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.Nguyen 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 come from International University, Information Technology major:

Table 1.5: Individual responsibility and contribution

Name - Github username	ID	Responsibility	Contribute
Nguyễn Hoàng Bảo	ITITSB22027	Write report	50%

Trân - *Btran2404*		Draw class diagram Design UI Design entity Implement JavaFx Fxml loader Research for references and technical documents Presentation slide	
Lê Nguyễn Thanh Trúc - *Selena166*	ITITWE22168	Write report Draw class diagram Create Github repository Create game controller Create SceneController Finalize and clean up code Evaluate and test for potential bugs Presentation slide	50%

Project's GitHub: Link

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

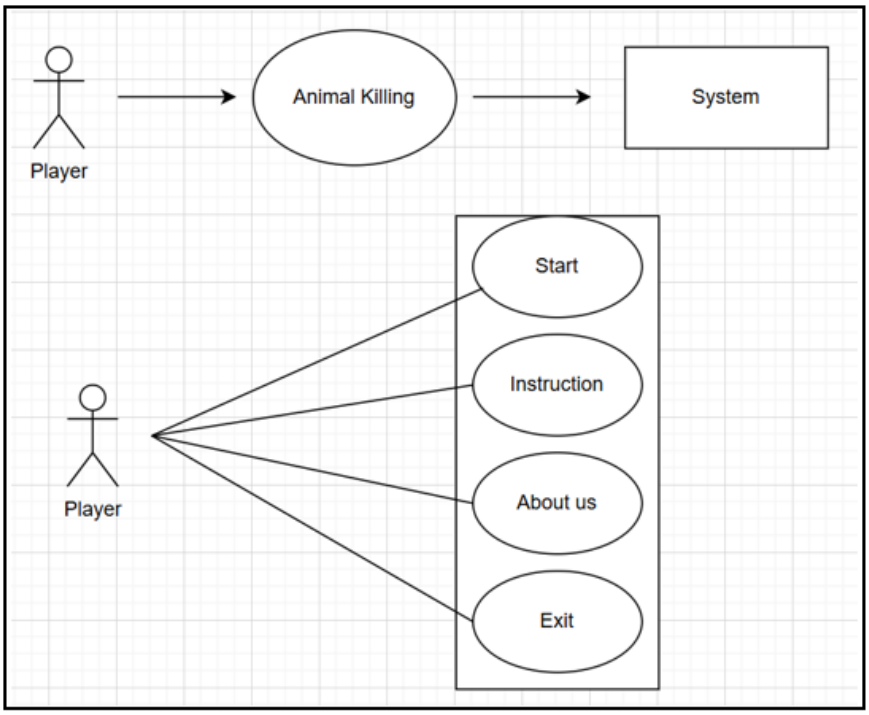
2.5. Use Case Scenario

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

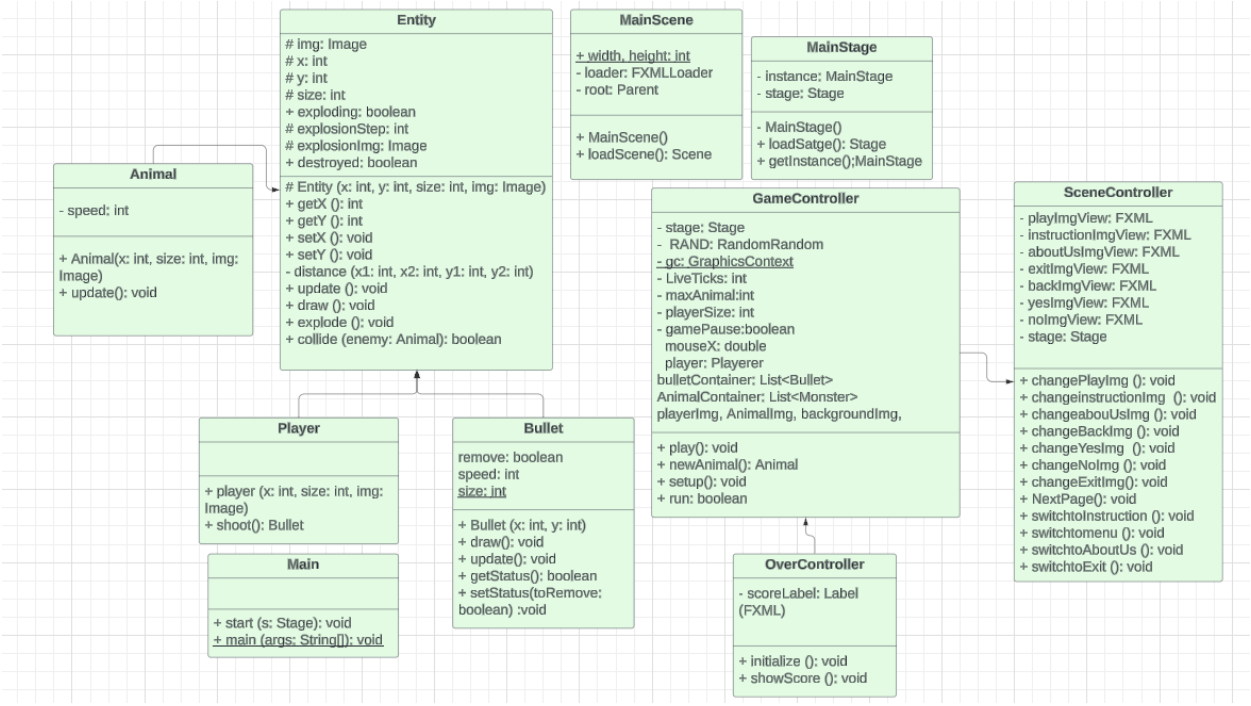
2.6. Use Case diagram

Figure 2.6: Case díagram



2.7. Class diagram

Figure 2.7: UML diagram



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.

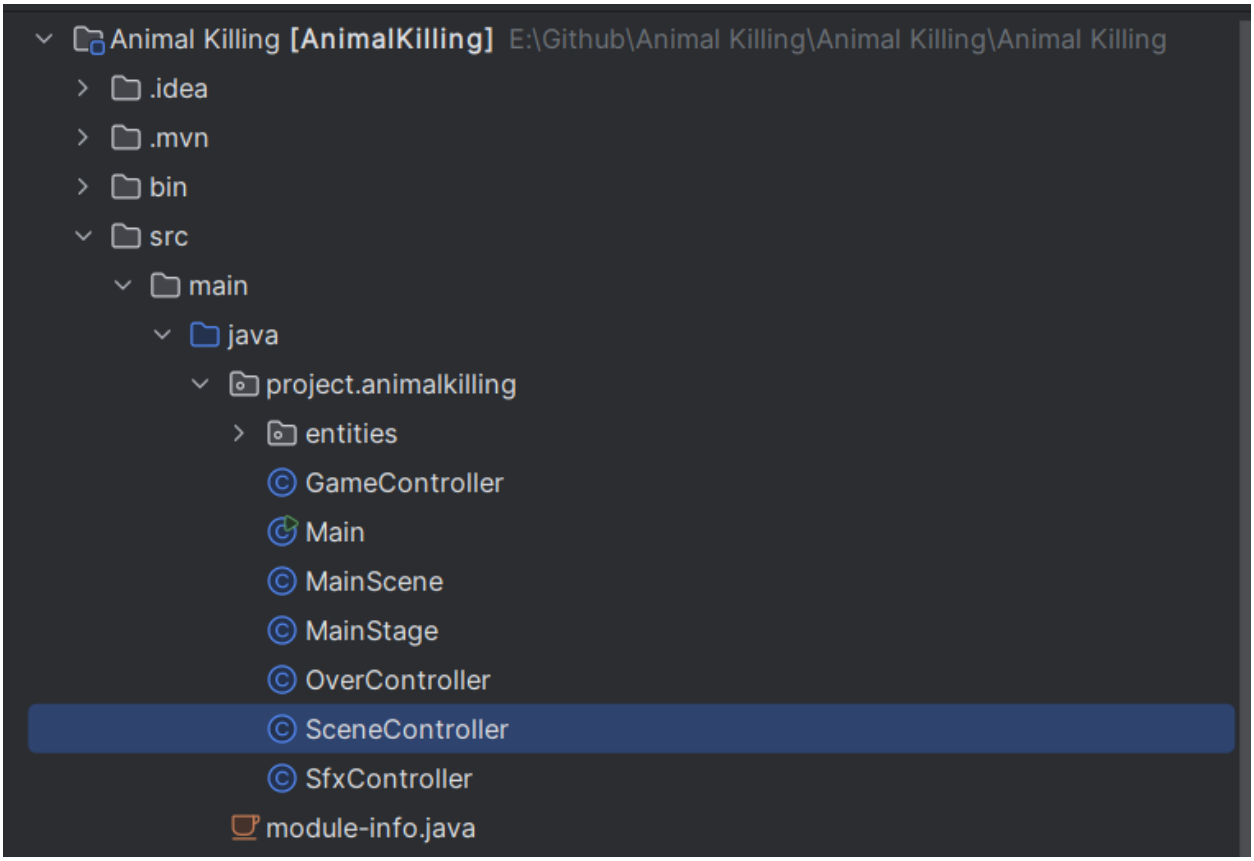
The **Bullet** class, derived from the **Entity** class, represents the projectiles fired by the Ship. It includes attributes such as speed, size, and a Boolean flag called **remove**, which indicates whether the bullet should be deleted. Additionally, it features methods for handling movement, detecting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

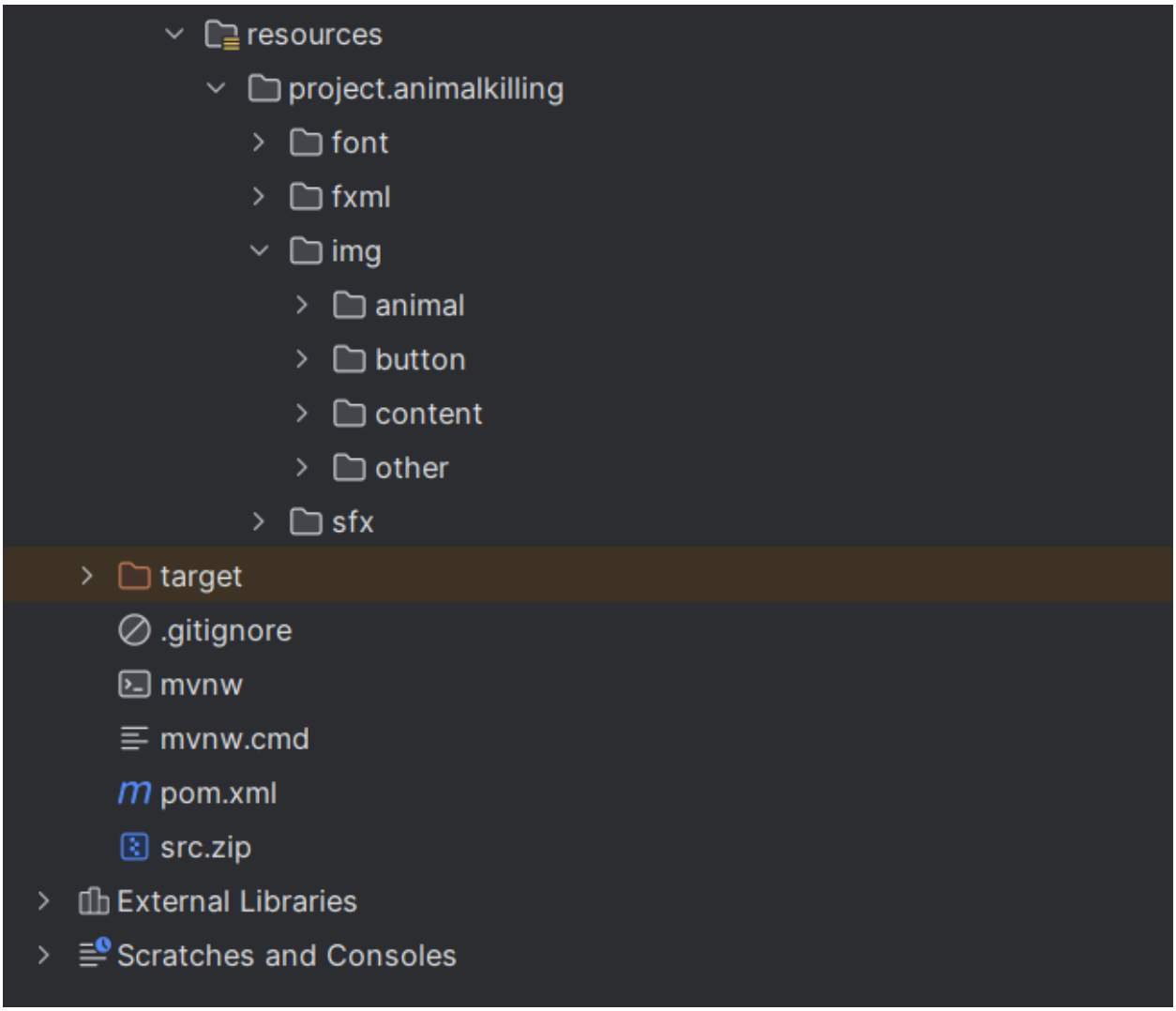
The **Main** class functions as the entry point of the game application, managing the initialization process and setting up the fundamental aspects of the game stage.

CHAP 3: DESIGN AND IMPLEMENTATION

3.1. Project Structure



Project's GitHub: Link



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:

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.

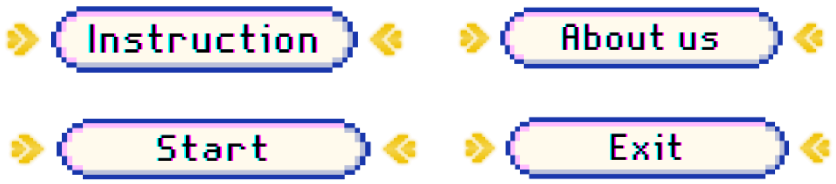
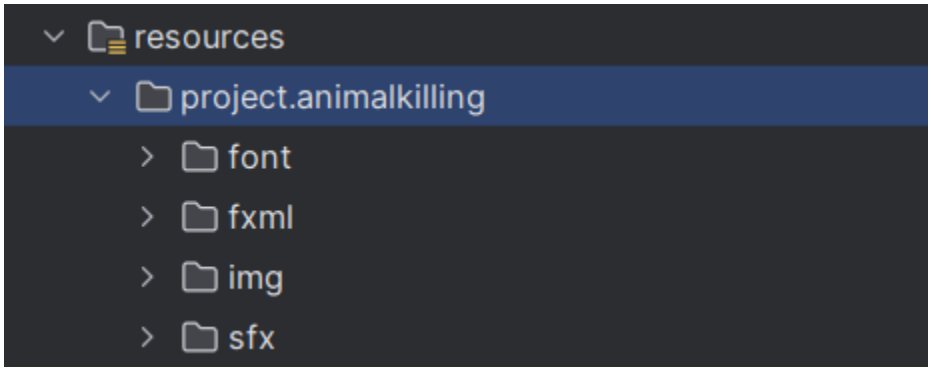


Figure 1: Click to play

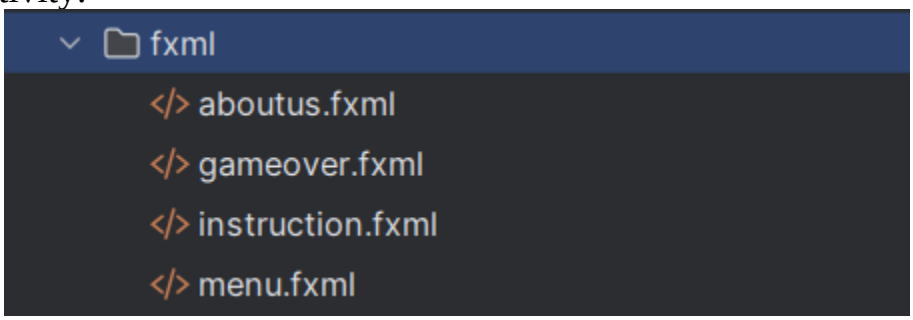
Figure 2: Animal

Figure 3: Player

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

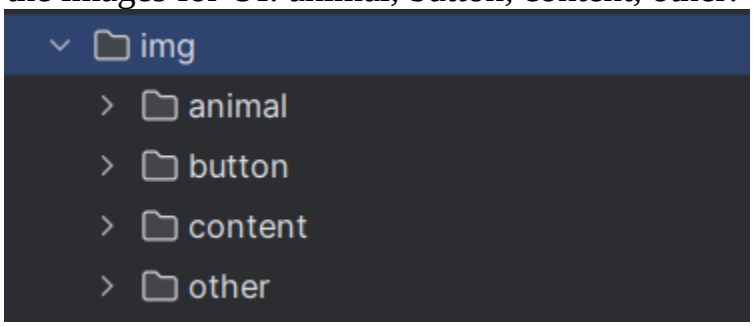


- **fxml:** A layout defines the structure for a user interface in your app, such as in an activity.

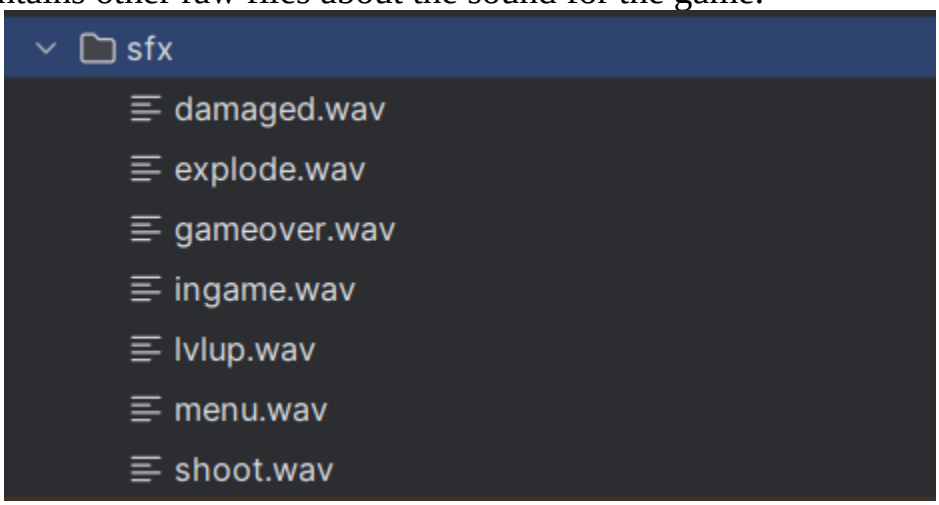


- **font:** Some fonts and file xml fontfamily which declairs our addition fonts for later use.

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



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



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.

```java
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;
    }
}
```

```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);
}
```

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)

4.2. Demo video

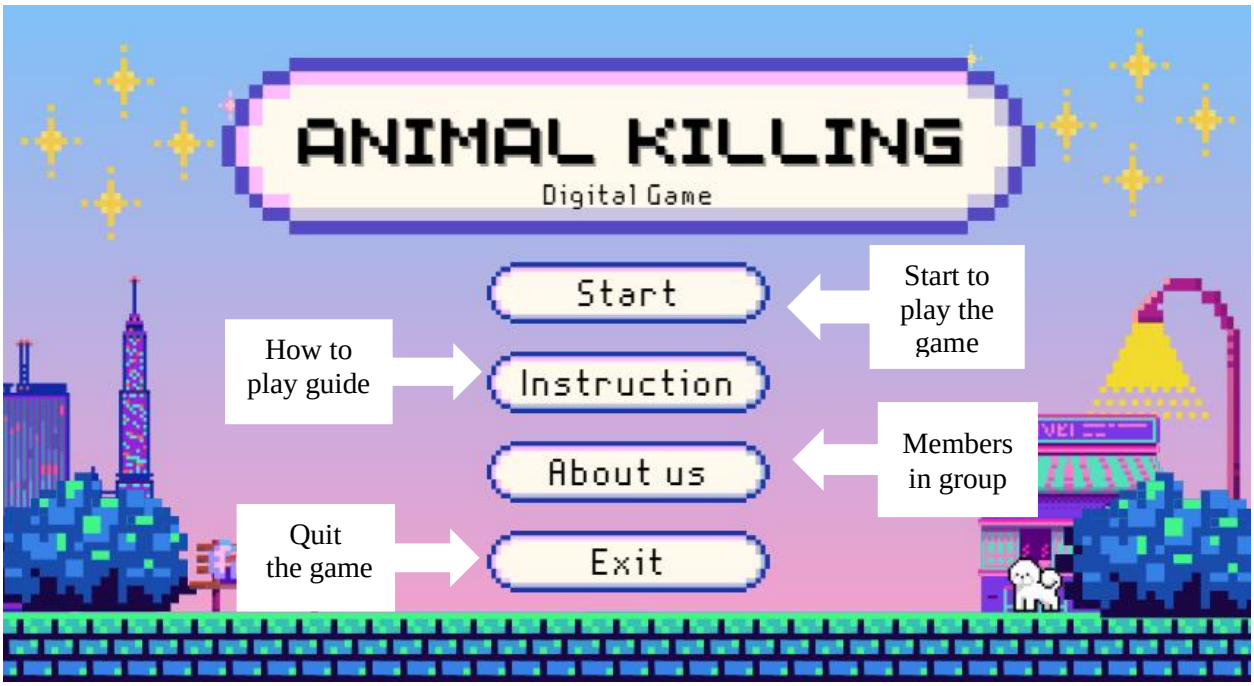
https://youtu.be/JHqTWL32Gl4

4.3. Instruction

4.3.1. Begin the game

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

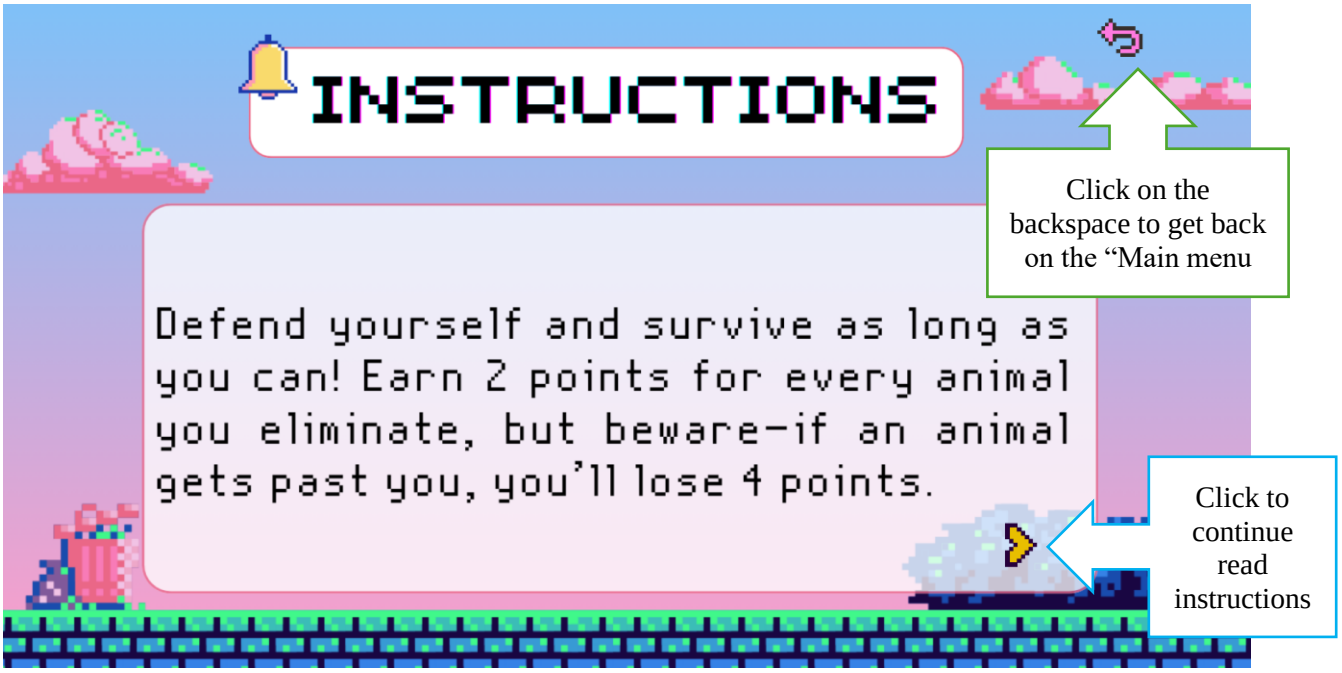
4.3.2. Main menu



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:

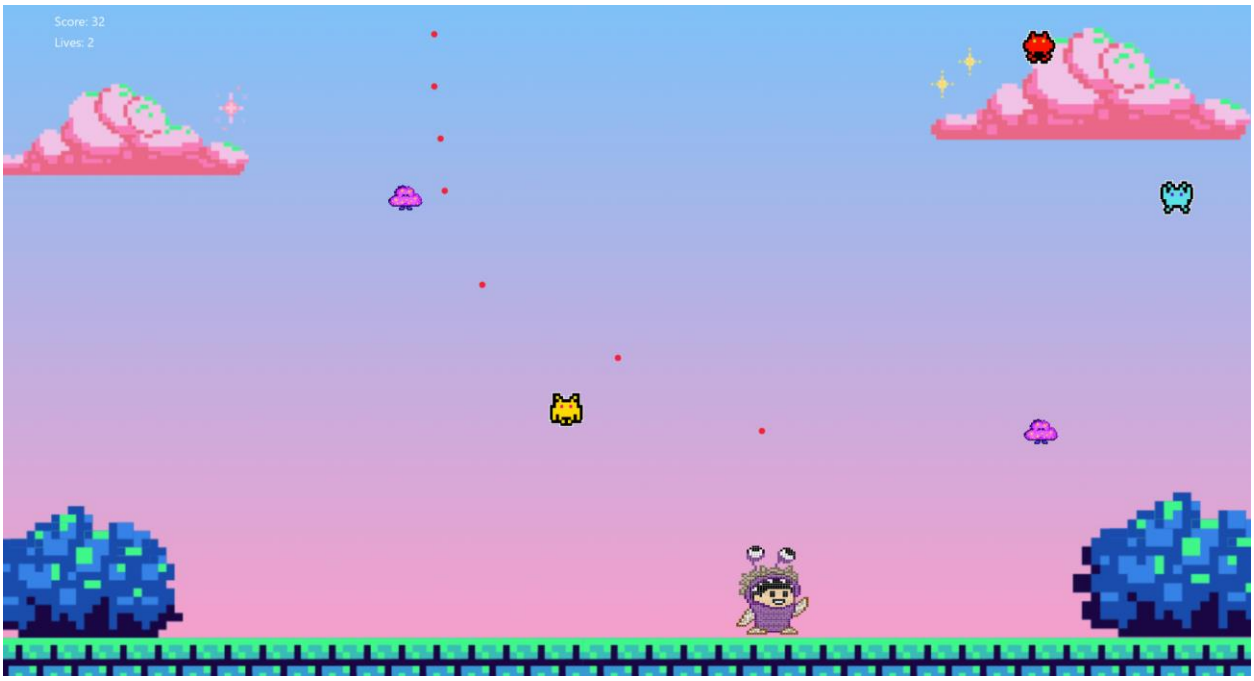


4.4.4. Game over

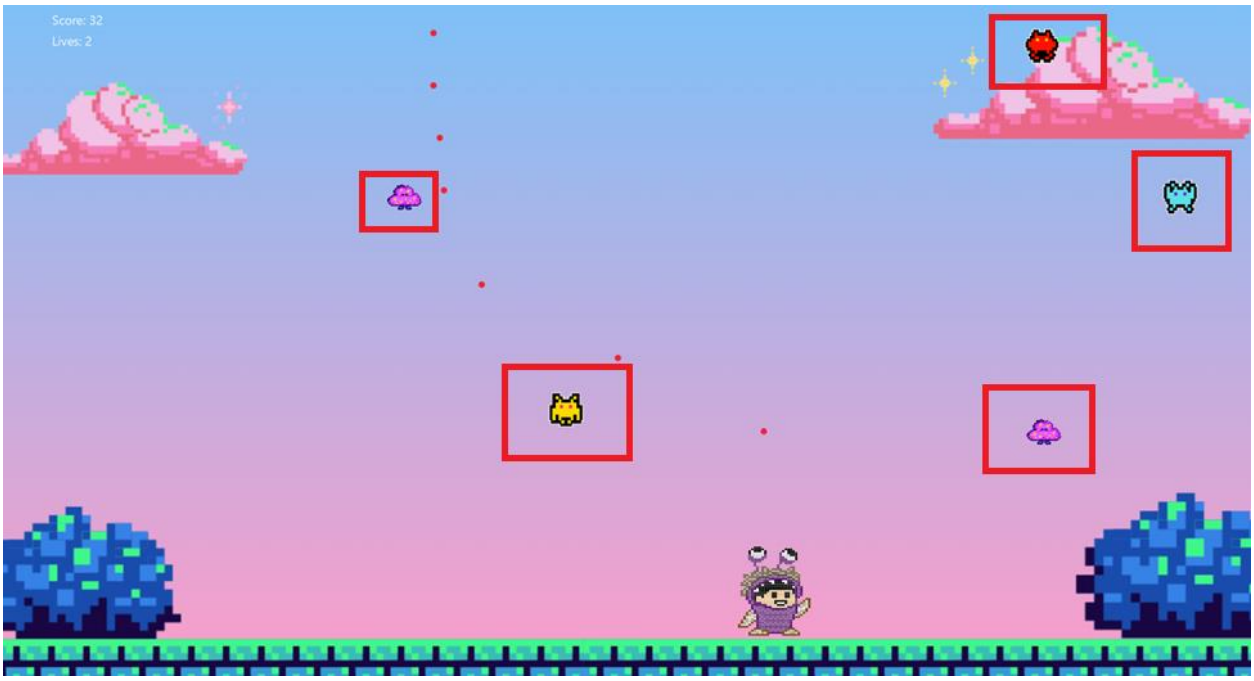
When the player is defeated, the "game over" menu appears. It presents two choices: "YES" to continue playing or "NO" to return to the main menu. These options are highlighted when hovered over.

4.4.5.Let’s play

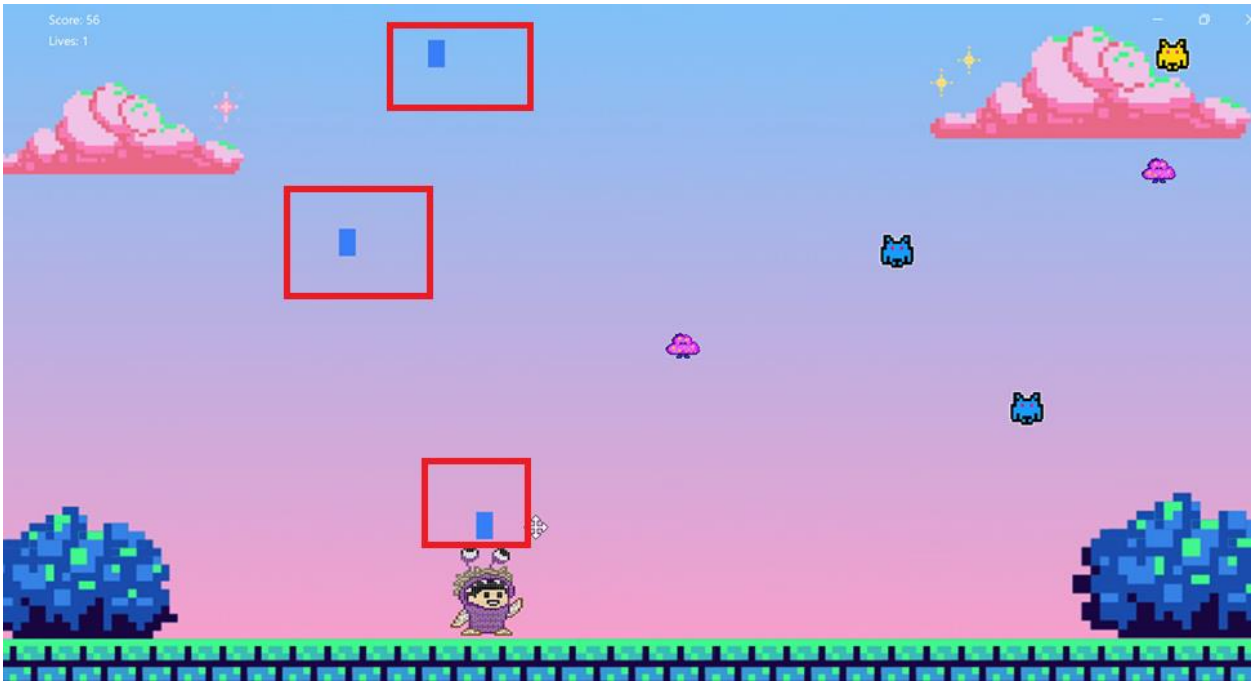
- Start the game



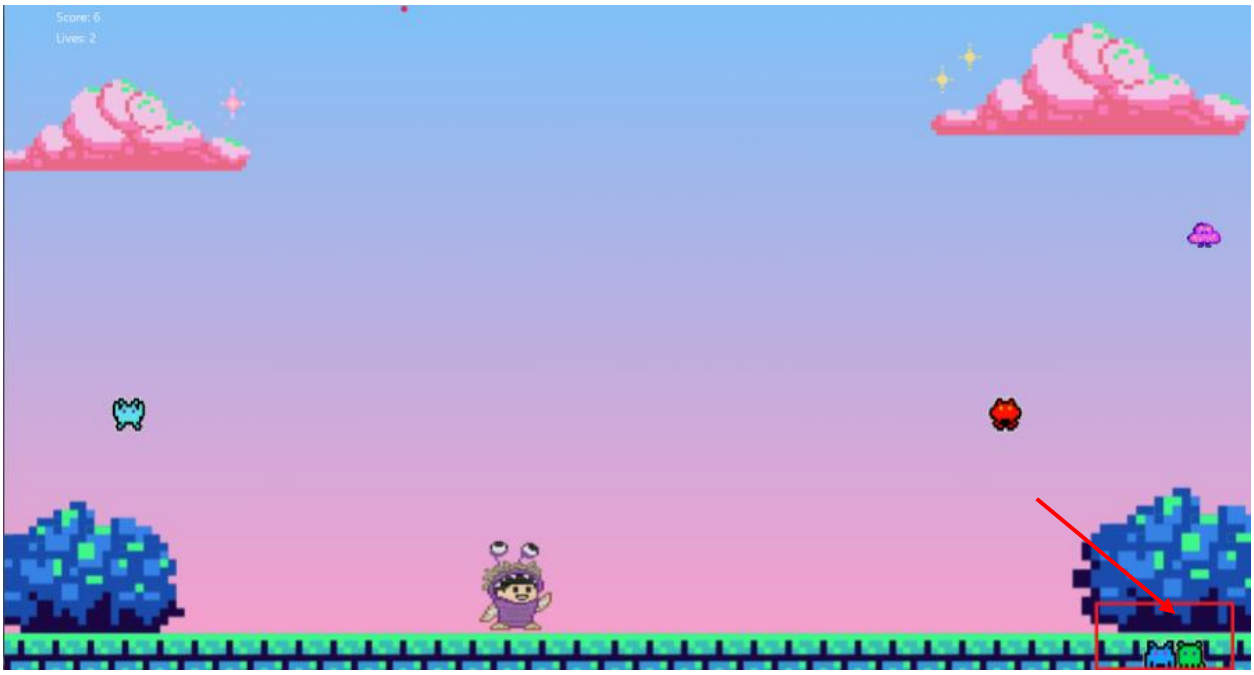
- Defeat one animal to gain two points



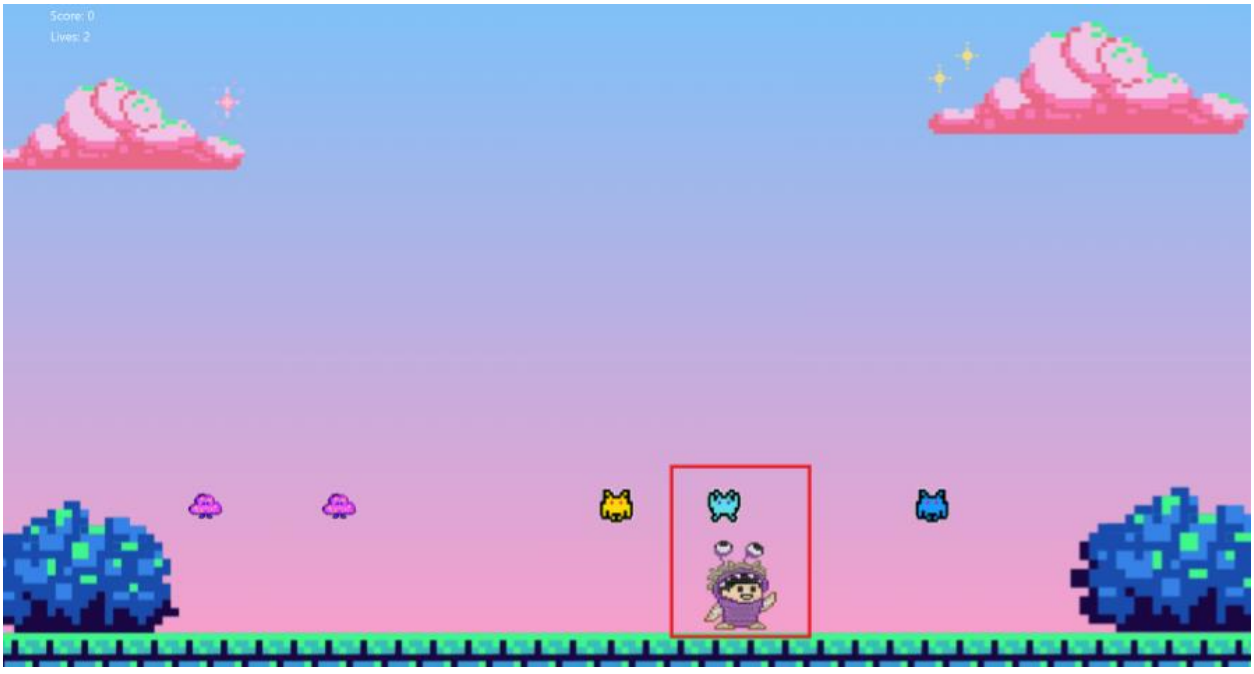
- If you score above 50 your bullets will be upgraded



- 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

Project's GitHub: Link