VIETNAM NATIONAL UNIVERSITY – HCM

INTERNATIONAL UNIVERSITY

SEMESTER 1 (2023-2024)

OBJECT-ORIENTED PROGRAMMING

FROST KING'S TALE GAME

Members:
Hoàng Ngọc Quỳnh Anh - ITCSIU22256
Trương Hồng Quân - ITCSIU22275
Nguyễn Minh Thuận - ITCSIU22269
Trần Thanh Bình - ITCSIU22255
Lê Hoài Bảo - ITCSIU22259

TABLE OF CONTENT

CHAPTER 1 : INTRODUCTION

Our professor required us to create a game using basic object-oriented programming for our final project in Object-oriented programming course, so we chose to create a 2D platformer game called "Frost King's tale". So that we can handle and demonstrate to our professor that the code we write is based on OOP characteristics such as inheritance, abstract, encapsulation, polymorphism,..And some basic design pattern which is singleton pattern,....

The only coding language we utilized for this project was Java. Java is a based-class, object-oriented programming that helps us handle our code based on OOPs features.

In this report, we will go over this game in greater detail, including how it may be created, what type of game it is, and how players can participate. We also talk about how we coded the game utilizing OOPs.

CHAPTER 2: RULE AND GAME PLAY

The game we made is a top-down 2D- platformer game inspire by youtuber name ryoSnow where our character is moving around in different levels to find objects that need to kill some monster, to increase its blood, to light up around when night comes and also use that object to move to the next level or winning the game.

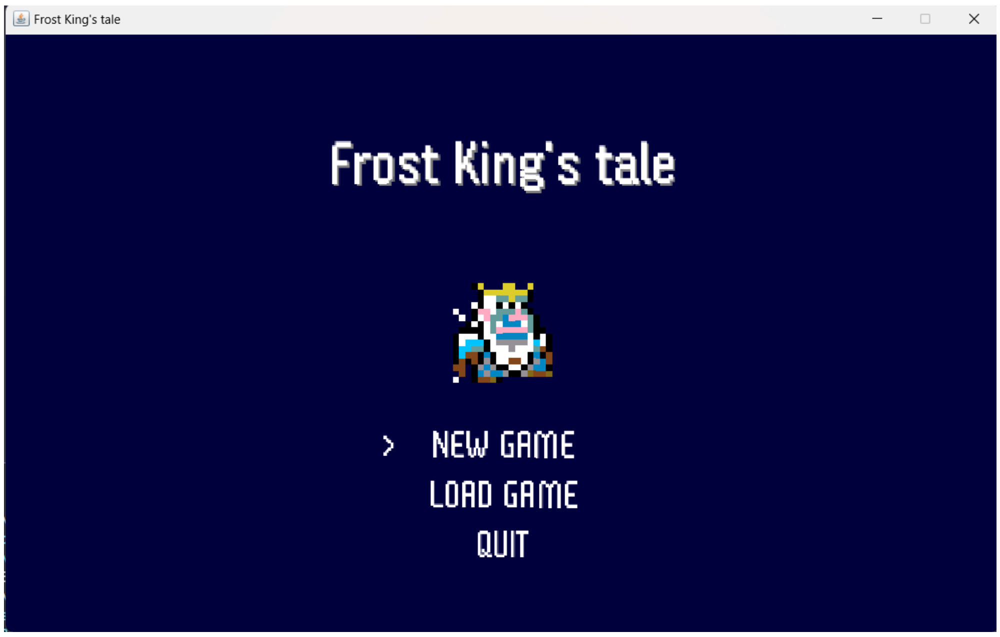
The main character in this game will be Frost king, the losing king in his adventure journey.

To play this game, we always use four main push buttons: W to go up, A to turn left, S to move down, and D to turn right and LEFT MOUSE to attack the monster. The other push button is ENTER, which the user uses when they want to talk with the npc that we created to talk with the character or select any object that the character collects inside its pocket. Another one is ESC which is often used to open the setting and turn off the menu, full screen of the game when we turn it on .

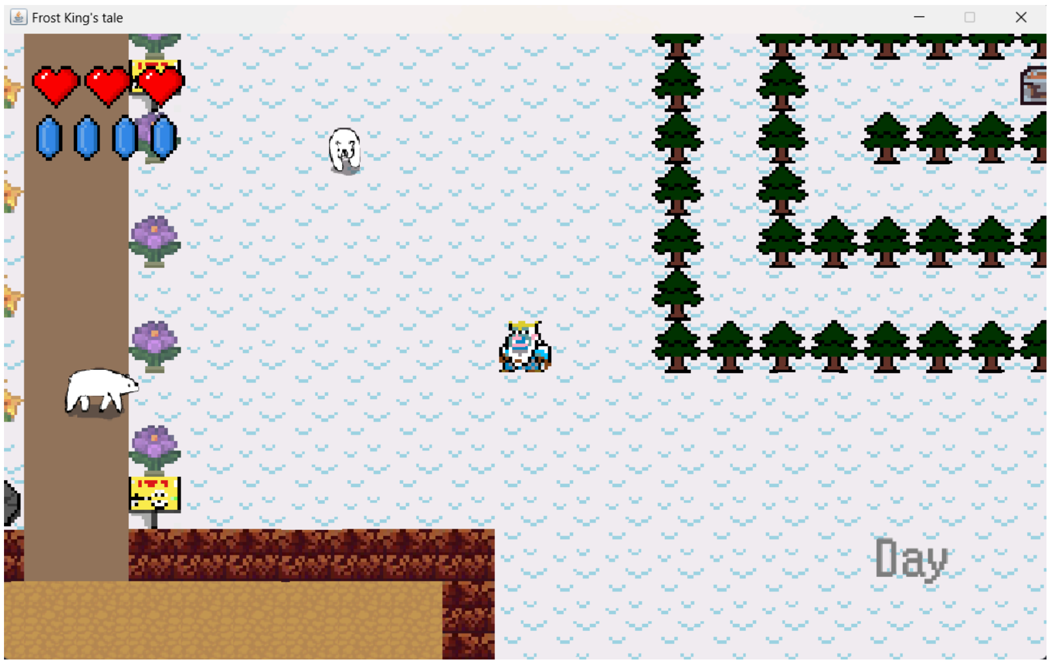
Frost King's Tale game has two levels and two different maps; this report will be divided into two parts to help you understand what's inside and how to play the game in two different environments.

1. Level one (Outside map)

When you open the game, the first appearance will be look like this

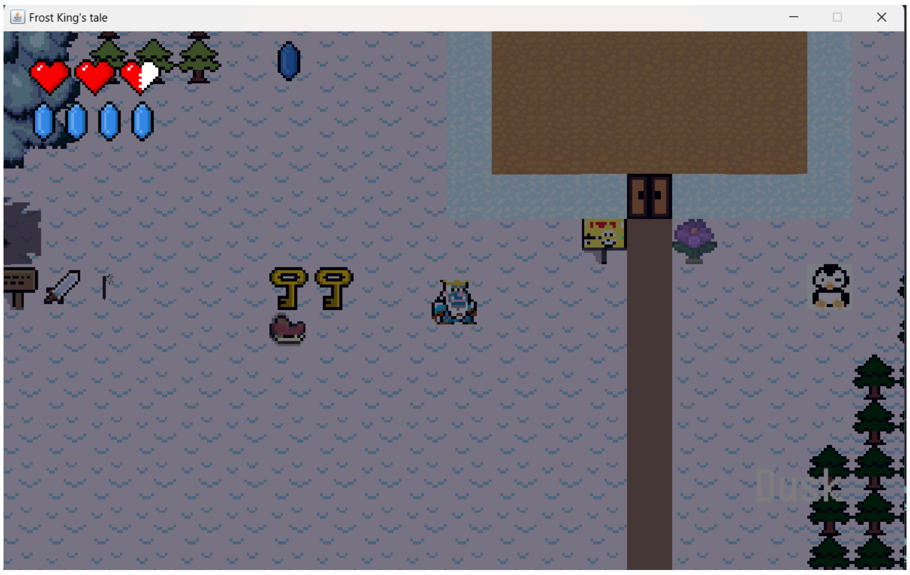


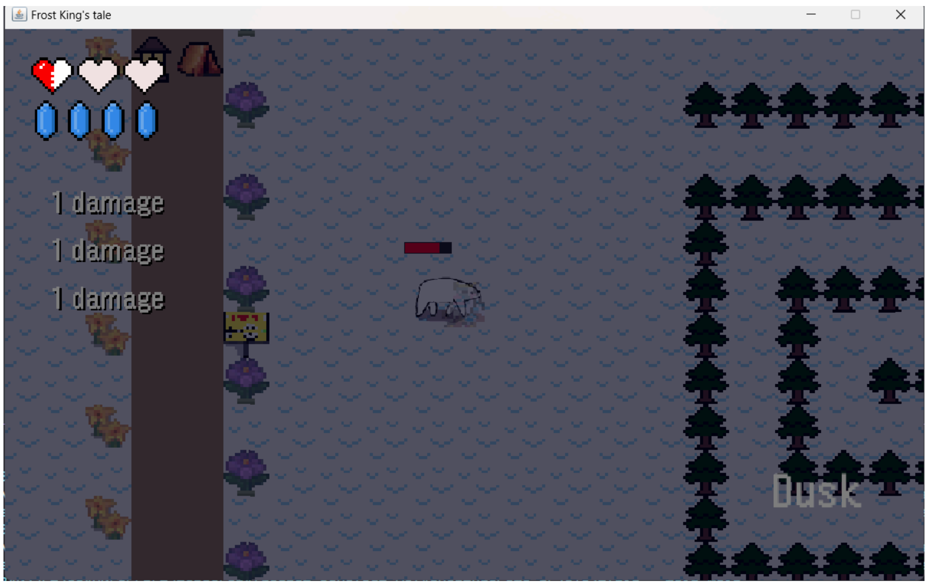
Then, click "New game" to begin the game or "quit" to end it. After clicking "New game," the application will take you to the first map, which represents the first level of the "Frost King's Tale game."

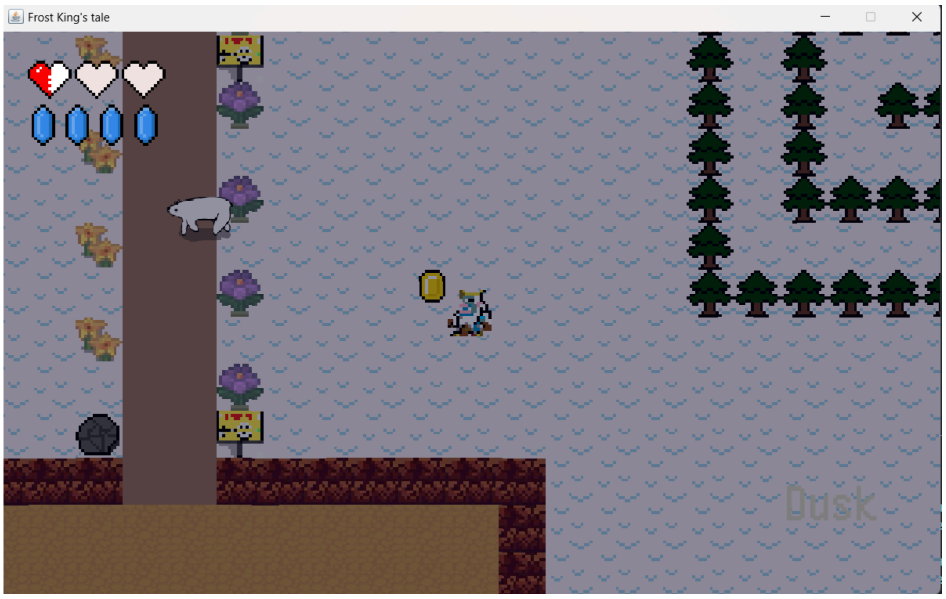


In this first level, the user should control the main character to move and look around to find things for map level two. However, in order to collect things, the character should have to run away and fight from the monster called polar bear to gather things and cash for map two as well as buy some objects to level up its skill.

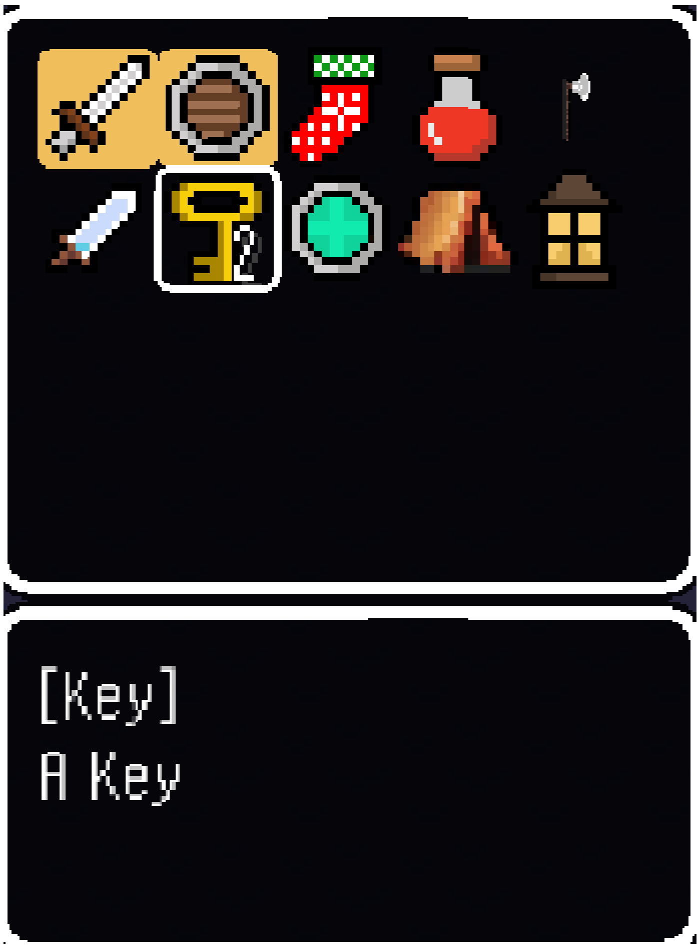
Once the character successfully kills a polar_bear or cuts any trees he will receive a coin which can be used to buy objects in trade/sale house.


Frost King's tale
Dusk

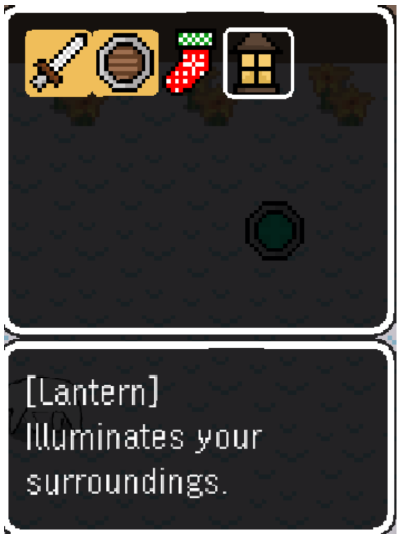

Frost King's tale
1 damage
1 damage
1 damage
Dusk



There are several items that are created for users to obtain and make it simpler to cut trees kill the bear and the monster on map two.They are Diamond_shield, red_potion, axe, diamond_sword, chirstmas_boot and lantern,....

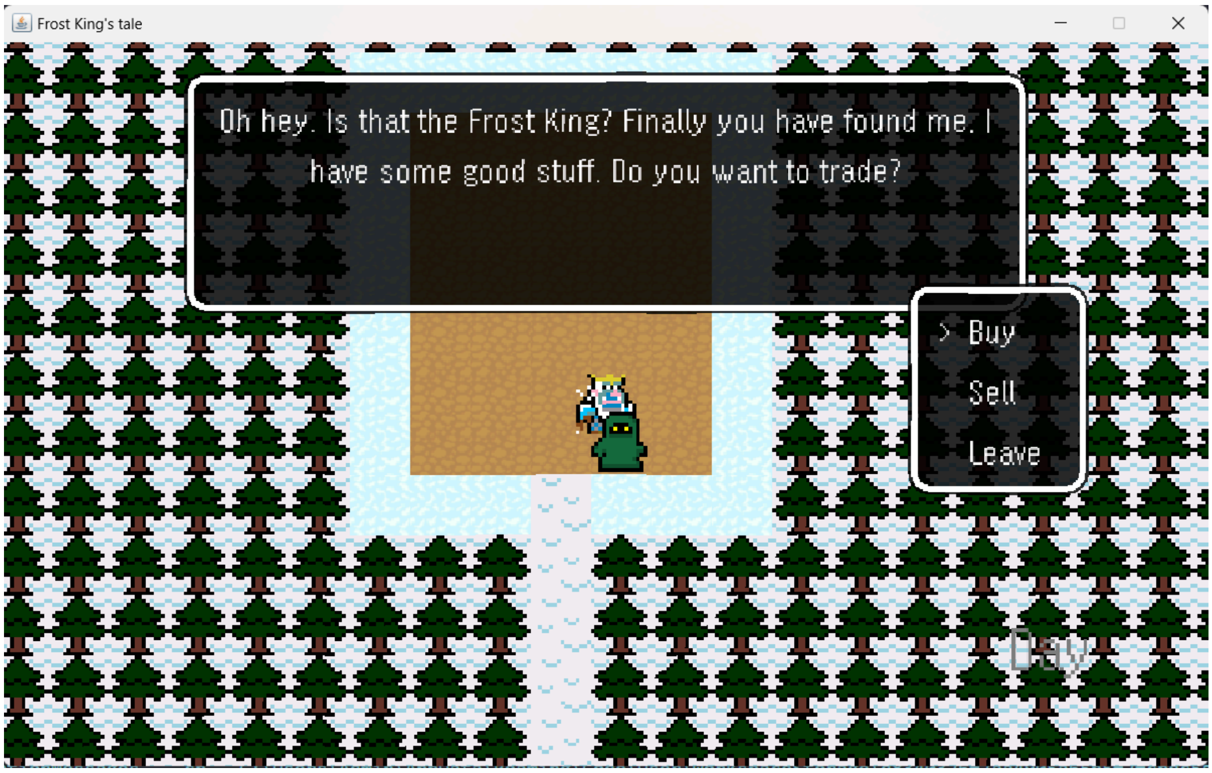


In map one, our group also creates lightning effects with day, dusk and night so if the character wants the light at night to make it easier to collect things, the user should control the character find and collect the lantern to deal with that. Also, collecting the lantern will help for map level two.

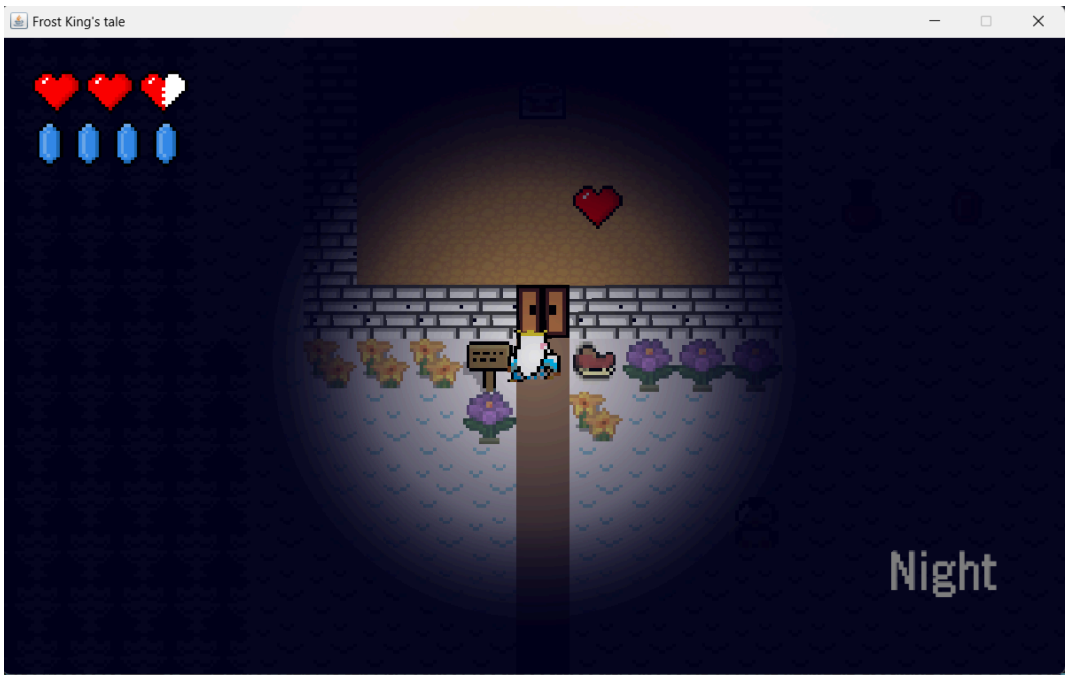

[Lantern]
Illuminates your surroundings.

So, after collecting items in map one, the player can control the player to find the way to where they can trade and sell objects to level up the character's skill to kill the boss in map 2 and then the character must discover a way to get to map two .

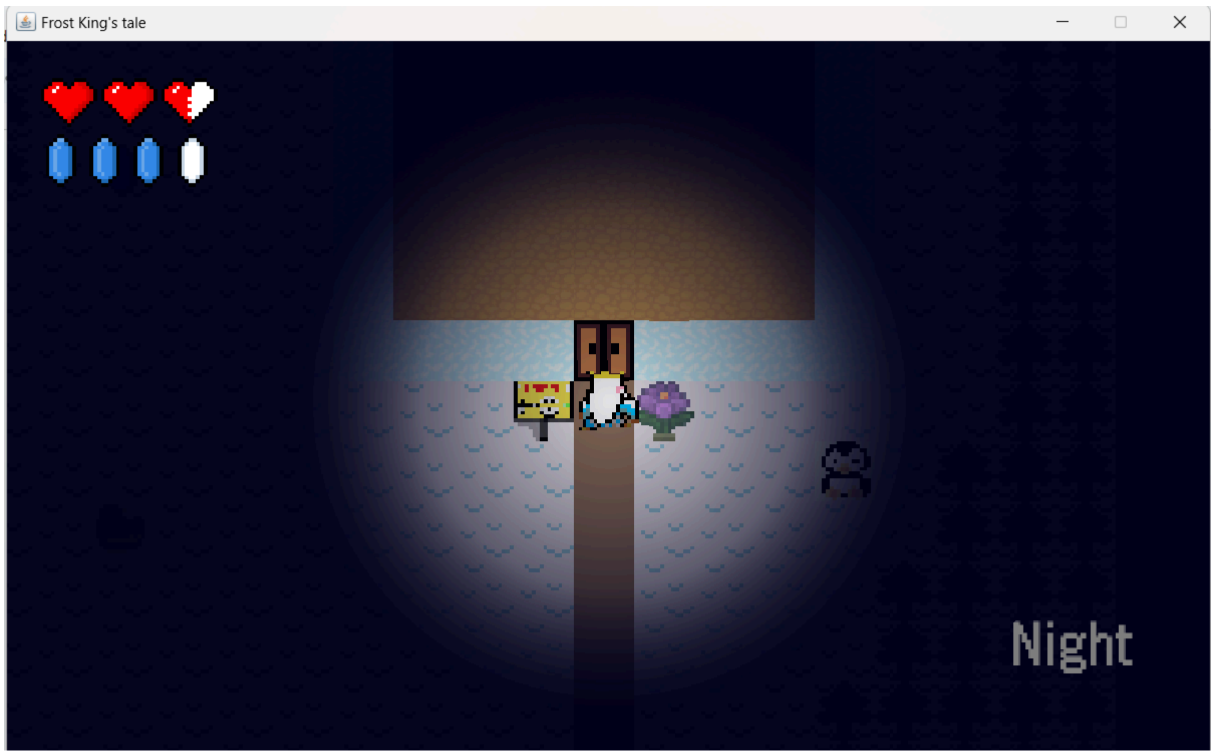
This is what trading and selling look like.


Oh hey. Is that the Frost King? Finally you have found me. I have some good stuff. Do you want to trade?

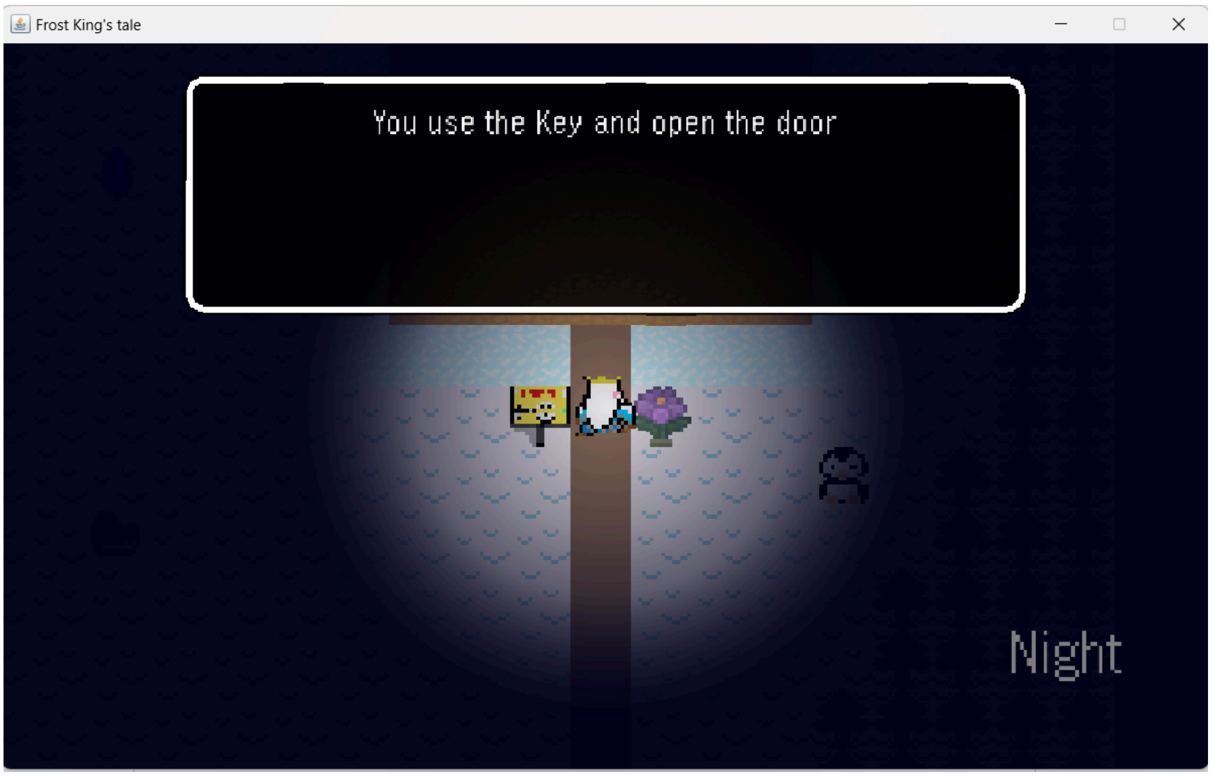
This is the first path in which the character can proceed directly to the trade/sell object house.



This is the first path in which the character can proceed directly to the map level 2.



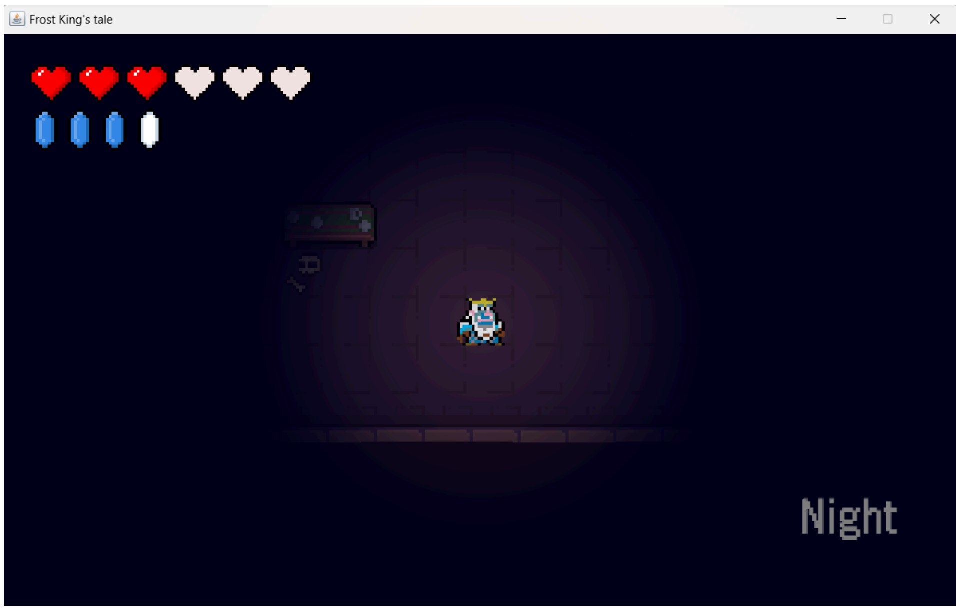
One more thing players should remark when they play the frost king's tale game, in order to unlock the door to map two and trade/sales house , the character must gather a key located throughout map one, which can be found in a chest or anyplace on the map.



You use the Key and open the door

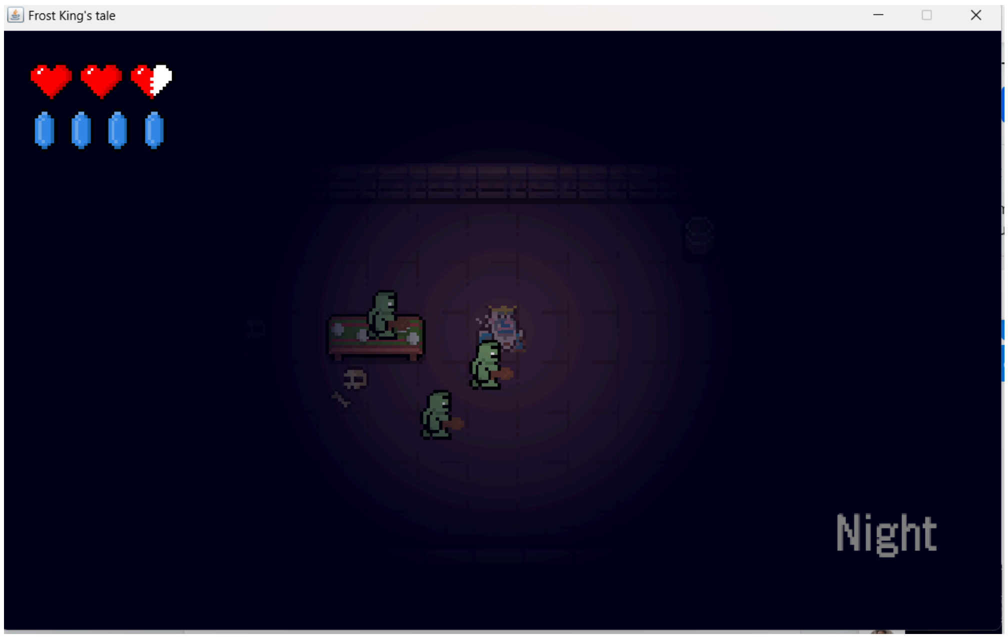
2. **Level two (Dungeon map)**

After determining the correct way to level two, the game will immediately transport the character to map two, which will look something like this. However, one thing to remember is that the dungeon map is always in the dark filter, therefore the lantern is always necessary in map two and should be collected in map one.

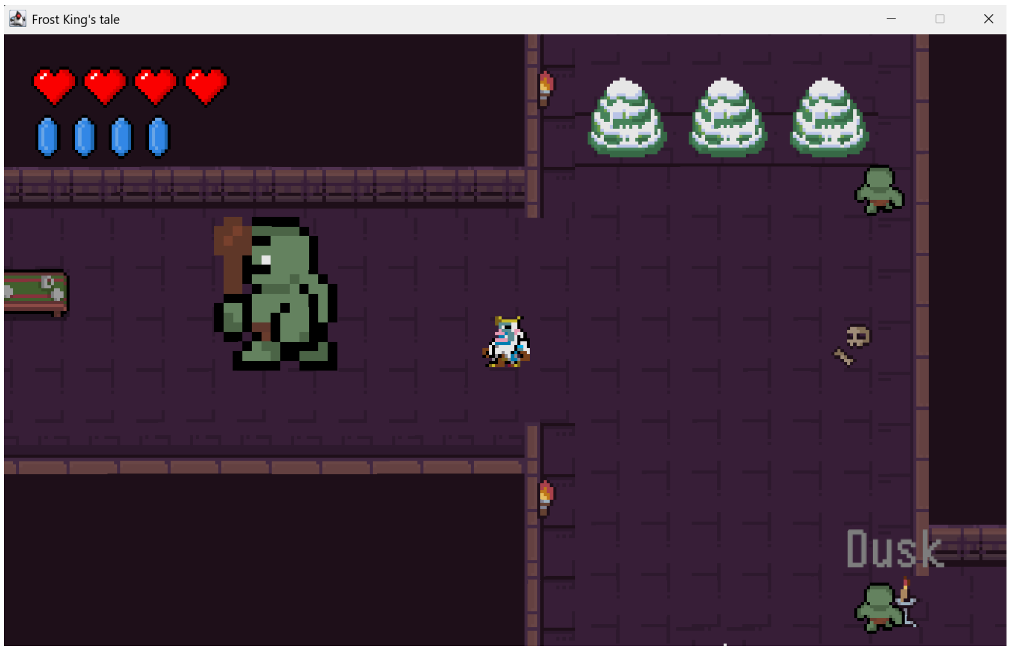


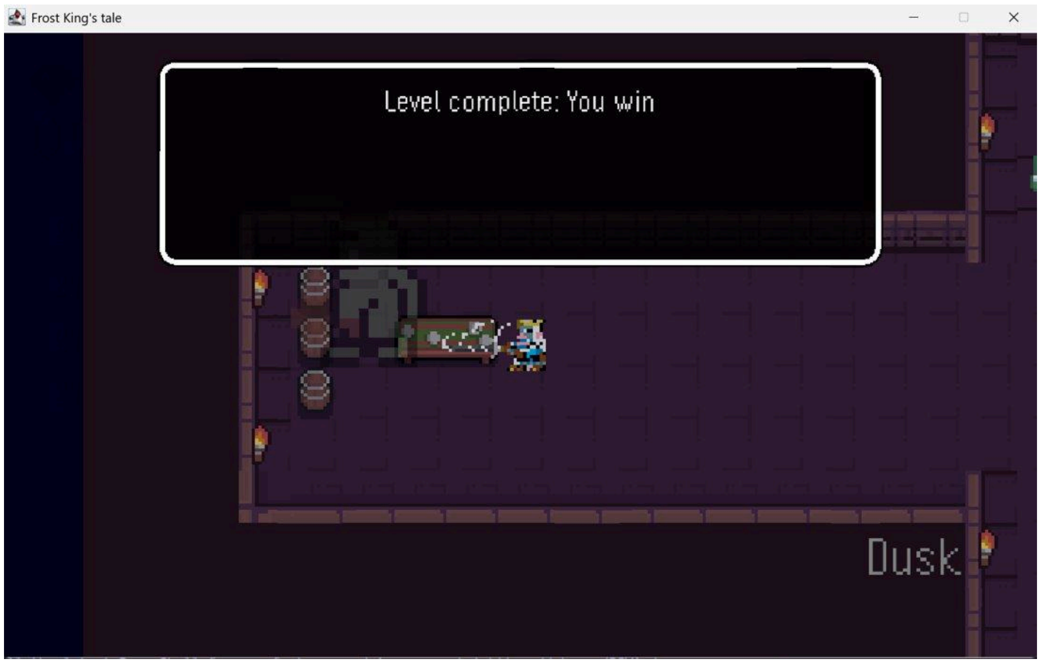
In map level two, the user should control the character who will fight these monsters, whose attacks are much stronger than the polar bear in map one. Our character cannot win without changing his weapon, which is why the user must collect a lot of items in map one.

Then, the frost king's need to kill all the monsters (especially Lord of The Orcs) which exist in the map to win.



The Orc Lord will appear at the top of the map, go find him!





Once you beat all the monsters, the game is finally done. Congratulations on your victory!

CHAPTER 3: THE LOGIC DETAILS AND GAME TECHNIQUE

1. KeyHandler.java:

Keyboard Input Handling:
Responds to key presses and releases, translating them into specific actions within the game.
Manages various game states, including title, play, pause, dialogue, character, options, game over, and trade states.

Mouse Input Handling:
Detects mouse clicks and releases, particularly for the left mouse button, contributing to player interactions within the play state.
Manages different substrates related to mouse input during the attack.
Game State Transitions:
Facilitates transitions between different game states based on user input.
Allows users to navigate through menu screens, initiate dialogue, and control character movements.

Inventory Interaction:
Enables players to interact with their inventory during the character state, navigating through slots and selecting items.
Manages both player and non-player character (NPC) inventory interactions during trade sequences.

Debug Features:
Implements debug features, such as toggling debug text visibility, aiding developers in identifying and resolving issues.

Volume Adjustment and Options Menu:
Handles user input for adjusting volume levels in the options menu.
Manages navigation within the options menu, allowing users to customize game settings.

Event Triggers:
Listens for specific key or mouse events to trigger in-game events, such as initiating dialogue, character actions, and pausing the game.

2. Entity.java:

Position and Collision Handling:
Manages entity positions (using worldX and worldY) and collision detection for interactions with other entities and obstacles.
Sprite Animation: Animates entity movements through sprite sheets, controlling animations via keyboard (WASD), and sending the signal to the sprite numbers to transit the character frames.

Attributes and Counters:
Defines attributes (speed, life, etc.) and counters for actions, invincibility, and time-related events.

Entity Types:
Categorizes entities into types (player, NPC, monster) for easy identification and classification.
Inventory and Items:
Manages entity inventory, allowing item use with attributes like value, attack, and defense.
Interactions and Dialogue: Supports entity interactions, including speaking and dropping items. Incorporates a dialogue system for NPC communication.

Pathfinding and Movement:
Implements pathfinding for entity navigation and controls movement based on calculated paths and random directions.

Combat and Damage:
Handles combat functionality, including damage calculations, invincibility periods, and projectile shooting.

Graphics and Rendering:
Manages entity rendering and visual effects, such as alpha changes for invincibility and dying animations.

3. **Player.java:**

Constructor and Initialization:

Takes input from the game panel and key handler.

Sets the initial screen position and initializes attributes like hasKey and counters.
Default Values:
Establishes default attributes such as position, speed, and player stats (life, strength, dexterity, etc.).
Initializes the player's weapons, shields, and projectiles.
Item Management:
Manages the player's inventory and item-related actions.
Handles item pickup, usage, and removal from inventory.
Enables interaction with objects based on key presses.
Movement and Collision Handling:
Manages player movement in response to keyboard input.
Checks for collisions with tiles, objects, NPCs, and monsters.
Initiates attack actions and handles attack animations.

Combat and Damage:
Calculates and applies damage to monsters, considering player attributes and weapon properties.
Implements knockback effects on monsters upon successful attacks.
Tracks the player's life and triggers game state changes if life reaches zero.

Leveling Up:
Checks and handles player-level progression based on experience points.
Increases stats and maximum life, and updates the next experience threshold.

Interaction with NPCs:
Allows the player to interact with NPCs, triggering dialogue sequences.
Transitions to the dialogue state within the game.

Graphics and Rendering:
Manages the visual representation of the player character.
Handles sprite animations for different directions and attack actions.
Implements visual effects such as invincibility.

Miscellaneous Methods:
Includes methods for handling various game mechanics like obtaining items, restoring life and mana, and selecting equipped items.

4. **GamePanel.java:**

Screen and Window Settings: Defines parameters for the game's display, including tile sizes, screen dimensions, and window modes.

World Map Configuration: Manages the game's world map, specifying dimensions, the number of maps, and the current map in use. Tiles represent specific elements or entities within the grid-based environment.

Entity and Object Management:
Efficiently handles various entities such as players, NPCs, monsters, and interactive tiles. Uses lists and arrays to organize and manipulate entities based on types and states.

Game State Handling:
Incorporates a comprehensive game state system for smooth transitions between different states, including title, play, pause, dialogue, and more.
Game Loop:

Implements a game loop within the run method, ensuring a consistent frame rate (FPS).
Graphics Rendering and Debugging:
Supports rendering of tiles, entities, interactive elements, and user interface components.

Audio Integration:

Seamlessly incorporates sound effects and background music. Provides methods for playing, stopping, and looping audio files to enhance the overall gaming atmosphere.

5. CollisionChecker.java:

Tile Collision Detection:

The class handles collisions between entities and the game tiles. It checks whether an entity's movement will collide with a tile, preventing entities from moving through walls or other obstacles.
Other detections have the same purpose as Tile Collision Detection.

6. AssetSetter.java:

This class is responsible for initializing and setting up various game assets, including objects, non-playable characters (NPCs), monsters, and interactive tiles.
Object Initialization:
Creates instances of different game objects such as doors, keys, shields, axes, potions, coins, hearts, mana crystals, chests, lanterns, tents, and upgraded swords.

Assigns specific world coordinates to each object for proper placement within the game environment.
Object Placement on the Map:
Places objects on a specific map by populating the gp.obj array with instances of game objects.
Coordinates are set based on the tile size to ensure proper alignment within the game world.

NPC and Monster Initialization:
Creates instances of NPCs, monsters and assign world coordinates to each of them for their initial positions on the respective maps.

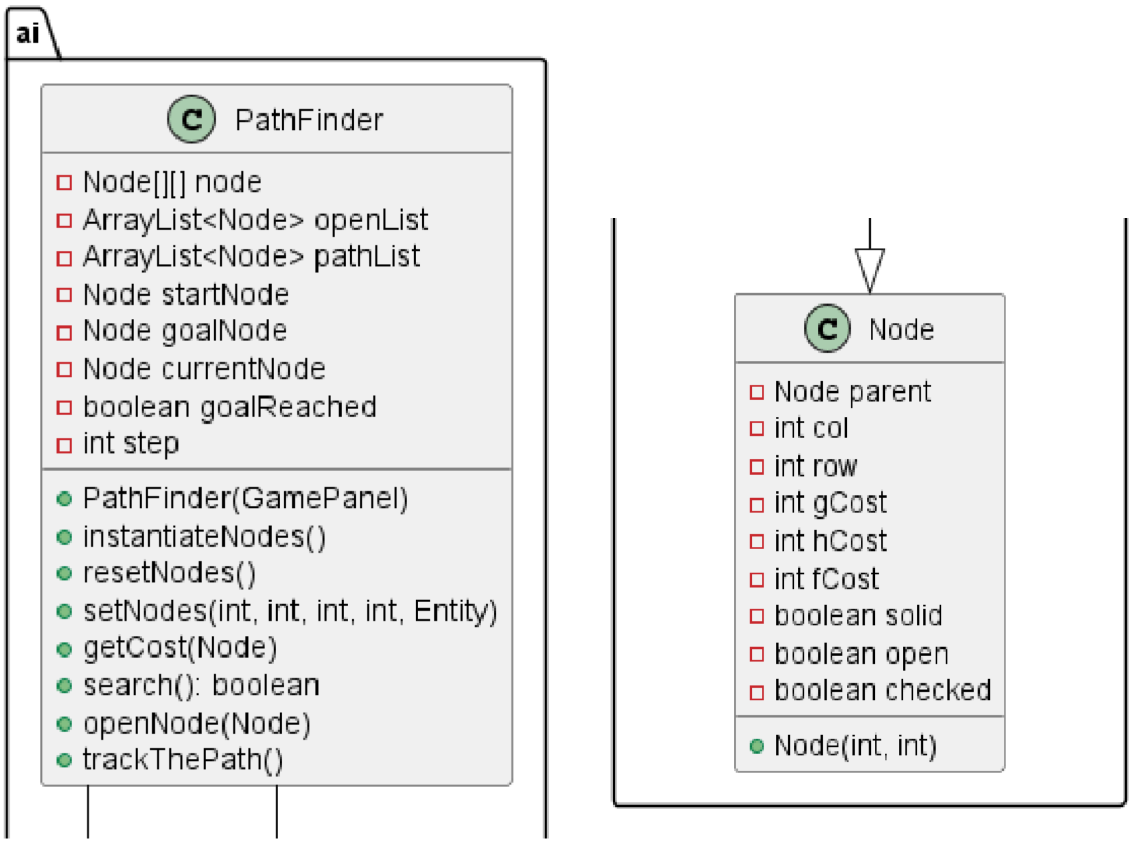
Interactive Tile Initialization:
Creates instances of interactive tiles on the map. The coordinates are provided for each tile to define their positions within the game environment.

Encapsulation of Asset Initialization Logic:
Encapsulates the logic for setting up game assets within the class, promoting code organization and maintainability.

CHAPTER 4: UML CLASS DIAGRAM

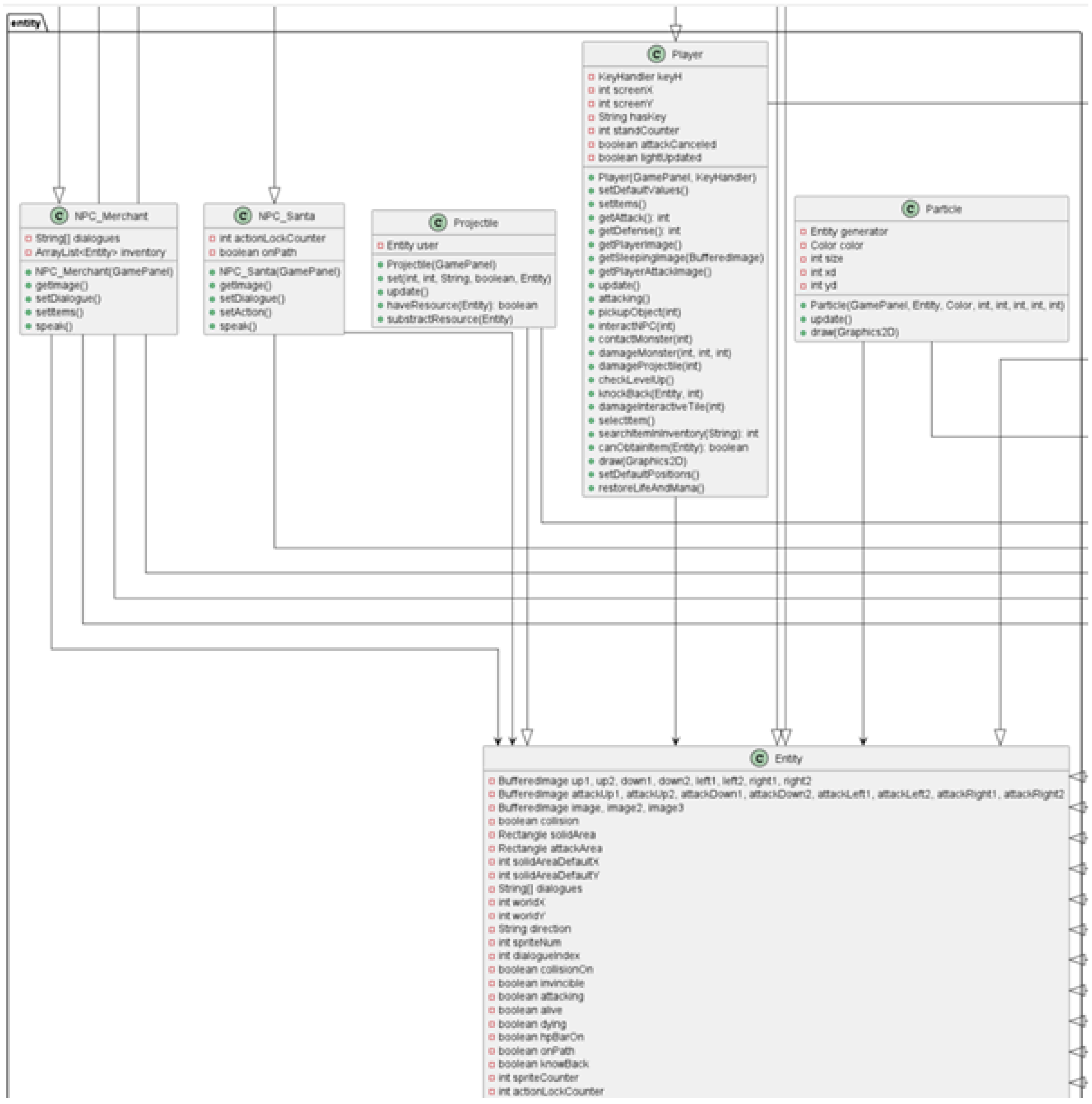
There are 8 main packages in our group's UML diagram, which are presented respectively below:

1. AI



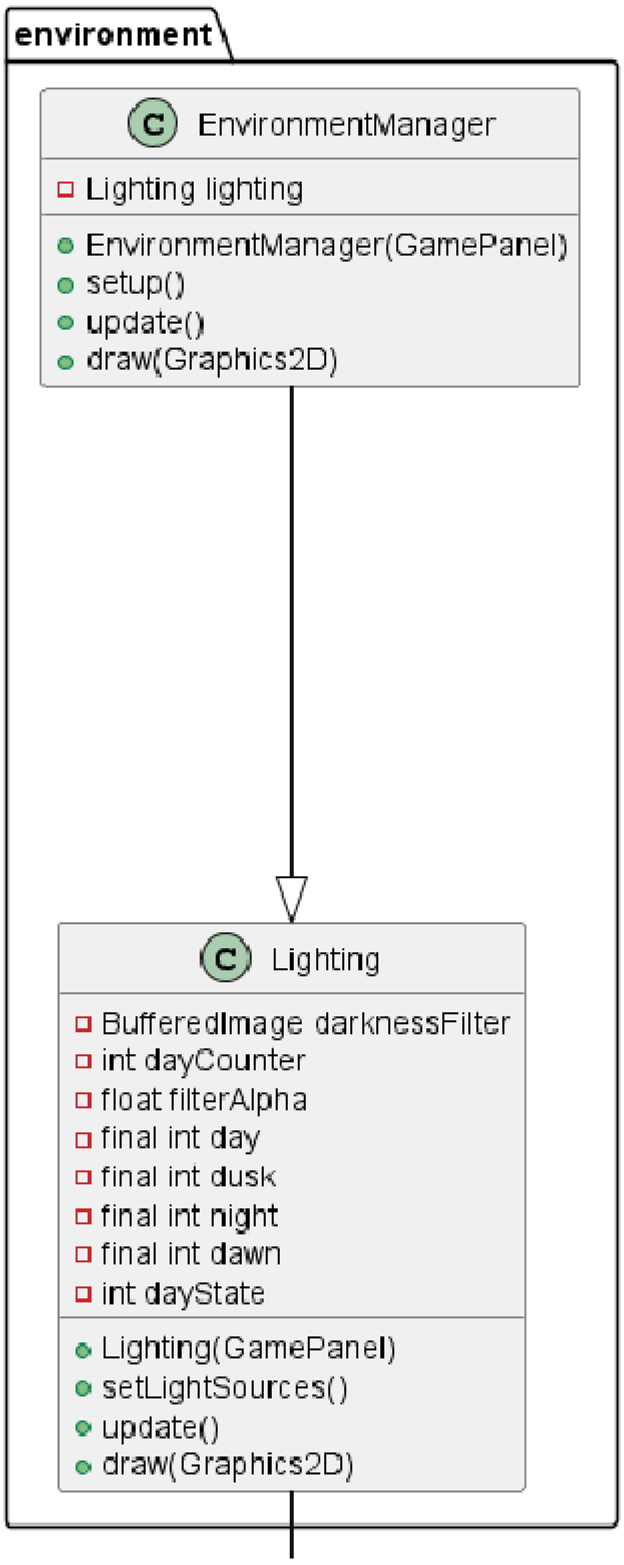
This package provides a set of classes and operations for pathfinding in a grid-based environment, with the *Node* class representing individual nodes and the *PathFinder* class orchestrating the pathfinding process.

2. Entity



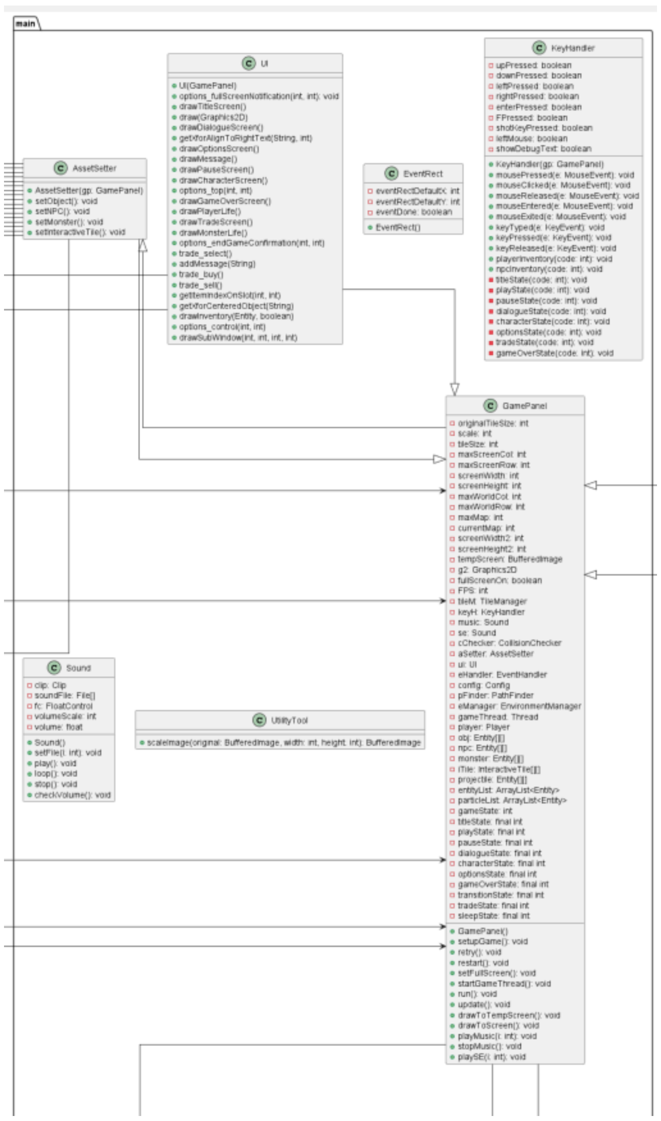
This package operates by providing a flexible and extensible framework for managing various game entities with specific behaviors, interactions, and visual effects.

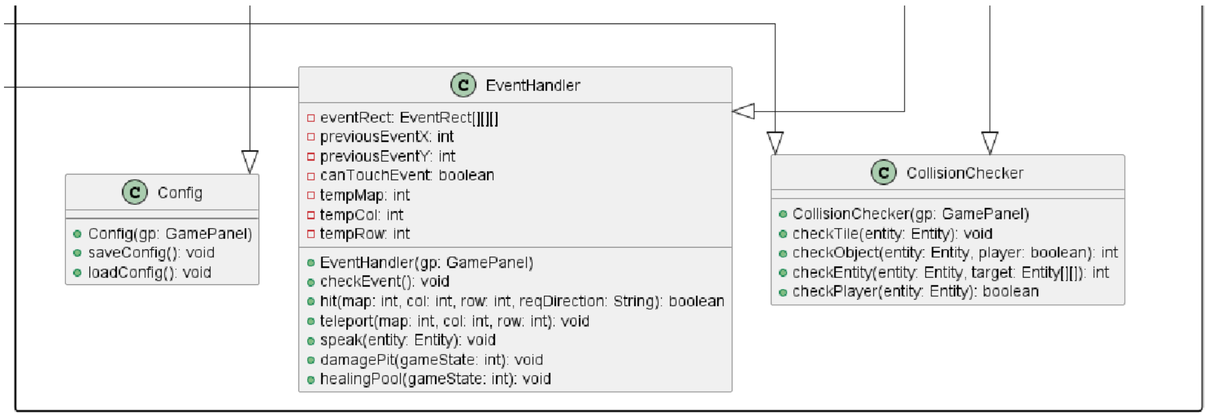
3. Environment



The *environment* package manages the game's environmental aspects, including lighting effects and the day-night cycle. It offers flexibility for dynamic changes in lighting conditions and seamlessly integrates with the game's rendering system. The package's design allows for the creation of immersive and visually appealing game environments.

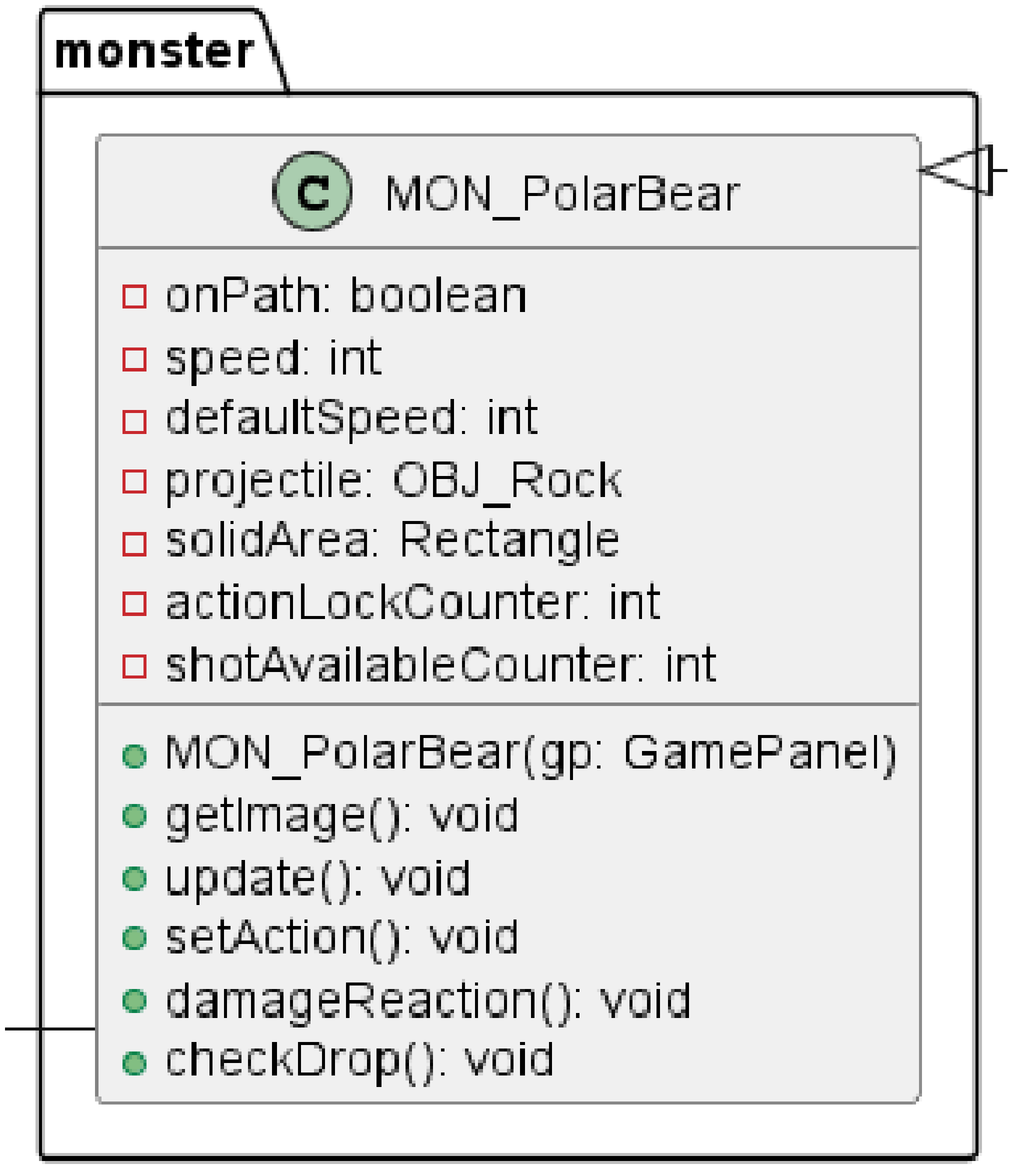
4. Main




EventHandler
eventRect: EventRect[][][]
previousEventX: int
previousEventY: int
canTouchEvent: boolean
tempMap: int
tempCol: int
tempRow: int
EventHandler(gp: GamePanel)
checkEvent(): void
hit(map: int, col: int, row: int, reqDirection: String): boolean
teleport(map: int, col: int, row: int): void
speak(entity: Entity): void
damagePit(gameState: int): void
healingPool(gameState: int): void
Config
Config(gp: GamePanel)
saveConfig(): void
loadConfig(): void
CollisionChecker
CollisionChecker(gp: GamePanel)
checkTile(entity: Entity): void
checkObject(entity: Entity, player: boolean): int
checkEntity(entity: Entity, target: Entity[][]): int
checkPlayer(entity: Entity): boolean

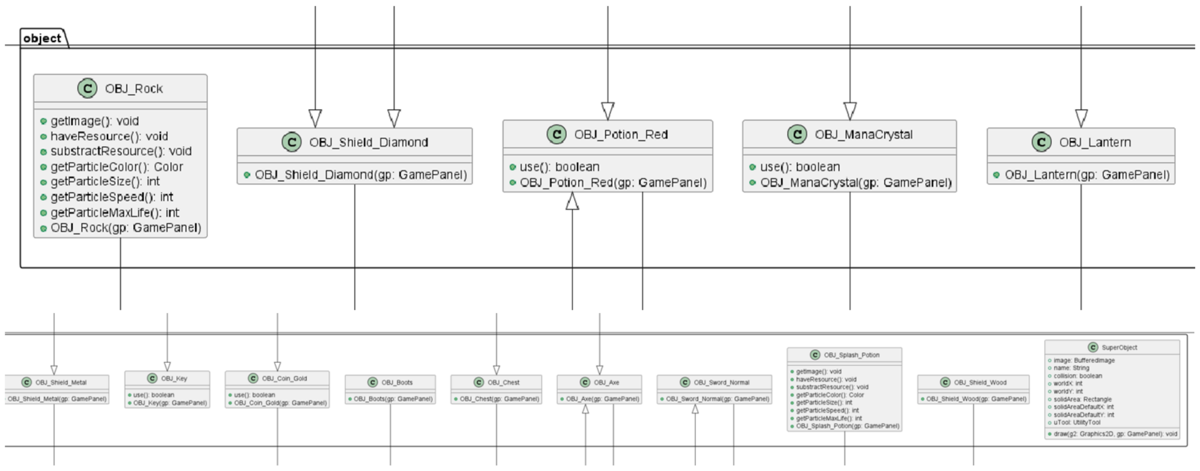
The *main* package contains classes responsible for setting up and managing various aspects of the game, including assets, collision checking, configuration, event handling, UI, sound, and key interactions. The package design allows for modular and organized management of different game functionalities.

5. Monster



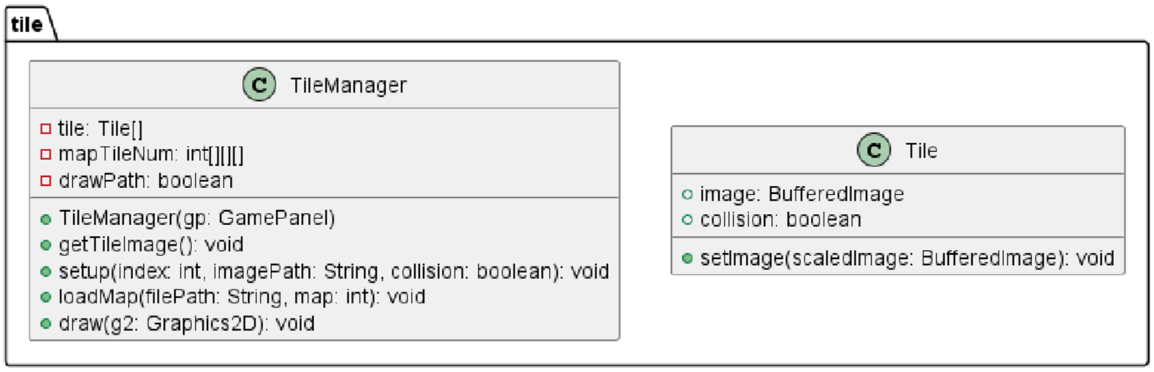
The *MON_PolarBear* class represents a polar bear entity within the monster package. It encapsulates attributes related to the polar bear's behavior, movement, and interactions. The methods are responsible for updating its state, managing actions, and handling reactions to various game events. The class operates within the broader context of the game's monster-related functionalities.

6. Object



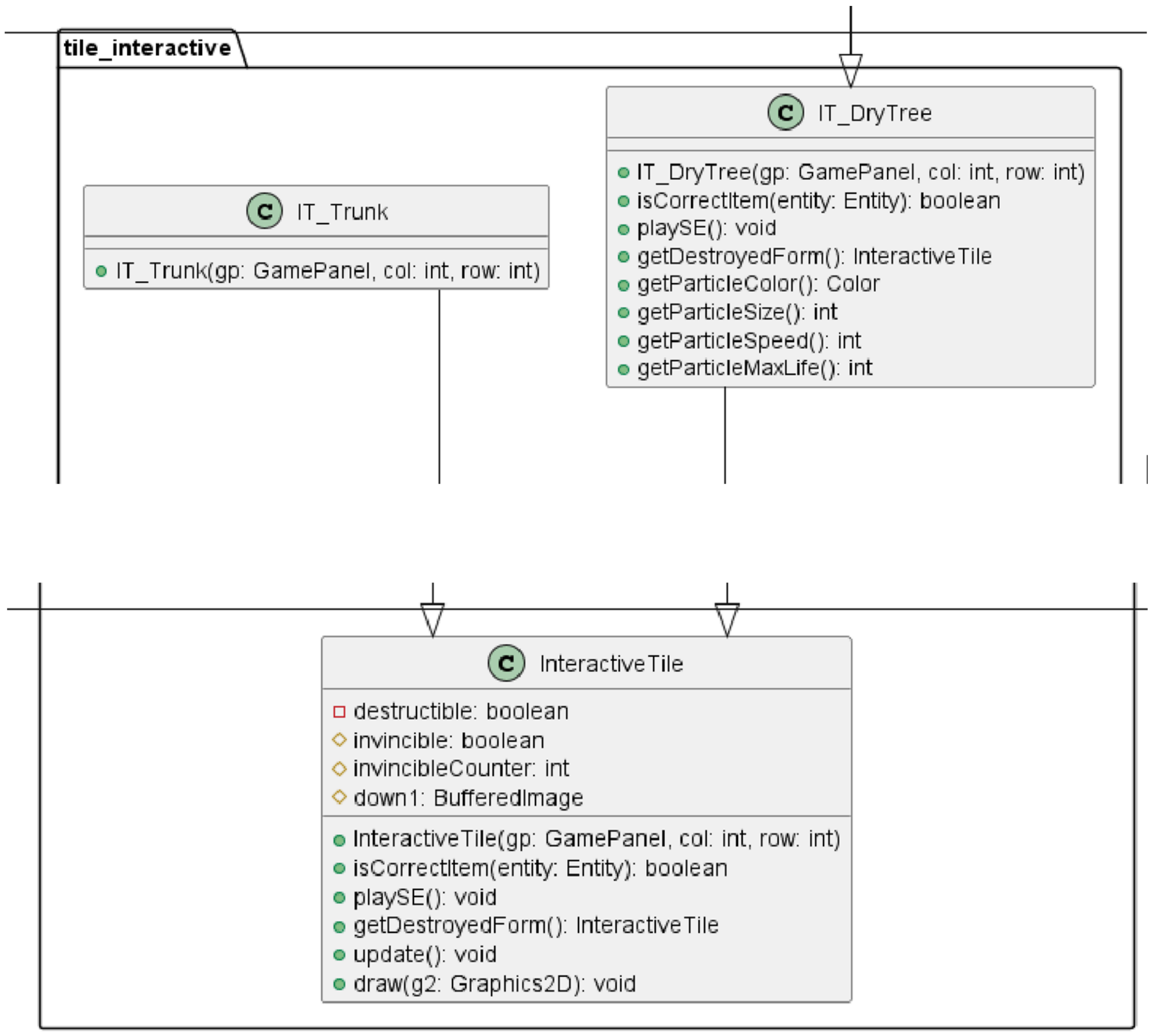
This package is an encapsulation of a variety of game objects with specific functionalities. The use of a common superclass *(SuperObject)* allows for a consistent interface and potentially shared behaviors among these game objects.

7. Tile


tile
TileManager
tile: Tile[]
mapTileNum: int[][][]
drawPath: boolean
TileManager(gp: GamePanel)
getTileImage(): void
setup(index: int, imagePath: String, collision: boolean): void
loadMap(filePath: String, map: int): void
draw(g2: Graphics2D): void
Tile
image: BufferedImage
collision: boolean
setImage(scaledImage: BufferedImage): void

This package is designed for managing and rendering tiles in a game environment. The *Tile* class encapsulates individual tile properties, while the *TileManager* class handles the overall management, setup, and drawing of tiles on the game panel.

8. Interactive Tiles (tile_interactive)


tile_interactive
IT_DryTree
IT_DryTree(gp: GamePanel, col: int, row: int)
isCorrectItem(entity: Entity): boolean
playSE(): void
getDestroyedForm(): InteractiveTile
getParticleColor(): Color
getParticleSize(): int
getParticleSpeed(): int
getParticleMaxLife(): int
IT_Trunk
IT_Trunk(gp: GamePanel, col: int, row: int)
InteractiveTile
destructible: boolean
invincible: boolean
invincibleCounter: int
down1: BufferedImage
InteractiveTile(gp: GamePanel, col: int, row: int)
isCorrectItem(entity: Entity): boolean
playSE(): void
getDestroyedForm(): InteractiveTile
update(): void
draw(g2: Graphics2D): void

This package defines a set of classes related to interactive tiles in a game. The *InteractiveTile* class serves as a base with common attributes and methods, while specialized tiles *(IT_DryTree, IT_Trunk)* inherit from it, providing specific implementations and additional methods. The use of encapsulation and inheritance promotes modularity and extensibility in handling different types of interactive tiles.

CHAPTER 5: EVALUATION

1. Performance Evaluation:

Quick response time: The game has a quick response time since we have maintained the fit connection between images during each motion of the characters, which provides the players with immediate feedback and a seamless interactive experience during a smooth gameplay.

Memory Usage: We have optimized our game to use resources efficiently. The average memory usage is **26 MB.** This low memory footprint ensures that our game runs smoothly even on devices with limited resources.

Load Time: The initial load time of our game, from launching the application to the start of gameplay, is approximately 2 seconds. This quick load time allows players to get into the action without unnecessary delays.

2. Code Quality:

Readability: The codes strictly follow clear routes organized from the beginnings so they work correctly as the initial prediction. So these codes' arrangements are clear to read.

Maintainability and code comments: We have added consistent naming systems so as to recognise objects and classes easily, we have included comments throughout our code to explain the purpose and functionality of complex code. This will help any future developers in understanding our code quickly.

3. User Experience:

Evaluate the game from a player's perspective: We test the game quite often under the player's points of view, the game is attractive and worth trying since it has a good story and game flow under a sense of discovery. Furthermore, there is eye catching activity during the gameplay, with brief instructions to guide players and maintain the experience.