CMPT276 Phase 2 Report

Sean Chan - smc26
Kaj Grant-Mathiasen - kgrantma
Ashhal Vellani - ava47
Bavneet Hothi - bkh6

Table of contents

1. Overall approach to game implementation

Our overall approach for implementing this phase of the game was to take an iterative approach. We started with an initial class for the player and board to implement basic input of movement commands and the respective response of the main character. That was followed by an implementation of monsters, with basic movement, and objects, with basic interactions with the player. An implementation of collisions between the two entities and an initial implementation of tiles followed. Subsequently we added collisions to the tiles. That was followed by an initial implementation of our UI design. At that point, we further improved our player-object interactions, as well as player-monster interactions. We followed with further improvements and additions to the UI and tiles. Finally, we added sound and cleaned up our code.

2. Adjustments and modifications to the initial design of project

We significantly diverged from our Phase 1 UML diagram.

Our Exit class no longer inherits from Barrier, and instead inherits from a more general Object class. Barrier itself instead inherits from an abstract Tile class. Tile itself was chosen as an easy way to extend the functionality of a given sprite while allowing for a centralized reference to a class. This is demonstrated in CollisionChecker's detectCollision method, which matches to the tile's type field.

Map was renamed to GameConsole and instead holds references to every core game data type. GameConsole now specifically holds a reference to a 2D array of Tiles, of which we can exploit for polymorphism as a centralized way to handle tiles. Map lost most of its displayScore/updateScore/win functionality, of which this behaviour was relegated to the UI class. Various fields were added to GameConsole. Characters become a single reference to a Player class.

Rewards, bonusRewards, were pushed to a general abstract object.Object array. Exit now instead subclasses this object.Object type, and was split into DoorOpen and DoorClosed for convenience.

Character, Enemy, Moving enemy now subclass an abstract Entity class. Since an enemy's moving state is boolean, we decided to merge Enemy and Moving enemy into one class, and handle creation of either type based on a boolean flag in the constructor.

The handling of all of the creation/population of these entities, instead of being pushed to a Board game factory class, was instead pushed to its own individual utility class which populated the arrays present in GameConsole. Tiles are populated by TileManager, Entities/Objects are populated by the AssetSetter class. More specifically, we chose to have TileManager read in maps from CSV files in order to streamline map creation, and to allow for easy expansion supposing we want to add levels or multipart stages.

We also slightly moved away from our use case diagram. Consider the use case where we describe starting and configuring the game. We did not implement step 3, 4. Instead we chose to let there be only one difficulty. The only other difference is in the use case where we describe moving around the map. Instead of moving around with the arrow keys as noted in step 3, we chose to map movement to the WASD keys.

3. Management process and division of roles & responsibilities

For this phase, we divided the workload into several areas. We started with an initial implementation that involved core gameplay mechanics, so that more complex functions could be done in parallel. Once we had this in place, we divided the workload into several categories, namely:

1. Core game mechanics (sound, key handler, game console) - Everyone
2. Collision detection, map design, and tile interaction - Sean
3. Objects and their interactions - Kaj and Bavneet
4. Static and moving monsters, and their effects on the game - Kaj
5. Player and camera control - Bavneet
6. UI - Ashhal

4. Justification and Explanation of External Libraries

For UI, we used the javax swing library and AWT. This was necessary for the graphics and for the window functionality.
The AWT library, specifically, provided tons of simple and easy to understand methods that helped us implement all the UI features we planned to integrate within the game. This was particularly helpful to those of us that didn't have much experience programming in Java. The Graphics2D class within the AWT library, helped us use the drawString and drawImage methods to design all the different screens for each of the game states.

We also used the Math library as well as the Random library. This was necessary for the randomization of monster, object, and diamond spawns as well as timing of the diamond objects.

5. Code quality enhancements and methodology

To prevent repeated code sections, we re-used functions and methods wherever possible. One example of this would be in the CollisionChecker class. We had initially separate collision methods for both player and monster, but through successive iterations were able to combine the two separate methods into one. We also made use of inheritance and polymorphism when possible to decrease repetition as well.

An example of this is in the Tile class, where extending subclasses took advantage of the singleton pattern in order to minimize the loading of graphics, an expensive operation.

Keeping similar code consistent was another method we used to improve code quality and the readability of our code. When we created the objects within the game, we had several initial iterations which had the asset setter methods return an integer, an object itself, or nothing at all. To ensure similar behavior, we modified how our game console and the asset setter object interacted, so that all objects could be initialized within the game console, and a method for each object type could be called that was of void return type. By doing this, we kept initialization localized to the game console, and the instantiation and initial value setting in the asset setter object. We specifically kept code encapsulated within specific packages such that only similar code was kept together.

One major measure we took to improve the quality of our code was making sure we weren't instantiating any objects within the game loop. Before implementing this change, we encountered a lot of performance issues and the smoothness of the game was heavily impacted. This was because the game loop executed continuously to update the game state and render the graphics. This created a lot of unnecessary overhead as new objects were created and destroyed repeatedly, causing the garbage collector to work harder. This moderation, therefore, ensured all the objects were instantiated outside the game loop, i.e. during initialization, and then reused in the loop. This was especially imperative in the UI class and we made sure the images, ArrayLists, Fonts and new Colors were instantiated in the UI constructor.

6. Challenges encountered

In implementing the monster class, we encountered several challenges related to the monster AI. Based on the given instructions, the monsters should follow the player and attempt to chase them. We had to make several decisions to balance the AI feeling realistic, difficulty, and ease of design. Initially, we made the monsters simply follow the same collision rules as the player. Unfortunately, when this is implemented, the monster can get easily stuck in a position behind a wall or in a corner. To solve this, we added an extra step when the monster calculates the best direction to move in. It calculates the moves currently available to it and the position it would be in if it selected that move. Based on this potential move, the monster will then calculate which move best reduces the distance between itself and the player. By approaching the movement of the AI in this manner, the moving enemy will never stay static in a single position, but rather jiggle back and forth. This gave the impression of the AI feeling more realistic, while still keeping the difficulty reasonable.