CMPT 276 Project Phase 2 (Implementation) Report

Group 10
Ewan Brinkman, Ariel Lin, Taaibah Malik, Hoi Chun Hogan Mok

Overall Approach

Our overall approach to this phase of the project started with a meeting in which we established a timeline and a list of tasks that were required, then divided them amongst ourselves. We concluded that we would follow an agile development process. We discussed the IDEs that each of us uses (Intellij and NetBeans) to ensure we were able to integrate and transfer code between us and run it with minimal conflict. We began by reviewing our Design folder from Phase 1 and setting up the classes based on our UML diagrams. We discussed how best to implement the functionality in our use cases, as well as what libraries to use for consistency and efficiency when we got to integrating our code. Next, we started the coding process with parent classes and expanded to children classes based on our assigned tasks. Each person continually worked on their parts. As we made progress we held smaller meetings on our Discord server for merging our separate branches to try and minimize merge conflicts later on. Throughout the entire process, we ensured to communicate with each other about anything we were stuck on, merges, requirements, and so on.

Modifications From Initial Design

We primarily followed our UML diagram, but refactored some of the class/file locations for efficiency and ease of use. Creating modular code from the beginning of implementation helped minimize changes required. In the one-page statement from Phase 1, we planned to make twenty levels, but as we were incorporating a world flip in addition to all requirements, this was unrealistic with our time frame. We decided to create four levels - three “regular” levels and a boss level at the end. We also originally planned to include a boss fight and final treasure in the boss level. However, including the fighting system and the final treasure was unrealistic within limited time in phase two.

We added classes for the title screen, avatar selection screen, settings and info pages, game over screen. Most of these were considered in our use cases, but not in our UML diagram, so we had to plan out how to incorporate them into our existing design. The last modification we made to the original design was incorporating sound effects for passing through doors, the world flip, and the bonus reward.

Management Process

Our approach to the management process in this phase was to divide tasks according to the components that were most interconnected based on our UML diagram from Phase 1. We decided to create two subteams - Ewan and Hogan handled the game and levels, as well as incorporating audio into the game, while Taaibah and Ariel managed the title, settings, avatar selection, and some in-game UI. As mentioned earlier, we communicated with each other through our discord server, where we continually gave updates on what we were working on. A detailed breakdown of everyone's individual contributions are displayed below:

Ewan:

- Created the initial app window with game logo
- Setup the game loop, which runs at a speed based on the FPS
- Implemented the game physics, using physics kinematics equations. Entities only control their acceleration, which in turn modifies their velocity, which affects displacement
- Implemented drawing all entities and tiles to the screen. Made a base Drawable class to handle animation frames. Also, created a sprite class that stores the position, any amount of hitboxes, and can be drawn.
- Made it so all tiles and entities can be animated.
- Loaded in tilemaps, and created an abstract tilemap and extended it to load in specific sets of images (such as character images) for our used dungeon tileset.
- Loaded in player images, and loaded in all non-boss and non-enemy images with proper hitboxes.
- Created the entity class, a base for non-tiles sprites that are drawn. Furthermore, created a MovableEntity class that handles movement for all entities, based on each entity's physics vectors. It can also draw its hitboxes and physics vectors.
- Handled collision detection between sprites, and made all movable entities collide with wall tiles.
- Dealt with creating a thread for the game loop, and avoiding errors between the GUI thread and game thread
- Created the game over screen, displaying game results and safely going between threads.
- Added items, including their bob functionality using sinusoidal motion.
- Created enemies, which track the player. Made enemies add like electrons and repel themselves, so they don't bunch up.
- Created config JSON files to easily change settings. Also, created a class to easily read in resource files, and config data.

- Created classes to store each level, each level's tile maps, and each level's tile layer.
- Implemented the special flip functionality of our game. Made sure to only flip if a player won't flip onto a wall. If the player attempts to flip onto a wall, made the game draw the other tile layer with a transparency value, and play an error sound. If an enemy flips onto a wall, made them freeze and become transparent.
- Implemented a scrolling camera, and to only draw sprites on the screen for efficiency.
- Detected game input, and stored it in an easy to access way.
- Made generous use of enums to avoid errors, such as for sprite animations, input types, tileset types, audio types, directions, enemy types, and player types.
- Made the mute button apply globally.
- Implemented the drawing of the current level, and switching to the next level.
- Developed util classes for detecting collisions, flipping animations horizontally, and vector calculations for scaling, and getting vector angles and magnitudes.
- Designed the format to store level and level map data.
- Created a class to manage the current level and store all levels.
- Added some Javadoc comments.

Hogan:

- AudioManager and Audio
    - Finding different soundtracks for the game and checking their licenses
    - Created audio functions: playAudioOnce, playAudioLoop, stopAudio, etc.
    - Created a function in recourcesReader to read wav file
    - Playing different audios depending on the current game scenario
- Created different levels
    - Including two maps (main map, flipped map)
    - Displays the maps in layers
    - Includes different entities in level.json
- Punishment class and Punishment Factory
- Door class and Door Factory: Allows users to go to the next level
- Tile hitboxes: Adding the array of hit box for every tiles we may use
- Creating different levels with location of entities, and designing maps with arrays
- Creating a function to create current level's entities based on the json file

Taaibah:

- TitleScreen class: displays the game title and methods to access the settings and info, avatar selection, and start game classes.

- Settings class: Allows users to turn audio on/off in game, go to the “how to play” page and return to the title screen.
- HowToPlay class: Explains how to use features specific to our game and includes a method return to the Settings screen
- ChooseAvatar class: Created methods so user can select one of three avatars to play with in the game and return to the title screen, and applied to Game class
- Incorporating all classes listed above in the App class to run properly with the Game interface.
- Designing background images for all non-game screens (TitleScreen, Settings, HowToPlay, ChooseAvatar, GameOverScreen) with Tiled
- Creating a level with the map - two maps per level
- General debugging and contributing to the report

Ariel:

- Fixing a bug with wav audio files

External Libraries and Frameworks Used

The only external library used in our project is the org.json package. We used this library so we could easily read in JSON data of levels and config information.

- groupID: org.json
- artifactID: json
- Version: 20220924

In addition, for the GUI we used Java Swing, which we did not have to add as a dependency to our pom.xml, as it is an internal library.

Measures Taken to Enhance Code Quality

- For calculating distances, compared the squared distances instead of taking square roots
- Create an abstract tileset so that more tilesets can easily be added.
- Have animations automatically done by a drawable class
    - Entities only need to provide animation images and the order of which image indices to play, and the rest is handled.
- Have config JSON files to easily change game and app settings. This data is easily accessed in the code.
- Have util classes for various methods that don’t belong in any one class.
- Generous use of factories for creating sprites.

- Use enums for things such as defining animation names, for names of entities such as player and enemy types, for tileset types, and audio names. Using enums instead of strings helps avoid errors
    - Classes that use animation enums have a generic type to be the animation enum type
- Each sprite can have as many hitboxes as needed
- Fast tile collisions by only checking nearby tiles
- Efficient drawing by only drawing sprites on the screen.
- All spirits and entities inherit from base classes which handle common animation and tile collision detection behavior.
- Custom entities can easily be added very fast by extending base classes which cover almost everything. This is useful if one wants to create special tiles and items.

Challenges

Throughout Phase 2, there were some issues that we all faced, the most prominent one being git merge conflicts. As mentioned earlier, each of us continually merged individual branches to the develop branch to reduce merge conflicts later on. However, there were a few times where resolving conflicts took over an hour. What really helped with this was holding calls whenever we wanted to merge - having two or more people look at the code with different perspectives helped to find issues significantly faster.

Another issue that was particularly difficult was dealing with concurrency and threads. Having both the game thread and Java Swing's thread need to use the same data caused a ConcurrentModificationException. It was time consuming and difficult to figure out the best way to deal with it. Additionally, the threads caused multiple problems when trying to switch from the game to the game over and title screens when the game ended. A method couldn't simply be called to display the game over screen at the end of the game's run method, since that method was associated with the game thread, and not the Java Swing thread. Instead, an event update had to be sent to let the App.Java class know when the game ended.

A third issue we faced was creating the maps - it proved to be difficult to choose the locations of the walls and put them in the right places, as well as making sure the array sizes for each level were correct.

Lastly, we struggled to find different soundtracks that fit our game's theme and weren't copyrighted. Our first audio choice for the game was so large that it delayed the game compilation significantly, so we had to search for audios with smaller file sizes.