Critter Elections – An online team building game using smart devices as controllers

by

Krasimir Krasimirov Balchev

In partial fulfilment of the
requirements for the degree of
Bachelor of Science
in
Software Engineering

Department of
Computer and Information Sciences

March, 2023

Abstract

Various forms of team building are used throughout companies across the world. Their primary goal is to improve teamwork. However, in settings like universities, where students are frequently required to form teams with other students for short periods, team building is often ignored. This can cause the group members to feel distant to one another, which can reduce the benefits of the experience of working in a team.

The purpose of this project is to develop a short team based game, which can be used as a team-building tool, particularly for small newly formed teams. The result was a fully playable multiplayer game, which required complicated state management across multiple software applications for its creation. This report includes detailed information about the design, implementation and testing of said game.

Acknowledgements

My sincere thanks to my supervisor for the project, Dr. Fredrik Nordvall Forsberg, for agreeing to supervise my project suggestion in the first place and giving me helpful advice and feedback until the very end. I would also like to thank Dr. Feng Dong for his role as a second marker.

It would be a crime not to mention the effort my dear friend, Divna Calladine, put into the art assets of the final product. She made every single sprite in the game, all the while juggling her other duties. I wish to thank her husband, Jaedan Calladine, as well, for creating the music tracks for the game, which livened things up significantly. The sound effects used in the game are a part of a sound effect collection I bought and are by Damien Combe.

Last but not least, I would like to thank the people who participated in the evaluation and the surveys of the project. Their feedback was invaluable to the improvement of the final product.

Contents

1 Introduction

The idea for this project came from personal experience when working in teams in which the members did not know each other initially. Oftentimes I would find that my teammates would barely interact with one another. Even if everyone was diligent enough to do their assigned work, it seldom felt like I was in a team. Most of the time everyone worked on their own and we only interacted as a team when assigning what everyone's part is and at the end when we had to cobble everyone's work into something cohesive, which was rarely the case.

I thought that some sort of mini version of team building would act as a nice icebreaker. For example, during my yearlong industrial placement, I was able to get to know my team better thanks to us occasionally playing board games after work hours. I wanted to try a similar approach – to have team members play a game to get them to know each other better.

1.1 Aims & Objectives

The aim of this project is to create an online multiplayer team-based video game that will make it easier to break the ice between members in a newly formed team.

The game should be easy to set up for multiple players. Having separate dedicated controllers for every player is inconvenient and expensive [1]. An inexpensive and easy way for players to join a game, maybe through using technology they already have, should be made. Making the game competitive instead of cooperative should also make it more engaging and replayable. Most importantly, the game should encourage communication between team members within the teams in the game, be it verbal communication or through in-game mechanics.

1.2 Outcome Overview

The result of this project is a fully playable online video game for PC. Players can create a private room through the game. To join a room, players go to a website from which they can select their team members and play the game. The design of the game encourages teamwork and frequent verbal communication amongst players in a team.

Four test groups, each consisting of four people, played the game together. Three of the groups played through Discord and the remaining played in-person. A survey was conducted after each play

session. All of the groups expressed interest in the game. Some of the test groups asked to play again even after the evaluation process was over and almost unanimously everyone expressed their desire to play the game in the future. Furthermore, the main goals of this project were met based on the survey results.

Figure 1: A screenshot from the final version of the game

1.3 Report Structure

Section 2 covers the research done to determine the scope and goals of the project. In Section 3, the problem is clearly defined with specifications based on the work done in the previous section. Following that, Section 4 talks about the system and interface architecture. Section 5 continues from the previous section but goes into detail regarding the implementation of the core components for the game. Section 6 discusses the testing methods used and the challenges in testing games. Section 7 takes a look at the results of the user evaluation that was undergone and possible interpretations of it. Section 8 reflects on the project as a whole and suggests ideas for further development. Finally, Section 9 is the closing remarks.

2 Related Work

This section talks about what influenced the decisions in this project and is divided into three parts. Subsection 2.1 discusses what makes video games a viable tool for improving teamwork. Subsection 2.2 covers games that inspired the final product, both in what to include and try to achieve in the game and in what to avoid. Subsection 2.3 is about the research done regarding the possibilities and limitations of the main technology used in this project, Godot.

2.1 Why Video Games?

Video games have had a bad reputation in the past. Primarily because most of the popular games depicted violent content and were considered to have a negative influence on children [2]. However, in present days, video games have, for the most part, shaken off the past controversies and are even considered to have positive benefits associated with them [3]. Not only that, more and more studies are being done on using games as educational tools in classrooms [4] and as safe training environments for medical students [5]. It is only natural for there to be an interest in how video games might affect teamwork.

It is widely believed that some of the requirements of good teamwork are open communication and trust. According to a study by Greitemeyer and Osswald [6], games, which have prosocial elements in them, are likely to illicit prosocial thoughts in their players. Having more empathy and desire to help others is a definite plus when it comes to working in a team. The game with prosocial elements used in the study is Lemmings, which is a single-player video game in which the player guides a group of lemmings. It can be assumed that a multiplayer team based game, which from it requiring teamwork makes it prosocial, will have similar effects on its players.

As mentioned, trust is an indispensable part of good teamwork. Games can also be used as trust-building tools [7]. It is better to establish some form of trust in a low stakes setting rather than later during the actual work needed to done where the stakes are likely to be higher.

While there are more ways in which games are able to strengthen skills important for teamwork, such as role delegation based on what skills a person has [8], I have decide to mainly focus on trust-building and empathy towards your teammates for this project.

2.2 Similar Product Analysis & Inspirations

While there is an abundance of all kinds of video games, it was particularly interesting to see which ones would have the elements I was looking for and how those elements could be combined to create a game that would fit the goals of this project. It was easier to find and come up with cooperative board games than video games so I decided to start from there.

2.2.1 Board Games

Captain Sonar [9] is a real-time, team-based versus game, in which two teams take control of a submarine and try to locate and sink the opposing team's submarine. Each member in a team has a unique task only they can do. For example, one player acts as the captain and gives orders to move the ship. Another plays as the engineer and keeps track of issues in the submarine and tries to fix them. Everyone plays an important role and is a vital part to the team's overall success. Because of this frequent communication is required between the members in a team.

Figure 2: A group of people playing Captain Sonar (picture taken from a review of the game [10])

After looking at various types of team-based games, the approach Captain Sonar takes seemed like a good fit for the project for multiple reasons. For one, in another popular board game called Pandemic [11], because of its slower turn-based nature and all the information being public, experienced players can have the tendency to take control of their teammates' actions. As an article by Mathew Kumar suggests [12], a possible way of dealing with that is to have the game be in real-time or have secret information between team members. Captain Sonar is played in real time and while all the information within a team is public, every player is too preoccupied with their own tasks which makes them unable to keep track of all the information, which in turn is exactly why communication

between team members is so vital. When coming up with the basis for the game mechanics for this project, Captain Sonar and other team-based games that are either real-time or have unique team roles with secret information, like Ladies & Gentlemen [13], were the primary inspirations.

2.2.2 The Jackbox Party Pack Series

There are multiple Jackbox Party Pack games [14] and each one is made up of a collection of party games. Players run the game on the system they bought it for and choose one of the games in the pack. From there, a room is created for that game. Players can join it and play using their smartphone or any device that can run a modern browser (Figure 3). The player who joins the room first is considered the room VIP and can control when the game starts, tutorial skipping and replays.

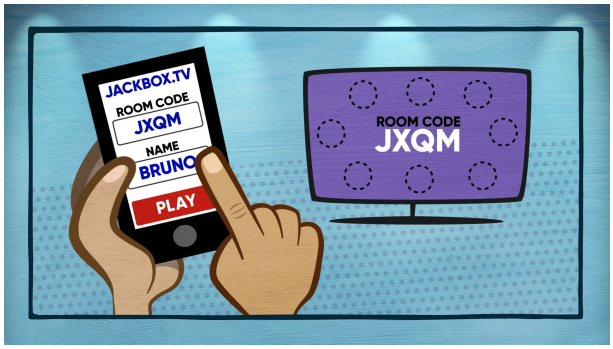

Figure 3: A slide from the how to play section of the Jackbox website showing how players can join a room

Most of the games require players to provide some input on their connected device. Once all players have completed the given task, attention is shifted to the screen of the system the game is running on where the results of everyone's inputs are displayed.

The Jackbox games have a very accessible method of allowing a huge number of players play together through a website that is used to connect players to a game and take their input. This is a great solution to the accessibility issue mentioned in 1.1. Instead of using dedicated controllers, players can use any smart device they have to play the game. The room system the game uses simplifies the process of joining a game by generating short room codes instead of requesting players to make up their own. Because of the ease of use of this system and it already being familiar to some people, a similar approach was decided for the lobby system of this project.

While the party pack series did provide a lot of useful techniques to consider during development, it fell short when it came to interesting team interaction. That is understandable, the main goal of

these collections of games is to provide a light-hearted environment for laughs which has its own benefits. Unfortunately, that does not align with the goals for the project.

2.2.3 Team-Based Video Games

There are many team-based video games, such as some of the most popular esports games, like League of Legends and CS:GO [15]. The problem with them is that they are heavily skill-based, not everyone can jump in quickly and play. While they do provide many great teaching elements of good teamwork, a lot of training is required to not feel like you are dragging your team down, which can have the opposite of the desired effects. The goal is not to have players get good at a game, but to give them a framework where they can have meaningful interactions of effective teamwork with other players and for that to happen everyone should at least feel that they can win. As Richard Garfield discusses in his talk [16], introducing randomness in a game can do exactly that. Of course, there needs to be a balance. If the game is all luck, the players have no agency, which leads to frustration.

Spaceteam [17] is a critically acclaimed cooperative game for phones and tablets (Figure 4). Each player has a control panel in front of them, which they have to manage according to the given prompts. The twist is that your teammates get the prompts and have to verbally communicate to you what you are supposed to do on your control panel. This often turns into humorous situations as players start shouting random technobabble at each other.

While undoubtedly fun, quick to get into and having plenty of randomness, Spaceteam was designed from the start as a local co-op game. While there are ways to play it remotely now [18], it loses some of its charm. With that in mind, one of the goals for this project will be to retain as much of the enjoyment of the game even when playing remotely through a service like Discord.

Figure 4: A screenshot from the Spaceteam trailer [19]

2.3 Research on Godot

Godot [20] is an open-source game-engine that supports deployment to multiple platforms. From the beginning, I knew Godot would be the main technology I used for this project. I have been developing small games with it for more than a year for game jams and was excited to use it for something bigger. What initially attracted me to Godot was mainly that it is open-source, unlike most of the other popular game engines. Godot already comes with hundreds of prebuilt features but if necessary, there is always the option to build my own custom version of it.

There were a few instances when I thought that Godot might not be fitted for this project after all. The plan was to use Godot to make the desktop game as well as the mobile app that would be used as a controller. However, after it was decided to have the controller be a website, I thought I might need to develop the player controller using a different technology. The reason for that is despite Godot projects being deployable to web, it is not expected for them to be used as whole websites. Using GitHub Pages [21] I tested how an exported Godot web project would run as an entire website. While the export size was significantly bigger than a typical website, luckily it loaded quite fast and ran on all the popular browsers. I could even reduce the size of the export by creating my own custom export template, thanks to the provided resources in Godot's documentation.

From the little research I had done for the project proposal, I discovered that Godot has support for various types of network communication. After the proposal was approved, I had to finally actually go into more detail on how to do this. Whether to use a peer-to-peer approach or a client-server approach, would the connection be through TCP or UDP, would Godot even give me the ability to choose?

My initial decision was to use WebSocket as the communication protocol, because of its reliability from using a TCP connection and even more importantly, it allows for bidirectional communication, unlike HTTP. However, I almost reconsidered after reading an article suggested in Godot's documentation [22] that UDP is the recommended communication protocol for games due to its speed. Thankfully, this is only true for games that have a lot of action and require frequent updates to the state of the game. The initial idea for the game for this project was for it to be turn-based, but even after changing it to a real-time game it is still not going to require as many updates to the game state as a typical action game.

3 Problem Description & Specification

This section defines the problem to be solved by the project and explains the methods used to create the requirements for the product as well as the requirements themselves.

3.1 Problem Description

From what was discussed in Section 2, it can be seen that there is no shortage of multiplayer games out on the market. However, what was hard to find was short easily accessible team based games that have a strong emphasis on meaningful team interactions. What is meant by meaningful is interactions that are more than reactionary, interactions based on strategic decisions and task delegation. Spaceteam is a great example of a short accessible team game, which is primarily reactionary. Most popular esports games are the opposite. They require a larger time sink and have quite a bit of depth.

This project aims to create a game that will fit that gap using a similarly accessible system to the Jackbox Party Pack games - a shared screen with public information combined with the ease of players being able to join and playing using any smart device of their choice. The combination of players needing to keep track of two separate screens, one with public and one with private information introduces interesting possibilities for meaningful interactions between players.

3.2 Methodology

Any feasibly big software project is hard to scope and write specifications. This is likely even more true when it comes to game projects. You might have done everything according to the project plan, everything is working and there are no bugs. However, something is not right. You start the game for a test run and you get the sudden realisation, it is not fun! It was not as you imagined it in your head. While similar things can happen in typical software development in the form of some system not being as user friendly, usually, once you build a system to do something, it is mostly done. With games you might realise you do not even need a certain system only after you build it and test it out, because often times what we think is fun in our head may not be in practice.

For this reason, prototyping and frequent iteration is key in game development. The catchall method for short iteration cycles is the agile method. Agile and its many forms are not only heavily recommended by most modern software companies, but by most game developers as well [23][24].

With that said, the methodology for the development of this project will be an iterative agile approach. No need to go into minutia and pick out a specific type of agile methodology. The important thing to follow is to get a working prototype and iterate on it.

3.3 Analysis

It is difficult to determine precise requirements for a game. It is not the same as a client asking for an application that has a specific use case scenario. At the start of the project, the most certain requirements were the non-functional ones as well as the functional requirements regarding the similarity to the lobby system of the Jackbox Party Pack games. As the idea and rules for the actual game started to solidify, more and more functional requirements were coming up. Of course, some need to be abandoned or changed, which is a normal process. Requirements change with time and new information [25]. The requirements listed below are the final requirements used.

3.4 Requirements

The requirements section is divided into three parts. The first one is functional requirements, it covers how all of the systems surrounding the game should work. The next section, the non-functional requirements, covers what the product is expected to be able to do as a whole. The last section is user stories, which describes concrete in-game functionality from the perspective of players in specific roles. I found it more useful to keep those requirements only as user stories.

3.4.1 Functional Requirements

This project requires the development of several applications and has different roles for the players of the game. Refer to Table 1 for the terminology that will be used in this report.

Table 1: Definitions of the terminology that is used

Server	The game server.
Main Game	The PC desktop game, on which the shared state of the game is shown. It is called main game, because it is where the bulk of the game interactions occur. Used interchangeably with desktop game.
Controller	The website users use to connect to Main Game. It is called a controller because players use it to control their actions in the game.
Player	A user of the product.
Room Master	The player who joined a room first with their Controller.

For more information about the interactions between **Server**, **Main Game**, and **Controller** refer to Subsection 4.1. The following functional requirements are prioritised using the MoSCoW method [26]. **Must Have** requirements are accompanied by a motivation to illustrate why they are necessary.

Must Have

- Player can create a room from Main Game.
  Motivation: Players need to create an environment where they can play.
- Server creates room with a generated room key upon request.
  Motivation: A password is needed so only the intended players are able to participate.
- Main Game displays a waiting room with the room key and joined players.
  Motivation: Players need to be informed on what is happening in the environment of the game in order to know how to act.
- Player can join a room after inputting a valid room key from Controller.
  Motivation: Players need to be able to join the environment they created.
- Player can select an available role in a team in the waiting room.
  Motivation: Players need to have the agency to choose their team and role in the game.
- Room Master can start the game if every player is in team, if every team that has a player in it is full and if there are at least two full teams.
  Motivation: Players need a way to start the game.
- Main Game shows the game map on game start.
  Motivation: Players need something on the screen to interact with.
- Main Game updates player information on Controller when required.
  Motivation: For the game to work the information between the Main Game and Controller needs to be synchronized.
- Server updates players and room state when required.
  Motivation: For the game to work the server needs to have information about the state of it to know how to act.
- Player can use Controller to perform actions on the map shown on Main Game.
  Motivation: Players need a way to interact with the game.

- If a player disconnects, Main Game and Server continue as if that player never gives any input.
  Motivation: There needs to be a way to handle disconnections because they inevitably happen.
- Player can reconnect to the room they were in, if they input the room key again and select their character.
  Motivation: Getting disconnected and locked off from a game would be a terrible experience for a player. There needs to be a way they can reconnect.
- When a round ends, Main Game displays the current placement of all of the teams.
  Motivation: Players need to know how they are doing in the game to know what to do and to be motivated to play in the first place.
- During the time Main Game displays the team placements, every Controller connected to that room shows an intermediary screen and locks their input.
  Motivation: To avoid unintended consequences.
- When Main Game finishes displaying the team placements, the Room Master can continue to the next round from Controller.
  Motivation: Players need a way to unlock their input after it being locked.
- When game ends, Main Game displays the winning team.
  Motivation: Players need to be able to know who won in order to care about the game and try their best when playing.
- When game ends, Room Master can exit game or restart the game.
  Motivation: The game needs to give options on what to do after it finishes or the players may think it is broken.

Should Have

- Player can deselect the role and team they have chosen in the waiting room.
- Player can go back to previous steps of the join room process before the game starts.
- Player can choose their nickname from a pool of words after they join a room.
- Player can view the rules of the game from the first screen of Controller.
- Player can change the audio volume from Main Game.

- Player can pause an active game from Main Game.
- If Main Game is paused, connected Controllers display a pause screen and do not allow input.
- Player can press their character on Controller to locate themselves on the map in Main Game.
- Controller can toggle desktop mode.
- If a Controller's desktop mode is on, the controller will display buttons for movement and allow keyboard movement.
- When game ends, Controllers of the players who are not Room Master show an end screen.

Could Have

- In waiting room, Room Master can fill empty team slots with AI players.
- Can select to play alone when selecting team and role in Controller.
- When creating a room from Main Game, allow to set number of rounds and other settings.
- Room Master can skip team placements by pressing skip on Controller.

Won't Have

- Playable version of the full game using only Controllers without a Main Game.
- Playable version of the game using only Main Game with no Controllers.

3.4.2 Non-Functional Requirements

- Controller can run on all popular modern browsers (Chrome, Edge, Firefox, Safari, Samsung Internet, and Opera) [27].
- Players can play remotely using services like Discord and have a similar experience to when playing in-person.
- Players in the same team have meaningful interactions.

3.4.3 User Stories

The following user stories are game rule and mechanic specific. They might be easier to understand with knowledge about how the game works. Appendix A.2 provides the rules shown to players in a

compact format. For a more detailed look at the rules, refer to Appendix A.1. The roles for the user stories will be the team roles in the game (Candidate and Bee Manager). A player will be considered as a user playing the game irrelevant of their team role.

- As a Player, I want to be able to move around the map so that I can interact with things on it.
- As a Player, I want to be able to find where I am on the map so that I do not get lost.
- As a Player, I want to be able to pick up resources on the ground so that I can increase the resources I have.
- As a Candidate, I want to be able to talk to critters on the map so that I can make deals.
- As a Candidate, I want to be able to make deals with critters when talking to them so that I can earn votes to win the game.
- As a Candidate, I want to be able to decline deals I do not want with critters so that a new critter with a new deal may appear.
- As a Candidate, I want to be able to outbid other Candidates' deals so that I can gain an advantage in the game.
- As a Candidate, I want to be able to give money to my team's Bee Manager so that they can use it get more resources faster.
- As a Bee Manager, I want to be able to enter buildings on the map so that I can use my bees.
- As a Bee Manager, I want to be able to place bees in buildings so that I can get resources from their work.
- As a Bee Manager, I want to be able to upgrade buildings so that they can give more resources and give exclusive bonuses to me.
- As a Bee Manager, I want to be able to enter beehives so that I can hire more worker bees.
- As a Bee Manager, I want to be able to hire worker bees so that I can earn more resources from my bees.
- As a Bee Manager, I want to be able to give resources to my team's Candidate so that they can use them to earn votes for my team.

4 Design

This section is divided into three parts. The system design subsection details how the communication between the three parts of this project work together. The next subsection covers the interface design. Following that is an overview of the game design decisions made during development.

4.1 System Design

When deciding between peer-to-peer and client-server, client-server was chosen because of it being overall safer to have everything go through a dedicated server instead of an unknown host. Not only that, but if the host player's connection is poor then the connection would be bad for everyone. Because of this decision, three separate projects were required: one for the server, one for the desktop version of the game (**Main Game**), and one for the website (**Controller**), which is used as a controller. Main Game and Controller are both clients in the client-server communication, just of different types. The server is able to maintain a set number of game rooms and each game room has a finite number of players in it (maximum of eight) as illustrated in Figure 5. A player is used to represent a Controller since every player is expected to join a room in Main Game from a Controller.

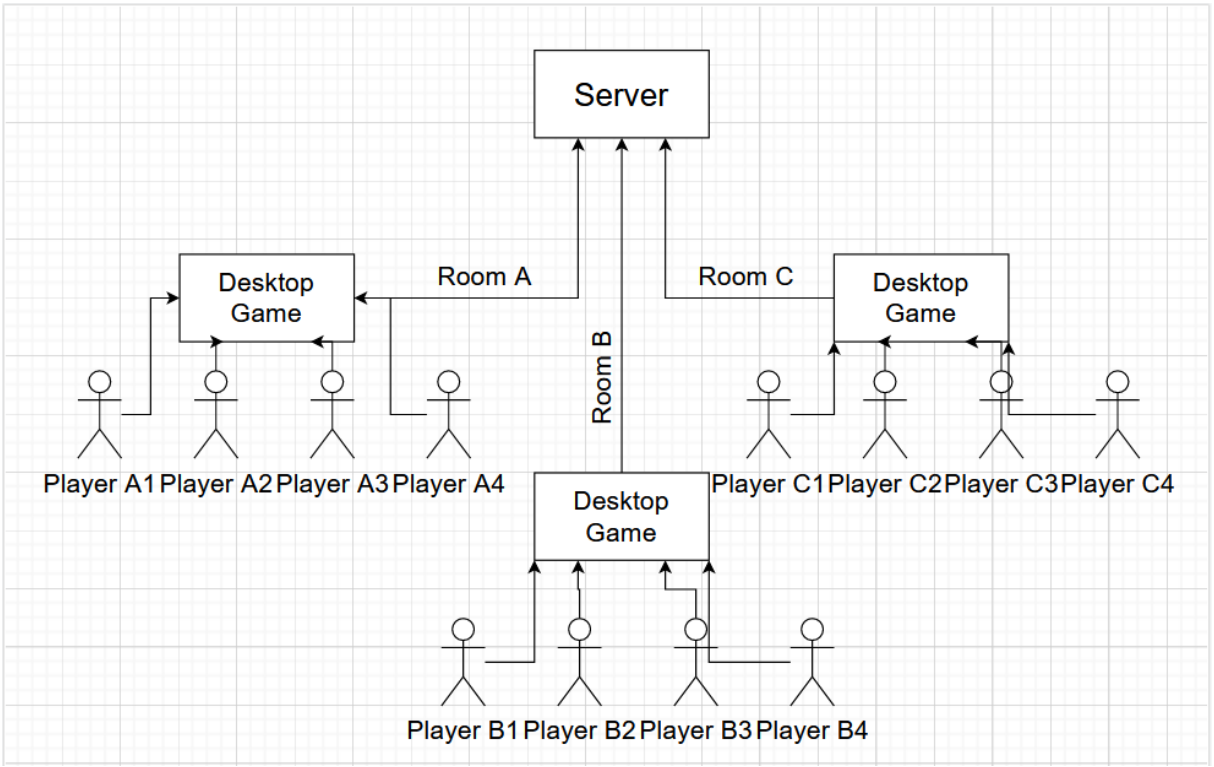

Figure 5: A diagram of the connections between the three projects

Controllers mainly communicate directly to Main Game. The only time a controller receives or sends information directly to the server is when they are in the middle of the process of joining a room or there are communication issues. Main Game communicates with controllers and the server frequently. Controllers send information about their desired action to Main Game, Main Game applies that action to the game state if it is valid and returns data to the controller about the new state of the game. The server stores data about all of the active rooms and the players inside them, so that if a player gets disconnected from a room, they can re-join it and all of their progress up to that point will be saved. For this to be done, Main Game updates the server on the state of a player when needed. Once a room is closed, be it by a player or from a disconnection, all of the room data along with the data about the players in the room is deleted from the server. Because of this, there is no need for a database.

4.2 Game Design

From the research done and takeaways from it in Section 2, there were already enough limitations on what the game could be for a more concrete idea to form. The game had to be team based, but not cooperative as to avoid one skilled player taking control. Due to it becoming a team versus game it would be competitive, but would need to include the right amount of randomness so that anyone could feel they could win. It would not be turn-based but it also could not have a lot of action all at once to avoid latency issues. Last but not least, it should have meaningful interactions rather than reactionary ones. While it is hard to describe how exactly where the exact idea for the game come from, these limitations served as the pillars for it.

The initial concept was that one player played as the candidate for an election in a village and to earn the support of the villagers, they would need to accomplish tasks they were given by those villagers. To make it more strategic, the village would have different areas and for a candidate to get elected, they would need to be the top candidate in the most areas. Those tasks would be of the form *"Get me X amount of resources"*. That player would also have a teammate who would act as a resources gatherer. Both would depend on each other: the candidate needs the resources from the gatherer to earn the trust of the villagers and get elected and the gatherer needs the candidate to win for them to win the game as well. There would not be only one candidate and one gatherer, there would be multiple competing against each other.

The concept so far accomplishes the goal of being a team based cooperative game where players in a team have different roles so it will be harder for a single player to take over the whole game or even just their team. The randomness would come from randomly generated events for the characters would need to accomplish and random elements thrown in the resource collection part. Setting it on a grid with tile-based movement would solve multiple issues. It is not turn-based and also avoids free-form movement, which would require more frequent player state updates. Grid-based movement is also easy to grasp and intuitive since a player only has at most four directions they can move in at any given time. Players in a team would need to assess the current state of the game and decide what the best options are at that time for them. For example, what resources they need to focus on based on their tasks or which areas to focus on completing tasks in.

The specific mechanics for each player role were chosen to fit the idea behind what that role did, but also based on personal preferences. For the gatherer, the worker placement mechanic [28] fit nicely. Worker placement games, typically have players in control of workers. Those workers are placed on designated spots and each spot produces a certain type of resource for the player who put a worker on it. However, these spots are limited and players have to make tough decisions as to where to place their workers. For the candidate, simply giving a set amount of resources to get something seemed uninteresting. Instead, having them place an offer for a reward and then other candidates being able to outbid them sounded a lot more engaging.

This was enough of a solid base to start prototyping and trying things out.

4.3 Interface Design

Rough sketches were used for the wireframes. As is normal for wireframes, some changes needed to be made during the actual implementation. However, for the most part, the designs have most of the components and very close layout to the final product.

The desktop game and controllers required multiple separate screens, especially the controllers. For the Main Game, only three screens were deemed necessary.

Because the controllers are websites, they needed to support various screen sizes and resolutions. However, while this was taken into account, the main intention for the controller website is to be accessed through a smartphone. That is the most likely device players will have on hand. Therefore, a smartphone targeted design was used that would also work on desktop.

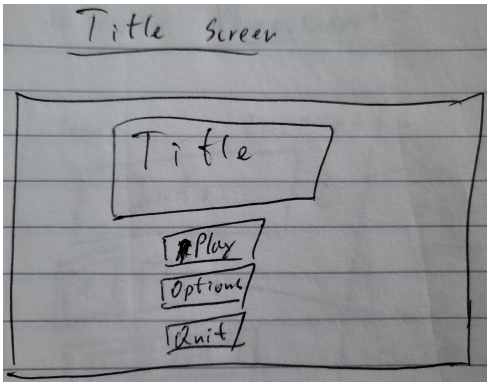

Figure 6: Main Game title screen

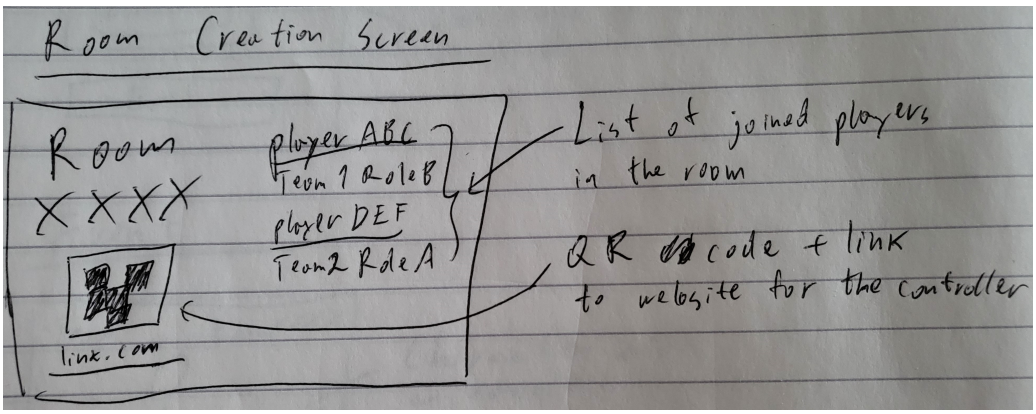

Figure 7: The room creation screen in Main Game

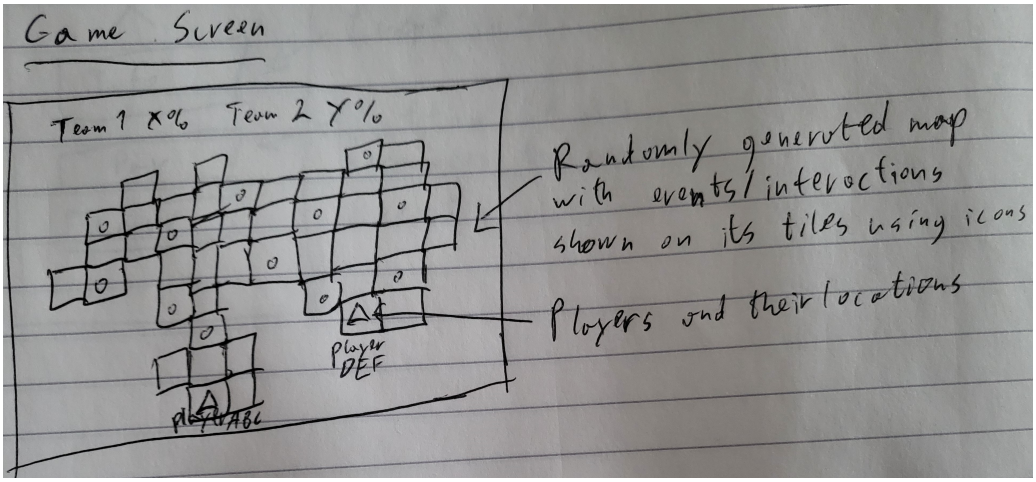

Figure 8: The in-game map screen in Main Game

The screens for the controller can be divided into two types: screens used for joining a room and screens used for playing the game.

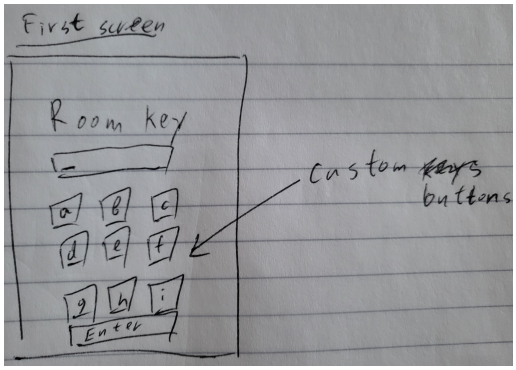

Figure 9: Enter room key screen

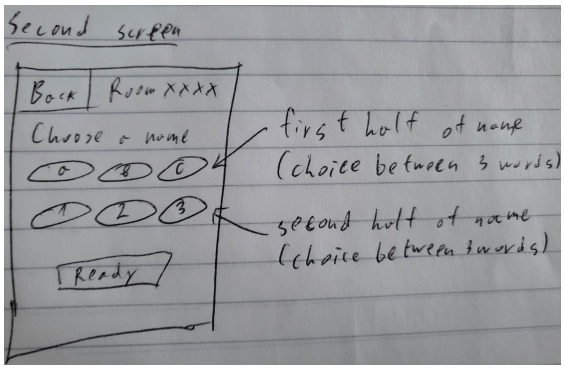

Figure 10: Name selection screen

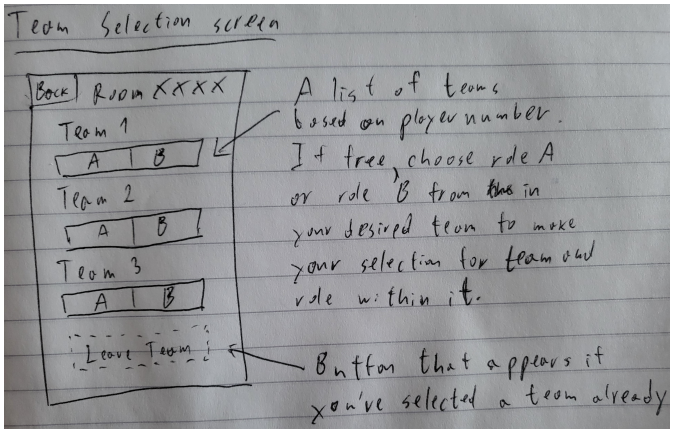

Figure 11: Team and role selection screen

For the screens used during play, there needs to be one for moving the player's character on the map displayed on the shared screen. The other screens are specific for certain interactions in the game and vary between roles.

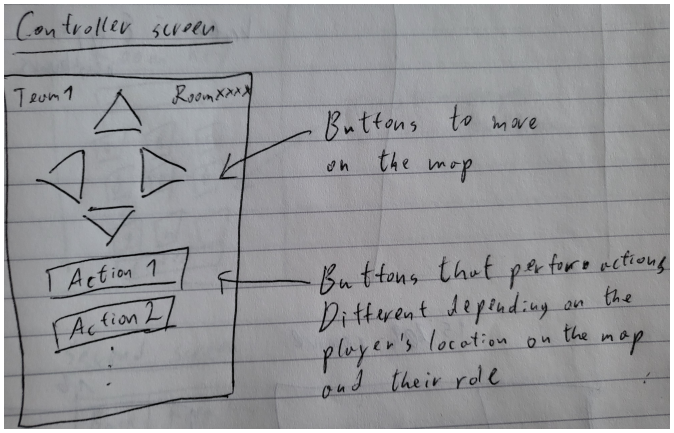

Figure 12: Character movement screen

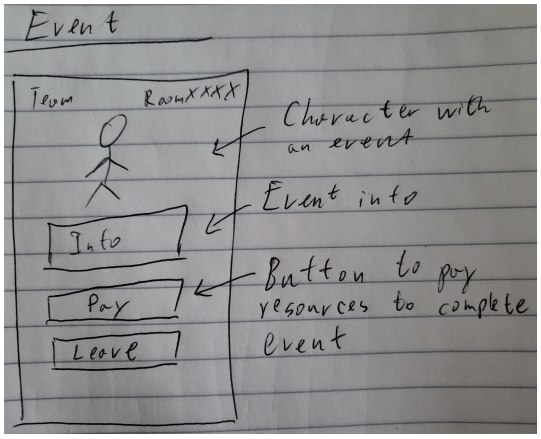

Figure 13: Event screen, used for Candidate role

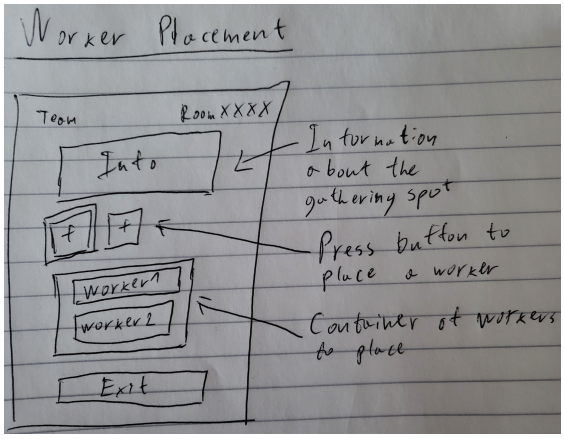

Figure 14: Worker placement screen, used for Bee Manager role

At this stage of development, the game did not have a theme yet, so the wording used in the sketches is vague. The Candidate and Bee Manager (called gatherer above) roles only got their names close to the end of development after considering what names would best describe what a player does when playing as one of the roles.

5 Implementation & Detailed Design

This section covers the languages and tools used for the development of this project in Subsection 5.1. Details about the implementation of the **Server**, **Main Game**, and **Controller** projects are laid out in Subsections 5.2, 5.3, and 5.4 respectively. The last subsections discuss the deployment of the final product (5.5) and how security was handled (5.6).

5.1 Technologies & Tools

5.1.1 Godot

As mentioned in Section 2.3, Godot was primarily chosen as the engine because of familiarity. Of course, if it did not provide the necessary equipment for the task it would not have been chosen but as it turned out, Godot had close to every functionality that would be required for this project and where it fell short, different approaches could be taken.

To name a few of the benefits of using Godot:

- Simple to use node-based architecture [29];
- Free and open-source [20];
- A thriving community that create tutorials and add-ons [30];
- A high-level online multiplayer API (more on that below);
- Streamlined exporting process for PCs and web platforms [31].

The high-level multiplayer API [32] significantly reduces the amount of work required during the creation of an online game. It provides an abstraction from the low-level workings of the communication protocols. Both UDP and TCP are available for use, but as stated in Subsection 2.3, the WebSocket protocol will be used for its useful bi-directional communication, which uses TCP connections.

5.1.2 GDScript

GDScript is a scripting language custom-built for Godot. What it lacks in performance it makes up in ease of use. GDScript resembles Python and Lua in syntax, which makes it a good language for building prototypes fast. As mentioned, it is not the best performance-wise but for the scope of this project that did not prove to be a problem.

Through GDScript you use one of Godot's primary concepts - signals. Signals are Godot's take on the Observer programming pattern [33]. Objects can emit signals, which any object can connect to and execute a function once the signal is emitted. An example can be seen in Figure 15.

```
346 func _instance_gathering_spot(location: Vector2) -> Node2D:
347     var instance: Node2D = GATHERING_SPOT_SCENE.instance()
348     instance.set_data_on_type(MapData.map[location].area_type)
349     instance.connect(
350         "ready_for_production",
351         self,
352         "_gathering_spot_ready_for_production")
353     instance.connect("worker_is_done", self, "_gathering_spot_worker_is_done")
354     instance.connect("upgraded", self, "_gathering_spot_upgraded")
355     return instance
```

Figure 15: An example of an object connecting to the signals of a node it created

5.1.3 GUT

Godot does not have much support when it comes to testing. However, as mentioned, Godot has a thriving community, which creates add-ons for it and one of those add-ons is GUT (Godot Unit Test) [34]. GUT is a unit testing framework, which allows you to write tests for GDScript code in GDScript. In general, automated testing for video games is difficult (more on that in 6.1), but some much needed base tests can be covered with the help of GUT.

5.1.4 Amazon Web Services

Two servers are needed for this project: the game server and a web server for the controller. Out of the companies providing hosting services, AWS (Amazon Web Services) [35] was deemed to be the most appropriate. Because of its popularity, there were many useful tutorials on how to use the service. Its free tier option is enough to support the running of both servers using EC2 instances [36]. More details in Subsection 5.5 down below.

5.1.5 Git & GitLab

Git was used as the version control system for this project along with GitLab [37]. The decision was made because Git was the primary way we were thought about version control and as students at the University of Strathclyde we were given GitLab accounts for our studies. GitLab's issues were used as a way to track tasks during development in the spirit of the agile methodology (Figure 16).

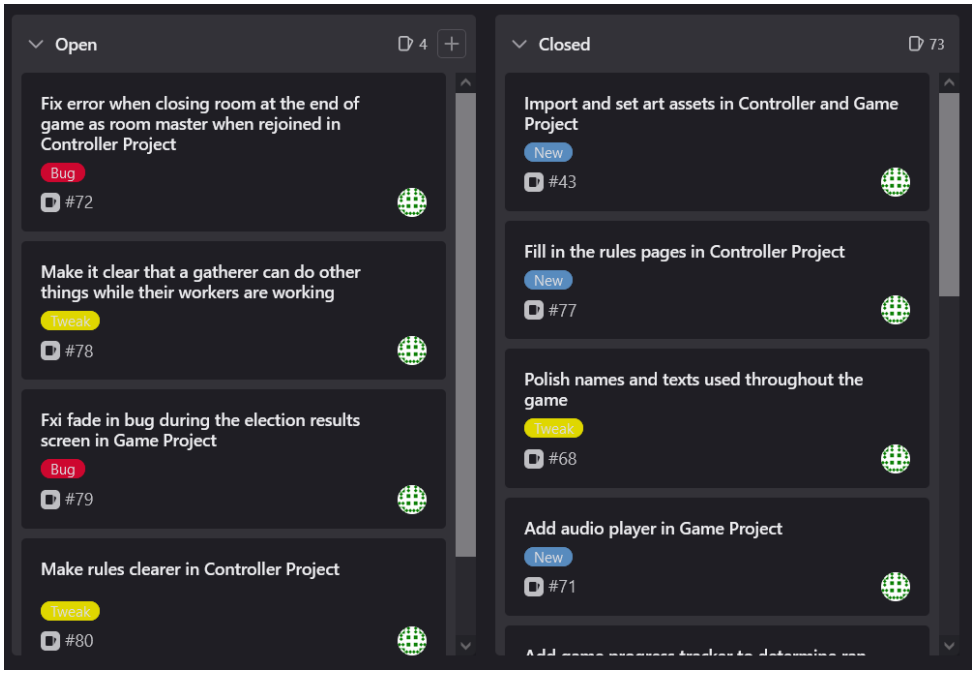

Figure 16: The issue boards in GitLab near end of development

5.2 Server

The game server was the smallest project in terms of functionality out of the three projects required for the game. This is to be expected because its tasks are fairly simple and straightforward. It serves as the gateway for the desktop game to create a room and for players to join that room. Once that is done it only receives updates about the player and room state every so often. It is also in charge of informing the clients in the room if one of the clients disconnected.

When a Main Game instance requests for a room to be created, the server follows this procedure:

1. Check if the maximum number of active rooms is reached. If it is, inform the client it cannot create a new room. If it is not reached, proceed;
2. Check if client has a room associated with them. If they already do, reject them. If they do not, proceed.
3. Generate a unique four-digit room key.
4. Generate data for the room (Figure 17) and store it.
5. Increase the active room count by one.

6. Send the room data to the client who made the request.

```
313 func _get_default_room_data(room_id: int) -> Dictionary:
314     return  {
315         "master": room_id,
316         "players": {},
317         "start_names": LabelData.PLAYER_NAMES.start_names.duplicate(),
318         "end_names": LabelData.PLAYER_NAMES.end_names.duplicate(),
319         "team_names": LabelData.TEAM_NAMES,
320         "team_colors": LabelData.TEAM_COLORS,
321         "allowed_number_of_teams": MIN_TEAMS_REQUIRED,
322         "has_started": false,
323     }
```

Figure 17: The initial room data

The *master* field is the network ID of the Main Game instance that made the request. *players* is a collection of the players in the room. *start_names* and *end_names* are lists of words used when players pick their nicknames in the games. *team_names* and *team_colors* is data about the names and team colors of the possible teams in a game. *allowed_number_of_teams* is the number of full teams required to start a game. *has_started* is a flag that is set when the game starts. It is used to determine how to treat player disconnects. If a player disconnects from a room before the game starts, then their data is deleted. However, if they get disconnected mid-game, their data is still preserved so that if they reconnect they can continue from where they got disconnected. The name and team data is included so as to not have to store it multiple times in Game and Controller Project. Just by updating the server, new names and colors will be used in the game without the need for players to update their games.

Here is the procedure that occurs when a client wants to join a room:

1. Check if the room key the client gave is the room key of an active room. If it is not, deny the request and send information on why it was denied. If the room exists, proceed.
2. Check if the room is mid-game. If it is, deny and inform the client. If it is not, proceed.
3. Check if the room has reached the maximum number of players allowed. If it has, deny and inform the client. If not, proceed.
4. Check if the client is in the room. If they are, deny and inform them. If not, proceed.
5. Generate data for the player (Figure 18) and store it in the *players* field of the room.
6. Update the client who made the request and the room with the data.

```gdscript
func _get_default_player_data(room_key: String, player_id: int) -> Dictionary:
    var number_of_players: int = rooms[room_key].players.size()
    return {
        "id": player_id,
        "name": "Player %d" % (number_of_players + 1),
        "start_names": _get_random_set_from_array(
            rooms[room_key].start_names, NAME_SELECTION_AMOUNT),
        "end_names": _get_random_set_from_array(
            rooms[room_key].end_names, NAME_SELECTION_AMOUNT),
        "is_disconnected": false,
        "team": Enums.Team.NOT_SELECTED,
        "role": Enums.Role.NOT_SELECTED,
        "join_order": number_of_players,
        "location": Vector2(-1, -1),
        "resources": {
            Enums.Resource.MONEY: 0,
            Enums.Resource.ICE: 0,
            Enums.Resource.LEAVES: 0,
            Enums.Resource.CHEESE: 0,
            Enums.Resource.HONEYCOMB: 0,
        },
        "workers": {},
        "is_exchanging": false,
    }
```

Figure 18: The initial player data

The *id* is the player's network ID. *name* is the player's nickname. It will be changed in a later step. *start_names* and *end_names* are subsets of the room's fields with the same names. A player can only create their name from a small number of words. To be certain different players in the same room do not create the same name, every player in the same room is given words that do not appear in any another player's word pool. For more information on why players can only pick names and not input their own names, refer to Subsection 5.4.1. *is_disconnected* is a flag to check if a player is disconnected from an active game. *team* and *role* is data about the player's team. *join_order* tracks whether a player joined first, second, third and so on. If a player is first, they are given the role of room master, which gives them the ability to start the game. If another player disconnects, the join orders need to be updated to reflect the new room master. *location* is the location of the player on the tiled map. *resources* tracks the number of resources a player has. *workers* tracks the workers Bee Managers (gatherers) have at their disposal. Finally, *is_exchanging* is a flag to determine whether a player is trying to exchange their resources with their teammate.

The process for joining a room as a player who got disconnected is a bit different. If a client tries to join a room that is mid-game but has disconnected players, they will be given a list of the disconnected players and asked to identify themselves. After that, they are sent the data that was stored on the server of that player and can continue playing the game.

The other main functions of the server are simple. If a client disconnects or requests to leave, forget them and inform any other clients who might be affected. When a Main Game instance wants to update the state of itself or a player, they send the updated state data to the server and the server overwrites its own data with it.

5.3 Desktop Game

Compared to the server project, the Main Game project has a lot more to it. Unlike the server project, not only did the communication need to be set up but there needed to be visual information displayed at all times. As expected, that makes everything harder from a design standpoint. Different states with different interactions between them would also need to be handled carefully.

5.3.1 State Management

While there are different states, they are not many and proved to be quite manageable by just separating them as scenes in Godot. As Godot is node-based, scenes are trees made from those nodes. So a scene is just root node and all of its children. For more information on Godot with code examples, refer to Appendix B. There are three main screens that would be shown to players from Main Game and each one of them can be its own scene, independent from the rest. Any inner states in the screens can also be handled using scenes that become active and inactive when necessary. In Figure 19 you can see the state diagram of Main Game.

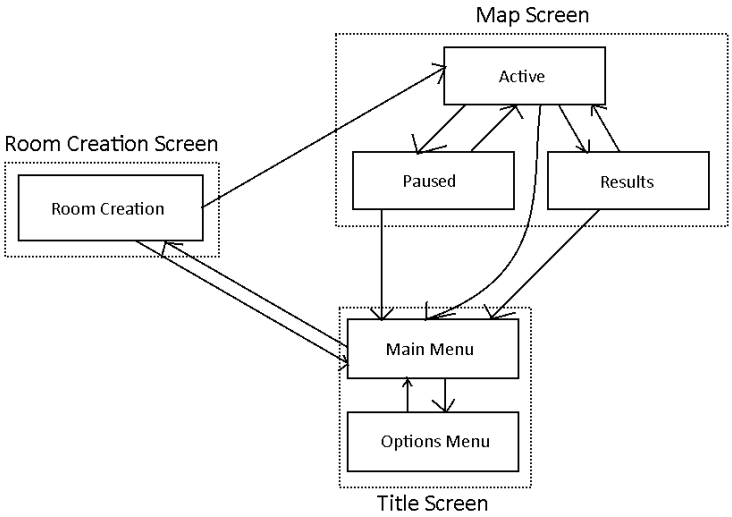

Figure 19: Main Game's state diagram

The states contained within a dotted rectangle make up one of the three screens. When a Godot game is ran, it has a starting scene, which it starts from. It only loads everything needed for that scene. Transitions to new scenes can be made with the *change_scene()* function. When that happens, all of the nodes of the current scene are freed from memory and the root node of the new scene along with all of its descendants are loaded in. Main Game has only three scenes it transitions between: the title screen (starting scene), room creation screen, and map screen. The states inside of them are represented by other nodes which are set invisible and inactive when transitioning between states in the same scene.

Figure 20: What a player sees during the main menu state

Figure 21: What a player sees during the options menu state

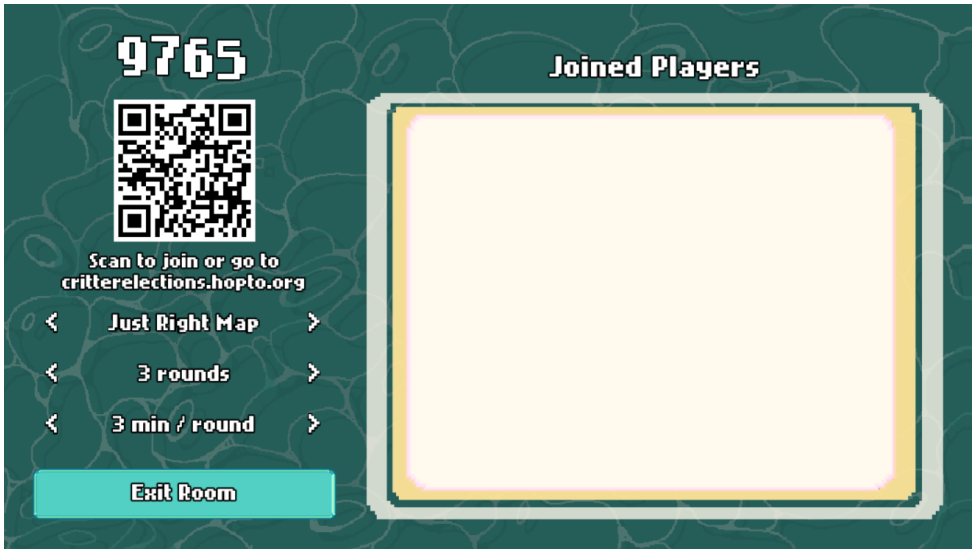

Figure 22: What a player sees during the room creation state. A QR code generator website was used to generate the QR code [38].

The map screen is the most complicated state in the Main Game project and is where the actual game occurs. The following subsections take a closer look at its inner-workings.

5.3.2 Map Generation

To introduce an element of randomness and to avoid players getting bored of the same map, it was decided that the map would be randomly generated. When the map screen scene is created, the first thing it does is to generate a tile-based map based on the size specified in the room creation screen settings as seen in Figure 22. The maps in the game are divided into three areas: Green Area, Snowy Area, and Urban Area. A tile can only be of a single area type and areas should not be divided into multiple subareas, meaning that every tile from one area should be able to reach every other tile in the same area just by only traversing tiles from that area. The first attempt at map generation was to make a rectangle shaped map, create smaller rectangles inside it to represent the areas, and to remove tiles one by one, randomly while following a list of rules that dictated when a tile could be removed and not. For example, if it would split the map into two isolated islands, it should not be removed. This proved difficult to program because of all the rules that needed tracking. Not only that but the initial attempts did not provide interesting map shapes, likely because of the many rules not allowing much variation.

The next map generation algorithm was sufficient enough and is the one that remains in the game now. It starts off empty and it randomly picks out three of the four corners of where the map would be (it was later changed to pick out the tiles diagonal of the corner tiles which resulted in more interesting maps). It assigns a random area type to each corner and that corner tile is used as the first tile of that area. Each corner tile is added to a pool of tiles belonging to one area. Then, a random tile is picked out from each tile area pool and a it randomly selects a spot next to it that is free. If there are no free spots, that tile is removed from the pool. That free spot is set to be a tile of the area type of the tile that selected it and is added to its corresponding pool of tiles.

When generating the map, each area also has a tile limit. Once it reaches that limit, it stops creating more tiles from that area type. This allows for there to be one big area, one medium area, one small area, and for holes to appear in the map.

Once all of the tiles are placed, a BDS (Breadth First Search) algorithm is used to determine whether the map is one connected island or multiple islands. If the map is not a single connected island, the algorithm starts from the beginning until a map that satisfies this constraint is made. This can turn into an endless loop with the wrong map size and area limits. However, since those values are hard coded into the game, the issue is avoided.

Figure 23 is an example of the end result of this algorithm. The labels indicating the possible sequence order and the red squares showing which were the first tiles are only added as an aid and are not present in the game.

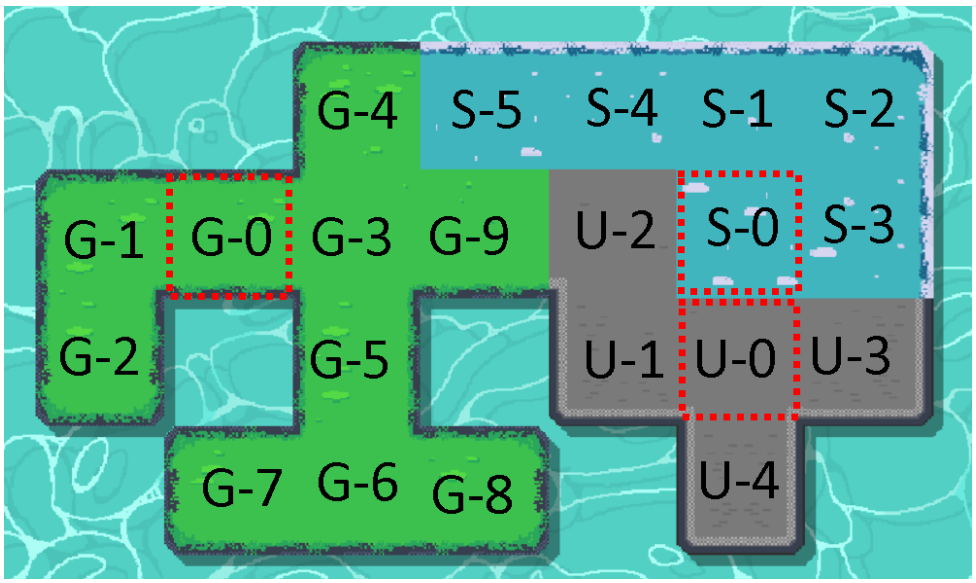

Figure 23: A representation of how the map generation algorithm works. The letter represents the area type. The number represents the cycle in which the tile was placed.

Each tile during the map generation process actually corresponds to a two by two grid of tiles in the final map. In other words, the generated map is enlarged to be four times bigger. This is done for two reasons. The first is to speed up the generation process. The second and main one is to avoid creating paths that are only one tile wide. Two players cannot be on the same tile, which leads to the possibility of a single player being able to block one or possibly multiple players on tile wide path. While this can still happen if two players block a path, it would require the cooperation of a whole team of players in these game's terms to stop what they are doing just to block one other player. It is a very unlikely event and would likely put the blocking team at the disadvantage due to inactivity.

This algorithm results in much more interesting and varied maps from game to game. The tiling itself is handled by Godot's TileMap system.

Once the map is generated, interactable objects, such as critters who have events for candidates and gathering spots for the bee managers, are randomly placed either within the whole map or a specific area. The script for the map screen serves as the factory in the Factory Method pattern [39] when creating the map objects. After that the players are also placed randomly on the map.

5.3.3 Visual Feedback

One of the bigger problems during development was figuring out how to display the information on the map to the players. To make the game more strategic, players should be able to analyse the information shown to them on the map fast. But the map was made up of over a hundred small tiles and multiple of them would need to have quite a bit of information displayed on them so that players know what to do. The answer was to let the theming and art do the hard work. I gave instructions on what sprites I wanted to the artist, Divna Calladine, but her interpretations definitely helped the readability of the game. For example, how the resource gathering spots are made out of the resource you get from them and are shaped as the critter that makes deals with that resource (as can be seen in Figure 25). The leafy bush which gives leaves is in the form of a snail, the cheese factory is made out of cheese and has a rat's face as an entrance. The igloos, which give ice, have an ice block on them with the face of a penguin.

Figure 24: A screenshot of the active game state

Figure 25: The art for the resource gathering spots

There remained some necessary information that needed to be placed, like the number of bees in a gathering spot, and whether someone had made a deal with a critter already and how much time until they leave.

When a player receives a resource or voters, a small text fade animation is played to inform them how

much they received and what. This is easily overlooked when playing, since it is not vital information because players are frequently getting resources and get overwhelmed with all of the numbers. When players need to know how many resources they have, they can look at their controllers, which display that information, which is enough. However, without showing players how much they get for what, the game can start to feel too random. Players ignore the exact numbers they see when they get resources but have a vague idea on how much they get from what, which is what is important to alleviate the feeling of randomness.

The way players win the game is by collecting votes. To keep in spirit with the team, players do not know exactly how many votes they have when playing. Yet, that information is important for them to make decisions. The solution to that problem was to show how many votes each team has in each area and overall in the form of a short presentations. These results are called election predictions and after the last round, the election results are shown in the same manner, which is the final result from the game and a how a winning team is chosen. These short presentations are done in the results state of the Main Game. Figure 26 shows how they look. During the results state, the objects in the map are paused using Godot's scene pause functionality.

Figure 26: A screenshot of the results state of Main Game

One of the requirements for the game was for it to be able to be paused. The pause state resembles the results state in that it is an overlay put on top of the map screen and pauses the objects on the map.

Figure 27: A screenshot of the paused state of Main Game

5.4 Controller

The controller's states are built in a similar way to the Main Game's in the sense that they are screens, which act as the loaded scenes shown to the player. While there are only two screens for the controller, each one of them has a more complicated state machine inside it. The reason only two scenes were used for the controller project was to avoid load times. As stated in Subsection 5.3.1, when Godot transitions of one scene to another it frees the current scenes nodes and has to load the new scene. The controller would be run in a browser, where there would be less resources and frequently loading of scenes could be problematic. Another reason is that even though the controller has multiple states, many of them share similar UI elements. It would be a waste to have to load the same UI elements over and over. It was decided that there would only be two scenes that would be used in the entire project. The first one would be the entry screen, which handles the process of joining a room and starting a game. The second is the controller screen, which handles all the elements that are needed for actually playing the game. As seen in Figure 28, the number of states

far exceeds that of the Main Game's and a lot of them are contained within just one scene. Because of this a new approach had to be undertaken to handle the state management, which is the State pattern [40].

The state pattern was implemented by creating a finite state machine script which would keep track of each state. A state was represented as an abstract class which has three functions: *on_entry, on_exit, and run*. When the finite state machine transitions from one state to another it calls the current state's *on_exit* function and the calls the next state's *on_entry* function. The *run* function is called by the finite state machine on every frame, but for the purposes of this game, the *on_entry* and *on_exit* functions did most of the heavy lifting.

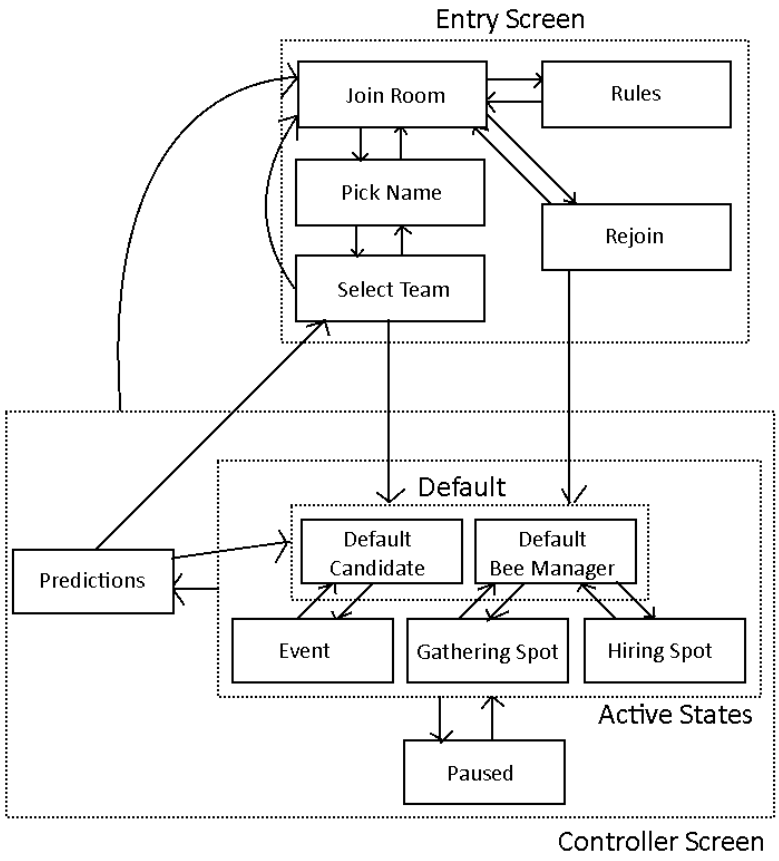

Figure 28: The state diagram of Controller

Each dotted rectangle in the diagram represents a scene (which is just a node and its descendants) with the exception of the default dotted rectangle. Because there are two unique roles a player can be, which act differently in some instances, a Default state for a player was created and the Default Candidate and Default Bee Manager states inherit from it, overriding any necessary parts.

When state transition arrows comes out from a dotted rectangle, that means that transition can happen from any concrete state within that rectangle. The state transitions going from Select Team and Rejoin to Default, in practice, go to the concrete state Default Candidate or Bee Manager depending on the player's role.

To explain each state and its transitions in more detail, Subsections 5.4.1 and 5.4.2, cover the entry screen states and the controller screen states respectively.

5.4.1 Entry Screen

The initial state of the controller is the Join Room state. From it a player can press the *Rules* button to go to the Rules state and read the rules. Once they have read the rules, they can type in the room key shown to them in Main Game's Room Creation screen and start the room joining process.

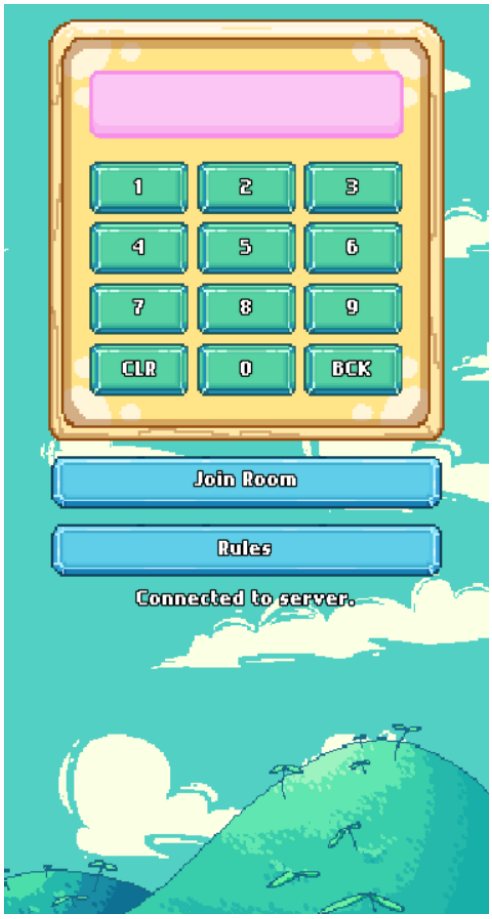

Figure 29: What a player sees during the Join Room state

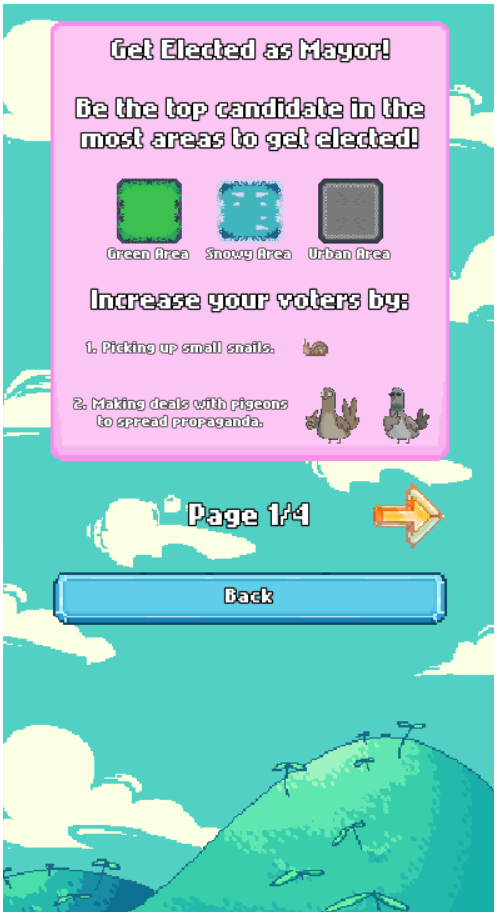

Figure 30: What a player sees during the Rules state

Initially, the join room screen looked a lot more like the one in the Jackbox games one seen in 3. Players could create their own username and type in the room key. However, during the first test on mobile, an issue popped up. Godot's input boxes do not call the phone's keyboard to appear when tapped. There were workarounds, such as add-ons that acted as phone keyboards. Instead, it was decided that the game would not need or allow any direct text based input from the player. Instead of a room key with letters, a number key was used and with it a custom number pad. The number pad does not accept input even if you play from a computer with a keyboard. Instead of typing in their names, players would pick out a name from a list of words, akin to the game Super Auto Pets [41] (Figure 31). This also had the added benefit of avoiding input checking and possible security issues through user input.

Figure 31: What a player sees during the Pick Name state

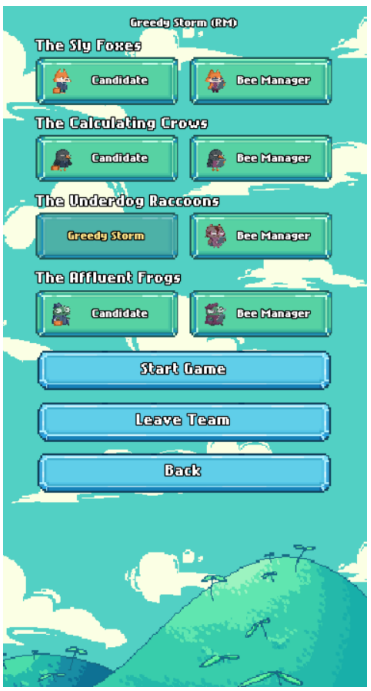

Figure 32: What a player sees during the Select Team state

Figure 33: What a player sees during the Rejoin state

The Select Team state (Figure 32) is the most complicated out of the entry screen states. It requires live information of what team and role every other player has selected. When a player selects a role, that player's controller informs the server and Main Game of that and they in turn inform every other player of the change. If another player in that room is also in the Select Team state, their screen will update by disabling the role in the team the first player chose and showing the nickname of the player who chose it. This avoids players selecting the same role in the same team. Once enough teams have

been filled out the *Start* button on the room master's screen will be enabled, allowing them to start the game (Figure 32 is taken from a test build, in deployment the *Start* button would be disabled if only one player had selected their team and role). The scene transitions to the controller screen and the controller screen's instance of the finite state machine sets its starting state to Default Candidate or Default Bee Manager, based on the role the player chose. The way the player data is preserved between scene transitions, even though all of the nodes from the pre-transition scene should be freed along with their data, is by using a global scene persisting script that holds it. This is the equivalent of the Singleton pattern [42] in Godot.

The Rejoin state (Figure 33) is used when a player is trying to reconnect to a game they were disconnected from. As discussed in 5.2, the server sends the player a list of disconnected players in that room. Those disconnected players are displayed in the Rejoin state as buttons. Once the player chooses one of them, they are taken through the same transition discussed just above with the Select Team state. A player can rejoin from a different device and, of course, someone else can rejoin as that player. This makes the process of swapping out a player mid-game if someone has to leave easy, because they can continue the game as the player who left without needing their device.

Every state, besides the Join Room state, has a back button in its view, which allows the player to go back to the previous state.

5.4.2 Controller Screen

The entry screen states combined only have two entry points to transition to the controller screen. On the opposite end, the controller screen can transition back to the Join Room state from any one of its states. This is allowed because disconnections have to be taken into account. If the player's controller loses connection to either the server or the Main Game which hosts the room they are in, they are kicked out to the Join room state.

The Default state (Figure 34) is the state the controller screen will stay in the most. The differences in it between the roles is what context sensitive buttons it shows when it is interacting with the map objects. Another difference is the resources shown at the top of the screen. The Bee Manager role has a few more stats related to their workers. Otherwise, everything else is the same. The role of the Default state is to allow the player to move around the map and interact with critters, buildings, and fallen items. From those interactions transitions to the other states occur.

Figure 34: What a player sees during the Default state

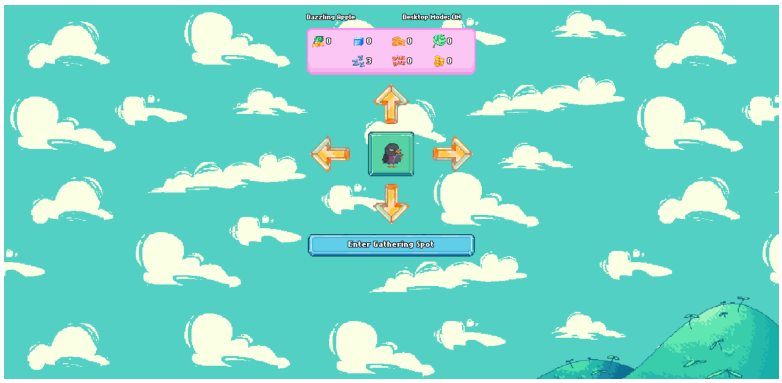

Figure 35: What a player sees during the Default state of the controller screen when playing in Desktop mode

As can be seen in Figure 34 and Figure 35, there are two input methods for movement. Initially, only the directional buttons in Figure 35 were used. From the first test on a smartphone it was evident that they were not a good fit. Unlike physical buttons, touch buttons have no tactility to them. Which might be good enough for some games on mobile but for this particular game, in which players will mostly look at the shared screen of Main Game, not being sure you pressed a button can get frustrating. Most of the times I would tap the correct button and move where I wanted to but every so often my hand would move a bit and I would miss my desired button, causing me to look down at my phone to readjust my finger. It did not feel good to play.

To combat this, a new input method was added, one much more suited and intuitive for touch controls - swiping. The decision to use buttons to move was an automatic one, once thoughtfully considering the main platform for the controller it became clear that swiping would be superior [43]. After swiping was added, I was able to move freely without thought, it felt natural. A small drawback was that swiping is slower than tapping, so moving as a whole was a bit slower. In the end, that sacrifice was worth it. The directional buttons remained in the game for players who used their

controllers on PC in the form of Desktop mode. The Desktop mode option is located next to the player's name and can be turned off and on, no matter the state of the controller screen. While in the Default state in Desktop mode, the arrow keys and WASD are also a supported way of moving. Another difference between Desktop mode on and off is that when it is off, the blue context buttons are moved to be further down. This is another design decision made after considering the fact that it is more comfortable to reach and tap things in the lower half of a smartphone's display. For more on input handling, refer to Appendix B.3.

While in the Default Candidate state, if the player is on the same tile as a critter, a *Discuss Deal* button will appear which if pressed takes them to the Event state (Figure 36). While in Default Bee Manager, if the player is on the same tile as a gathering spot (a leafy bush, a cheese factory, or an igloo), the *Enter* button will appear. If pressed, they transition to the Gathering Spot state (Figure 37). If a Bee Manager is on the same tile as a beehive, an *Enter* button will appear again but this time it will take them to the Hiring Spot state (Figure 38). All of the data shown in these states is sent from Main Game to the controller. It is data associated with the map object that was needed for the interaction and these screens are a way of showing that data to the player.

Figure 36: Event state

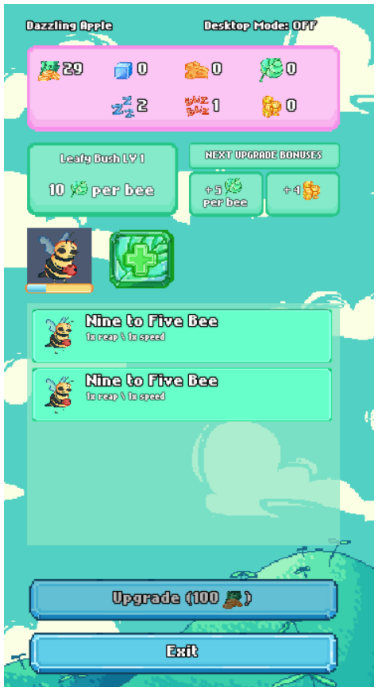

Figure 37: Gathering Spot state

Figure 38: Hiring Spot state

The Event state consists of a sprite of a critter and a message to the player about the deal, which

is randomly generated in the Main Game based on what stage the game is in. Players (specifically Candidates) can make offers, decline offers and outbid other players to get the benefits of the deals.

The Gathering Spot state is for when Bee Managers task their workers with getting resources. When a player places a worker, the Main Game is updated with that information and starts a timer for that worker. Once the timer times out, the worker returns to the player with resources. The progress bar below the bee portrait in Figure 37 is to show how much time is left until the worker and the spot it occupies become free. What is interesting is that you might assume the progress bar receives frequent updates for its progress from the worker timer in Main Game. However, it seemed wasteful to be sending so many messages about data that was not that important. What is done instead is that when the player enters the Gathering Spot state it only receives one update on what every workers' progress is. The controller then starts to run its own timers for each worker to update their progress bars. These timers are only to show the progress, nothing will happen to even if they reach the end. Only Main Game has permission to release the workers once its own timers finish. Using timers in the controller is simulates the actual timers that happen in Main Game close enough that it is not noticeable that they are separate.

The Hiring Spot state uses the same list of worker buttons used in the Gathering Spot state. The difference is that a price label is made visible on the worker button and the *pressed* signal of the button gets connected to a different function.

All of the states of the controller screen discussed so far are a part of the active states. The active states are states that involve play with the game's mechanics. The remaining two states of the controller screen are both used to forbid players from giving input but in different circumstances.

The first is when Main Game enter its Paused state (Figure 39), as it enters it, it sends information to all the players in its room that they have to enter the paused state as well. The finite state machine by default makes sure no state conflicts, meaning that while it is transitioning from one state to another, if it gets notified to transition to another state, it will ignore it. But a state transition can be forced, meaning that no matter what happens, the forced state transition should be the one that occurs. The transition to the Paused state is a forced one. The only way to transition out of the Paused state is either getting notified by Main Game the the game is not paused anymore or by disconnecting. Once the Main Game informs the controllers that the pause is over, all of the controllers transition back to the state before the Paused state.

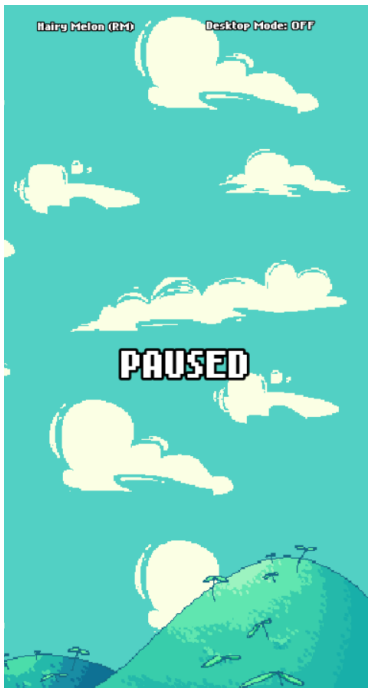

Figure 39: What a player sees during the Paused state

Figure 40: What a room master sees during the Predictions state

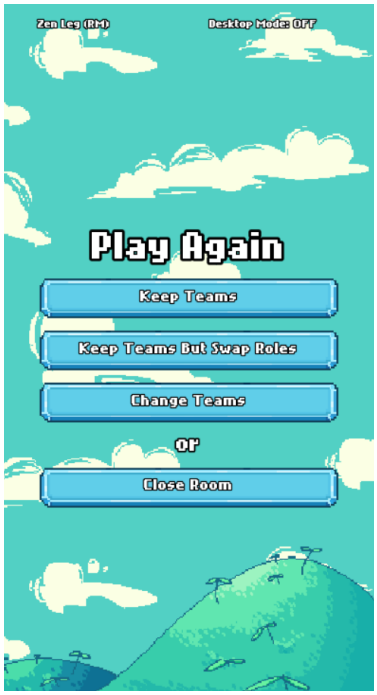

Figure 41: What a room master sees at the end of the game

The Predictions state (Figure 40) is used when Main Game is showing results. It's similar to the Paused state as in it does not allow input and it has a forced transition. One of the differences is that Main Game sends information about the number of rounds when giving the command to controllers to go to the Predictions state. Based on the round, different text options are displayed. When the Results state finishes its presentation, the Main Game sends information to the room master about that and the room master can continue the game for everyone. Instead of continuing from the state before the Predictions state (like the Paused state does), the game continues from the Default state. This is done because a new round has begun and the map state may have changed. To avoid conflicts like players being in a state caused by a map object that doesn't exist anymore, the Default state is preferred.

If this was the last round, the room master will instead have the ability to restart the game or close the room (Figure 41). This is why a pretty weird transition from the Predictions state to entry screen's Select Team state exists. If the room master presses the *Change Teams* button, every player in the room is brought back to the Select Team state to choose new roles and teammates. The other restart options bring back the players to the Default state. The *Close Room* button sends every player back to the entry screen's Join Room state.

5.5 Deployment

As mentioned in 5.1.4, AWS was used to host the game server and web server for the project. Both EC2 instances used run Linux. For the machine for the game server, it was only necessary to install Godot to be able to run a server version of the build. Godot 4 supports running a headless (windowless, no requirements for graphics) build with just a command line argument but for reasons stated in Appendix C, all of the projects were made with Godot 3.5.1 and were not updated to support Godot 4.

An Apache web server was used for the controller project. No-IP [44] was used to get a free DNS. The controller is hosted on: `http://critterelections.hopto.org/`. However, the server is not left to run at all times as to not incur unnecessary costs.

For a detailed explanation on how to run the projects in deployment, refer to Appendix C.

5.6 Security

The approach to security during development was to avoid any need for it. No personal data is collected. Players are not required to enter any personal information or any information at all. The only data the game uses is its own for running the game. Once a room is closed, all of the data associated with that room is erased from the server as well. A database was not required either.

When it comes to cheating, it is possible to create a Project to simulate communication with server as a player or a room and do unintended behaviour but because of there being no valuable data to steal, the only reason for this would be to just cheat when playing. The effort required to mitigate cheating through such means is out of the scope for the project and since cheating in the game does not really pose any real concerns it was ignored during development.

The controller is hosted through http and not https. This is out of Godot and browser restrictions. A secure connection with a self-signed SSL certificate was attempted but the communication between the controller and server could not be established because of that. It should be possible for the connection to work if both the web and game server have SSL certificates approved by a certificate authority. However, that proved difficult to set up and would require extra costs for domain names and the certificate depending on the authority. As stated above, neither the server nor the controller handle any sensitive data, so mishandling of such data is avoided.

6 Verification & Validation

This section covers the various methods of testing employed during the development of this project to ensure that everything worked as expected.

6.1 Automated Testing

As mentioned in 5.1.3, an add-on for Godot called GUT (Godot Unit Test) was used as the testing framework. The add-on had to be installed in all of the projects. The framework resembles most typical testing frameworks, providing assert functions to check the validity of the code's results. It allows for before and after functions to be ran after every test in a script and other similar functionality you would expect. As mentioned in 5.1.2, Godot heavily uses the Observer pattern in the form of signals. GUT allows you to test whether signals have been emitted when you expect them to be. An example of this can be seen further below. Figure 42 is an example of a typical test written using the GUT framework. Figure 43 shows all of the tests for the server passing.

While GUT did help quite a bit by enabling the use of some automated testing for this project there were limitations both on part of GUT and this project, which resulted in the writing of useful tests being quite difficult.

```
140 func test_is_role_in_team_taken() -> void:
141     _set_dummy_room_data(1, 3)
142     var room_key: String = MatchRoomManager.rooms.keys()[0]
143     var player_one_id: int = MatchRoomManager.rooms[room_key].players.keys()[0]
144     MatchRoomManager.rooms[room_key].players[player_one_id].team = \
145         Enums.Team.TEAM_A
146     MatchRoomManager.rooms[room_key].players[player_one_id].role = \
147         Enums.Role.GATHERER
148     var is_team_a_gatherer_taken: bool = \
149         MatchRoomManager._is_role_in_team_taken(room_key,
150             Enums.Team.TEAM_A,
151             Enums.Role.GATHERER)
152     var is_team_a_campaigner_taken: bool = \
153         MatchRoomManager._is_role_in_team_taken(room_key,
154             Enums.Team.TEAM_A,
155             Enums.Role.CAMPAIGNER)
156     var is_not_selected_taken: bool = \
157         MatchRoomManager._is_role_in_team_taken(room_key,
158             Enums.Team.NOT_SELECTED,
159             Enums.Role.NOT_SELECTED)
160     assert_true(is_team_a_gatherer_taken,
161         "Gatherer role in Team A should not be available!")
162     assert_false(is_team_a_campaigner_taken,
163         "Campaigner role in Team A should be available!")
164     assert_false(is_not_selected_taken,
165         "Not selected roles should always be available!")
```

Figure 42: A test for checking the helper function *_is_role_in_team_taken* in *MatchRoomManager*

```
1  Godot version:  3.5.1
2  GUT version:  7.4.1
3
4
5  res://test/unit/test_match_room_manager.gd
6  * test_update_room_players
7  * test_handle_room_creation
8  * test_start_game
9  * test_generate_room_key
10 * test_get_disconnected_players_data
11 * test_get_new_mid_game_join_order
12 * test_get_player_id_by_network_id
13 * test_reset
14 * test_get_player_id_by_player_name
15 * test_get_default_room_data
16 * test_is_role_in_team_taken
17 * test_player_default_data
18 * test_add_player_to_room
19 * test_handle_room_closing
20 * test_handle_rejoin_candidate_disconnect
21 * test_deincrement_join_orders
22 55/55 passed.
23
24
25 res://test/unit/test_server.gd
26 * test_is_network_server
27 1/1 passed.
```

Figure 43: The result of running the server tests

The main elements of all of the server, Main Game and Controller required a connection to be established and held. While the server connection itself was able to be tested by running the server

and then running the tests for Main Game and Controller (Figure 44), almost everything else was hard to test because the functionality in code used what Godot calls remote procedure calls.

```
3  func test_connected_to_server() -> void:
4      Server.attempt_connection()
5      yield(yield_to(Server, "connected_to_server", 30), YIELD)
6      assert_signal_emitted(
7          Server,
8          "connected_to_server",
9          "Could not connect to server!")
```

Figure 44: A test for checking the server connectivity in Controller

These remote procedure calls are how the high-level multiplayer API in Godot works. To use one you use the function *rpc*. You pass in a name of a function and that functions arguments. Godot then calls the given function with those arguments in every instance connected to the server. There is an *rpc_id* remote procedure call (Figure 45), which allows you to only call the passed in function in a client with a specified network ID.

```
38 # Allocates room key for a new room and sends the new room data to the peer
39 # that requested the room creation.
40 master func create_room() -> void:
41     var sender_id: int = get_tree().get_rpc_sender_id()
42     print("SERVER: Create room request received from %d" % sender_id)
43     if active_rooms == MAX_ROOMS:
44         print("SERVER: Room limit reached! Cannot create more rooms!")
45         rpc_id(sender_id, "print_error", "Room limit reached! Cannot create more rooms!")
46         return
47     var room_key: String = _handle_room_creation(sender_id)
48     print("SERVER: Room %s created!" % room_key)
49     rpc_id(sender_id, "set_room_key", room_key)
50     rpc_id(sender_id, "update_room", rooms[room_key])
```

Figure 45: The *create_room* function in the server, which uses *rpc_id* to communicate with Main Game

GUT did not have a way to simulate these remote procedure calls, meaning that I could not test functions that used these calls because they would fail when trying to call a function in non existing peer. This was a big issue, which limited the amount of automated testing that could be done.

In general, game software is infamous for being difficult to write automated tests for [45]. Be it the tight coupling between UI and game mechanics or the randomness involved in a lot of the main functionality of the game, it can be hard to write useful tests.

6.2 Manual Testing

After every implementation of a feature, manual testing was done to assess if it worked as expected. Different browsers were tested as well, to keep in line with the non-functional requirements. Print

statements along with Godot's debugger were used to debug any issues that occurred.

Unfortunately, manual testing did not prove itself to be simple either. The project is a team-based multiplayer game, which just from its description is evident that for proper testing you would need more than one person. Multiple instances of the different projects could be ran at once and I played as multiple characters by switching from one instance to another. Of course, this is no where close to representing how the game would be played in practice but it was the only option.

Another issue with manually testing a game is that most functionality requires multiple steps to even reach the point at which that functionality can be tested. For example, when I added the results overlay in Main Game, the way it works is that it plays at the end of every round. I could hard code the rounds to be really short or to play the results immediately on start but then I would have to fix it back later and if another issue occurred due to an interaction between the results overlay and some other functionality, I would have to go back in the code to hard code specific elements for testing. This can get quite messy quite fast. To combat the need for this, a debugging tool was created (Figure 46).

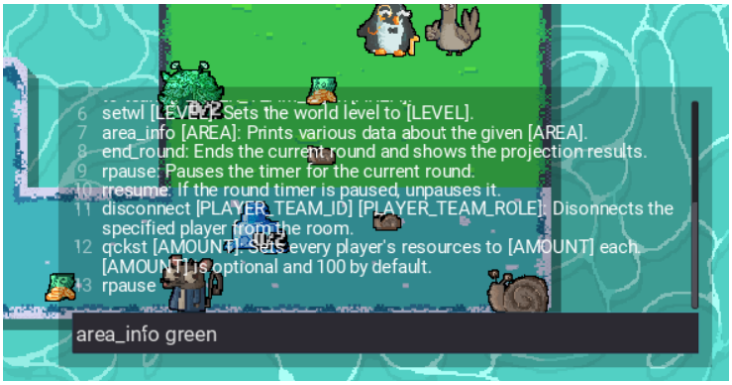

Figure 46: The Debug Helper tool console

The Debug Helper is a console, which can be given commands that affect the game state. The list of commands is as follows:

- **create [OBJ_TYPE] [ROW] [COLUMN]** - Creates a map object of the specified type in the specified location.
- **give [PLAYER_TEAM_ID] [PLAYER_TEAM_ROLE] [AMOUNT] [RESOURCE_TYPE]** - Gives the specified resources to the specified player.

- **support [PLAYER_TEAM_ID] [AREA] [AMOUNT]** - Adds supporters (votes) to the specified team in the specified area.
- **setwl [LEVEL]** - Sets the world level. The world level affects stats.
- **area_info [AREA]** - Prints various data about the given area.
- **end_round** - Ends the current round and plays the predictions.
- **rpause** - Pauses the timer for the current round.
- **rresume** - If the round timer is paused, unpauses it.
- **disconnect [PLAYER_TEAM_ID] [PLAYER_TEAM_ROLE]** - Disconnects the player.
- **qckst [AMOUNT]** - Sets every player's resources to the specified amount.

Thanks to this tool created to help with the testing for this project, the manual testing was more pleasant, fast, and bugs could be identified and resolved much quicker.

6.3 Playtesting

As discussed in the previous subsection, it was difficult to accurately test the game with just one player. When it came time for the user evaluation, which is discussed in greater detail in the following section, the play sessions were also considered as part of the testing process. Until that point, the game had not been tested in a real environment with enough people to play the game. While the play sessions were mostly problem free, there were a few bugs that were made apparent with the first test groups. This was to be expected and the issues were hard to find edge cases.

For example, in order for one of the bugs to be reproduced, the map needed to have generated with an empty top left corner. This by itself rarely happens because the map generation's algorithm places the starting tiles very close to the corners so in most scenarios the map would be generated with a tile in the top left corner. If that happens, if a player talks to a specific type of critter at a specific point in the game, the game would crash. This edge case scenario would have been hard to discover if not for the help of the playtesters.

By the end of the user evaluation, players were able to play the game with no bugs or crashes from start to finish multiple times. This should be enough proof that the final product ended up being sufficiently bug free despite the various challenges faced during the testing process.

7 Results & Evaluation

This section covers the outcome of the project and how it compares to the specifications outlined in Subsection 3.4. This section also takes a look at the evaluation process that was undertaken, the results from it, and possible interpretations of the those results.

7.1 Final Product

As mentioned in previous sections, the final result of this project is a fully playable online team-based game for PC for 4 to 8 players. To play it, players click on the *Play* button in the title screen to create a private room (Figure 47). From there, players can scan the QR code or navigate to the link shown on screen. The link takes the players to the page depicted in Figure 48.

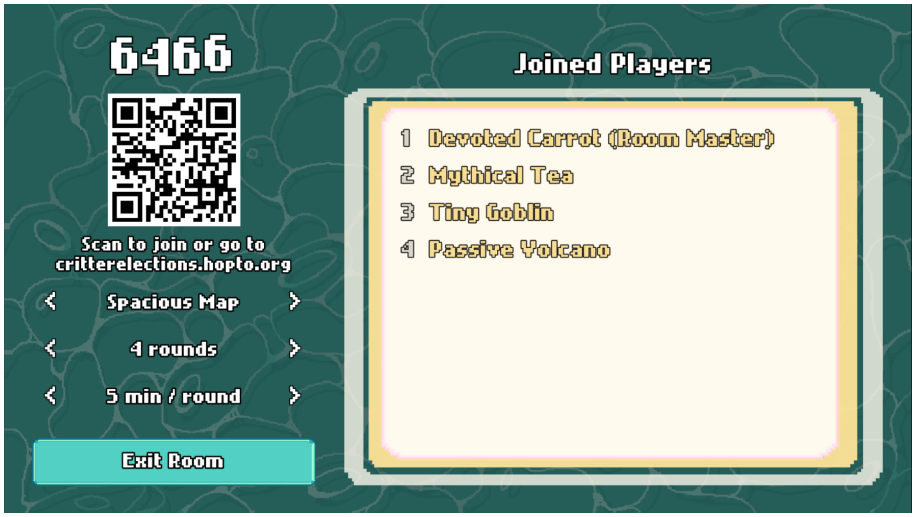

Figure 47: The room creation screen. Players can see the room key and the players who have already joined. Game settings, such as map size, number of rounds and round duration, can be set from this screen as well.

Figure 48: The first screen of the website players navigate to.

Players can enter the room shown on the game's screen (Figure 47) to join the private room. After that players can select a nickname and a team and a role. Once everyone has selected their role, the player who joined the room first can start the game by pressing the *Start Game* button on the controller (Figure 49). The time for each round is shown on the sailboat at the top of the screen. The sailboat itself will move from the left side of the screen to the right as a way to tell how much time is left in the round.

Players look at the map screen to see where their character is and move it with their preferred input method (for more info refer to Section 5.4.2).

Figure 49: How to start the game once every player is in the room and has selected a role.

Players can interact with various elements on the map by going on the same tile as them. These are some of the screens they might see on their controller from those interactions.

Figure 50: Making deals

Figure 51: Hiring bees

Figure 52: Placing bees

If a player gets disconnected while playing, be it from connection issues or they accidentally closed the tab in browser they were playing the game in, a *Disconnected* message will appear on their head (Figure 53). The game will still go on as if that player is not giving any input. The player can try to reconnect by going to the controller's website again and entering the same room key. The room key is always displayed during the game in the bottom right corner for this specific reason. From there they can select their character again and continue the game.

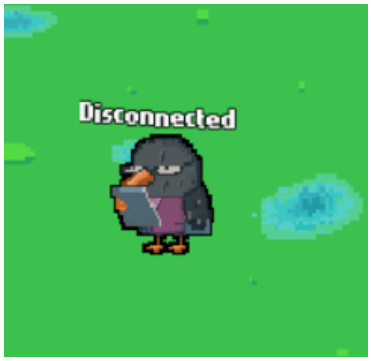

Figure 53: A disconnected player

Figure 54: The winner screen

When the final round ends, the scores are announced and a winning team is selected (Figure 54). For more information on how to play the game, refer to Appendix A. For more details on how the game works, refer to Section 5.

7.2 Evaluation Process

In order to properly evaluate the product user evaluation was required. An ethics application was submitted to the Ethics Committee and was approved. Details included in Appendix D.

The recruitment process was done by asking friends and other students through social media, whether they wished to participate. Those who agreed were put into groups of four people. To simplify the recruitment process the groups were made up of people who knew each other. The participants were made up of young adults in their twenties.

In total sixteen people participated in the evaluation for a total of four groups. The meetings for three of those groups were done through Discord and in-person for the remaining group. I was present for each of these meetings. The participants of each group were asked to play the game two times.

During the first play session, participants were asked to play the game as if I was not present. This meant that they would have to figure out how to play the game and interact with the user interface on their own. This was done to simulate a real world example of the game being played. The participants were encouraged to talk to the other participants in the group and try to figure out how to play together. They were also encouraged to vocalise their thought processes when playing the game. After the players finished the first game, they could give their impressions, feedback, and ask

questions about the rules of the game.

During the second play session, the participants were asked to play as a different role and make a team with a different person from the first game. This was done to allow the participants the chance to experience the most they could from the game during the given time. During this session, participants were free to ask me any questions about the game and I would answer them. This was done to see if, even after participants understood my intentions for the game, they were still confused about certain elements of the software.

After the second game ended, participants were asked to fill out a survey on their experience with the game. The questions in the survey used a 7-point Likert scale. The 7-point Likert scale was chosen as it is believed to more accurately measure participant's true evaluation when compared to a 5-point scale [46]. Some of the questions in the survey were about the participant's general enjoyment of the game. Others were about whether the game required good team skills and what they thought of their teammates in the game. The final questions were left open for players to elaborate on any of the previous questions or to give ideas for what they think could improve the game. A list of the questions included in the survey can be seen in Appendix D.6.

7.3 Evaluation Results

Out of the sixteen participants, fourteen submitted their answers to the survey. The results of the 7-point Likert scale questions can be seen in Table 2. For a more detailed look at the survey results, refer to Appendix D.7.

Table 2: Definitions of the terminology that is used

	Min	Max	Avg
1. The game was fun to play.	5	7	6.43
2. The game was stressful to play.	2	7	4.29
3. The game rules were easy to understand.	2	5	3.86
4. The game was challenging.	3	7	5.07
5. Good teamwork was required to excel at the game.	4	7	6.36
6. I felt closer to my teammates after playing the game compared to before I started.	2	7	4.71
7. I trusted my teammates in the game.	4	7	6.36
8. I gained a newfound appreciation of my first teammate's role after playing as that role during the second game.	3	7	5.86
9. I would play the game again.	4	7	6.5

For question **"Is there anything you would like to elaborate on regarding your answers to questions 1 - 9?"**, nine participants responded. Most of them expressed their enjoyment of the game. Comments regarding input delay and the game being stressful were also made.

For question **"Is there anything you think could be improved upon the game? (instructions, gameplay, readability, etc.) If so, what do you think can be done to improve it?"**, eleven participants responded. Most of the them expressed that they wished the rules of the game were explained better. The second most common type of comment was about UI improvements.

7.4 Interpreting the Results

When it comes to the enjoyment of the game, most participants seem to have had fun playing based on the results in Table 2. Out of all of the questions, question 1 and 9 are the highest. During play, players did engage in frequent conversation and expressed emotions of joy from succeeding as well as frustration when another team was in their way. To add on top of that, two of the four test groups asked to play again even after they had already played two games in a row, which is promising. The game was successful in being fun and eliciting various emotions from its players.

Most players felt that the game was a little stressful. This is likely due to the slight competitive nature of the game and the timed rounds. As a whole the score for whether the game is stressful is close to being neutral. This is a desired result since players should feel some pressure to succeed in the game but that should not overwhelm them.

Importantly, almost everyone felt that in order to succeed in the game good teamwork was required. This a great success since it was one of the goals for this project. Most players also said they felt like they could trust their teammate to do well and help them in the game, which is also important to note for the goals of this project.

Question 6 has a lower average than expected. This is likely because the players in the group already knew the people they were playing with and did not know how to answer. One of the comments later expresses exactly this and during the in-person group meeting, one of the participants also expressed the same feeling. The question was not suited for the people who ended up participating.

The biggest failure, which was felt during the play sessions and can also be seen in the results of the survey, is that the rules could have been clearer. I want to mention that from my perspective, a

majority of players figured out the ideas behind the game during their first play session.

7.5 Iterating on Feedback

Of course, I did not ignore the feedback the participants gave. After each test group, alterations were made in preparation for the next one. This subsection covers some of those changes.

During the very first play session with the first group out of the four groups, one of the participants played as the Bee Manager role. When that player entered gathering spots to place bees in them, they waited in there until the bees finished their job. This is not how you are intended to play as the Bee Manager. The intended way is for the Bee Manager to place their bees and leave to do something else. The player may have thought that they needed to be there with the bees when they finished working in order to get the resources or maybe the progress bars below the bees made them think they have to wait.

With inspiration from a talk done by George Fan [47], a message box was added to the gathering spot screen in the controller. This box would only show up if a player stayed in there for too long without performing any actions (Figure 55). This way only players who misunderstand the functionality of the bees will be told how it actually works, while players who intuitively get it will never see it.

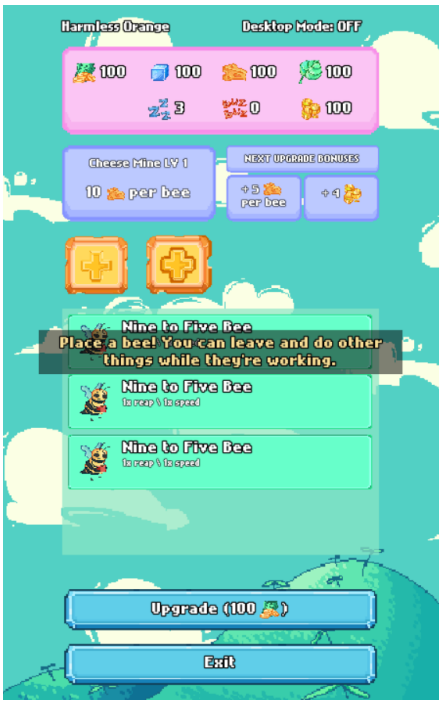

Figure 55: The message that shows up

One of the alterations I made was not based on feedback given to me directly but from watching how the participants played. When creating the beehive functionality of the game (this is where Bee Managers can hire bees from), I thought that it would be a good idea to only place the building once someone gets honeycombs (the currency used in the beehive) for the first time. Because at the beginning of the game, everyone starts with zero so no bees can be hired. I presumed that the beehive would be confusing if someone went in it at the beginning, saw they could not do anything there and wondered what they were supposed to do. By placing the beehive only once a player got their first honeycombs, I thought that they would notice the beehive appearing on the map, go to it, and find out immediately how the honeycombs they just got are supposed to be used.

What I did not foresee was that when a player gets their first honeycombs, they are looking at the controller, not the map screen. By the time they looked back at the map, the beehive was there but with so many other sprites moving on the map, it did not stand out as something that just appeared. So my initial idea to help players actually led to some players completely missing the beehive and with that a core element of the Bee Manager's gameplay. The solution was to just place the beehive on the map from the start. By doing that, players naturally visited it at least once in the beginning when surveying the map and had an idea that it existed.

The part of the game that went through the most iterations is likely the rule pages in the controller. As seen from the survey results, players were confused by the rules. The very first version of the rule pages had a lot of flaws. There was little mention on how to actually get votes which is the way you win the game and had a lot of fluff that just made the rules longer and harder to read. The next iteration added more sprites to represent the mechanics in the game. In next iterations the goal of the game was plastered on the very first page, with big lettering. Some parts of the rules were completely removed as they were intuitive enough on their own in the game. For example, it was unnecessary to tell players that if they can pick up fallen money. Players intuitively did that when playing and knew what happened when they did it. Examples were added as to what each role is supposed to do in the game and how they interact. What made the rules much better was to keep everything as short as possible, add sprites to show what is what, and examples of what players actually did in the game not just rule explanations. The final version can be seen in Appendix A.2.

7.6 Specification Satisfaction

When it comes to the functional requirements that were laid out in 3.4.1, all of the must have and should have requirements as well as one of the could have requirements (the one about having settings for the game in the room creation screen). All of the user stories from 3.4.3 have also been met as they were needed for the game mechanics to work properly. All of them including the functional requirements that were met were manually tested to work multiple times.

As for the non-functional requirements:

- **Controller can run on all popular modern browsers (Chrome, Edge, Firefox, Safari, Samsung Internet, and Opera).**

  The controller was tested on all of the specified browsers and proved to function as expected.

- **Players can play remotely using services like Discord and have a similar experience to when playing in-person.**

  As mentioned above in the user evaluation sections, three of the test groups' meetings happened through Discord. That includes playing the game by sharing the screen of the player who is running Main Game and players watching that as they play. All of the groups were able to play the game and have fun, which means that the delay introduced by the stream was not severe enough to impact the enjoyment of the game. The requirement can be considered as being met.

- **Players in the same team have meaningful interactions.**

  As can be seen in the survey results, players almost unanimously said the game required good teamwork. To add to this, from being present at the play sessions, I can attest to players often communicating about what their plan of action in the game is. Judging from that, the requirement has been met.

Overall, all of the main specifications in 3.4 have been met.

8 Discussion & Reflection

8.1 Reflection

One of the main mistakes made during this project was starting the development of the game too late. It should be expected that a project will take longer than anticipated because it is hard to realise how much work there truly is to be done until the work has been started. Because development began late, by the time playtesting started to become feasible there was not enough time. With a game project ideally half of the time should be spent on playtesting and iterating.

As a whole, the game met the main requirements and was fun enough for some players to want to play even after the user evaluation was over. As the person who proposed the idea for this project, I can say that I am satisfied with the end result.

8.2 Challenges

As mentioned in Section 6, testing as a whole was quite challenging. Automated tests were difficult to set up and the manual testing was not ideal due to the game being multiplayer and requiring a minimum of four players.

The complicated state machines in the Main Game and Controller projects, combined with them influencing one another with possible delays due to connection issues, was challenging. If either Main Game or Controller ends up in a wrong state, it is likely any commands or information they receive from the other will not be interpreted properly, which can lead to a player being locked in a state or the software crashing. The state management needed to be done carefully to avoid such scenarios.

The deployment was personally difficult because it was the first time I had to set up a server. That by itself did not prove to be too hard, but during the first attempts the connection between the server and the controller did not work due to various tiny details, like browser restrictions and limitations from Godot.

8.3 Limitations

One of the initial non-functional requirements was for the server to be able to run at least ten separate rooms at once without issues. This was removed from the list because there was no easy way to

test. Which means that there is no certainty what load the server can handle.

The main idea for this project was to make a game that would be played by newly formed teams. In other words, people that do not know each other. Unfortunately, it proved difficult to gather such groups for testing and due to lack of time friend groups were brought on for the user evaluation. The product did not end up getting tested by its target audience.

Because there were two servers that needed to be hosted, it was hard to find where to host them for free. The solution was to use AWS's free tier but even the free tier would not allow for both servers to run at all times. Ideally, both servers would run 24/7, which would reduce the need for setup during playtesting.

8.4 Future Work

The main one would be to do more playtesting and iteration. As mentioned in above, there was not enough time left after the development of the project to playtest with a group of players.

For possible functionalities, a way for an odd number of players to be able to play would be on the top of the list. Two ways of achieving this were include as could have functional requirements: AI players or being able to play alone as opposed to having to play in a team. Both of these are antithetical to the main spirit of the game, which is teamwork. With an AI player, it would be hard for a player to properly communicate with them what they need them to do. Functionality could be added which allows a player who is in a team with an AI player to give out commands but that would be similar to playing alone instead of in a team. Playing alone, as both roles in one character, can be done quite easily at present but will likely not be as fun. It was not the intended way for playing the game and therefor was not considered during development, which may lead to other unintended consequences.

During the user evaluations one of the groups gave an interesting idea. A new unique role that is for playing alone. This role would not be a combination of the two existing roles but an entirely new one with its own mechanics. The mechanics could be built in a way where that role had to interact with the other teams frequently. Maybe one round it would play for one team, the next round another. Ideally, it would be incorporated in a way that would allow it to have meaningful interactions with other players even though that player is not in a team.

9 Conclusion

Through the course of this project, an online team based video game titled "Critter Elections" was designed, developed in the Godot game engine, and evaluated. Automated testing, manual testing through custom software, and multiple rounds of playtesting were done to verify and validate the product. It is accessible by being playable by anyone who has a smart device. The evaluation proved the game can facilitate good teamwork skills. The project was able to meet its specified goals, which leaves me as the author, pleased with the outcome.

References

[1] Randy. Why are game controllers getting so expensive? (explained). URL: https://whatsabyte.com/why-are-game-controllers-getting-so-expensive-explained/ (visited on 03/28/2023).

[2] A timeline of video game controversies. URL: https://ncac.org/resource/a-timeline-of-video-game-controversies (visited on 03/23/2023).

[3] B. Chaarani, J. Ortigara, D. Yuan, H. Loso, A. Potter, and H. P. Garavan. Association of video gaming with cognitive performance among children. *JAMA Network Open*, 5, 10, 2022. URL: https://jamanetwork.com/journals/jamanetworkopen/fullarticle/2797596 (visited on 03/23/2023).

[4] A. Novotney. Do educational computer and video games lead to real learning gains? psychologists say more research is needed. *American Psychological Association*, 46:46, 4, Apr. 2015. URL: https://www.apa.org/monitor/2015/04/gaming (visited on 03/23/2023).

[5] T. Lougheed. Video games bring new aspects to medical education and training. *CMAJ*, 191:E1034–E1035, 37, Sept. 2019. DOI: 10.1503/cmaj.1095784.

[6] T. Greitemeyer and S. Osswald. Playing prosocial video games increases the accessibility of prosocial thoughts. *The Journal of Social Psychology*, 151:121–128, 2, Feb. 2011. DOI: 10.1080/00224540903365588.

[7] E. Tan and A. L. Cox. Trusted teammates: commercial digital games can be effective trust-building tools. *CHI PLAY '19 Extended Abstracts: Extended Abstracts of the Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*:705–713, Oct. 2019. DOI: 10.1145/3341215.3356296.

[8] T. Simmons. Team building in world of warcraft. May 2010. URL: https://todd-simmons.com/docs/MBA_BusComm_TeamBuilding.pdf (visited on 03/23/2023).

[9] Captain sonar. URL: https://www.matagot.com/en/catalog/details/expert-games/1/captain-sonar/808 (visited on 03/23/2023).

[10] T. Olsen. Captain sonar: someone finally made a great 8p board game. URL: https://arstechnica.com/gaming/2016/12/captain-sonar-someone-finally-made-a-great-8p-board-game/ (visited on 03/28/2023).

[11] The world of pandemic. URL: https://www.zmangames.com/en/games/pandemic/ (visited on 03/23/2023).

[12] M. Kumar. 5 problems with co-op game design (and possible solutions). URL: https://www.gamedeveloper.com/design/5-problems-with-co-op-game-design-and-possible-solutions- (visited on 03/23/2023).

[13] Ladies & gentlemen. URL: https://boardgamegeek.com/boardgame/124380/ladies-gentlemen (visited on 03/23/2023).

[14] How to play - jackbox games. URL: https://www.jackboxgames.com/how-to-play/ (visited on 03/23/2023).

[15] D. Murko. Most watched esports games of 2022. Jan. 2023. URL: https://escharts.com/news/most-watched-esports-games-2022 (visited on 03/23/2023).

[16] R. Garfield. Richard garfield - "luck in games" talk at itu copenhagen. Sept. 2013. URL: https://www.youtube.com/watch?v=av5Hf7u0u-o (visited on 03/23/2023).

[17] Spaceteam - a game of cooperative shouting for phones and tablets. URL: https://spaceteam.ca/ (visited on 03/23/2023).

[18] Spaceteam faq. URL: https://spaceteam.ca/faq/ (visited on 03/23/2023).

[19] H. Smith. Spaceteam trailer. Sept. 2013. URL: https://www.youtube.com/watch?v=y3fsvKnIVJg&t=66s (visited on 03/28/2023).
[20] Godot engine - free and open source 2d and 3d game engine. URL: https://godotengine.org/ (visited on 03/23/2023).
[21] Github pages — websites for you and your projects, hosted directly from your github repository. just edit, push, and your changes are live. URL: https://pages.github.com/ (visited on 03/23/2023).
[22] G. Fiedler. What every programmer needs to know about game networking. Feb. 2010. URL: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/ (visited on 03/23/2023).
[23] In B. Bates. *Game Design*. 2nd edition. 2nd edition, 2004. Chapter Selecting the Right Lifecycle Model, pages 225–230.
[24] S. Aleem, L. F. Capretz, and F. Ahmed. Game development software engineering process life cycle: a systematic review. *Journal of Software Engineering Research and Development*, 4, Nov. 2016. DOI: 10.1186/s40411-016-0032-7.
[25] I. Sommerville. *Software engineering*. In 10th edition. Pearson Education Limited, USA, 10th edition, 2016. Chapter Requirements change, pages 130–134.
[26] Chapter 10: moscow prioritisation. URL: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html (visited on 03/23/2023).
[27] Browser market share worldwide. 2023. URL: https://gs.statcounter.com/browser-market-share (visited on 03/23/2023).
[28] Worker placement. URL: https://boardgamegeek.com/boardgamemechanic/2082/worker-placement (visited on 03/27/2023).
[29] Overview of godot's key concepts. URL: https://docs.godotengine.org/en/3.5/getting_started/introduction/key_concepts_overview.html#nodes (visited on 03/28/2023).
[30] Godot engine - community. URL: https://godotengine.org/community/ (visited on 03/28/2023).
[31] Exporting for the web. URL: https://docs.godotengine.org/en/3.5/tutorials/export/exporting_for_web.html (visited on 03/28/2023).
[32] High-level multiplayer. URL: https://docs.godotengine.org/en/3.5/tutorials/networking/high_level_multiplayer.html (visited on 03/28/2023).
[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patters: elements of reusable object-oriented software*. In Addison Wesley, USA, 1994. Chapter Observer, pages 293–305. ISBN: 0-201-63361-2.
[34] Gut. URL: https://github.com/bitwes/Gut (visited on 03/28/2023).
[35] Aws. URL: https://aws.amazon.com/ (visited on 03/28/2023).
[36] Amazon ec2. URL: https://aws.amazon.com/ec2/ (visited on 03/28/2023).
[37] The devsecops platform — gitlab. URL: https://about.gitlab.com/ (visited on 03/28/2023).
[38] Qr code generator. URL: https://www.the-qrcode-generator.com/ (visited on 03/28/2023).
[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patters: elements of reusable object-oriented software*. In Addison Wesley, USA, 1994. Chapter Factory Method, pages 107–116. ISBN: 0-201-63361-2.
[40] R. Nystrom. *Game programming patterns*. In USA, 2014. Chapter State, pages 120–144. ISBN: 978-0-9905829-2-2.

[41] Super auto pets. URL: https://teamwoodgames.com/ (visited on 03/28/2023).
[42] R. Nystrom. *Game programming patterns*. In USA, 2014. Chapter Singleton, pages 102–119. ISBN: 978-0-9905829-2-2.
[43] Z. Gage. Controls you can feel: putting tactility back into touch controls. 2012. URL: https://www.youtube.com/watch?v=OIOitNx9RHI (visited on 03/28/2023).
[44] Dynamic ip address got you down? URL: https://www.noip.com/ (visited on 03/28/2023).
[45] C. Politowski, F. Petrillo, and Y.-G. el Gu eh eneuc. A Survey of Video Game Testing, 2021. DOI: 10.48550/arXiv.2103.06431.
[46] K. Finstad. Response interpolation and scale sensitivity: evidence against 5-point scales. *Journal of Usability Studies*, 5:104–110, 3, May 2010.
[47] G. Fan. How i got my mom to play through plants vs. zombies. 2012. URL: https://www.youtube.com/watch?v=fbzhHSexzpY (visited on 03/29/2023).

A Appendix - Game Rules

A.1 Detailed Rules

The following rules are not an ideal way to learn how to play the game. They are included here as a complete overview of how the game and its mechanics work.

A.1.1 How to Start

Open the PC game and click *Play*. A private room will be created which players who have the displayed room key can join by following the shown URL or scanning the QR code. The player who joins the room first is appointed the room master. Once players have selected their names, teams and roles, if there are at least two teams and every team has exactly two players in it, the room master can start the game from their device. The shared PC screen will show a tile based map on which the players can do various context sensitive actions.

A.1.2 How to Win

The goal of the game is to get your team's candidate elected as mayor of Critter Island. To get elected, your team needs to be the top candidate in the most areas of the map compared to the other teams. To be a top candidate in an area, your team needs to have the most votes in that area (not majority). There are three areas in the game (Green Area, Snowy Area, Urban Area), which means that in most cases a team needs to have control over two of them to win.

A.1.3 Rounds

The game is divided into a customizable number of rounds. Each rounds lasts for a customizable amount of time. The default is three rounds for three minutes each. At the end of every round players are shown how many voters they have gathered in all of the areas and how they compare to the other teams. At the end of the last round, the election results are shown and winner is selected based on them.

A.1.4 Teams & Roles

Players are divided into teams of two players. The game can be played by two to four teams. Players in a team select one of two unique roles: Candidate or Bee Manager. Only one player per team can

be a certain role. Players in a team, do not have a shared pool of resources.

A.1.5 Shared Actions

Shared actions that any player, no matter their team or role, can perform.

Players can move on the map shown to them on the shared screen by swiping on their mobile device in the direction they want to move. If they are playing on desktop, they can turn on desktop mode which shows them directional buttons they can click to move. In desktop mode, players can also move with the arrow keys or WASD on their keyboard. Multiple players cannot stay on the same tile. Players can also tap on their character, which causes a short message in a speech bubble to be displayed next to their character on the shared screen of the game. This can be used as a way for players to find themselves on the map if they get lost.

Players can pick up fallen money on the ground as well as small snails when they step on the same tile as them. Money is one of the main resources in the game and picking it up from the ground adds a random amount of it that player's pool of resources. Picking up small snails gives that player's team a small amount of voters in the area the snail was picked up in.

Players of the same team can also exchange resources amongst themselves, but exchanging as a candidate from exchanging as a bee manager. Exchanging will be explained in greater detail in A.1.8.

A.1.6 The Candidate Role

The candidate is the only role that can interact with non-player critters. Critters are the residents of Critter Island. When a candidate is on the same tile as critter, they can press the *Discuss Deal* button that pops up on the device they are using to play to talk to that critter. The critter will give them a deal of the form: *"I want at least X amount of resources. In return I will give you Y amount of resources"*. Candidates can do one of two things: decline or give them an offer.

There are five types of critters:

- Rats – want cheese and give money in return;
- Penguins – want ice and give money in return;
- Snails (different from the small snails) – want leaves and give money in return;

- Positive Pigeons – want money and give voters in the area they are located in
- Negative Pigeons – only accept deals, by candidates who are not the top candidate in the area the pigeon is located in. They want money and take voters from the top candidate in the area they are in and give them to the candidate who gave them the best offer.

If a candidate declines a deal, they are locked off from making a deal with that instance of the critter until the end of the game. If all of the candidates decline a certain critter's offer, that critter will disappear, making room for a new one to appear with a new deal.

A candidate can give an offer to the critter by either directly pressing the "Offer" button, in which case they will give the critter the minimum amount of resources the critter wanted. A candidate can also change their offer by using a slider to set the amount of resources they are willing to give to that critter. In both cases, the critter accepts, takes the resources from the candidate, and gives a response of the form: *"I will wait to see if another candidate gives me a better offer. If I get a better offer, I will return what I took from you. If I do not, I will give you my part of the deal"*. The critter will wait a certain amount of time (20 seconds in the final version) to give other candidates a chance to give an offer.

If another candidate shows up, they are given the same message the first candidate was at first, with the difference that the critter tells them that another candidate has already made them an offer so they will have to outbid them if they want the critter's reward. The amount the first candidate gave is kept a secret. If the candidate who talked to the critter second ends up outbidding the first candidate, they get the critter's reward and the amount of resources the first candidate gave is returned to them. If the second candidate fails to outbid the first player, they are shown a message informing them so. In that scenario, the first candidate receives the reward from that critter. Critters only wait for two candidates to give them an offer before giving their reward. A third candidate will not be given the opportunity to make an offer to that critter.

To prevent players from blocking other player from making deals with critters while their timers are active, the timer only counts down if there are not players on the same tile as the critter.

A.1.7 The Bee Manager Role

The bee manager is the only role that go into buildings. Bee managers are called that, because they place bees in buildings to gather resources for them.

There are four types of buildings:

- Cheese Mines – produce cheese;
- Ice Igloos – produce ice;
- Leafy Bushes – produce leaves;
- Beehives – a place where bee managers can hire more bees from in exchange for honeycombs.

To enter a building, a bee manager needs to be on the same tile as it and press the *Enter* button. Buildings have a limited amount of free spaces to place bees in them. Once inside a building that produces resources, they can press one of the free locations there to put a bee. The bee will work there for a set amount of time, which is different depending on the type of bee. Once the bee finishes its work, it will return to the bee manager with the resources it collected. Bee managers do not need to stay in a building while their bees work there. They can leave and go elsewhere. The amount of resources a bee can get from a building is displayed in the info section of the building. However, different bees can affect the amount of resources they can reap from a building.

Bee managers can also upgrade buildings that produce resources. To upgrade a building they need to pay the specified amount of money for the upgrade. Upgrades come with two benefits, one for the building and one for the bee manager that upgraded it. The building will either get an extra free space where a bee can work in or produce more resources per bee. The bee manager will get honeycombs, which can be used in beehives.

Bee managers can hire more bees from beehives. Beehives show a list of bees that can be hired and their price. The price of bee is decided by its type. However, the lower a bee is on the list in the beehive the more expensive it is. When a bee is hired, all the other bees below it move up a space, reducing their cost by one honeycomb. This means that the bee at the top of the list will have a price, which is exactly the same as its default price based on its type, but every other bee in the list will have an increased price of one honeycomb for every bee in the last above it. This is done so that if a bee is undesirable, it will eventually reach the top of the list and get cheaper, making it a

more tempting deal.

There nine types of bees:

- Nine to Five Bee – the standard bee, bee managers start the game with three of them;
- Hustling Bee – reaps twice the amount of resources when placed;
- Swift Bee – finishes its work two times faster than the average bee;
- Gambling Bee – it has a 50/50 chance of either getting no resources when places in a building or get five times the resources;
- Home Cook Bee – when it finishes its work, gives one honeycomb to the its bee manager;
- Mirror Bee – copies the ability of the bee to the left of it when placed to work in a building;
- Cheese Loving Bee – reaps three times the amount of resources if placed in a Cheese Mine;
- Ice Loving Bee – reaps three times the amount of resources if placed in an Ice Igloo;
- Leaf Loving Bee – reaps three times the amount of resources if placed in a Leafy Bush;

A.1.8 Exchanging Resources Between Teammates

When players in the same team are adjacent to one another, they can trade the resources they have gathered so far. A button will pop out and if both players press it, the exchange will occur. The candidate gives all of their money to the bee manager and the bee manager gives all of their cheese, ice, and leaves to the candidate. This is the only type of trade players can do. Candidates cannot give cheese, ice, and leaves to bee managers and bee managers cannot give money to candidates.

A.1.9 General Strategy

Candidates need money to get voters from the pigeons but will have a difficult time gathering enough money if they only pick it up from the ground. Candidates get a lot more money by making deals with rats, penguins, and snails. However, they need resources to make these deals and they have no way of getting resources on their own.

Bee managers are very inefficient at first at gathering resources. They only have three bees. They need to get more bees and make their buildings better by upgrading them. To get more bees, they

need honeycombs. The only way they can get honeycombs at first is by upgrading buildings, which costs money. Bee managers can pick up money from the ground but it will prove to be an insufficient way of doing it. They need a lot of money to get better at getting resources.

Candidates and bee managers need to trade often to give each other what they need. While bee managers can give their resources to the candidate without worry, since they do not need them for anything, candidates need to be mindful when they trade because they will give all of their money to the bee manager and cannot take it back. Candidates need their money to make deals with pigeons, which give them votes, which is what their team ultimately needs to win the game. However, they also need to give money to their bee manager to increase the amount of resources the bee manager can give them later and to stay competitive with the other teams.

A.2 In-Game Rules

To avoid players getting overwhelmed with rules, only the most important information was included in the rules pages to teach players. A majority of players were able to figure out the finer details once they started playing and due to the nature of it being a team based multiplayer game, players helped each other in understanding the game. The game rules were updated after every evaluation session based on the feedback given by the testers. The following rule pages are the final version of the rules.

(a) Page 1

(b) Page 2

(c) Page 3

(d) Page 4

Figure 56: The rule pages included in the game

B Appendix - Godot Overview & Code Examples

B.1 Godot & the Godot Editor

Godot is a free open-source game engine and was the primary tool used in the creation of this project. Godot comes with an editor, which is interestingly enough, built in Godot itself (Figure 57). The map screen of the Main Game project will be used to illustrate some of Godot's main functionality along with examples the code that is used to when the screen is first loaded.

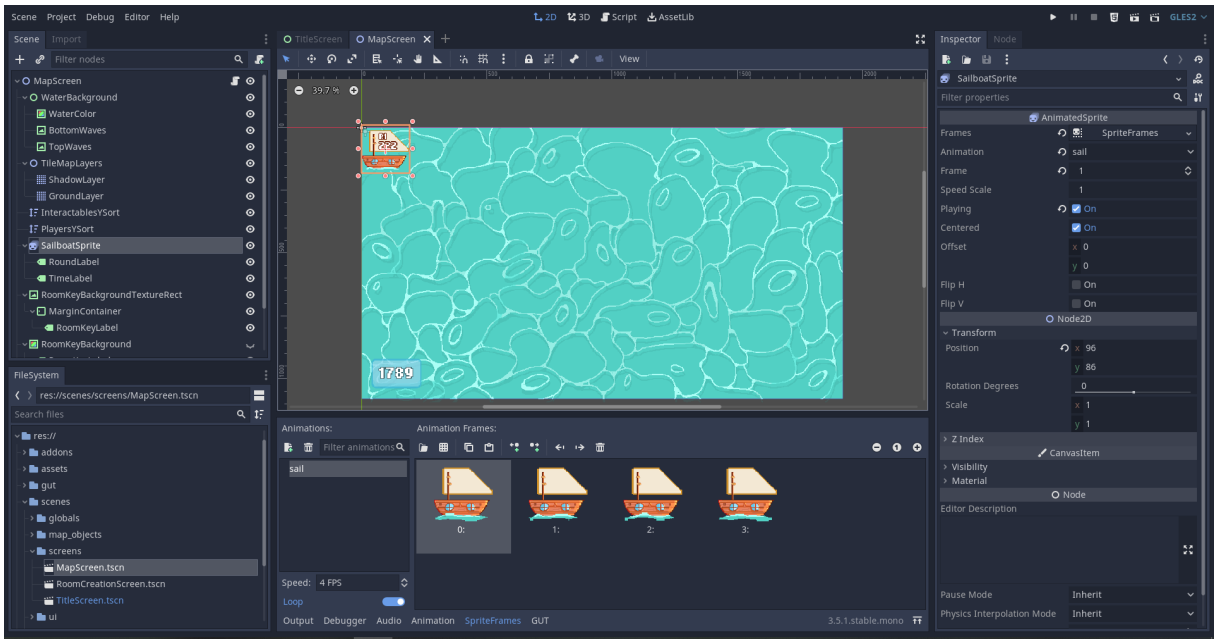

Figure 57: The Godot editor

Godot uses a node-based system. Practically, every object that is used in the Godot editor is a node and inherits the base *Node* class. For example, the class hierarchy of the Label node looks like this (Figure 58).

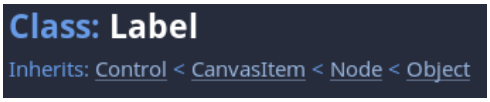

Figure 58: The class hierarchy of the Label node

Nodes have children, which are also nodes. This relationship builds a tree structure. A node can be saved as a file which is referred to a scene. These scenes can then be reused in other scenes and instanced in code, they can be thought as reusable nodes. To start a Godot game, a start scene needs to set. For Main Game, the starting scene is the title screen. When the game starts, all of the

nodes in the title screen scene are loaded in. In Main Game, when players press the *Play* button, the title screen nodes are freed from memory and the room creation screen scene nodes get loaded. Eventually the players will join the room and the actual game will start, which is held entirely in only one scene, the map screen scene (Figure 59).

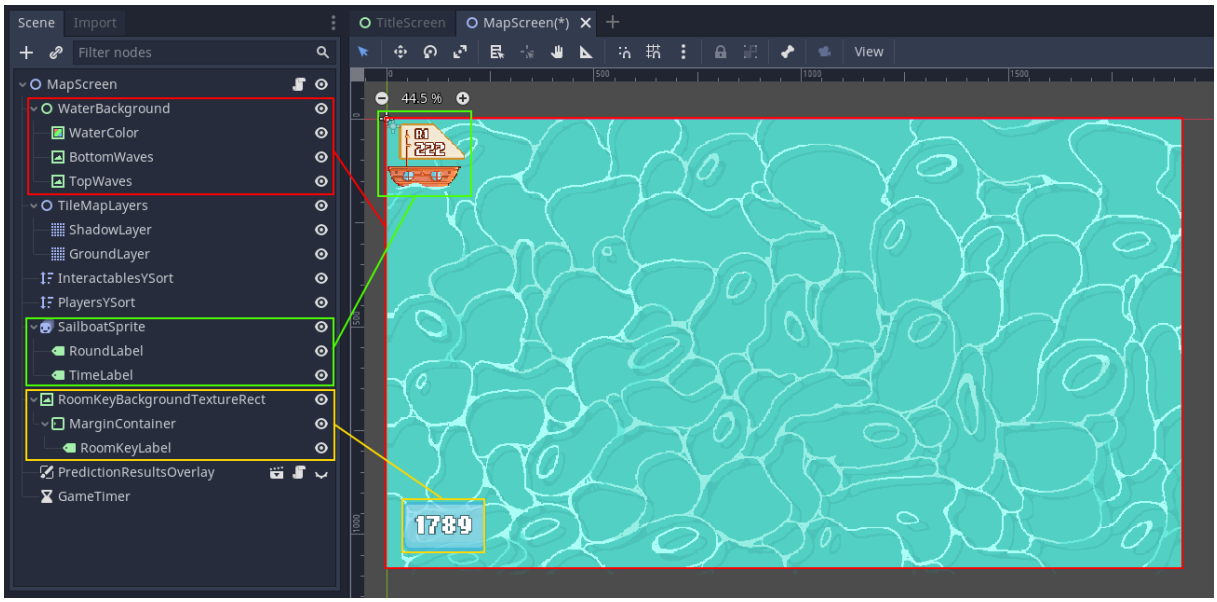

Figure 59: The map screen's scene tree

Godot comes with a plethora of prebuilt nodes for various uses. Sprite nodes which are used to display sprites, button nodes which have the base functionality expected of a button, and dozens more. As illustrated in the above figure, the water background in is made up of Control node (which is the base node for UI elements), which has three children which are texture nodes that represent the layers of the water. The sailboat sprite node has two label nodes as children, which are used to show the time left in a round and the round number. The nodes which seemingly show nothing on the screen, have their properties changed in code or are used as parent nodes for nodes created in code. More on that further below. The blue nodes are nodes that are used for 2D games, the green nodes are for UI, and the white nodes are miscellaneous, which have varying uses. An example of a white node is the Timer node (named *GameTimer*), which can be seen at the bottom of the node hierarchy in Figure 59. The node directly above it is a actually a scene, which contains all of the nodes used in the results overlay. The *PredictionsResultsOverlay* scene is quite complicated on its own and uses many types of UI nodes (Figure 60). It also has other scenes within it. The basics of Godot is creating more specialised complicated nodes with simpler ones.

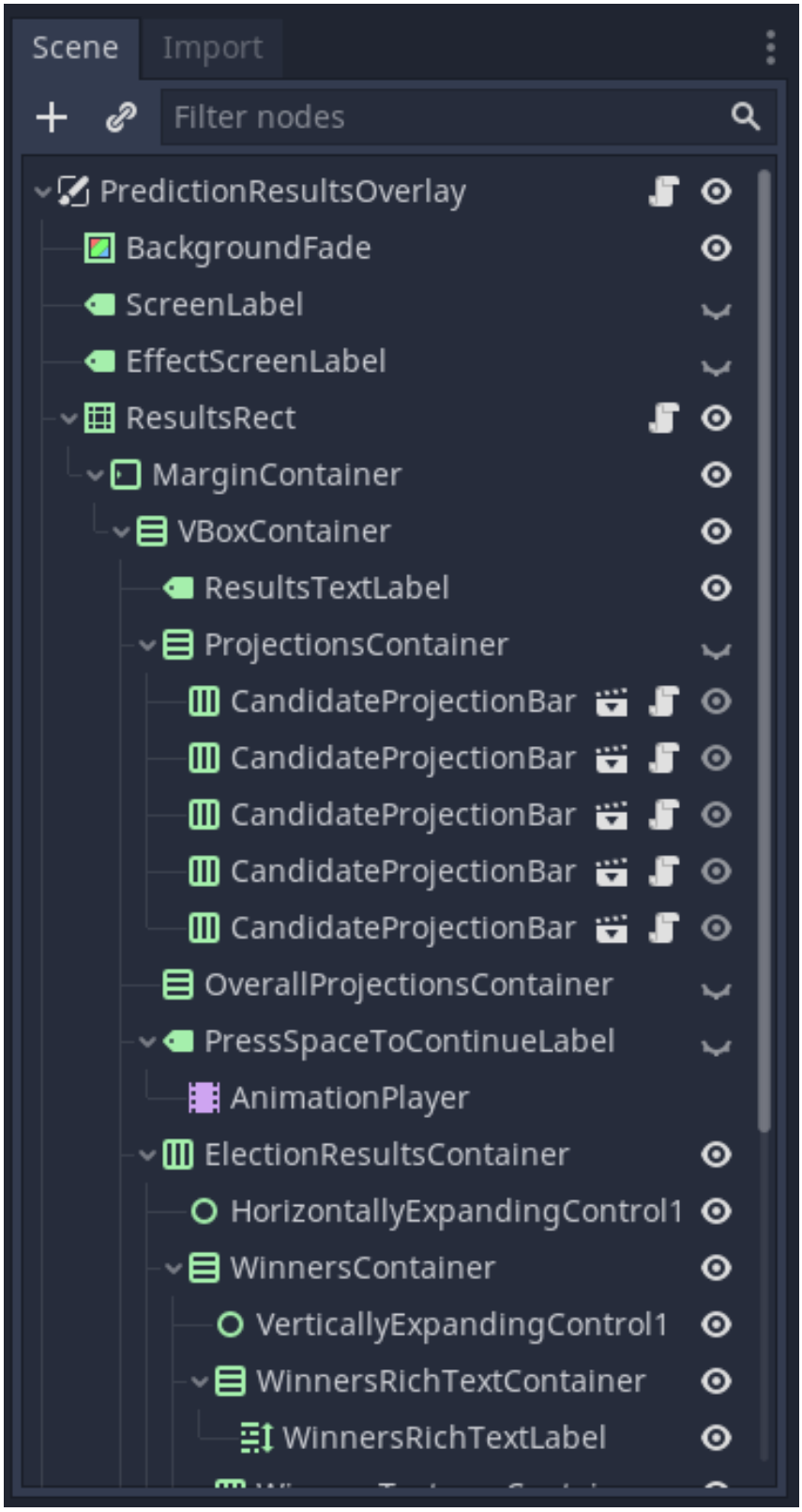

Figure 60: *PredictionsResultsOverlay*'s scene tree

B.2 Code Examples

The reason the map screen scene looks simpler than the overlay scene, even though when playing the game it is obvious that the map has a lot more elements in it is that all of those elements for the map are created in code. The nodes placed in the editor are retrieved in code as seen in Figure 61. The properties of those can then be manipulated. New nodes and scenes can also be created from within the code.

```
39  onready var tile_map_layers: Node2D = $TileMapLayers
40  onready var shadow_layer: TileMap = $TileMapLayers/ShadowLayer
41  onready var ground_layer: TileMap = $TileMapLayers/GroundLayer
42  onready var players_y_sort: YSort = $PlayersYSort
43  onready var interactables_y_sort: YSort = $InteractablesYSort
44  onready var prediction_results_overlay: CanvasLayer = $PredictionResultsOverlay
45  onready var room_key_bg_texture_rect: TextureRect = \
46      $RoomKeyBackgroundTextureRect
47  onready var room_key_label: Label = \
48      $RoomKeyBackgroundTextureRect/MarginContainer/RoomKeyLabel
49  onready var sailboat_sprite: AnimatedSprite = $SailboatSprite
50  onready var round_label: Label = $SailboatSprite/RoundLabel
51  onready var time_label: Label = $SailboatSprite/TimeLabel
52  onready var game_timer: Timer = $GameTimer
```

Figure 61: How the nodes from that can be seen in the editor are retrieved in code

When a node and all of its descendants have loaded, the *_ready* function of that node is called. The map screen's *_ready* function (Figure 62) creates, the map, the elements on it, the players and stars various timers to keep track of rounds and when to instantiate other map objects. Further details on some of the functions called in the map screen's *_ready* function below.

```
54  func _ready() -> void:
55      randomize()
56      _set_map_size()
57      _set_tile_map_offset()
58      _set_tileset_cells()
59      _set_map_data()
60      _init_interactables()
61      _init_players()
62      _place_players()
63      _connect_signals()
64      _start_resource_timer()
65      _start_event_character_timers(MapData.projections.keys())
66      _set_room_key_elements()
67      _start_round()
68      PauseOverlay.is_pausible = true
```

Figure 62: MapScreen's *_ready* function

In *_set_tileset_cells*, the map is randomly generated in the *_generate_valid_map* function and based on it, the cells in the tile set nodes of the map screen are set (Figure 63).

```
116 func _set_tileset_cells() -> void:
117     # Clear tilemaps:
118     shadow_layer.clear()
119     ground_layer.clear()
120     # Generate random map.
121     var map: Dictionary = _generate_valid_map()
122     # Set cells according to map.
123     for y in range(rows):
124         for x in range(columns):
125             if map.has(Vector2(x, y)):
126                 # Set shadow layer cells.
127                 shadow_layer.set_cell(x * 2, y * 2, 3)
128                 shadow_layer.set_cell(x * 2 + 1, y * 2, 3)
129                 shadow_layer.set_cell(x * 2, y * 2 + 1, 3)
130                 shadow_layer.set_cell(x * 2 + 1, y * 2 + 1, 3)
131                 # Set ground layer cells.
132                 ground_layer.set_cell(x * 2, y * 2, map[Vector2(x, y)])
133                 ground_layer.set_cell(x * 2 + 1, y * 2, map[Vector2(x, y)])
134                 ground_layer.set_cell(x * 2, y * 2 + 1, map[Vector2(x, y)])
135                 ground_layer.set_cell(x * 2 + 1, y * 2 + 1, map[Vector2(x, y)])
136     # Update tilemap bitmask regions.
137     shadow_layer.update_bitmask_region()
138     ground_layer.update_bitmask_region()
```

Figure 63: The function that generates the map and sets the tiles

In *_init_interactables*, the various map objects are created using the Factory pattern based on the map data, which was set beforehand (Figure 64).

```
327 func _init_interactables() -> void:
328     for location in MapData.map.keys():
329         _init_interactable_at_location(location)
330
331
332 func _init_interactable_at_location(location: Vector2) -> void:
333     if not MapData.is_inside(location):
334         return
335     var instance: Node2D
336     match MapData.map[location].type:
337         Enums.LocationType.GATHERING_SPOT:
338             instance = _instance_gathering_spot(location)
339         Enums.LocationType.HIRING_SPOT:
340             instance = _instance_hiring_spot()
341         Enums.LocationType.EVENT:
342             instance = _instance_character_event(location)
343         Enums.LocationType.PICKABLE_RESOURCE:
344             instance = _instance_pickable_resource()
345     _set_interactable(instance, location)
```

Figure 64: The various map objects that players interact with, being created using the Factory Method pattern

Most of the nodes in the game have a *_connect_signals* function, which connects signals from nodes to functions. This is how one node can communicate with another node in Godot (Figure 65).

```gdscript
func _connect_signals() -> void:
    game_timer.connect("timeout", self, "_timeout_game_timer")
    prediction_results_overlay.connect("closed_prediction_results", self, "_closed_prediction_results")
    WorldLevelStats.connect("upgraded_first_gathering_spot", self, "_world_level_stats_upgraded_first_gathering_spot")
    MovementManager.connect("player_moved", self, "_movement_manager_player_moved")
    MatchRoomManager.connect("closed_room", self, "_match_room_manager_closed_room")
    MatchRoomManager.connect("requested_character_speech", self, "_match_room_manager_requested_character_speech")
    MatchRoomManager.connect("restarted_game", self, "_match_room_manager_restarted_game")
    MatchRoomManager.connect("requested_different_teams", self, "_match_room_manager_requested_different_teams")
    MatchRoomManager.connect("requested_teammate_exchange", self, "_match_room_manager_requested_teammate_exchange")
    MatchRoomManager.connect("cancelled_teammate_exchange", self, "_match_room_manager_cancelled_teammate_exchange")
    DebugHelper.connect("requested_create_command", self, "_debug_helper_requested_create_command")
    DebugHelper.connect("requested_show_results", self, "_debug_helper_requested_show_results")
    DebugHelper.connect("requested_pause_round", self, "_debug_helper_requested_pause_round")
    DebugHelper.connect("requested_resume_round", self, "_debug_helper_requested_resume_round")
```

Figure 65: Connecting to signals of other nodes, this is Godot's form of the Observer pattern

The *_start_round* function sets the round number, updates the round label, plays the in-game loop track, and starts the timer (Figure 66). As seen on the topmost row in Figure 65, when the*game_timer* times out, a function is called, which will end the round.

```gdscript
func _start_round() -> void:
    var curr_round: int = prediction_results_overlay.curr_round + 1
    if curr_round == Settings.number_of_rounds:
        round_label.text = FINAL_ROUND_TEXT
    else:
        round_label.text = ROUND_TEXT_FORMAT % curr_round
    game_timer.start(Settings.round_duration)
    AudioController.play_music(AudioController.GAME_LOOP)
```

Figure 66: The *_start_round* function

This is what happens when the map screen scene is loaded. Every other object on on the map has a script attached to it that also tells it what to do when it loads and all of those nodes communicate by using signals.

B.3 Input Handling

Godot simplifies input handling by providing various classes for different types of events. Every node also has an *_input* function which is called when input is detected. The input event can then be checked for what type it is and from there the according action can be performed. Godot also supports touch input out of the box. For example, the pressed property of the input event that the *_input* function receives when it is called, is toggled both for clicking with the mouse and tapping on a touch screen. The functionality to consider mouse and touch input as the same can be toggled on and off in the project settings.

However, there is no direct input event for swiping, which was needed in the controller project. Since screen tap detection was possible, swipe detection was not hard to implement through it. When the screen is tapped initially a timer is started and the start tap position is remember, if the next input input event is not a tap (meaning the player stopped pressing the screen) and the timer has not stopped, a swipe is detected. If the distance between that start tap and end tap is bigger than a set threshold, the swipe is valid. If the timer stops before the player stops pressing on the screen, the swipe is also canceled. This allows players to cancel a movement they have started if they want to.

```
29 func _input(event: InputEvent) -> void:
30     if not is_visible_in_tree():
31         return
32     if not event is InputEventScreenTouch:
33         return
34     if event.pressed:
35         _start_detection(event.position)
36     elif not timer.is_stopped():
37         _end_detection(event.position)
```

Figure 67: The *_input* function for the swipe box in the controller project

Almost all of the other input in the game is through UI elements. UI nodes come with their own input event signals, which get emitted when that event is detected on the node. Any other node can connect to that signal if needed.

C Appendix - User & Maintenance Guide

C.1 Prerequisites

Have a version of Godot 3.5.1 download on your system. Godot 4 introduces a lot of reworks from the ground up and uses GDScript 2, which has significant differences from the previous version, especially when it comes to multiplayer networking. The project was developed in Godot 3.5.1, because Godot 4 did not have a stable version at the start of development and by the time it was officially released, development was nearing its end. Updating was considered but because of the numerous differences in systems from Godot 3.5 to 4, it was deemed risky to do near the deadline.

To download Godot 3.5.1, follow this link and the details provided there: `https://downloads.tuxfamily.org/godotengine/3.5.1/`

C.2 Project Download

The files for the project can be found at `https://github.com/krasi070/Critter-Elections`. Clone the repository or download it as a zip file. There are three Godot projects: Server Project, Game Project, and Controller Project. To open the projects in Godot, open Godot and import the *.godot* file for each of the three projects. Once imported, double click on a project in the Godot Project Manager to open it.

C.3 How to Run Locally

In the Game and Controller projects, go to *scripts/server/server.gd*. Make sure to uncomment the line that says it is used for testing.

```
37  	# Uncomment and use for production.
38  	#var url = "ws://%s:%d" % [SERVER_IP, PORT]
39  	# Uncomment and use for testing.
40  	var url = "ws://127.0.0.1:%d" % PORT
```

Figure 68: Server IP lines in *server.gd*

The default port is set to 9080. If you wish to change it, you can find the constant PORT at the top of *server.gd*. You will also have to change it in the Server Project, which also has a *scripts/server/server.gd* script.

To run a project in Godot, click the play button circled in red in Figure 69, which is located in the top right corner of the window.

Figure 69: Run project buttons in Godot

Run the Server Project first, then the Game Project, and finally the Controller Project. Everything should be set up now. However, you will not be able to play a game because you need at least four separate instances of the Controller Project to set up a full game. To run multiple instances, you can run the Controller Project through your browser by clicking the HTML5 button in Figure 69 instead of the play button. If there is no HTML5 button that means there are no HTML5 exports of the project presently. Exporting is covered in the beginning of Subsection C.4.2. Once the project has been exported the HTML5 button should appear. To run multiple instances, simply copy the link of from your browser and paste it in another tab. Repeat as many times as you want. Now you should be ready and set to play a full game, albeit not as intended with all the players using one device for the game and for all the controllers.

C.4 How to Run for Production

To experience the game properly, you will need a server host for the Server Project and a web host for the Controller Project. I used two EC2 instances in Amazon Web Services, one for the server and one for the website. The following instructions assume you will use Linux machines to host the server and website. Instructions on how to use AWS will not be provided, only on how to set up the Linux machines to run the projects.

For the communication between the server, controllers, and game to work, the hosts need to be left to use *ws* not *wss*. The communication will be blocked if one of them uses a secure connection and the others do not. It should be possible to set up for all of them to work with a secure connection with an officially signed SSL certificate but that proved difficult to do and I did not manage to set it up, so a secure set up will not be covered here. It should be noted that a self-signed certificate alone will also prove insufficient and the communication will be blocked. That being said, no sensitive data is handled anywhere in the projects so it should be fine even when using an insecure connection.

C.4.1 Setting up the Server Project

First, you will need to export it. Open the Server Project in Godot, go to the Project option in the menu bar and click *Export...* as shown in Figure 70.

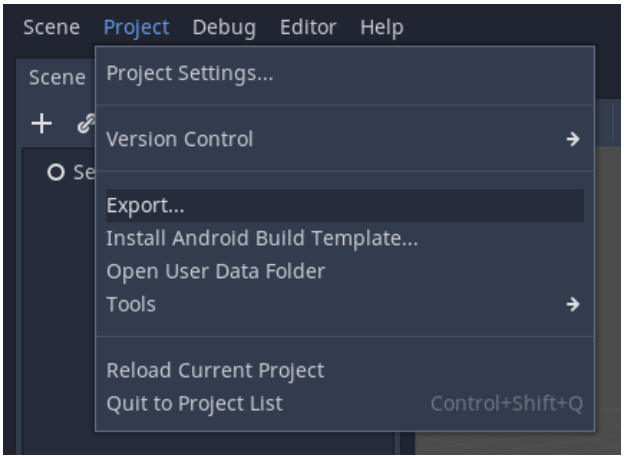

Figure 70: Export option in Godot

If you do not have the required export presets, Godot will prompt you to install them. Select the Linux export preset, if it is not present you can add it by clicking *Add...* as seen in Figure 71.

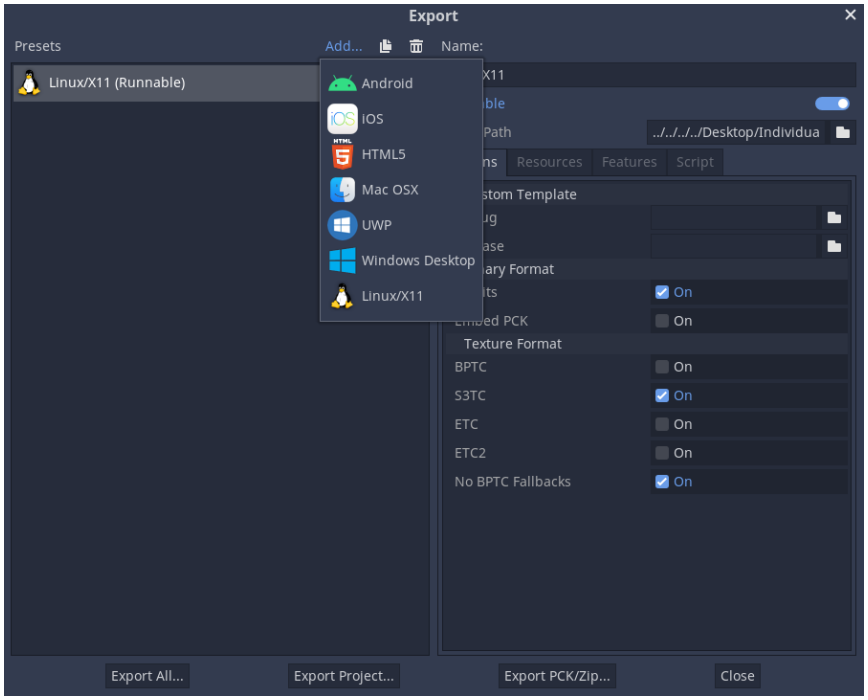

Figure 71: Godot's export window

Leave everything to the default settings. Click the *Export Project...* button, and select a folder

where you want to save the files. Make sure to uncheck the *Export with Debug* when selecting a folder. You should now have two files: a *.pck* file and *.x86_64* file.

Move the downloaded files to your server Linux machine. Install Godot on it with this command:

```
sudo yum install wget && sudo yum install unzip && wget
https://downloads.tuxfamily.org/godotengine/3.5.1/Godot_v3.5.1-
stable_linux_server.64.zip && unzip Godot_v3.5.1stable_linux_server.64.zip
```

Go to the directory where you placed the export files and use this command to run the server:

```
./Godot_v3.5.1-stable_linux_server.64 --main-pack your_server_name.pck
```

You can stop it by pressing Ctrl + C.

C.4.2 Setting up the Controller Project

Open the Controller Project in Godot. In *scripts/server/server.gd* make sure to use the line for production. Set the *SERVER_IP* constant to the IP or DNS of your server as well. To make sure the project connects to the server, run it and check if you see the *"Connected to server"* message. If all that is working, proceed to exporting the project as shown in Figure 21 and Figure 22, but this time with the HTML5 export preset.

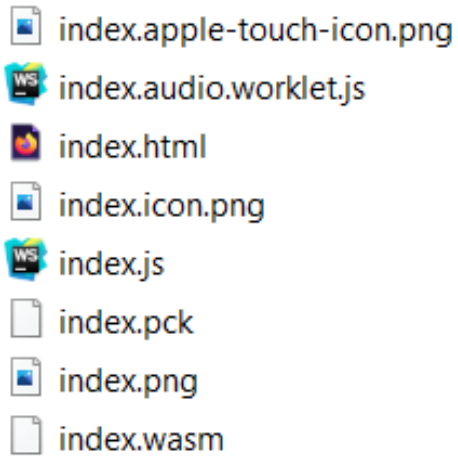

Figure 72: The files you should have after the export

Depending on the host provider you are using, moving all the files to where your *index.html* file of the website will be should be enough. If you need to set up a Linux machine to run the web server, here are the commands you need to run.

Make sure everything is up to date:

```
sudo su
yum update -y
```

Install and start the Apache Web Server:

```
yum install -y httpd
systemctl start httpd.service
systemctl enable httpd.service
```

Move all of the export files to */var/www/html/* in the Linux Machine. That should be all for the web server to be running.

C.4.3 Setting up the Game Project

Open the Game Project in Godot. In *scripts/server/server.gd* make sure to use the line for production. Set the *SERVER_IP* constant to the IP or DNS of your server as well.

Optionally, open the *RoomCreationScreen* scene, located at *scenes/screens/RoomCreationScreen.tscn*. Change the URL link label to your web server's IP or DNS and create a new QR code for it.

D Appendix - User Evaluation

In order to evaluate the project, a survey was designed using Google Forms. Players were separated into test groups of four people. Each test group played two sessions of the game in my presence. Once the second session finished, the players were asked to complete the survey.

D.1 Ethical Approval Form

Ethics application has been approved

W www-data@cis.strath.ac.uk <www-data@cis.strath.ac.uk>

13.2.2023 r. 21:54

To: Krasimir Balchev (Student)

Hello,

Your ethics application "Online team building game using smart devices as controllers" (ID: 2112) has been approved.

URL: https://local.cis.strath.ac.uk/wp/extras/ethics/index.php?view=2112

CIS Ethics Approval System.

Figure 73: The approval email for the ethics application

Tile of research:

Online team building game using smart devices as controllers

Summary of research (short overview of the background and aims of this study):

The project's aim is to develop an online multiplayer team-based video game that will make it easier to break the ice between members in a newly formed team. To increase accessibility, instead of requiring dedicated controllers or keyboards for every player, players connect to the game through a website from which they can give inputs and control what they do in the game. The aim of this study is to get feedback on how players feel about the game and whether it had an effect on how they perceive their teammates after playing the game. I wish to conduct both online only interviews and fully in-person interviews to see if there is a noticeable difference between the participants' answers to the survey depending on whether they participated online through a service like Zoom or Discord, or participated in-person.

How will participants be recruited?

I will ask students from the university that I know whether they wish to be a part of the study. I will also ask friends from outside of the university whether they wish to participate as well.

How will consent be demonstrated?

The consent form will be included in a Google Form, that is why it says "By pressing Next you agree to the following".

What will participants be expected to do?

Survey link: `https://docs.google.com/forms/d/e/1FAIpQLSdXNd5xet22j-HzNsUgBcWdIWfvxJbIdWL8wJm9G8JSfkQxng/viewform`

All of the required information I have submitted for this application was reviewed and approved by my supervisor first.

What data will be collected and how will it be captured and stored?

The game does not collect any personal data and once a game session is over all of the data for that session is deleted. The only data collected will be the survey responses. The survey does not ask for any personal information, meaning only anonymous data will be collected, which does not fall within the scope of GDPR.

How will the data be processed?

The data will be exported from Google Forms to a .csv file and will be visualised in various graphs.

How and when will data be disposed of?

Once all of the survey responses have been collected through Google Forms, the data will be exported to a password-protected Excel file and deleted from Google Forms. The file will be stored only on the university's server and disposed by the end of September 2023.

D.2 Participant Information Sheet

Name of department: Computer and Information Sciences

Title of the study: Online team building game using smart devices as controllers

Introduction

My name is Krasimir Krasimirov Balchev a 5th-year Software Engineering student at the University of Strathclyde, Glasgow. I am developing an online multiplayer team- based video game that aims to make it easier to break the ice between members in a newly formed team.

What is the purpose of this research?

The purpose of this research is to create and evaluate an online multiplayer team-based video game. The gathered feedback will be used to resolve issues, possibly introduce new features, and assess whether the project requirements have been satisfied.

Do you have to take part?

Participation is completely voluntary and participants are free to withdraw from the project at any time without having to give a reason and without any consequences. In-person and online participation is possible.

What will you do in the project?

As part of this study, you are being asked to play two sessions of the game. More detailed instructions will be given before the start of each session. You are free to withdraw at any time. Once both play sessions are complete, you will be asked to complete a survey.

What hardware do you need to participate?

Any smartphone, tablet or computer/laptop that has access to the Internet and can visit websites through a browser should be sufficient.

How long will the study take?

Roughly about 45-60 minutes.

What information is being collected in the project?

The video game itself does not collect any data whatsoever. The only data being collected is the survey responses which are entirely anonymous and require no personal information.

Who will have access to the information?

The information gathered will be used solely for the purpose of evaluating this project. The data is available only to the survey owner – Krasimir Krasimirov Balchev.

Where will the information be stored and how long will it be kept for?

All anonymous data from the survey will be exported as a password -protected Excel file and uploaded to the University of Strathclyde's servers. At most, the data will be kept by September 2023.

Thank you for reading this information – please ask any questions if you are unsure about what is written here.

Student Contact Details:

Krasimir Krasimirov Balchev

krasimir.balchev.2018@uni.strath.ac.uk

Supervisor Contact Details:

Dr Fredrik Nordvall Forsberg

fredrik.nordvall-forsberg@strath.ac.uk

Departmental Ethics Committee Details:

Secretary to the Departmental Ethics Committee

Department of Computer and Information Sciences

Livingstone Tower

26 Richmond Street

Glasgow

G1 1XH

ethics@cis.strath.ac.uk

D.3 Consent Form

Name of department: Computer and Information Sciences

Title of the study: Online team building game using smart devices as controllers

Student Contact Details:

Krasimir Krasimirov Balchev

krasimir.balchev.2018@uni.strath.ac.uk

By pressing *Next* you agree to the following:

- I have read and understood the Participant Information Sheet for the above project and the student has answered any queries to my satisfaction.
- I understand how the anonymous information I provide will be used and what will happen to it (i.e. how it will be stored and for how long).
- I understand that my participation is voluntary and that I am free to withdraw from the project at any time without having to give a reason and without any consequences.
- I understand that I do not have to answer every question in the survey and may only answer those I feel comfortable doing so, if any.
- I understand that anonymised data (i.e. data that does not identify me personally) cannot be withdrawn once it has been included in the study.
- I consent to being a participant in the project.

D.4 Instructions Script

Hello, my name is Krasimir Balchev. As I mentioned when I asked for your participation, I am developing an online team building video game as part of my university project. Any feedback you give is appreciated and will be used to improve the project in the future and check whether the project meets its goals.

I will first ask you to go to the survey I have shared with you and read through the participation information and decide whether you still want to participate. Once you are done, if you still wish to participate, you will be asked to play two game sessions of the game. During the first session, you will play the game without my aid, meaning I will leave all the rules explanation to the game itself. This is done to simulate how a group of people would learn how to play the game in a normal setting without the developer being present. Of course, you are free to communicate amongst yourselves and explain things to one another if you wish. You can also pick whatever team and role you want

for this first game session. Once the session is done, a second one will begin. This time you are free to interact with me and ask me questions you might have while you play. During the second session, you will also be asked to choose a different teammate from the first game and a different role from your originally chosen one. Once the second session is done, you will return to the survey and fill it out. That will be the last thing asked of you. Any questions?

[Answer any questions asked]

[Wait for the participants to read the participation information and consent form]

[Open the game and create a room for players to join]

If everyone here has read the participation information and has agreed to the consent form, please go to the link that will be shown on the screen. From there you can join the game and read the "how to play" rules. The participant who first joins the game through the website has the power to start the game when everyone is ready. You are free to start when ready.

[Observe the first play session]

Now, I will go back to the start game screen. From there you can follow the link shown and join the game as you did during the first game. I will ask you to choose a different teammate for this session and a different role in the game as well. You are also free to ask me any questions about the game during this session. Feel free to start when everyone is ready.

[Observe second session and answer any questions asked]

Thank you for playing! Lastly, I will ask you to go back to the survey link and fill out the post game survey section.

[Wait for the participants to fill out the survey]

That's it! Thank you for participating!

D.5 Survey Link

Link to survey: `https://forms.gle/VyzrQLomCMU46Sic9`

D.6 Survey Questions

Question 0 is about whether you participated in-person or online.

Answer questions 1 - 9 on a seven point scale in which the closer you are to 1 the more you disagree with the statement above and the closer you are to 7 the more you agree with it.

Question 10 is an optional open-ended question, in which you can elaborate in detail on questions 1 - 9.

Question 11 is also an optional open-ended question, in which you can give feedback on any kind of improvement you would like to see in the game.

0. Did you play in-person with the other participants or online through a service (Zoom, Discord, etc.)?
1. The game was fun to play.
2. The game was stressful to play.
3. The game rules were easy to understand.
4. The game was challenging.
5. Good teamwork was required to excel at the game.
6. I felt closer to my teammates after playing the game compared to before I started.
7. I trusted my teammates in the game.
8. I gained a newfound appreciation of my first teammate's role after playing as that role during the second game.
9. I would play the game again.
10. Is there anything you would like to elaborate on regarding your answers to questions 1 - 9?
11. Is there anything you think could be improved upon the game? (instructions, gameplay, readability, etc.). If so, what do you think can be done to improve it?

D.7 User Evaluation Results

Questions 1 - 9 used a 7-point Likert scale. 1 represents strongly disagree while 7 represents strongly agree.

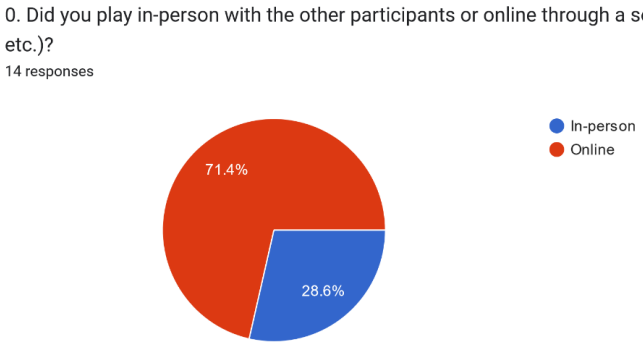

Figure 74: Survey question 0

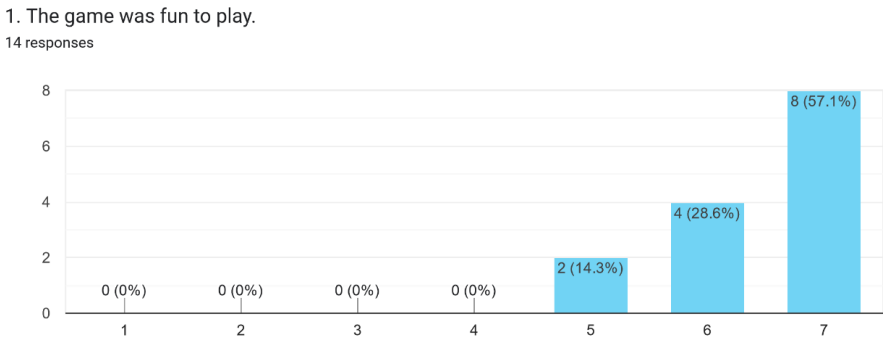

Figure 75: Survey question 1

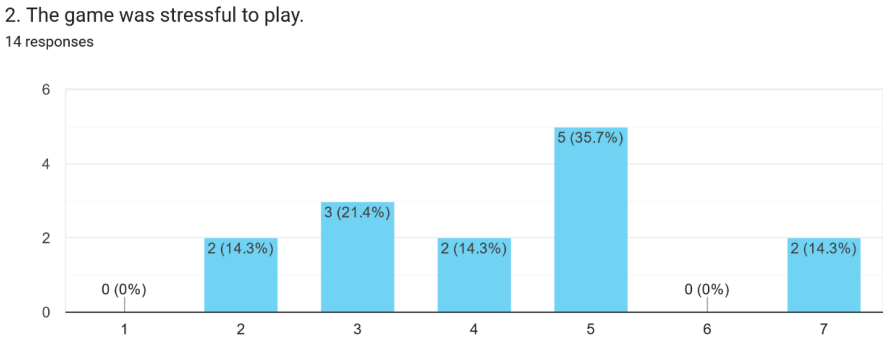

Figure 76: Survey question 2

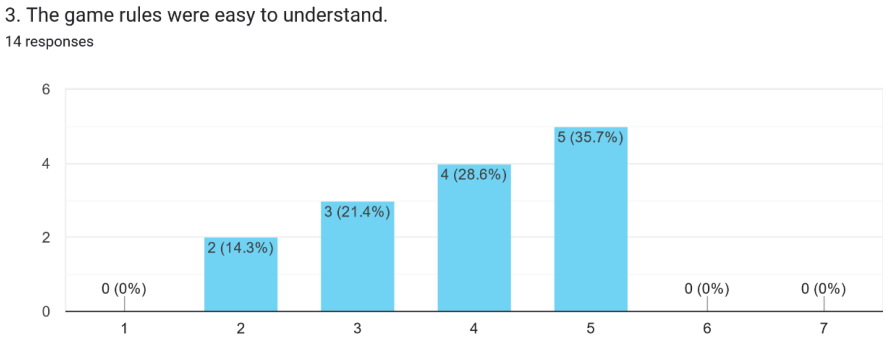

Figure 77: Survey question 3

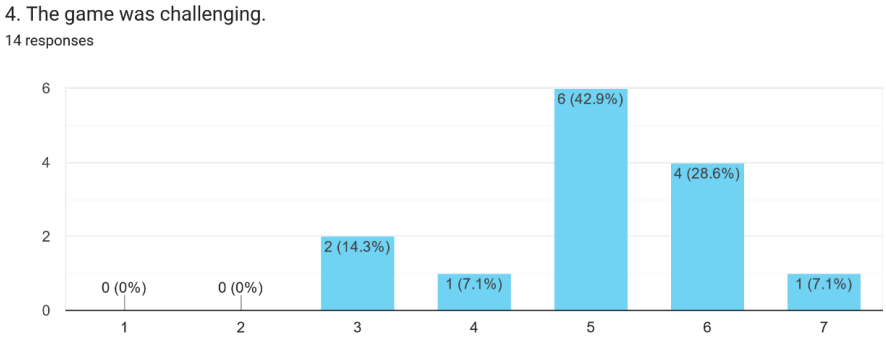

Figure 78: Survey question 4

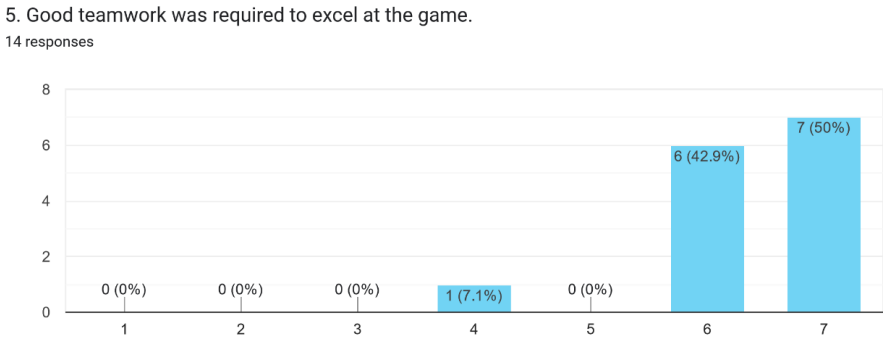

Figure 79: Survey question 5

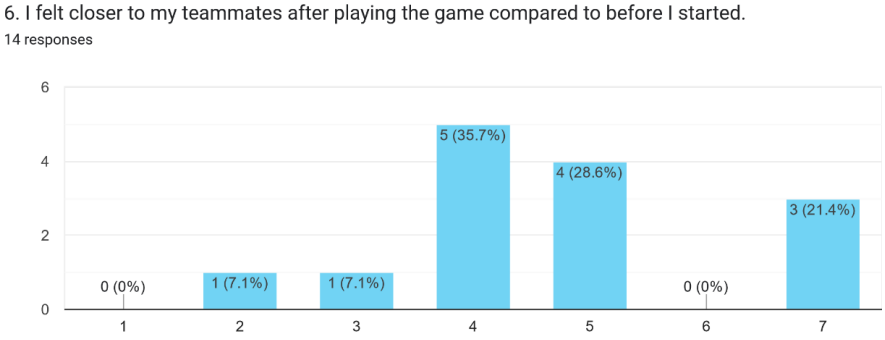

Figure 80: Survey question 6

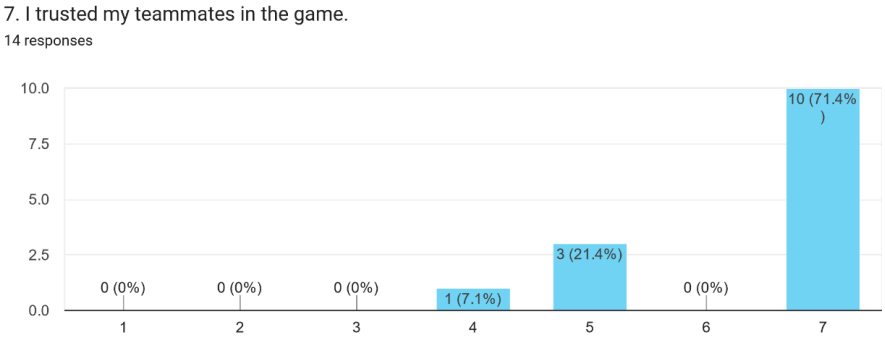

Figure 81: Survey question 7

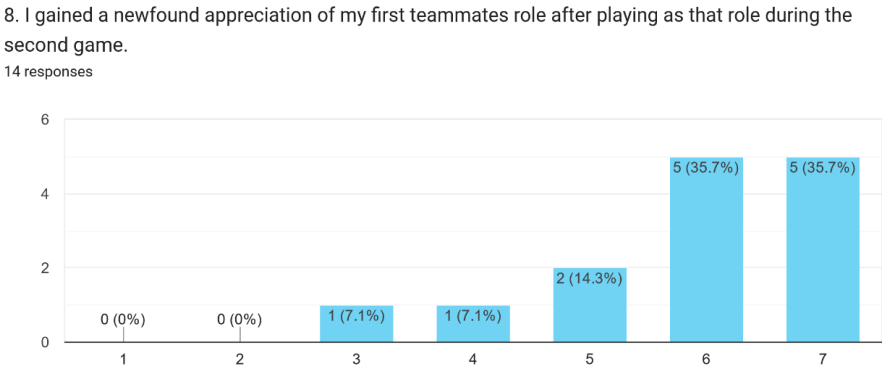

Figure 82: Survey question 8

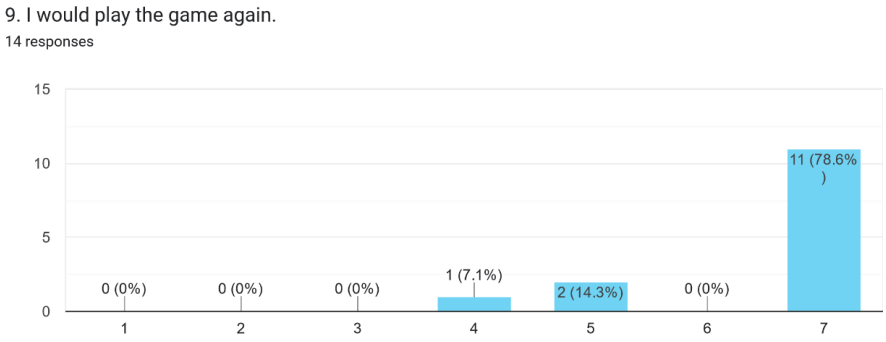

Figure 83: Survey question 9

Is there anything you would like to elaborate on regarding your answers to questions 1 – 9?

- very stressful
- Not its all good.
- I was playing on my laptop and there was a delay in the response time when i used the keyboard to play.
- N/A
- It was a really nice game. Lovely game!
- not sure how clear to understand were the rules tbh but i overall enjoyed the game a lot! ¡3
- 6. i can't exactly answer this properly bc i've been best friends w all of the players for years lol
- 10/10 IGN
- The game felt challenging in a good way and required good teamwork to excel. I think after the first play through of each role it becomes very clear how to play the game.

Is there anything you think could be improved upon the game? (instructions, gameplay, readability, etc.) If so, what do you think can be done to improve it?

- the buttons should be a bit bigger for phone
- the rules should be clearer, pretty much the only fault i have with the game, it took until halfway through the second game to fully understand. The buttons on the phone screen could

also be a little larger but thats not the end of the world. otherwise I really had fantastic time. i want to show this to my family and play it with them.

- Some things can be explained better, f.ex. the fact that you do not need to be standing on thegathering spots while you are gathering resources
- Maybe a short tutorial at the beginning of the game to make sure everyone understood the instructions right
- Maybe the option of changing the role between parts.
- Definitely readability. I have glasses and even though the paragraphs were short, it'd be nice if the formating could be done differently? Or separate the role of the candidate and manager on different pages because that was confusing.
- maybe clearer rules/explanations about some things like the little snails and the beehives
- a little description of the icons would be good. i kind of forgot that there is a button for the rules lol bc i'm a dumb dumb and so an option to tap on the lil icons to get a quick description as to what they are would help people like me also i preferred tapping/clicking on the arrow keys rather than the keyboard or swiping. additionally, it was much comfier playing on my phone than on my laptop, though that might've been bc of trying to watch the stream to see where i was going and having the browser tab open at the same time
- Countdown or audio cue for how much time is left
- Slide bar for deals more visible.
- It's not immediately obvious when a button has appeared on your phone.