THE REPORT

A Systematic and Iterative Development Process

Our game development approach began with the implementation of classes outlined in our UML diagram. This foundational step was crucial for establishing the core structure of our game. Subsequently, we focused on implementing basic gameplay mechanics to create a functional prototype. As we progressed, we adopted an iterative development methodology, continuously refining and expanding the game's features. Throughout this process, we conducted rigorous testing to ensure functionality and user experience met our standards. Additionally, we maintained comprehensive documentation to track our progress and facilitate collaboration within the team. This cyclical process ensured that our game evolved steadily, maintaining stability and functionality throughout its development.

Our approach to implementing the game first began with translating the design laid out in our UML diagram to concrete classes within our java project. This step involved analyzing the relationships, attributes, and methods outlined in the UML diagram and translating them into class definitions. By following the UML diagram as a blueprint, we ensured that our code structure aligned closely with our initial design, laying a solid foundation for the rest of the implementation process. This approach helped us maintain consistency between our design and implementation, making it easier to track our progress and make adjustments as needed.

After establishing the foundational classes based on our UML diagram, our focus shifted towards implementing fundamental gameplay mechanics. This involved creating the core functionalities that drive the game, such as player movement, collision detection on the map, and basic game states such as the game menu and the playing state. We prioritized building these features to ensure that the game was playable and functional from an early stage. By starting with basic gameplay mechanics, we were able to create a playable prototype that allowed us to test key interactions and iterate on our design. This iterative process helped us refine the gameplay experience and identify any potential issues early on in the development cycle.

Following the implementation of basic gameplay mechanics, our development process transitioned into an iterative approach. This involved continuously refining and expanding the game's features based on feedback and testing. We iterated on various aspects of the game, including animations, rewards, user interface, and game mechanics, to enhance the overall player experience. Each iteration involved implementing new features, addressing bugs and issues, and refining existing elements based on playtesting feedback. This iterative development process allowed us to gradually improve the game while ensuring that it remained stable and functional at each stage of development.

Our group emphasized the importance of maintaining thorough documentation to ensure clarity and consistency throughout the development process. We utilized both our GitHub repository and a Google document to document various aspects of the game's design, development, and implementation. This documentation included detailed descriptions of class structures, methods, and variables, as well as design decisions, game mechanics, and user interface elements. By documenting our progress and decisions in a centralized and accessible manner, we were able to effectively communicate and collaborate as a team, track changes and updates, and ensure that everyone involved in the project was on the same page.

Our game development process has been characterized by a systematic and iterative approach. We began by implementing the classes defined in our UML diagram, followed by the incorporation of basic gameplay mechanics. Our iterative development process involved continuous refinement and expansion of the game's features based on feedback and testing. Throughout this process, we maintained thorough documentation both on our GitHub repository and our Google document, ensuring clarity and consistency in our development efforts.

Resources and References

During the development process, we consulted various Java tutorials, playlists, and online resources to enhance our understanding and implementation of key concepts. Specifically, we found the following YouTube videos helpful:

1. Platformer Tutorial - Java
   Creator: Kaarin Gaming
   Link: https://www.youtube.com/playlist?list=PL4rzdwizLaxYmltJQRjq18a9gsSyEQQ-0
   These videos provided valuable insights and guidance during the development of our project. The playlist helped us implement key features in our game such as player movement, loading animations, and creating our game screen.

2. How to Make a 2D Game in Java
   Creator: RyiSnow
   Link:
   https://www.youtube.com/playlist?list=PL_QPQmz5C6WUF-pOQDsbsKbaBZqXj4qSq
   We referred to this playlist to understand and implement various features in our game such as collision within our game, drawing tiles to generate a map, and a camera centred on the player.

By leveraging these resources, we were able to improve our coding practices and enhance the overall quality of our project.

Shortly after starting to implement our game using our UML diagram as a reference, we came to realize that we had missed a lot of important classes and factors in the game.

We added a helper class called Position to keep track of the x and y values of an entity on our 2d map array. Adding this class kept things consistent across all entities and helped us determine when the Player collided with an entity and the shortest path from the Enemy to the Player.

We added a static class called Animation constants to keep track of the array sizes of each entity's 2d Sprite array. This was necessary as a Player constantly switches directions based on the KeyInputs and the Animation Constants contain switch cases to quickly pass in array values to find the corresponding movement to sprite.

We also added a bunch of different classes to help display our game. GameWindow holds the frame of our game which extends JFrame, a class in the AWT framework. Also GamePanel which paints our objects and entities to our GameWindow. For our map, we also added in a Tile class which holds the image of the tile we want displayed on the map (wall, free space or border) and a TileManager which displays the tiles to our game screen. The TileManager also contains the 2-D array for the Map, which stores our map values. These map values are specific numbers for walls, free spaces and borders. The TileManager also has a 1-D array to store the individual sprites on our map. To help with keeping track of our score and stopping entities from moving outside our map boundaries, we created a class CollisionChecker.

Another feature we added was Gamestates which is a package that contains multiple new classes/interfaces, Gamestate, State, Statemethods, Playing, Menu, and GameOver. This was important for us to add so we could have a starting menu, be able to pause our game and a restart menu when the player dies. The enum Gamestate contains multiple enums to reflect which state we are in. The interface Statemethods ensures methods that read keyboard input are implemented by each Playing, Menu, GameWin, and GameOver class. The Menu class contains all of our methods for when the user starts the game (they see our starting screen). The class Playing is the state where the user is actually playing the game. The GameWin class is the state when the user wins the game. The GameOver class is the state when the user dies and can restart their game.

For the aesthetics of our game, we added a Camera class which follows our player around and restricts the view of the user. This adds more difficulty and makes our game look nicer. We also added two classes called MusicManager and SoundManager to control the music in the background of our game and when the Player runs into traps.

We created a new AssetManager class which renders objects to the game screen/map. This class contains all the static entities in our game. We also created a Door class, which will change the Playing state to the GameWin state when the player has achieved specific conditions to win (collected all 3 grad caps).

Finally, to help with the path finding from the Enemy to the Player, we added a PathFinder class and a TileNode class. TileNode contains every tile of our map and puts attributes such as the GCost, HCost, and FCost to help with finding the shortest path. PathFinder contains all the searching logic for an A* algorithm to find the shortest path from the Enemy to the Player. It also contains the TileNode 2-D array and pathList array which is crucial for using A* algorithm.

Overall, adding these classes makes finding errors easier, our code more readable and understandable, the classes are open for extensions, and there is more separation of tasks.

We originally wanted users to adjust the difficulty of the game, as well as customizing their main character, however it was slightly ambitious for us. So we decided that we will leave this out unless we have time at the end of our project. This means users will not be able to click Customize or Difficulty on our game.

For the management process of Phase 2, we set 3 deadlines for what we wanted to be done and had lots of meetings every week. Our first deadline was on March 1 where we wanted all our classes to be worked out and to determine what external libraries we would be using. Our second deadline was March 4, which was our half-way deadline. By this point, we wanted to have a map with a moveable player. Our last deadline was March 18, where we would have our fully playable game done. We consistently met these deadlines by keeping each other accountable and splitting up tasks.

The roles and responsibilities were divided up as following:
Claire: Handled the MoveableEntities, Animations, and PathFinding
Evan: Handled the Display, GameStates, KeyInputs, and Movement
Tim: Handled the StaticEntities, Timer, Score, and some GameStates
Moussa: Handled the Collisions, Camera, and Sound/Music

Everyone helped out where it was needed nearing the end of the deadline.

The external libraries we decided to use were Swing and AWT, these two packages helped us with the GUI of our game. While both libraries are used for GUIs, Swing provides more advanced components and features compared to AWT. Using these two libraries combined helped create a user friendly game.

<u>Biggest Challenges:</u>

1. Git merge conflicts: Sometimes, multiple group members simultaneously work on and modify the same files. This often results in confusion, as we have to examine each change manually and either approve or reject it.
2. Rendering the objects: Since most of our group had limited experience with Java prior to this project, we were unfamiliar with Java's various graphics libraries. This led to the group often encountering challenges with displaying multiple objects simultaneously, as the first object may be drawn over by the second. To address this issue, we had to refer to tutorials and consult the TAs.

<u>Measures to Enhance Quality of Code:</u>

1. Created separate directories to organize different packages so finding specific files would be easier.
2. Ensured methods and variable names were descriptive and followed the Camel case standard.
3. Created JavaDocs to document what a class/method does and added comments to explain more complex algorithms so group members can more easily work with them.