VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

THE INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

OBJECT-ORIENTED PROGRAMMING (IT069IU)

ANIMAL KILLING PROJECT

REPORT PROJECT FINAL

Semester 1, 2024-2025

Submitting date: 22/12/2024

No	Full name	Student ID
1	Nguyễn Hoàng Bảo Trân	ITITSB22027
2	Lê Nguyễn Thanh Trúc	ITITWE22168

Instructor: Dr. Tran Thanh Tung

Project's GitHub: Link

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1. The current state of gaming

The gaming industry is flourishing, characterized by a wide range of platforms, cutting-edge technology, and significant cultural impact. It spans from highly immersive virtual reality experiences and lifelike graphics to straightforward mobile games.

We are developing a game as part of our object-oriented programming course, a four-credit class. This project provides us with a valuable opportunity to enhance our Java programming skills while applying much of the theory we learn in class through practical practice.

1.2. About the game project

The Animal Killing project is inspired by the popular chicken shooting games found on platforms like Google Play and the App Store. Built using object-oriented programming principles, the game offers a fresh take on the classic gameplay, incorporating modern mechanics and enhancements to deliver an engaging and immersive experience for players.

1.3. Highlights of Animal killing game:

As mentioned, we will add more features to our animal killing game to make users more excited when playing this game. Players can access a wide array of weaponry, including laser beams, rockets, and electric pulse explosives. Each weapon can be enhanced progressively, matching the increasing challenges of each stage.

The game features straightforward yet visually appealing graphics, highlighted by charming character designs. Notably, the birds come in a variety of forms and outfits, adding a playful and creative touch.

The game features dynamic sound effects that enhance the excitement of firing weapons and defeating enemies. Its upbeat background music further amplifies the thrill, making the gameplay experience even more immersive and energetic.

1.4. References

1. mostafaHegab. (n.d.). mostafaHegab/Monster-Inc: Simple Monster Inc Game with JAVA. Retrieved from https://github.com/mostafaHegab/Chicken-Invaders
2. UML class: Lucidchart. (n.d.). Retrieved from https://lucid.app/lucidchart/c6e383ae-e074-4fc5-ae3765484a9a7a34/edit?invitationId=inv_0a814f23-793c-4559-a23f30c2dc288661&page=HWEp-vi-RSFO#
3. Best 55+ 8. (n.d.). Retrieved from https://wallpapercave.com/w/wp7872568
4. Programming Space Invaders in Java (fx) Tutorial 1/2. (2019). Retrieved from https://www.youtube.com/watch?v=0szmaHH1hno&list=WL&index=4&t=18s
5. Programming Space Invaders in Java (fx) Tutorial 2/2. (2019). Retrieved from https://www.youtube.com/watch?v=dzcQgv9hqXI
6. (N.d.). Retrieved from https://openjfx.io/#
7. Super Pixel Vintage Gaming Presentation. (n.d.). Retrieved from https://slidesgo.com/theme/super-pixel-vintage-gaming?fbclid=IwAR20XFCnI-9j9m-fXtvFrVnvp8Cf6NYDbKq0FbsinzJ7hjRIAdrVbHhHqkQ

1.5. Developer team

We have two members who come from International University, Information Technology major:

Table 1.5: Individual responsibility and contribution

Name - Github username	ID	Responsibility	Contribute
Nguyễn Hoàng Bảo Trân - *Btran2404*	ITITSB22027	Develop first version, build entities, create UML, develop first version, design UI, create UML, prepare report	50%
Lê Nguyễn Thanh Trúc - *Selena166*	ITITWE22168	Set up controllers, set up SFX, tester, GitHub repository host, design UI, design game mechanic, prepare presentation slide	50%

CHAPTER 2: SOFTWARE REQUIREMENTS

2.1. What we have:

- A system designed with user-friendliness at its core, ensuring smooth navigation and an intuitive experience for every user.
- Simplicity in operation, allowing even those with minimal technical knowledge to use it effortlessly without complications.
- Minimal maintenance requirements, reducing both time and costs while ensuring long-term reliability and efficiency.

2.2. What we want:

- Delivering maximum high resolution for crystal-clear visuals that elevate the user experience to the next level.
- Crafting the entire system with efficiency in mind, ensuring seamless performance and optimized functionality.
- Enhancing features and graphics through thoughtful upgrades, offering a modern and captivating interface.
- Ensuring effortless updates, making it simple to stay ahead with the latest improvements and innovations.

2.3. Project objectives

This project aims to strengthen your understanding and practical application of Object-Oriented Programming (OOP) principles by developing an engaging game. By the project's conclusion, you will have honed your skills in several critical areas, including encapsulation, inheritance, polymorphism, and abstraction. Here's a detailed breakdown of the objectives:

Mastering Key OOP Concepts:

- **Encapsulation:** You will design classes with clear boundaries between data and methods, ensuring that each class's responsibilities are well-defined. By implementing access controls, such as private and protected modifiers, you will protect data integrity and reinforce data hiding. This approach highlights the importance of abstraction, enabling you to focus on relevant details while shielding users from complex underlying implementations.

- **Inheritance:** Learn to create class hierarchies that effectively represent relationships between entities in your game world. By reusing code through inheritance, you'll enhance maintainability and reduce redundancy. The concepts of superclasses and subclasses will be explored, along with method overriding, to enable more flexible and dynamic behavior.
- **Polymorphism:** You'll design methods capable of working with various object types at runtime, fostering flexibility and adaptability in your game mechanics. By implementing virtual methods and method overriding, you will enable dynamic behavior that allows your program to handle diverse situations seamlessly. This principle ensures your game code is extensible and responsive to new requirements.
- **Abstraction:** Develop your ability to define interfaces and decouple game components, simplifying your program's structure and enhancing modularity. You'll focus on essential features while concealing implementation complexities, leading to reusable and scalable code.

Building Design and Development Expertise:

- **Object-Oriented Design:** Learn to apply OOP principles to create modular and well-organized game architecture. You'll design classes that accurately model real-world game entities and define meaningful relationships between them, ensuring a cohesive and interactive game world.
- **Game Development Techniques:** This project will involve implementing core game mechanics using OOP concepts, enabling you to integrate user input and manage events within the game loop. Additionally, you'll design an engaging and visually appealing interface to enhance the player experience.
- **Software Testing and Refactoring:** Testing will be a crucial part of this project, helping you identify and resolve bugs effectively. You'll also practice refactoring techniques to make your code cleaner and more maintainable. By following best practices, you'll improve the readability and efficiency of your game's codebase.

2.4. Working tools, platform:

Table 2.4: tools

Software	Purpose
JDK 21	for running Java programs
JavaFx	main Java library for creating the game
SceneBuilder	make designing Fxml files easier
Aseprite	create in game entity images
Apache Maven	Java project builder
IntelliJ	IDE to run game code
Canva	Design entity
GitHub	Upload project's code and manage contributions from different individuals

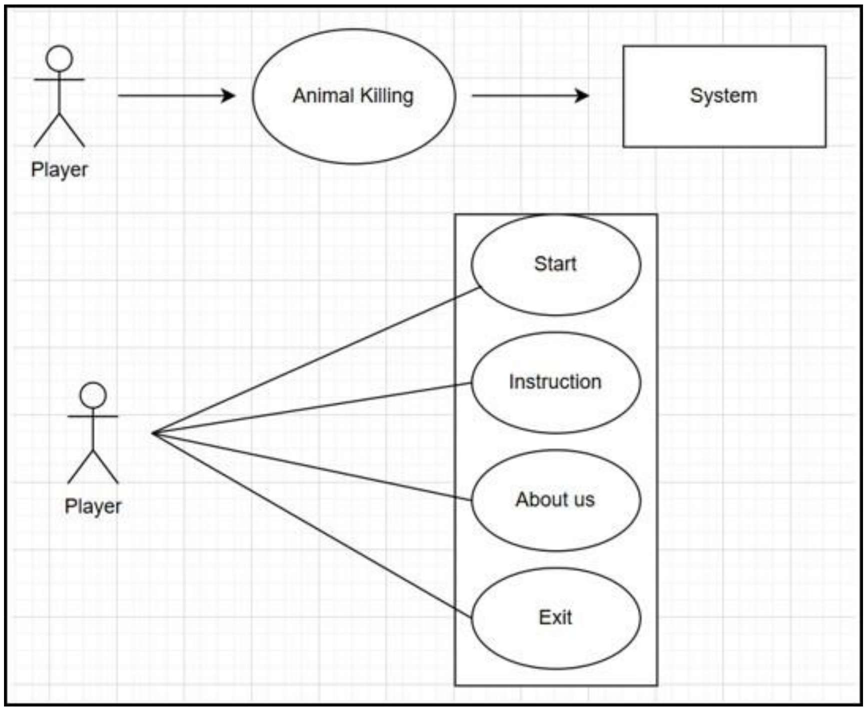
2.5. Use Case Scenario

Table 2.5: Using Case Scenario

Animal Killing	**START**	Start the game
	INSTRUCTION	Show how to play the game
	ABOUT US	Information about the game creators
	GAME OVER	Show your score and ask if you want to continue playing
	EXIT	Exit the game

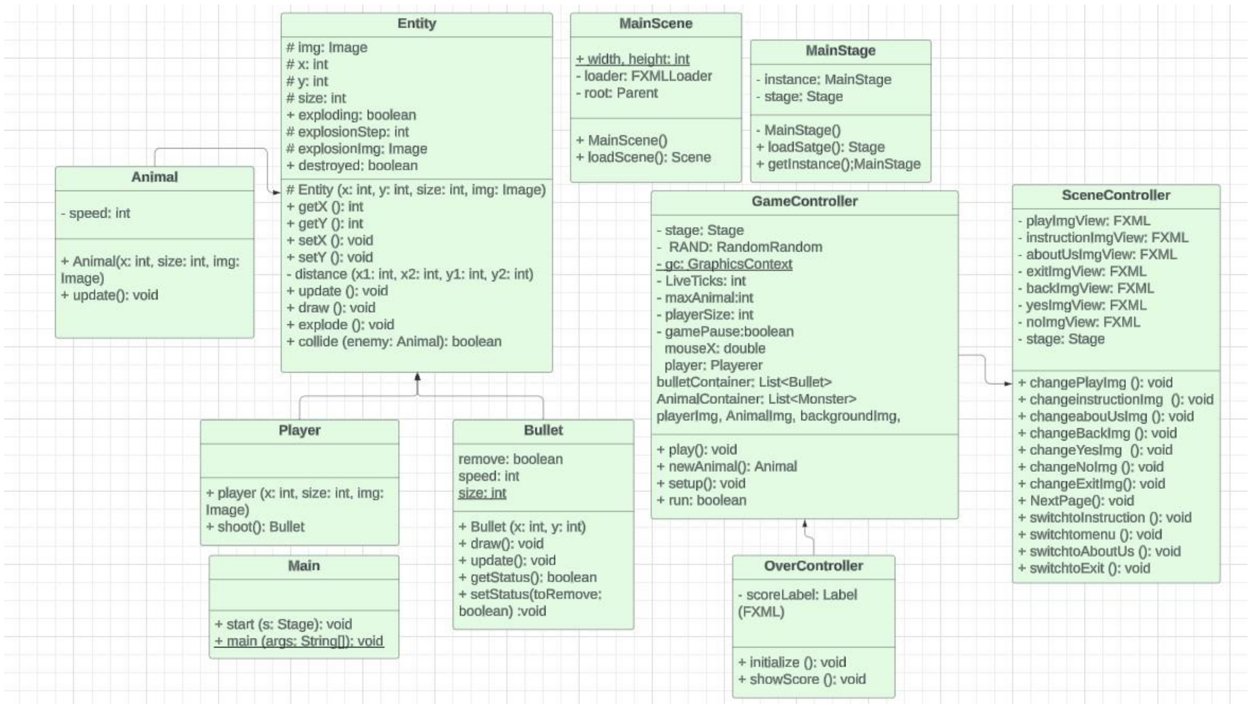
2.6. Use Case diagram

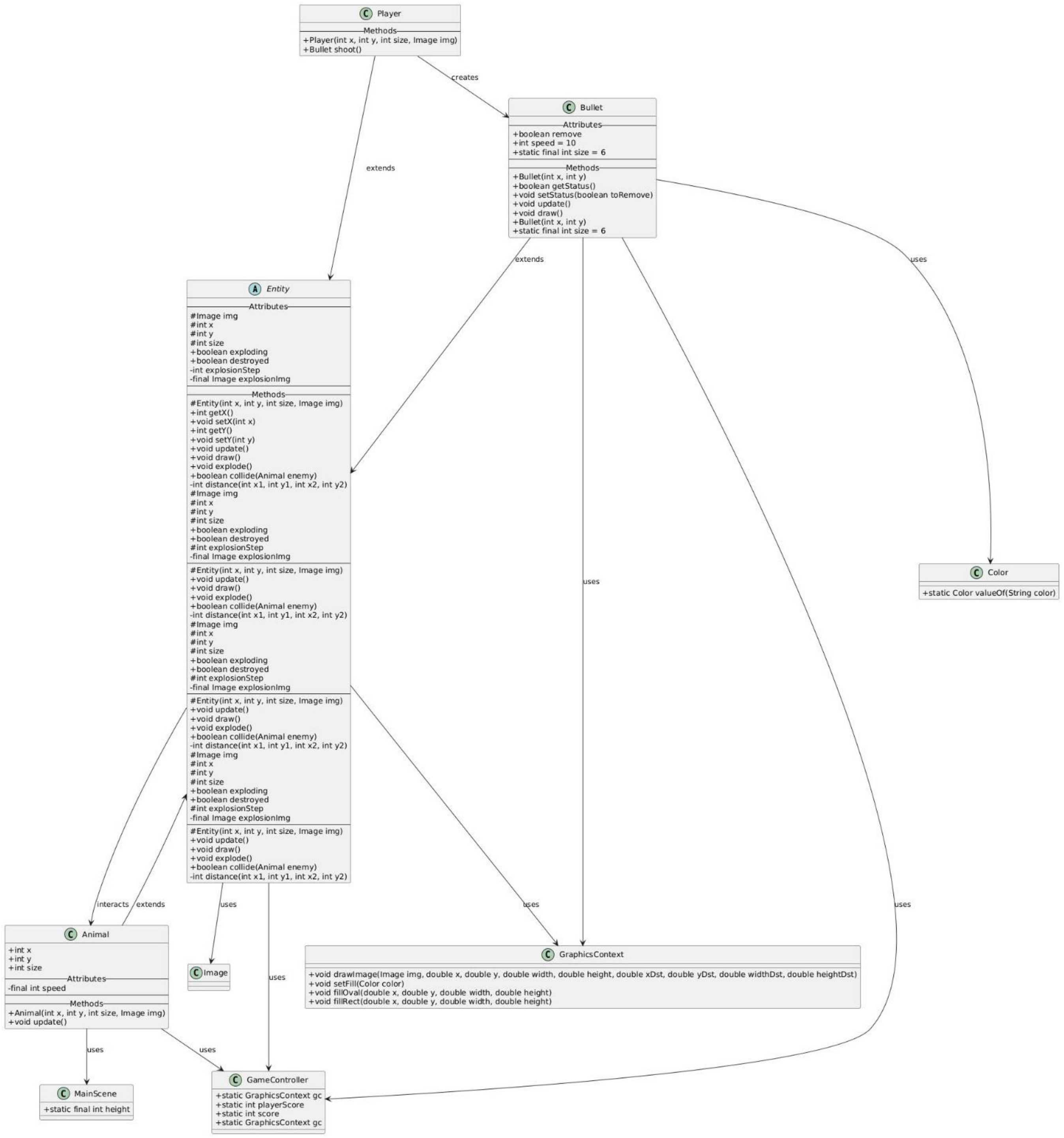
Figure 2.6: Case díagram



2.7. Class diagram

Figure 2.7: UML diagram




Player
Methods
+Player(int x, int y, int size, Image img)
+Bullet shoot()
creates
extends
Bullet
Attributes
+boolean remove
+int speed = 10
+static final int size = 6
Methods
+Bullet(int x, int y)
+boolean getStatus()
+void setStatus(boolean toRemove)
+void update()
+void draw()
+Bullet(int x, int y)
+static final int size = 6
extends
uses
Entity
Attributes
#Image img
#int x
#int y
#int size
+boolean exploding
+boolean destroyed
-int explosionStep
-final Image explosionImg
Methods
#Entity(int x, int y, int size, Image img)
+int getX()
+void setX(int x)
+int getY()
+void setY(int y)
+void update()
+void draw()
+void explode()
+boolean collide(Animal enemy)
-int distance(int x1, int y1, int x2, int y2)
#Image img
#int x
#int y
#int size
+boolean exploding
+boolean destroyed
#int explosionStep
-final Image explosionImg
#Entity(int x, int y, int size, Image img)
+void update()
+void draw()
+void explode()
+boolean collide(Animal enemy)
-int distance(int x1, int y1, int x2, int y2)
#Image img
#int x
#int y
#int size
+boolean exploding
+boolean destroyed
#int explosionStep
-final Image explosionImg
#Entity(int x, int y, int size, Image img)
+void update()
+void draw()
+void explode()
+boolean collide(Animal enemy)
-int distance(int x1, int y1, int x2, int y2)
#Image img
#int x
#int y
#int size
+boolean exploding
+boolean destroyed
#int explosionStep
-final Image explosionImg
#Entity(int x, int y, int size, Image img)
+void update()
+void draw()
+void explode()
+boolean collide(Animal enemy)
-int distance(int x1, int y1, int x2, int y2)
Color
+static Color valueOf(String color)
uses
interacts
extends
uses
uses
uses
Animal
+int x
+int y
+int size
Attributes
-final int speed
Methods
+Animal(int x, int y, int size, Image img)
+void update()
Image
uses
GraphicsContext
+void drawImage(Image img, double x, double y, double width, double height, double xDst, double yDst, double widthDst, double heightDst)
+void setFill(Color color)
+void fillOval(double x, double y, double width, double height)
+void fillRect(double x, double y, double width, double height)
uses
uses
MainScene
+static final int height
GameController
+static GraphicsContext gc
+static int playerScore
+static int score
+static GraphicsContext gc

The **Entity** class serves as the foundational superclass for all game objects, encompassing shared attributes and functionalities. These include essential properties such as position, velocity, and collision detection, as well as the ability to spawn enemy monsters.

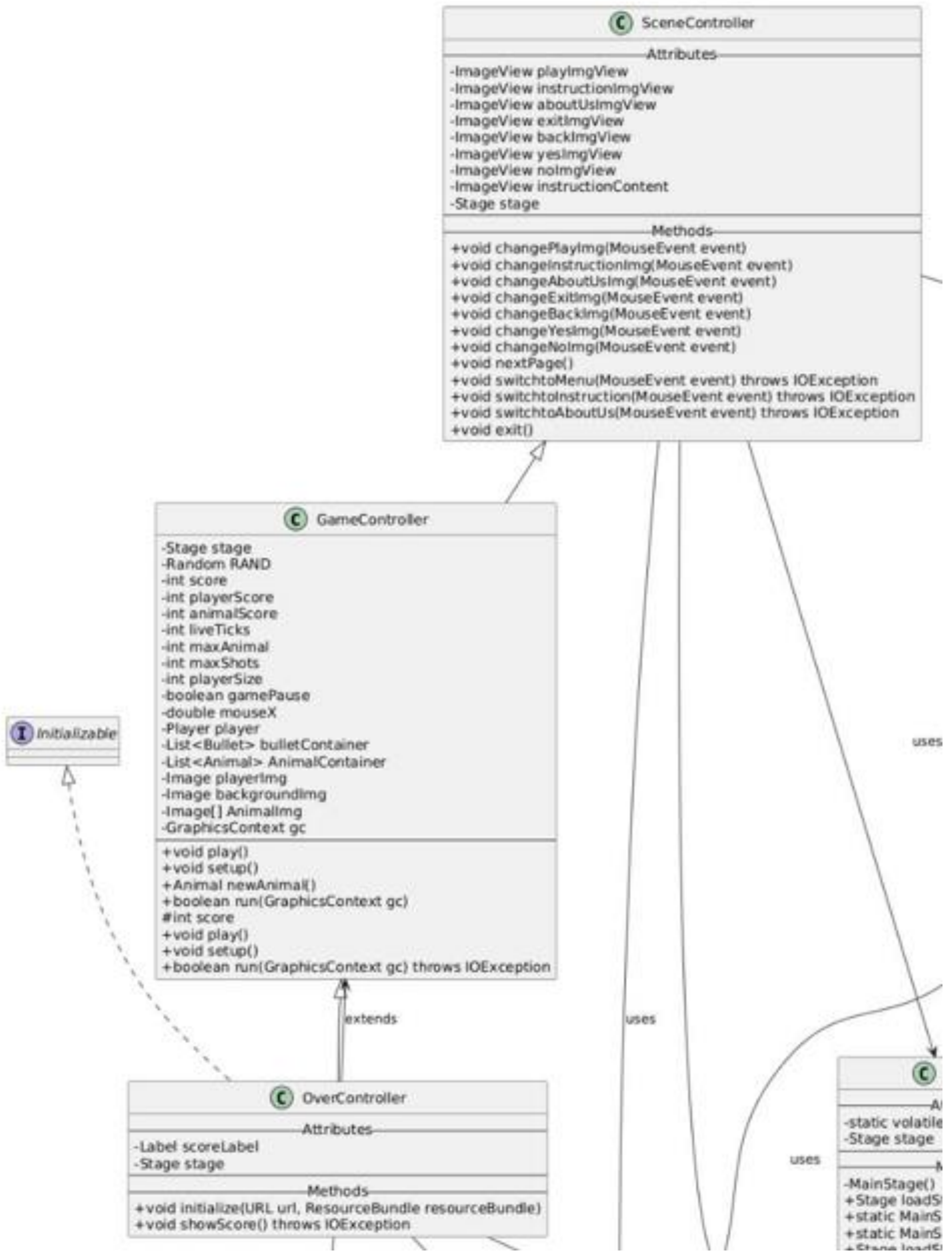
The **Player** class, derived from the **Entity** superclass, represents the player-controlled character in the game. In addition to inheriting basic properties and behaviors, it introduces a specialized method, **Shoot**, for firing bullets. This class also enables interactions with other entities within the game environment.

The **Bullet** class, derived from the **Entity** class, represents the projectiles fired by the Ship. It includes attributes such as speed, size, and a Boolean flag called **remove**, which indicates whether the bullet should be deleted. Additionally, it features methods for handling movement, detecting collisions, and managing impact effects.

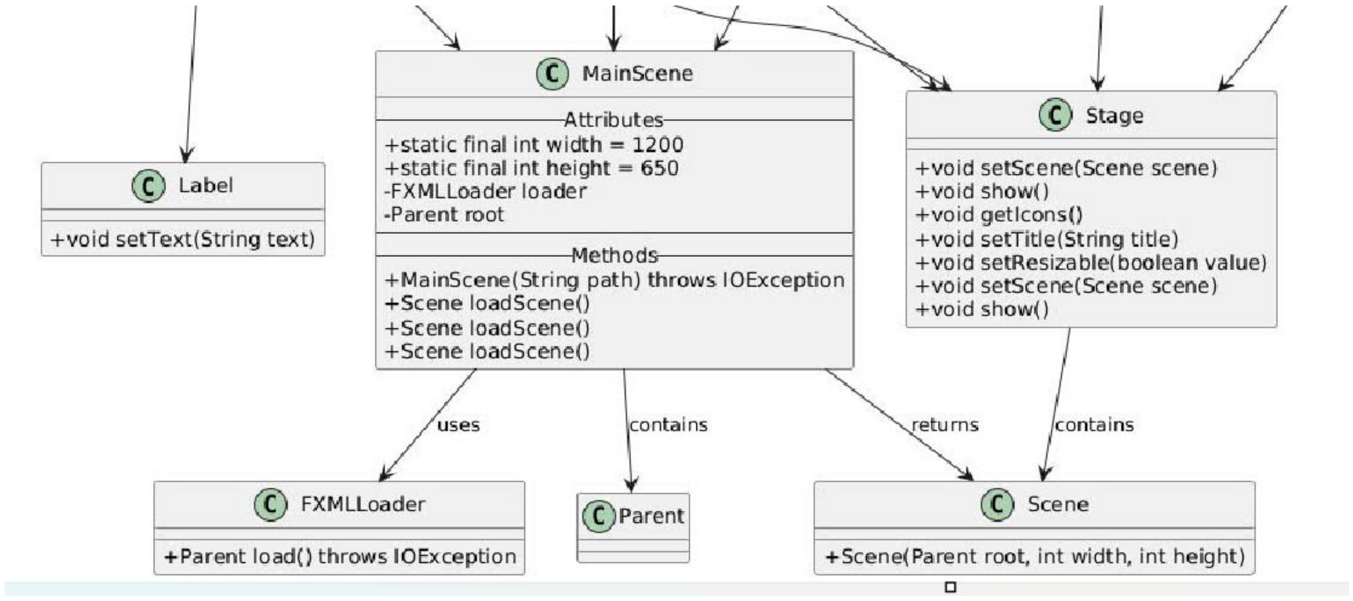
The **Animal** class, extending the **Entity** class, represents enemy units within the game. It includes unique attributes such as movement speed, which define its behavior.

The **Main** class functions as the entry point of the game application, managing the initialization process and setting up the fundamental aspects of the game stage.

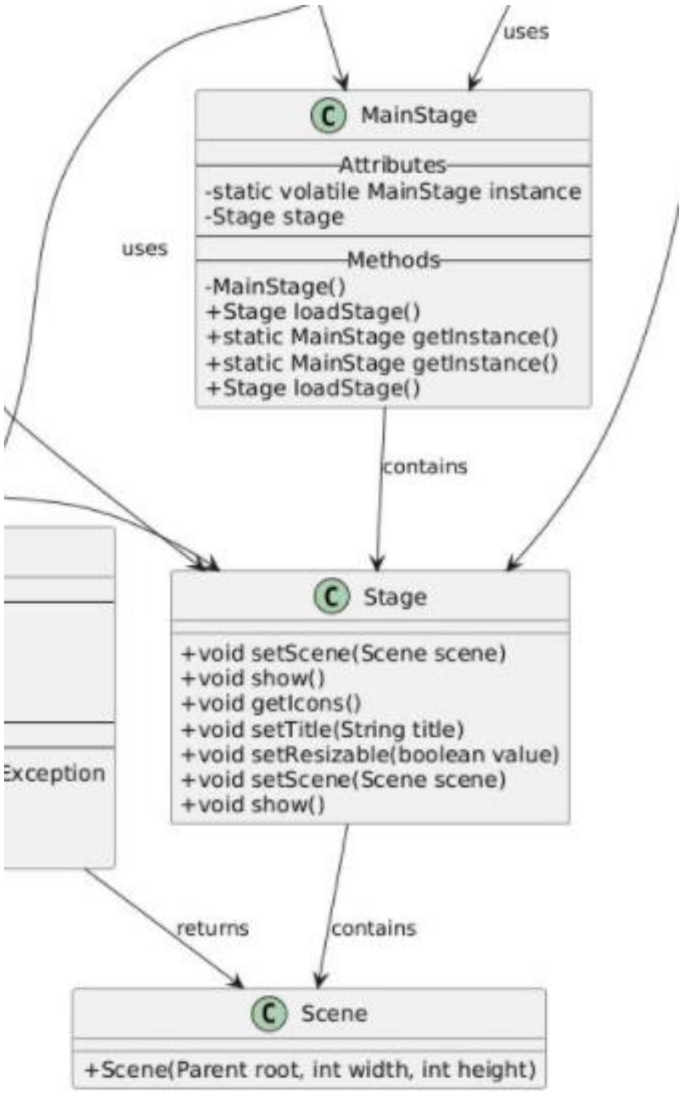
Controller process diagram


SceneController
Attributes
-ImageView playImgView
-ImageView instructionImgView
-ImageView aboutUsImgView
-ImageView exitImgView
-ImageView backImgView
-ImageView yesImgView
-ImageView noImgView
-ImageView instructionContent
-Stage stage
Methods
+void changePlayImg(MouseEvent event)
+void changeInstructionImg(MouseEvent event)
+void changeAboutUsImg(MouseEvent event)
+void changeExitImg(MouseEvent event)
+void changeBackImg(MouseEvent event)
+void changeYesImg(MouseEvent event)
+void changeNoImg(MouseEvent event)
+void nextPage()
+void switchtoMenu(MouseEvent event) throws IOException
+void switchtoInstruction(MouseEvent event) throws IOException
+void switchtoAboutUs(MouseEvent event) throws IOException
+void exit()
GameController
-Stage stage
-Random RAND
-int score
-int playerScore
-int animalScore
-int liveTicks
-int maxAnimal
-int maxShots
-int playerSize
-boolean gamePause
-double mouseX
-Player player
-List<Bullet> bulletContainer
-List<Animal> AnimalContainer
-Image playerImg
-Image backgroundImg
-Image[] AnimalImg
-GraphicsContext gc
+void play()
+void setup()
+Animal newAnimal()
+boolean run(GraphicsContext gc)
#int score
+void play()
+void setup()
+boolean run(GraphicsContext gc) throws IOException
Initializable
extends
uses
uses
uses
OverController
Attributes
-Label scoreLabel
-Stage stage
Methods
+void initialize(URL url, ResourceBundle resourceBundle)
+void showScore() throws IOException
-static volatile
-Stage stage
-MainStage()
+Stage loadS
+static MainS
+static MainS

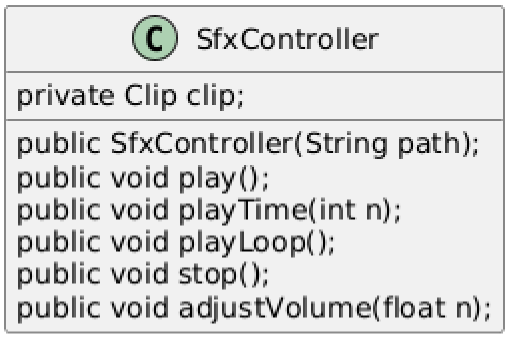
Scenes process diagram


Label
+void setText(String text)
MainScene
Attributes
+static final int width = 1200
+static final int height = 650
-FXMLLoader loader
-Parent root
Methods
+MainScene(String path) throws IOException
+Scene loadScene()
+Scene loadScene()
+Scene loadScene()
Stage
+void setScene(Scene scene)
+void show()
+void getIcons()
+void setTitle(String title)
+void setResizable(boolean value)
+void setScene(Scene scene)
+void show()
uses
contains
returns
contains
FXMLLoader
+Parent load() throws IOException
Parent
Scene
+Scene(Parent root, int width, int height)

Stage process diagram

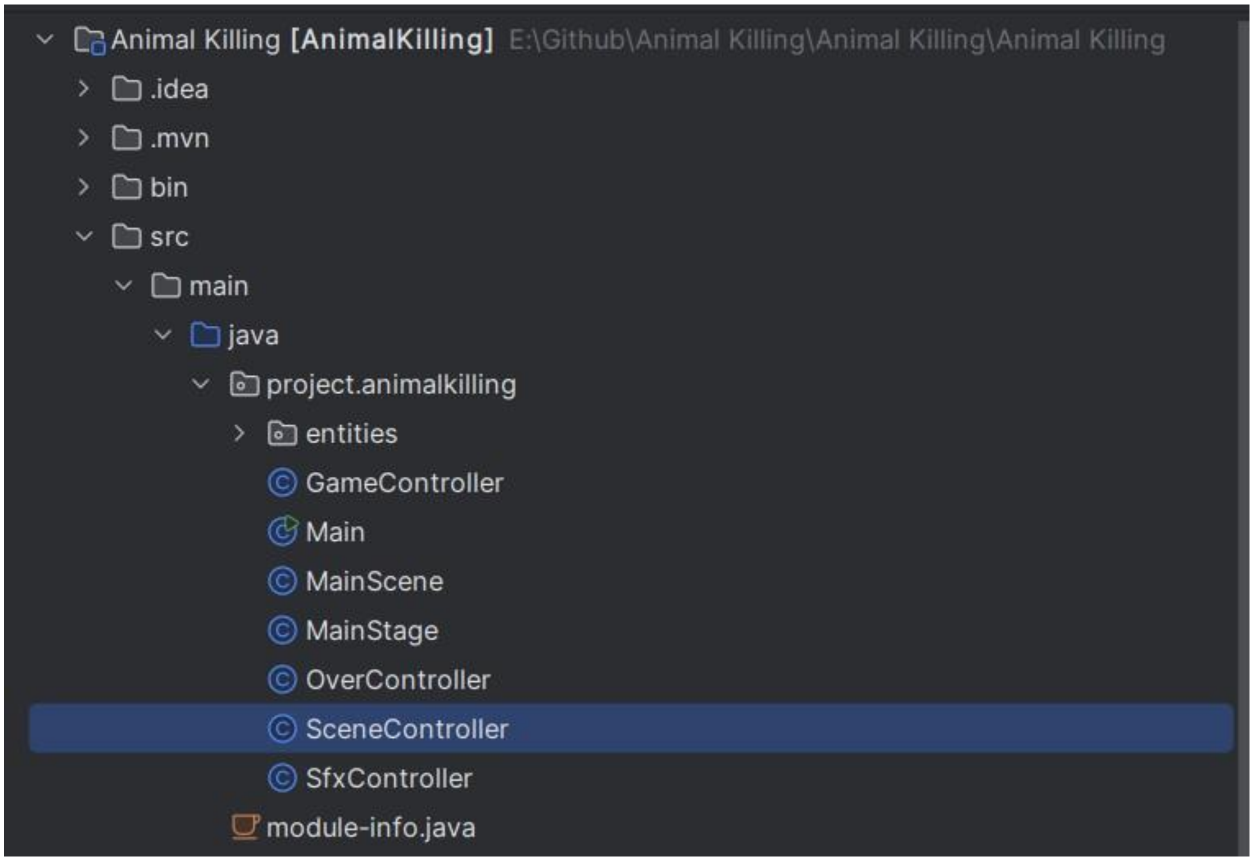

uses
uses
MainStage
Attributes
-static volatile MainStage instance
-Stage stage
Methods
-MainStage()
+Stage loadStage()
+static MainStage getInstance()
+static MainStage getInstance()
+Stage loadStage()
contains
Stage
+void setScene(Scene scene)
+void show()
+void getIcons()
+void setTitle(String title)
+void setResizable(boolean value)
+void setScene(Scene scene)
+void show()
Exception
returns
contains
Scene
+Scene(Parent root, int width, int height)

Sounds process diagram

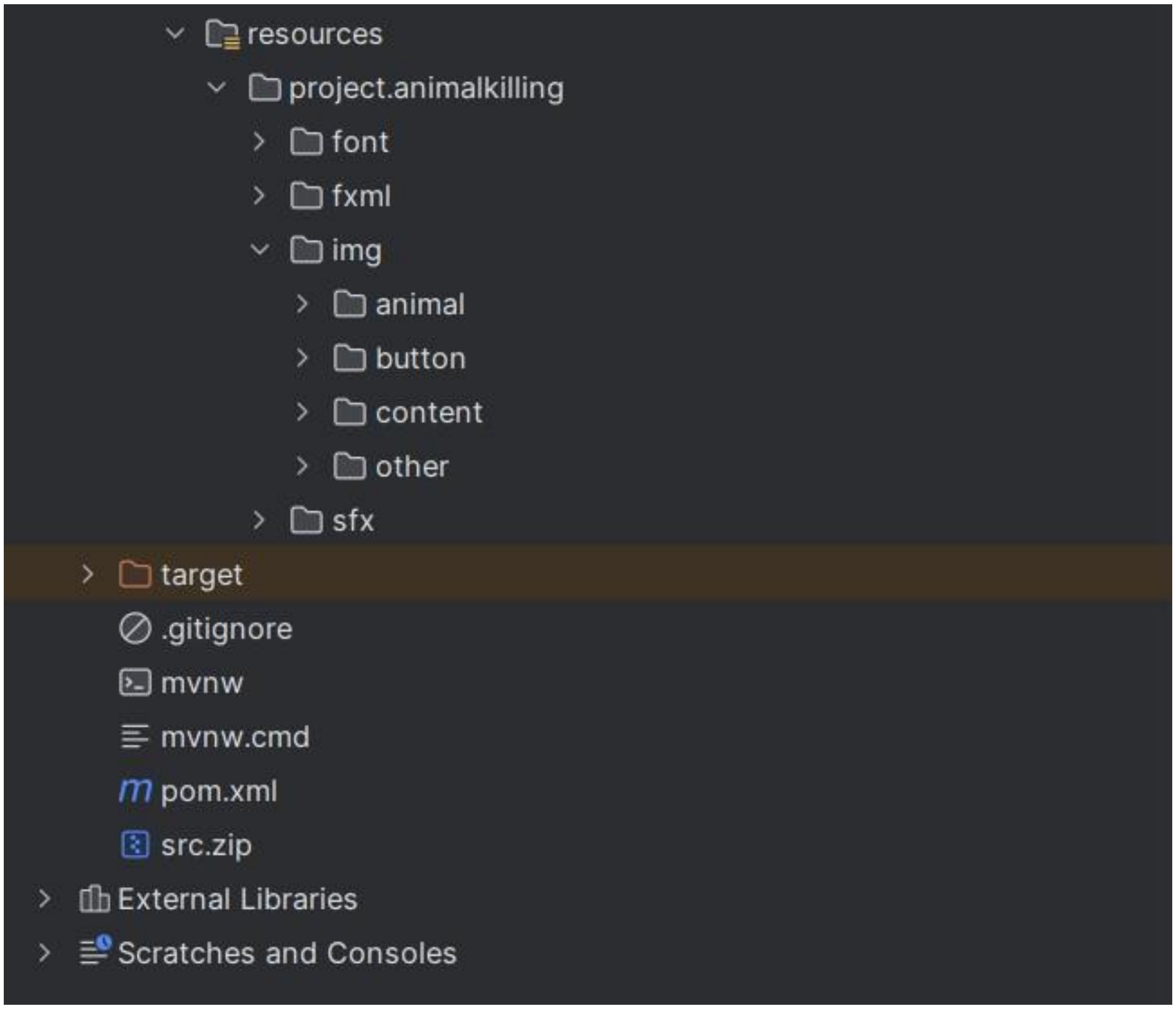

SfxController
private Clip clip;
public SfxController(String path);
public void play();
public void playTime(int n);
public void playLoop();
public void stop();
public void adjustVolume(float n);

CHAP 3: DESIGN AND IMPLEMENTATION

3.1. Project Structure



```
Animal Killing [AnimalKilling] E:\Github\Animal Killing\Animal Killing\Animal Killing
  .idea
  .mvn
  bin
  src
    main
      java
        project.animalkilling
          entities
          GameController
          Main
          MainScene
          MainStage
          OverController
          SceneController
          SfxController
        module-info.java
```



```
        resources
          project.animalkilling
            font
            fxml
            img
              animal
              button
              content
              other
            sfx
    target
    .gitignore
    mvnw
    mvnw.cmd
    pom.xml
    src.zip
  External Libraries
  Scratches and Consoles
```

The default configuration of this project is created with JavaFx generator and Maven build and it consists of two main directories: java and resources. The Java directory stores all the classes, while the resources directory holds the other asset files, such as images and Fxml files. We can notice that there is an outer directory named "project.animalkilling" that encloses both directories. This will later enable the java.lang. Class methods to access files across different packages without importing them.

3.2. Class Structures

3.2.1. Main.java:

The Main class is the entry point of the application and extends the JavaFX Application class. It initializes and displays the primary application stage, sets the user interface, and plays background audio effects.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage s) throws Exception {
```

- **Loading the Stage:** This line fetches a singleton instance of the MainStage class using its *getInstance()* method. Singleton pattern ensures that only one instance of MainStage exists. After that, the method *loadStage()* is used to get and work with the JavaFX Stage object that represents the main application window.

```
Stage stage = MainStage.getInstance().loadStage();
```

- **Creating the Scene**: The MainScene class is instantiated with the path to the menu.fxml file. The *loadScene()* method creates a Scene for the application, setting the width (1200) and height (650) defined in the MainScene class.

```
Scene menu = new
MainScene("fxml/menu.fxml").loadScene();
```

- This line loads an image resource located in the "img/other/logo.png" file. The *getResource()* retrieves the resource as a URL, and *toExternalForm()* converts it to a string to be used as the image's URL.

```
Image logo = new
Image(getClass().getResource("img/other/logo.png").toEx
ternalForm());
```

- Adds the logo image as an icon for the application window (stage).

```
stage.getIcons().add(logo);
```

- Creates a new SfxController object for playing sound effects. The "sfx/menu.wav" file is an audio file for the menu's background music. The sfx.playLoop() starts playing the sound effect indefinitely.

```
SfxController sfx = new SfxController("sfx/menu.wav");
sfx.playLoop();
```

- Sets the title of the application window to "Animal Killing".

```
stage.setTitle("Animal Killing");
```

- The line ensures the application window cannot be resized by the user.

```
stage.setResizable(false);
```

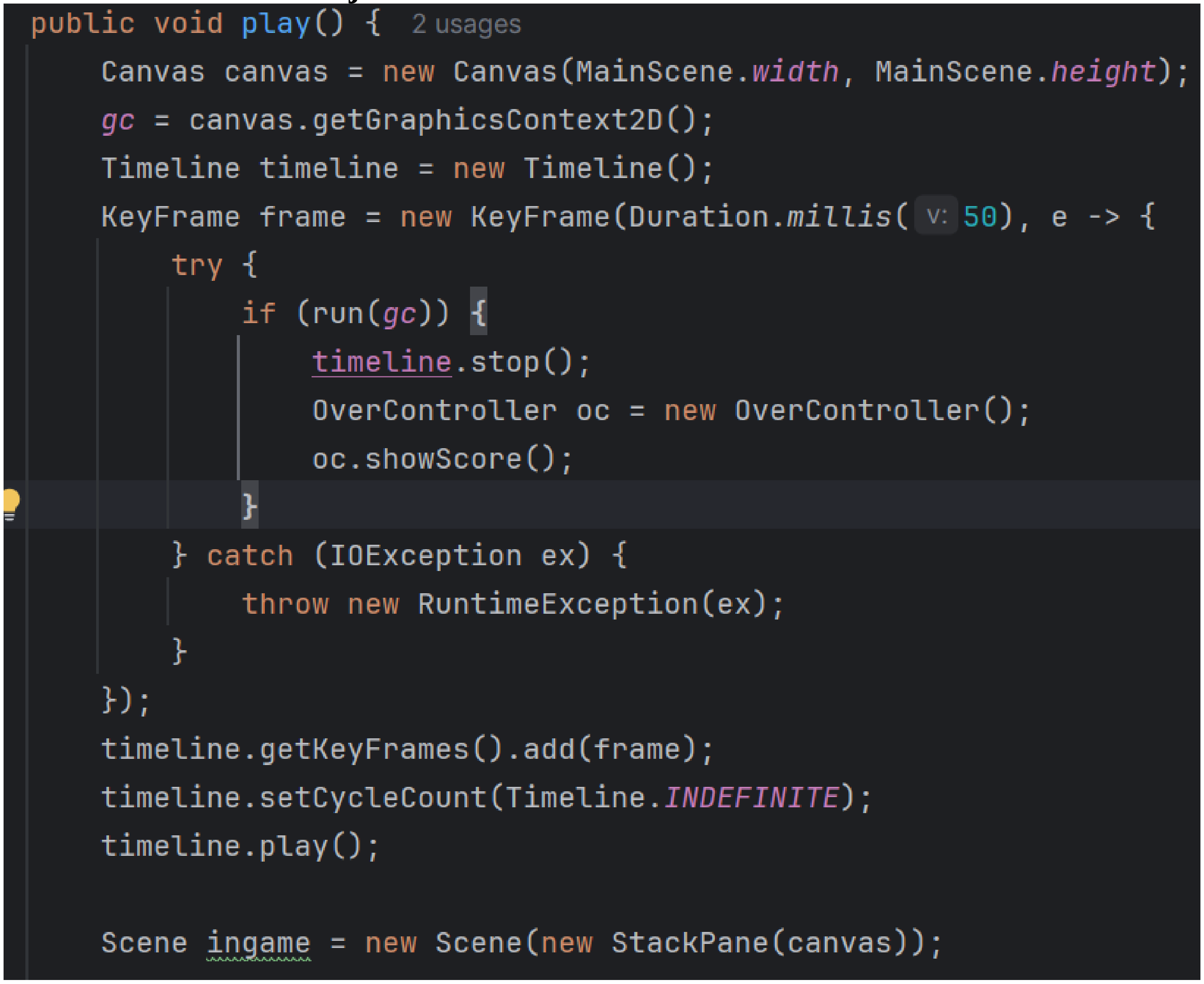
- Places the menu scene (loaded from the .fxml file) into the application stage. And the *stage.show()* makes the application window visible.

```
stage.setScene(menu);
stage.show();
```

- This is the entry point of the Java application. The *launch()* method is called, which starts the JavaFX application lifecycle, ultimately calling the *start()* method.

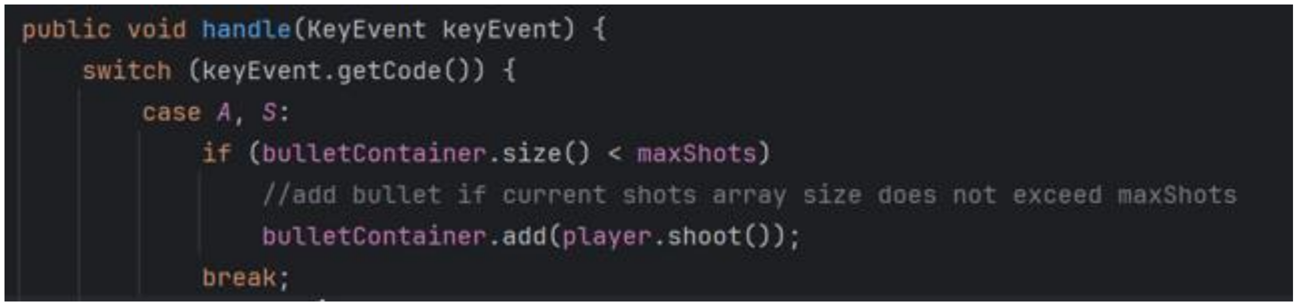
```
    public static void main(String[] args) {
        launch();
    }
}
```

3.2.2. GameController.java:

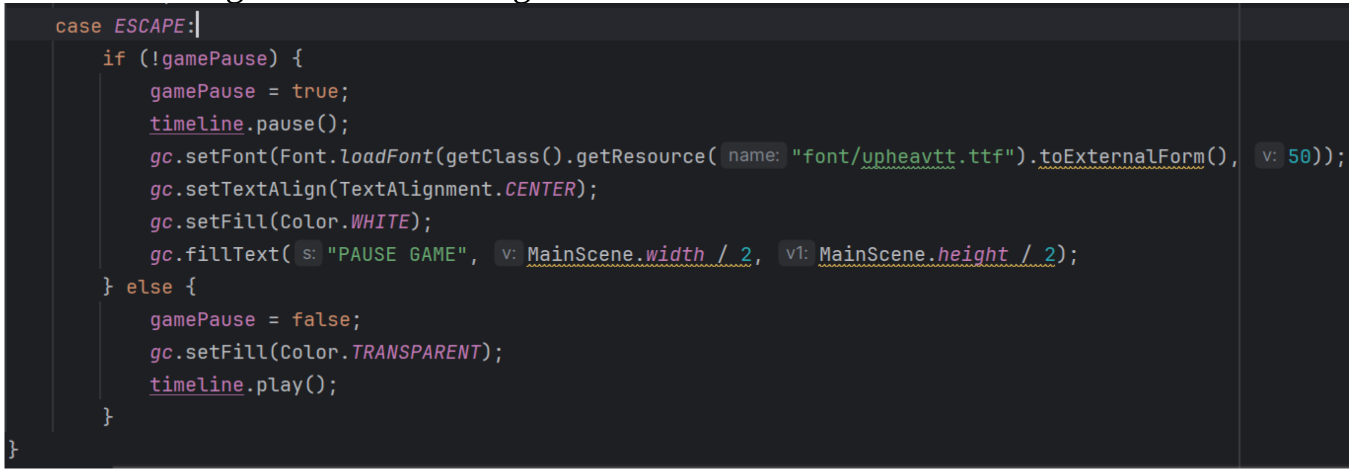


```
public void play() {  2 usages
    Canvas canvas = new Canvas(MainScene.width, MainScene.height);
    gc = canvas.getGraphicsContext2D();
    Timeline timeline = new Timeline();
    KeyFrame frame = new KeyFrame(Duration.millis( v: 50), e -> {
        try {
            if (run(gc)) {
                timeline.stop();
                OverController oc = new OverController();
                oc.showScore();
            }
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }
    });
    timeline.getKeyFrames().add(frame);
    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.play();

    Scene ingame = new Scene(new StackPane(canvas));
```

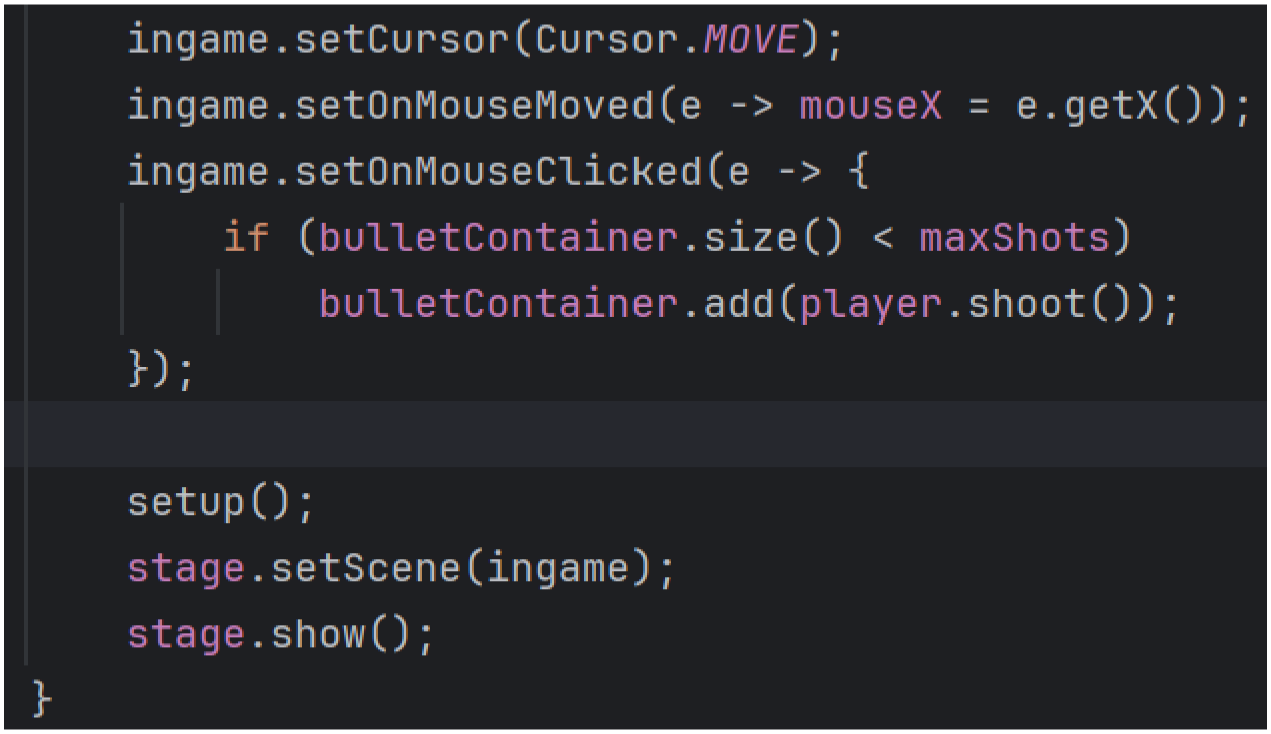
- A KeyFrame defines what happens during each "tick" or "frame" of the game loop.
- Duration.millis(50): Specifies that the loop executes every 50 milliseconds. This effectively sets the frame rate to 20 frames per second (as 1000ms / 50ms = 20).
- Game Over Handling:
  - If the game ends (run(gc) == true):
  - The Timeline is stopped (timeline.stop()), halting the game loop.
  - An OverController object (oc) is created to handle the game-over interface.
  - The showScore() method of OverController is called to display the final score.


```
public void handle(KeyEvent keyEvent) {
    switch (keyEvent.getCode()) {
        case A, S:
            if (bulletContainer.size() < maxShots)
                //add bullet if current shots array size does not exceed maxShots
                bulletContainer.add(player.shoot());
            break;
```

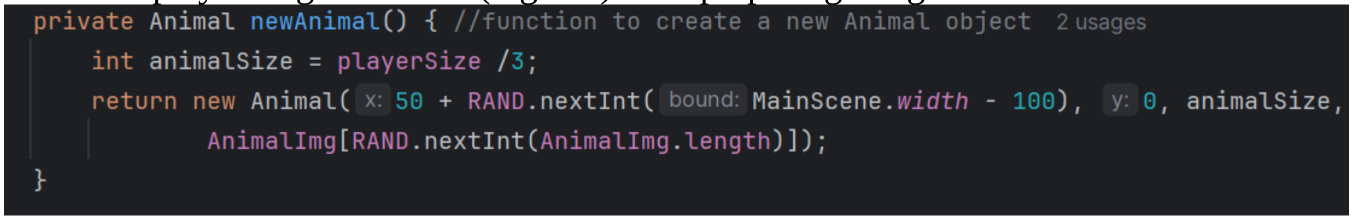
- Allows the player to shoot bullets when either the A or S key is released
- Checks if the current number of bullets (bulletContainer.size()) is less than the maxShots limit.
- If the bullet count doesn't exceed the maximum allowed (maxShots), a new bullet is created via player.shoot() (presumably a method in the Player class).
- The new bullet is added to the bulletContainer, which tracks all active bullets for rendering and movement logic.


```
    case ESCAPE:
        if (!gamePause) {
            gamePause = true;
            timeline.pause();
            gc.setFont(Font.loadFont(getClass().getResource( name: "font/upheavtt.ttf").toExternalForm(),  v: 50));
            gc.setTextAlign(TextAlignment.CENTER);
            gc.setFill(Color.WHITE);
            gc.fillText( s: "PAUSE GAME",  v: MainScene.width / 2,  v1: MainScene.height / 2);
        } else {
            gamePause = false;
            gc.setFill(Color.TRANSPARENT);
            timeline.play();
        }
}
```

- If the game is already paused (gamePause == true):
- Resumes gameplay by setting gamePause to false.
- Clears the pause screen by resetting the gc fill color.
- Restarts the Timeline (timeline.play()), resuming the normal game loop.

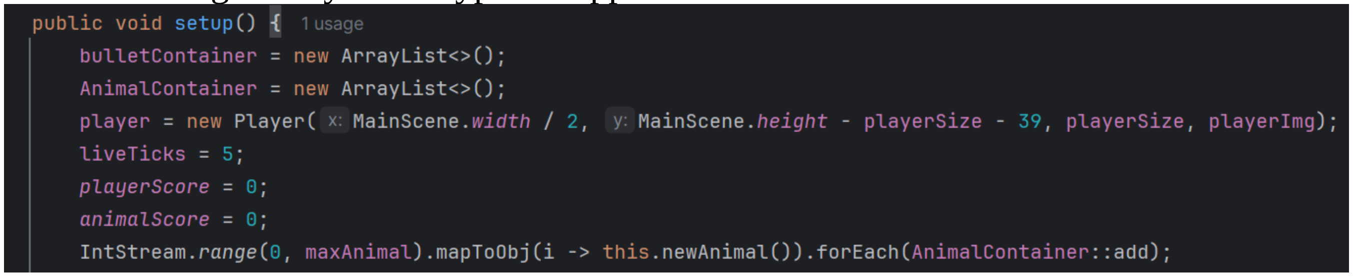

```
    ingame.setCursor(Cursor.MOVE);
    ingame.setOnMouseMoved(e -> mouseX = e.getX());
    ingame.setOnMouseClicked(e -> {
        if (bulletContainer.size() < maxShots)
            bulletContainer.add(player.shoot());
    });

    setup();
    stage.setScene(ingame);
    stage.show();
}
```

- Allows the player to fire bullets by clicking the mouse.
- Calls a setup() method to initialize important components like the player, enemies, and scoring system.
- Displays the game scene (ingame) after preparing the game environment.


```
private Animal newAnimal() { //function to create a new Animal object  2 usages
    int animalSize = playerSize /3;
    return new Animal( x: 50 + RAND.nextInt( bound: MainScene.width - 100),  y: 0, animalSize,
            AnimalImg[RAND.nextInt(AnimalImg.length)]);
}
```

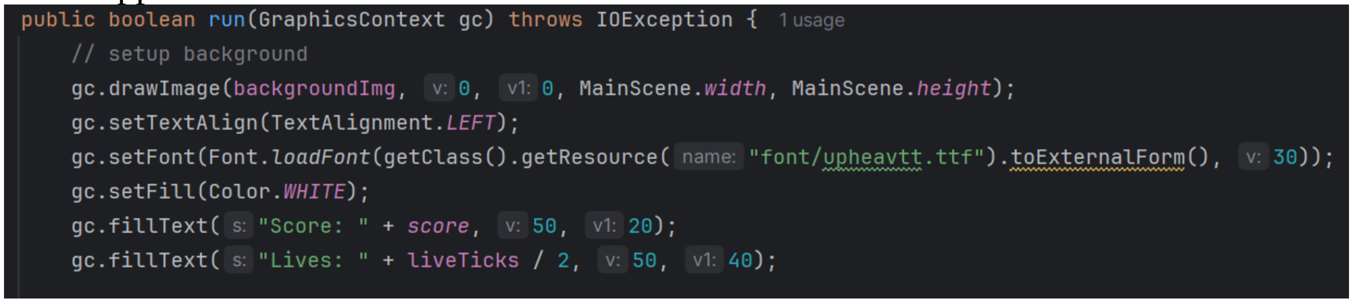
- The size is set to one-third the size of the player (playerSize) for consistency in scaling between the player and enemies. Smaller animals relative to the player make gameplay more challenging, as smaller enemies are harder to hit.
- X-Coordinate (50 + RAND.nextInt(MainScene.width - 100)): The horizontal position of the animal is randomized within a range to ensure it appears somewhere on the screen. RAND.nextInt(MainScene.width - 100) generates a random integer between 0 and the screen width minus a margin (100). Adding 50 ensures the animal does not spawn too close to the edges of the screen.
- Y-Coordinate (0): The vertical starting position is explicitly set to 0. This places the animal at the top of the screen, suggesting it will move downward during gameplay.
- Size (animalSize): The size of the animal is set to one-third of the player's size, creating smaller enemies relative to the player.

- AND.nextInt(AnimalImg.length) randomly selects an index from the array, ensuring variety in the types or appearances of animals.



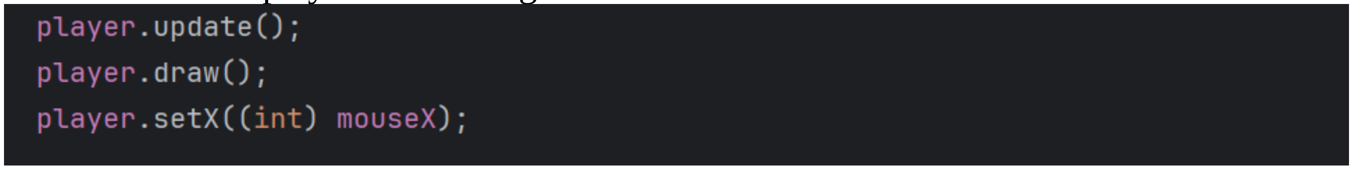
```java
public void setup() {  1 usage
    bulletContainer = new ArrayList<>();
    AnimalContainer = new ArrayList<>();
    player = new Player( x: MainScene.width / 2,  y: MainScene.height - playerSize - 39, playerSize, playerImg);
    liveTicks = 5;
    playerScore = 0;
    animalScore = 0;
    IntStream.range(0, maxAnimal).mapToObj(i -> this.newAnimal()).forEach(AnimalContainer::add);
```

- Creates an empty ArrayList to store the bullets fired by the player during gameplay.
- Creates an empty ArrayList to store Animal objects. AnimalContainer holds all the currently active enemies in the game.
- Creates a new Player object positioned at the bottom-center of the screen.The Player object is initialized with the following properties:
  - X-Coordinate (MainScene.width / 2): Places the player horizontally in the middle of the screen.
  - Y-Coordinate (MainScene.height - playerSize - 39): Positions the player at the bottom of the screen, leaving space for the character's size (playerSize) and some margin (39 pixels).
- Size (playerSize): Determines the player's size.
- Image (playerImg): Defines the player's visual representation using an image.
- Ensures the player starts the game at a consistent position and with the correct appearance



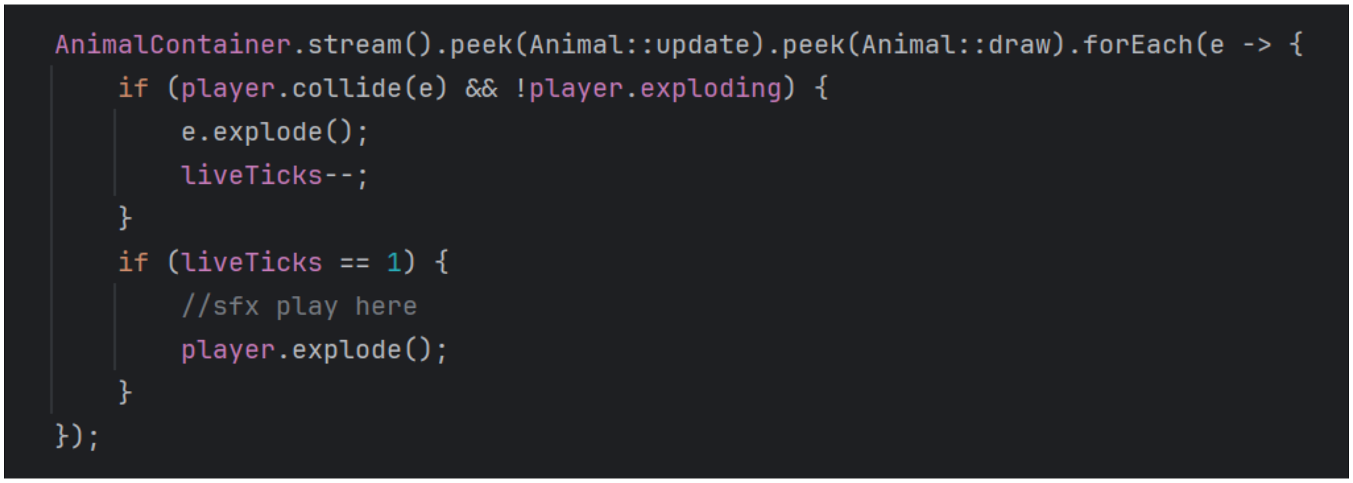
```java
public boolean run(GraphicsContext gc) throws IOException {  1 usage
    // setup background
    gc.drawImage(backgroundImg,  v: 0,  v1: 0, MainScene.width, MainScene.height);
    gc.setTextAlign(TextAlignment.LEFT);
    gc.setFont(Font.loadFont(getClass().getResource( name: "font/upheavtt.ttf").toExternalForm(),  v: 30));
    gc.setFill(Color.WHITE);
    gc.fillText( s: "Score: " + score,  v: 50,  v1: 20);
    gc.fillText( s: "Lives: " + liveTicks / 2,  v: 50,  v1: 40);
```

- Ensures that all subsequent text (like score and lives) is aligned to the left side of its starting position.
- Sets the color of all subsequent text to white.
- Draws the player's current score on the screen.
- Draws the player's remaining lives on the screen.



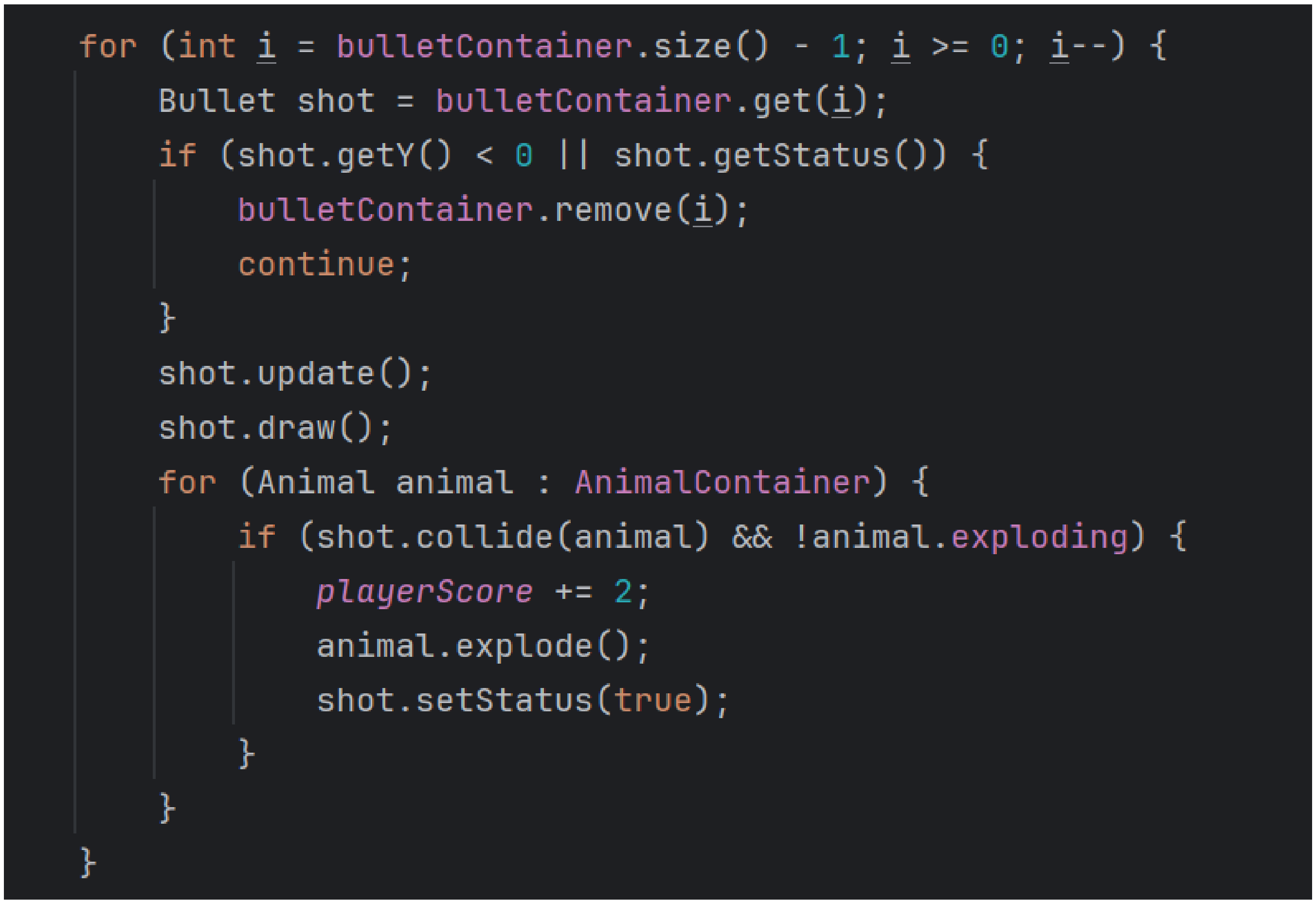
```java
player.update();
player.draw();
player.setX((int) mouseX);
```

- Updates the player position, redraws it on the screen, and synchronizes the player's X-coordinate with the current mouse position.



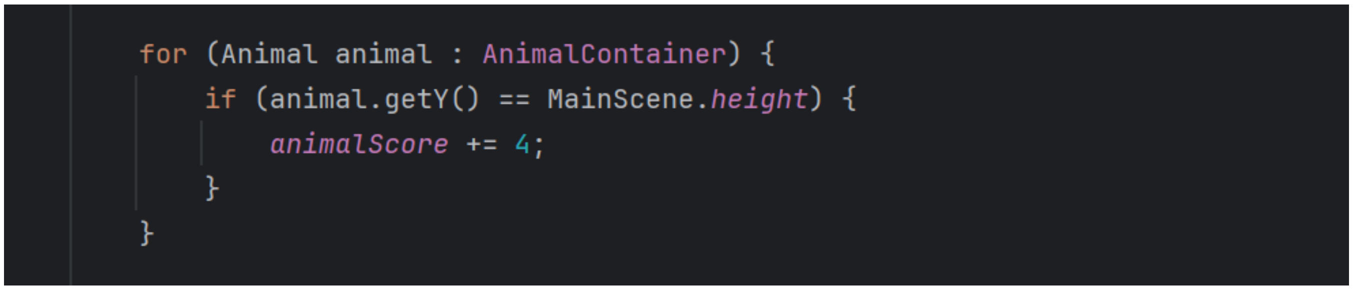
```java
AnimalContainer.stream().peek(Animal::update).peek(Animal::draw).forEach(e -> {
    if (player.collide(e) && !player.exploding) {
        e.explode();
        liveTicks--;
    }
    if (liveTicks == 1) {
        //sfx play here
        player.explode();
    }
});
```

- stream(): Converts the AnimalContainer list into a stream for functional programming.
- .peek(Animal::update): Updates the movement and state of each Animal object (e.g., moves them downward).
- .peek(Animal::draw): Draws each Animal object at its updated position.
- .forEach(e -> { ... }): For each animal (e):
- player.collide(e): Checks if the player collides with the current animal (e).
- !player.exploding: Ensures the player isn't already in an exploded state.
- Action: If a collision occurs:
  - The animal is destroyed by calling e.explode().
  - The player's lives (liveTicks) are reduced by 1.



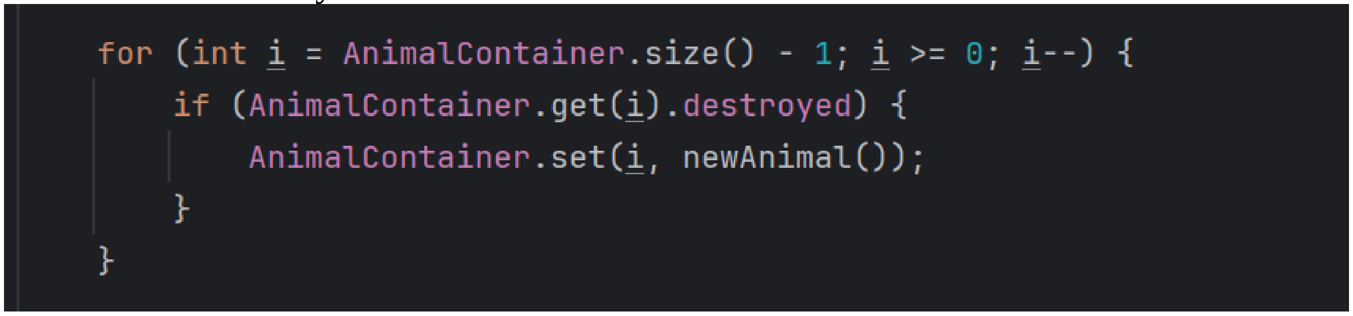
```java
for (int i = bulletContainer.size() - 1; i >= 0; i--) {
    Bullet shot = bulletContainer.get(i);
    if (shot.getY() < 0 || shot.getStatus()) {
        bulletContainer.remove(i);
        continue;
    }
    shot.update();
    shot.draw();
    for (Animal animal : AnimalContainer) {
        if (shot.collide(animal) && !animal.exploding) {
            playerScore += 2;
            animal.explode();
            shot.setStatus(true);
        }
    }
}
```

- Iterate Through Bullets: Loops through the bulletContainer list from the last bullet to the first (reversing prevents index issues when removing items).
- Bullet Cleanup: If a bullet's Y-coordinate (shot.getY()) is out of bounds (above the screen) or if it has already hit a target (shot.getStatus()), it is removed from bulletContainer.
- Update Each Bullet:
  - Calls shot.update() to update the bullet's position.
  - Calls shot.draw() to render the bullet on the screen.
- Check Collision with Animals: For each animal, checks:
  - shot.collide(animal): If the bullet collides with an animal.
  - !animal.exploding: Ensures the animal isn't already in an exploded state.
- Action:
  - The animal is destroyed by calling animal.explode().
  - The bullet is marked as "used" by setting shot.setStatus(true).
  - The player gains points (playerScore += 2).



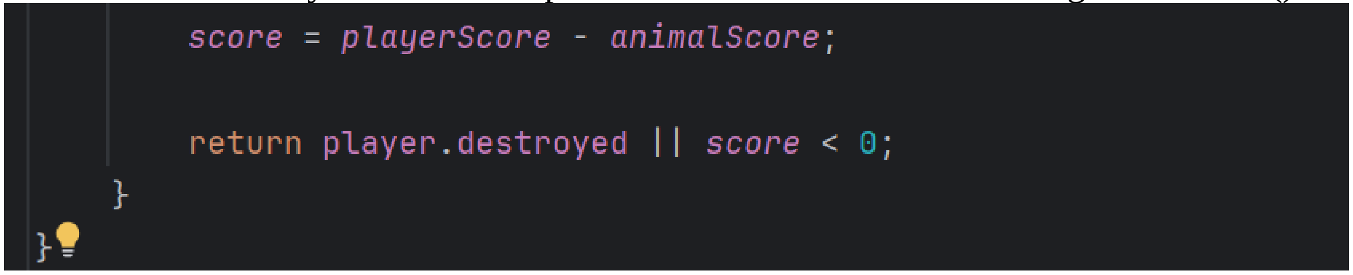
```java
for (Animal animal : AnimalContainer) {
    if (animal.getY() == MainScene.height) {
        animalScore += 4;
    }
}
```

- animal.getY(): Retrieves the Y-coordinate of the current animal.
- MainScene.height: Represents the height of the game screen (the bottom edge).
- Action: If the animal's Y-coordinate matches the bottom edge, it increments the animalScore by 4.



```java
for (int i = AnimalContainer.size() - 1; i >= 0; i--) {
    if (AnimalContainer.get(i).destroyed) {
        AnimalContainer.set(i, newAnimal());
    }
}
```

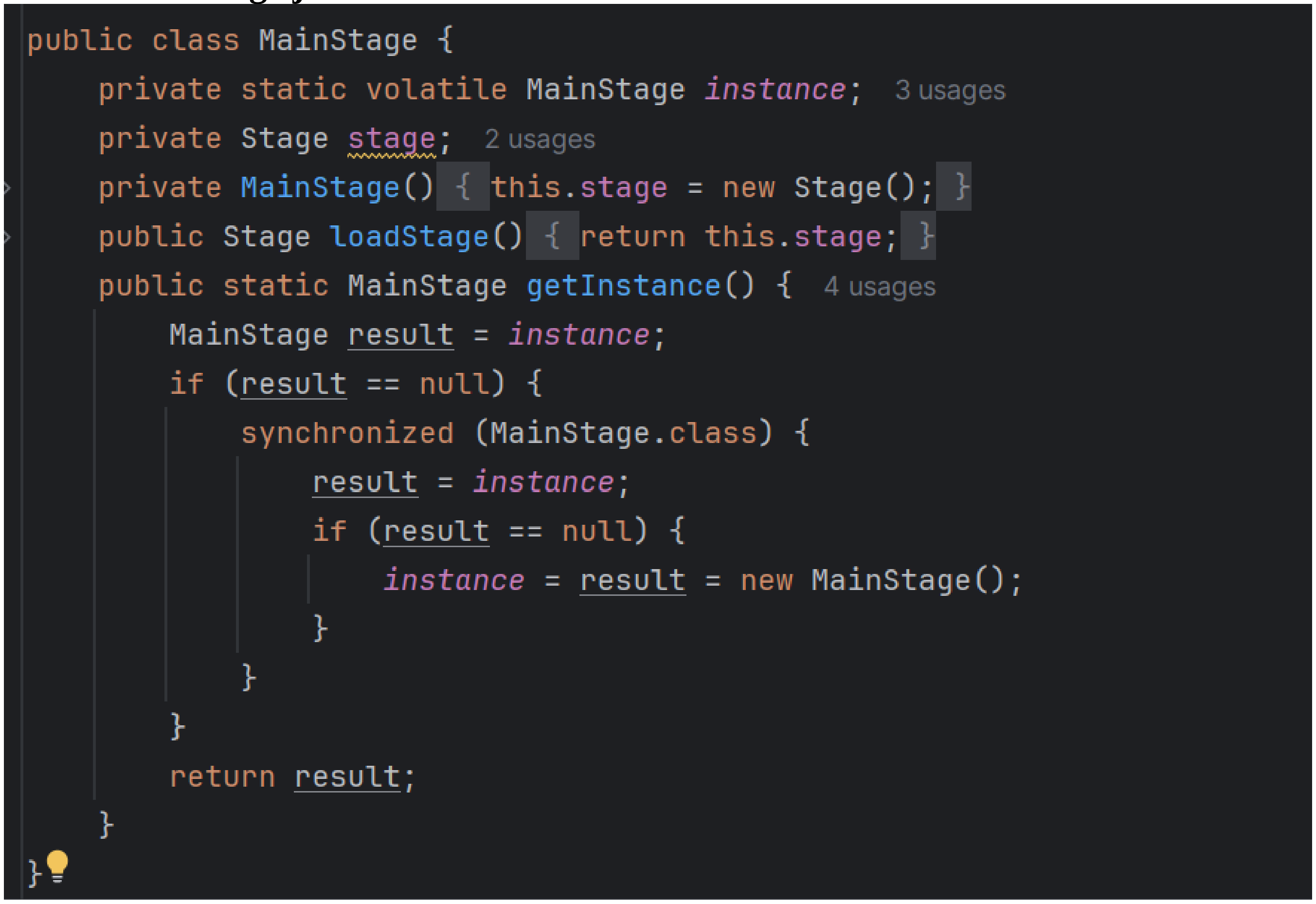
- Iterates through the AnimalContainer in reverse order.
- Checks:
  - AnimalContainer.get(i).destroyed: If the animal at index i is marked as "destroyed".
- If true:
  - The destroyed animal is replaced with a new one created using newAnimal().



```java
        score = playerScore - animalScore;

        return player.destroyed || score < 0;
    }
}
```

  - Returns a boolean value to indicate whether the game should terminate:
    - true: The game is over (player is destroyed or the score is negative).
    - false: The game can continue (player is still alive, and the score is non-negative).

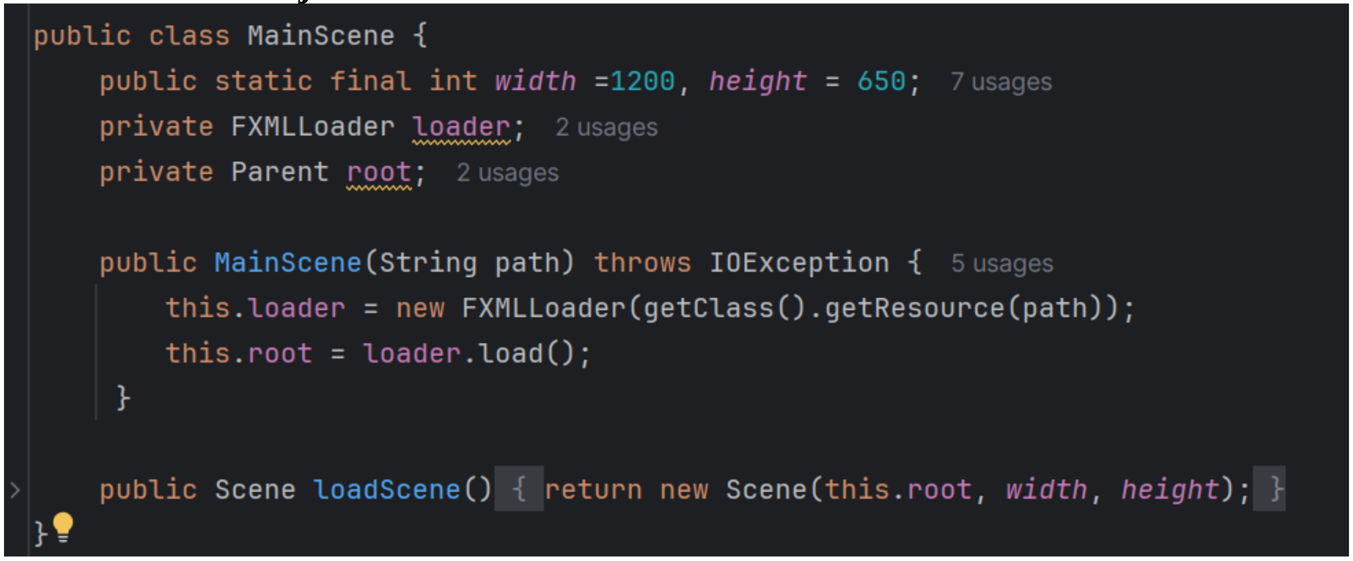
3.2.3. MainStage.java:



```
public class MainStage {
    private static volatile MainStage instance;  3 usages
    private Stage stage;  2 usages
    private MainStage() { this.stage = new Stage(); }
    public Stage loadStage() { return this.stage; }
    public static MainStage getInstance() {  4 usages
        MainStage result = instance;
        if (result == null) {
            synchronized (MainStage.class) {
                result = instance;
                if (result == null) {
                    instance = result = new MainStage();
                }
            }
        }
        return result;
    }
}
```

- Only one Stage is created and reused throughout the program.
- Thread-safe lazy initialization.
- Easy access to the Stage via the getInstance() and loadStage() methods

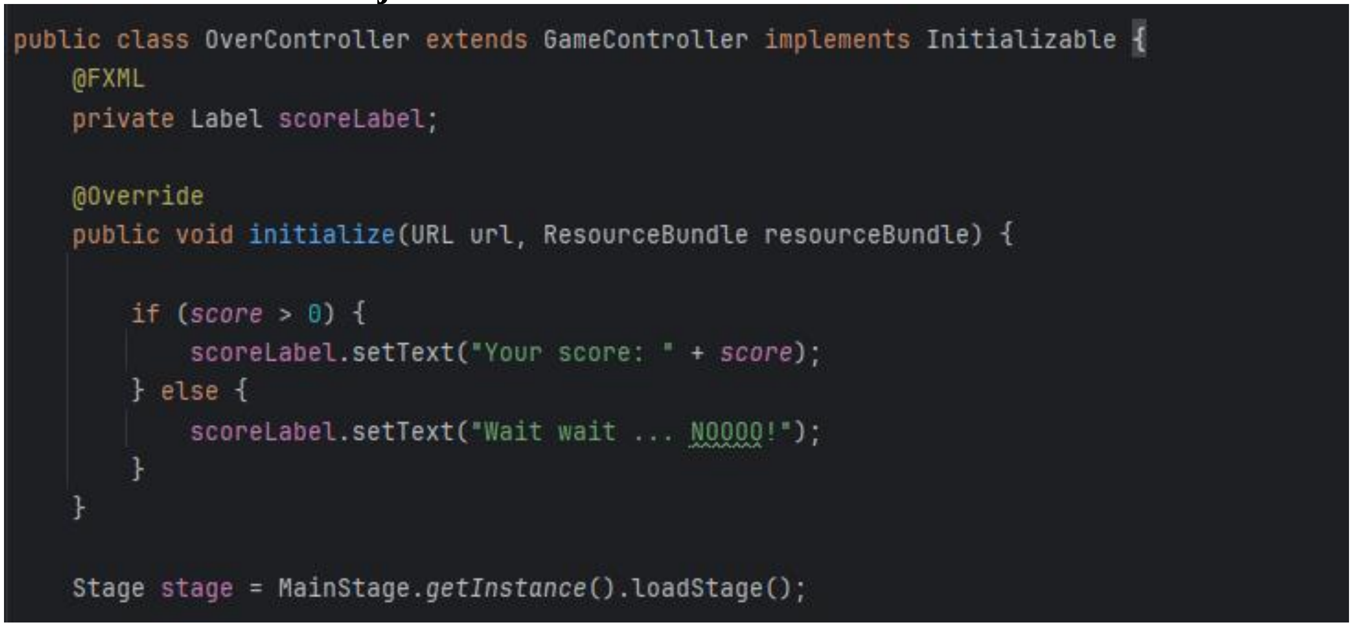
3.2.4. Mainscene.java:



```
public class MainScene {
    public static final int width =1200, height = 650;  7 usages
    private FXMLLoader loader;  2 usages
    private Parent root;  2 usages

    public MainScene(String path) throws IOException {  5 usages
        this.loader = new FXMLLoader(getClass().getResource(path));
        this.root = loader.load();
    }

    public Scene loadScene() { return new Scene(this.root, width, height); }
}
```

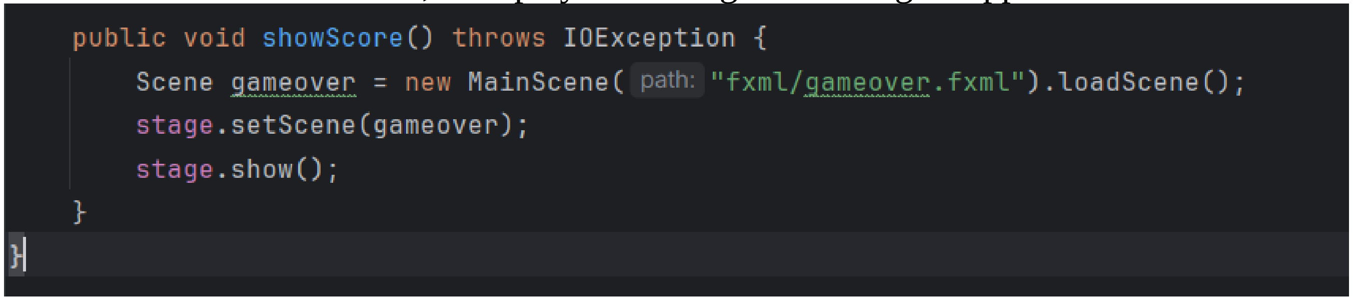
- Loading a GUI layout from an FXML file.
- Creating a Scene with a fixed width and height (1200x650).
- Dynamically generating the GUI structure, allowing flexibility in managing layouts.
- This code enables a modular and reusable way to manage scenes in JavaFX applications, especially for projects with multiple GUIs (such as menus, settings, and game screens).

3.2.5. OverController.java:



```
public class OverController extends GameController implements Initializable {
    @FXML
    private Label scoreLabel;

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {

        if (score > 0) {
            scoreLabel.setText("Your score: " + score);
        } else {
            scoreLabel.setText("Wait wait ... NOOOO!");
        }
    }

    Stage stage = MainStage.getInstance().loadStage();
```

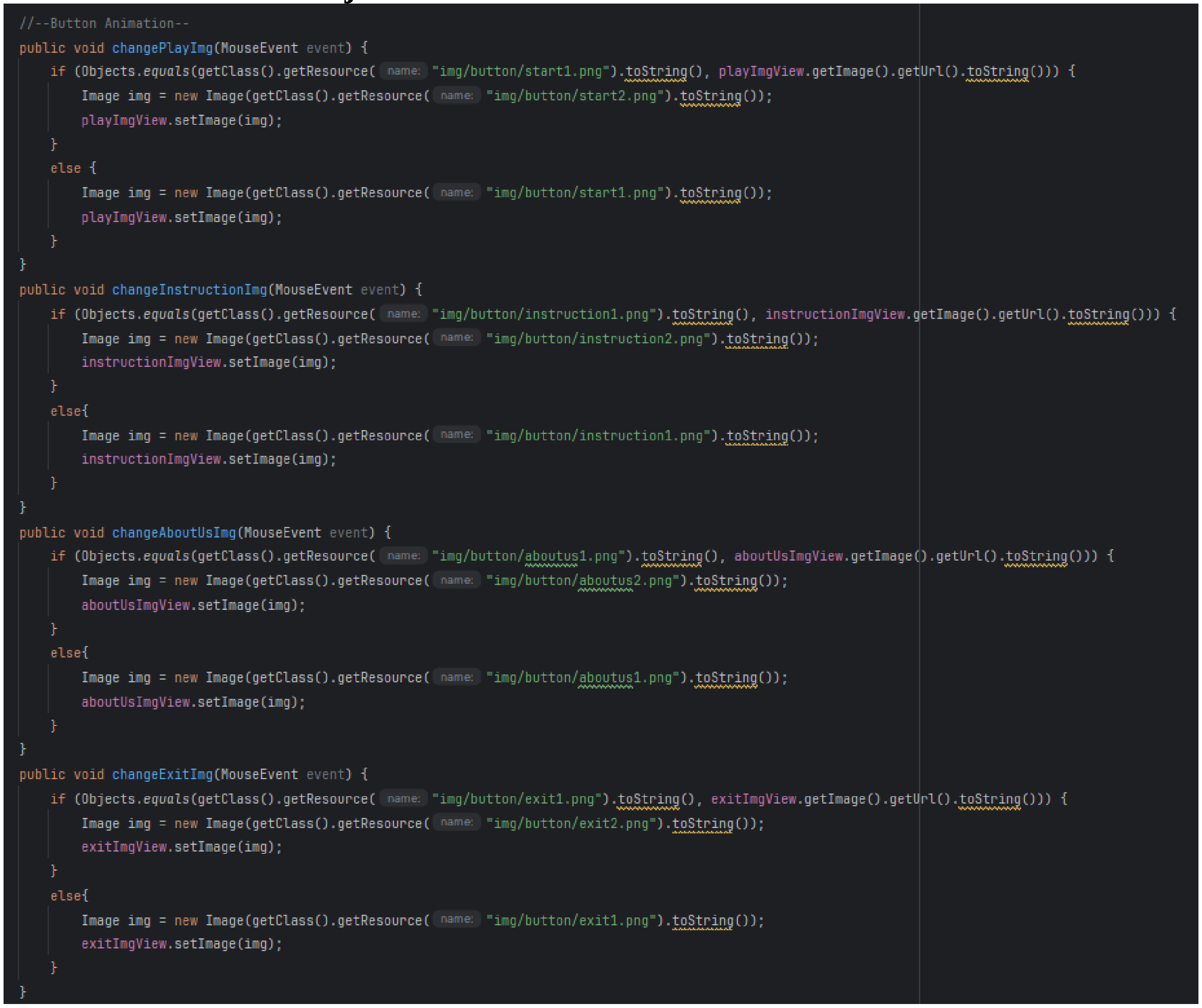
- This method checks the score variable and updates the scoreLabel accordingly:
- If score is greater than 0, it displays the player's score.
- If score is 0 or less, it displays a message indicating disappointment.



```
    public void showScore() throws IOException {
        Scene gameover = new MainScene( path: "fxml/gameover.fxml").loadScene();
        stage.setScene(gameover);
        stage.show();
    }
}
```

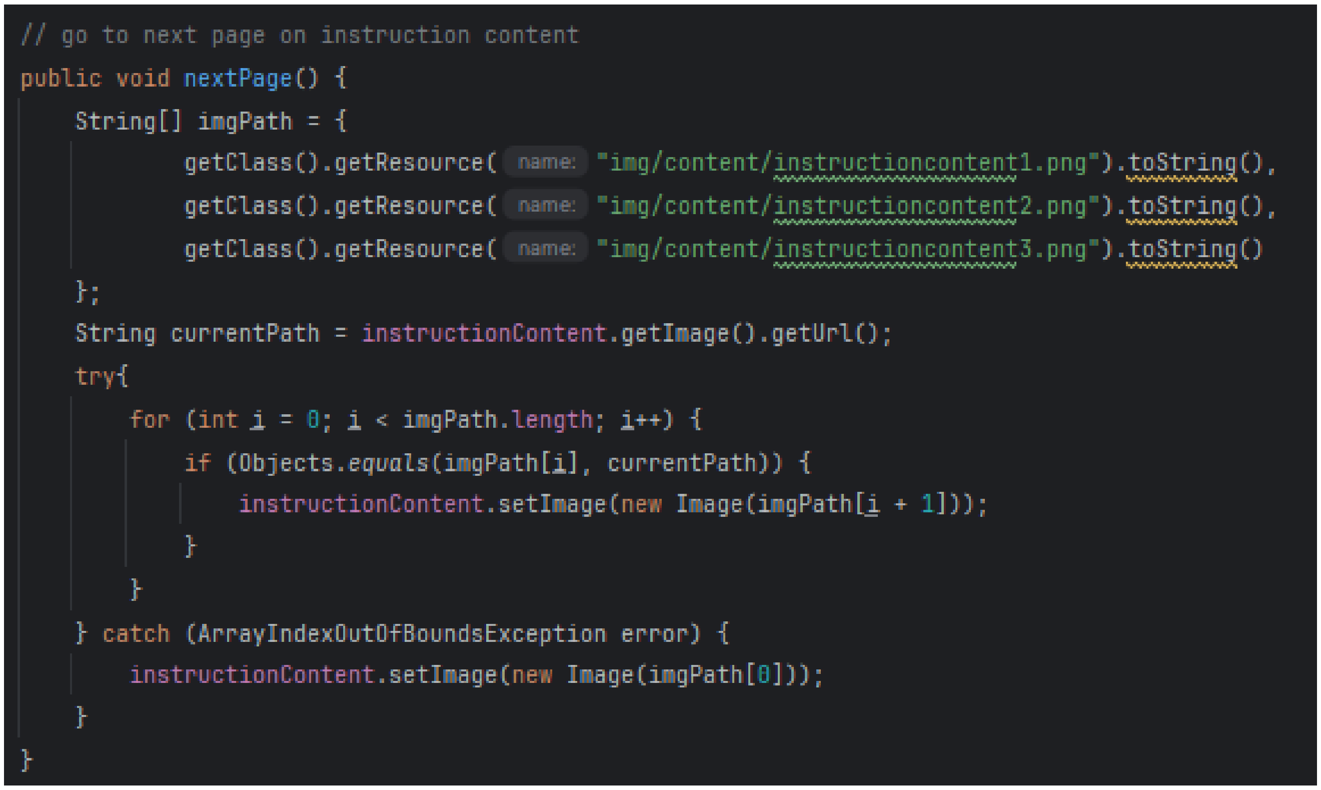
- This method is responsible for transitioning the application to the game over scene. It loads a new scene from an FXML file (gameover.fxml) and sets it as the current scene of the stage. Finally, it makes the stage visible by calling *stage.show()*.

3.2.6. SceneController.java:



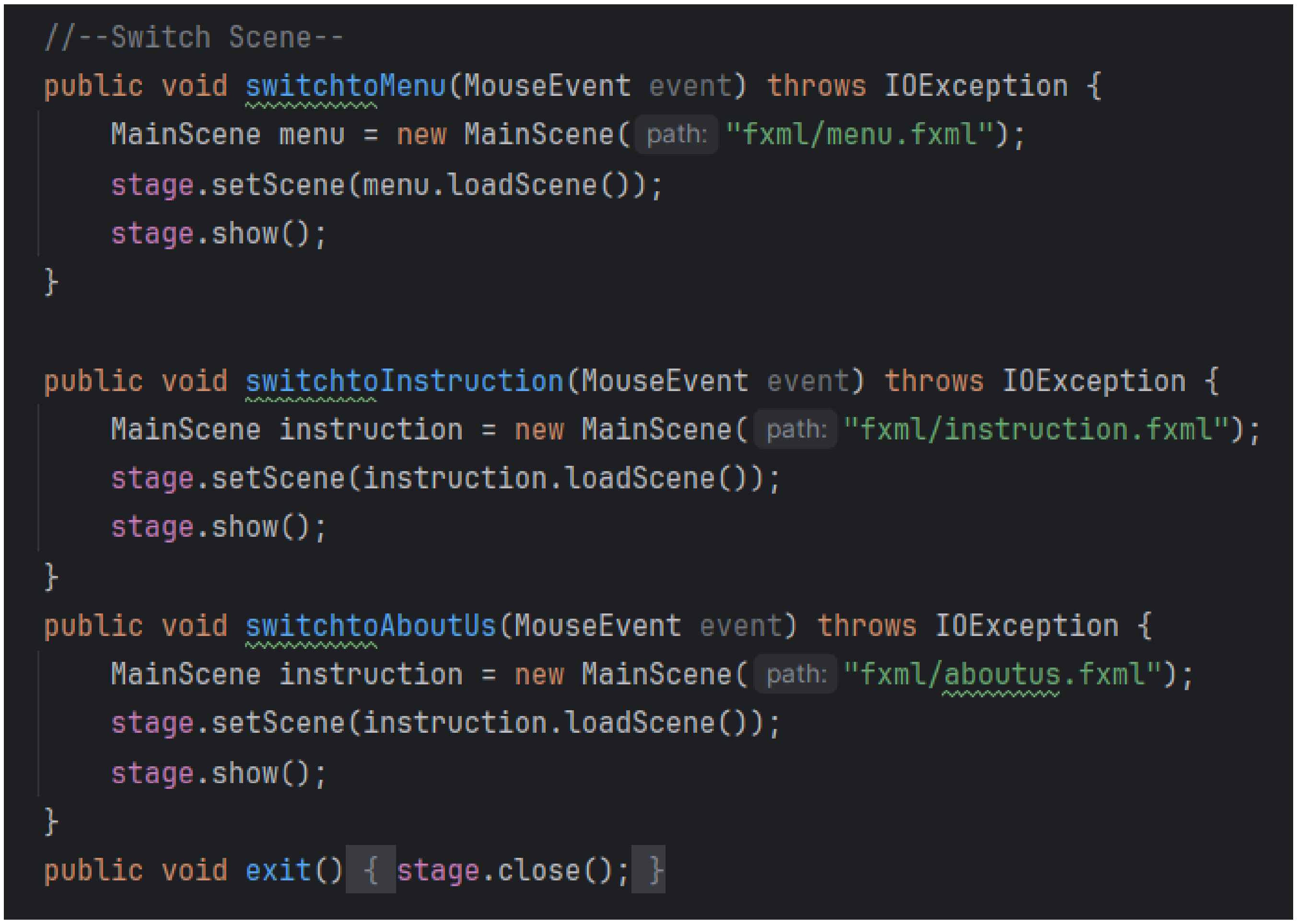
```java
//--Button Animation--
public void changePlayImg(MouseEvent event) {
    if (Objects.equals(getClass().getResource( name: "img/button/start1.png").toString(), playImgView.getImage().getUrl().toString())) {
        Image img = new Image(getClass().getResource( name: "img/button/start2.png").toString());
        playImgView.setImage(img);
    }
    else {
        Image img = new Image(getClass().getResource( name: "img/button/start1.png").toString());
        playImgView.setImage(img);
    }
}
public void changeInstructionImg(MouseEvent event) {
    if (Objects.equals(getClass().getResource( name: "img/button/instruction1.png").toString(), instructionImgView.getImage().getUrl().toString())) {
        Image img = new Image(getClass().getResource( name: "img/button/instruction2.png").toString());
        instructionImgView.setImage(img);
    }
    else{
        Image img = new Image(getClass().getResource( name: "img/button/instruction1.png").toString());
        instructionImgView.setImage(img);
    }
}
public void changeAboutUsImg(MouseEvent event) {
    if (Objects.equals(getClass().getResource( name: "img/button/aboutus1.png").toString(), aboutUsImgView.getImage().getUrl().toString())) {
        Image img = new Image(getClass().getResource( name: "img/button/aboutus2.png").toString());
        aboutUsImgView.setImage(img);
    }
    else{
        Image img = new Image(getClass().getResource( name: "img/button/aboutus1.png").toString());
        aboutUsImgView.setImage(img);
    }
}
public void changeExitImg(MouseEvent event) {
    if (Objects.equals(getClass().getResource( name: "img/button/exit1.png").toString(), exitImgView.getImage().getUrl().toString())) {
        Image img = new Image(getClass().getResource( name: "img/button/exit2.png").toString());
        exitImgView.setImage(img);
    }
    else{
        Image img = new Image(getClass().getResource( name: "img/button/exit1.png").toString());
        exitImgView.setImage(img);
    }
}
```

- Button Animation: Each of the following methods changes the image displayed in the corresponding ImageView when the user interacts with the button. Each method checks the current image URL of the ImageView. If it matches the first image (start1.png), it switches to the second image (start2.png), and vice versa. This toggling effect provides visual feedback to the user.



```java
// go to next page on instruction content
public void nextPage() {
    String[] imgPath = {
            getClass().getResource( name: "img/content/instructioncontent1.png").toString(),
            getClass().getResource( name: "img/content/instructioncontent2.png").toString(),
            getClass().getResource( name: "img/content/instructioncontent3.png").toString()
    };
    String currentPath = instructionContent.getImage().getUrl();
    try{
        for (int i = 0; i < imgPath.length; i++) {
            if (Objects.equals(imgPath[i], currentPath)) {
                instructionContent.setImage(new Image(imgPath[i + 1]));
            }
        }
    } catch (ArrayIndexOutOfBoundsException error) {
        instructionContent.setImage(new Image(imgPath[0]));
    }
}
```

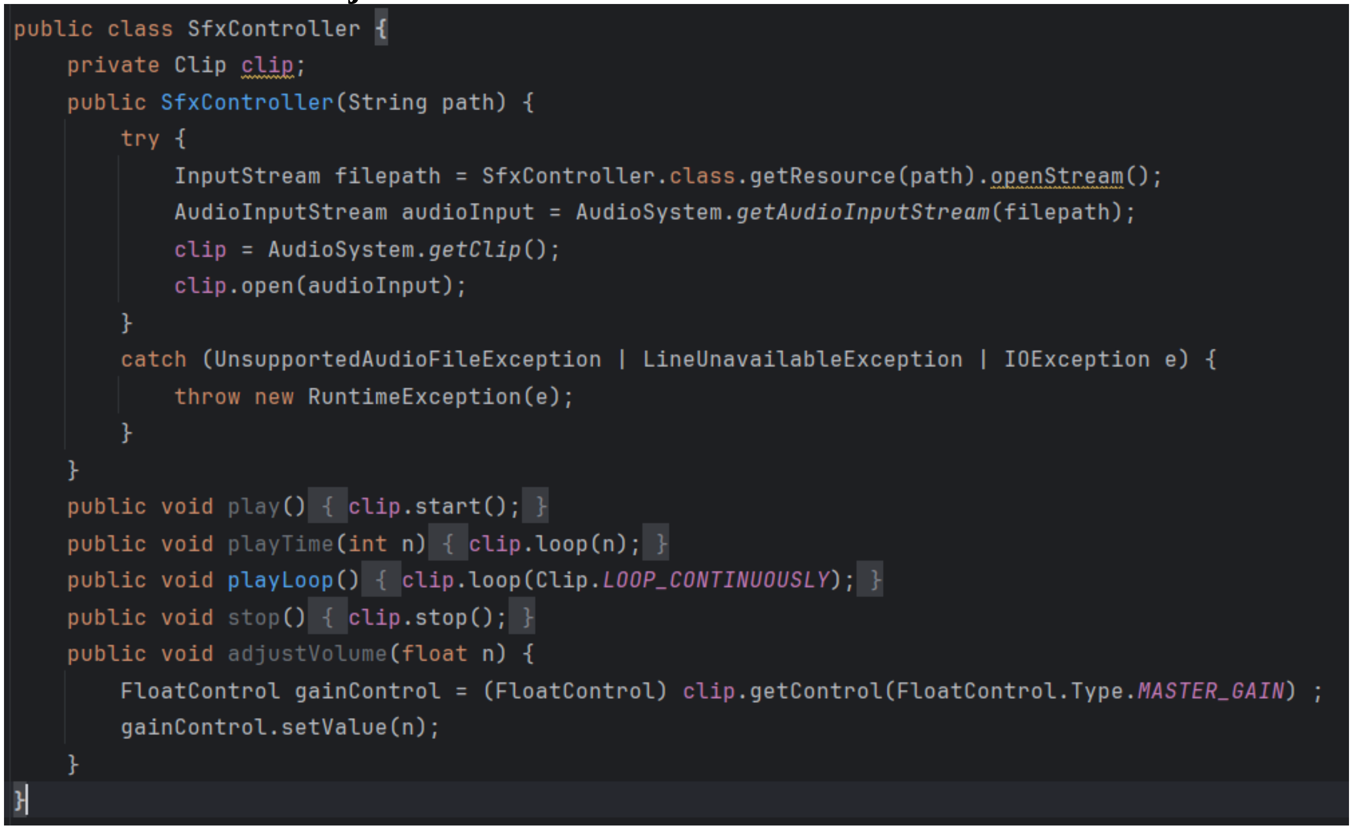
- This method handles navigating through instruction content images. It defines an array of image paths and checks the current image displayed in *instructionContent*. If the current image matches one in the array, it sets the next image. If the end of the array is reached, it resets to the first image.



```java
//--Switch Scene--
public void switchtoMenu(MouseEvent event) throws IOException {
    MainScene menu = new MainScene( path: "fxml/menu.fxml");
    stage.setScene(menu.loadScene());
    stage.show();
}

public void switchtoInstruction(MouseEvent event) throws IOException {
    MainScene instruction = new MainScene( path: "fxml/instruction.fxml");
    stage.setScene(instruction.loadScene());
    stage.show();
}
public void switchtoAboutUs(MouseEvent event) throws IOException {
    MainScene instruction = new MainScene( path: "fxml/aboutus.fxml");
    stage.setScene(instruction.loadScene());
    stage.show();
}
public void exit() { stage.close(); }
```

- These methods change the current scene based on user actions (clicking buttons). Each method creates a new MainScene object with the path to the desired FXML file, sets it as the current scene, and displays it.

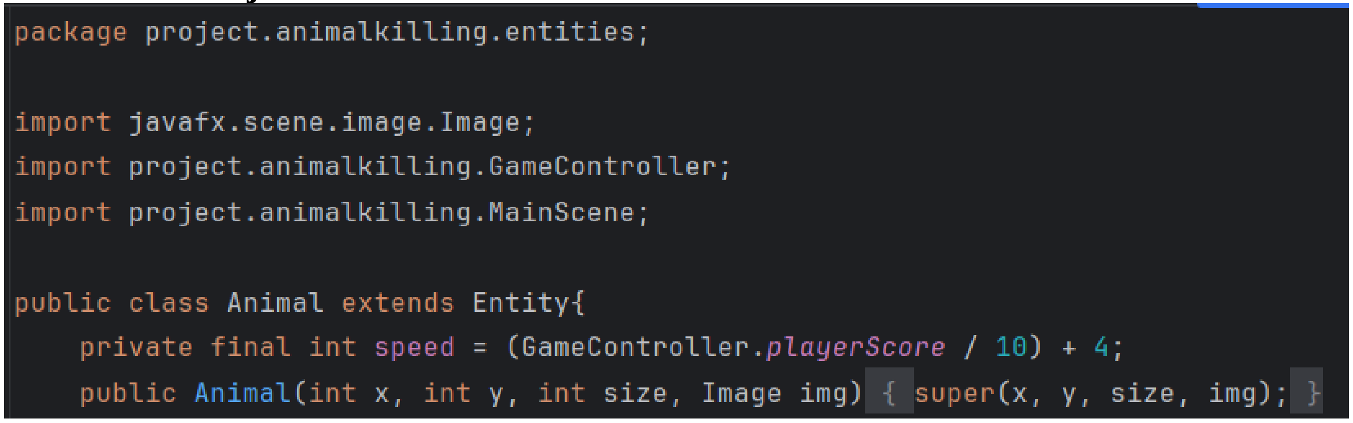
3.2.7. SfxController.java:



```java
public class SfxController {
    private Clip clip;
    public SfxController(String path) {
        try {
            InputStream filepath = SfxController.class.getResource(path).openStream();
            AudioInputStream audioInput = AudioSystem.getAudioInputStream(filepath);
            clip = AudioSystem.getClip();
            clip.open(audioInput);
        }
        catch (UnsupportedAudioFileException | LineUnavailableException | IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void play() { clip.start(); }
    public void playTime(int n) { clip.loop(n); }
    public void playLoop() { clip.loop(Clip.LOOP_CONTINUOUSLY); }
    public void stop() { clip.stop(); }
    public void adjustVolume(float n) {
        FloatControl gainControl = (FloatControl) clip.getControl(FloatControl.Type.MASTER_GAIN) ;
        gainControl.setValue(n);
    }
}
```

- The SfxController class provides a structured way to manage sound effects in a Java application. It allows for loading audio files, playing them, looping, stopping playback, and adjusting volume.

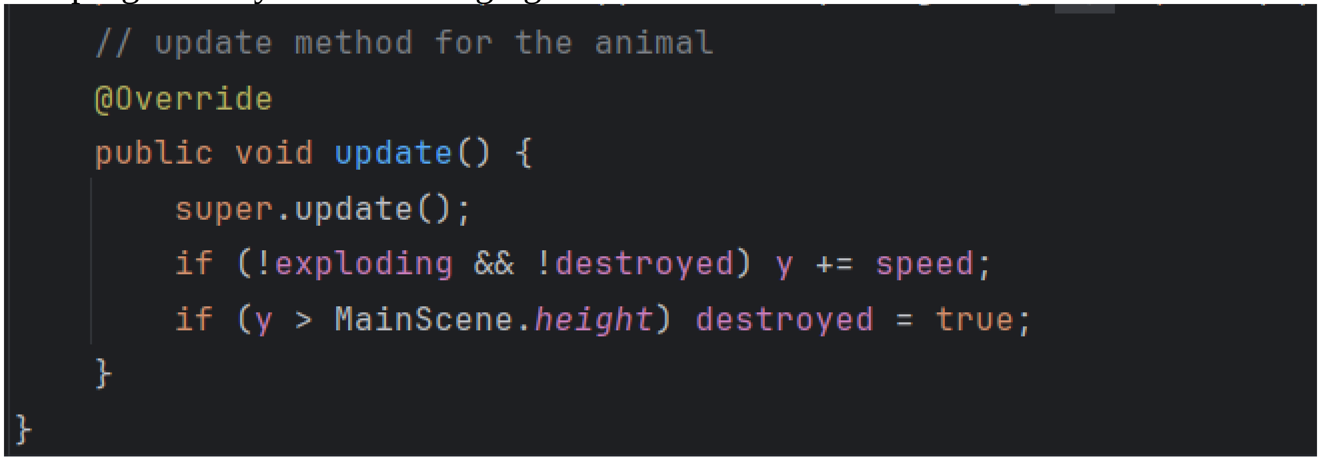
3.2.8. Animal.java:



```java
package project.animalkilling.entities;

import javafx.scene.image.Image;
import project.animalkilling.GameController;
import project.animalkilling.MainScene;

public class Animal extends Entity{
    private final int speed = (GameController.playerScore / 10) + 4;
    public Animal(int x, int y, int size, Image img) { super(x, y, size, img); }
```

- The Animal class extends to the Entity class, inheriting its properties and methods. This suggests that Animal is a type of game entity, possibly representing creatures in the game.
- The speed variable is defined as a final integer that determines how fast the animal moves down the screen. It is calculated based on the player's score from the

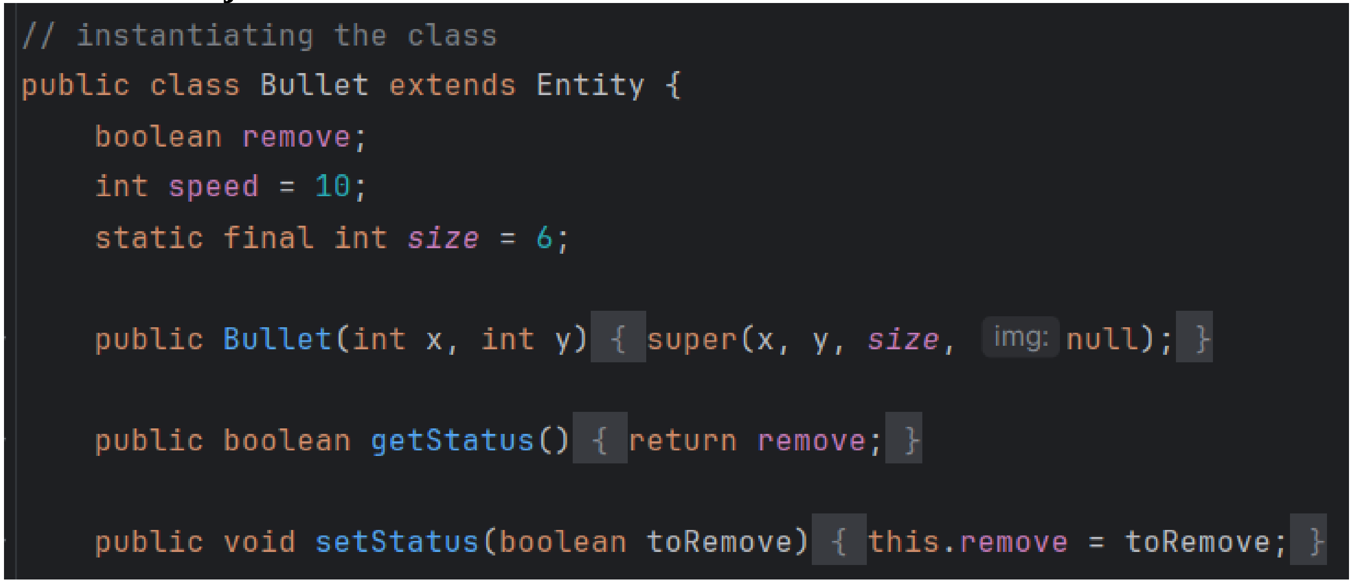
GameController: the formula *(GameController.playerScore / 10) + 4* means that as the player's score increases, the animal's speed also increases, making the game progressively more challenging.



```java
    // update method for the animal
    @Override
    public void update() {
        super.update();
        if (!exploding && !destroyed) y += speed;
        if (y > MainScene.height) destroyed = true;
    }
}
```

- The update method is overridden to define specific behavior for the Animal class:
- Call Super Update: It first invokes the update method of the superclass (Entity), which may handle general updates or state changes.
- Movement Logic: If the animal is not exploding and not destroyed, it moves down the screen by adding speed to its y coordinate.
- Destruction Check: If the y position of the animal exceeds the height of the MainScene, the animal is marked as destroyed. This likely means it has left the visible game area and should be removed or handled accordingly.

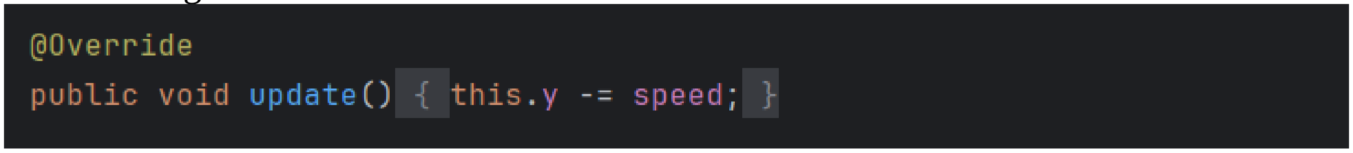
3.2.9. Bullet.java:



```java
// instantiating the class
public class Bullet extends Entity {
    boolean remove;
    int speed = 10;
    static final int size = 6;

    public Bullet(int x, int y) { super(x, y, size,  img: null); }

    public boolean getStatus() { return remove; }

    public void setStatus(boolean toRemove) { this.remove = toRemove; }
```

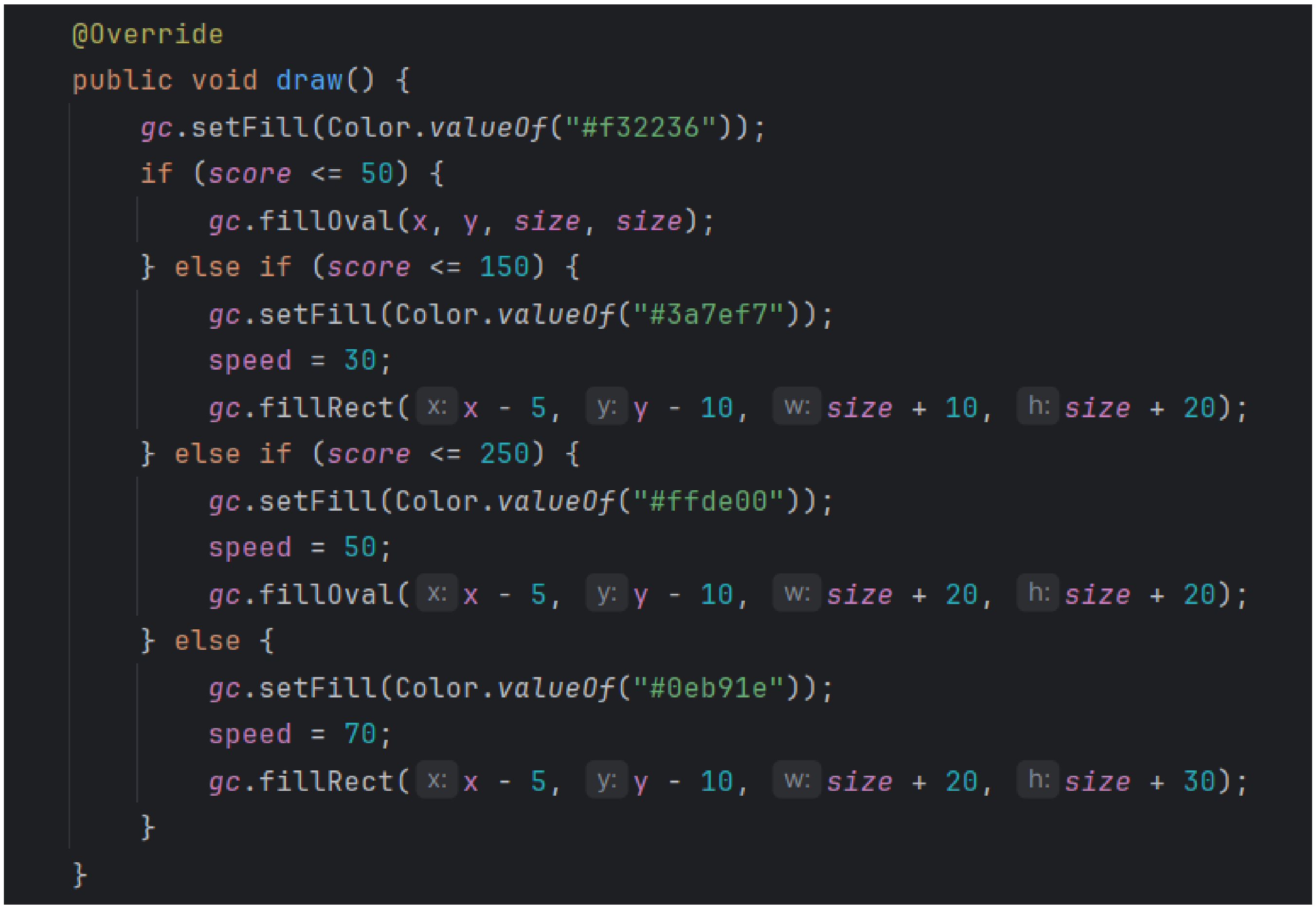
- remove: A boolean that indicates whether the bullet should be removed from the game.
- speed: An integer representing the speed at which the bullet moves upward (default

is 10).
- size: A constant that denotes the size of the bullet, set to 6 pixels.
- *getStatus():* Returns the status of the remove variable, indicating whether the bullet should be removed from the game.
- *setStatus(boolean toRemove):* Sets the remove variable based on the parameter, allowing external control over whether the bullet is marked for removal.



```java
@Override
public void update() { this.y -= speed; }
```

- The update method is overridden to define specific behavior for the Bullet class. It decreases the y position of the bullet by its speed, causing it to move upward on the screen.

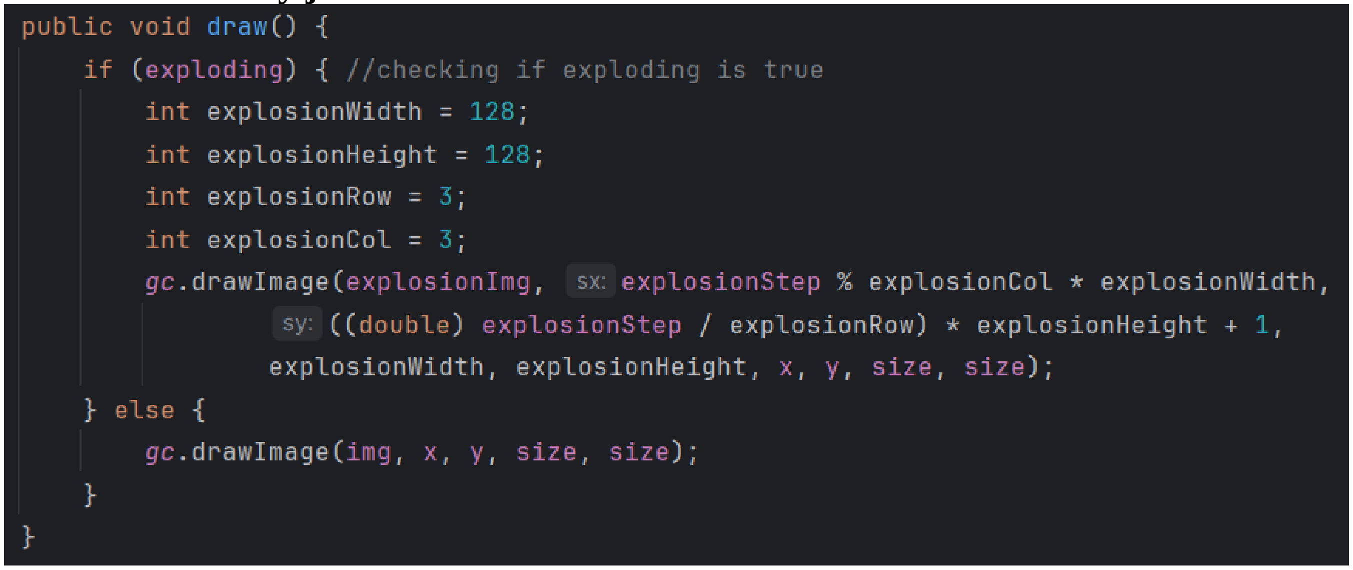


```java
@Override
public void draw() {
    gc.setFill(Color.valueOf("#f32236"));
    if (score <= 50) {
        gc.fillOval(x, y, size, size);
    } else if (score <= 150) {
        gc.setFill(Color.valueOf("#3a7ef7"));
        speed = 30;
        gc.fillRect( x: x - 5,  y: y - 10,  w: size + 10,  h: size + 20);
    } else if (score <= 250) {
        gc.setFill(Color.valueOf("#ffde00"));
        speed = 50;
        gc.fillOval( x: x - 5,  y: y - 10,  w: size + 20,  h: size + 20);
    } else {
        gc.setFill(Color.valueOf("#0eb91e"));
        speed = 70;
        gc.fillRect( x: x - 5,  y: y - 10,  w: size + 20,  h: size + 30);
    }
}
```

- The *draw()* method is overridden to render the bullet on the screen. The bullet's color and shape change based on the player's score:
  - Score ≤ 50: Draws a small red oval.
  - Score ≤ 150: Changes color to blue, increases speed to 30, and draws a larger **rectangle.**

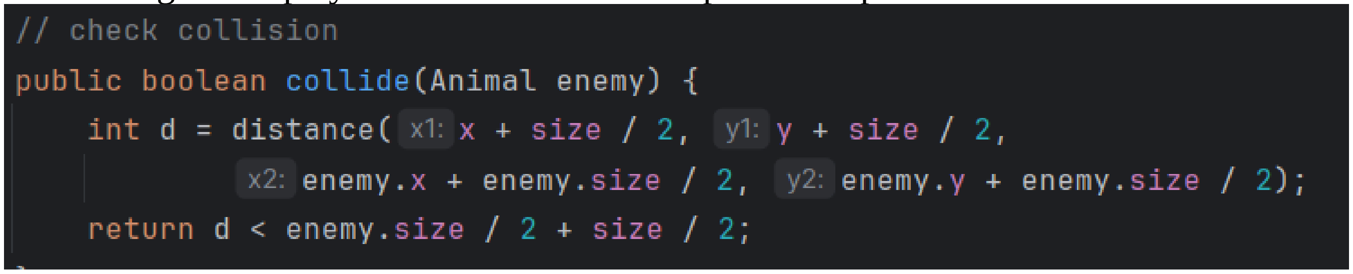
- Score ≤ 250: Changes color to yellow, increases speed to 50, and draws a larger oval.
- Score > 250: Changes color to green, increases speed to 70, and draws an even larger rectangle.

3.2.10. Entity.java:



```java
public void draw() {
    if (exploding) { //checking if exploding is true
        int explosionWidth = 128;
        int explosionHeight = 128;
        int explosionRow = 3;
        int explosionCol = 3;
        gc.drawImage(explosionImg, sx: explosionStep % explosionCol * explosionWidth,
                sy: ((double) explosionStep / explosionRow) * explosionHeight + 1,
                explosionWidth, explosionHeight, x, y, size, size);
    } else {
        gc.drawImage(img, x, y, size, size);
    }
}
```

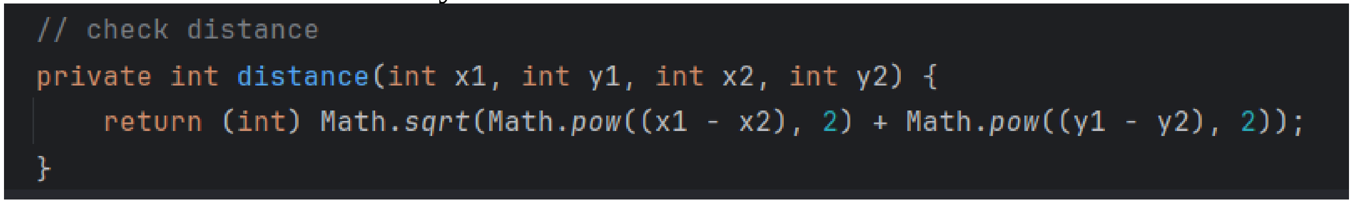
- The draw method is responsible for rendering the entity on the screen:
- If the entity is exploding, it draws the explosion animation based on the explosionStep.
- If not exploding, it simply draws the entity's image at its current position and size.
- The explosion animation involves calculating the correct portion of the explosion image to display based on the current explosionStep.



```java
// check collision
public boolean collide(Animal enemy) {
    int d = distance( x1: x + size / 2,  y1: y + size / 2,
            x2: enemy.x + enemy.size / 2,  y2: enemy.y + enemy.size / 2);
    return d < enemy.size / 2 + size / 2;
```

- This line calculates the distance d between the centers of the two entities (the current entity and the Animal). Center Calculation:
- For the current entity: *x + size / 2 and y + size / 2* compute the center coordinates based on its position (x, y) and its size.
- **For the Animal:** enemy.x + enemy.size / 2 and enemy.y + enemy.size / 2 **do the same for the Animal entity.**
- This line checks if the distance d is less than the sum of the radii of the two entities. If d is less than the combined radius, it indicates that the two entities are overlapping, which means a collision has occurred. Radius Calculation:

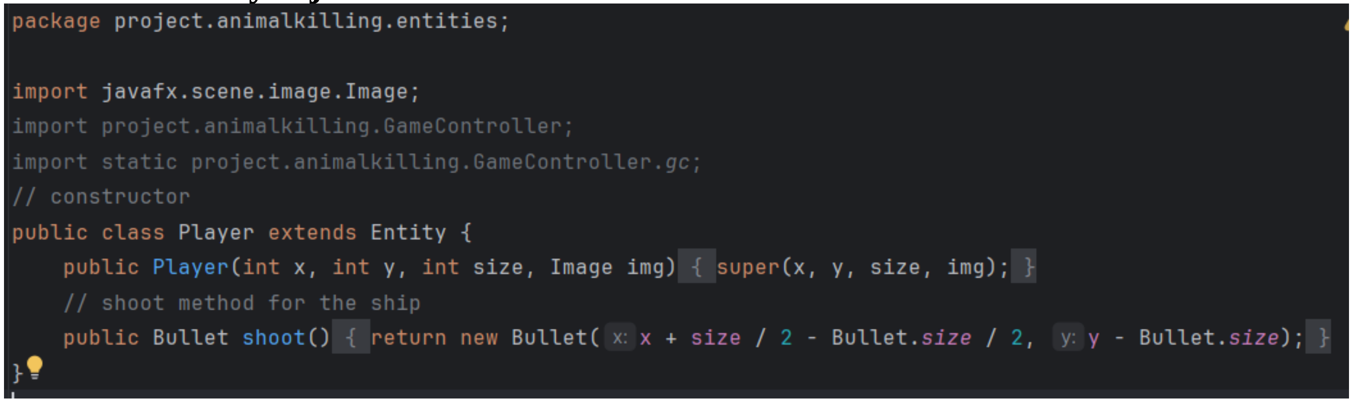
- For the Animal: *enemy.size / 2*.
- For the current entity: *size / 2*.



```
// check distance
private int distance(int x1, int y1, int x2, int y2) {
    return (int) Math.sqrt(Math.pow((x1 - x2), 2) + Math.pow((y1 - y2), 2));
}
```

- This private method computes the Euclidean distance between two points (x1, y1) and (x2, y2). It is used in the collide method to determine if two entities are within collision range.

3.2.11. Player.java:



```
package project.animalkilling.entities;

import javafx.scene.image.Image;
import project.animalkilling.GameController;
import static project.animalkilling.GameController.gc;
// constructor
public class Player extends Entity {
    public Player(int x, int y, int size, Image img) { super(x, y, size, img); }
    // shoot method for the ship
    public Bullet shoot() { return new Bullet( x: x + size / 2 - Bullet.size / 2,  y: y - Bullet.size); }
}
```

- The shoot method creates and returns a new Bullet object. This method defines the behavior when the player shoots:
- The bullet's initial x coordinate is calculated by centering it relative to the player’s position: *x + size / 2 - Bullet.size / 2*. This ensures the bullet starts at the center of the player.
- The bullet's initial y coordinate is set to *y - Bullet.size*, positioning it just above the player, where bullets typically originate.

3.3. UI Design

Creating a unique game design can be challenging, but an iterative process of experimenting with various artistic styles led to the decision to adopt a pixel-based approach. This shift from the original concept was facilitated by SceneBuilder and Aseprite. SceneBuilder, a visual layout tool for JavaFX applications, enables beginners to design user interfaces easily through drag-and-drop functionality, which generates FXML code. Complementing this, Aseprite, a pixel art editor for animated 2D sprites and game graphics, made it possible to bring the pixel-based concept to life. The game's main menu serves as the primary navigation interface, featuring a prominent title and four interactive buttons "Start," "Instruction," "About us," and "Exit." Each button highlights on mouse hover, subtly prompting players to confirm their selection.

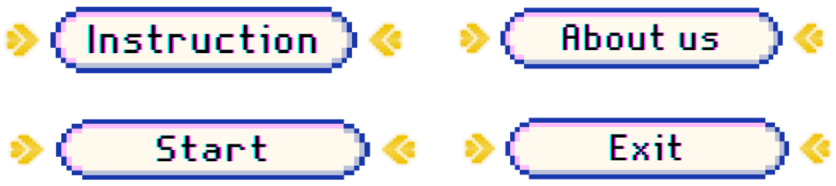


Figure 1: Click to play

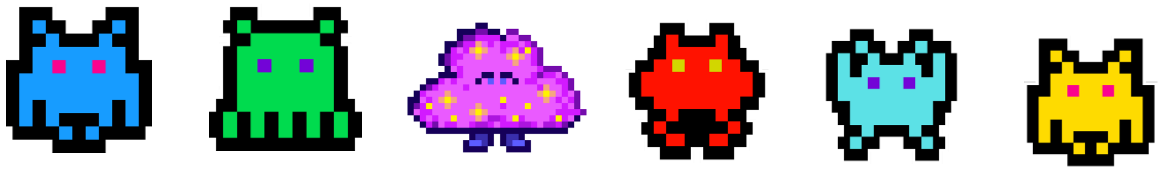


Figure 2: Animal

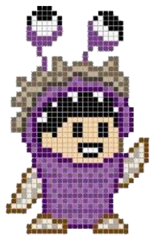
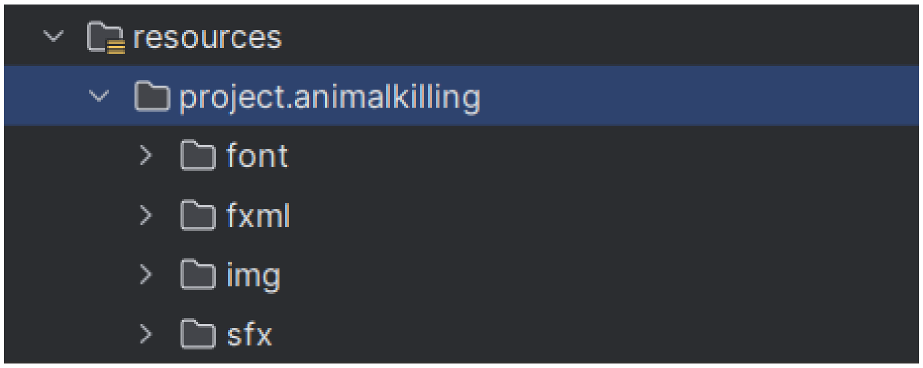


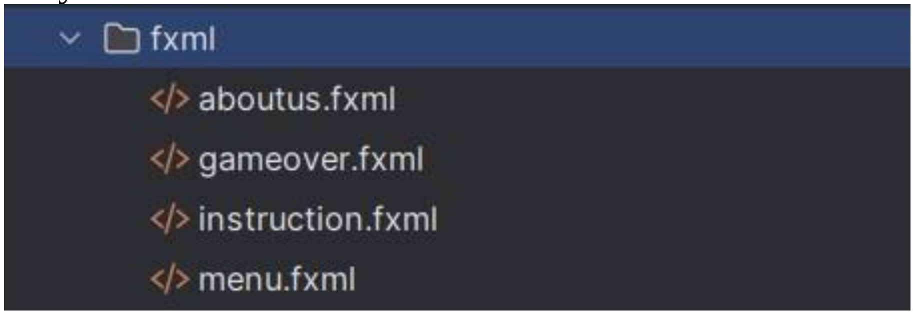
Figure 3: Player

Resources files will be stored in project.animalkilling folder. We can put them in different directories with diffrent categories:



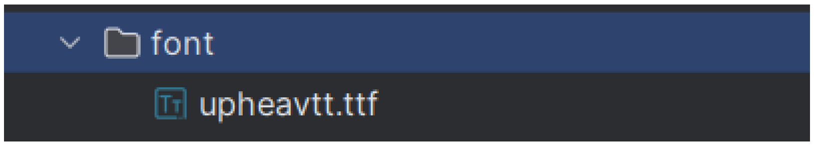
```
resources
  project.animalkilling
    font
    fxml
    img
    sfx
```

- **fxml:** A layout defines the structure for a user interface in your app, such as in an activity.



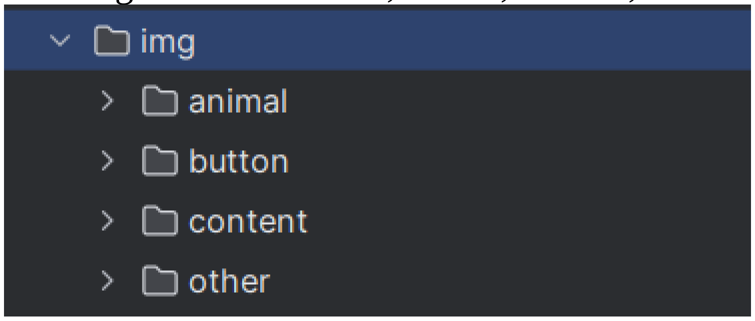
fxml
aboutus.fxml
gameover.fxml
instruction.fxml
menu.fxml

- **font:** Some fonts and file xml fontfamily which declairs our addition fonts for later use.



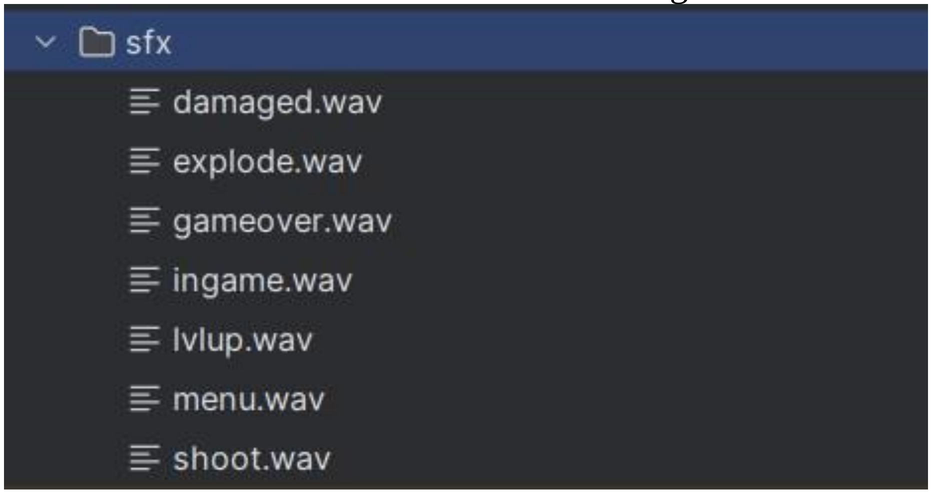
font
upheavtt.ttf

- **img:** store of the images for UI: animal, button, content, other.



img
animal
button
content
other

- **sfx:** contains other raw files about the sound for the game.



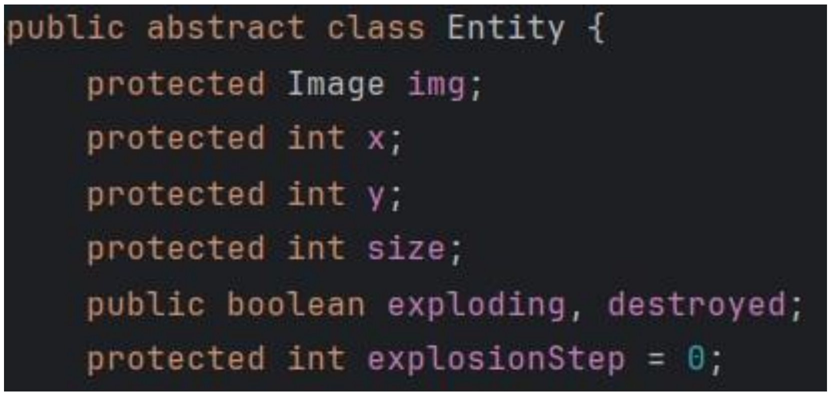
sfx
damaged.wav
explode.wav
gameover.wav
ingame.wav
lvlup.wav
menu.wav
shoot.wav

3.4. OOP concepts implementation

This segment dissects the intricate implementation of the four principal concepts of object-oriented programming including: Encapsulation, Inheritance, Abstraction, and Polymorphism. Additionally, it examines the implementation of the "Singleton" design pattern with reference to relevant libraries or frameworks if applicable.

3.4.1. Encapsulation

- Create classes that clearly separate data from methods.
- Use access control methods to safeguard data integrity.
- Recognize the advantages of data hiding and abstraction.

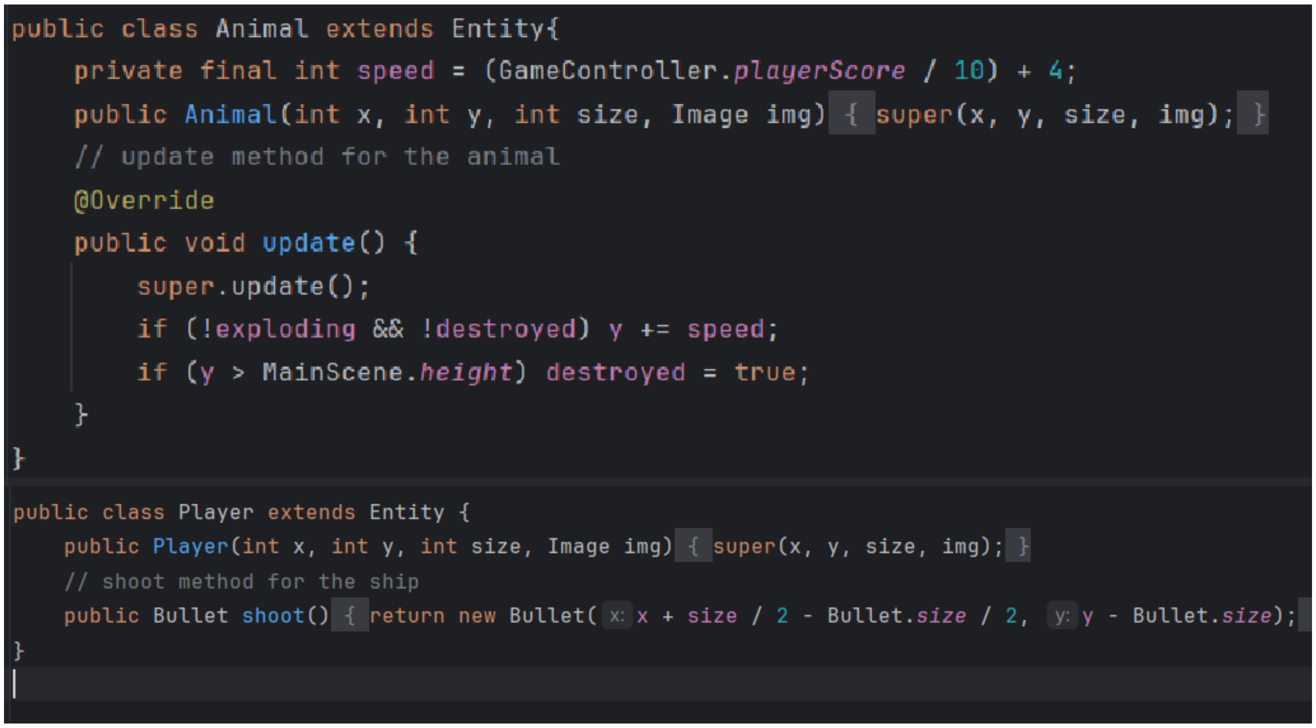


```
public abstract class Entity {
    protected Image img;
    protected int x;
    protected int y;
    protected int size;
    public boolean exploding, destroyed;
    protected int explosionStep = 0;
```

By employing access modifiers such as private, protected, or public for class fields, we can ensure encapsulation. This encapsulation prevents unauthorized access from other classes, thereby enhancing the security and integrity of the data within the class. This practice is a fundamental aspect of object-oriented programming and contributes to the robustness and maintainability of the code. In the Figure 7, the data are declared in Entity class and have their accessed levels set to protected and public. By doing this, we ensure that only some data is allowed to access from outside of the class, others are restricted to only classes that are child of the Entity class.

3.4.2. Inheritance

- Develop class hierarchies that reflect the relationships among various entities in the game world.
- Leverage inheritance to reuse code, enhancing maintainability.
- Understand the principles of superclasses, subclasses, and method overriding.



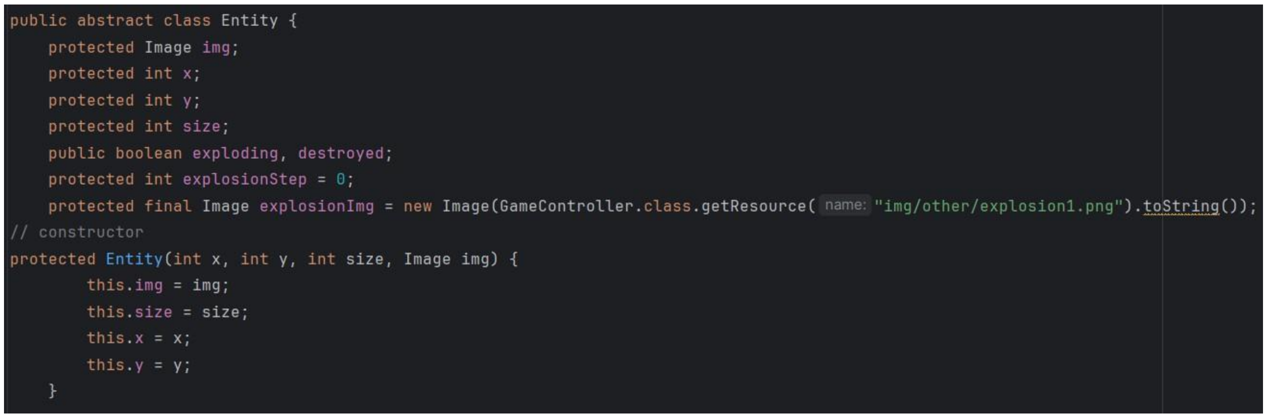
```
public class Animal extends Entity{
    private final int speed = (GameController.playerScore / 10) + 4;
    public Animal(int x, int y, int size, Image img) { super(x, y, size, img); }
    // update method for the animal
    @Override
    public void update() {
        super.update();
        if (!exploding && !destroyed) y += speed;
        if (y > MainScene.height) destroyed = true;
    }
}
```

```
public class Player extends Entity {
    public Player(int x, int y, int size, Image img) { super(x, y, size, img); }
    // shoot method for the ship
    public Bullet shoot() { return new Bullet( x: x + size / 2 - Bullet.size / 2,  y: y - Bullet.size);
}
```

Inheritance is a concept where new classes share the structure and behavior defined in other classes. These classes can also be modified by adding new or overriding methods and fields from their parent. Inheritance represents the IS-A relationship which is also known as a parent-child relationship. Figure 8 is one of the usages from our project, Animal class extended from class Entity which indicates that Animal is the subclass and Entity is the super class. Hence, the relationship between the two classes is Animal IS-A Entity. Furthermore, class Player also extended from class Entity, therefore the relationship between Player, Animal and Entity is a hierarchical inheritance.

3.4.3. Abstraction

- Define clear interfaces for different game components, such as menu, instruction, instruction, and game over.
- Emphasize key characteristics and functionalities while concealing implementation details.
- Use abstraction to enhance code reusability and streamline program structure.

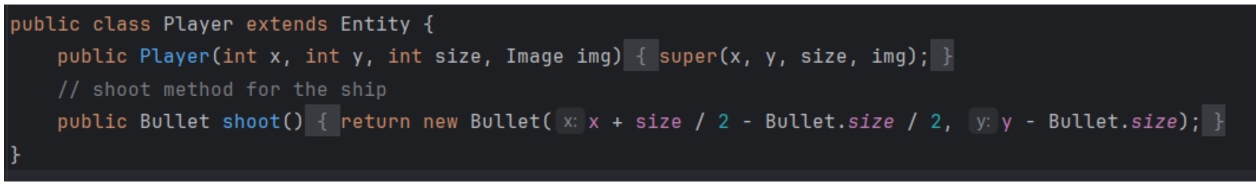


```java
public abstract class Entity {
    protected Image img;
    protected int x;
    protected int y;
    protected int size;
    public boolean exploding, destroyed;
    protected int explosionStep = 0;
    protected final Image explosionImg = new Image(GameController.class.getResource( name: "img/other/explosion1.png").toString());
// constructor
protected Entity(int x, int y, int size, Image img) {
        this.img = img;
        this.size = size;
        this.x = x;
        this.y = y;
    }
```

An abstract class is a class which cannot be instantiated; therefore the creation of an object is not possible with an abstract class, but it can be inherited. It is usually designed to serve as a blueprint for other classes. Java allows an abstract class to have both the regular methods and abstract methods (methods with empty body). This class outlines a common structure and mandates the implementation of certain methods in its subclasses. It ensures that all subclasses maintain a consistent interface while allowing for individualized behavior. Figure 9 shows the abstract Entity which was used as a blueprint for other classes like Animal and Player, it contains all of the reusable methods and fields so that others do not have to instantiate again. Understand abstraction, help us to have a cleaner, more precise code structure.

3.4.4. Polymorphism

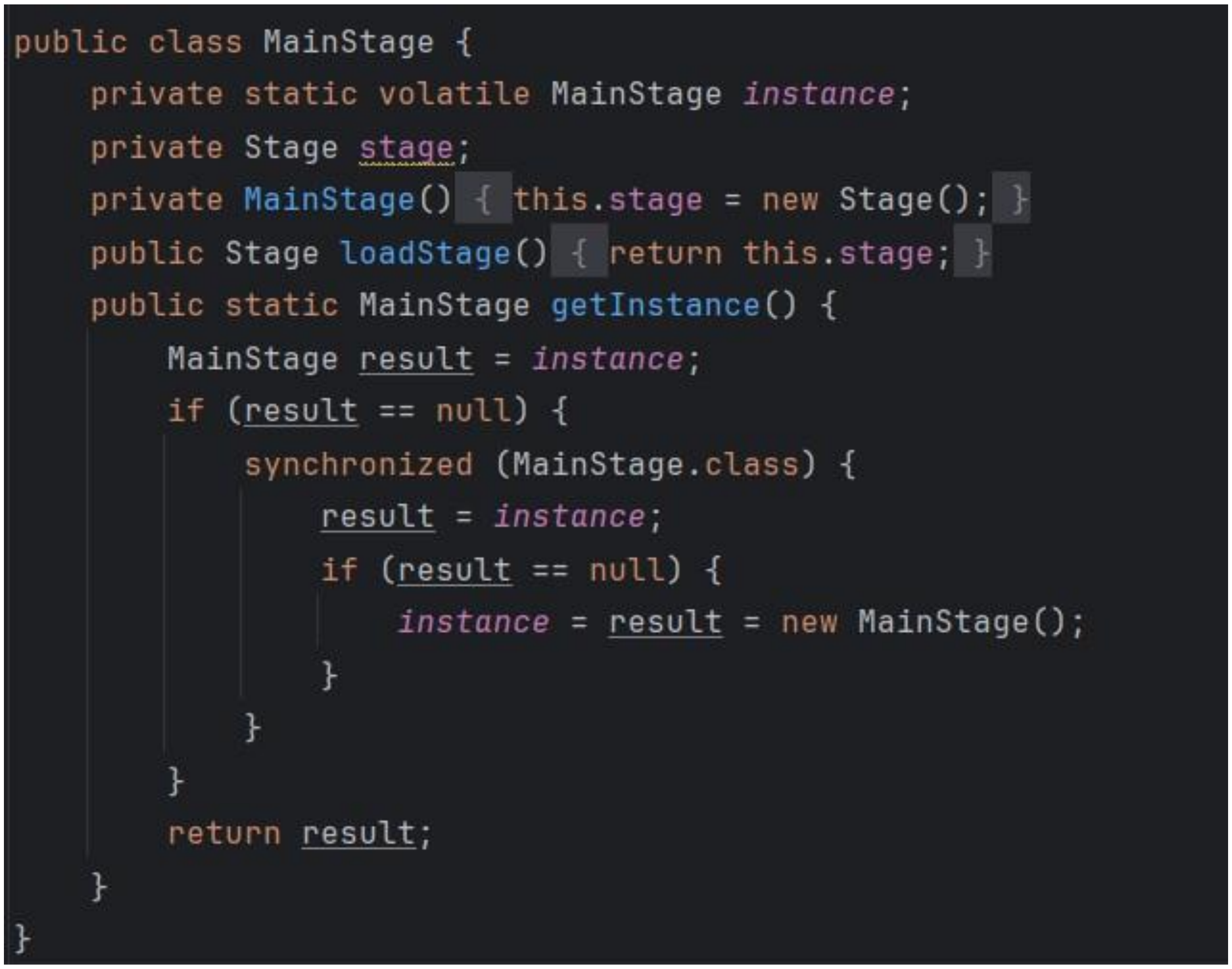
- Design methods that can handle different types of objects at runtime.
- Implement virtual methods and method overriding for dynamic behavior.
- Leverage polymorphism to create flexible and extensible game mechanics.



```java
public class Player extends Entity {
    public Player(int x, int y, int size, Image img) { super(x, y, size, img); }
    // shoot method for the ship
    public Bullet shoot() { return new Bullet( x: x + size / 2 - Bullet.size / 2,  y: y - Bullet.size); }
}
```

Polymorphism in Java is a concept by which we can perform a single action in different ways. In this sample, the constructor of the “Ship” class in Figure 9 is a constructor polymorphism. It calls the constructor of the superclass “Entity” with its parameters. This allows “Ship” to be initialized with their specific attributes while still being an “Entity”.

3.4.5. Design pattern



```
public class MainStage {
    private static volatile MainStage instance;
    private Stage stage;
    private MainStage() { this.stage = new Stage(); }
    public Stage loadStage() { return this.stage; }
    public static MainStage getInstance() {
        MainStage result = instance;
        if (result == null) {
            synchronized (MainStage.class) {
                result = instance;
                if (result == null) {
                    instance = result = new MainStage();
                }
            }
        }
        return result;
    }
}
```

Singleton is one of the 5 design patterns from the Creational Design Pattern group. Singleton will ensure that at any point of time there is only one instance of its kind exits and provide a single point of access to it from any other parts of the application. There are many ways to implement singleton, but in this project we used the Double Check Locking Singleton. Figure 10 shows the implementation of the approach for the "MainStage" class. It ensures that only one instance of a particular class exists during the runtime of an application. The "getInstance()" method is provided to check whether an instance of Stage exists or not, if not then it creates a new one, if it does then it returns the instance of that Stage. The "loadStage()" method is then called on the singleton instance of "MainStage". This method appears to be responsible for loading the instance of the Stage.

CHAPTER 4: FINAL APP GAME

4.1. Source code (link GitHub)

https://github.com/Selena166/Animal-Killing

4.2. Demo video

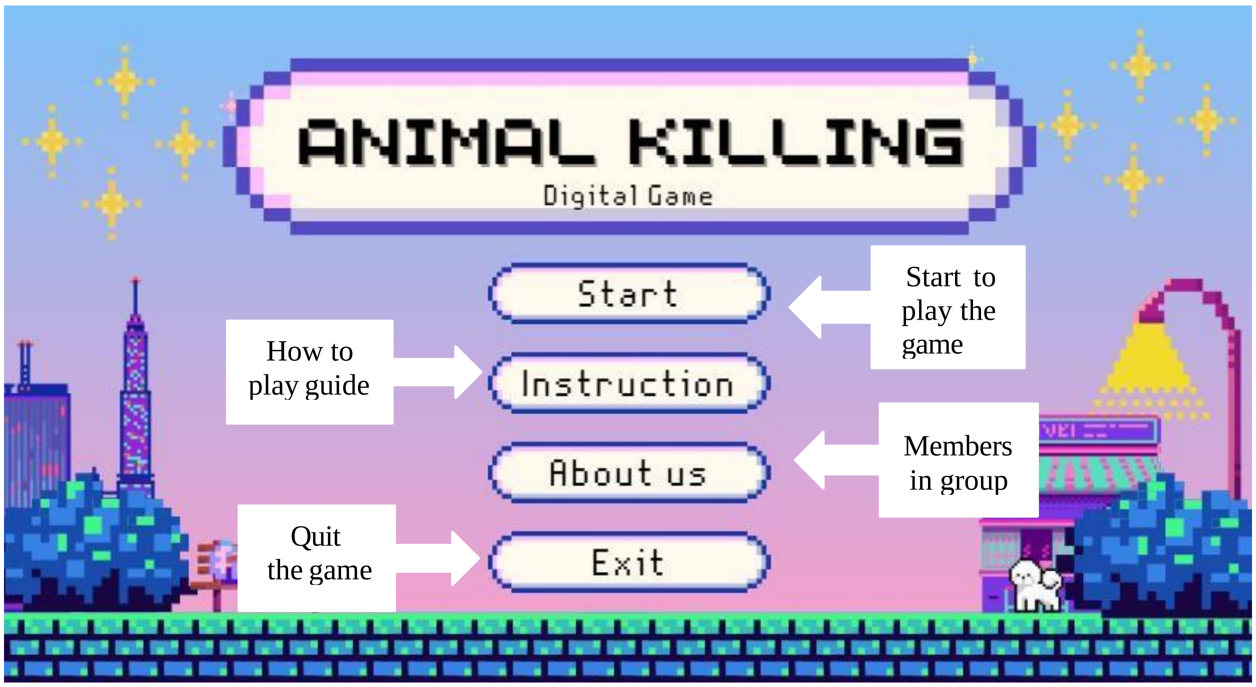
https://youtu.be/JHqTWL32Gl4

4.3. Instruction

4.3.1. Begin the game

Tap the icon to launch the game. A loading screen will be displayed.

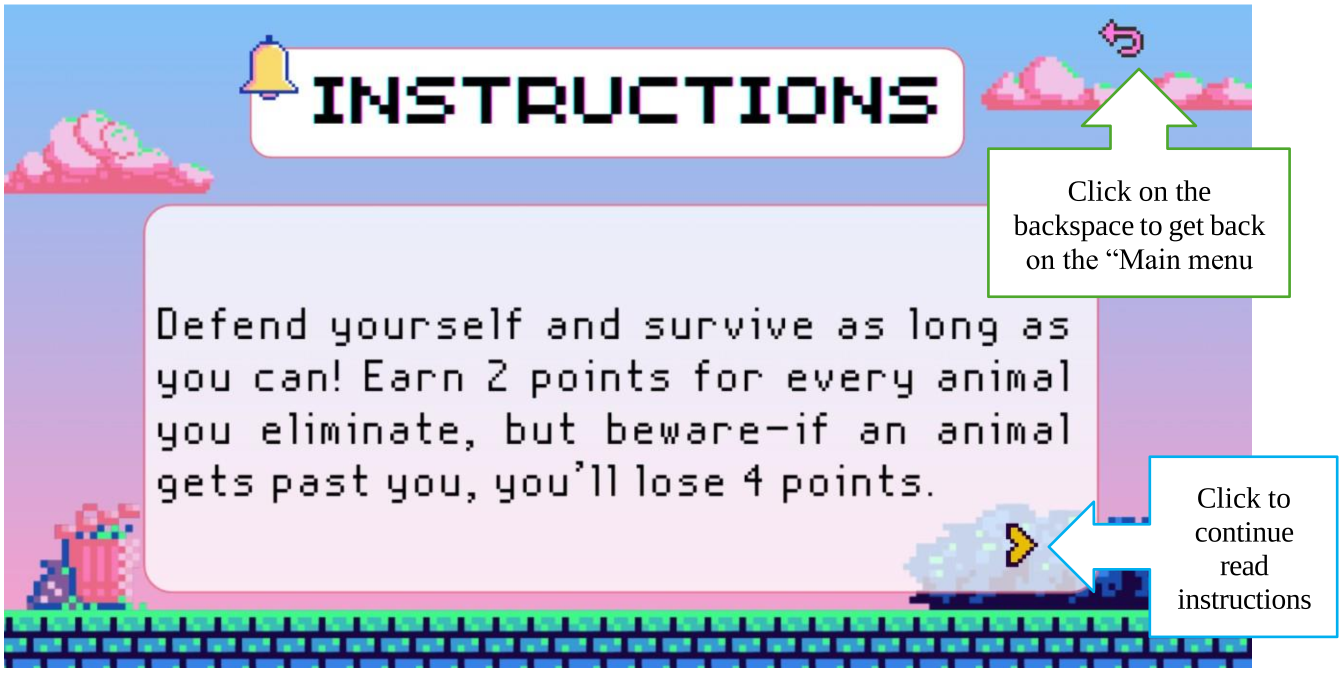
4.3.2. Main menu



Clicking the "Start" button triggers a transition to the gameplay screen while simultaneously generating ten randomly selected "bird" objects positioned above the player.

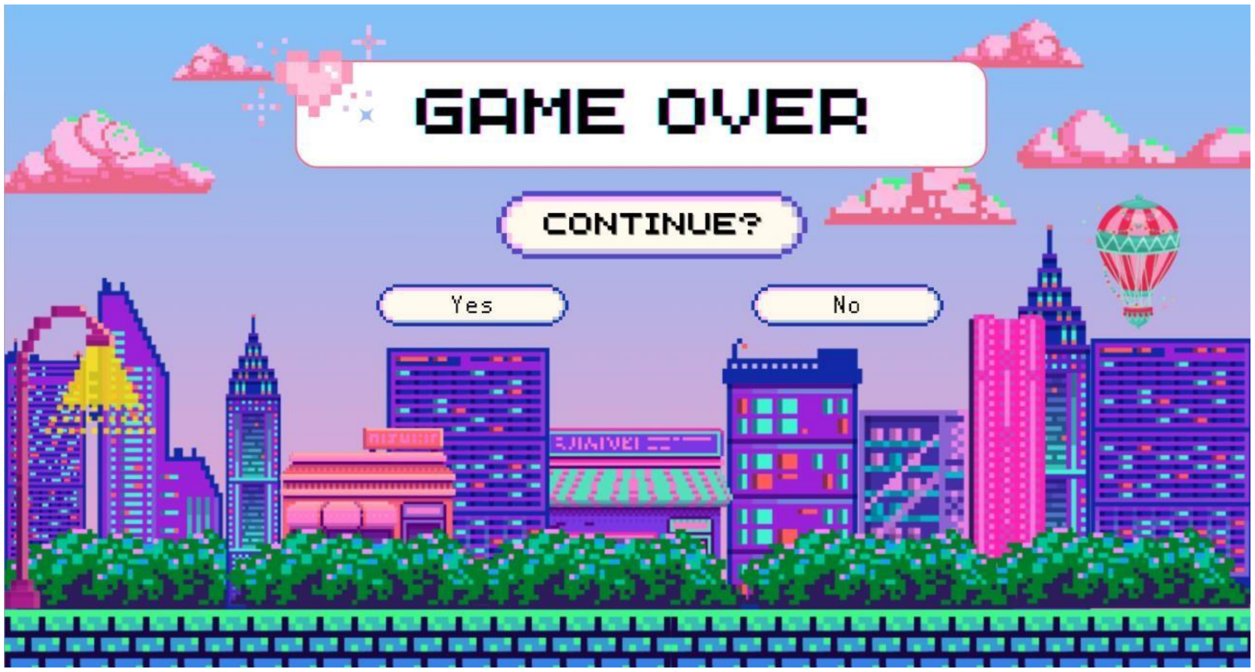
4.3.3. How to play

Show user the rules of the game:


INSTRUCTIONS
Click on the backspace to get back on the “Main menu
Click to continue read instructions

Defend yourself and survive as long as you can! Earn 2 points for every animal you eliminate, but beware-if an animal gets past you, you'll lose 4 points.

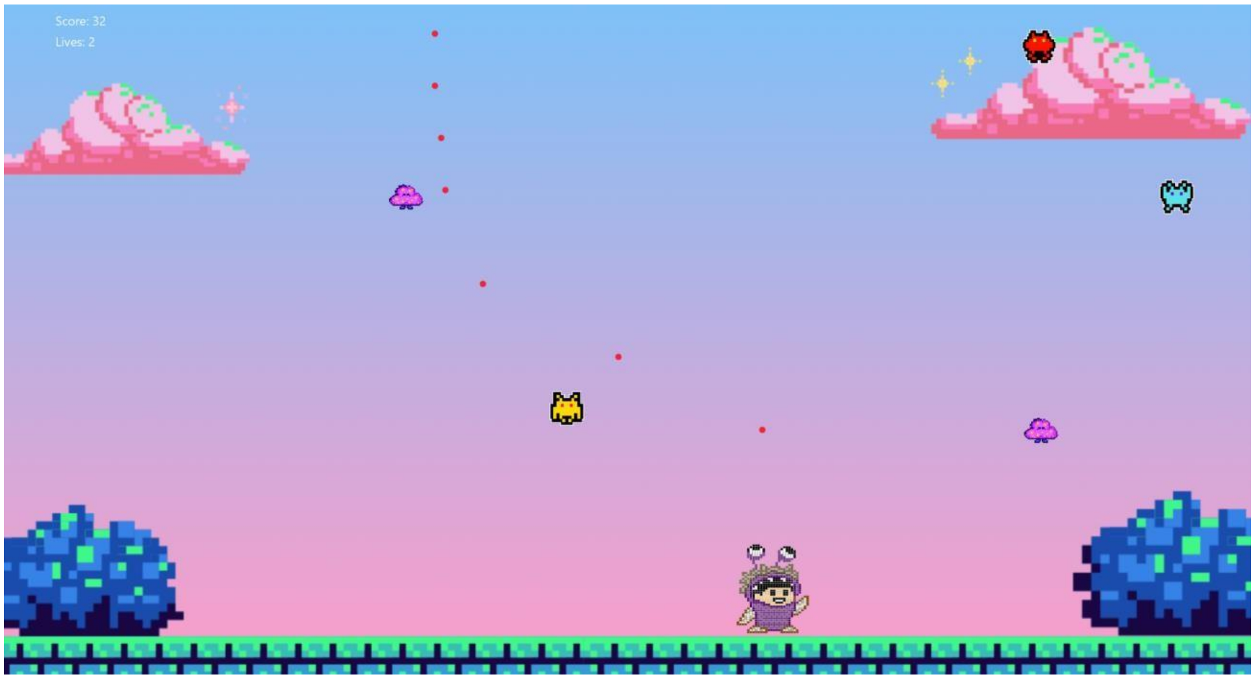
4.4.4. Game over



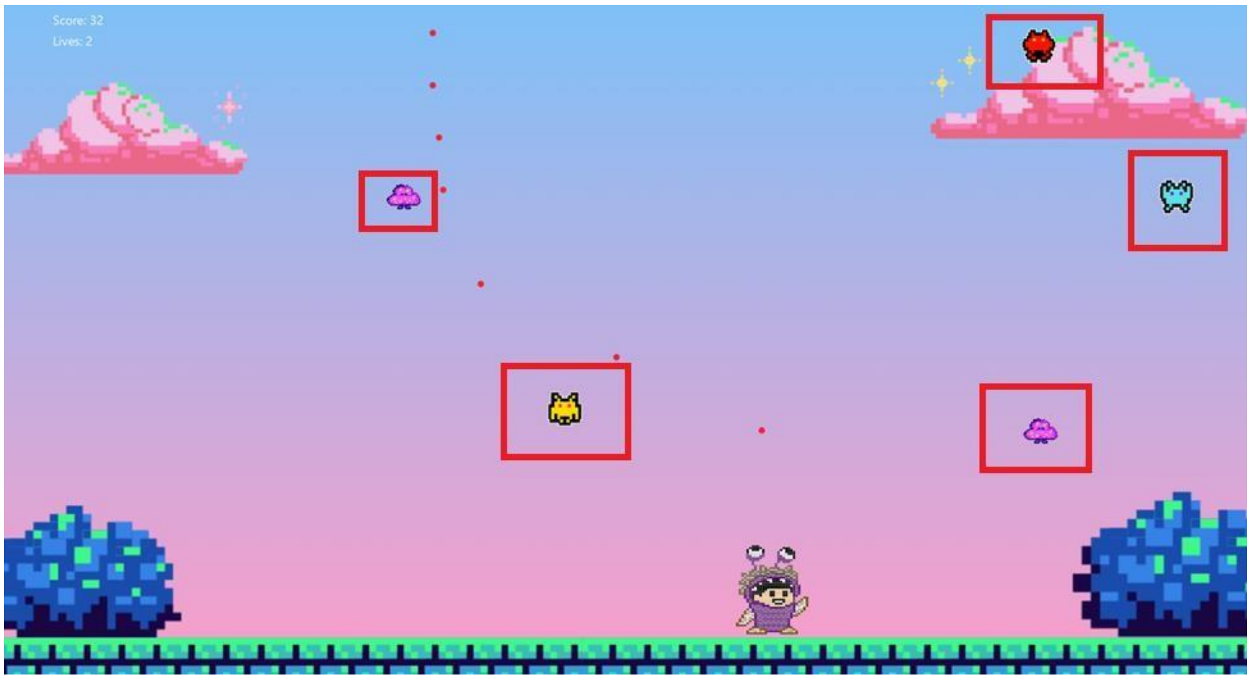
When the player is defeated, the "game over" menu appears. It presents two choices: "YES" to continue playing or "NO" to return to the main menu. These options are highlighted when hovered over.

4.4.5. Let’s play

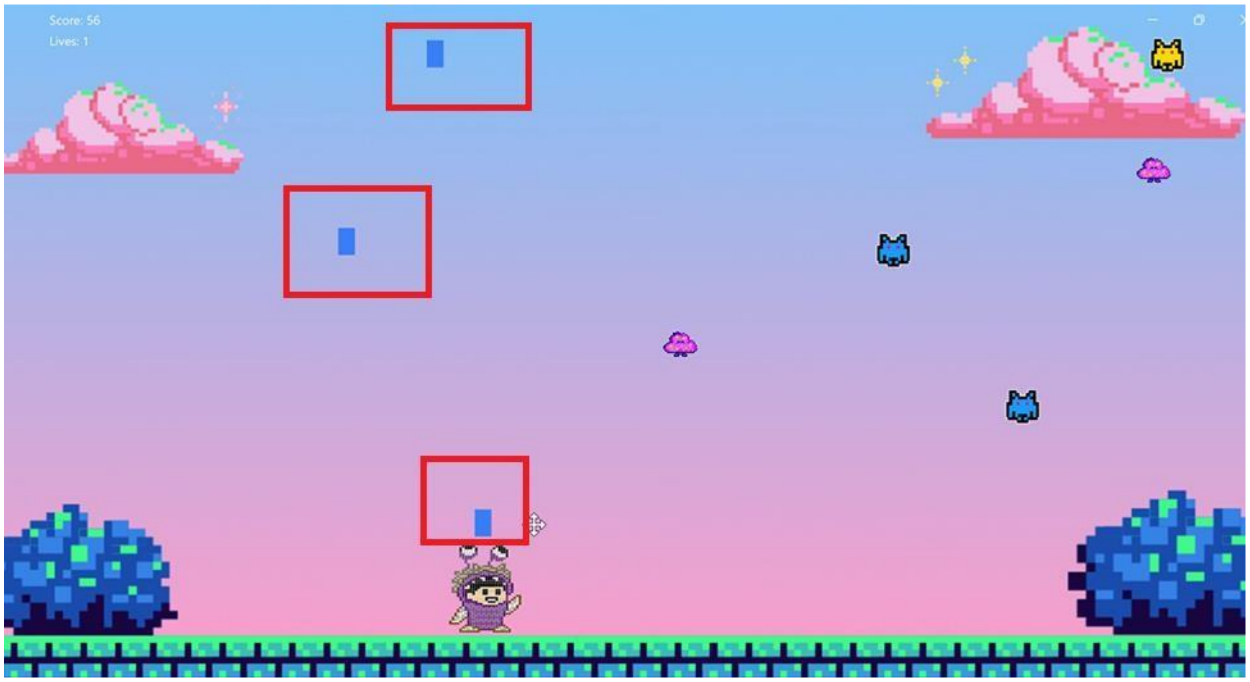
- Start the game



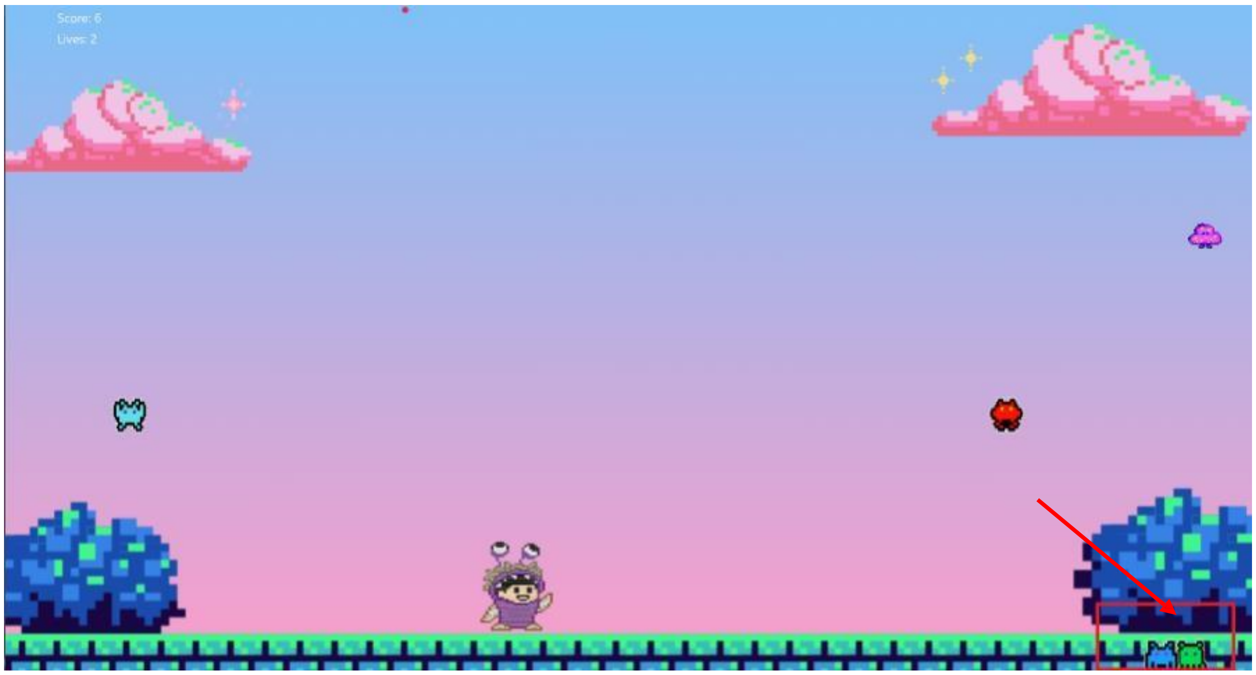
- Defeat one animal to gain two points



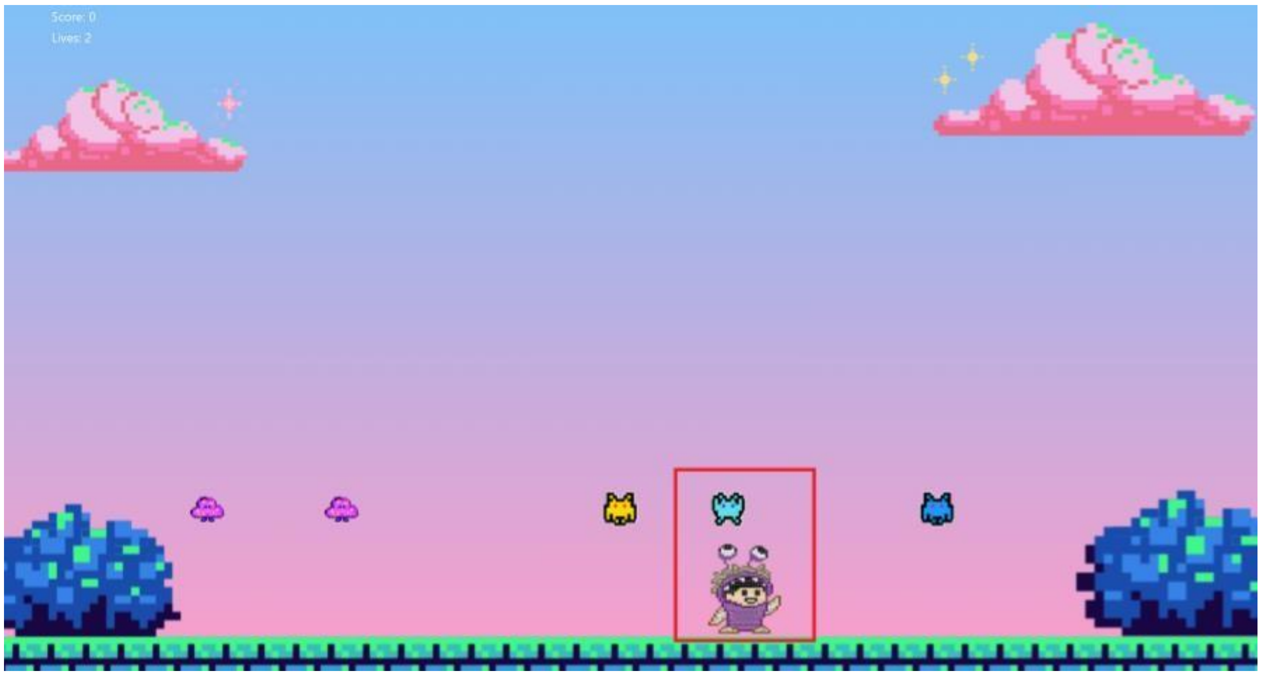
- If you score above 50 your bullets will be upgraded



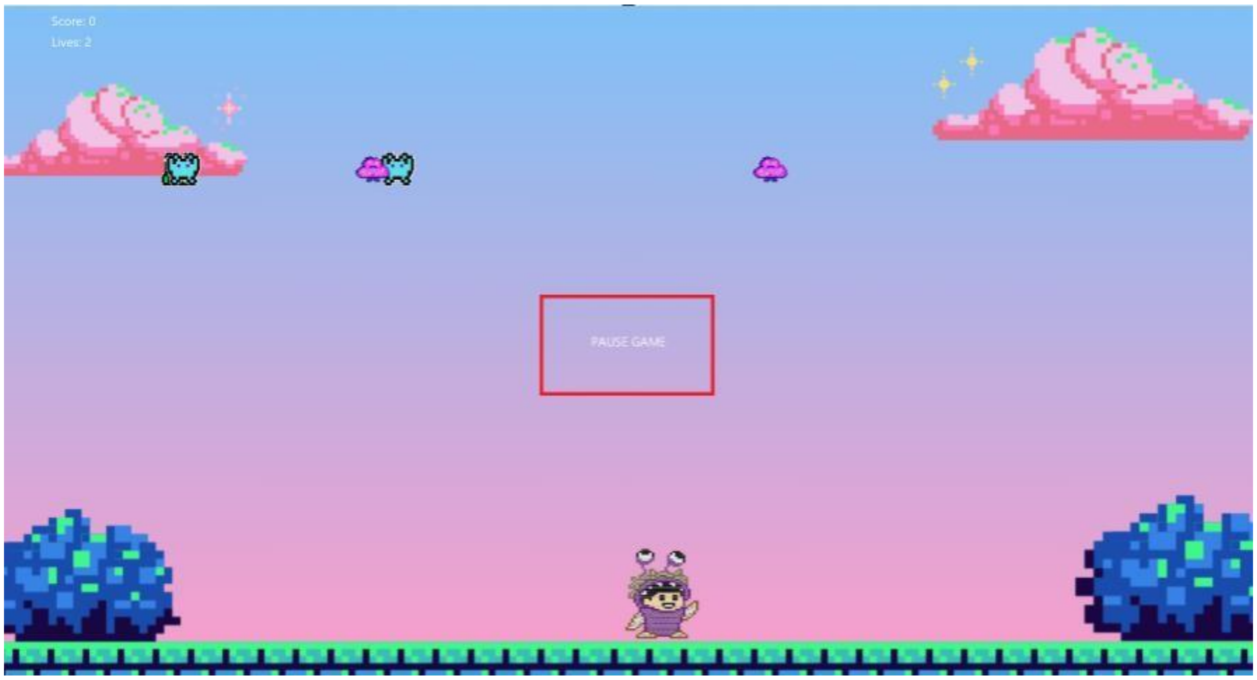
- If an animal passes you, you will lose four points



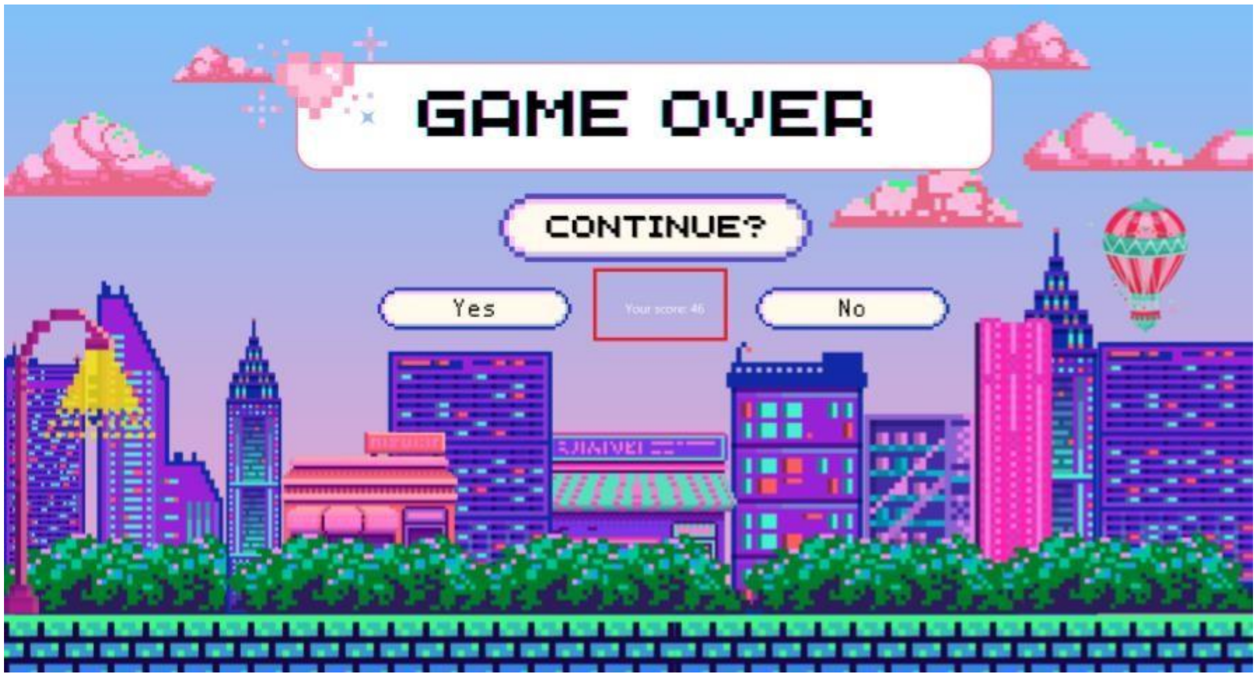
- If animal touch player, you could lose one lives.



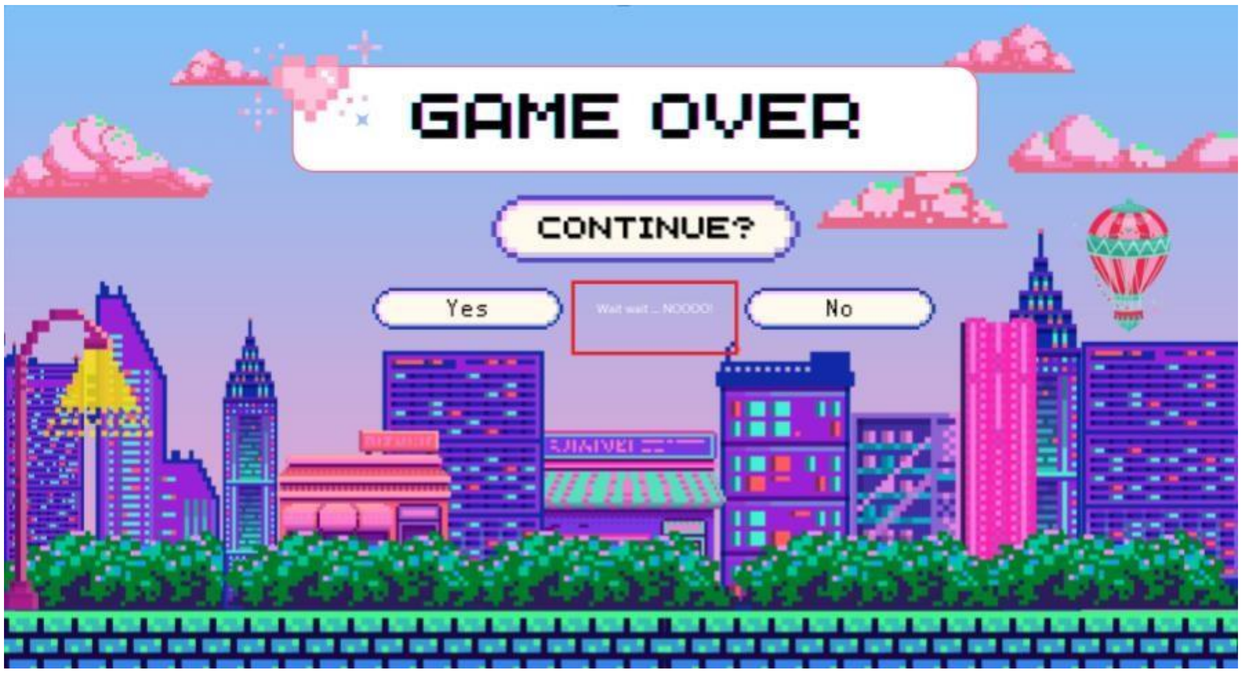
- If you want to pause the game, you enter ESC in keyboards to pause.



- If you lost 2 lives, screen would appear “game over” and display your score. Pressing “Yes” to continuedly play and “No” to back main menu



- In case you have no points the screen will say “wait wait... NOOOO!”


GAME OVER
CONTINUE?
Yes
Wait wait ... NOOOO!
No

CHAPTER 5: CONCLUSION AND EXPERIENCE

5.1. Accomplishment

- **Effective Use of OOP Principles**: We designed and implemented a variety of classes to represent game entities such as Entity, Monsters, and Bullet Shot. This approach enabled us to encapsulate related data and behaviors within distinct objects, effectively applying core Object-Oriented Programming (OOP) principles, including encapsulation, abstraction, polymorphism, and inheritance.
- **User Interface Design**: Using Java's GUI libraries, we created a visually engaging and interactive 2D-style user interface. This enhanced the game's overall appeal and usability, making it more enjoyable for players.
- **Game Logic Development**: Leveraging inheritance and polymorphism, we constructed a robust game logic system. Various types of entities were created as subclasses of a general Entity class, illustrating the OOP principle of inheritance while allowing seamless management and interaction among game components.

5.2. Difficult

- **Game Simplicity:** The game focuses on a few fundamental features and lacks advanced functionalities that could enhance its difficulty and increase its appeal to attract more players.
- **Scoring System Bug:** The scoring system had an intermittent issue that sometimes resulted in incorrect score calculations. Despite efforts, this bug could not be fully resolved within the project timeline.

5.3. Future works

- **Level Progression:** Introducing this feature would allow players to advance to a new level upon reaching a specific score, with the game's difficulty increasing as they progress.
- **Resurrection Mechanic:** Players can instantly revive during battles by collecting special items, allowing them to continue the game without leaving

the main screen.

- **Player Selection or Random Mode**: Introducing this feature would enable players to compete against an opponent, increasing the challenge as they race to defeat animals and earn points. The player with the highest score at the end of the game would be declared the winner.

5.4. Experience

In our OOP course, the project involved developing a game. After more than a month of collaboration, we gained valuable experiences from completing this game project. A successful game project requires numerous elements to be complete, including a storyline, gameplay mechanics, characters, and in-game environments, among others.

Beyond the valuable lessons learned in class, hands-on projects like this are essential for gaining practical experience and applying theoretical knowledge in real-world scenarios. Additionally, self-directed learning is crucial for IT students, as the ever-evolving nature of technology demands continuous updates and learning to keep pace with current advancements.

Lastly, we would like to express our heartfelt gratitude to our instructor for their dedication and guidance throughout the learning process. The materials and resources they provided have been an invaluable foundation for both our studies and the development of this project. We also deeply appreciate the efforts and contributions of each team member, whose hard work made it possible to achieve the best possible outcomes for our project.

We are excited to continue this journey, eager to gain even more insights and knowledge as we progress further in this field.

THE END.