CMPT 276 Group 18 Phase 2 Report

Rodrigo Arce Diaz, Caleb Hengeveld, Alicia Lam, Shiqi Zhong

Game Implementation

The overall approach we used to implement Jurassic Meteor is a modular design that separates different functionalities into specific screens and classes. For example, MazeGame manages the overall lifecycle methods- create is used to initialize game resources, render is used to draw and run the main game loop, and dispose is used for cleanup. The game is then separated into different screens that each have their own logic with their own actions.

The three main screens that we use in our game are:

- MainMenuScreen
- GameScreen
- EndScreen

MainMenuScreen serves as the initial interface that a user will encounter when starting the application. It provides players with options to start, exit, or view instructions via help. Using the LibGDX framework, which will be expanded upon below, we can create Stage and Actor elements to manage interactive UI elements. We set the stage with buttons that act when pressed. The Start button changes the screen to GameScreen to commence the game. The Help button changes that stage to display helpful instructions on how to play the game, with a button to go between the two help pages, and another exit button which exits back to the main menu. The main menu's exit button exits the application.

GameScreen is where the main gameplay, core mechanics, and interactions of the game take place. It is designed to manage all the in-game elements, including the main character, enemies, collectibles, and any other objects that are relevant to the gameplay experience. Some of the core game functions include rendering and updating the game state, handling input, detailing the game logic connecting it to the graphics, and win/lose conditions.

EndScreen is the final screen that is opened after the player wins or loses the game. It shows the player's final score and how much time the game took, an exit button that exits to the main menu, and- depending on whether or not the player won or lost- a "Play Again" or "Try Again" button which starts a new game.

Project Management

During this phase, we used a similar process to the SCRUM agile process to organize the development of the game. At the beginning, we set two milestones:

- *October 27th, 2024:* complete the classes in the UML class diagram

- *November 3rd, 2024:* use external libraries to design a user interface and connect it with the game logic

We had several meetings to discuss and identify the core features that needed to be implemented. During each meeting, we set due dates to get these features done by, adjusting these due dates if needed in the next meeting, and each group member chose which of the identified features they would like to work on. Initially, we met to complete the Board, Block, Mapsite, etc. classes as a group, and then divided the remaining classes between us, balancing equal workloads with reducing overlap in our workspaces. Once the initial classes were implemented and the external library found, we moved on to implementing the user interface and connecting it with the game logic. At this point, new tasks and problems were coming up every day, and the majority of the work was being done on the same few files, so we used fast-acting and frequent communication between the entire group through a group chat to adaptively divide work and resolve merge conflicts.

External Libraries

External Libraries used:

- LibGDX

For the GUI of our game, we decided to use LibGDX, a framework for game development with Java, which we used to render the game and switch screens by extending LibGDX's Game and ScreenAdapter classes. It also included objects and methods to implement game textures, fonts, and input listeners. It simplified a lot of the code needed to draw the game and related windows on the screen. In addition, there is an extensive amount of documentation and external resources/tutorials to help learn how to use it, which was needed as our group did not have experience in game development.

Modifications to Initial UML Class Diagram Design

- *Character*
  - Moved `private int score` to GameScreen and deleted the scorechange method.
  - Changed `private int reward_collect` to `regRewardsCollected` to better reflect its purpose, and added the `public void addRegReward()` method to increment the counter.
  - Added `private TextureRegion playerTexture` to represent the character on the board.
  - Added `private Direction facing` to show which way the character was facing on the board.
  - Changed default `Character()` constructor to include a `TextureRegion` parameter because we would never make a Character object without setting its texture.
  - Added Character constructor with parameters that set its starting position and texture.
  - Changed `public void direction(char)` to `public boolean direction(char, Board)` to access the `enter()` of the block the character was moving to and to indicate whether or not the move was successful.

- *Enemies*
    - Added `private TextureRegion enemy_texture` to represent the enemy on the board.
    - Added `private Direction facing` to show which way the enemy was facing on the board.
    - Added Enemy constructor with parameters that set starting position and texture.
    - Changed `public void direction(char)` to `public boolean direction(char, Board)` to access the `enter()` of the block the character was moving to and to indicate whether or not the move was successful.
- *Moving_Enemies*
    - Changed `public void find_player()` to `public char find_player(Character,Board)` to access the character's position, the `enter()` of the block the enemy was considering moving to, and to return the input corresponding to the direction which the enemy wanted to move in for the `direction(char)` method to use.
- *PatrollingEnemies*
    - Added `private char moveTo` to hold the input corresponding to the direction which the enemy wanted to move in for the `direction(char)` method to use.
    - Added `private int xMin`, `xMax`, `yMin`, and `yMax` to represent the boundaries of the board.
    - Added a direction method that overrides Enemy's direction, calls `find_player` to set the moveTo variable, and then calls `super.direction` with the moveTo variable as the char parameter to move the enemy along its set path.
- *Rewards*
    - Added `private TextureRegion ReardTexture` to read the image for the reward
    - Added `Reward(int, int, int)` method to allow us to set position and score.
- *Regular_Reward*
    - Added `Regular_Reward()` and `Regular_Reward(int, int, int, TextureRegion)` methods to set position, texture, and score for Regular Reward.
- *Bonus_Reward*
    - Added `Bonus_Reward()` and `Bonus_Reward(int, int, int, TextureRegion, int, int)` methods to set position, texture, score, start time and end time.
- *Punishments*
    - Added `private TextureRegion PunishmentsTexture` to represent the punishment on the board.
    - Added `Punishment(int, int , int, TextureRegion)` constructor to set position, score, and texture.
- *Normal_Punishments*

    - Added `NormalPunishments(int, int, int, TextureRegion)` to set position, score, and texture.
- *Bonus_Punishments*
    - Added: `BonusPunishment()`, `BonusPunishment(int, int, int, TextureRegion, int, int)` to set position, score, time, and texture.
- *Direction*
    - Change North, South, West, and East to Up, Right, Left, Down for readability
- *Board*
    - Added `height, width` attribute to make it easier to change dimensions.
    - Added private attributes `startRoomBlock` and `endRoomBlock` to track the start and end.
    - Added ArrayList `array` to keep track of the board layout.
    - Add helper methods `createWall, CreateLongWall, createWallChunk` for wall creation.
    - The reward and punishments were moved to the board so all none moving objects on the board could be initialized when the board was created.
        - Added private ArrayList attributes `array_regReward, array_bonReward, array_regPunishment, array_bonPunishment` to keep track of all rewards/punishments.
        - Helper methods `addRegReward, addBonReward, addRegPunishment, addBonPunishment` were created to add rewards and punishments to the board.

      - Added `totalRegRewardCnt` to keep track of how many rewards are collected or game end condition.
    - Added public methods `regRewardCollect, bonRewardCollect, regPunishmentCollect` to be able to collect rewards/punishments.
    - Added private method `isRewardHere, isPunishmentHere` to check if there is a reward/punishment at the given coordinates.
    - Added `genNewBonus` to generate the timed bonus reward.
    - Added private attributes `t_last, T_PERIOD` to time bonus reward.
- *Block*
    - Added Private Attribution: `int x, int y,` TextureRegion blockTexture use `x y` to find out the position for block and blockTexture to read image for block
    - Added Method: `Block(), Block(int, int, TextureRegion)` use to collect the information for the block
- *RoomBlock*
    - Moved private `int x` and private `int y` to Block
    - Added RoomBlock constructor with parameters that set position and texture.
- *BarrierBlock*
    - Moved Private `int x` and `int y` to Block

    - Added Method: `BarrierBlock(int, int, TextureRegion)` for collect the information for BarrierBlock
- *MazeGame*
    - All of the MazeGame methods were deleted and their functionality was moved into GameScreen.
    - MazeGame extends LibGDX's Game class.
    - Added public void `create()` to allocate memory for and open the MainMenuScreen.
    - Added public void `render()` that overrides Game's `render` to call setScreen with a new MainMenuScreen object as a parameter, opening the MainMenuScreen.
    - Added public `void dispose` that overrides Game's `dispose` to call dispose, disposing of the current screen.

The Character, Enemy, Rewards, Punishments, and Block classes all have a draw method which calls LibGDX's draw method to render them on the screen with a certain height, width, texture, etc. Character and Enemy have public void draw(Batch batch, int tileSize, float offset) for smooth movement, while the other classes have public void draw(Batch batch, int tileSize).

In addition to the modifications to the classes in our UML class diagram, we added the MainMenuScreen, GameScreen, EndScreen, and Launcher classes, which are detailed more in the Game Implementation section.

Maintaining Code Quality

When coding our game, we made sure to keep attributes private or (if necessary) protected, put as many methods as possible in the parent class if it had subclasses, and used setter methods with if conditions to make sure the attributes were not being set to invalid values. After our game had achieved functionality, we cleaned up the code by removing unnecessary imports and attributes, deleted unused or redundant classes such as MapSite and Door, and added JavaDoc comments above our classes and methods.

Biggest Challenges

One of the biggest challenges in this phase from a project management standpoint, was evenly distributing the workload without getting in one another's way. At first, when we were designing the classes in the UML class diagram, it was relatively easy to divide the work with little to no overlap. However, once we started working on the user interface, the tasks had less concrete boundaries and we often ended up working on the same classes, and needed to be very careful when we pushed to and pulled from the remote repository. From a more technical standpoint, learning how to implement more advanced UI features such as buttons on windows, the Pause and Help windows, etc. was very challenging because we were not familiar with this framework.