CMPT 276 Group 2 | Phase 2 Report

Overall Approach & Plan

Our overall approach to Phase 2 for our project focuses on creating a functional and maintainable implementation of our game in Java and JavaFX. We aim to create a solid foundation for the game's logic with a focus on our core gameplay mechanics in the first milestone. To establish a clean architecture based on our designs and packages from Phase 1, we will keep our logic decoupled from rendering. The core game classes are written independently of JavaFX, while the rendering layer (*game.ui* package) observes entity states and draws them on the screen, building around our pre-established logic. This design allows our game to remain flexible and easy to extend, with new entities, maps, or visuals added without altering the core logic.

For the **midway-progress deadline**, we aim to have a working prototype of the game with the following completed:

- Maven project setup fully configured with a *pom.xml* file
- Core packages (*map*, *reward*, *entity*, *core*, & *behaviour*) implemented with working constructors and movement logic
- Basic game mechanics, including:
    - Player movement logic using directional input
    - Entity interactions and their consequences (enemy and player collisions)
    - A minimal working, prototype pathfinding strategy for mobile enemy movement that pursues the player
- Roughly 50% of our entire codebase should be implemented.
- Sufficient code documentation with JavaDoc comments for all implemented classes

For the **final phase 2 deadline**, our game should be fully implemented. After the midway-progress deadline, we will focus on integrating and rendering the user interface with the following completed:

- Integrated JavaFX rendering with completed game logic, with a complete *game.ui* package
- Finalized game sprites
- Fully implemented scoring system, win/loss conditions, and game/board loading
- Fine-tuned A* pathfinding strategy for mobile enemy movement

Adjustments & Modifications

- Type *Direction* was not defined in UML - now defined in game.core as *enum Direction*, needed by game.core.Game and game.entity (might also be needed by game.behaviour).
- Encapsulation issues in game.core.Game (dot rule/law of demeters), as a result:
    - Added method *collidesWith(Entity e)* in game.entity.Entity

- Added the following methods to game.map.Map for convenience: *isBlocked(), isPassable(), setTile(), createBorder(),* and exit/entry point setters, and overloaded *getTile(int, int)* for cleaner code.
- Added *getPosition()* to game.map.Position that returns a copy rather than the Position object itself, preventing issues where multiple entities share the same position reference.
- Adjusted *isBlocked()* in game.map.Map to treat out-of-bounds positions as blocked walls, making the code more defensive so we don't have to use *inBounds()* before *isBlocked()* every time.
- Changed *totalRewards* and *totalCollected* to be static fields since they need to track all rewards across the game (global tracking).
- To decouple or logic from rendering, we added a *game.ui* package complete with the following classes: GameCanvas, GameController, GameWindow, GameApp, GameConfig, GameEndScreen, LoseScreen, MenuScreen, PauseScreen, ShowHowToPlaym, WinScreen HUD (Heads-Up Display), HowToPlayScreen, InputController, ResourceLoader, SpriteManager.
- Integrating the A* algorithm for pathfinding with MobileEnemy made it expose the player's position to pathfinder, so we replaced the bulky constructor with a Builder.
- We removed one of the types of moving enemies we originally planned to have in Phase 1 (moving boulders), as we didn't have the time to implement it and having the skeleton enemies seemed sufficient.
- We changed our regular rewards from golden coins to blue gems (just a sprite design difference - all logic is the same).
- We simplified the winning conditions so that the player doesn't need to collect a key to reach the final cell. Once they have collected all 10 regular rewards, a final reward (golden totem) spawns on the board (in the same position each time), and once they collect it they win.

Management Process & Division of Responsibilities

Throughout implementation we will commit and communicate our respective work often to avoid merge conflicts. We aim to review each other's code after another member commits to provide feedback, look over changes, and resolve issues as early as possible. We used Discord to communicate and a shared document to track tasks and progress towards the midway deadline. Work was organized around major milestones; first completing the core game logic, then focusing on rendering for the second half of the phase. Each member was responsible for specific classes or subsystems, with work done in both the logic and UI.

Member	**Core Responsibilities**
John	• Setup Maven project structure • Implement *game.map* & *game.reward*

	● Implement *game.ui.HowToPlayScreen, game.ui.GameConfig & game.ui.MenuScreen* ● Create the static enemy sprites and menu UI ● Troubleshooted and implemented fixes to enemy class behaviour ● Created spike, and floor tile sprites
Lex	● Implemented core game module in **game.core.Game** handling main game logic, entity management, map handling, scoring system, and collision detection. ● Started implementing the final reward system: created **game.reward.FinalReward** class for the golden totem and integrated reward collection logic with Game.java. ● Implemented main JavaFX application: created **game.ui.GameApp** handling stage/scene initialization, game loop, and scene switching. ● Implemented heads-up display: created **game.ui.HUD** displaying score, timer, and reward tracking. ● Implemented game state management: created **game.ui.GameManager** to handle starting, pausing, and restarting games. ● Implemented sprite and asset management: created **game.ui.SpriteManager** for caching and rendering tiles and entities, and **game.ui.ResourceLoader** for loading images and other assets. ● Implemented end-of-game screens: created **game.ui.GameEndScreen** as a base class, **game.ui.WinScreen** for victory, and **game.ui.LoseScreen** for defeat with final score and time display. ● Edited instructions screen: updated **game.ui.ShowHowToPlay** to display instructions with the same background as the main screen and improved colour scheme for readability. ● Created final reward sprites: added graphics for the golden totem used in FinalReward.java.
Sydney	● Implement *game.entity* ● Implement *game.ui.GameCanvas, game.ui.GameWindow, & game.ui.InputController* ● Create the reward (bonus and regular), moving enemy, player, floor tile, and wall tile sprites ● Write-up the report
Diar	● Implemented game.behaviour module with the A* pathfinding algorithm for intelligent enemy pursuit

	• Implemented core game loop in Game.java handling scoring system, collision detection, and win/loss condition evaluation (checkWin(), checkLose(), resolveCollisions()) • Implemented final reward system with door unlocking logic: integrated FinalReward in Game.java and dynamic door sprite rendering in GameCanvas.java that switches between locked (door.png) and unlocked (door_open.png) states based on reward collection • Created door sprites: door.png (locked) and door_open.png (unlocked) as well as earlier versions of wall and tile. • Implemented pause functionality: Created PauseScreen.java with time-synced stats display, integrated pause/resume logic via InputController.java, and implemented time preservation using Game.resetTimeStamp() to prevent time jumps when resuming • Implemented win/lose pop-up screens: Created WinScreen.java and LoseScreen.java with final score/time display and menu navigation • Created comprehensive unit tests: AStarPathfindingTest.java for pathfinding and game core functionality

Each member did much more than the core responsibilities above, taking on functionalities and implementing changes as they came.

External Libraries

JavaFX: we chose this library since it is one of the most widely used Java event-driven GUI systems, and its simplicity and support for modular design will help us build a clean 2D interface while keeping our game's logic and rendering layers separate.

Measures for Quality Code

To maintain high code quality, we emphasized clarity, modularity, and consistency across all packages. We followed strong object-oriented design principles including encapsulation and inheritance to reduce redundancy and simplify maintenance. We carefully reviewed classes to follow the Law of Demeter/dot product rule to reduce coupling. Each class follows a clear single-responsibility structure, with game logic and rendering kept separate. We implemented reusable utility classes such as *GameConfig* and *ResourceLoader* to centralize configuration and resource management, avoiding hardcoded values and redundant loading. Each class consists of descriptive JavaDoc comments and consistent formatting, helping ensure readability and easier collaboration across the team. Together, these practices improved the maintainability, scalability, and overall quality of our codebase.

Challenges

One of the main challenges in this phase was integrating the JavaFX rendering system with our existing game logic while keeping both layers decoupled. Since our design plan was to separate game logic from the renderer, we had to carefully manage how data flowed between the logic classes (*Game*, *Player*, *Map,* etc.) and the UI components (*GameWindow*, *GameCanvas*, *GameApp*, etc.). This required rethinking some dependencies to ensure that no logic classes directly referenced JavaFX elements, as well as creating many new classes to maintain reusability and follow the single-responsibility structure we were aiming for.

As our codebase expanded, keeping classes focused yet cooperative was challenging. We addressed this by refining package responsibilities and trying to maintain constant communication through Discord.

Sprite creation, UI integration, and gameplay logic all required extensive coordination. We tried to distribute visual and programming tasks evenly while also creating cohesive components.

Some other challenges included:

- Maintaining clear and frequent communication with all team members.
- Managing classes to have high cohesion and low coupling while also interacting with other packages to meet functionality requirements.
- Maintaining efficient code with low redundancy.