CMPT 276: Phase 2 Report

Approach to Implementation

The game we chose to implement was a 2D pixel-style game called “Monkey Escape”. In this game, the user controls the monkey protagonist whose goal is to collect keys to escape the zoo. During the escape, there are numerous opportunities to gain points by collecting bananas, but the player must be cautious to not get caught by the zookeepers. To start creating the game, we analyzed our UML diagrams and use-case scenarios created in Phase 1. Our first step in implementation was setting up our Maven build system. We then met as a team to further discuss our design and determine which classes had the heaviest dependencies on other classes. We used this information to create a schedule of classes to implement, where we focused on the classes that provided the core game functionality such as Game.java, Map.java, Panel.java, and Window.java and Entity.java. Our ramp-up was relatively slow as we wanted each team member to understand the base structure before implementing additional functionality. Next we implemented character movement and added the game graphics we created. After this, we began to go through our use-cases again to determine which specific features needed to be added to our currently implemented base functionality. After completing all use-cases, we added additional features to increase the quality of the game such as background music, sound effects, and game replay functionality.

Adjustments to Original Design

CHANGE: Added State class.
REASON: Once we implemented numerous game menus, we needed to be able to more closely manage the state of gameplay, and respond accordingly. To ensure we followed the principles of object oriented programming, we created an additional class to abstract the state change logic from the rest of the program.

CHANGE: Changed Pause trigger key from ESC to SPACE.
REASON: While implementing our menu feature, we noticed that having multiple sequential menus using the same key triggers resulting in unstable behavior. Sometimes the game would skip over the Pause Menu and quit the game, while other times it would follow the intended flow of menus. We discovered that this issue was likely due to the speed of our game thread reading the key while the switch value was still unstable (due to mechanical button bouncing). To reduce unpredictable behavior we changed our key trigger for the Pause Menu to be a different button than we use to exit the game.

CHANGE: Added a Pathfinding class for entity movement.
REASON: This decision both cleaned up the code by moving all pathfinding code into a new class, as well as allowed for further expansion of the game as any entity can allow pathfinding movement to move towards their goal.

CHANGE: Added a function to the zookeeper which checks its distance to the monkey.
REASON: This is used by the spawning function of the zookeeper. This solves the unforeseen bug we discovered while testing, that the zookeeper would spawn too close to the monkey making it not possible to win.

CHANGE: Added lifecycle variable to the Banana Class (Bonus Reward).
REASON: This was added to keep track of how long it has been alive. This way the banana is only on the map for a few ticks, and increases the difficulty to obtain the bonus rewards.

CHANGE: Added a random position generating function.
REASON: This function was created for the fixed entities (bananas, keys, and lion pits) created to find a position while being spawned in that is not already occupied and is not near the start or the exit. Having this randomly generated reduces the complexity of designing fixed generation patterns used for an unknown number of levels.

CHANGE: Added Sound class.
REASON: We added a class that controls the sounds in the game that was created from scratch as we did not have any plans for this in the original design. This was to enhance the quality and interactiveness of the game, making it a better playing experience. This change was accompanied by adding a method to play music in the game class which would begin to play the background music.

CHANGE: Added update level methods
REASON: We added methods in Game, Panel, and TileMap to help update the level. The method in game calls the method in the panel which creates the new entities and then calls the method in tilemap which creates a new map for the next level.

CHANGE: Substituted Menu class for Painter Class
REASON: Since we already have a class that handles the overall state of the game and our Menu screens are just images, we decided to create a Painter class instead. The Painter class consists of methods which get called by the Panel class to “paint” the appropriate menu images on the Panel depending on the game state.

CHANGE: Split Entity class into two - Moving and Fixed
REASON: We realized that moving and fixed entities have different implementations and functionality, including the way they are drawn on the screen, whether or not they require key input, etc… Hence, it made more sense to add an extra layer of abstraction by creating an entity interface and having the Moving and Fixed entity classes implement this interface

CHANGE: Changed the map from 2d array of Integers to 2d Array of Tiles
REASON: Integers did not hold enough information about what was on the tile so a Tile class was created to hold information such as, the image of the tile, and the Fixed Entity on the Tile if it had one and whether this Tile could block movement.

CHANGE: Separate Collision class was created to handle collisions
REASON: Collisions between a moving entity and a fixed entity, a moving entity and another moving entity and a moving entity and the background required types of collision checking beyond just checking the tile the entity is on. A new class Collision was created to handle each type.

CHANGE: MapGenerator class instead of a function
REASON: Creating a random map that connected the start and the exit and had enough space for the player to move through and dodge enemies required many different functions. So it made sense to move this functionality into another class. This also allows us to generate different types of maps in the future by just changing the type of MapGenerator used.

Management of Execution

To organize the execution of the implementation of our game, we generally followed the Agile Scrum Methodology. We first collected our team member's availability using when2meet, so we could determine when the best time to meet was. During the ramp up of our implementation, we met twice a week to give updates, discuss the project direction, and assign action items that we would have completed at the next stand-up meeting. To increase the efficiency of these meetings we created meeting notes so that we could track what we discussed, assigned, and completed each meeting. As the project progressed, we increased the frequency of these meetings to be approximately every second day. We found that meeting regularly for a short amount of time was the most effective for staying on schedule and provided ample opportunities to address any progress blockers or pending merge requests. When defining action items, we aimed to break tasks into encapsulated sections so we could divide them amongst the group and increase the parallelism in our workflow. Each member got 1-2 tasks, and was expected to complete them by our next meeting. To set deadlines we worked backwards from the Phase 2 deadline to define soft-deadlines for encapsulated modules. These deadlines took into account which features of the game were more important than others so they would get the most time for execution and be farther away from the real deadline. The tasks we planned to be completed closer to the real deadline had simpler implementation or were not necessary in the game's core function.

External Libraries

Java Swing (java.swing)
The main external library that we used for our GUI was java swing. The main reason for this is that it is very beginner friendly and lightweight. We used the java swing jpanel for our Panel class, and the java swing jframe in our Window class. Using the jpanel saved a lot of time for us

and made it easier to understand what is happening at a high level of the GUI. Moreover, using the jframe made it exceptionally easier to create the window for our game. Since both jframe and jpanel are from java swing, it was easy to implement both in our game as they pair nicely with each other.

Java Abstarct Window Tookit (java.awt)

We also used the Abstract Window Toolkit library to assist in handling keyboard input and painting 2D graphics onto our windows. SImilarly to the Java Swing library, we picked this particular library since it offered a very intuitive and easy way for us to monitor key input. We used both the KeyListener and KeyEvent interfaces to construct our own KeyHandler class which abstracted the logistics of processing user input from the rest of our program.

Quality Enhancement Measures

The main measure we enforced to increase the quality of our code was mandatory code reviews/merge requests before any change could be pushed to the master branch. This process improved the overall standard of code in our repository greatly. From a functional perspective, we were able to incorporate direct and timely feedback from other group members and utilize the group's combined knowledge base. In addition, we could catch and remove any remaining debug statements or implementation comments that were missed before incorporating the new feature into the code base. From a management perspective, these code reviews allowed each group member to be aware of the changes to the current state of the repository and feature implementation between meetings. Throughout our implementation, we ensured that the code we committed to master had clean javadoc comments. As a group we decided that we were not allowed to merge any new features until the appropriate documentation had been written. As mentioned in the implementation section of this report, we aimed to define and implement the core functionality first, such as character movement and map layout before adding additional system complexity. We found that this allowed us to establish a solid foundation to build off of, and reduce any major bugs during development.

Challenges

One of the biggest challenges we had was drawing out the UML diagram and planning our development of the game without having any game development experience. This caused the implementation to differ from what we planned out, which resulted in more time spent thinking about what the layout and structure of our code should look like. As a team we struggled to know how to start and what resources would help us achieve our goal. Ultimately, we spent a good chunk of our implementation time researching before we could implement anything. We found that implementing the first part of the game was particularly slow since the majority of the core implementation was dependent on each other, and blocked further work from being done.