection

This project truly helped me get comfortable with Java, which I was not in the past. I now understand better syntax, how to not hard-code as much, how to use abstractions, how to make my code more readable and updatable for future use, as well as how to use Java Swing. I knew this assignment was going to be extremely difficult for me but I realized that not understanding something is the first step to understanding. I appreciate the started code given, with out it I would be kind of lost, to be honest. The videos also definitely helped! The other aircraft game given by the professor was also helpful, especially in understanding collisions. The design of my game was inspired because I love Super Mario video games. Watching demos from previous year the professor posted also helped me create the map layout. I think the hardest part for me is not knowing if what I'm coding is technically efficient. I feel like I do not get much exposure to good code, but now I understand more principles to abide by while coding due to the weekly quizzes and practice. This project is a great way for a student to challenge themselves and improve their Java abilities. I am curious to know how other students approaches the project, if they had a similar UML diagram to me, and if there are ways to improve the way I approach an assignment. This project also taught me when to ask for help before it's too late to complete an assignment, and finding better coding resources. The time management that went into this project paid off because I would consistently work on it every other day up until the day it was due.

9. Project Conclusion/Results

The project was a success, the game works as expected and could be played by any two players! Something I could have improved is simplifying the TRE class with another class. There are definitely ways to improve the project but for now it runs smoothly. In the future I would like to add a power-up that makes a players bullets more powerful.