CMPT 276 Phase 2 Report

You will create a plan for at least two milestones for your construction, with the last milestone being the full implemented game. Include your plan in the report, particularly specifying what you plan to achieve by the midway-progress deadline.

Milestone 1: Implement core classes as defined in the UML diagram and create basic functionality for all of the Animate and Inanimate objects. Also create early movement mechanics and map generation.

1. Discuss with group, distribute workload and create the milestone plan.
2. Create core classes and begin adding rudimentary functionality like drawing objects to frame or panning the camera.
3. Create object textures/images/sprites that will be used in-game for enemies, the main character, the walls, the floor etc.
4. Add main character, and implement his in-game movement and collision mechanics
5. Add enemies, rewards and traps and implement their in-game logic such as removing score and collision mechanics.

Milestone 2: Create the real map and tweak the movement mechanics of enemies and implement the GUI along with Sound effects.

6. Create the Haunted House map by adding walls, floors, candy, black licorice etc.
7. Add enemy tracking main character functionality.
8. Add sound effects and spooky music
9. Display User score in game
10. Add endscreen, title screen, and pause screen.

By the midway deadline we planned to have up to and including the last step in milestone one completed. On the full phase 2 deadline, we finished up to and including step 8 since the enemy tracking functionality was harder and took longer than expected. We also thought that UI changes were fairly easy to add and wanted to make sure that we had the core implementation of the enemies done right before moving on.

– describe your overall approach to implementing the game.

Before full implementation of the game, development of our game was an iterative process constantly refining on our previous plan. Our UML design was inadequate, and needed major changes. Although we had an abstract factory design pattern in mind during development it was not applied to the game in its entirety, because core functionality like collision detection is a mechanic shared between not so related classes such as animate, inanimate, and tiles (map asset). So we kept abstract designs to classes and their super classes, i.e., Animate class being an abstract class for character, zombie, and ghosts.

Our approach to implementation was to use Jpanel, and Jframe to create a game window. GUI development and implementation using Jpanel and Jframe allowed us to change features such as background color and placement of UI elements. We also used a GameScreen class that instantiated all classes used in our game, and implemented methods that are used in the game window. The GameScreen

construction created properties for game screen (Jpanel, and Jframe), keylistener (to get user input for keyboard control), sound, and general startup. Additionally, run(), update(), setup(), and startGameInstance() are methods from this class that allow us to fully control our functional implementations of our game. Other imports such as Color, Graphics, Graphics2D were necessary in conjunction with Jpanel and Jframe for GUI implementation.

Implementation of map using .txt file was made possible using java library imports like, InputStream, InputStreamReader, BufferedReader. A .txt file made up of 0's and 1's (0 for floor, and 1 for wall, and more to be added later) is read using InputStream and Buffered reader and parsed into an int 2D array to generate a 2D tile map. .txt file based map generation allowed us to easily generate and create custom maps that were often used to test for game functionalities like collision detection and object generation and removal.

Collision detection was implemented by creating a Rectangle and by using the size of the PNG asset. PNG tile assets for floors and walls are a 16x16 pixel png that is scaled up by 3, similar to character and monster png assets. Because walls can use its entire pixel size as collision hitbox java Rectangle function was not required. However, although character, enemy, objects are of the same pixel size as tile assets they differ in a way that some of the pixels are transparent. In order to create a collision detection for such classes required rectangles smaller than the scaled up size to account for transparent pixels present in character, monster, and object 2D models. Upper, lower, left, and right perimeter of the rectangle was then used to calculate collision between animate-animate, animate-inanimate, and animate-tile.

We used raycasting for the enemy class path finding in our game. To implement raycasting java library import Line2D were used. Raycasting was used in contrast to other algorithms such as BFS, A*, and Dijkstra because it suited our game design. Beginning of phase 1, we decided on two types of monsters, zombies, and ghosts. Where ghosts can go through walls; while, zombies can't go through walls and are slower and ultimately dumb (being a zombie after all). We tried BFS and A* for path finding but could not make zombies appear dumb as algorithm logic made zombies constantly move towards the player. After implementing raycasting, which was simpler to implement as vector distance calculation was all that was needed to generate the shortest path, it allowed us to customize monster characteristics. Implementation of the dumb characteristic of zombies was solved using intersection points between two raycast lines. Because walls have their length recorded as a raycast line, comparison between this and the path towards the player can be made, if an intersection between path line and wall line exists then the zombie halts and begins a random movement algorithm which involves random number generation to change direction.

Lastly, sound implementation used AudioSystem, AudioInputStream, Clip java library imports to load .wav files where background music loops iteratively.

– state and justify the adjustments and modifications to the initial design of the project (shown in class diagrams and use cases from Phase 1).

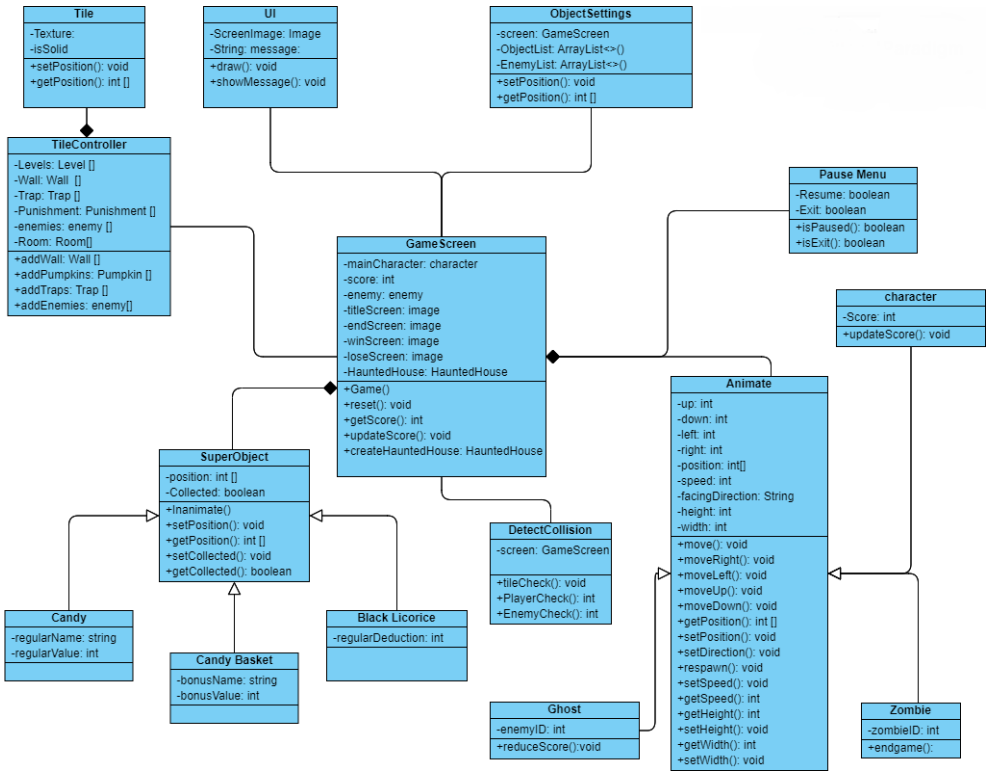
We decided to slightly change part of the UML diagram from phase 1 since after looking into actually coding the project, some classes don't make sense or were redundant. We also added a few new ones that

we didn’t know we needed back in phase 1. The main changes were that we combined the wall and room classes into a single class called Tile as well as added new classes to draw/update the frame and handle the GUI. We chose to combine the wall and room classes since they simply don't need their own class. The logic for the rooms can just be implemented using only a sequence of wall tiles. Furthermore, wall was changed to Tile inorder to generalize the block. Since the tile didn't have to be a wall, it could be a floor tile as well. As a result of this change we also implemented the TileController class to handle the tiles. Furthermore, the TileController class ended up replacing the Level and HauntedHouse class in our old UML diagram simply because it held the outline of tiles that created the map. Hence, the Level and HauntedHouse classes were redundant and had no purpose when TileController existed.

Next we added a bunch of housekeeping classes that deal with functionality we didn't think of during phase 1 such as a DetectCollision class used to determine the collision of the player, enemies, rewards and walls, a ObjectsSettings class that create and places and ArrayList of enemy, reward and trap objects. Also a UI class that handles the candy count, and a Sound class that calls the sound resources we used. Or lastly, in the future the endscreen, title screen and pause screen classes for our game.

On top of these changes, we named the Game class to GameScreen, since it made more sense seeing we generate a windowed game when running our code and we named Inanimate to SuperObject. We also removed classes such as Reward, Punishment, and Enemy once again to reduce redundancy as all of these objects can inherit directly from their parent classes SuperObject or Animate.

The new UML diagram looks something like the following:



– explain the management process of this phase and the division of roles and responsibilities.

As a group our project management process includes a 5 stage process to complete phase 2. First stage was initiation, where we defined the project scope, objectives, stakeholders and the expectations (part of phase 1). Second stage was Planning, where we created a detailed project plan outlining what we have to complete and the deadlines. As well as what each team member's responsibilities for phase 2. As for our source of communication we decided to use Discord and had weekly meetings to see the progress everyone made with their part. When facing an issue everyone resolved the issue through a scheduled meeting over a call. Third stage was Execution, during this stage we executed our parts of phase 2 and gave feedback over discord to the other group members. We shared our work using github. Before committing any work every member gets notified through discord to avoid merge conflicts. Fourth stage was Monitoring and controlling. During this stage we ran our code to make sure there are no errors that anyone missed. During this stage we made sure that the standards of our game are being met. Fifth stage was project closing where we made sure all the requirements have been completed for the project and worked on finishing the documentation on the project.

Division of roles was based on UML diagram, where Savinu Weerabaddana Dissanayake was responsible for inanimate classes and its subclasses along with the codes associated with the inanimate classes, Xander Soriano was responsible for map creation and associated classes like walls, doors,etc along with the code associated with it, Joshua Kwon was responsible for animate classes, its subclasses, and the codes, and lastly Samuel Yang was responsible for the GUI and associated classes and codes. Additionally, teammates worked collaboratively with frequent weekly meetings and constant discord chat to exchange questions and discuss solutions to game implementation challenges. It was a shared management process among teammates to keep track of finished and unfinished portions of the game.

– list external libraries you used, for instance for the GUI and briefly justify the reason(s) for choosing the libraries.

In order to implement the GUI, we used the Java swing framework, namely JPanel. JPanel was great since it's commonly used to create lightweight window based GUIs, which was exactly what we needed. JPanel had great customizability in its paint/draw functions and had many built in event handling functions which made the implementation of creating the map, moving the animate objects and panning the frame very easy. Furthermore, Java swing was great since it's already a part of the core foundation classes of java so no additional requirements or downloads were necessary.

– describe the measures you took to enhance the quality of your code.

In order to enhance the quality of code, communication played the biggest factor. We were always bouncing ideas off of each other and asking for one another's opinions of the sections we programmed. We mostly communicated through discord chat, which was efficient enough and created a log of potential improvements and comments that would benefit each group member. While communicating via discord, we would also update and let the group know what we did so that they had an understanding of what's going on in other sections as well as where we are at in terms of milestone completion. At the start we also made sure to specifically assign each group member a section so that way there would be no overlap

or confusion on who is doing what. This also helped with reducing redundant work and merge conflicts since only one person would be working on a section. Only when a member would ask for help, other members would edit a different section than their own. Lastly, we chose to do a top down approach when building the game. Meaning we started with a JFrame and added features to it such as the characters, camera, enemies, and inanimate objects. This greatly improved the quality of the code since it allowed us to continuously test and implement. Whereas if we were to do a bottom up approach, creating each subclass first, we would have to implement the entire thing before being able to run. This method definitely reduced the amount of compilation errors we got

– and discuss the biggest challenges you faced during this phase.

Each group member faced different challenges such as file path errors with images when pulling the repo, merge conflicts, Maven versioning differences and errors that weren't showing up on the VSCode or Eclipse IDE. There were also many pull conflicts and a lot of instances where we overwrote code we wanted to keep. But the biggest challenge was definitely planning out and managing the time it takes to do certain tasks. For example, when we were trying to implement enemy tracking past walls, it took us way longer than expected and actually prevented us from completing the GUI by the end of phase 2. Time management in general was also difficult since all students had other classes, assignments, work and midterms going on.