INTERNATIONAL UNIVERSITY - HCMIU

SCHOOL OF COMPUTER SCIENCE

AND ENGINEERING

OBJECT-ORIENTED PROGRAMMING

Semester 1, 2023 - 2024

Instructor: Dr. Tran Thanh Tung

Topic: Platform Game 2D

Adventure To The Fallen Kingdom

NAME	STUDENT ID	CONTRIBUTION
Nguyen Quang Sang	ITDSIU21113	20%
Van Phu Minh Sang	ITDSIU21112	20%
Phan Manh Son	ITDSIU21116	20%
Tran Quynh Nhu	ITDSIU21105	20%
Duong Khanh Trang	ITDSIU21127	20%

Table Of Contents

I. BACKGROUND

1. Introduction

The 2D Platformer Pixel game created for the first semester of Object-Oriented Programming at VNUHCM - International University (2022-2023) is designed to be a captivating and immersive experience. Drawing inspiration from the timeless appeal of the 2D platformer genre, the game offers a nostalgic yet engaging gameplay style.

2. Members & Task allocation

Number	Student Name	Task
1	Duong Khanh Trang	Entities, Collision, Utilz, UML
2	Tran Quynh Nhu	Inputs, Sound, Object, Utilz, UML
3	Phan Manh Son	Objects, Levels, Maps, Utilz, UML
4	Nguyen Quang Sang	Background, Entities, Utilz, UML
5	Van Phu Minh Sang	Game States, GUI, Utilz, UML

3. Motivation To Reality

Our objective was to create a challenging 2D pixel game that would attract fans of classic games. We aimed to assess players' determination and abilities while inspiring them with a feeling of noteworthy achievements. By incorporating intense combat, a well-rounded gameplay system, and a dark fantasy backdrop, the game captures the difficulty and ominous atmosphere found in the Dark Souls series. The game's difficulty arises from the fact that every decision made can influence the outcome, emphasizing the value of perseverance.

4. Technologies

- Language: Java
- IDEs: VSCode, Eclipse
- Library: JavaSwing

- Packages: java.awt.image, java.awt.Graphics, java.awt.geom.Rectangle2D, java.awt.Color, java.util.ArrayList, java.awt.Point, java.awt.event.MouseEvent, java.awt.event.KeyEvent, java.util.Random, java.util.ArrayList, javax.swing.JPanel, java.awt.Dimension
- Game Engine: Java2D
- Sound: JavaSound

II. ABOUT THE GAME

1. Game Mechanics

Adventure to the Fallen Kingdom offers an immersive experience that transports players to a captivating world filled with pixelated wonders. As you embark on your journey, the game presents you with the grand ambition of becoming a renowned knight. The three different levels you'll encounter are thoughtfully crafted, each boasting unique pixel graphics that add a touch of nostalgia and charm.

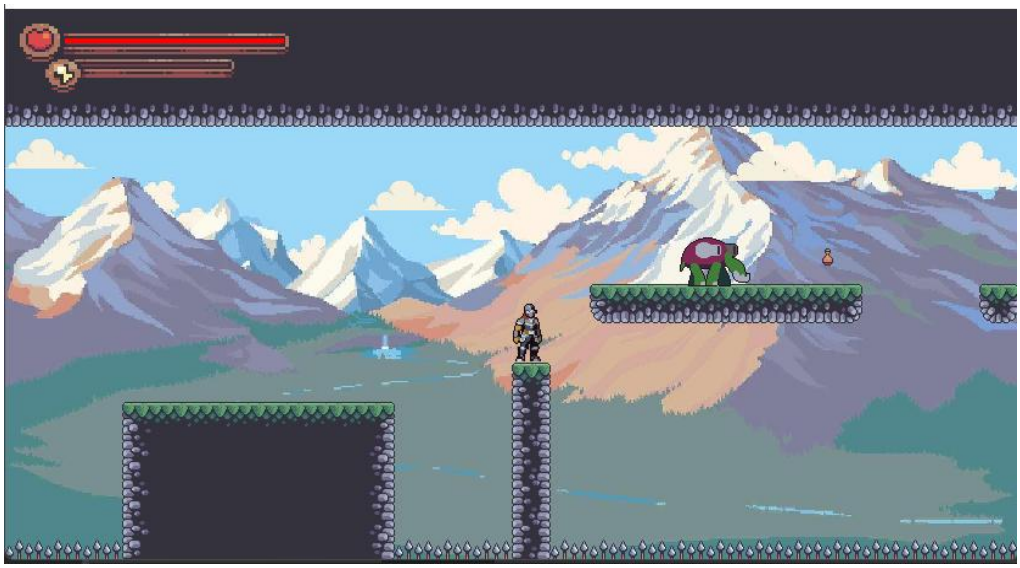
Accompanying your exploration is a carefully curated selection of calming classical music, which enhances the overall atmosphere and sets the tone for your adventure. It creates a sense of tranquility and focus, allowing players to fully immerse themselves in the game's world.

However, your path to knighthood is not an easy one. The fallen kingdom is plagued by horrifying creatures that threaten the safety of humanity. It is your duty to confront and overcome these menacing foes, displaying your courage and skill in the face of danger.

Assuming the role of the chosen knight, you will undergo a transformative journey that goes beyond physical prowess. Adventure to the Fallen Kingdom encourages you to delve deep within yourself to discover your untapped potential and embrace your inner talent and bravery. It is through conquering

obstacles and emerging victorious that you will establish your worth as a respected and honorable knight.

Level 1



Level 2

Level 3

2. How To Play

Key	Action
A	Move Left
D	Move Right
Space	Jump

- Use A, D, Space to move the knight.
- Use Left Click Mouse to attack the enemies.
- Crack up boxes to discover mana or healing pills.

3. Design & Capabilities

To manipulate the colors of pixels in our game, we utilize the BufferedImage class from the java.awt.image package in Java. This class acts as a representation of image files and allows us to work with individual pixels. In our game, each pixel represents a tile, entity, or object, and their distinct characteristics are determined by the red, green, and blue channels of the image.

The Level class plays a vital role in loading level data from the BufferedImage. It employs methods like loadLevelData, loadEntities, and loadObjects to analyze the colors of pixels and generate corresponding game elements. By leveraging this technique, we optimize development time and effort by sourcing game resources such as sounds, music, and images from Google.

Several exciting updates have been implemented in the game. It includes keyboard-based character control, a slashing technique for attacking the map by colliding with tiles, a tutorial at the first level, challenging bosses at the final level, an energy system for jumping, and potion-based recharging.

To handle different game modes and transitions, we adopted the State pattern. This design pattern allows us to have separate classes for each state and employs a Gamestate class to assign tasks to the current state object. This approach ensures flexibility, maintainability, and readability of the codebase.

Additionally, we adhere to the Single Responsibility principle by creating a Constants class, which organizes all game constants in a centralized manner. This design principle promotes code readability and maintainability by assigning distinct purposes to each class. Moreover, the Constants class categorizes constants based on their respective categories, such as Projectiles and UI.

III. FOUR CONCEPTS OF OOP

1. Encapsulation

In our game project, we have used classes like Player, Game, Enemy (including 3 Enemies class respectively for 3 level game), ... etc to provide encapsulation. Certain methods and properties are encapsulated in each class and kept secret from other classes. The Enemy class, for instance, has methods like isPlayerInRange, ...etc and the attributes like walkSpeed, maxHealth, currentHealth, etc. To allow for safe access and modification, getters and setters are employed. This encourages control and structure in the handling of the game's components.

2. Abstraction

By employing abstract classes and interfaces, we managed to identify shared characteristics between game components. The StateMethod interface ensured that different game states would function uniformly (e.g., Playing, Menu, and GameOption are examples of classes in the GameStates packages that implement the Static Methods interface). Abstract classes like Enemy and Entity, which encapsulated fundamental behavior common by all enemies, encouraged code reuse. Abstraction improved modularity and maintainability, facilitating more efficient feature sharing and easier development.

3. Inheritance

By using Inheritance to build subclasses, we were able to establish a hierarchical relationship and reuse code. We overriding particular procedures to alter the actions of Enemies such as Carnivorous, Big Bloated, and Turtle. In a similar vein, we added subclasses from the GameObject class, such as Potion, ObjectContainer, and Spike, to expand the functionality of objects. We did the same thing with the PauseButton class in the UI packages, extending its capabilities with subclasses like the SoundButton, UrmButton, and VolumeButton. By encouraging modular programming and minimizing code duplication, inheritance streamlined development and maintenance. It improved flexibility and customization by making it easier to create specialized subclasses that expanded upon the methods and attributes of parent classes.

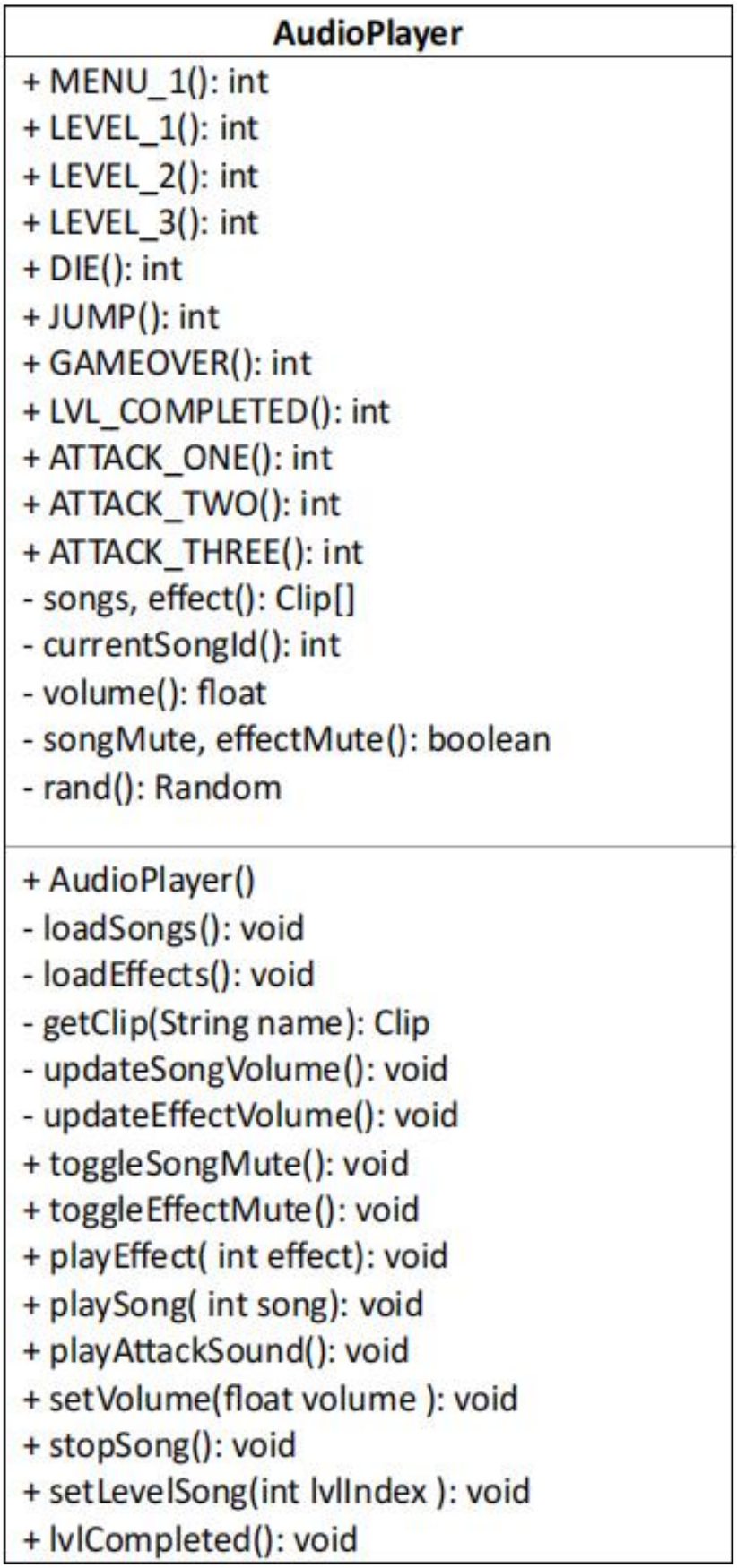
4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. This is often achieved through method overriding, where subclasses provide a specific implementation of a method that is already defined in their superclass. In the project, the concept of polymorphism is utilized through method overriding in the subclasses of different enemy types (Carnivorous, Turtle, Big_Bloated). Specifically, the update() method is overridden in these subclasses to provide their specific update behavior.

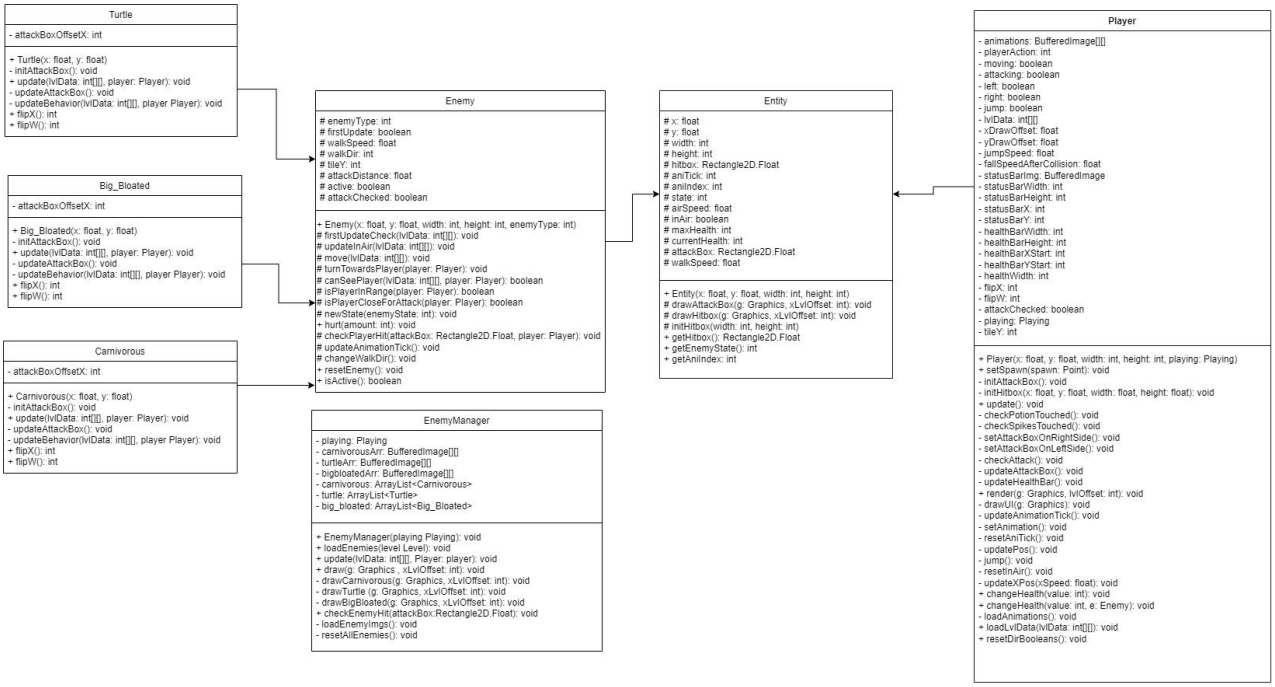
5. Others

- **OOP Project Presentation**
- **UML Diagram**

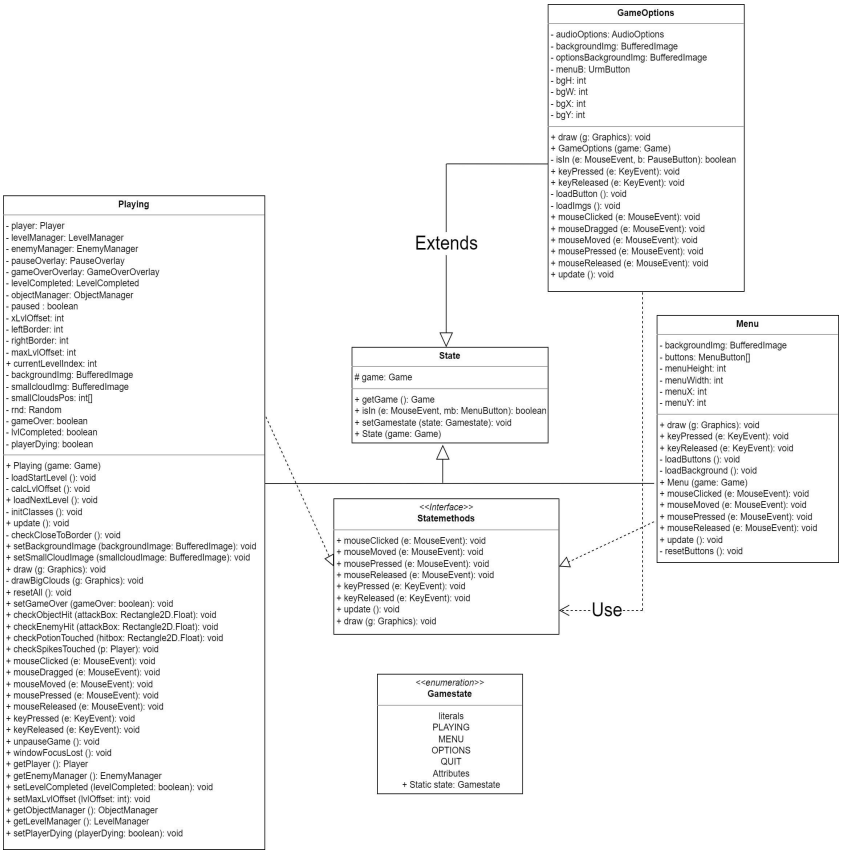
Audio Diagram



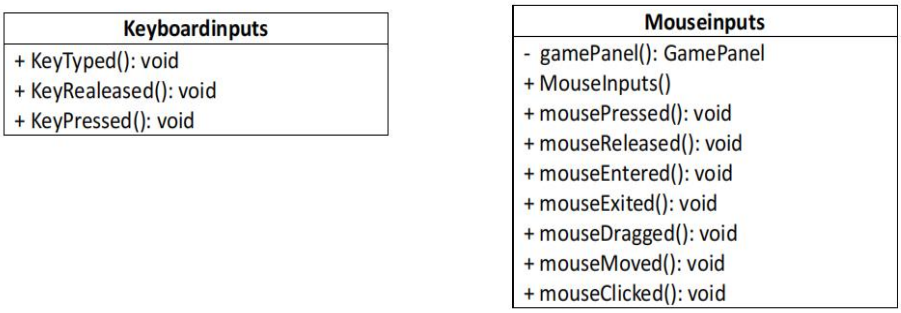
Entities Diagram



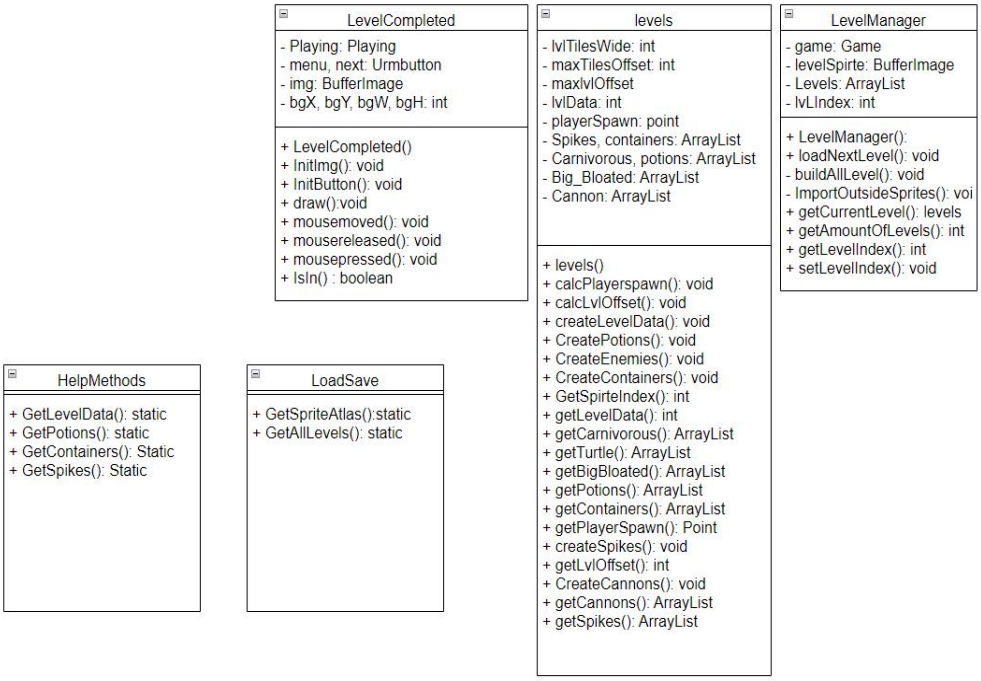
Game states Diagram



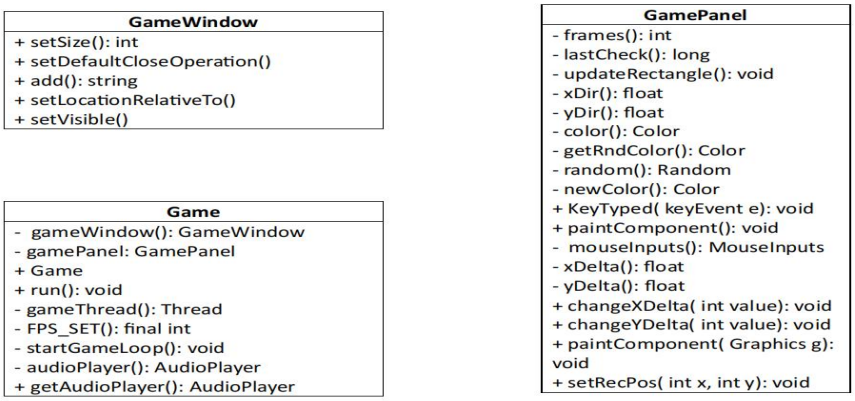
Inputs Diagram



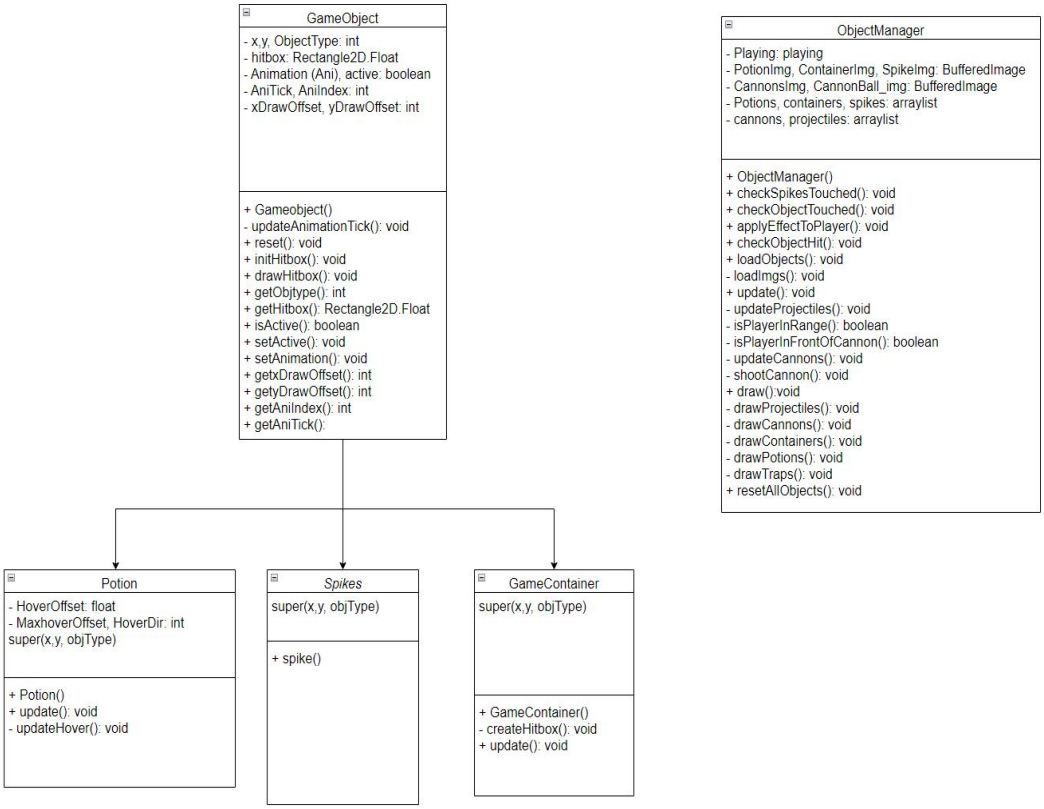
Levels Diagram



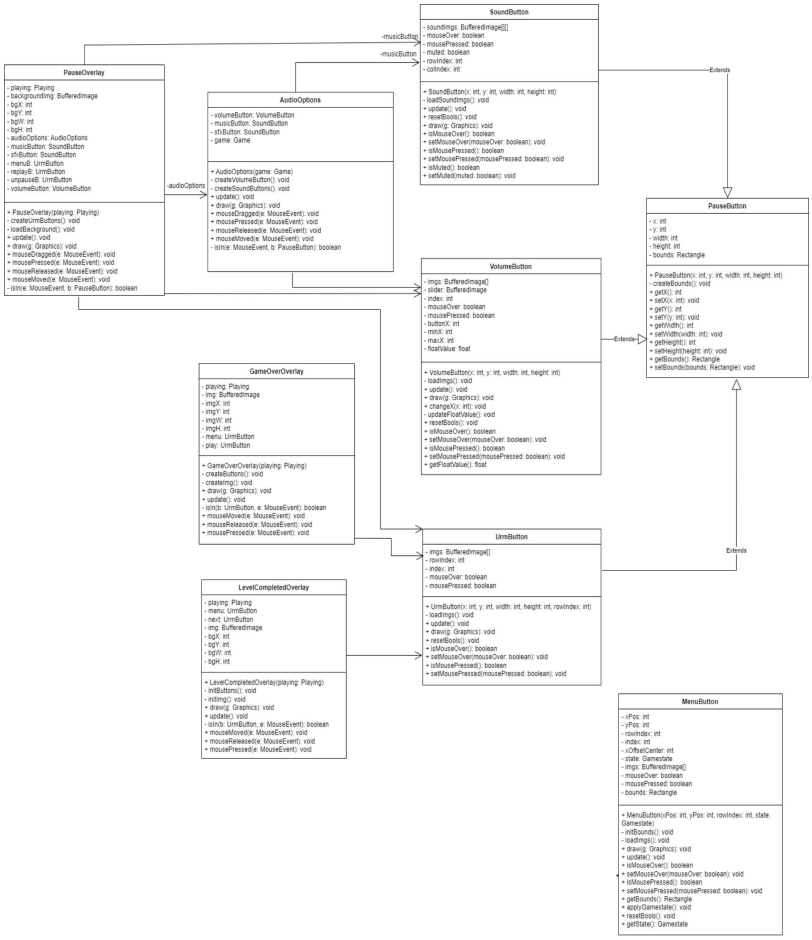
Main Diagram



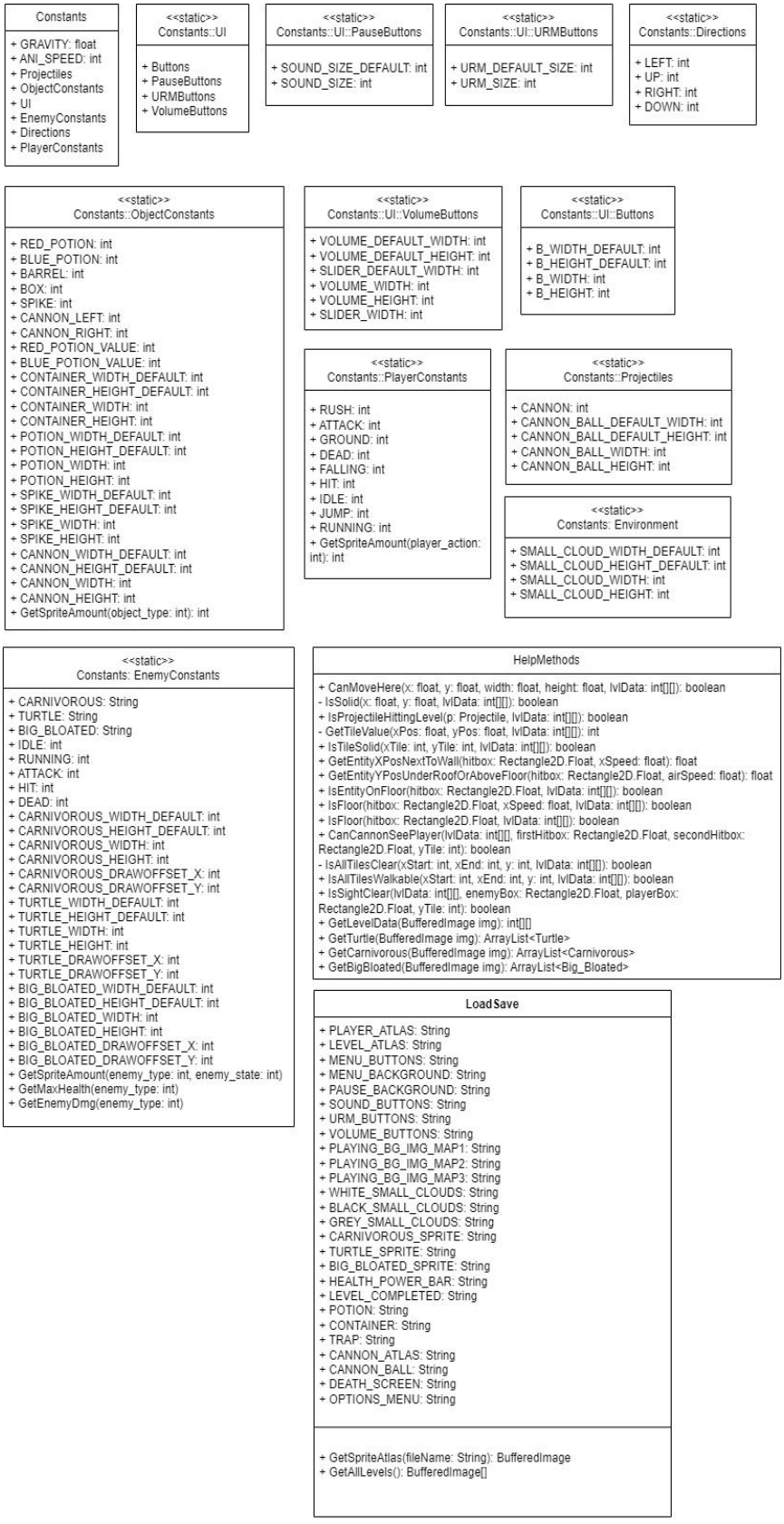
Objects Diagram



GUI Diagram


SoundButton
- soundImgs: BufferedImage[][]
- mouseOver: boolean
- mousePressed: boolean
- muted: boolean
- rowIndex: int
- colIndex: int
+ SoundButton(x: int, y: int, width: int, height: int)
- loadSoundImgs(): void
+ update(): void
+ resetBools(): void
+ draw(g: Graphics): void
+ isMouseOver(): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(): boolean
+ setMousePressed(mousePressed: boolean): void
+ isMuted(): boolean
+ setMuted(muted: boolean): void
-musicButton
-musicButton
Extends
PauseOverlay
- playing: Playing
- backgroundImg: BufferedImage
- bgX: int
- bgY: int
- bgW: int
- bgH: int
- audioOptions: AudioOptions
- musicButton: SoundButton
- sfxButton: SoundButton
- menuB: UrmButton
- replayB: UrmButton
- unpauseB: UrmButton
- volumeButton: VolumeButton
+ PauseOverlay(playing: Playing)
- createUrmButtons(): void
- loadBackground(): void
+ update(): void
+ draw(g: Graphics): void
+ mouseDragged(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mouseMoved(e: MouseEvent): void
- isIn(e: MouseEvent, b: PauseButton): boolean
audioOptions
AudioOptions
- volumeButton: VolumeButton
- musicButton: SoundButton
- sfxButton: SoundButton
- game: Game
+ AudioOptions(game: Game)
- createVolumeButton(): void
- createSoundButtons(): void
+ update(): void
+ draw(g: Graphics): void
+ mouseDragged(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mouseMoved(e: MouseEvent): void
- isIn(e: MouseEvent, b: PauseButton): boolean
PauseButton
- x: int
- y: int
- width: int
- height: int
- bounds: Rectangle
+ PauseButton(x: int, y: int, width: int, height: int)
- createBounds(): void
+ getX(): int
+ setX(x: int): void
+ getY(): int
+ setY(y: int): void
+ getWidth(): int
+ setWidth(width: int): void
+ getHeight(): int
+ setHeight(height: int): void
+ getBounds(): Rectangle
+ setBounds(bounds: Rectangle): void
VolumeButton
- imgs: BufferedImage[]
- slider: BufferedImage
- index: int
- mouseOver: boolean
- mousePressed: boolean
- buttonX: int
- minX: int
- maxX: int
- floatValue: float
+ VolumeButton(x: int, y: int, width: int, height: int)
- loadImgs(): void
+ update(): void
+ draw(g: Graphics): void
+ changeX(x: int): void
- updateFloatValue(): void
+ resetBools(): void
+ isMouseOver(): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(): boolean
+ setMousePressed(mousePressed: boolean): void
+ getFloatValue(): float
Extends
GameOverOverlay
- playing: Playing
- img: BufferedImage
- imgX: int
- imgY: int
- imgW: int
- imgH: int
- menu: UrmButton
- play: UrmButton
+ GameOverOverlay(playing: Playing)
- createButtons(): void
- createImg(): void
+ draw(g: Graphics): void
+ update(): void
- isIn(b: UrmButton, e: MouseEvent): boolean
+ mouseMoved(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
UrmButton
- imgs: BufferedImage[]
- rowIndex: int
- index: int
- mouseOver: boolean
- mousePressed: boolean
+ UrmButton(x: int, y: int, width: int, height: int, rowIndex: int)
- loadImgs(): void
+ update(): void
+ draw(g: Graphics): void
+ resetBools(): void
+ isMouseOver(): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(): boolean
+ setMousePressed(mousePressed: boolean): void
Extends
LevelCompletedOverlay
- playing: Playing
- menu: UrmButton
- next: UrmButton
- img: BufferedImage
- bgX: int
- bgY: int
- bgW: int
- bgH: int
+ LevelCompletedOverlay(playing: Playing)
- initButtons(): void
- initImg(): void
+ draw(g: Graphics): void
+ update(): void
- isIn(b: UrmButton, e: MouseEvent): boolean
+ mouseMoved(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
MenuButton
- xPos: int
- yPos: int
- rowIndex: int
- index: int
- xOffsetCenter: int
- state: Gamestate
- imgs: BufferedImage[]
- mouseOver: boolean
- mousePressed: boolean
- bounds: Rectangle
+ MenuButton(xPos: int, yPos: int, rowIndex: int, state: Gamestate)
- initBounds(): void
- loadImgs(): void
+ draw(g: Graphics): void
+ update(): void
+ isMouseOver(): boolean
+ setMouseOver(mouseOver: boolean): void
+ isMousePressed(): boolean
+ setMousePressed(mousePressed: boolean): void
+ getBounds(): Rectangle
+ applyGamestate(): void
+ resetBools(): void
+ getState(): Gamestate

Utilz Diagram



IV. CONCLUSION

1. Achievements

“Proficient use of object-oriented programming (OOP)” - We utilized object-oriented programming (OOP) principles effectively in our game development. We designed and implemented multiple classes that represent different purposes. This approach enabled us to encapsulate relevant data and behaviors within each object, demonstrating key OOP concepts such as encapsulation, abstraction, polymorphism, and inheritance.

"Knight in Training" - Successfully completing the tutorial level, where players learn the fundamental controls and mechanics of the game. It serves as a warm-up and introduction to the game's world and gameplay.

"Pixel Explorer" - Players must fully explore and conquer all three unique levels in the game. Each level presents its own distinct challenges, environments, and enemies, providing a sense of progression and discovery.

"Master of Combat" - Players who demonstrate exceptional combat skills. They must defeat a significant number of enemies using a variety of combat techniques and abilities at their disposal. It encourages players to experiment with different strategies and become proficient in combat.

"Creative Storyline" - The main storyline of the game revolves around protecting the fallen kingdom from horrifying creatures. This achievement is earned by successfully completing the main story, overcoming all obstacles and emerging victorious. It signifies the player's role as the savior of the kingdom.

Overall, the development team behind the game has put great effort into creating an enjoyable and immersive experience for players. By combining captivating gameplay mechanics, classic visual style, and a compelling

narrative, the game project showcases the creativity and technical skills of the developers while aiming to deliver a memorable and enjoyable gaming experience.

2. Restrictions

“Power Attack” - Despite the presence of this energy bar, it has not been put to use because the developers did not implement a feature or item, such as an energy recovery tank, that would allow the character to replenish their energy.Without an energy recovery mechanism, the energy bar remains unused, and the character's energy level cannot be replenished, potentially limiting their abilities or actions within the game or software.

“Player’s Ground surfing state” - The implementation of this ground surfing state has not been carried out yet. The reason for this is that the developers or designers are facing difficulties in designing a suitable map or environment that aligns with the requirements and mechanics of the ground surfing state. Designing a map that complements the ground surfing state likely involves creating terrain, obstacles, or pathways that allow the character to smoothly navigate and perform the intended ground surfing actions. Due to the challenges associated with this task, the implementation has been delayed or put on hold until a suitable map design can be achieved.

“Lack proper organization code” - code reuse has not been effectively employed. By not leveraging code reuse, developers are forced to repeatedly write similar code, resulting in redundancy, increased development time, and decreased sustainability of the codebase. Repeatedly writing code that could be reused not only wastes time and effort but also makes the codebase harder to maintain and update in the future.

3. Potential improvements

Code Refactoring: Invest time and effort into refactoring the existing codebase to improve its cleanliness, organization, and adherence to coding best practices. This includes proper naming conventions, modularization, and documentation. Refactoring can increase code readability, maintainability, and facilitate easier future enhancements.

Design Patterns: Explore and implement commonly used design patterns, such as Singleton, Factory, or Observer patterns, to promote code reuse and modularity. Design patterns provide proven solutions to common programming problems and can enhance code scalability and maintainability.

V. REFERENCES

1. Platformer tutorial - https://www.youtube.com/playlist?list=PL4rzdwizLaxYmltJQRjq18a9gsSyEQQ-0
2. Java Code Programming Github - https://github.com/KaarinGaming
3. Pixel SpriteSheet - https://craftpix.net/
4. Oracle Java Documentation - https://docs.oracle.com/javase/tutorial/java/IandI/index.html
5. Inheritance in Java Tutorial - https://www.tutorialspoint.com/java/java_inheritance.htm
6. Polymorphism in Java Tutorial - https://www.tutorialspoint.com/java/java_polymorphism.htm
7. Abstraction in Java Tutorial - https://www.tutorialspoint.com/java/java_abstraction.htm
8. Encapsulation in Java Tutorial - https://www.tutorialspoint.com/java/java_encapsulation.htm