CMPT 276 Project - Phase 2
Group 13

Implementation Approach

Our game starts by running the main class in the **Launcher** class which calls the Game class initialize method.
The **Game** class then initiates the window of our game using **Window** class, determines which "state" the game currently is in, and calls one of the appropriate State classes:

The **MenuState** class:

-Is the default state when player first starts the game
-Displays the game title
-Transitions to Level1State when the player clicks on the screen

The **Level1State** class:

-Populate the map with tile graphics using the World class and Tile classes.
-Initializes the GameObject objects in level (Player, enemies, rewards, etc.) and their locations to be displayed on top of the background tiles.
-Initiates threads of MainCharacter and Enemy threads they run concurrently throughout the gameplay.
-Runs on a loop (managed by Game class) to:
-Repeatedly updates and renders the location and status of GameObject objects.
-Checks if the player has won the game to transition to WinState class.
-Checks if the player has lost to transition to GameOverState class.

The **GameOverState** class:

-Ran when the player has 0 points or runs into the enemy
-Displays "Game Over"

The **WinState** class:

-Ran when the player exits with 5 rewards
-Displays "You win!"

In the gameplay, our environment is composed of World class, Tile classes and GameObject classes.

The **World** class and **Tile** classes:

-Our more static, non-changing tiles are categorized into one of our Tile classes:
-FloorTile with ID 0
-BarrierTile with ID 1
-ExitTile with ID 2
-SpawnTile with ID 3
-When creating a World in Level1State, we pass in a Level1.txt file with a matrix where the locations of each tile is represented by a number, AKA its ID. The World class reads this file and populates the map with the appropriate tile.

The **GameObject** classes:

-For our more dynamic classes that change more often, we use our abstract GameObject classes. They contain:

-MainCharacter (Runs on its own thread)
-Enemy (Runs on its own thread)
-Trap
-Bonus and Regular rewards

-These objects differ from our static tiles as their location/appearance change, so their status is checked and updated in the Level1State loop

The input of the player is handled by the **KeyInput** class.
The input of the player is handled by the **MouseInput** class.

Changes from phase 1

Many small features that we wished to implement have been abandoned as time was limited:

- Reduced amount of levels from 5 to 1. However, we did implement the game in a way that allows more levels to be added in the future.
- Removed personal high score on levels as the implementation was more complex than intended.
- Removed in-game menu, tutorial state, exit button, and more.
- Removed select level screen
- Removed many unstable randomizations such as barrier randomization, spawn randomization.
- Traps no longer visually disappear when the game starts as it causes confusion.
- No longer have a handler class and enumeration as we found another way around it.
- Used a tile-based world system that allows flexible map generation.

Project role and management

We used “issues” on gitlab as a way to project manage. When a team member thinks of something we need to implement, they create an issue. This way, team members can claim issues they feel comfortable with and create a branch from it. It allows us to keep track of who is implementing which features, avoid duplication of implementation, and gives us a clear timeline. The separate branches also allow us to clearly revisit old code for reference and keep track of different versions. We also had meetings whenever possible.

External Libraries used

No external libraries were used in this project, however internal libraries were used to help create the game, with the libraries being **awt**, **swing**, **ImageIO** and **IOException**.

We also studied the basics of game implementation from: CodeNMore

---

Code Quality Considerations

We evaluated and considered many different options while programming to consider the best one. As an example, we changed which objects would be threads many different times to come up with the ideal amount, and created another class (Tiles) for graphics which will not become threads.

In addition, we evaluated any situation for the potential to implement any design patterns learnt about in class:

1) We recognized that the Game class creates one singular Game object which is then widely used by many other classes. Instead of constantly passing it through as a parameter, we made it a singleton class to be widely used and without worry that more than one will be created.
2) We also recognized that it is best to split the game into multiple states so it becomes simple when we transition the game from the level state to WinState, menu to level etc.
3) We actively used abstractions and subclasses to define the structure of our objects.
4) We used packages to group up the main “chunks” of our code.
5) We used javadoc to explain the functions and classes.
6) As we have the intention to add more features/ levels to our game in the future, we created many classes and methods that allow the game and the maps to be generated and modified easily.

Challenges

1) A few of our group members are new to java, so learning a new language can be quite challenging when implementing a project under a time crunch
2) As we have never implemented a game before, we had trouble visualizing how our objects were going to look when implementing our UML diagram which resulted in drastic differences when actually implementing our code.
3) This is the first large project for many of our team members, so managing git branches, merging process, and merge conflicts took quite a bit of learning and time.
4) Since there was not much guideline of what our games should be, it is hard to implement the game as a whole since we get stuck at times brainstorming ideas
5) The enemy’s pathfinding to the player was rather difficult due to our lack of knowledge when it comes to designing AI and pathfinding, making it difficult to come up with a solution for the enemy AI.
6) Threading was difficult for all of us since it was a relatively new concept and that learning the concept of threads took quite an amount of time for all of us. Implementing the

threads was also deemed difficult as the threads can create some unexpected errors due to the nature of threads.

7) Time management was an issue for all team members because everyone has a different time schedule, making it difficult at times to find time to start meetings and discuss the progress of the game.
8) Recreating the graphic design we wanted was much harder.