Phase 2 Report

Overall Approach to Implementing RabbitRun:

1. Game Design and Planning

- **Define Game Concept**: Determine the main objective, gameplay mechanics, and user experience.
- **Theme**: Established a Jungle-themed game i.e. a rabbit navigating through a forest collecting clover and carrot as rewards.
- **Game Mechanics**: Decide on game mechanics such as character movement, point collection, enemy interactions, and winning and losing conditions.

2. Architecture and Structure

- **Game Loop**: Implement a main game loop that handles updates and rendering. The loop should control the game's state, including playing, paused, and game over states.
- **Game States**: Create enumerated states for menuState, playState, pauseState, youWonState, youLostState and guideState.
- **Classes and Responsibilities**:
    - **Main Class**: Initializes the game window, sets up the game panel, and starts the game loop.
    - **GamePanel**: Manages the main game logic, including input handling, game state updates, and rendering.
    - **UI**: Handles all graphical rendering, including HUD elements like score and time.
    - **Player Class**: Manages player attributes, movement, and interactions.
    - **Enemy Class**: Defines enemy behavior and interactions with the player.
    - **KeyHandler**: Captures keyboard input and controls player movement and game state transitions.
    - **mouseListener :** Captures mouse clicks and game state transitions.

3. Game Development Steps

- **Implement Main Class**: Create the main class that initializes the game window and adds the game panel.
- **Design the GamePanel**:
    - Implement the game loop for continuous updating and rendering.

    - Add methods for initializing the game state, starting the game thread, and handling game logic.
- **Develop UI Class**:
    - Load images for points, backgrounds, and game over screens.
    - Implement methods for rendering each game state (menu, play, you won, you lost).
    - Integrate a timer and message display for game events.
- **Create Player and Enemy Classes**:
    - Define attributes and methods for movement, collision detection, scoring, and interactions.
- **Implement Game Controls**:
    - Use the KeyHandler class to manage player input and transitions between game states.
- **Add Game Mechanics**:
    - Implement scoring logic for collecting points.
    - Design enemy behaviors and interactions
    - Create winning and losing conditions to transition to respective game states.
- **Build the Menu System**:
    - Create a simple menu for starting the game, viewing instructions, and exiting.

Modifications:

- We decided to transition from a level-based structure to a single, continuous map, enhancing the game's flow and immersion. This shift allowed players to explore the environment more freely without interruptions, fostering a more seamless experience and enabling us to focus on a cohesive world with richer interactions, bonus rewards, and more challenges throughout the map. Displaying the entire map at execution reveals all rewards, punishments and enemies.

- As an added level of difficulty we implemented a camera that is centered on the player. This allows us to hide part of the map, as well as punishments and enemies, thus forcing the player to explore the map to find all rewards.

Management Process:

- For the division of roles, each team member was assigned specific classes based on their strengths and the difficulty of each part. We roughly split the code so that one person worked on player mechanics, another on the enemy class, another on the UI and item placement, and finally, one person focused on collision detection between all game components. As we progressed, each of us had to build upon the code developed by others, which required clear communication and regular code reviews to ensure compatibility and cohesion. This approach

allowed us to integrate our individual components smoothly, resulting in a unified and functional game.

- For the management process, we scheduled weekly meetings both in person and over Discord. During the first two meetings, we decided on the Java external libraries we would use and divided the code for the classes among our group members. Each of us worked on our parts and communicated over Discord if any problems arose. In the following meetings, we provided progress updates, addressed any challenges, and determined the next steps for everyone. This collaborative approach ensured we stayed on track and quickly resolved issues as a team, allowing for steady progress throughout the project.

Milestone 1 To Do (Oct 27th):

- **Meeting 1 (Oct 18th):** Choose framework/GUI, start learning GUI basics
- **Meeting 2 (Oct 24th):** Review feedback, finalize class structure, assign tasks, begin coding

Milestone 2 To Do (Nov 4th):

- **Meeting 3/4 (Oct 29th/Nov 1st)**: Share progress, address concerns, outline remaining tasks
- **Meeting 5 (Nov 4th)**: Final updates, resolve issues, finalize code and report

External Libraries Used:

- **Java Swing** - Java Swing, a part of the Java Standard Library, is used to create and manage the main game window along with its graphical user interface (GUI) elements. Because Swing is built into Java, there's no need to add extra dependencies, which simplifies setup and reduces overhead. Swing's lightweight and flexible components make it easy to design a responsive, user-friendly interface with custom UI elements like buttons and labels that integrate seamlessly into the game.

- **Java AWT** - Java AWT (Abstract Window Toolkit) is also an inbuilt library in Java, providing core graphical capabilities without requiring additional downloads. AWT is responsible for rendering key game components, such as player characters, obstacles, rewards, and other interactive elements. Its low-level rendering functions offer direct control over graphical elements, enabling smooth animations and detailed visuals. By using AWT's foundational capabilities, we achieved a cohesive, visually engaging experience for players, all while maintaining a streamlined setup.

Enhancements :

To improve code readability and collaboration, we established a uniform naming convention across the codebase that all team members followed. This consistency made it easier to navigate, debug, and extend the code, as everyone used the same style for variables, methods, and class names. Additionally, we organized the project into descriptive packages, grouping related classes and functionality logically. This structured approach to naming and organization enhanced the maintainability of our code and made it easier to collaborate effectively, resulting in a cleaner and more organized project overall.

Challenges and Solutions:

Map Generation

- In our initial phase 1 plan we had decided that we would have multiple maps which are to be considered as levels. As the player progresses throughout the game, the levels become increasingly challenging and varied. However, creating and placing objects for each possible level is an arduous task. Therefore a more efficient map generation method had to be implemented.

Solution

- The map layout exists within a text file that is read from the load map method. The text file contains a grid of integers representing grass and tree tiles. By placing our map generation within a simple editable text file we were able to streamline our game map creation. However, due to time constraints we were unable to utilize this map generation method to create multiple maps and only a single map was created for the final version of the game.

Enemy Class

- In the enemy class, a simple algorithm to calculate the distance between the enemy and player was used. The enemy determines the horizontal and vertical distance between itself and the player, then chooses to move in the direction to minimize this distance. However, there was an issue of enemies getting stuck in one position after colliding with a barrier.

Solution

- To address this issue, methods were implemented to help the enemy choose a different direction to move in. When the enemy collides with a barrier, it chooses an alternate direction, and if it remains stuck, it will reverse direction. Additionally, if an enemy stays in one tile for too long, an offset is applied to move the enemy forward or backward by a random distance.

Collision Detection

- Creating collision detection was problematic as simply blocking the player object from entering the same tile space as a solid tile would cause a glitch as the character is snapped back into its previous position before the collision occurred. We also faced some issues while implementing object interactions which is a part of collision detection as the game would crash when trying to pick bonus rewards which appear and disappear randomly.

Solution

- Placing a symbolic rectangle around the player character allowed us to vary the size of the collision hit box, thus allowing some overlap between the player character and solid tiles. We also had to create separate cases in which the direction the player is moving is also factored into collision detection whereby the player is returned to the previous location before they collide with a solid tile.
- We fixed the bonus reward collision issue by using the game thread to track the runtime of the game by capturing the time at the start of the game and calculating the elapsed time within the game loop and using this elapsed time to display the bonus rewards randomly across the map.