Our approach to the game and division of labor

When we began making our game, we started by creating all the classes that we would need to use and filled them with outlines of the different attributes and methods they would need with comments on what they would do. While working on the outline of the game, we would come up with more classes that we would need to create to make the game function properly that were not included in the design. Once the layout of most of the classes and all their components was completed, it was easy to see all the different sections that needed work, so we split up and each implemented our own parts on our own branches.

We decided that Eric would implement the engine as he had the most experience in general and because the engine was the most complicated aspect of the game. Meanwhile, Cameron and Jimmy, both having about the same amount of experience and being new to java, would work on the game's logic and associated classes. Cameron focused on the characters' logic, such as how different characters interact with each other and move around, while Jimmy focused on the pathfinding algorithm used mostly to find a path from an enemy's position to the player's position. While we were all working on our own sections, we made sure to keep in constant communication so that we were able to help each other out, as well as relay any design decisions that one of us had made and would affect the others.

Once enough of the work was completed on our sections, we merged together what we had so that we could test all of our components. It was hard to test our work individually as they all needed the other sections that our teammates were working on in order to function. After we merged our code and fixed any errors, we had a working base game on which we could then start implementing more features and immediately test if they worked or not. From here we added things such as interaction with objects, UI elements, 3-dimensional walls, and more until we were complete with phase 2.

Adjustments and modifications to the initial design

Our game largely implemented what we had designed it to in the UML diagram and use cases. The player is able to move a character around, fight enemies, interact with items, end the game by getting a key reaching the exit, etc. However, when we started writing code and implementing our designs, we realized our UML diagram was fairly insufficient. The game logic needed many more classes to store and generate the map and lists of characters for each level, and the game logic driver class itself needed lots more methods to run the game. Additionally, the UML diagram did not include any classes related to rendering the UI. Thus, adding in more classes related to the game's logic than the initial design called for was crucial to have a functioning game, and creating classes that dealt with having a UI made the game much more user friendly and enjoyable to play. Ultimately our idea of how the game should work remained the same but our initial design failed to encompass everything that was needed.

Libraries used

For sound and rendering libraries we used LWJGL(Lightweight Java Game Library). Mainly the libraries interface with OpenGL and OpenAL. Using these libraries we built the engine that our game runs on.

Designing the Engine

The core design of our engine is the ability to extend a single GameObject class into any component of our game that is required. This ranges from things that you can see to things to things that are abstract such as game logic. We designed the engine to be able to track these GameObjects through the Scene class which also allows the engine to track whether or not these GameObjects should be rendered.

In order to achieve this one class extending into all the things you need to create a game with, we implemented several core methods in each GameObject. Update, start, and destroy.

- The update method allows us to implement logic we wish to be performed on our GameObject on each logical loop. The specific implementation is that the game engine attempts to call a specified number of updates on each GameObject within the Scene class every second.
- The start method allows us to implement logic we want to be called only one time when the object is created within the scene.
- The destroy method allows the game engine to properly destroy the created GameObject, especially if it is a composite GameObject construct of many GameObject components. This method would have to be specified by us when we design the GameObject.

With these methods, any GameObject we create will be able to process logic on every logical tick, can be properly initialized, and can be properly destroyed. However, there is still the aspect of the human interface that is not captured by just this design above.

To properly provide an interface to working with peripherals, we implemented LWJGL listener methods to read keyboard input. We originally intended more but had to cut those implementations due to time. However, LWJGL listener methods cannot directly interface with our GameObjects and thus we implemented listener methods within each GameObject that would be called by our engine when LWJGL listener methods were fired. The overall design:

LWJGL → Engine → GameObjects

We added override methods such as onKeyDown(int key) to our GameObjects that allows us to create GameObjects that would perform logic when a key is pressed down. These methods allow each GameObject to respond uniquely to human input.

The final step was allowing simple methods to add and remove GameObjects from our Scene class for rendering and tracking by the engine. Part of this design was deciding how

much responsibility was on our users side when creating a new GameObject and how to handle its deletion. Thus we only created methods to put and remove the GameObject from the Scene of the game.

Measures taken to enhance our code

While writing our code, we would try to abstract as much as we could for objects with shared properties. One example would be the different types of enemies like minions and bosses, which inherit common traits between all enemies and use polymorphism to override certain methods to change their behavior. We also implemented some classes using the singleton design pattern as, for example, we only needed one object that would run all of the game's logic. In terms of documentation, we would write javadocs and inline comments describing what exactly something did before or immediately after writing a class or method so that they were easily understandable for our teammates and for our future selves if we needed to work on it later on.

Biggest challenges during this phase

As mentioned before, one challenge all of us faced initially was the inability to test our code that we created and receive feedback on how well it functioned when we were all working separately. This problem was soon taken care of when our work was merged together to create a basic functioning game.

Building a pathfinder algorithm was another challenge because when implementing a pathfinder algorithm, one has to consider the structure of the other parts of the game and build the algorithm in a way that will not break the game. The A* path finding algorithm was a known solution suited for this kind of problem. Still, we faced a challenge in implementing it because our game required a solution adapted to it's structure. Ultimately, we built our own algorithm that accomplishes the task, even though we still need to optimize its functionality.