VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Object-Oriented Programming
IT079IU

FINAL REPORT

Course by Prof. Tran Thanh Tung

Topic: The Last Pirate

BY NEWBIE - MEMBER LIST

Name	Student ID	Member	Contribution
Nguyễn Thị Mỹ Tuyền	ITITIU22236	Team Leader	100%
Nguyễn Huỳnh Tuyết Nhi	ITITIU22237	Team Member	100%
Huỳnh Thanh Trúc	ITITIU22169	Team Member	100%
Đặng Danh Hương	ITCSIU22053	Team Member	100%

TABLE OF CONTENTS

CHAP 1: INTRODUCTION

The Last Pirate is an exciting pixel-art platformer game where players embark on an adventurous journey as a brave pirate. Navigate through perilous levels filled with traps, enemies, and challenges while collecting treasures and defeating foes. The game's objective is to traverse various platforms, defeat enemies like crabs and cannons, and ultimately conquer all levels. With its retro aesthetic and engaging mechanics, **The Last Pirate** immerses players in a swashbuckling adventure that tests their agility, strategy, and timing.

1.1 Gaming in the Field:

In the rapidly evolving domain of software engineering and development, video game creation stands out while also sharing similarities with other software projects. What makes it distinctive is its integration of efforts from diverse disciplines, including art, music, acting, and programming. Additionally, achieving captivating gameplay often relies on iterative development and prototyping.

We are creating **The Last Pirate** as a game project for our "Object-Oriented Programming" course, which is a four-credit subject in our degree program. This project was chosen to deepen our understanding of the software development cycle, improve our ability to manage development timelines effectively, and practice designing visually appealing graphics. Through this project, we aim to strengthen our programming skills while applying key theoretical concepts from class, including object-oriented programming principles like inheritance, encapsulation, and polymorphism.

1.2 About the game project:

There are many platformer games similar to **The Last Pirate** available on various platforms. However, many of them follow repetitive gameplay mechanics, which can make them less engaging over time.

To address this, our team has developed **The Last Pirate** with a fresh approach, featuring captivating level designs, challenging enemies, and exciting obstacles to keep players entertained. We have focused on creating an immersive gameplay experience through well-designed mechanics, appealing pixel-art graphics, and a dynamic progression system.

In summary, our goal is to provide a more enjoyable and engaging experience for players with **The Last Pirate**, offering a standout platformer game that holds their interest and brings them back for more.

1.3 Our The Last Pirate game:

As mentioned, our goal with **The Last Pirate** is to add exciting features that enhance the player experience and make the game more enjoyable.

The core gameplay involves navigating a pirate through challenging platformer levels filled with obstacles, enemies. Players must use their skills to progress through increasingly difficult stages while enjoying a visually appealing pixel-art environment.

In addition to the basic gameplay, we have included several enhancements to improve the experience:

- **Engaging Level Design**: Carefully crafted levels with dynamic obstacles and hidden treasures to keep players intrigued.
- **Unique Enemy Mechanics**: Diverse enemies with different attack patterns to test players' strategy and reflexes.
- **OOP Principles**: The game applies key object-oriented programming concepts, including inheritance, encapsulation, and design patterns like Singleton and Adapter, to ensure efficient development and scalability.

By incorporating these features, **The Last Pirate** aims to deliver a polished and enjoyable platformer game that keeps players entertained and coming back for more.

1.4 References:

- Pixel image, URL: https://pixelfrog-assets.itch.io/.
- Learn IntelliJ IDEA, URL: https://www.jetbrains.com/help/idea/getting-started.html.
- Platformer game tutorial, URL:https://www.kaaringaming.com/platformer-tutorial.
- Learn GIMP, URL: https://www.gimp.org/tutorials/.

1.5 Developer Team

Name	Student ID	Contribute
Nguyễn Thị Mỹ Tuyền	ITITIU22236	Write report and Make a PowerPoint slide Write CLoud Movement Write checkCLoseToBorder
Nguyễn Huỳnh Tuyết Nhi	ITITIU22237	Write Menu Activity Make a PowerPoint slide Detailed analysis of each class
Huỳnh Thanh Trúc	ITITIU22169	Draw Case Class Write Main Class Process data from mouse and keyboard
Đặng Danh Hương	ITCSIU22053	Support every part of the project Fix bugs Write enemy enities into the games

CHAP 2: SOFTWARE REQUIREMENTS

2.1 What we have:

a. 2D platformer gameplay with intuitive controls.
b. Simple yet visually appealing graphics.
c. Engaging level design with increasing difficulty.
d. Collectibles and unlockables to enhance replayability.
e. Compatibility with both PC and mobile platforms.

2.2 What we want:

a. Develop the game within limited cost.
b. Maximum high definition graphics and visuals.

c. Design the game mechanics and levels in an efficient manner.

2.3 Working tools:

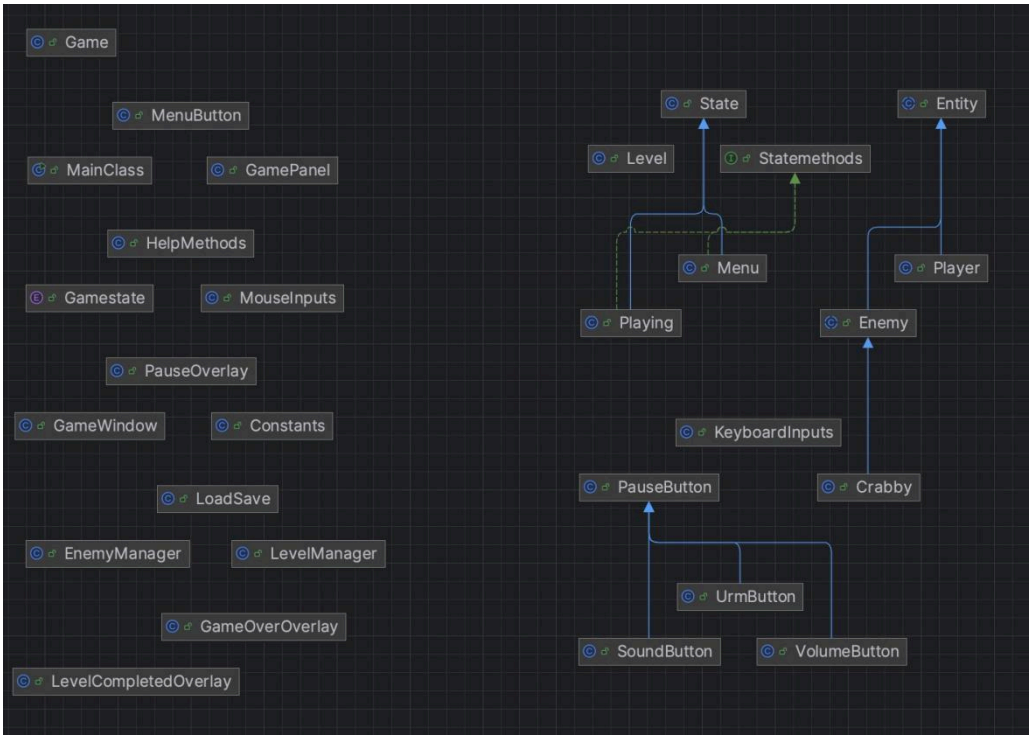
Use Java with JetBrains IntelliJ IDEA :developing backend development to manage and organize the game logic, design the frontend to captivate users and enhance the gameplay experience.
Use GitHub: for version control to track code changes and collaborate effectively on the project.

2.4 Use Case Scenario:

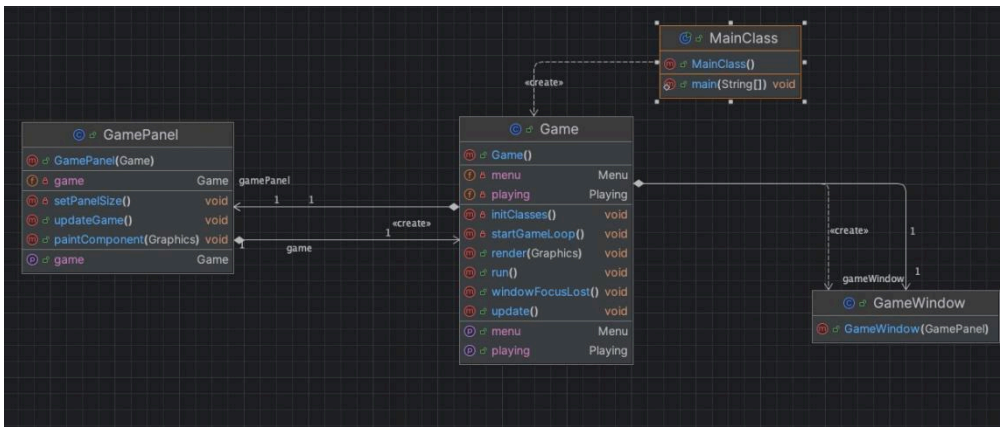
We have created the use cases based on the UX view of the game.

The Last Pirate	PLAY	Play the game
		Option
	Death Screen	Exit the game
		Restart
	QUIT	Exit
		Exit the game

2.5 Class Structure:

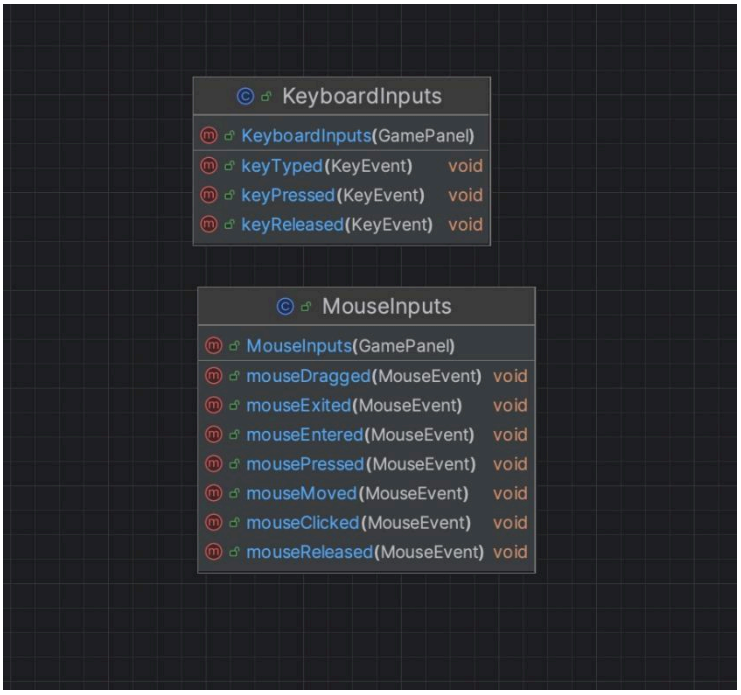


Main process diagram

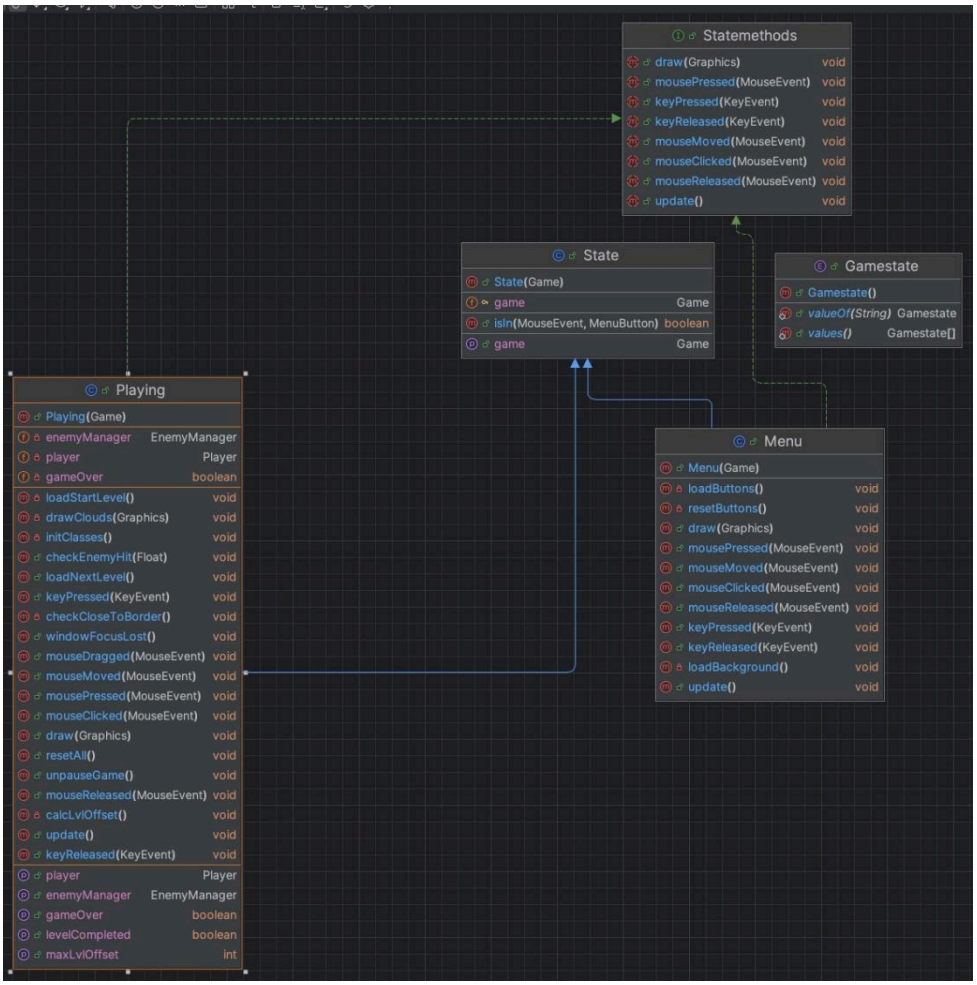


UI process diagram

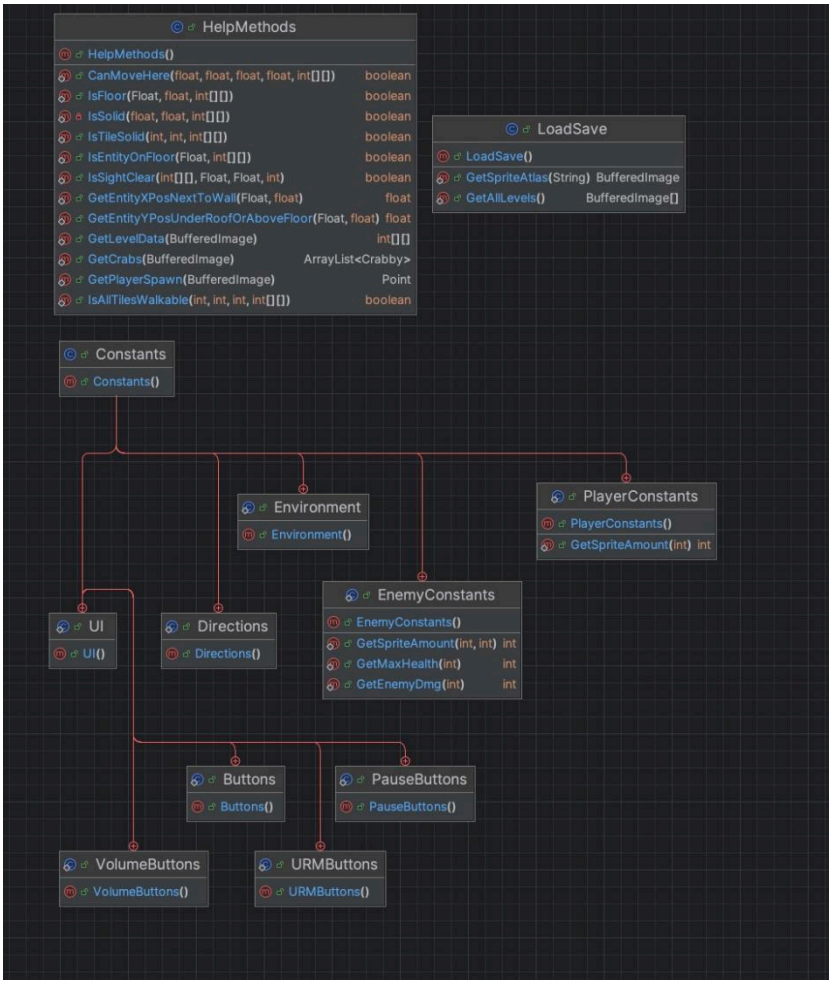
Inputs process diagram



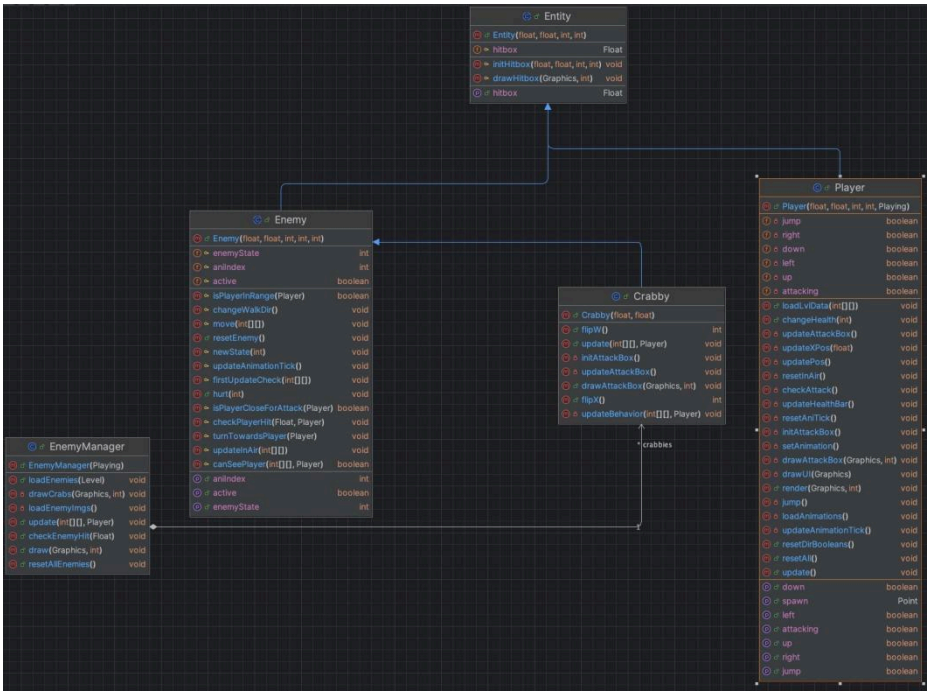
Gamestates process diagram



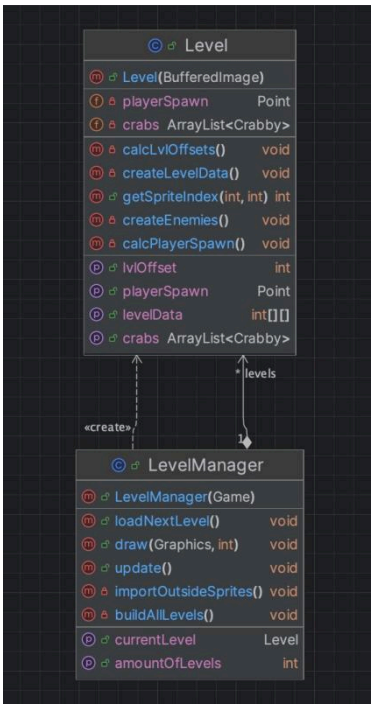
Utilz process diagram



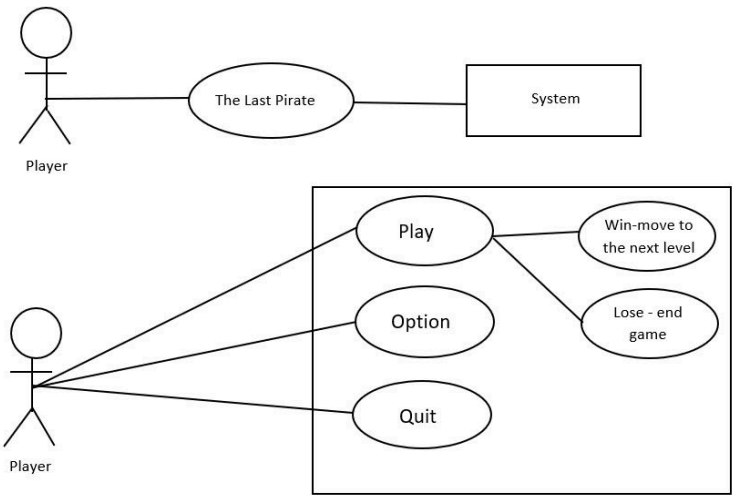
Entities process diagram



Level process diagram



2.6 Case Diagram:



CHAP 3: DESIGN AND IMPLEMENTATION

class Game:

The Game class in this code acts as the "heart" of the game project. It is responsible for managing the entire game loop, game state, and coordinating other components to create the game experience.

```
public class Game implements Runnable {
```

```java
@Override
public void run() {
//method to control the game state update rate(UPS) and frame drawing rate(FPS)
```

```java
    while (true) {
    //The game loop runs continuously to control the flow of the game.
        long currentTime = System.nanoTime(); //Calculate elapsed time

        deltaU += (currentTime - previousTime) / timePerUpdate;
        deltaF += (currentTime - previousTime) / timePerFrame;
        previousTime = currentTime;

        if (deltaU >= 1) { //Update status (Logic Update)
            update();
            updates++;
            deltaU--;
        }

        if (deltaF >= 1) {  //Draw the frame (Render Frame)
            gamePanel.repaint();
            frames++;
            deltaF--;
        }

        if (System.currentTimeMillis() - lastCheck >= 1000) {  //Check and display FPS, UPS
            lastCheck = System.currentTimeMillis();
            System.out.println("FPS: " + frames + " | UPS: " + updates);
            frames = 0;
            updates = 0;

        }
    }
}
```

class MouseInputs:

```java
public class MouseInputs implements MouseListener, MouseMotionListener {

//This class allows the system to handle mouse events in a game.
//It uses the MouseListener and MouseMotionListener interfaces to respond to mouse actions
```

```java
@Override
  public void mouseDragged(MouseEvent e) {
  //Handle when the user drags the mouse
    switch (Gamestate.state) {
    case PLAYING:
      gamePanel.getGame().getPlaying().mouseDragged(e);
      break;
    case OPTIONS:
      gamePanel.getGame().getGameOptions().mouseDragged(e);
      break;
    default:
      break;
    }
  }
  @Override
  public void mouseMoved(MouseEvent e) {
  //Handles when the user moves the mouse without pressing a button
    switch (Gamestate.state) {
    case MENU:
      gamePanel.getGame().getMenu().mouseMoved(e);
      break;
    case PLAYING:
      gamePanel.getGame().getPlaying().mouseMoved(e);
      break;
    case OPTIONS:
      gamePanel.getGame().getGameOptions().mouseMoved(e);
```

```
            break;
        default:
            break;
        }
    }
    @Override
    public void mouseClicked(MouseEvent e) {
    //Handle when the user clicks
        switch (Gamestate.state) {
        case PLAYING:
            gamePanel.getGame().getPlaying().mouseClicked(e);
            break;
        default:
            break;
        }
    }
    @Override
    public void mousePressed(MouseEvent e) {
    //Handle when the user presses the mouse button
        switch (Gamestate.state) {
        case MENU:
            gamePanel.getGame().getMenu().mousePressed(e);
            break;
        case PLAYING:
            gamePanel.getGame().getPlaying().mousePressed(e);
            break;
        case OPTIONS:
            gamePanel.getGame().getGameOptions().mousePressed(e);
            break;
        default:
            break;
        }
    }
    @Override
    public void mouseReleased(MouseEvent e) {
    //Handle when the user releases the mouse button
        switch (Gamestate.state) {
        case MENU:
            gamePanel.getGame().getMenu().mouseReleased(e);
            break;
        case PLAYING:
            gamePanel.getGame().getPlaying().mouseReleased(e);
            break;
        case OPTIONS:
            gamePanel.getGame().getGameOptions().mouseReleased(e);
            break;
        default:
            break;
        }
    }
    @Override
    public void mouseEntered(MouseEvent e) {
    //Handles when the mouse enters the area of the game window.
    }

    @Override
    public void mouseExited(MouseEvent e) {
    //Handles when the mouse leaves the area of  the game window.
    }
```

class KeyboardInputs:

```
public class KeyboardInputs implements KeyListener {
//This class handles in-game keyboard events by implementing the KeyListener interface
```

class GameWindow:

GameWindow is the game window management class. It uses JFrame to create windows and add interface elements like GamePanel.

```
public class GameWindow {
//The GameWindow class has the function of creating and managing game windows
```

```
jframe.add(gamePanel);
//Add a gamePanel (a JPanel containing game content) to the window.
```

class GamePanel:

GamePanel processes and draws the game interface. The paintComponent method calls the render() function to display the graphics, acting as a bridge between the logic and the interface, ensuring everything displays correctly.

```
public void paintComponent(Graphics g) {
//draw the game in class JPanel
    super.paintComponent(g);
    game.render(g);
}
```

class Constants:

Constants is a class that stores common constants in the game, supporting the organization and management of fixed values, making the code easy to maintain and change.

```
public class Constants {
//Java class designed to store constants related to various components of the game
//The main purpose of this class is to organize and manage constants for easy access and maintenance
```

AbstractclassEntity

```
public abstract class Entity {
// class serves as an abstract base class for all game entities, such as players, enemies, or interactive objects
```

The Entity class serves as the foundation for all entities in the game. Entities are any items or characters in the game universe, including the player, adversaries, and interacting objects. This class contains general attributes and methods that are shared by all entities, with the aim that other particular entity classes (such Player, Enemy,...) will inherit from it.

```
protected void drawAttackBox(Graphics g, int xLvlOffset) {
// Draws the entity's attack box
```

```
protected void drawHitbox(Graphics g, int xLvlOffset) {
    // For debugging the hitbox
```

```
protected void initHitbox(int width, int height) {
// Initializes the hitbox for the entity based on its position and dimensions.
```

```
public Rectangle2D.Float getHitbox() {
// Allows other parts of the game (e.g., collision detection systems) to retrieve the hitbox of the entity.
    return hitbox;
```

Player

```java
public class Player extends Entity {
//  define the player's properties and behavior
// preparing for game functionality such as animation handling, movement, collision detection, and rendering
    private BufferedImage[][] animations;
    private int aniTick, aniIndex, aniSpeed = 25;
    private int playerAction = IDLE;
    private boolean moving = false, attacking = false;
    private boolean left, up, right, down, jump;
    private float playerSpeed = 1.0f * Game.SCALE;
    private int[][] lvlData;
    private float xDrawOffset = 21 * Game.SCALE;
    private float yDrawOffset = 4 * Game.SCALE;
```

The Player class handles everything connected to the player's character, including movement mechanics, health, and animations. It allows you to interact with the environment (e.g., collect potions, avoid spikes, strike foes)**.**

```java
public Player(float x, float y, int width, int height, Playing playing) {
// Initializes the player's position, dimension, gameplay related attributes, and collision properties
    super(x, y, width, height);
    this.playing = playing;
    loadAnimations();
    initHitbox(x, y, (int) (20 * Game.SCALE), (int) (27 * Game.SCALE));
    initAttackBox();
}
```

```java
 public void setSpawn(Point spawn) {
 // Updates the player's position (x and y) and synchronizes it with the player's hitbox to ensure accurate collision detection.
 // This is typically used when the player is spawned or respawned in the game.
    this.x = spawn.x;
    this.y = spawn.y;
    hitbox.x = x;
    hitbox.y = y;
 }
```

```java
private void initAttackBox() {
// creates a rectangular area (Rectangle2D.Float) called attackBox that is used to detect collisions between the player's attack and other entities (e.g., enemies)
    attackBox = new Rectangle2D.Float(x, y, (int) (20 * Game.SCALE), (int) (20 * Game.SCALE));
}
```

```java
  private void drawUI(Graphics g) {
  // drawing the user interface (UI) elements on the screen, specifically the status bar and the health bar
  // The health bar is drawn in red to show how much health the player has remaining
      g.drawImage(statusBarImg, statusBarX, statusBarY, statusBarWidth, statusBarHeight, null);
      g.setColor(Color.red);
      g.fillRect(healthBarXStart + statusBarX, healthBarYStart + statusBarY, healthWidth, healthBarHeight);
  }
```

```java
public void update() {
// updating the player's state, health , animation, and handling interactions like attacking, and spikes

    updateHealthBar();
// Updates the health bar based on the player's current health
```

```java
    updateAttackBox();
// This method updates the boundaries or hitbox for the player's attack
    updatePos();
//  This method updates the player’s position on the screen
    if (attacking)
```

```java
private void updateHealthBar() {
// updating the health bar's display based on the player's current health
    healthWidth = (int) ((currentHealth / (float) maxHealth) * healthBarWidth);
}
```

```java
    private void checkAttack() {
//  Checks if the player's attack hits an enemy
        if (attackChecked || aniIndex != 1)
            return;
        attackChecked = true;
        playing.checkEnemyHit(attackBox);
```

```java
private void setAnimation() {
// Determines the player's animation based on the current action
    int startAni = playerAction;
```

```java
private void updateAttackBox() {
// updates the position of the player's attack box, depending on which direction the player is facing (left or right)
    if (right)
        attackBox.x = hitbox.x + hitbox.width + (int) (Game.SCALE * 10);
    else if (left)
        attackBox.x = hitbox.x - hitbox.width - (int) (Game.SCALE * 10);

    attackBox.y = hitbox.y + (Game.SCALE * 10);
}
```

```java
public void render(Graphics g, int lvlOffset) {
Draws the player's current animation and the UI
    g.drawImage(animations[playerAction][aniIndex], (int) (hitbox.x - xDrawOffset) - lvlOffset + flipX, (int) (hitbox.y - yDrawOffset), width * flipW, height, null);
    drawHitbox(g, lvlOffset);
    drawAttackBox(g, lvlOffset);
```

```java
private void updatePos() {
updating the character's position in the game world and simulating realistic physics like gravity

    moving = false;
```

```java
private void jump() {
// initiating the player's jump, allows the player to jump only if they are not already in the air
    if (inAir)
        return;
    inAir = true;
    airSpeed = jumpSpeed;
}
```

```java
private void resetInAir() {
// reset the player's state after they have finished jumping or falling
    inAir = false;
    airSpeed = 0;
}
```

```java
public void resetDirBooleans() {
//  Resets the movement booleans (left, right, up, down)
    left = false;
    right = false;
    up = false;
    down = false;
```

```java
private void updateXPos(float xSpeed) {
Updates the player's attack box based on movement and changes its position
    if (CanMoveHere(hitbox.x + xSpeed, hitbox.y, hitbox.width, hitbox.height, lvlData))
```

```java
private void loadAnimations() {
Loads the player's sprite animations from a sprite atlas
```

```java
public void resetAll() {
// Resets the player's state, including position, health, and animation. It sets the player back to idle and fully healed
    resetDirBooleans();
```

```java
public void changeHealth(int value) {
//  Adjusts the player's health by a specified value and ensures the health remains within bounds
    currentHealth += value;
```

```java
 private void loadAnimations() {
 // loadAnimations() is responsible for loading and organizing the animation sprites for different states of the player character
 // Initializes a 2D array to store the animation frames.
     BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.PLAYER_ATLAS);
     animations = new BufferedImage[7][8];
     for (int j = 0; j < animations.length; j++)
         for (int i = 0; i < animations[j].length; i++)
             animations[j][i] = img.getSubimage(i * 64, j * 40, 64, 40);

     statusBarImg = LoadSave.GetSpriteAtlas(LoadSave.STATUS_BAR);
 }
```

```java
public void loadLvlData(int[][] lvlData) {
// assigns the level's tile data to the player and checks if the player is grounded or in the air
    this.lvlData = lvlData;
    if (!IsEntityOnFloor(hitbox, lvlData))
        inAir = true;
```

LoadSave

```java
public static BufferedImage GetSpriteAtlas(String fileName) {
Loads an image file from the resources folder
    BufferedImage img = null;
```

```java
public static BufferedImage[] GetAllLevels() {
Loads all level images from the "lvls" directory located in the resources folder and returns them as an array of BufferedImage
    URL url = LoadSave.class.getResource("/lvls");
```

The LoadSave class is generally used to load image assets and level data from files located within the game's resources. It offers techniques to retrieve sprite atlases and level images.

LevelManager

```java
public class LevelManager {
// managing levels in a game
```

```java
public LevelManager(Game game) {
// Prepares all the levels for the game
    this.game = game;
```

```java
private void buildAllLevels() {
// Constructs all levels by reading level data from external files.
    BufferedImage[] allLevels = LoadSave.GetAllLevels();
```

```java
public void loadNextLevel() {
Loads the next level in the sequence.
    lvlIndex++;
```

```java
private void importOutsideSprites() {
// Loads the sprite atlas used to render level tiles.
// Extracts individual tile images and stores them in the levelSprite array.
    BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.LEVEL_ATLAS);
    levelSprite = new BufferedImage[48];
```

```java
public void draw(Graphics g, int lvlOffset) {
// This method iterates through all the tiles in the current level and draws them on the screen.
```

The class LevelManager oversees level loading and rendering, handles level transitions, and provides level-related data.

Level

```
public class Level {
// represents a game level, managing data such as enemies, items, objects, and the player's spawn location
```

```
public int getSpriteIndex(int x, int y) {
// Retrieves the tile or sprite index at a specific position in the level.
    return lvlData[y][x];
```

The Level class does not appear to actually perform anything; it is basically a data structure and initializer for the game level.

HelpMethod

```
public class HelpMethods {
// provides utility methods

    public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) {
//  Checks if an entity can move to a specific position without colliding with solid tiles.
```

```
private static boolean IsSolid(float x, float y, int[][] lvlData) {
Determines if a specific point on the level corresponds to a solid tile.
Checks the value in lvlData at the specified tile coordinates
    int maxWidth = lvlData[0].length * Game.TILES_SIZE;
    if (x < 0 || x >= maxWidth)
```

```
public static float GetEntityYPosUnderRoofOrAboveFloor(Rectangle2D.Float hitbox, float airSpeed) {
// Calculates the y-position of an entity when it collides with a floor or roof.
// Ensures proper placement of entities during vertical movement
    int currentTile = (int) (hitbox.y / Game.TILES_SIZE);
    if (airSpeed > 0) {
        // Falling - touching floor
        int tileYPos = currentTile * Game.TILES_SIZE;
        int yOffset = (int) (Game.TILES_SIZE - hitbox.height);
        return tileYPos + yOffset - 1;
    } else
        // Jumping
        return currentTile * Game.TILES_SIZE;
```

```
public static boolean IsTileSolid(int xTile, int yTile, int[][] lvlData) {
Checks if a specific tile (identified by xTile and yTile) is solid
    int value = lvlData[yTile][xTile];
```

```
public static boolean IsFloor(Rectangle2D.Float hitbox, float xSpeed, int[][] lvlData) {
Checks if the entity's current horizontal movement (xSpeed) will cause it to collide with the floor
    if (xSpeed > 0)
```

```
public static boolean IsAllTilesWalkable(int xStart, int xEnd, int y, int[][] lvlData) {
Checks if all tiles between xStart and xEnd on the specified y coordinate are walkable
    for (int i = 0; i < xEnd - xStart; i++) {
```

```
public static boolean IsSightClear(int[][] lvlData, Rectangle2D.Float firstHitbox, Rectangle2D.Float secondHitbox, int yTile) {
 Checks if there is a clear line of sight (no obstacles) between two entities.
```

```
public static int[][] GetLevelData(BufferedImage img) {
 Converts a level image into a 2D integer array representing the level data
```

```
public static ArrayList<Crabby> GetCrabs(BufferedImage img) {
 Extracts the positions of crabs (enemies) from a level image and returns them as an array list of Crabby entities
    ArrayList<Crabby> list = new ArrayList<>();
```

```
public static Point GetPlayerSpawn(BufferedImage img) {
 Finds and returns the spawn point for the player from the level image
```

```java
public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) {
 Checks if an entity can move to the given coordinates without colliding with solid tiles
    if (!IsSolid(x, y, lvlData))
```

```java
public static float GetEntityXPosNextToWall(Rectangle2D.Float hitbox, float xSpeed) {
// Calculates the x-position of an entity when it collides with a wall.
// Prevents entities from overlapping with walls during movement.
    int currentTile = (int) (hitbox.x / Game.TILES_SIZE);
    if (xSpeed > 0) {
        // Right
        int tileXPos = currentTile * Game.TILES_SIZE;
        int xOffset = (int) (Game.TILES_SIZE - hitbox.width);
        return tileXPos + xOffset - 1;
    } else
        // Left
        return currentTile * Game.TILES_SIZE;
}
```

```java
public static boolean IsEntityOnFloor(Rectangle2D.Float hitbox, int[][] lvlData) {
// Checks if an entity is standing on a floor tile.
// check pixel below bottom left and bottom right
    if (!IsSolid(hitbox.x, hitbox.y + hitbox.height + 1, lvlData))
        if (!IsSolid(hitbox.x + hitbox.width, hitbox.y + hitbox.height + 1, lvlData))
            return false;
    return true;
}
```

The HelpMethods class is a utility class that contains static methods for interacting with game-level data and entities. It doesn't maintain state or have non-static members, therefore it doesn't need a main function**.**

Game

```java
public Game() {
initializes the game by calling initClasses() to set up necessary game components
    initClasses();

    gamePanel = new GamePanel(this);
    gameWindow = new GameWindow(gamePanel);
    gamePanel.setFocusable(focusable:true);
    gamePanel.requestFocus();

    startGameLoop();
```

```java
public void update() {
// This method checks the current game state (Gamestate.state) and calls the update() method of the corresponding game state class: MENU,PLAYING,OPTIONS,QUIT or default
    switch (Gamestate.state) {
    case MENU:
        menu.update();
        break;
    case PLAYING:
        playing.update();
        break;
    case OPTIONS:
        gameOptions.update();
        break;
    case QUIT:
    default:
        System.exit(status:0);
        break;
```

```java
@Override
public void run() {
This is the core game loop that runs in a separate thread
```

```java
public void windowFocusLost() {
This method is called when the game window loses focus
```

```
public Menu getMenu() {
Returns the menu object, used to interact with the game's menu state
    return menu;
}
public Playing getPlaying() {
Returns the playing object, used to interact with the game's playing state
    return playing;
```

```
public void render(Graphics g) {
// This method handles the drawing of the game based on the current state
    switch (Gamestate.state) {
```

The Game class serves as the game's foundation, controlling, structuring, and maintaining everything from the operational logic to the graphical user interface. It serves as the primary link between all of the game's aspects, ensuring a consistent and seamless gameplay experience.

StateMethod

```
public interface Statemethods {
// This interface is likely used to handle different game states such as the menu, gameplay, options, and others
    public void update();

    public void draw(Graphics g);
// used to render the game world, including the player, enemies, the environment

    public void mouseClicked(MouseEvent e);
// triggered when the mouse is clicked (both pressed and released) in the game window, provides information about the mouse action (such as the position of the click)

    public void mousePressed(MouseEvent e);
// This method is invoked when the mouse button is pressed down, but before it is released.

    public void mouseReleased(MouseEvent e);
// Used to finalize mouse interactions after the user releases the mouse button

    public void mouseMoved(MouseEvent e);
// It's used to track the mouse movement

    public void keyPressed(KeyEvent e);
// It is used to detect user input when a key is pressed

    public void keyReleased(KeyEvent e);
// It's used to detect the end of a key press, often for actions that need to stop when the key is released (ex: stopping movement).
```

The Statemethods interface provides a framework for handling game states in a systematic and consistent manner. It outlines the key techniques that any game state must implement to deal with updates, graphics, and user inputs properly.

State

```
public class State {
// a base class for managing different game states, such as the menu, playing, paused, game over
    protected Game game;
```

Playing

```
   private void initClasses() {
/setting up all the critical components required to run the gameplay in the Playing state.These components include managing the player, enemies, objects, level data
    levelManager = new LevelManager(game);
```

```
 public void windowFocusLost() {
use when the window is no longer the active application on the user's screen
    player.resetDirBooleans();
```

The Playing class represents the main gameplay state of a game. This class serves as the game's "controller," ensuring that all game components operate together and are properly updated throughout the gameplay process.

GameState

```
public enum Gamestate {
// This enum can be used in the game logic to switch between different states playing, menu, options, quit
    PLAYING, MENU, OPTIONS, QUIT;
```

Menu

```
    private void loadBackground() {
//  Loads and scales the background images
        backgroundImg = LoadSave.GetSpriteAtlas(LoadSave.MENU_BACKGROUND);
        menuWidth = (int) (backgroundImg.getWidth() * Game.SCALE);
        menuHeight = (int) (backgroundImg.getHeight() * Game.SCALE);
```

```
        menuX = Game.GAME_WIDTH / 2 - menuWidth / 2;
        menuY = (int) (45 * Game.SCALE);
    }
    private void loadButtons() {
//   Initializes the menu buttons with their positions and corresponding game states
        buttons[0] = new MenuButton(Game.GAME_WIDTH / 2, (int) (150 * Game.SCALE), 0, Gamestate.PLAYING);
        buttons[1] = new MenuButton(Game.GAME_WIDTH / 2, (int) (220 * Game.SCALE), 1, Gamestate.OPTIONS);
        buttons[2] = new MenuButton(Game.GAME_WIDTH / 2, (int) (290 * Game.SCALE), 2, Gamestate.QUIT);
    }
    @Override
    public void update() {
//  Updates the state of each button in the menu.
        for (MenuButton mb : buttons)
            mb.update();
    }
    @Override
    public void draw(Graphics g) {
//  Draws the background and buttons onto the screen.
        g.drawImage(backgroundImgPink, 0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, null);
        g.drawImage(backgroundImg, menuX, menuY, menuWidth, menuHeight, null);
        for (MenuButton mb : buttons)
            mb.draw(g);
    }
    @Override
    public void mouseClicked(MouseEvent e) {
//  Detects when a mouse button is pressed over a button and marks the button as pressed.
        // TODO Auto-generated method stub
    }
    @Override
    public void mousePressed(MouseEvent e) {
//  Detects when a mouse button is pressed over a button and marks the button as pressed
        for (MenuButton mb : buttons) {
            if (isIn(e, mb)) {
                mb.setMousePressed(true);
            }
        }
    }
    @Override
    public void mouseReleased(MouseEvent e) {
//  Checks if a button was clicked (released after being pressed)
        for (MenuButton mb : buttons) {
            if (isIn(e, mb)) {
                if (mb.isMousePressed())
                    mb.applyGamestate();
                break;
            }
        }
        resetButtons();
    }
    private void resetButtons() {
//  Resets the button states
        for (MenuButton mb : buttons)
            mb.resetBools();
    }
    @Override
    public void mouseMoved(MouseEvent e) {
//  Detects mouse movement and highlights buttons when the mouse hovers over them
        for (MenuButton mb : buttons)
            mb.setMouseOver(false);
        for (MenuButton mb : buttons)
            if (isIn(e, mb)) {
                mb.setMouseOver(true);
                break;
            }
    }
    @Override
    public void keyPressed(KeyEvent e) {
//  Detects key presses; if Enter is pressed, the game state switches to PLAYING
        if (e.getKeyCode() == KeyEvent.VK_ENTER)
            Gamestate.state = Gamestate.PLAYING;
    }
    @Override
    public void keyReleased(KeyEvent e) {
//   Placeholder for key release actions (not implemented in this case).
        // TODO Auto-generated method stub
    }
}
```

MenuButtons

```
public class MenuButton {
// rovided is part of the MenuButton class, which represents a button in a menu for a game
    private int xPos, yPos, rowIndex, index;
    private int xOffsetCenter = B_WIDTH / 2;
    private Gamestate state;
    private BufferedImage[] imgs;
```

```
    private boolean mouseOver, mousePressed;
    private Rectangle bounds;
    public MenuButton(int xPos, int yPos, int rowIndex, Gamestate state) {
//  Creates a menu button with the following input(xPos, yPos, rowIndex, state)
        this.xPos = xPos;
        this.yPos = yPos;
        this.rowIndex = rowIndex;
        this.state = state;
        loadImgs();
        initBounds();
    }
    private void initBounds() {
//  Initializes the hitbox (bounding rectangle) of the button to handle mouse interactions
        bounds = new Rectangle(xPos - xOffsetCenter, yPos, B_WIDTH, B_HEIGHT);
    }
    private void loadImgs() {
//  Loads the button's images from the sprite atlas
        imgs = new BufferedImage[3];
        BufferedImage temp = LoadSave.GetSpriteAtlas(LoadSave.MENU_BUTTONS);
        for (int i = 0; i < imgs.length; i++)
            imgs[i] = temp.getSubimage(i * B_WIDTH_DEFAULT, rowIndex * B_HEIGHT_DEFAULT, B_WIDTH_DEFAULT,
B_HEIGHT_DEFAULT);
    }
    public void draw(Graphics g) {
//  Renders the button on the screen
        g.drawImage(imgs[index], xPos - xOffsetCenter, yPos, B_WIDTH, B_HEIGHT, null);
    }
    public void update() {
//  Updates the button's state based on whether the mouse is hovering over or pressing it
        index = 0;
        if (mouseOver)
            index = 1;
        if (mousePressed)
            index = 2;
    }
    public boolean isMouseOver() {
//  Returns the status of the mouseOver
        return mouseOver;
    }
    public void setMouseOver(boolean mouseOver) {
//  Updates the mouseOver states
        this.mouseOver = mouseOver;
    }
    public boolean isMousePressed() {
//  Returns the status of the mousePressed flags
        return mousePressed;
    }
    public void setMousePressed(boolean mousePressed) {
//  Updates the mousePressed states
        this.mousePressed = mousePressed;
    }
    public Rectangle getBounds() {
//  Returns the hitbox (bounding rectangle) of the button.
        return bounds;
    }
    public void applyGamestate() {
//  Changes the current game state to the one associated with the button.
        Gamestate.state = state;
    }
    public void resetBools() {
//  Resets the button's flags (no longer hovering or being pressed)
        mouseOver = false;
        mousePressed = false;
    }
}
```

PauseOverlay

```
public class PauseOverlay {
    private Playing playing;
    private BufferedImage backgroundImg;
    private int bgX, bgY, bgW, bgH;
    private SoundButton musicButton, sfxButton;
    private UrmButton menuB, replayB, unpauseB;
    private VolumeButton volumeButton;
    // Command: Constructor initializes the PauseOverlay and sets up all components
    public PauseOverlay(Playing playing) {
        this.playing = playing;
        loadBackground();
        createSoundButtons();
        createUrmButtons();
        createVolumeButton();
    }
```

```java
    // Command: Create the volume button
    private void createVolumeButton() {
        int vX = (int) (309 * Game.SCALE);
        int vY = (int) (278 * Game.SCALE);
        volumeButton = new VolumeButton(vX, vY, SLIDER_WIDTH, VOLUME_HEIGHT);
    }
    // Command: Create URM buttons (Menu, Replay, Unpause)
    private void createUrmButtons() {
        int menuX = (int) (313 * Game.SCALE);
        int replayX = (int) (387 * Game.SCALE);
        int unpauseX = (int) (462 * Game.SCALE);
        int bY = (int) (325 * Game.SCALE);
        menuB = new UrmButton(menuX, bY, URM_SIZE, URM_SIZE, 2);
        replayB = new UrmButton(replayX, bY, URM_SIZE, URM_SIZE, 1);
        unpauseB = new UrmButton(unpauseX, bY, URM_SIZE, URM_SIZE, 0);
    }
    // Command: Create sound buttons (Music and SFX)
    private void createSoundButtons() {
        int soundX = (int) (450 * Game.SCALE);
        int musicY = (int) (140 * Game.SCALE);
        int sfxY = (int) (186 * Game.SCALE);
        musicButton = new SoundButton(soundX, musicY, SOUND_SIZE, SOUND_SIZE);
        sfxButton = new SoundButton(soundX, sfxY, SOUND_SIZE, SOUND_SIZE);
    }
    // Command: Load background image and set dimensions
    private void loadBackground() {
        backgroundImg = LoadSave.GetSpriteAtlas(LoadSave.PAUSE_BACKGROUND);
        bgW = (int) (backgroundImg.getWidth() * Game.SCALE);
        bgH = (int) (backgroundImg.getHeight() * Game.SCALE);
        bgX = Game.GAME_WIDTH / 2 - bgW / 2;
        bgY = (int) (25 * Game.SCALE);
    }
    // Command: Update all UI components
    public void update() {
        musicButton.update();
        sfxButton.update();
        menuB.update();
        replayB.update();
        unpauseB.update();
        volumeButton.update();
    }
    // Command: Draw all UI components on the screen
    public void draw(Graphics g) {
        // Background
        g.drawImage(backgroundImg, bgX, bgY, bgW, bgH, null);
        // Sound buttons
        musicButton.draw(g);
        sfxButton.draw(g);
        // UrmButtons
        menuB.draw(g);
        replayB.draw(g);
        unpauseB.draw(g);
        // Volume Button
        volumeButton.draw(g);
    }
    // Command: Handle mouse dragged event for the volume button
    public void mouseDragged(MouseEvent e) {
        if (volumeButton.isMousePressed()) {
            volumeButton.changeX(e.getX());
        }
    }
    // Command: Handle mouse pressed event for all UI components
    public void mousePressed(MouseEvent e) {
        if (isIn(e, musicButton))
            musicButton.setMousePressed(true);
        else if (isIn(e, sfxButton))
            sfxButton.setMousePressed(true);
        else if (isIn(e, menuB))
            menuB.setMousePressed(true);
        else if (isIn(e, replayB))
            replayB.setMousePressed(true);
        else if (isIn(e, unpauseB))
            unpauseB.setMousePressed(true);
        else if (isIn(e, volumeButton))
```

```
            volumeButton.setMousePressed(true);
    }
    // Command: Handle mouse released event and execute relevant actions
    public void mouseReleased(MouseEvent e) {
        if (isIn(e, musicButton)) {
            if (musicButton.isMousePressed())
                musicButton.setMuted(!musicButton.isMuted());
        } else if (isIn(e, sfxButton)) {
            if (sfxButton.isMousePressed())
                sfxButton.setMuted(!sfxButton.isMuted());
        } else if (isIn(e, menuB)) {
            if (menuB.isMousePressed()) {
                Gamestate.state = Gamestate.MENU;
                playing.unpauseGame();
            }
        } else if (isIn(e, replayB)) {
            if (replayB.isMousePressed()) {
                playing.resetAll();
                playing.unpauseGame();
            }
        } else if (isIn(e, unpauseB)) {
            if (unpauseB.isMousePressed())
                playing.unpauseGame();
        }
        musicButton.resetBools();
        sfxButton.resetBools();
        menuB.resetBools();
        replayB.resetBools();
        unpauseB.resetBools();
        volumeButton.resetBools();
    }
    // Command: Handle mouse moved event for hover effects
    public void mouseMoved(MouseEvent e) {
        musicButton.setMouseOver(false);
        sfxButton.setMouseOver(false);
        menuB.setMouseOver(false);
        replayB.setMouseOver(false);
        unpauseB.setMouseOver(false);
        volumeButton.setMouseOver(false);
        if (isIn(e, musicButton))
            musicButton.setMouseOver(true);
        else if (isIn(e, sfxButton))
            sfxButton.setMouseOver(true);
        else if (isIn(e, menuB))
            menuB.setMouseOver(true);
        else if (isIn(e, replayB))
            replayB.setMouseOver(true);
        else if (isIn(e, unpauseB))
            unpauseB.setMouseOver(true);
        else if (isIn(e, volumeButton))
            volumeButton.setMouseOver(true);
    }
    // Command: Check if a mouse event occurred within a button's bounds
    private boolean isIn(MouseEvent e, PauseButton b) {
        return b.getBounds().contains(e.getX(), e.getY());
    }
}
```

SoundButtons

```
public class PauseOverlay {
    private Playing playing;
    private BufferedImage backgroundImg;
    private int bgX, bgY, bgW, bgH;
    private SoundButton musicButton, sfxButton;
    private UrmButton menuB, replayB, unpauseB;
    private VolumeButton volumeButton;
 // Command: Constructor initializes the PauseOverlay and sets up all components
    public PauseOverlay(Playing playing) {
        this.playing = playing;
        loadBackground();
        createSoundButtons();
        createUrmButtons();
        createVolumeButton();
    }
```

```java
    // Command: Create the volume button
    private void createVolumeButton() {
        int vX = (int) (309 * Game.SCALE);
        int vY = (int) (278 * Game.SCALE);
        volumeButton = new VolumeButton(vX, vY, SLIDER_WIDTH, VOLUME_HEIGHT);
    }
    // Command: Create URM buttons (Menu, Replay, Unpause)
    private void createUrmButtons() {
        int menuX = (int) (313 * Game.SCALE);
        int replayX = (int) (387 * Game.SCALE);
        int unpauseX = (int) (462 * Game.SCALE);
        int bY = (int) (325 * Game.SCALE);
        menuB = new UrmButton(menuX, bY, URM_SIZE, URM_SIZE, 2);
        replayB = new UrmButton(replayX, bY, URM_SIZE, URM_SIZE, 1);
        unpauseB = new UrmButton(unpauseX, bY, URM_SIZE, URM_SIZE, 0);
    }
    // Command: Create sound buttons (Music and SFX)
    private void createSoundButtons() {
        int soundX = (int) (450 * Game.SCALE);
        int musicY = (int) (140 * Game.SCALE);
        int sfxY = (int) (186 * Game.SCALE);
        musicButton = new SoundButton(soundX, musicY, SOUND_SIZE, SOUND_SIZE);
        sfxButton = new SoundButton(soundX, sfxY, SOUND_SIZE, SOUND_SIZE);
    }
    // Command: Load background image and set dimensions
    private void loadBackground() {
        backgroundImg = LoadSave.GetSpriteAtlas(LoadSave.PAUSE_BACKGROUND);
        bgW = (int) (backgroundImg.getWidth() * Game.SCALE);
        bgH = (int) (backgroundImg.getHeight() * Game.SCALE);
        bgX = Game.GAME_WIDTH / 2 - bgW / 2;
        bgY = (int) (25 * Game.SCALE);
    }
    // Command: Update all UI components
    public void update() {
        musicButton.update();
        sfxButton.update();
        menuB.update();
        replayB.update();
        unpauseB.update();
        volumeButton.update();
    }
    // Command: Draw all UI components on the screen
    public void draw(Graphics g) {
        // Background
        g.drawImage(backgroundImg, bgX, bgY, bgW, bgH, null);
        // Sound buttons
        musicButton.draw(g);
        sfxButton.draw(g);
        // UrmButtons
        menuB.draw(g);
        replayB.draw(g);
        unpauseB.draw(g);
        // Volume Button
        volumeButton.draw(g);
    }
    // Command: Handle mouse dragged event for the volume button
    public void mouseDragged(MouseEvent e) {
        if (volumeButton.isMousePressed()) {
            volumeButton.changeX(e.getX());
        }
    }
    // Command: Handle mouse pressed event for all UI components
    public void mousePressed(MouseEvent e) {
        if (isIn(e, musicButton))
            musicButton.setMousePressed(true);
        else if (isIn(e, sfxButton))
            sfxButton.setMousePressed(true);
        else if (isIn(e, menuB))
            menuB.setMousePressed(true);
        else if (isIn(e, replayB))
            replayB.setMousePressed(true);
        else if (isIn(e, unpauseB))
            unpauseB.setMousePressed(true);
        else if (isIn(e, volumeButton))
```

```
            volumeButton.setMousePressed(true);
    }
    // Command: Handle mouse released event and execute relevant actions
    public void mouseReleased(MouseEvent e) {
        if (isIn(e, musicButton)) {
            if (musicButton.isMousePressed())
                musicButton.setMuted(!musicButton.isMuted());
        } else if (isIn(e, sfxButton)) {
            if (sfxButton.isMousePressed())
                sfxButton.setMuted(!sfxButton.isMuted());
        } else if (isIn(e, menuB)) {
            if (menuB.isMousePressed()) {
                Gamestate.state = Gamestate.MENU;
                playing.unpauseGame();
            }
        } else if (isIn(e, replayB)) {
            if (replayB.isMousePressed()) {
                playing.resetAll();
                playing.unpauseGame();
            }
        } else if (isIn(e, unpauseB)) {
            if (unpauseB.isMousePressed())
                playing.unpauseGame();
        }
        musicButton.resetBools();
        sfxButton.resetBools();
        menuB.resetBools();
        replayB.resetBools();
        unpauseB.resetBools();
        volumeButton.resetBools();
    }
    // Command: Handle mouse moved event for hover effects
    public void mouseMoved(MouseEvent e) {
        musicButton.setMouseOver(false);
        sfxButton.setMouseOver(false);
        menuB.setMouseOver(false);
        replayB.setMouseOver(false);
        unpauseB.setMouseOver(false);
        volumeButton.setMouseOver(false);
        if (isIn(e, musicButton))
            musicButton.setMouseOver(true);
        else if (isIn(e, sfxButton))
            sfxButton.setMouseOver(true);
        else if (isIn(e, menuB))
            menuB.setMouseOver(true);
        else if (isIn(e, replayB))
            replayB.setMouseOver(true);
        else if (isIn(e, unpauseB))
            unpauseB.setMouseOver(true);
        else if (isIn(e, volumeButton))
            volumeButton.setMouseOver(true);
    }
    // Command: Check if a mouse event occurred within a button's bounds
    private boolean isIn(MouseEvent e, PauseButton b) {
        return b.getBounds().contains(e.getX(), e.getY());
    }
}
```

Add SmallCloud and BigCloud into class LoadSave

```
// Command: Path to the image file for big clouds in the game's resources
public static final String BIG_CLOUDS = "big_clouds.png";
// Command: Path to the image file for small clouds in the game's resources
public static final String SMALL_CLOUDS = "small_clouds.png";
```

SmallCLoud and BigCloud

```
public static class Environment {
    // Command: Default width and height of big clouds (unscaled)
    public static final int BIG_CLOUD_WIDTH_DEFAULT = 448;
    public static final int BIG_CLOUD_HEIGHT_DEFAULT = 101;
    // Command: Default width and height of small clouds (unscaled)
    public static final int SMALL_CLOUD_WIDTH_DEFAULT = 74;
    public static final int SMALL_CLOUD_HEIGHT_DEFAULT = 24;
    // Command: Scaled width and height of big clouds based on game scale
```

```
    public static final int BIG_CLOUD_WIDTH = (int) (BIG_CLOUD_WIDTH_DEFAULT * Game.SCALE);
    public static final int BIG_CLOUD_HEIGHT = (int) (BIG_CLOUD_HEIGHT_DEFAULT * Game.SCALE);
    // Command: Scaled width and height of small clouds based on game scale
    public static final int SMALL_CLOUD_WIDTH = (int) (SMALL_CLOUD_WIDTH_DEFAULT * Game.SCALE);
    public static final int SMALL_CLOUD_HEIGHT = (int) (SMALL_CLOUD_HEIGHT_DEFAULT * Game.SCALE);
}
```

```
public Playing(Game game) {
    // Command: Calls the constructor of the parent class to initialize with the provided Game instance
    super(game);
    // Command: Initializes various classes used in the Playing state
    initClasses();
    // Command: Loads the background image for the playing state
    backgroundImg = LoadSave.GetSpriteAtlas(LoadSave.PLAYING_BG_IMG);
    // Command: Loads the sprite for big clouds from resources
    bigCloud = LoadSave.GetSpriteAtlas(LoadSave.BIG_CLOUDS);
    // Command: Loads the sprite for small clouds from resources
    smallCloud = LoadSave.GetSpriteAtlas(LoadSave.SMALL_CLOUDS);
    // Command: Initializes positions for small clouds
    smallCloudsPos = new int[8]; // Command: Creates an array to hold 8 small cloud positions
    for (int i = 0; i < smallCloudsPos.length; i++) {
        // Command: Assigns random positions to small clouds within a specified range
        smallCloudsPos[i] = (int) (90 * Game.SCALE) + rnd.nextInt((int) (100 * Game.SCALE));
    }
    // Command: Calculates the level offset for smooth level scrolling
    calcLvlOffset();
    // Command: Loads the starting level for the playing state
    loadStartLevel();
}
import static utilz.Constants.Environment.*;
public void draw(Graphics g) {
    // Command: Draw the main background image to fill the game screen
    g.drawImage(backgroundImg, 0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, null);
    // Command: Draw the clouds in the environment
    drawClouds(g);
    // Command: Render the level layout using the level manager and apply level offset
    levelManager.draw(g, xLvlOffset);
    // Command: Render the player character with the applied level offset
    player.render(g, xLvlOffset);
    // Command: Render enemies managed by the enemy manager with the applied level offset
    enemyManager.draw(g, xLvlOffset);
    // Command: Check the current game state to apply overlays
    if (paused) {
        // Command: Apply a semi-transparent black overlay to indicate the game is paused
        g.setColor(new Color(0, 0, 0, 150));
        g.fillRect(0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT);
        // Command: Draw the pause menu overlay
        pauseOverlay.draw(g);
    } else if (gameOver) {
        // Command: Draw the game over overlay if the game has ended
        gameOverOverlay.draw(g);
    } else if (lvlCompleted) {
        // Command: Draw the level completion overlay if the level is finished
        levelCompletedOverlay.draw(g);
    }
}
```

Cloud Movement

```
private void drawClouds(Graphics g) {
    // Command: Draw three big clouds that scroll slightly slower than the level offset
    for (int i = 0; i < 3; i++) {
        g.drawImage(bigCloud,
            i * BIG_CLOUD_WIDTH - (int) (xLvlOffset * 0.3),  // Command: Horizontal position adjusts with parallax
scrolling
            (int) (204 * Game.SCALE),                       // Command: Fixed vertical position adjusted by scale
            BIG_CLOUD_WIDTH, BIG_CLOUD_HEIGHT,              // Command: Cloud dimensions
            null);
    }
    // Command: Draw small clouds at varying vertical positions with faster parallax scrolling
    for (int i = 0; i < smallCloudsPos.length; i++) {
        g.drawImage(smallCloud,
            SMALL_CLOUD_WIDTH * 4 * i - (int) (xLvlOffset * 0.7),  // Command: Horizontal position adjusts with
faster parallax scrolling
```

```
            smallCloudsPos[i],                                  // Command: Randomized vertical positions
            SMALL_CLOUD_WIDTH, SMALL_CLOUD_HEIGHT,              // Command: Cloud dimensions
            null);
    }
}
```

NewVariables

```
private int xLvlOffset;
// Command: Variable to store the horizontal offset of the level, used for scrolling.
private int leftBorder = (int) (0.2 * Game.GAME_WIDTH);
// Command: Defines the left border of the screen as 20% of the total game width.
// Command: When the player moves beyond this point, the level starts scrolling left.
private int rightBorder = (int) (0.8 * Game.GAME_WIDTH);
// Command: Defines the right border of the screen as 80% of the total game width.
// Command: When the player moves beyond this point, the level starts scrolling right.
private int maxLvlOffsetX;
// Command: Stores the maximum allowable horizontal offset for the level.
// Command: Prevents the level from scrolling beyond its bounds.
```

checkCLoseToBorder

```
private void checkCloseToBorder() {
    int playerX = (int) player.getHitbox().x;
    // Command: Gets the current x-coordinate of the player's hitbox.
    int diff = playerX - xLvlOffset;
    // Command: Calculates the difference between the player's position and the level's offset.
    if (diff > rightBorder)
        xLvlOffset += diff - rightBorder;
    // Command: If the player is beyond the right border, increase the level offset to scroll right.
    else if (diff < leftBorder)
        xLvlOffset += diff - leftBorder;
    // Command: If the player is beyond the left border, decrease the level offset to scroll left.
    if (xLvlOffset > maxLvlOffsetX)
        xLvlOffset = maxLvlOffsetX;
    // Command: Ensure the level offset does not exceed the maximum level offset (prevent over-scrolling to the
right).
    else if (xLvlOffset < 0)
        xLvlOffset = 0;
    // Command: Ensure the level offset does not go below zero (prevent over-scrolling to the left).
}
@Override
public void draw(Graphics g) {
    g.drawImage(backgroundImg, 0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, null);
    // Command: Draws the background image, scaled to the game's width and height.
    drawClouds(g);
    // Command: Calls the method to draw the clouds in the background.
    levelManager.draw(g, xLvlOffset);
    // Command: Draws the current level, offset by `xLvlOffset` for scrolling.
    player.render(g, xLvlOffset);
    // Command: Renders the player, adjusting its position based on the level offset.
    enemyManager.draw(g, xLvlOffset);
    // Command: Draws the enemies, adjusted for the level offset.
    if (paused) {
        g.setColor(new Color(0, 0, 0, 150));
        // Command: Sets a semi-transparent black color for the pause overlay.
        g.fillRect(0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT);
        // Command: Draws a translucent rectangle over the entire game screen when paused.
        pauseOverlay.draw(g);
        // Command: Draws the pause overlay UI.
    } else if (gameOver)
        gameOverOverlay.draw(g);
        // Command: If the game is over, draw the game-over overlay.
    else if (lvlCompleted)
        levelCompletedOverlay.draw(g);
        // Command: If the level is completed, draw the level-completed overlay.
}
```

```
public void render(Graphics g, int lvlOffset) { // Method to render the player and UI components
    g.drawImage(animations[playerAction][aniIndex], // Draw the current player animation frame
        (int) (hitbox.x - xDrawOffset) - lvlOffset + flipX, // Calculate X position with offsets and flipping
        (int) (hitbox.y - yDrawOffset), // Calculate Y position with offsets
        width * flipW, // Calculate width with flipping
        height, // Set the height of the image
        null); // No ImageObserver required
```

```
//      drawHitbox(g, lvlOffset); // Uncomment to draw the player's hitbox for debugging
//      drawAttackBox(g, lvlOffset); // Uncomment to draw the player's attack box for debugging
    drawUI(g); // Draw the user interface components
}
```

```
public void draw(Graphics g, int lvlOffset) { // Method to draw the game level tiles
    for (int j = 0; j < Game.TILES_IN_HEIGHT; j++) // Loop through each row of tiles
        for (int i = 0; i < levels.get(lvlIndex).getLevelData()[0].length; i++) { // Loop through each column of
tiles
            int index = levels.get(lvlIndex).getSpriteIndex(i, j); // Get the sprite index for the current tile
            g.drawImage(levelSprite[index], // Draw the tile image based on the sprite index
                Game.TILES_SIZE * i - lvlOffset, // Calculate the X position of the tile with level offset
                Game.TILES_SIZE * j, // Calculate the Y position of the tile
                Game.TILES_SIZE, // Set the width of the tile
                Game.TILES_SIZE, // Set the height of the tile
                null); // No ImageObserver required
        }
}
```

In Constants class adding EnemyConstants

```
public class Constants { // Constants class to hold various game constants

    public static class EnemyConstants { // Inner class for enemy-specific constants  4 usages
        public static final int CRABBY = 0; // Enemy type identifier for Crabby  6 usages

        public static final int IDLE = 0; // Enemy state: Idle  4 usages
        public static final int RUNNING = 1; // Enemy state: Running  3 usages
        public static final int ATTACK = 2; // Enemy state: Attacking  4 usages
        public static final int HIT = 3; // Enemy state: Hit  4 usages
        public static final int DEAD = 4; // Enemy state: Dead  3 usages

        public static final int CRABBY_WIDTH_DEFAULT = 72; // Default Crabby width  3 usages
        public static final int CRABBY_HEIGHT_DEFAULT = 32; // Default Crabby height  3 usages

        public static final int CRABBY_WIDTH = (int) (CRABBY_WIDTH_DEFAULT * Game.SCALE); // Scaled Crabby width  2 usages
        public static final int CRABBY_HEIGHT = (int) (CRABBY_HEIGHT_DEFAULT * Game.SCALE); // Scaled Crabby height  2 usages

        public static final int CRABBY_DRAWOFFSET_X = (int) (26 * Game.SCALE); // X offset for Crabby drawing  1 usage
        public static final int CRABBY_DRAWOFFSET_Y = (int) (9 * Game.SCALE); // Y offset for Crabby drawing  1 usage

        public static int GetSpriteAmount(int enemy_type, int enemy_state) { // Method to get sprite count based on enemy type
            switch (enemy_type) {
                case CRABBY: // Case for Crabby enemy
                    switch (enemy_state) {
                        case IDLE:
                            return 9; // Sprite count for Idle state
                        case RUNNING:
                            return 6; // Sprite count for Running state
                        case ATTACK:
                            return 7; // Sprite count for Attack state
                        case HIT:
                            return 4; // Sprite count for Hit state
                        case DEAD:
                            return 5; // Sprite count for Dead state
                    }
```

Create Enemy Class

```
    public Enemy(float x, float y, int width, int height, int enemyType) { // Constructor to initialize the enemy  1 usage
        super(x, y, width, height); // Call the superclass constructor
        this.enemyType = enemyType; // Set the enemy type
        initHitbox(x, y, width, height); // Initialize the hitbox dimensions
        maxHealth = GetMaxHealth(enemyType); // Set the maximum health based on enemy type
        currentHealth = maxHealth; // Initialize current health to maximum health
    }

    protected void firstUpdateCheck(int[][] lvlData) { // Method to perform checks during the first update  1 usage
        if (!IsEntityOnFloor(hitbox, lvlData)) // Check if the entity is not on the floor
            inAir = true; // Set inAir flag to true
        firstUpdate = false; // Set firstUpdate flag to false
    }

    protected void updateInAir(int[][] lvlData) { // Method to update behavior when the enemy is in the air  1 usage
        if (CanMoveHere(hitbox.x, y: hitbox.y + fallSpeed, hitbox.width, hitbox.height, lvlData)) { // Check if the entity can
            hitbox.y += fallSpeed; // Apply fall speed to the Y-coordinate
            fallSpeed += gravity; // Increment fall speed by gravity
        } else { // If movement is not possible
            inAir = false; // Set inAir flag to false
            hitbox.y = GetEntityYPosUnderRoofOrAboveFloor(hitbox, fallSpeed); // Adjust Y-coordinate to floor or roof
            tileY = (int) (hitbox.y / Game.TILES_SIZE); // Update the tileY position
        }
    }
```

```java
protected void move(int[][] lvlData) { // Method to move the enemy  1 usage
    float xSpeed = 0; // Initialize xSpeed to 0

    if (walkDir == LEFT) // Check if the walking direction is LEFT
        xSpeed = -walkSpeed; // Set xSpeed to negative walk speed
    else // If the walking direction is RIGHT
        xSpeed = walkSpeed; // Set xSpeed to positive walk speed

    if (CanMoveHere( x: hitbox.x + xSpeed, hitbox.y, hitbox.width, hitbox.height, lvlData)) // Check if the enemy can move
        if (IsFloor(hitbox, xSpeed, lvlData)) { // Check if there is a floor to move on
            hitbox.x += xSpeed; // Apply xSpeed to the X-coordinate
            return; // Exit the method
        }

    changeWalkDir(); // Change the walking direction
}

protected void turnTowardsPlayer(Player player) { // Method to turn the enemy towards the player  1 usage
    if (player.hitbox.x > hitbox.x) // Check if the player is to the right
        walkDir = RIGHT; // Set walking direction to RIGHT
    else // If the player is to the left
        walkDir = LEFT; // Set walking direction to LEFT
}
```

```java
protected boolean canSeePlayer(int[][] lvlData, Player player) { // Method to check if the enemy can see the player  1 usa
    int playerTileY = (int) (player.getHitbox().y / Game.TILES_SIZE); // Get the player's Y position in tiles
    if (playerTileY == tileY) // Check if the player is on the same Y level
        if (isPlayerInRange(player)) { // Check if the player is in range
            if (IsSightClear(lvlData, hitbox, player.hitbox, tileY)) // Check if there is a clear line of sight
                return true; // Return true if the player can be seen
        }

    return false; // Return false if the player cannot be seen
}

protected boolean isPlayerInRange(Player player) { // Method to check if the player is in range  1 usage
    int absValue = (int) Math.abs(player.hitbox.x - hitbox.x); // Calculate the absolute distance to the player
    return absValue <= attackDistance * 5; // Return true if the player is within range
}

protected boolean isPlayerCloseForAttack(Player player) { // Method to check if the player is close for an attack  1 usage
    int absValue = (int) Math.abs(player.hitbox.x - hitbox.x); // Calculate the absolute distance to the player
    return absValue <= attackDistance; // Return true if the player is close enough
}

protected void newState(int enemyState) { // Method to change the enemy's state  5 usages
    this.enemyState = enemyState; // Set the new state
    aniTick = 0; // Reset animation tick
    aniIndex = 0; // Reset animation index
}
```

```java
public void hurt(int amount) { // Method to hurt the enemy  1 usage
    currentHealth -= amount; // Reduce current health by the given amount
    if (currentHealth <= 0) // Check if the enemy's health is zero or less
        newState(DEAD); // Set the state to DEAD
    else // If the enemy is still alive
        newState(HIT); // Set the state to HIT
}

protected void checkPlayerHit(Rectangle2D.Float attackBox, Player player) { // Method to check if the player is hit  1 usage
    if (attackBox.intersects(player.hitbox)) // Check if the attack box intersects the player's hitbox
        player.changeHealth(-GetEnemyDmg(enemyType)); // Reduce the player's health by the enemy's damage
    attackChecked = true; // Set the attackChecked flag to true
}

protected void updateAnimationTick() { // Method to update the animation tick  1 usage
    aniTick++; // Increment the animation tick
    if (aniTick >= aniSpeed) { // Check if the animation tick exceeds the speed
        aniTick = 0; // Reset the animation tick
        aniIndex++; // Increment the animation index
        if (aniIndex >= GetSpriteAmount(enemyType, enemyState)) { // Check if the animation index exceeds the sprite amount
            aniIndex = 0; // Reset the animation index

            switch (enemyState) { // Switch based on the current enemy state
                case ATTACK, HIT -> enemyState = IDLE; // Change to IDLE after ATTACK or HIT
                case DEAD -> active = false; // Deactivate the enemy if it is DEAD
            }
        }
    }
}
```

```
protected void changeWalkDir() { // Method to change the walking direction 1 usage
    if (walkDir == LEFT) // Check if the walking direction is LEFT
        walkDir = RIGHT; // Change to RIGHT
    else // If the walking direction is RIGHT
        walkDir = LEFT; // Change to LEFT
}

public void resetEnemy() { // Method to reset the enemy to its initial state 1 usage
    hitbox.x = x; // Reset X position
    hitbox.y = y; // Reset Y position
    firstUpdate = true; // Reset firstUpdate flag
    currentHealth = maxHealth; // Reset health to maximum
    newState(IDLE); // Set the state to IDLE
    active = true; // Reactivate the enemy
    fallSpeed = 0; // Reset fall speed
}
```

Creating Crabby Class which extends the Enemy Class

```
private void initAttackBox() { // Method to initialize the attack box 1 usage
    attackBox = new Rectangle2D.Float(x, y, (int) (82 * Game.SCALE), (int) (19 * Game.SCALE)); // Set the attack box dimensi
    attackBoxOffsetX = (int) (Game.SCALE * 30); // Set the horizontal offset for the attack box
}

public void update(int[][] lvlData, Player player) { // Method to update Crabby's state
    updateBehavior(lvlData, player); // Update Crabby's behavior based on level data and player position
    updateAnimationTick(); // Update the animation frame
    updateAttackBox(); // Update the position of the attack box
}

private void updateAttackBox() { // Method to update the attack box position 1 usage
    attackBox.x = hitbox.x - attackBoxOffsetX; // Set the x-coordinate of the attack box
    attackBox.y = hitbox.y; // Set the y-coordinate of the attack box
}
```

```
private void updateBehavior(int[][] lvlData, Player player) { // Method to update Crabby's behavior 1 usage
    if (firstUpdate) // Check if this is the first update
        firstUpdateCheck(lvlData); // Perform initial update checks

    if (inAir) // Check if Crabby is in the air
        updateInAir(lvlData); // Update behavior while in the air
    else { // If Crabby is on the ground
        switch (enemyState) { // Switch based on the enemy's current state
            case IDLE: // If the state is IDLE
                newState(RUNNING); // Change state to RUNNING
                break;
            case RUNNING: // If the state is RUNNING
                if (canSeePlayer(lvlData, player)) { // Check if Crabby can see the player
                    turnTowardsPlayer(player); // Turn towards the player
                    if (isPlayerCloseForAttack(player)) // Check if the player is close enough for an attack
                        newState(ATTACK); // Change state to ATTACK
                }
                move(lvlData); // Move Crabby based on level data
                break;
            case ATTACK: // If the state is ATTACK
                if (aniIndex == 0) // Check if the animation index is at the start
                    attackChecked = false; // Reset the attack check flag
                if (aniIndex == 3 && !attackChecked) // Check if it's the attack frame and attack hasn't been checked
                    checkPlayerHit(attackBox, player); // Check if the attack hits the player
                break;
            case HIT: // If the state is HIT
                break; // Do nothing
        }
    }
}

public void drawAttackBox(Graphics g, int xLvlOffset) { // Method to draw the attack box for debugging no usages
    g.setColor(Color.red); // Set the drawing color to red
    g.drawRect((int) (attackBox.x - xLvlOffset), (int) attackBox.y, (int) attackBox.width, (int) attackBox.height); // Draw
}
```

```
    public int flipX() { // Method to get the flip value for x-coordinate 1 usage
        if (walkDir == RIGHT) // Check if walking direction is RIGHT
            return width; // Return the width value
        else // If walking direction is LEFT
            return 0; // Return 0
    }

    public int flipW() { // Method to get the flip value for width 1 usage
        if (walkDir == RIGHT) // Check if walking direction is RIGHT
            return -1; // Return -1 to flip the width
        else // If walking direction is LEFT
            return 1; // Return 1 to keep the width normal
    }
}
```

Creating the EnemyManager Class

```java
public class EnemyManager { // Define the EnemyManager class 4 usages

    private Playing playing; // Reference to the Playing game state 2 usages
    private BufferedImage[][] crabbyArr; // Array to hold Crabby enemy sprites 5 usages
    private ArrayList<Crabby> crabbies = new ArrayList<>(); // List to store Crabby enemies 5 usages

    public EnemyManager(Playing playing) { // Constructor to initialize the EnemyManager 1 usage
        this.playing = playing; // Assign the Playing instance
        loadEnemyImgs(); // Load enemy images
    }

    public void loadEnemies(Level level) { // Method to load enemies from a level 2 usages
        crabbies = level.getCrabs(); // Get Crabby enemies from the level
    }

    public void update(int[][] lvlData, Player player) { // Method to update all enemies
        boolean isAnyActive = false; // Flag to track if any enemies are active
        for (Crabby c : crabbies) // Loop through all Crabby enemies
            if (c.isActive()) { // Check if the enemy is active
                c.update(lvlData, player); // Update the enemy
                isAnyActive = true; // Set the flag to true
            }
        if (!isAnyActive) // Check if no enemies are active
            playing.setLevelCompleted(true); // Mark the level as completed
    }

    public void draw(Graphics g, int xLvlOffset) { // Method to draw all enemies
        drawCrabs(g, xLvlOffset); // Call the method to draw Crabby enemies
    }
```

```java
    private void drawCrabs(Graphics g, int xLvlOffset) { // Method to draw Crabby enemies 1 usage
        for (Crabby c : crabbies) // Loop through all Crabby enemies
            if (c.isActive()) { // Check if the enemy is active
                g.drawImage(
                        crabbyArr[c.getEnemyState()][c.getAniIndex()], // Get the current sprite of the enemy
                        (int) c.getHitbox().x - xLvlOffset - CRABBY_DRAWOFFSET_X + c.flipX(), // Calculate X position
                        (int) c.getHitbox().y - CRABBY_DRAWOFFSET_Y, // Calculate Y position
                        CRABBY_WIDTH * c.flipW(), // Calculate width
                        CRABBY_HEIGHT, // Set height
                        null // No image observer
                );
//              c.drawHitbox(g, xLvlOffset); // (Optional) Draw the hitbox for debugging
//              c.drawAttackBox(g, xLvlOffset); // (Optional) Draw the attack box for debugging
            }
    }

    public void checkEnemyHit(Rectangle2D.Float attackBox) { // Method to check if an enemy is hit 1 usage
        for (Crabby c : crabbies) // Loop through all Crabby enemies
            if (c.isActive()) // Check if the enemy is active
                if (attackBox.intersects(c.getHitbox())) { // Check if the attack box intersects the enemy's hitbox
                    c.hurt( amount: 10); // Hurt the enemy by 10 points
                    return; // Exit the method after hitting an enemy
                }
    }

    private void loadEnemyImgs() { // Method to load enemy images 1 usage
        crabbyArr = new BufferedImage[5][9]; // Initialize the array for Crabby sprites
        BufferedImage temp = LoadSave.GetSpriteAtlas(LoadSave.CRABBY_SPRITE); // Load the sprite atlas
        for (int j = 0; j < crabbyArr.length; j++) // Loop through rows of sprites
            for (int i = 0; i < crabbyArr[j].length; i++) // Loop through columns of sprites
                crabbyArr[j][i] = temp.getSubimage( // Extract individual sprites
                        i * CRABBY_WIDTH_DEFAULT, // X position in the sprite sheet
                        j * CRABBY_HEIGHT_DEFAULT, // Y position in the sprite sheet
                        CRABBY_WIDTH_DEFAULT, // Width of the sprite
                        CRABBY_HEIGHT_DEFAULT // Height of the sprite
                );
    }
```

```java
    public void resetAllEnemies() { // Method to reset all enemies to their initial state 1 usage
        for (Crabby c : crabbies) // Loop through all Crabby enemies
            c.resetEnemy(); // Reset the enemy
    }

}
```

The HelperMethods class provides many important methods for upper classes.

```java
public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) { 4 usages
    // Check if an entity can move to the specified area
    if (!IsSolid(x, y, lvlData)) // Check top-left corner
        if (!IsSolid( x: x + width,  y: y + height, lvlData)) // Check bottom-right corner
            if (!IsSolid( x: x + width, y, lvlData)) // Check top-right corner
                if (!IsSolid(x,  y: y + height, lvlData)) // Check bottom-left corner
                    return true; // Return true if none are solid
    return false; // Return false if any are solid
}

@   private static boolean IsSolid(float x, float y, int[][] lvlData) { 8 usages
    // Check if a point is solid
    int maxWidth = lvlData[0].length * Game.TILES_SIZE; // Calculate maximum width
    if (x < 0 || x >= maxWidth) // Check if x is out of bounds
        return true; // Return true for solid
    if (y < 0 || y >= Game.GAME_HEIGHT) // Check if y is out of bounds
        return true; // Return true for solid
    float xIndex = x / Game.TILES_SIZE; // Get x tile index
    float yIndex = y / Game.TILES_SIZE; // Get y tile index

    return IsTileSolid((int) xIndex, (int) yIndex, lvlData); // Check if the tile is solid
}

@   public static boolean IsTileSolid(int xTile, int yTile, int[][] lvlData) { 3 usages
    // Check if a specific tile is solid
    int value = lvlData[yTile][xTile]; // Get tile value

    if (value >= 48 || value < 0 || value != 11) // Check if value indicates solid
        return true; // Return true for solid
    return false; // Return false if not solid
}
```

```java
@   public static float GetEntityXPosNextToWall(Rectangle2D.Float hitbox, float xSpeed) { 1 usage
    // Calculate the entity's X position next to a wall
    int currentTile = (int) (hitbox.x / Game.TILES_SIZE); // Get the current tile index
    if (xSpeed > 0) { // Check if moving right
        int tileXPos = currentTile * Game.TILES_SIZE; // Calculate tile X position
        int xOffset = (int) (Game.TILES_SIZE - hitbox.width); // Calculate offset for width
        return tileXPos + xOffset - 1; // Return adjusted X position
    } else // If moving left
        return currentTile * Game.TILES_SIZE; // Return current tile X position
}

@   public static float GetEntityYPosUnderRoofOrAboveFloor(Rectangle2D.Float hitbox, float airSpeed) { 2 usages
    // Calculate the entity's Y position under a roof or above the floor
    int currentTile = (int) (hitbox.y / Game.TILES_SIZE); // Get current tile index
    if (airSpeed > 0) { // Check if falling
        int tileYPos = currentTile * Game.TILES_SIZE; // Calculate tile Y position
        int yOffset = (int) (Game.TILES_SIZE - hitbox.height); // Calculate offset for height
        return tileYPos + yOffset - 1; // Return adjusted Y position
    } else // If jumping
        return currentTile * Game.TILES_SIZE; // Return current tile Y position
}

@   public static boolean IsEntityOnFloor(Rectangle2D.Float hitbox, int[][] lvlData) { 4 usages
    // Check if the entity is on the floor
    if (!IsSolid(hitbox.x,  y: hitbox.y + hitbox.height + 1, lvlData)) // Check bottom-left corner
        if (!IsSolid( x: hitbox.x + hitbox.width,  y: hitbox.y + hitbox.height + 1, lvlData)) // Check bottom-right corne
            return false; // Return false if neither are solid
    return true; // Return true if at least one is solid
}

public static boolean IsFloor(Rectangle2D.Float hitbox, float xSpeed, int[][] lvlData) { 1 usage
    // Check if the entity has floor beneath
    if (xSpeed > 0) // Check if moving right
        return IsSolid( x: hitbox.x + hitbox.width + xSpeed,  y: hitbox.y + hitbox.height + 1, lvlData); // Check right s
    else // If moving left
        return IsSolid( x: hitbox.x + xSpeed,  y: hitbox.y + hitbox.height + 1, lvlData); // Check left side
}
```

```java
public static boolean IsAllTilesWalkable(int xStart, int xEnd, int y, int[][] lvlData) { 2 usages
    // Check if all tiles in a range are walkable
    for (int i = 0; i < xEnd - xStart; i++) { // Loop through tiles
        if (IsTileSolid( xTile: xStart + i, y, lvlData)) // Check if tile is solid
            return false; // Return false if solid
        if (!IsTileSolid( xTile: xStart + i, yTile: y + 1, lvlData)) // Check if tile below is not solid
            return false; // Return false if not solid
    }
    return true; // Return true if all tiles are walkable
}

public static boolean IsSightClear(int[][] lvlData, Rectangle2D.Float firstHitbox, Rectangle2D.Float secondHitbox, int
    // Check if sight is clear between two hitboxes
    int firstXTile = (int) (firstHitbox.x / Game.TILES_SIZE); // Get first hitbox's tile index
    int secondXTile = (int) (secondHitbox.x / Game.TILES_SIZE); // Get second hitbox's tile index

    if (firstXTile > secondXTile) // Check if first is after second
        return IsAllTilesWalkable(secondXTile, firstXTile, yTile, lvlData); // Check all tiles in range
    else // If first is before or equal
        return IsAllTilesWalkable(firstXTile, secondXTile, yTile, lvlData); // Check all tiles in range
}

public static int[][] GetLevelData(BufferedImage img) { 2 usages
    // Generate level data from an image
    int[][] lvlData = new int[img.getHeight()][img.getWidth()]; // Initialize level data array
    for (int j = 0; j < img.getHeight(); j++) // Loop through rows
        for (int i = 0; i < img.getWidth(); i++) { // Loop through columns
            Color color = new Color(img.getRGB(i, j)); // Get color of pixel
            int value = color.getRed(); // Extract red channel value
            if (value >= 48) // Check if value exceeds threshold
                value = 0; // Reset value to 0
            lvlData[j][i] = value; // Assign value to level data
        }
    return lvlData; // Return level data
}
```

```java
public static ArrayList<Crabby> GetCrabs(BufferedImage img) { 2 usages
    // Generate a list of Crabby enemies from an image
    ArrayList<Crabby> list = new ArrayList<>(); // Initialize list of Crabby
    for (int j = 0; j < img.getHeight(); j++) // Loop through rows
        for (int i = 0; i < img.getWidth(); i++) { // Loop through columns
            Color color = new Color(img.getRGB(i, j)); // Get color of pixel
            int value = color.getGreen(); // Extract green channel value
            if (value == CRABBY) // Check if value matches Crabby constant
                list.add(new Crabby( x: i * Game.TILES_SIZE, y: j * Game.TILES_SIZE)); // Add Crabby to list
        }
    return list; // Return list of Crabby
}

public static Point GetPlayerSpawn(BufferedImage img) { 2 usages
    // Get player spawn point from an image
    for (int j = 0; j < img.getHeight(); j++) // Loop through rows
        for (int i = 0; i < img.getWidth(); i++) { // Loop through columns
            Color color = new Color(img.getRGB(i, j)); // Get color of pixel
            int value = color.getGreen(); // Extract green channel value
            if (value == 100) // Check if value matches player spawn constant
                return new Point( x: i * Game.TILES_SIZE, y: j * Game.TILES_SIZE); // Return spawn point
        }
    return new Point( x: 1 * Game.TILES_SIZE, y: 1 * Game.TILES_SIZE); // Default spawn point
}

} // End of HelpMethods class
```

CHAP 4: FINAL APP GAME:

4.1 Source code (link github):

https://github.com/dangnguyengroup23/The-Last-Pirate

4.2 Demo video:

https://youtu.be/vSUhk6iifRI

4.3 Instruction:

a. Begin the game:

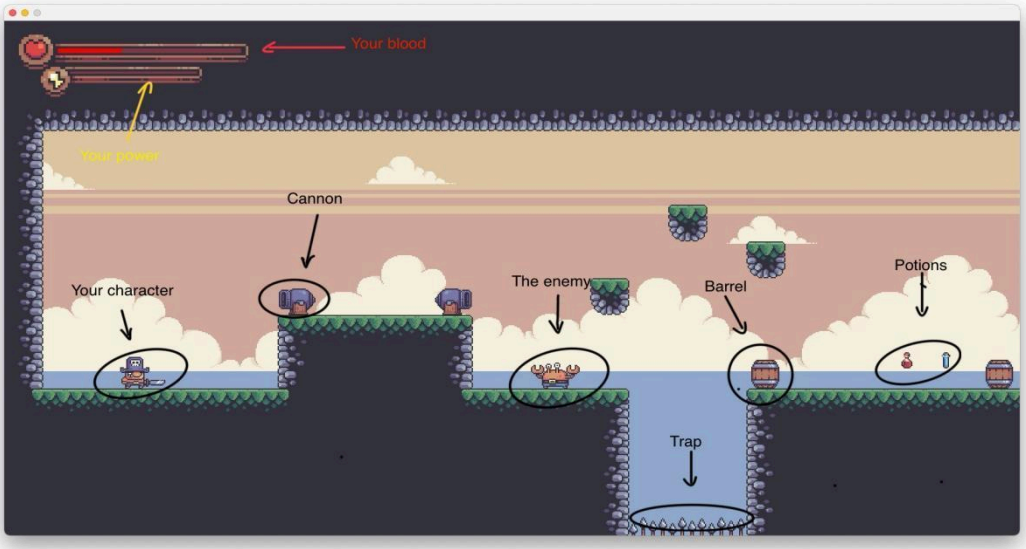
When you start the game, it will show you the Main Menu desktop.

b. Main Menu

Click the button **Play** on the **Menu** to start the game

Then you can see the game scene.

c. How to play:



You can control your character using your keyboard like:

- Button D: go to the right.
- Button A: go to the left.
- Button W: go up.
- Button S: go down.

And you can right click to attack.

d. Move to the next level.

When you killed all the enemy, you completed the level and move to the next .

Right click the **Green** button to continue or **Red** button to back to the **Main Menu.**

And so on the difficulty will be gradually increased to challenge you.

e. Game over and play again:

When you are killed by enemies, cannons, or fall into a trap, the game is over. And it will show you that:

Right click the **Green** button to try again or **Red** button to back to the **Main Menu.**

CHAP 5: CONCLUSION