Phase 2 Report

Approach

Our development took an iterative design approach. Implementation of features were done consecutively where every new feature implemented was built upon the last. The features we implemented went in the following order:

1. Implemented logic to generate a basic game window that can render graphics.
2. Wrote the playable entity and its movement controls.
3. Implemented a basic testing map and its collisions with walls.
4. Entities and their pathfinding but without any assets or interaction functionality.
5. Created the main menu screen with the option to select a player type.
6. Created in-game pause menu and level complete menu.
7. Created traps, eggs, and keys but without any interaction functionality with the player.
8. Added sprites to the traps, eggs, keys and enemies.
9. Implemented the interactions between entities.
10. Created a death screen menu for when the player fails the level.
11. Added logic to end the game when the player wins by collecting the keys or dies by losing all their health.
12. Implemented finalized versions of the level maps.
13. Added a level selection menu in the main menu.
14. Added missing animations to entities, traps and the player.

UML Updates

After beginning development, we noticed numerous design oversights that underestimated the complexity of features, misunderstood how specific parts would work, or just led our code to be messy. The changes we made to out UML are as follows:

- Removed Collectables since we realized that traps also needed logic used in the collectable class.
- Removed Spikes and Scientist class and implemented their logic in Traps and Enemy. respectively. This is because we only had one type of trap and scientist so having a super class that only was being extended by one sub class seemed unnecessary.
- Created a separate "update" method in Entity and Level classes. This allowed us to pause game logic and animations while continuing to render the objects behind the pause screen.
- Created a Pathfinding class, this class holds the data of all the valid and invalid tiles and stores the pathfinding algorithm that searches for the player.
- Created subclasses for Levels, these classes redefine the class constructor to change the layout of the map.
- Created AssetLoader class, to help load graphic resources.
- Added InLevelState, MainMenuState, and PlayerSelectionState to divide some of the code that was originally meant to go in the Game class.

- Added a LevelState enum to be used while determining if the player is still playing, has lost or won the game.
- Added classes for different UI elements to divide some of the code that was originally supposed to go in other classes.
- Created different State classes that were meant to divide some of the code that was originally meant to go into the Game class.
- Created GraphicGrid class to help set easily entities on the screen.
- As our deadline came closer, we made a tactical decision to skip the implementation of some features. One of the features that succumbed to this unfortunate fate was the ability to save the score after completing a level, as well as the ability to lock and unlock levels to play. We deeply regret having to use revolving tactics to complete our product, but it was necessary in our quest to ship something.

Management Process

We used GitLab issues to distribute and maintain a to-do list of features that needed to be implemented and bugs that needed to be fixed. Each issue was either assigned to an individual to prevent multiple people from working on the same feature or an individual could call out what issue they wanted to work on. This approach allowed us to stay organized and focused throughout the development process, ensuring that all tasks were completed efficiently and effectively. The use of GitLab issues also enabled the group to track progress and resolve issues in a timely manner, leading to a successful project outcome.

External Libraries

For building the GUI, we used JFrame as an external library. While there are better graphics libraries out there, we decided to stick with JFrame because one of our team members was already familiar with it. It also provided us with all the tools we needed to complete our project. Using a familiar library allowed us to work more efficiently and effectively, as we were able to avoid spending time on learning a new library.

Code Quality

To maintain code quality, we used GitLab merge requests for code approvals. Before merging any code, at least one group member reviewed and approved the changes. This approach helped us to ensure that our code was consistent, maintainable, and free of errors. The use of merge requests also enabled us to verify if our teammates' implementation was adequate before merging it with the master branch. This reduced the risk of introducing bugs into the codebase and allowed us to provide feedback and suggestions to improve the quality of the code.

Challenges Faced

During the development process, we encountered several challenges that required us to adapt and find new solutions. The most significant challenges we faced were caused by certain implementations turning out to be ineffective for the tasks they were required to do. For example, when we changed the render logic, we needed to rewrite the collisions code. Similarly, we had to rewrite the collisions and positioning code to fix pathfinding with enemies. This was a challenge because replacing the existing code meant that anyone who was implementing another feature that required the ineffective code would have to rewrite their implementation as well.

To prevent these challenges from becoming a roadblock, we tried implementing two strategies. Firstly, instead of rewriting existing implementations, we tried to add more functions while simultaneously leaving the older ones working but marking them as deprecated. This allowed us to implement new features without disrupting the functionality of existing code. Secondly, we had everyone implement code that did not have much association with the code other group members were working on. This allowed us to work on multiple features simultaneously, reducing the risk of disruptions caused by rewriting existing code.