Overall Description

Ghost Town is a 2D arcade style game heavily influenced by the likes of Pac-Man and its contemporaries. The story behind Ghost Town involves an outbreak of a deadly disease in a small town. The infection is spread through viral gas clouds and infected food. The player controls the Doc (short for Doctor), who is seemingly the only one who hasn't been infected. As the Doc, the player must escape the town. To protect himself and his friends from out of town, the Doc must collect face masks scattered around the town. Only once the Doc has collected enough face masks, can he leave.

The game includes three levels, each with a different map layout containing barriers, enemies, rewards, and punishments. The difficulty of the game gradually increases with each level. To win the game, the player must move around the board and collect all regular rewards (face masks). Additionally, the player can collect bonus rewards (experimental vaccines) to increase their score and restore health. The board also contains punishments (infected meat) and moving enemies (virus) that, on collision with the player, will reduce the player's health. If the player's health is reduced to zero, the player loses the game.

Changes

To a large extent, we have been faithful to our original plan and design since the game was implemented with reference to the UML class diagram created during phase 1 of the project. However, some modifications have been made to the original planned structure of the system during implementation in phase 2. This is because we were not experienced with designing the entire structure of a system before any implementation so creating a complete UML class diagram when we were planning was difficult. Some necessary classes and properties were left out as a result.

During implementation, we added some classes to handle map generation using the factory method. The classes for generation of maps were not included originally because of oversight. The concept of a map is more abstract as it is only a collection of game objects such as barriers and rewards so we neglected to assign classes for map generation when planning. Handling map generation using the factory method makes our code more robust, less coupled and easier to extend.

We also created helper classes for playing sounds, loading sprites and implementing breadth first search which increases modularity. Maximizing code reusability and modularity was not considered in-depth during planning so the original UML class diagram did not include helper classes.

We made some classes singleton to more easily implement some features.

Originally, we planned to add more features such as power-ups, special enemy types, character upgrades and more. However, it turned out that the game is harder to implement than we thought and we overpromised on adding those features that would require substantial time commitment.

Important Lessons

Since most of us have never worked on a project of this kind of scale before, we all learned just how important writing structured, scalable code is, especially when working in a team.

Before this project, some of us had mostly avoided using IDE's when writing code but when working on this project in Eclipse, we realized how much value an IDE can bring. The convenience afforded to users in importing packages and especially in debugging were some of the huge advantages we realized when working on this project.

In our past assignments and projects, testing was largely unstructured and overlooked. Through this project, we learned the value in designing and implementing good unit and integration tests. Allowing us to make quick changes, run our tests, and alleviate most of our worries.

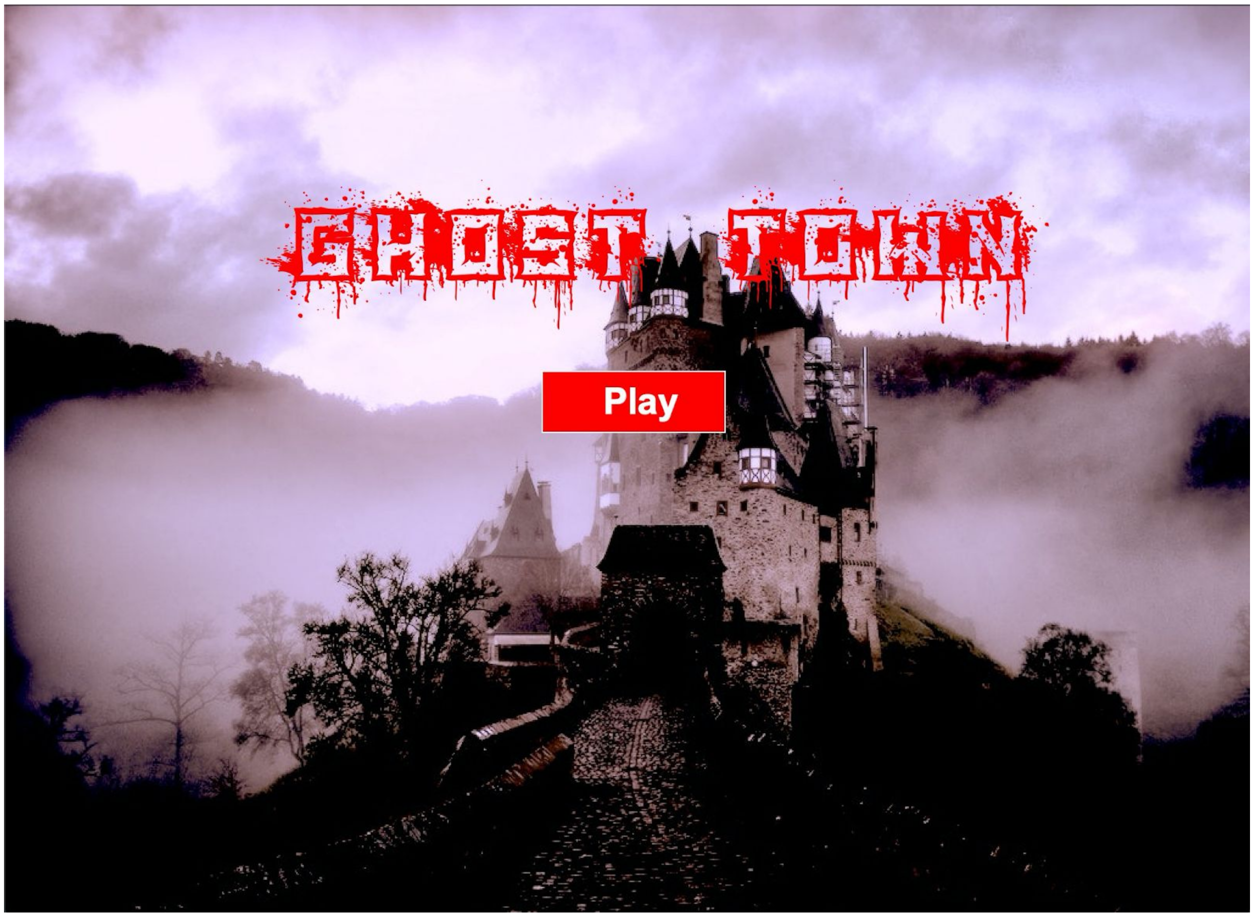
Most of us had limited experience with GitHub but through collaborating on this project, we learned the importance of frequent commits and merges, branches, good commit descriptions, and more.

Moreover, we have learned how to use Maven to manage a project on a team. Before that, we weren't using any build automation tool. Maven lets us realize how much power and time saving a build automation tool can bring.

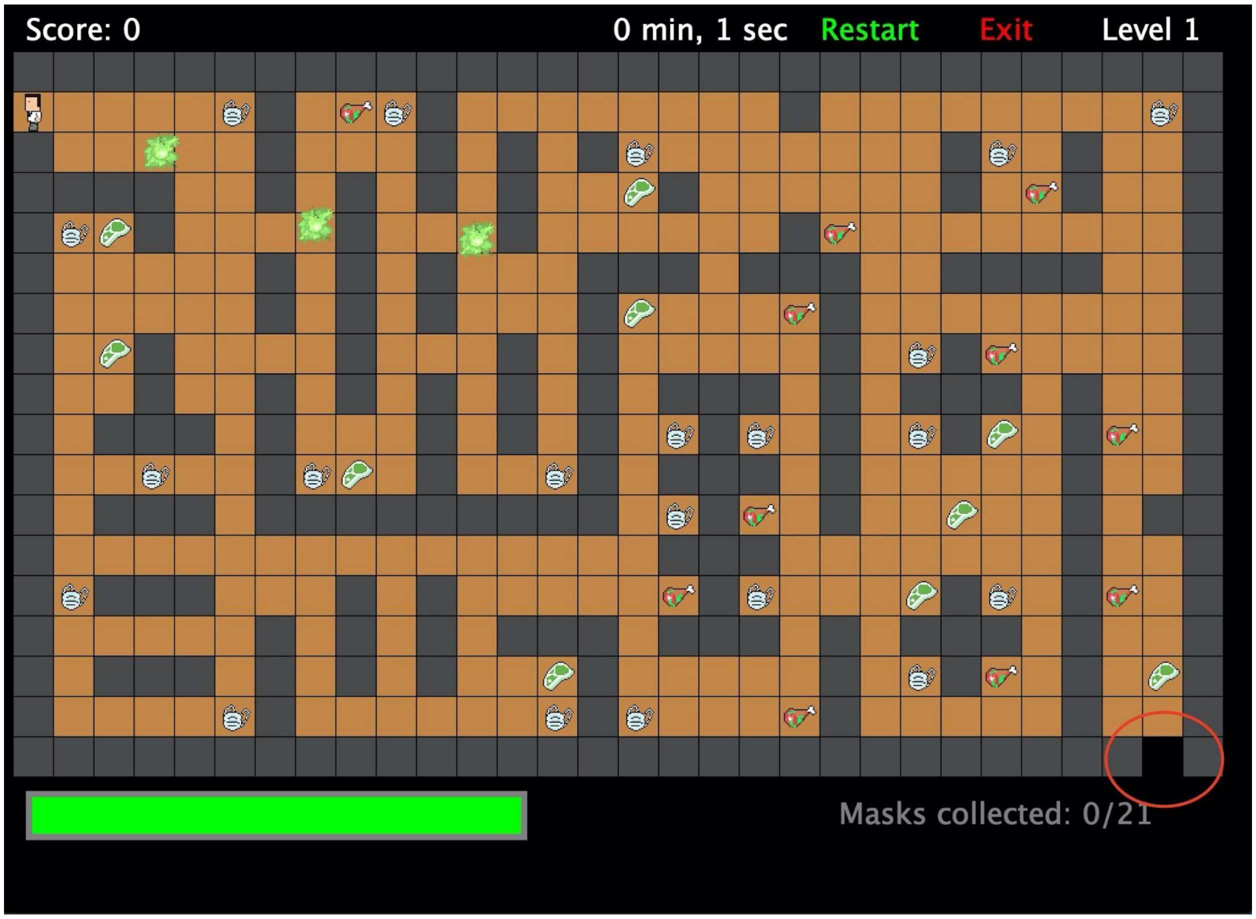
Design patterns can not be left out. We have successfully seen the beauty and ease of mind these design patterns, such as the singleton pattern, can bring to the code.

Last but not least, we all have learned how to work as a team. Whether it is on the coding level, such as git collaboration, or on the interpersonal level, such as using proper communication, this experience has let us know the importance of teamwork.

Tutorial



The above screenshot is the first screen after a user launched the game. The user needs to click the “Play” button to go to the next screen.

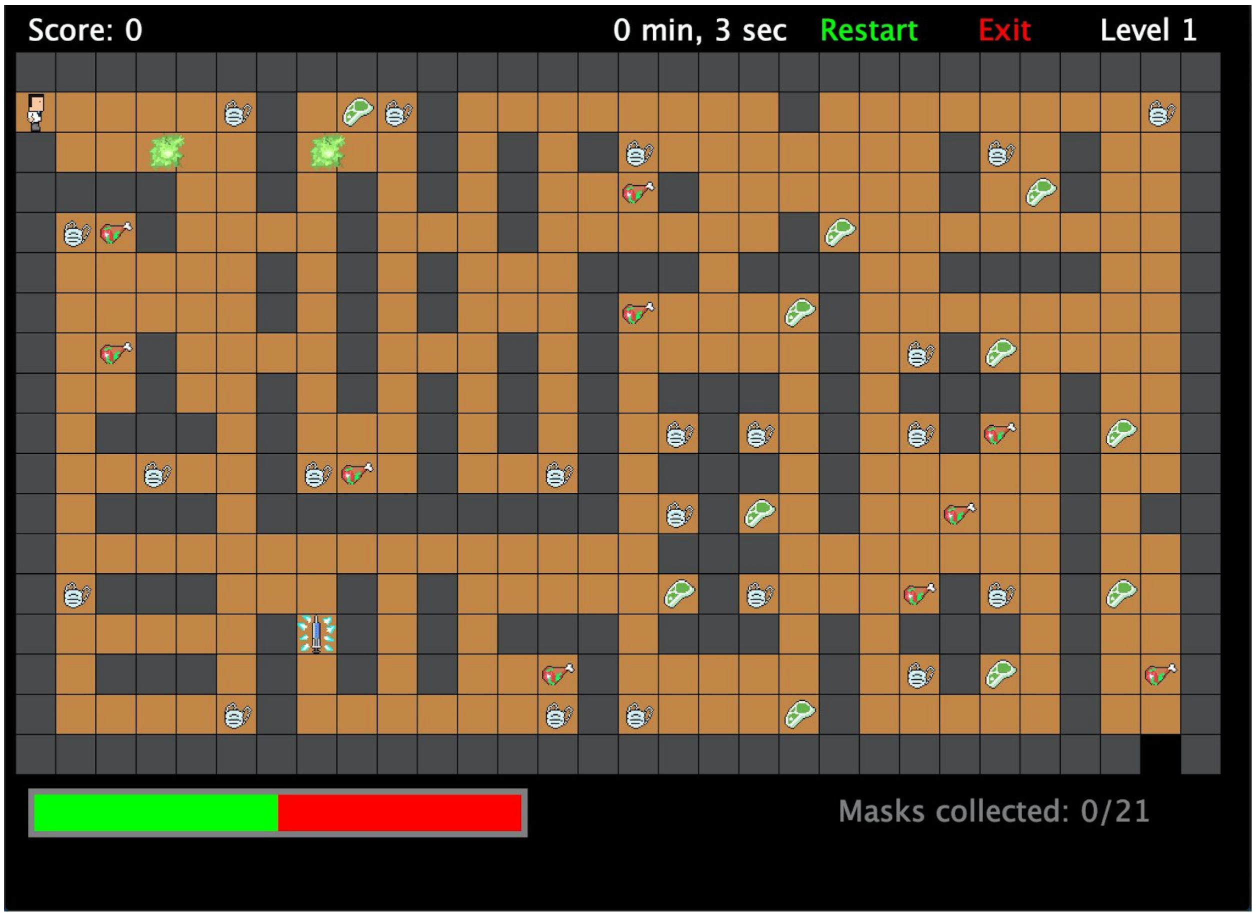


The user will then see this screen. Their goal will be to avoid the 3 ghosts (the green objects) and to consume all the masks on the screen.

To navigate the main character on the screen, use 4 keys:

W (go up), S (go down), A (go left), D (go right)

The main character's health (noted by the green bar) will decrease if they collide with the ghosts. Once it reaches 0, the game will restart on the current level.
After the main character consumed all the masks, they will need to arrive at the end cell (noted by the red circle) to move to the next level.
The total number of levels is three. If the main character can successfully navigate all three levels, they will see a success message.
To restart the game, click the "Restart" button on the top.
To exit the game, click the "Exit" button on the top.



This is the screen the user should see once a ghost collides with the main character. The ghost will disappear, and the health bar will decrease (the red bar denotes the decreased portion). Colliding with a ghost reduces the character's health by half of the maximum health. The only way to restore health is by collecting the bonus reward which restores half of the character's maximum health. If the character's health is reduced to 0, the game will restart on the current level.