CMPT 276 Phase 2 Report

“ DA ONE PIECE “ by Group 2

Nathan Chan
Michael Chan
Sri Srisathanantham
Shawn Xie

Due: Sunday, November 12, 2023
Semester: FALL 2023

Table of Contents

Approach to Implementation

Design Approach

From a design perspective, we designed our game following Object Oriented Programming design principles we learned from our lectures and applied them in Phase 1 of this project. We decided to use a form of Agile to allow us to be able to stay flexible in the implementation of our program and make any changes we deemed necessary to the original plan.

Technical Approach

Initial implementation would closely follow our plans from Phase 1, and the parent classes would be the first to be implemented. In this stage, we heavily incorporated pair programming into our workflow. The idea behind this is that as classes began to branch out with their own subclasses, all our team members should be familiar with the overarching parent classes to make implementation easier. As subclasses/inheritor classes were created, we would split off and begin implementing them individually.

Milestones

Half-Way Deadline: November 1

The focus and goal leading up to the half-way deadline was to implement the structure of our project based off of our UML diagram from Phase 1. The idea was to bridge the gap between Phase 1 and Phase 2 to see if the structure of our project made sense in practice. We also began discussing implementation details, for example what external libraries we would use or how we would generate the maze. Any deviations that were made from our original plan were to be recorded and noted in our report. The following goals were put in place:

1. Create the classes/interfaces reflecting our UML diagram
    a. Implement their attributes and functions
    b. Restructure the classes as needed
        i. Note down any changes for the report
2. Investigate map generation techniques (i.e. hardcoded vs generated)
3. Investigate GUI/drawing external libraries (i.e. swing vs Javafx)
    a. Find corresponding resources such as sprites

Completion: November 12

As this was the final deadline of this phase of the project, the core functionality of the game was to be completed. The goal by this milestone was to be able to start the game and play it to completion. The goals we created were as follows:

1. The game must be able to compile without error
2. The game must be in a playable state, conforming to all the requirements outlined in Phase 1
3. The report should be written, edited, and proofread
    a. All notes regarding changes made between Phase 1 and Phase 2 should be compiled and placed into the report

Adjustments and Modifications

Most of the changes we made were to the overall structure of our program, although many would be related to specific implementations of ideas (i.e. map generation) or thematic changes. As we began implementing the game, we noted down areas of improvement to be made or redundancy and discussed different ways to modify our project. Below is a list of our changes:

Structural Changes

1. The Punishment class was made to extend Actor rather than Rewards.

The punishment class had previously extended Rewards. However, we found that Rewards had too many attributes that were redundant for Punishments. In addition, Punishment extending Rewards did not conform to the “Is a” principle since a Punishment is NOT a Reward.

2. The Rewards class was removed, and BaseRewards, formerly RegularRewards, would take its place. BonusRewards was also made to extend BaseRewards.

Now that Punishment was no longer reliant on Rewards, it did not make much sense to have BaseRewards extend Rewards as there wasn’t anything really worth inheriting. We removed Rewards as BaseRewards adopted all its functions. BonusRewards began extending BaseRewards too as they had much shared functionality with small differences.

3. Character became a parent class; Actor was changed into StaticEntity.

While we were implementing movement into our game, it became apparent that having both moving/non-moving entities inheriting the same class didn’t make much sense. We separated these concerns and Character became the parent class for entities that could move, and StaticEntity was the parent class for all non-moving entities.

4. The Movement interface and Direction class were removed

While implementing movement, we used KeyListner/KeyHandler. Using these directly was found to be easier than creating a whole interface for movement, so they were removed.

5. Removed subclasses for Board

Initially there were BorderWall and BoardWall subclasses that extended from a parent class Board. We went with that method because we considered the possibility of having multiple 2D arrays each holding the indices of different entities. To simplify the implementation of collision and map generating we decided to use a single 2D array that holds the indices for every entity on the board which will only require a single class.

Thematic Changes

Thematically, we deviated from our original “One Piece” pirate themed game. The main reason we chose to do this is finding assets that we could freely use was difficult. As such, we chose to use assets that were easiest to find (sources to our sprites linked in their associated classes).

Management and Division of Duties

We used a form of Agile and Scrum to manage our project. We would try to have different iterations where we implemented, tested, and modified new features. We used Sprint-like short meetings to discuss goals and progress alongside longer programming sessions where we would all meet together to code together. Duties were divided up based on personal strengths as well as progress in individual assignments, as team members were frequently reassigned to assist in each other’s work.

List of Duties

Nathan:

- Sprite creation and animations
- Player movement
- Enemy movement
- Report

Michael:

- Collision
- Maze Implementation
- Keyhandler

Sri:

- Collision
- Maze implementation
- Player movement
- Enemy movement

- Score handling

Shawn:
- Maze generation algorithm
- Maze implementation
- Collision
- Report

External Libraries

We did not use any external libraries in the creation of our project and exclusively used libraries that could be found in the Java Development Kit. This was an intentional choice, with the idea being to eliminate the possibility of “dependency hell” and prevent any other issues that could arise from using external libraries. The libraries from the JDK we used are as follows:

1. Swing

We mainly used Swing to draw the game board that everything would be drawn on.

2. AWT

Used largely for drawing elements onto the screen, animating sprites, and moving the characters via KeyHandler.

3. IO and ImageIO

These were used in conjunction in order to successfully read in files (our sprites) so that they could be displayed.

4. Lang and Util

Used for generating random numbers that would be used for dictating enemy movement and sprite selection.

Improving Code Quality

Coding Standards

The first thing we did prior to implementing our game was to decide on a standardized coding standard that we would follow. This would increase the readability of our code, allowing team members to jump between different classes written by different people. Some examples of

coding standards would be using camel case for variable names (ex. anExample), all capitalized final variable names (ex. FINAL_INT), and capitalized function names (ex. Function).

JavaDocs and Comments

In an effort to create further readability, we added JavaDocs and comments throughout the code. Due to our collaborative team dynamic, we were all frequently looking at different classes and adding these in allowed whomever was taking a look to grasp the concept of the class without much trouble.

Challenges

The biggest challenges we faced was deciding on how to create and implement the maze. This challenge could be divided into two stages; algorithm and implementation.

Maze Generation Algorithm

Algorithmically, we began with a recursive division algorithm that would recursively divide the maze into subdivisions while creating random openings. This algorithm was scrapped because it was unreliable, often creating rooms with no openings. The second algorithm recursively created nodes that would branch into numerous paths. This also was later removed because the mazes were too inconsistently sized and some unsolvable. After much trialing, recursive backtracking was settled upon because it could reliably create solvable mazes at a consistent size.

Maze Implementation and Collision

Implementation-wise, each team member had their own ideas about how each maze would be generated and how collision would be implemented. Initially, everyone assumed that everyone was on the same page. This led to classes being written with the expectation that the maze would function according to their own idea (i.e. one 2D array, two 2D arrays, no arrays etc.) and it was a while before this issue was found. An entire session was spent discussing the details of this and we ironed out the bugs, preventing it from being a further issue. We learned the importance of communicating our ideas to one another, as assuming that everyone else thought the same way could have been potentially disastrous had we let it go on for too long.