Phase 3 Report

This report discusses the testing phase pertaining to Group 6.

Features that need to be unit tested

The following are the features:

- The `Map` class allows for requesting a `Cell` in the map given some map position (`getCell()`).
- The `Passage` class changes its collision depending on if the player has collected every star (regular reward) (`isUnlockable()`).
- The `CollisionHandler` class deals with all the types of collision (main character-barrier, main character-reward, main character-moving enemy, etc.)
- The `Cell` class loads an image to display based on it's key on the screen and tracks its collision.
- The `FLoor` class is a `Cell` that doesn't have collision, acting as the path for the player to walk along.
- The `Barrier` class and `wall` class are `Cell` s that blocks the movement of the player, preventing them from moving onto that `Cell` . It should not be unlockable by the player.
- The `TrapSpawner` class creates traps in predefined locations and updates those traps according to the player's collision with one of the traps.
- The `SlimeSpawner` class creates slimes in predefined locations as well as a few in random locations and updates those slimes according to the player's collision with one of the slimes.
- The `StarSpawner` class creates star pieces in predefined locations and updates those star pieces according to the player's collision with one of the stars.
- The `RewardSpawner` class creates a coin every few seconds for a total of 10 coins. If there are 10 coins already, it will remove a random coin off the map and replace it in a new location. It also updates the coins according to the player's collision with one of the coins.
- The `Reward` class should update the players points when collected

Unit tests

The following are the unit test cases/classes that cover the features:

- In `MapTest` , the tests `getCellFromMapBoundary()` and `getCellTooFar*()` covers `getCell()` .
- `PassageTest` covers if a `Passage` changes it collision properly.
- The `CollisionHandlerTest` tests all the collision features of `CollisionHandler` .

- The `BarrierTest` tests to make sure that any `Barrier` s created have collision and can not be unlocked by the player.
- The `CellTest` tests to make sure each cell created has an image associated with it.
- The `EntityTest` tests to make sure that all the classes that extend `Entity` have constructors that won't result in a null object. The `FloorTest` tests ensure that all `Floor` `Cell` s do not have collision and can not be unlocked.
- The `GameTest` tests make sure that the game has the correct GameState with specific actions.
- The `SpawnerTest` tests make sure that all `Spawners` that spawn `Entity` s have their respective spawns after calling the necessary functions.
- The `WallTest` tests to make sure that any `Wall` s created have collision and are not unlockable by the player.
- `RewardTest` covers score updates

Interactions that need to be integration tested

The following are the interactions of our system:

- The `Map` loaded the map grid from the `/resource` folder (`loadGrid()`), now refactored to test `loadGrid()` .
- The `Cell` and `Entity` class and their respective subclasses load `.png` files (their sprites) from the `/resource` folder.
- The `MainCharacter` moves either up, down, left, or right based on player input from the keyboard (WASD). Refactoring was needed to have a "mock" keyboard, or programmatically provide key inputs.
- The `Game` , the `Map` , and subclasses of `Entity` all have a `draw()` method which draws some graphic or preloaded image on the player's screen. Unfortunately, we found this interaction difficult to test and was unable to create integrations test for it.

Integration tests

The following are the integration tests/classes that covers the interactions:

- In `MapTest` , the test `loadMapGridFromResourceFolder()` covers `loadGrid()` .
- In `CellTest` , the test `imageIsNotNull()` covers loading images from `/resource` s for `Cell` s.
- The `PlayerMovementTest` covers the interaction between player inputs and player direction/movement.

Coverage of test suite and overall test quality

We have about 58% line coverage and 28% branch coverage. The main reason for this is that it's difficult to write unit tests that simulate the game loop which calls the `draw()` and `update()` functions of all the `Entity` objects. Aside from the `draw()` and `update()` functions, we also were unable to write unit tests for our `KeyHandler` class since we were unable to simulate a key press from a keyboard using a unit test. Instead, we opted to test key presses indirectly by creating `PlayerMovementHandler` which abstracts the key presses away from the `MainCharacter` that attempts to simulate it. We also opted to not tests the getter and setter methods as we thought those would be trivial.

In an attempt to ensure our test quality, we tried to refer to our `Design/use_case.pdf` file to cover as much functional testing as possible. Then we opted to try and understand the input spaces of the methods we were testing in order to get most of the important combinations. We also tried to our code well documented.

Important findings

We had a number of refactors:

- A new `MapGridLoader` class was added to test `loadGrid()` which used to be a private method from `Map`
- A new `PlayerMovementHandler` class to (1) as the class is named, handle the movement of the player, and (2) to abstract the player key presses away from the `MainCharacter` class to actually test the player-controlled interaction.
- A new `StarSpawner` class was added to test the spawns for the star piece reward which were previously in the `Game` class. The necessary changes in other classes (`Game` `Star` `GUI`) were made to accomodate this change.