Object-Oriented Programming

Final Project Report

Name of Lecturer: Jude Joseph Lamug Martinez, MCS

Made by:

Orlando Jonathan Padiman (2702337615)

Class:

L2BC

BINUS UNIVERSITY INTERNATIONAL

JAKARTA

2024

TABLE OF CONTENTS

Chapter 1

PROJECT SPECIFICATIONS

1.1. Project Description

For my final project, I attempted to recreate the popular game “Buckshot Roulette” by Indie Game developer Mike Klubnika. This game puts a dangerous twist on the already dangerous russian roulette. Instead of a regular ‘one-in-the-chamber’ revolver, the game involves a pump-action shotgun, loaded with a combination of a random amount of live and dud bullets. In this dance of death, both the player and dealer take turns in taking shots at each other or even themselves.

1.2. Project Link

The GitHub Repository of the project can be accessed through the link provided below:

GitHub Repository

1.3. Features

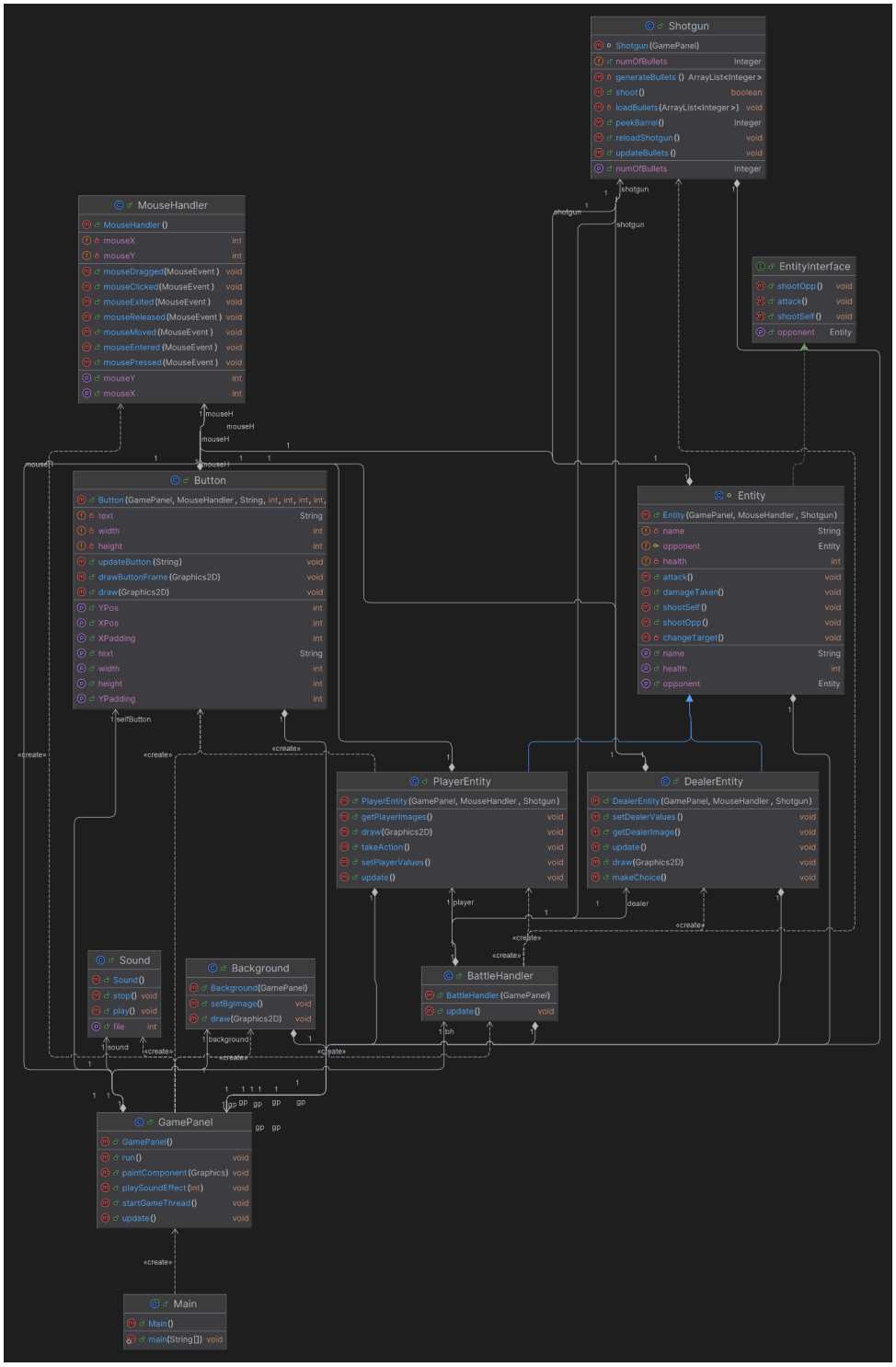
1. Enemy AI:
    - AI thinking algorithm which does the following:
        - Will act accordingly if they know what the current shell is. Usually on the last shell in the gun. For example, if the last shell is a live, then the AI will choose to aim at the player before shooting.
        - Otherwise, AI will flip a coin to randomly decide who to shoot

2. Smart Shotgun Algorithm
    - The shotgun used in the game’s situation is shared between the player and dealer, meaning that they share the same bullet configuration, number of bullets and current bullet type. The shotgun also automatically reloads every time it runs out of bullets to indicate the start of a new round.

Chapter 2

SOLUTION DESIGN

2.1. Class Diagram



2.2. Relationships

1. The classes PlayerEntity and DealerEntity are both subclasses of the Entity abstract class, which implements from the EntityInterface Interface.
2. The GamePanel Class is the driver for the entire game and is connected to other major aspects of the game like an instance of the MouseHandler, BattleHandler and Background class, as well as 2 instances of the Button class.
3. The BattleHandler holds an instance of the PlayerEntity class and DealerEntity class and handles the events that occur during the battle.

2.3. Classes

Class Name	Extends/Inherits	Dependencies	Function
Shotgun	None	java.util.ArrayList java.util.Collections java.util.Random java.util.Stack	Represents a shotgun object, which will be the main weapon used during the game's runtime.
EntityInterface	None	None	Groups the methods to be used by an Entity object.
Entity (Abstract)	Implements EntityInterface	GamePanel MouseHandler Shotgun	Used as a template for Entity objects, but require unique sets of methods from each other.
PlayerEntity	Extends Entity	javax.imageio.ImageIO java.awt.* java.awt.image.BufferedImage java.io.IOException	Represents a Player's character. Provides access to methods for the player to interact with the shotgun and use it against the dealer.
DealerEntity	Extends Entity	javax.imageio.ImageIO java.awt.* java.awt.image.BufferedImage java.io.IOException	Represents a Dealer character. Contains dedicated methods for the Dealer AI to function.

		java.util.Random	
MouseHandler	Implements MouseListener, MouseMotionListener	javax.swing java.awt.event.MouseEvent java.awt.event.MouseListener java.awt.event.MouseMotionListener	Acts as a handler for mouse events. Tracks for mouse clicks and updates related variables related to certain mouse clicks.
Button	None	java.awt.*	Represents a button object to be drawn over the screen. Constructor contains properties for the Button like position, text, etc.
Background	None	javax.imageio.ImageIO java.awt.* java.awt.image.BufferedImage java.io.IOException	Represents the background image for the entire game.
Sound	None	javax.sound.sampled.Clip javax.sound.sampled.AudioSystem javax.sound.sampled.AudioInputStream java.net.URL	Acts as the control center for the game's sound effects.
GamePanel	Extends JPanel & Implements Runnable	javax.swing.* java.awt.*	Acts as the game's driver. Manages all the game's assets based on each object's status and draws them on the panel, also keeping them updated.
Main	None	javax.swing.*	The file running the game.

2.4. Implementation

1. Java Swing:

   The foundation for the GUI of the program. Without the implementation of Java Swing, the game would only remain a terminal-based game with no means of displaying what goes on in the game.

2. Graphics2D Class:

   The implementation of the Graphics2D Class allows for the 2D Assets like the dealer sprites, shotgun sprites, as well as the button and other UI aspects to be drawn on the game's window.

3. Audio Input Stream, Audio System, Clip Class:

   Although only playing a minor role, the implementation of these classes allow for sound effects to be played as while the game runs in order to provide a more immersive experience.

4. Random Class:

   The Random class plays a major role in the game as it plays a part of the shotgun's random number of bullets, types of bullets, as well as the in the AI's decision making algorithm.

2.5. Important Code Snippets

1. Dealer AI

```
if (canMove) {
    /*
    If the dealer can move, it will go through
    the AI thinking algorithm which is described as the following:
    1. Will use as many items as they can
    2. Will act accordingly if they know what the current shell is.
    3. Otherwise, will flip a coin to randomly decide who to shoot.
     */
    if (gun.getNumOfBullets() == 1 || knowsNextShot) { // When there's only one shell left or knows next shot
        if (gun.peekBarrel() == 0) { // If the last shell is a dud,
            if (aimingAtOpponent) { // If aiming at opponent with the last bullet,
                changeTarget(); // Change to aim at self
            }
            attack(); // If aim is already on self, then shoot
        } else { // If dealer knows last shell is live,
            if (!aimingAtOpponent) {
                changeTarget();
            }
            attack();
        }
    } else { // If dealer doesn't know what the next shot is, dealer will flip a coin to decide
        System.out.println("Dealer is flipping a coin");
        int decision = random.nextInt( bound: 2); // Coin flip to make decision to either shoot self or player.
        if (decision == 0) {
            aimingAtOpponent = true; // Aim at player
        }
        if (decision == 1) {
            aimingAtOpponent = false; // Aim at self
        }
        attack(); // Shoots after coin flip;
      }
}
```

2. Shotgun Shell Generation

```
private ArrayList<Integer> generateBullets() {
    /*
    Generate a random set of live and dud bullets.
    Randomly generated bullets will be shown to the players.
     */
    ArrayList<Integer> bulletsToLoad = new ArrayList<>();

    // Insurance so that at least 1 live and 1 dud bullet.
    bulletsToLoad.add(0);
    bulletsToLoad.add(1);

    int numExtraBullets = random.nextInt( bound: 6); // How many bullets are generated in a single reload. (2-8)
    for (int i = 0 ; i < numExtraBullets ; i++) {
        // Randomly generates 0 or 1 and adds it to the list of bullets,
        bulletsToLoad.add(random.nextInt( bound: 2));
    }
    // Shows the order to the player.
    Collections.sort(bulletsToLoad);
    System.out.println(bulletsToLoad + "\n");
    return bulletsToLoad;
}
```

3. Shotgun Shooting Handling

```
public void updateBullets() {
    /*
    After shooting, removes the bullet from the top of stack
    and subtracts the number of the bullets ian the gun by one.
     */

    loadedBullets.pop();
    numOfBullets--;
    if (getNumOfBullets() == 0){
        isLoaded = false;
    }

}

2 usages
public boolean shoot() {
    if (peekBarrel() == 1) {
        updateBullets();
        gp.playSoundEffect( i: 1);
        return true; // Shoots live
    } else {
        updateBullets();
        gp.playSoundEffect( i: 0);
        return false; // Shoots blank
    }
}
```

4. Entity Attack Handling

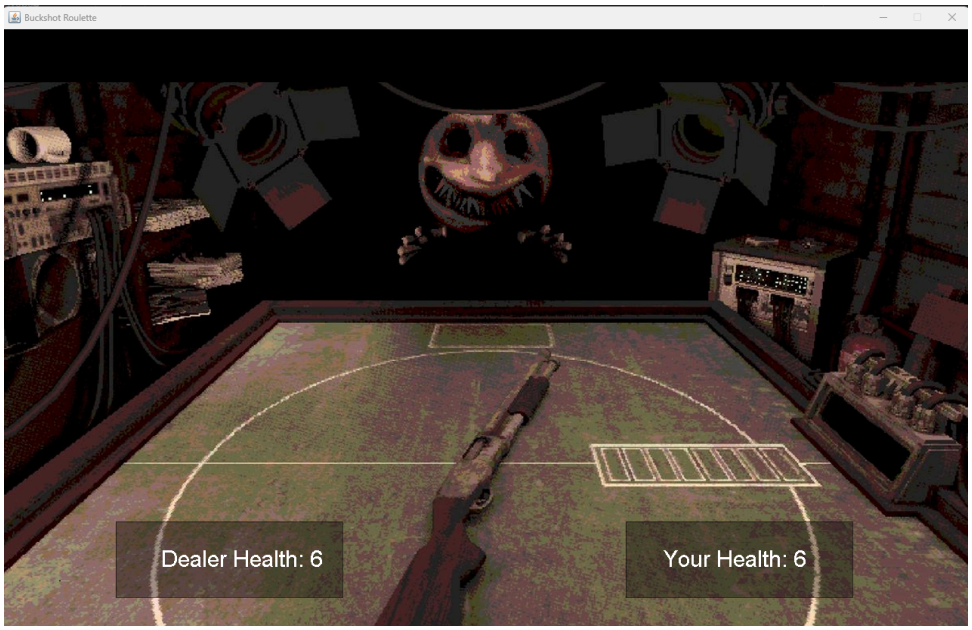
```java
@Override
public void attack() {
    if (aimOnOpp) {
        if (shotgun.shoot()) {
            opponent.damageTaken();
            System.out.println(opponent.getName() + " was shot");
            System.out.println();
            inAction = false;
        }
        else {
            System.out.println();
            inAction = false;
        }
    } else {
        if (shotgun.shoot()) {
            damageTaken();
            System.out.println("shot self with live");
            System.out.println();
            inAction = false;
        }
        else {
            System.out.println("dud bullet on self");
            System.out.println();
            shotSelfWithDud = true;
        }
        System.out.println();
    }
    if (health <= 0) {
        isDead = true;
    }
    if (!isDead && !opponent.isDead && shotgun.loadedBullets.isEmpty()){
        shotgun.reloadShotgun();
        System.out.println("reloading from participant class");
    }
}
```

Chapter 3

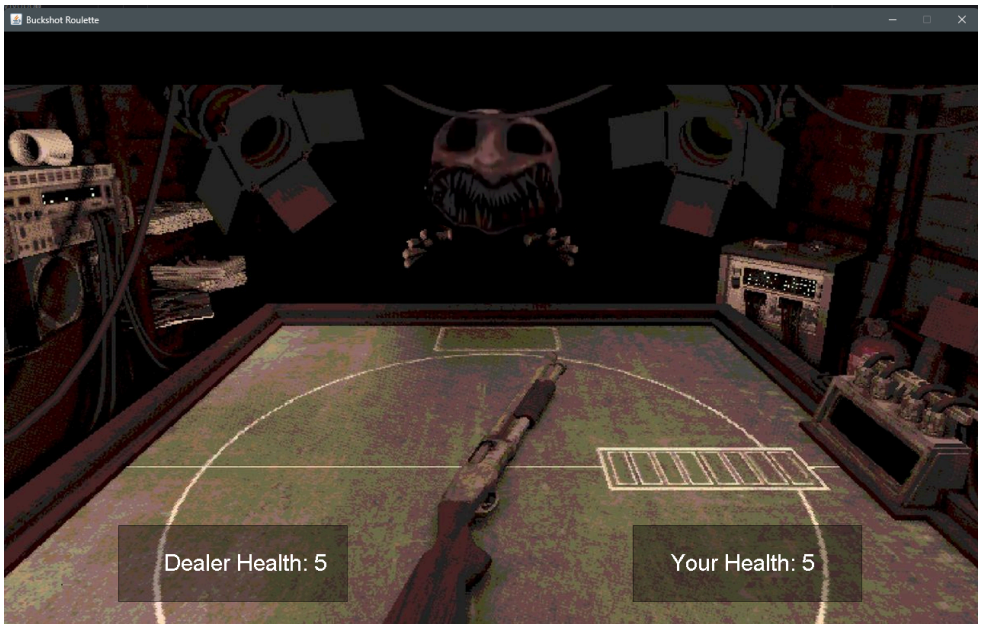
DOCUMENTATION

3.1. Screenshots

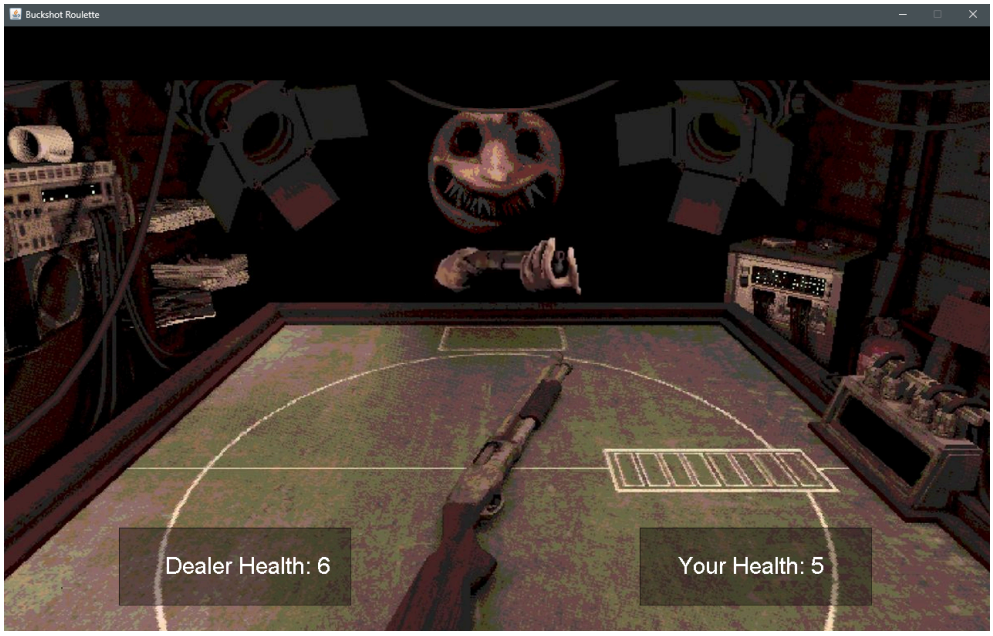
Game Start State:



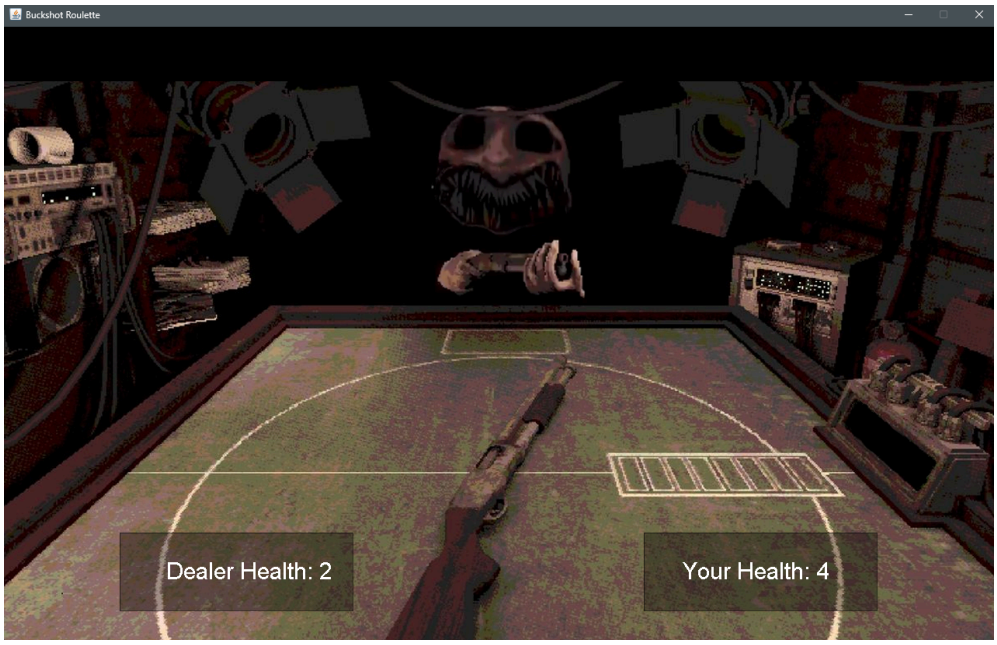
Dealer Taking Damage for the First Time



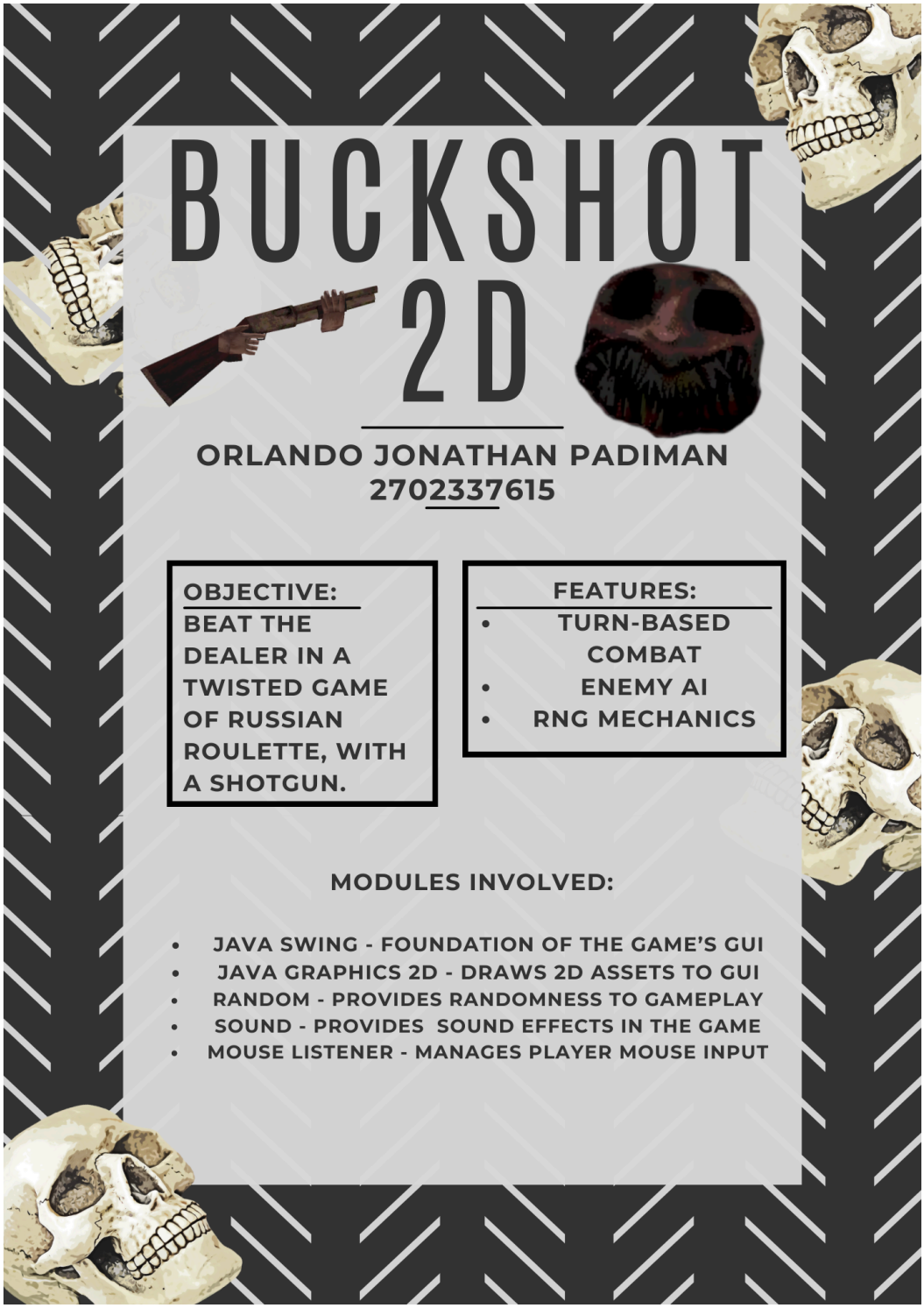
Dealer Holding the Gun on Their Turn



Angry Dealer Holding Gun on Their Turn



3.2. Poster


BUCKSHOT
2D
ORLANDO JONATHAN PADIMAN
2702337615
OBJECTIVE:
BEAT THE DEALER IN A TWISTED GAME OF RUSSIAN ROULETTE, WITH A SHOTGUN.
FEATURES:
• TURN-BASED COMBAT
• ENEMY AI
• RNG MECHANICS
MODULES INVOLVED:
• JAVA SWING - FOUNDATION OF THE GAME'S GUI
• JAVA GRAPHICS 2D - DRAWS 2D ASSETS TO GUI
• RANDOM - PROVIDES RANDOMNESS TO GAMEPLAY
• SOUND - PROVIDES SOUND EFFECTS IN THE GAME
• MOUSE LISTENER - MANAGES PLAYER MOUSE INPUT

Chapter 4

EVALUATION AND REFLECTION

4.1. Lessons Learnt

From this project, I learnt how to use Java to an advanced degree, or at least to an introductory level for game programming. This was also my first time working with Java's Swing toolkit and with Java GUI as a whole, so there was a lot to take in during the learning process. I also learnt that it is best to implement both the backend and the GUI at the same time instead of completing one first before implementing the other. This is because I spent quite a decent amount of time working on the backend for the game's code thinking that connecting it to the UI will be smooth sailing. However, two days before the deadline, I figured out how difficult it actually was to adapt a UI to an already existing backend.

I have also been humbled and taught about the importance of time management as, if I had started earlier, I would've had more time to actually work on so many features as well as fully implementing a working GUI. I also learnt this from the countless amount of times I had to restart my project files over and over again.

4.2. Future Improvements

There were still so many features that I could've implemented for this game. However, I didn't understand how to implement some of them. From these regrets, the first thing I have to improve over everything else is time management. If I had managed my time better, I would have had more time to add in and implement more features into the game.

Another field that needed more improvement is my understanding of the GUI. I could have improved the GUI to implement a more interactable GUI. However, my understanding of Java GUI design is still very limited while working on this project.

The same could be said for animations. There could have been animations added into the game to improve the overall aesthetics of the game and make it look like a functional game.