Overall Approach

Since no one in our group had ever developed a video game before, we took inspiration from the YouTuber RyiSnow's video series "How To Make a 2D Game in Java" to build the foundation of our game and learn the basics of game development in Java. In addition to this, our team also utilized their tile editor program which allowed us to import our own tilesets and visualize our game maps rather than using a text editor and imagining the game level in our heads. After getting our feet wet, we started to see the limitations of RyiSnow's tutorial series and began to deviate from it, adding our own custom features and mechanics and rewriting our games code to be more efficient and adherent to software engineering principles. Our group started by developing the fundamentals of the game which included programming its game engine, collision detection, object interaction and other critical systems. Once the core of our game was functional, we began implementing more specific components and features such as user interfaces, audio management, and enemy AI.

Changes From Phase 1

Naturally, as novice game developers, our initial plans and designs of the game were rudimentary at best and thus subject to many changes throughout the second phase. Our most prominent changes were made to our classes. Many classes and methods shown in our UML diagrams were ultimately not included as they were not necessary for the development of our game. These included classes like ButtonType, Camera, CollisionType, Direction, GameState and Position. Many of the attributes and methods that were planned to be housed in these classes were implemented in other objects to make our code more succinct. Other classes were eliminated from our game design as they proved to be superfluous.

Our games use cases saw some major adjustments as well throughout phase two. At first, we planned for the player to be able to interact with the game's menus using their mouse. This was altered so that the player must use the WASD or arrows and the enter key instead since it would streamline the games control scheme and allow the user to play the entire game without the usage of their mouse. Only allowing keyboard inputs furthermore simplifies the process of adding controller support in the future as the keys only need to be remapped to the various controller buttons. Another use case that was modified was the aftermath of entering an exit cell. Rather than restoring the player's health after completing each stage, their health points would carry over from the previous level. This change was made as we thought the game's difficulty would be too easy.

Many smaller changes were also made to the design of our game as we progressed through the development process. Originally, the playable character was supposed to be a goose but was changed to be a chicken instead as we were unable to find suitable assets for goose sprites that would fit the game's retro art style. The bonus reward was previously an apple but was changed to an egg later on as we thought it fit the theme of the game much better.

Management Process

Our group knew that organizing the development process of our project would be indispensable to the success of our game. We created a server for our group on Discord so that we could communicate easily and quickly with each other about our project. In addition to this, our team also scheduled in-person meetings in the library group study rooms two or three times a week so we could work together more conveniently and effortlessly. These in-person meetings and work periods proved vital to the creation of our game as it allowed us to make important decisions and changes to our design promptly as well as help our group utilize collaborative software development techniques such as pair programming. Eventually, we began using the work management website, Asana, to partition our workload amongst ourselves and keep track of our assignments to complete at home before our next in-person rendezvous.

Each member of our team was delegated specific components and features to develop so that every person we had an equal amount of programming to do. This was also to ensure that our group members wouldn't be working on the same classes or methods, reducing the probability of potential merge conflicts. For instance, when we began implementing the core features of our game, our group divvied up the work as follows:

Person 1: Game loop and key input

Person 2: Animation and game sprites

Person 3: Game world, tiles and camera

Person 4: Collision detection and audio

During our in-person meetings, we would explain what features and components were completed since our last meeting and discuss any possible issues we ran into.

As development proceeded, our team members mostly continued updating and maintaining the sections of the codebase that they implemented earlier since they would be most familiar with that segment of our game. By adding plenty of comments to our code, communicating on Discord and in-person, members of our team could work on other sections of our game as well if any one of us needed extra assistance to solve a bug or implement a feature. This worked effectively as in most cases, a fresh pair of eyes and influx of new ideas from each other helped us quickly remedy bugs and improve our overall code. As a group, we would decide what improvements and changes to our initial and overall design should be made and who should be responsible for implementing them. Moreover, we also assigned specific tasks and parts to develop when it was not clear who should work on what.

External Libraries

In order to reduce the overall complexity of our codebase and limit the possibility of being stuck in dependency hell, our group tried to only use dependencies included in the Java Standard Library. However, sometimes this was unavoidable. For example, our game uses Jackson-Databind dependency to read and write to a JSON file to store the player's settings. This library was chosen as one of our team members had previous experience with it and internet benchmarks had shown that it was the most efficient option when compared to other JSON libraries such as GSON or JSON.simple.

Code Quality

While we tried to observe software engineering principles as we developed our game, our group preferred to follow a program first, fix later mindset. Our goal was to implement the majority of our game first. We decided that doing things like writing Javadocs during this stage of development would be a waste of time as our codebase is still volatile and susceptible to large changes. However, during this process, we still tried to make an effort to make our code readable and understandable by adding comments and descriptive variable and function names. Our group also heavily utilized helper functions such as getters and setters to reduce the length of our code and code reuse as well as keeping things organized in general. Related classes were also packaged together to further organize our code. Nonetheless, many of our solutions and bug fixes are ad hoc and palliative, and will have to be updated once we figure out a more elegant method. A major problem we currently have in our codebase is that many of our classes are not encapsulated properly; many attributes and methods are public when they should be protected or even private in some cases. Now that our game is functional, we plan to refactor and revise large portions of our codebase in the next phase, hopefully making our game code more efficient, readable and maintainable in the process.

Biggest Challenges

As amateur game developers, we faced many challenges throughout the development of our project. One of the biggest challenges our group dealt with was tracking down bugs related to our collision detection and pathfinding implementations. There are so many specific situations and scenarios that could occur during gameplay which made it very difficult to narrow down what was specifically causing a certain bug. When we attempted to fix said bug, we often accidentally introduced new issues which cost us a lot of time tracking them down. This cat and mouse process became very frustrating since the solution was more often than not just moving around or changing a couple lines of code. However, figuring out where the problem was located, what classes and objects were causing these bugs and which lines of code had to be corrected or moved around regularly took hours of thought and research.