

University of Bristol



University of BRISTOL

Faculty of Engineering: Computer Science

Development Report

Asteroid Zone

By Team I



Development Team:

David Nzewi (Team Manager)

Oliver Pope (Lead Programmer)

Omer Karaman (Lead Designer)

Milen Georgiev

Jordan Matthews

Jarrold Doyle

1. Contributions

1.1. Critical Notes:

Omer had extenuating circumstances in the last 2 weeks of the project which prevented him from contributing. The contribution weightings have been adjusted accordingly to reflect this.

1.2. Agreed Weightings:

David	Oliver	Omer	Milen	Jordan	Jarrold
0.99	1.04	0.95	1.04	0.98	1

1.3. Team Signatures:

David Nzewi (Team Manager):



Oliver Pope (Lead Programmer):



Omer Karaman (Lead Designer):



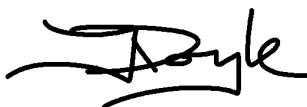
Milen Georgiev:



Jordan Matthews:



Jarrold Doyle:



2. Major Achievements

2.1. Presentation Video:

<https://youtu.be/n0k2hH4i5t0>

2.2. Five Key Achievements:

We created an entirely voice-controlled game, which works on all major browsers (using WebSockets).

We created a high-quality Voice Chat using WebRTC and SignalR, which is not based on any pre-existing Unity Library.

We implemented a PhotonNetwork multiplayer lobby.

We wrote over 12,000 lines of code (for Unity Scripts and ASP.NET Core Server).

We created all of the 3D models within the game.

3. Nine Aspects

3.1. Team Process

- We met weekly to playtest the game and discuss what each team member should work on (section 5.1 and 7.4)
- We used multiple text channels in discord to organise our communication (section 5.1)
- We split into subteam pairs to specialise in different aspects of the project (section 5.2)
- We used GitHub issue boards to keep an organised list of bugs and new features (section 5.2)
- We created and assigned tags to the issues to help us with prioritisation (section 5.2)
- We assigned ourselves to issues so that we knew which problems were being worked on (section 5.2)

3.2. Technical Understanding

- We researched and prototyped multiple voice recognition methods such as Unity's PhraseRecognitionSystem, Kaldi ASR, and the WebSpeech API (section 8.1)
- We researched multiple voice chat systems such as RecordRTC with WebSockets, and WebRTC with SignalR (section 8.2)

3.3. Flagship Technologies

- We created an entirely voice-controlled game for all browsers using WebSockets (section 8.1)
- We created 2 working voice chat systems for our game. One with WebSockets and RecordRTC and one using WebRTC and SignalR that is used in the final game (section 8.2)

3.4. Implementation & Software

- We developed a grammar system to support our voice recognition allowing for synonyms or corrections for words that are commonly misheard (section 8.1.3)
- We developed a dynamically updating pathfinding system using Unity NavMeshes so that moving ships/pirates can navigate around spawning asteroids (as can be experienced in the demo)
- We built a smooth tactical camera to allow the Station commander to view the whole map (section 8.4)
- We created a “fog of war” system for miners so that effective communication between crew members is required to win the game (section 8.5)

3.5. Tools, Development & Testing

- We used assertions to test the command recognition system (section 7.3)
- We used git flow to easily create and merge feature branches (section 7.6)
- We made an extensive manual testing checklist to ensure rigorous testing where automatic testing was impossible (section 7.4)

3.6. Game Playability

- We created engaging and natural to use voice controls (as can be experienced during the demo)
- We created "smart actions" which will perform additional commands depending on the context to reduce the amount and complexity of required voice commands from the player. For example "shoot pirates" also deactivates the mining laser and targets an enemy before shooting (as can be experienced during the demo)

3.7. Look & Feel

- We created all of the 3D models in the game using Maya with an appropriate level of detail (as can be experienced during the demo)
- We created all of the textures for our game with the exception of the SkyBoxes (as can be experienced during the demo)
- We chose a strong "brand" colour to use for a consistent UI (as can be experienced during the demo)
- We used a futuristic font to match the space setting and high tech voice-controlled ship systems (as can be experienced during the demo)

3.8. Novelty & Uniqueness

- We created a game that is entirely controlled using voice commands (as can be experienced during the demo)
- Having the station commander give commands and coordinate real players is unlike other games where these players would normally be AI (see video)
- All of the game constants are fully customisable allowing the user to change the difficulty of the game (see video)

3.9. Report & Documentation

- We documented all of our server-side and client-side code in NAME style (see code)
- We included additional comments for complex code to aid understanding (see code)
- We organised our project directory structures logically (see code)
- We created high-quality report diagrams using draw.io (section 8)

4. Game Description

4.1. Overview

Asteroid Zone is a 2-4 player cooperative multiplayer game, controlled entirely using voice commands, that runs in a web browser.

The players are the crew of a damaged space station that has drifted into a dangerous region of space. Players need to mine asteroids to gather resources, repair the space station and use the hyperdrive to escape to a safer region of space and win the game. They will also need to fight off space pirates that will try to destroy the station if they find it. However, there is a pirate mothership on its way which the players will not be able to defend against when it arrives. The game is lost if either the pirates manage to destroy the station, or the players run out of time and the pirate mothership arrives. At the end of the game, players are given a score based on their actions during the game, such as the amount of resources they gathered and the number of pirates they destroyed.

4.2. Player Roles

The players are split into 2 roles. One player is the station commander and the rest are miners.

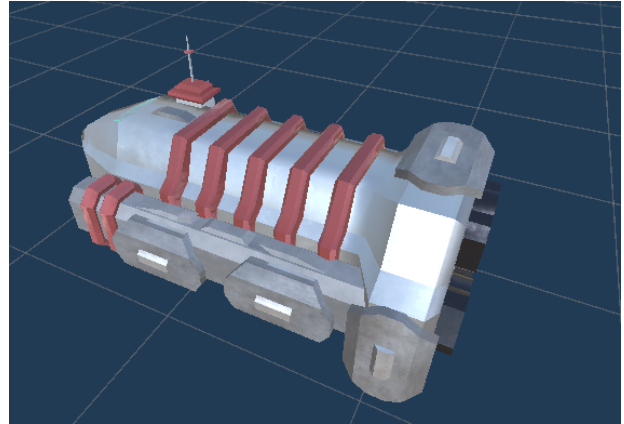
4.2.1. Station Commander

The station commander has a more tactical role in the game. They have a complete view of the entire game so a large part of their role is to give the other players information. They need to warn players about nearby pirates, advise players on where to find the most asteroids and tell players which role to perform. For example, they might tell one player to stay near the station for defence, or if there aren't any pirates around they might tell all the players to mine asteroids. They can use pings to mark a location on the player's minimaps or they can use voice chat to communicate with the other players.

Another major part of the station commander's role is to decide how to use the gathered resources. They can be used to respawn dead players, however, the cost of respawning a player increases each time. They can also be used to repair modules of the space station. Each module has a different function and the station commander has to decide which modules to prioritise. For example, the shield generator will create a defensive shield around the space station, if the hull health reaches 0 the space station will be destroyed, and the hyperdrive allows the players to win the game.

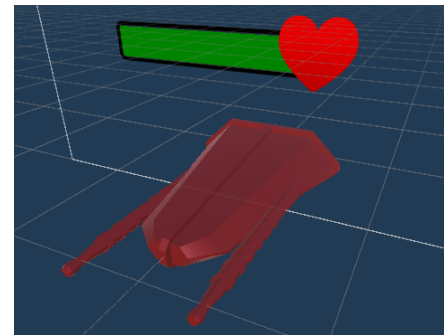
4.2.2. Miner

Miners pilot remote-controlled mining ships to fight pirates and mine asteroids. These ships only have a limited radar range so the miners only have a small radius of visibility, because of this they rely heavily on information provided by the station commander. Miners are responsible for mining asteroids and delivering the resources back to the space station. They also need to defend themselves and the space station from the pirates. If the pirates discover the station's location they will call for stronger reinforcements, so it is better if the miners can destroy them before they get near the station. As the miners perform these tasks they will level up, they have a combat level and a mining level. Each time a player levels up they get better at that task. For example, if a player increases their combat level, their damage will increase as well as their laser range. If a player increases their mining level, their mining speed and range will increase.



4.3. Pirates

There are 2 types of pirates in the game, scouts and elites. Scout pirates have less health and do less damage than elite pirates, however, they move faster. Scouts have a random chance of spawning every few seconds and search the region randomly. If they find a player they will follow and attack them. If they find the space station they will alert all the other pirates and some elite pirates will spawn. When alert, pirates will try to attack and destroy the space station. Some pirates will prioritise destroying the space station and will ignore players, others will prioritise attacking players before the station. If all pirates are destroyed they will become unalerted and any new scout pirates will start searching randomly again.



5. Team Process and Project Planning

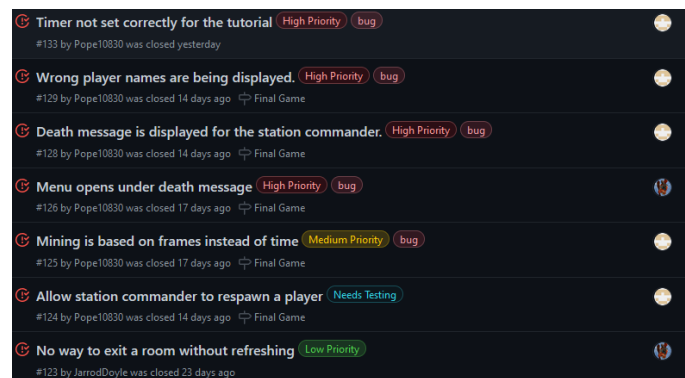
5.1. Communication & Planning

We decided to use Discord as our team communication platform as this allowed us to create separate text channels for different aspects of the project in order to communicate effectively and professionally. Initially, we created a channel for each of us to pitch our game ideas, by using discord we were able to then archive the ideas that were not chosen. We used a general channel for organising team meetings and general project discussions. During development, we made three channels for each of the major sections of the project, voice control, asset design and multiplayer. These were used for posting any bugs or errors that were discovered when testing the game, asking team members for help if any difficulties were encountered during development and discussing any additional features we wanted to implement. The final channel we created was used with a GitHub Bot that posted any updates made to the GitHub organisation.

We had weekly team meetings at the beginning of each week to decide what each person would be working on that week and if necessary organise pairs to work on more difficult tasks. We also used these weekly meetings as an opportunity to update the GitHub issue board with any new features we wanted to implement and to assign priority tags to the issues.

5.2. Project Management

We organised the team into three subteam pairs to make sure all team members were working on aspects that played to their strengths. While team members did cross over and work on different aspects of the project, splitting into these teams allowed team members to specialise and become more familiar with one aspect of the project.



We used the GitHub issue boards to keep an organised list of features to add and bugs that needed to be resolved. This allowed us to assign tags that we used to prioritise certain issues. We made sure to assign ourselves to an issue when we were working on it to prevent multiple team members from working on the same issue without telling each other.

We also used the milestones feature on GitHub to help us with time management. We assigned milestones such as “Easter Review” or “Final Game Release” to issues so we could keep track of which issues needed to be resolved first.

5.3. Time Management

Overall we managed our time well as we were able to get the game to a good standard considering the total time available to us. This is because of our meetings at the beginning of each week which allowed each member to plan out their schedule to accommodate the time taken to complete assigned tasks. If a task was taking longer than expected then this was communicated ahead of schedule so other team members or TAs could provide assistance. Employing these practices prevented any major issues halting the development process.

5.4. Evaluation

As a team we worked effectively to complete this project, mainly due to the consistent communication that we kept up throughout the project, however, if we were to start again we think it would be beneficial to schedule meetings at the end of each week as well as at the start of the week, especially in the latter stages of the project. Having an additional meeting would have allowed us to discuss any issues we had that week so we could start the next week knowing exactly what needed to be done. Furthermore, we think that using a Gantt chart would have helped aid project planning especially at the start of the development stage before we settled into our routine as it would have been easier to communicate information regarding schedules.

6. Individual Contributions

6.1. David Nzewi (Team Manager)

- Management [Throughout whole project]
 - Scheduled weekly core team meetings
 - Scheduled weekly meetings with TA
 - Had weekly check-ins with composers.
- Set up the multiplayer system [1 week]
 - Researched possible networking systems.
 - Added preliminary UI for main menu allowing players to enter their name and have it stored after restarting the game.
 - Used Photon Unity Network to allow different clients to connect the same room with their own unique avatar.
 - Added button allowing players to leave the room they are in.
- Player and Pirate Ship Synchronisation [1 day]
 - Synchronised transform position and rotation of each ship across the network.
 - Synchronised velocity and angular velocity of each player ship across the network.
- Space station synchronisation [3 days]
 - Synchronised adding of resources to the space station across the network.
 - Synchronised damage taken by the space station across the network.
- Combat synchronisation [1 day]
 - Synchronised laser projectiles for each player across the network.
 - Synchronised damage taken by pirates and players across the network.
 - Synchronised destruction of pirate game objects after being defeated by a player.
- Mining synchronisation [1 day]
 - Synchronised mining laser for each player across the network.
 - Synchronised destruction of asteroid game objects after being mined by a player.
 - Synchronised transferring mined resources to the space station.
- Testing [3 days]
 - Conducted manual testing on multiplayer features implemented
- Bug fixes concerning multiplayer [Throughout whole project]
- Report Documentation: sections 6.1, 8.3.1, 5.3, 5.4 [3 days]

6.2. Oliver Pope (Lead Programmer)

- Created the space station model [2 days]
 - Created spinning animation
 - Created damaged and functional textures for each module and make sure the correct texture is displayed on the model based on its health
- Speech recognition (Text to actions) [1 week and continuous improvement]
 - Created a grammar to convert the input phrases to Command objects
 - Created a system to predict commands for invalid input phrases
- Improved pirate AI [2 days]
 - Randomly searching for the space station
 - Alerting other pirates if the station is found and spawning reinforcements
- Added miner and station commander voice controls [1 week and continuous improvement]
 - Miner movement, resource transfers, shooting and mining
 - Commander pinging, repairing, respawning and module activation
- Implemented role separation [1 day]
 - Created the station commander prefab
 - Instantiate the correct prefab for each role and load the correct camera
 - Only allow players to perform actions valid for their role
- Added asteroid mining [2 days]
 - Check for asteroids in range using Physics.Raycast
 - Reduce the asteroid size and fade out if it's destroyed
 - Synchronised increasing players resources
- Implemented player shooting [2 days]
 - Created LaserProjectile prefab and instantiate it when a player shoots
 - Added collision logic
- Created the space station with modules [2 days]
 - Created base StationModule class and subclasses for specific module behaviour
 - Added module damage and repairing
- Added gameplay stats (asteroids mined, pirates destroyed, etc) and a leaderboard at the end of the game with points based on player stats [2 days]
- Implemented a fog of war system [1 day]
- Added quests for the station commander and additional miner quests [1 day]
- Created a menu for the host to edit game constants (player speed, pirate health, etc) [1 day]
- Colour coded player ships so they can be identified [1 day]
- Added game timer so the game ends after a certain time limit [1 day]
- Implemented player dying and respawning with multiplayer synchronisation [1 day]
- Added player levels for combat and mining [1 day]
- Unity code documentation [3 days]
- Implemented interactive tutorial with subtitles using coroutines [2 days]
- Overall bug fixing of the game [Throughout whole project]
- Report Documentation: sections 3, 4, 5.1, 5.2, 5.4, 6.2, 8.1.3, 8.5, 8.6 [1 week]

6.3. Omer Karaman (Lead Designer)

- Implemented grid-based asteroid spawning [2 days]
 - Asteroid spawn attempts are made every X seconds with probability Y as defined in the inspector
 - Spawn location is random within the sector grid width/height
- Improved tactical camera panning [1 day]
- Implemented a lock-on system [2 days]
 - Allows locking on to pirates and asteroids
 - Targets the nearest enemy
 - Created a lock-on indicator using a world space Unity Canvas
- Created asteroids [2 days]
 - Used Maya to create a detailed model
 - Made detailed asteroid textures implemented with Unity's material system
- Created and implemented various game sound effects [2 days]
 - Combat laser SFX
 - Initial mining laser SFX
- Expanded the voice recognition grammar [Throughout whole project]
- Worked on the player ship [3 days]
 - Created the base colour textures
 - Added ship thrusters using Unity's particle system
 - Made the ship thrusters only activate while the ship is moving
- Implemented a player quest system [3 days]
 - Provides gameplay hints to the player so they have an idea of what to do
 - Dynamically updates based on current game state e.g. defend the station if pirates are attacking
- Created the initial fog-of-war system (later updated by Oliver) [4 days]
- Worked on Main Menu and In-Game UI [1 day]

6.4. Milen Georgiev

- Server-side code for hosting the website using ASP.NET Core 5.0 Razor Pages and all of the web pages on the website [1 day]
- DevOps process: initialised server-side GitHub repository, created hosting environment process and connected the enabled automatic deployment of the repository to the cloud hosting website (Heroku), sorted out Website SSL certificates [2 days for initial structure and continuous maintenance]
- Enforcing HTTPS connection on server-side [1 day]
- Initial Main Menu Scene used to start, exit and overall navigate the game [1 day]
- Credits Scene with its animations [1 day]
- Pirate spawning and Pirate AI: randomly navigating through the map, following players and pointing at them and Pirate UI Health Bar (health bar always pointing at player's camera) and dying logic [2-3 days]
- Created the game minimap [2 days and continuous improvement]:
 - Game objects staying above players, pirates and space station in a separate layer, so that the position of objects is apparent
 - Ping game objects appearing on the map to inform players when the station commander shows them where to mine and where enemies can be found
- In-Game Menu (settings) with Volume controls and Tips [2 days]
- Voice chat communication between players [1 week]:
 - An initial version using RecordRTC library and WebSockets (bad voice quality)
 - final (production) version using WebRTC with .NET SignalR for client-to-server signalling ('perfect' voice quality)
 - In-game control of the voice chat, i.e. start/stop logic and switch from command to chat mode
- NavMesh, used for smoothly navigating objects through the map without bumping into obstacles (asteroids, players, pirates, the space station etc.) [3 days]
- Player's ship AI navigating through the map without bumping into objects using the NavMesh [1 day]
- Server-side code documentation [2 days]
- Game event messages panel (bottom left panel inside of the game showing information about events in the game) [1 day]
- Website front-end design [2 days]
- Speech recognition (audio input to text) [3 weeks]:
 - Non-Chrome Browser version with WebSockets and Google Cloud Speech Recognition System
 - Worked on Chrome version as well (bug fixing and improving)
 - Created SpeechRecognitionConsoleTest: application for testing Speech Recognition — used to research the C# Google Speech API
- Cockpit camera zoom in/out on mouse scroll
- Implemented asteroids NavMeshObstacle (and multiplayer synchronisation) so that AI travelling players and pirates do not bump into them [2 days]
- Overall bug fixing of the game [throughout whole project]
- Report Documentation: sections 6.4, 8.1.1, 8.1.2, 8.2 [1 week]

6.5. Jordan Matthews

- Created the Lobby system (Initial + Deployed):
 - Initial version that allowed to join specific games made by other players
 - Improved upon initial version to allow users to name their lobbies
 - Set-up a panel system that automatically updates to show all lobby listings available to the player
 - Added logic to prevent players joining lobby's that shouldn't be joined due to player limitations or lobby closure
 - Created PlayerListing prefab so other users can see the names of the other players in their lobby
- Researched alternative speech recognition methods (Kaldi ASR):
 - Installed Kaldi and setup an environment to trial the speech recognition
 - Installed example training set for testing
 - Read scientific papers on speech recognition to gain a better understanding for speech recognition programs
- Researched testing frameworks:
 - Researched Trillion a beta testing framework
 - Researched and attempted to implement unity test framework
 - Helped build manual unit tests using asserts where appropriate
- General bug testing at every meeting
- Created UI for Lobby and Room scenes
- Reduced loading buffers across scenes by having server connections occur at smarter timings.
- Used LaTeX to format the whole report for final submission

6.6. Jarrod Doyle

- Defined an initial basic team coding style [1 day]
- Grid setup [2 days]
 - Generate a grid of squares at run time based on dimensions defined in the room settings
- Tactical camera system [1 week]
 - Moving, pivoting, and zooming based on a child focus point GameObject
 - Selecting a specific object to focus using a raycast and adjusting the camera accordingly
 - Positional and zoom constraints so the station commander cannot move the camera too far
- Created the initial mining ship and pirate models [1 week]
 - Further developed the mining ship and pirate models
 - Ensured mining/combat lasers are shot from the correct position on the model
- End game win/lose animation [3 days]
 - Used Unity's animator to create the animations
 - Used Unity's particle effect system to create an explosion
- Redesigned the games UI [1 week]
 - Initial changes improved Visual clarity by reducing the number of stacking transparent panels
 - Used Unity's layout group systems to automatically and consistently layout UI elements
 - Improved visual consistency and clarity by introducing an improved colour palette and adjusting UI element scales
 - Added additional details to the room and player listings such as the number of players in a room and the player roles
- Tutorial Voice acting [2 days]
 - Recorded voice lines based on a provided script
 - Edited voice lines into separate audio files
 - Used Audacity to edit lines to make them sound like they come through a communications system
 - Implemented voice lines into the tutorial system
- Improved tutorial system flexibility [2 days]
 - Audio lines and their subtitles are loaded from public fields to allow for easier changes
 - Helper functions to simply play a line and the corresponding subtitle
 - Subtitle display length based on the length of the audio clip rather than a fixed time period
- Fixed UI bugs with element display order [2 days]
- Created networked voice chat prefab for cross-scene connection [1 day]
- Created presentation video [3 days]
- Report Documentation: sections 3, 6.6, 8.3, 8.5 [1 week]

7. Software, Tools, and Development

7.1. Initial Development Procedure

During the first phase of game development, all the features were not functioning fully. Therefore, we created a debugging environment that would allow us to mimic a fully functioning feature and allow us to test more developed features. This was initially for mimicking the voice recognition system by using key binds that would send phrases to our internal speech analysis implementation as if they were received by our chosen speech recognition system. This debugging tool then became a crucial aspect of game testing and game balancing in the final stages of development because the voice recognition input is done using JavaScript so it doesn't work when testing the game in the unity editor. Without this tool, we would have needed to redeploy the compiled code on every change.

7.2. Debug Environment

The debug environment had a variety of features that helped us throughout development. The first being a single player mode so that we could individually test specific scenarios alone without the need to coordinate the whole team for a playtest. On top of this feature all in-game variables from pirate health to player damage could be modified for balancing, however this became such an interesting dynamic that we implemented it in the production version such that the lobby host could edit these variables to vary the difficulty of the game and increase playability. Finally the debug environment would also allow for keybindings to represent voice commands as mentioned in 7.1. and was particularly useful in early stages of development due to the incomplete speech recognition at the time.

7.3. Testing Research

Using unity, we decided to research automatic testing frameworks to allow us to have a series of rigorous tests that any team member could run before pushing their updates to the main development branch for merging to the live version. Unfortunately, both these frameworks are inadequate for unit testing our project due to their lack of support for the WebGL platform we used. Trillion was a promising solution but is still in beta, it is not even available on the package manager and requires manual installation from their git page. This also meant there was extremely limited documentation due to the lack of extensive community usage meaning we were not confident it would fulfill the role needed.

7.4. Unit Testing

Since we didn't use a formal framework provided by the package manager we opted to do our own custom assertions for unit testing this was mostly done in the speech recognition part of our project. This reduced amount of unit testing meant our manual testing had to be extremely thorough to compensate.

```
Assert.AreEqual("a8", Grammar.GetGridCoord("a8"), "a8");  
Assert.AreEqual("b7", Grammar.GetGridCoord("7b"));  
Assert.AreEqual("a8", Grammar.GetGridCoord("a ate"));  
Assert.AreEqual("b6", Grammar.GetGridCoord("be 6"));  
Assert.AreEqual("b3", Grammar.GetGridCoord("bee free"));  
Assert.AreEqual("i3", Grammar.GetGridCoord("eye 3"));
```

7.5. Manual Testing

Manual testing was most of our testing procedure as it allowed us to rigorously test features that are difficult to run automated tests for, such as multiplayer interactions. We organised these testing sessions once a week at the beginning of the week so that we would catch errors as fast as possible. In these sessions we would run through multiple game sessions with various configurations taking advantage of the debug environment we had created whenever applicable. Once an error had been isolated a git issue would be created immediately and assigned a priority, we would do this until we had gone through all scenarios and edge cases we could come up with. Now the list of formulated tests is completed we would do a minimum of 2 playthroughs 1 where we would purposely lose and 1 where we would win. This allows us to catch any unpredictable crashes that might occur outside of the testing scope completed previously. Finally, we talk among ourselves to brainstorm what could be causing the issues found in the testing session and assign the git issue to whoever is most knowledgeable on the error. For example, an error found in the multiplayer would be assigned to the team members that worked on the multiplayer feature. This meant team members would rarely find themselves working on an issue that was completely different to their main workload.

7.6. Public Testing

A sample size of 6 users is small and therefore some errors and issues are bound to be missed. To minimise this, every fortnight we would attempt to host a game night where we would invite friends and family to test out the game. This doubled up as a bug test and a gameplay test to indicate anything that we should tune. After a quick 2-3 games we would gather feedback on the game. This feedback fell into 3 categories, bug reporting, gameplay additions and fluidity issues. Bug reporting followed a similar procedure as our in house manual testing – an issue was

created and assigned to the most qualified team member. Gameplay additions were suggestions from the users that they believed could enhance the gaming experience, although these did not always make it into the final game they were taken into consideration. Finally, fluidity issues were where we would receive feedback on how it felt to play the game, focused mainly on controlling the player in game; this led to adding a wide variety of grammar to allow for many to one command executions for the speech recognition system. An example of such a change would be including relative direction commands to be used as well as the original cardinal directions.

7.7. Git Flow

Git Flow is a branching model that builds on top of Git's version control system. The model prescribes certain usages to enable a robust version control. It is specifically good in this project's scope as its feature system is brilliant for single features that add significant value to the overall project. Issues cropped up in our manual testing phase would be handled on the bugfix branch as this branches off of the main deployment and therefore is what is tested in our regular meetings. These hotfix branches are intended to be singleton fixes and shouldn't span over multiple version updates so it's important they are handled quickly.

Since we used discord as a primary communication point we integrated a bot that automatically updated a designated channel with all pushes and feature changes. This gave us an easy and quick access point for keeping up with how peoples progress is coming along as it also indicates when issues are closed.

8. Technical Content

8.1. Speech Recognition Engine

8.1.1. Google Chrome

Initially the game was only using the WebSpeech API integrated directly into the Google Chrome browser. Speech recognition is accessed via the SpeechRecognition interface, which provides the ability to recognize voice context from an audio input (normally via the device's default speech recognition service) and respond appropriately [1]. It is a very easy engine to use as you simply need to add around 40-50 lines of javascript, which automatically starts listening to the user's microphone. The engine then opens a connection to Google Cloud's servers via which the microphone bytes are sent, this allows the servers to run its ML algorithms that extract the speech from the raw data. This data is returned to the javascript voice recognition client via the connection as a string and the client forwards them to the Unity context instance CommandListener's *GetResponse(string)* method (or *GetFinalResponse(string)* method if the result text is final). After that, the CommandListener object processes the words and executes them as in-game commands (described in 8.1.3).

In the future Google plans to enable the use of grammar, but right now it is not an option. The code for adding grammar to the SpeechRecognition interface is already implemented. Nevertheless, it does not have any impact on the recognition itself.

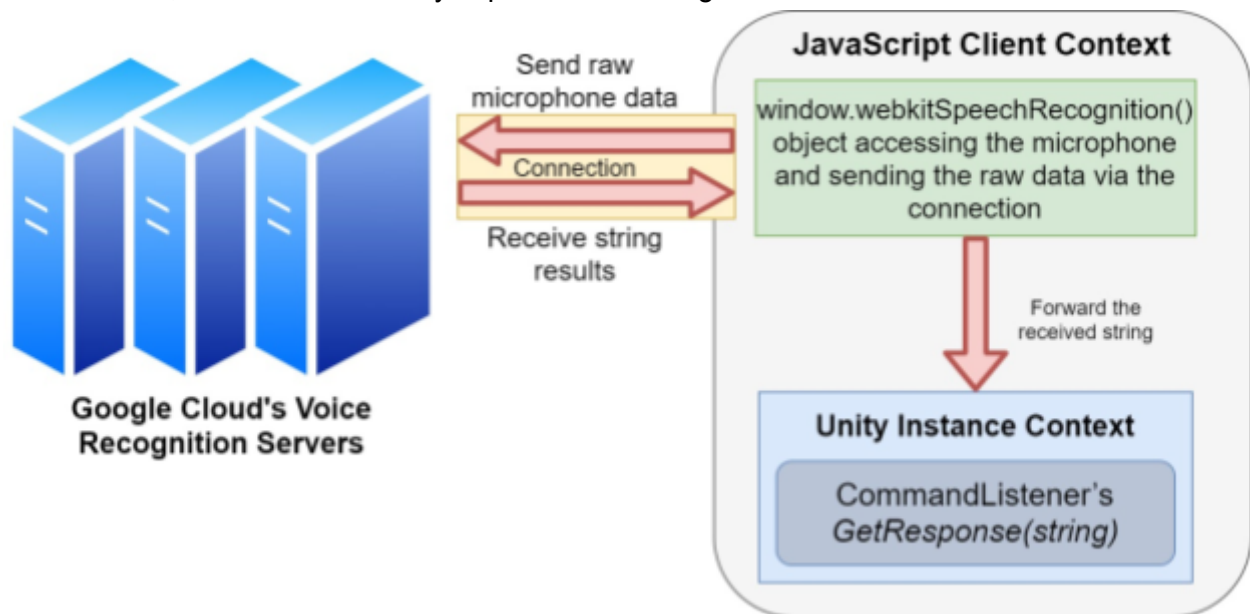


Figure 8.1. Process flow of Chrome Voice Recognition

8.1.2. Non-Chrome Browsers

Voice recognition on Non-Chrome Browsers is a lot more complicated. It is all because in order to use Google Cloud's Voice Recognition services, you need to create a Voice Recognition Project on Google Cloud Console in order to enable billing (as this is a paid service). To use the service you need to have provided an API Key or Google Application Credentials JSON file to your application, so that you can authorise your usage. In our case we used the JSON Credentials, which were obtained via the Google Cloud Console. They were inlined as a constant string inside the code as the live version hosted on heroku.com was managing the file system in a different way (mainly because they are using Linux) and a file could not be used inside of the code when the file was inside the project solution. Using the wwwroot folder for this file was also not possible as this file should not be accessible to the users because the speech-to-text services are paid. If someone has this file they can use it for their own purposes and we will be the ones to pay for their usage.

Furthermore, direct transmission of raw microphone data from the client to the Google Cloud API is not possible because you need the JSON Credentials file and this would require it to be shared with the client, which should not happen. Therefore the flow of raw microphone data is a bit indirect. The delay due to this 'detour' is around a few hundred milliseconds, which is an unrecognisable difference between a user on Google Chrome and a user on a different browser (e.g., Firefox).

We used a 3rd party library called UAParser [2] to determine which browser was being used. UAParser provides the functionality of extracting all types of information about the browser in which the javascript code is running. A function called isChrome() then uses the information from UAParser to check the name of the browser. This isChrome() function is used in the function starting the voice recognition to resolve which recognition engine should be used, i.e. the one described in 8.1.1 or the one here. For the Non-Chrome speech-to-text engine the approach was much harder to implement, but the result was more than satisfactory.

On the client-side, a websocket is opened between the client browser and the server-side (client-side code is available at SpeechRecognitionScript.js). The microphone audio is sent from the client to the server via this WebSocket. To send the microphone data, it had to be recorded to a stream using the navigator.getUserMedia API. This raw microphone data is then encoded in the necessary format using RecordRTC [3], so that it can then be easily decoded and its speech extracted by Google Cloud Services.

On the server-side, a connection is opened to Google Cloud Speech Recognition Servers via the SpeechClientBuilder API, which is imported to our .NET project by the Google.Cloud.Speech.V1 [4] Nuget package. In order to open the connection the JSON Credentials are used for authorisation when initialising the SpeechClientBuilder. The configuration of the engine is set, e.g. Audio Encoding, SampleRate (Hz), Audio Channels, Language, and Speech Context, which is supposed to provide the grammar and expected phrases. However, the Speech Context does not currently work (planned to be used in a future

release of the Google.Cloud.Speech.V1 library) and therefore, there is no grammar logic actually being executed. Interim Results are also enabled, because this makes the speech recognition more robust, even though the early results might not be as accurate as the final ones. Still when they are correct, the whole system feels much smoother and when they are incorrect, the game engine [section 8.1.3] just either ignores them or predicts the correct meaning.

Furthermore, a handler for the results received back from the Speech Recognition Engine is set and the handler just returns the recognised phrases straight back to the client browser javascript context via the opened WebSocket. Additionally, a handler is also set for receiving the microphone bytes over the socket and these bytes are taken inside of a buffer and redirected to the Google Cloud Engine via a StreamingRecognizeRequest object. In the end, there is logic which closes the connection when the client-side requests that and all objects will get cleaned by the garbage collector implemented in C# as the scope will get closed.

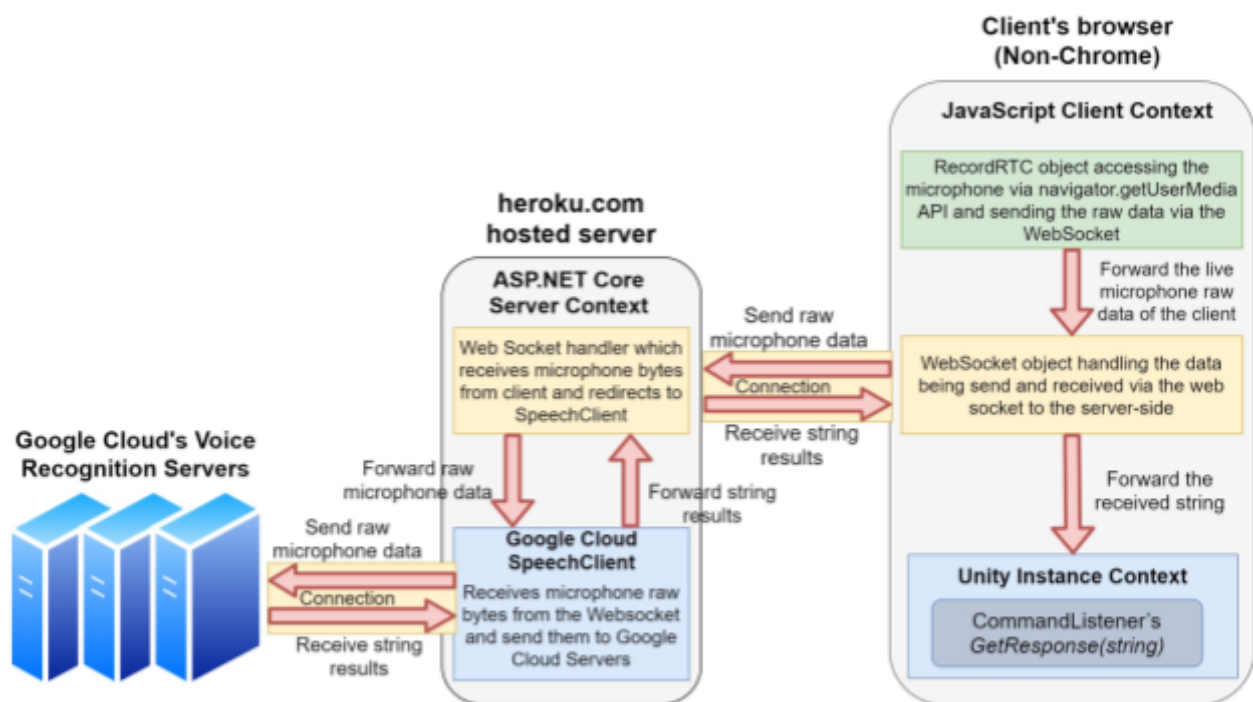


Figure 8.2. Process flow on Non-Chrome Voice Recognition

8.1.3. Text-to-actions words processing

Once the microphone input has been recognised and converted to a string, it is passed to the unity instance in one of two ways. If it is an intermediate result it is passed via the *GetResponse(string result)* method, if it is the final result it is passed via the *GetFinalResponse(string result)* method.

When an intermediate result is passed to the unity instance it is first added to a list of strings containing all detected phrases for that spoken phrase. It is then parsed using the grammar to get a *Command*, if the phrase is successfully parsed the command is performed.

When a final result is passed to the unity instance it does the same thing as the intermediate results until it has been parsed to get a *Command*. If any of the intermediate results or the final result were successfully parsed, the command is performed. Otherwise the *FindSuggestedCommands* method is called to find the closest command to each of the phrases stored in *_detectedPhrases*. The *FindBestSuggestedCommand* method is then called to select the best suggested command. The user is then prompted to repeat the new command or the command is performed depending on how certain the program is that the suggested command is what the user intended. Finally the detected phrases list is reset along with some other variables used during the speech-to-text process.

The grammar is a collection of lists of strings. The input phrase is then compared against the lists to check if the phrase contains any of the strings in the lists. Once one of the main command words is detected, it then checks to see if the phrase contains the other data required for that command. For example if a movement command word is detected, it would then try to find a direction or destination. If all the required data is detected a *Command* object is returned.

There are two methods used for finding a suggested command. The first method is based on the completeness of a command. A command with all the required data, including the main command word, would have a completeness value of 1. This method allows you to detect which command has the most of its required data provided. The second method is based on the Levenshtein Distance from the given phrase to each command word. The Levenshtein Distance is calculated for each word in the phrase to each command word and the smallest distance is returned. The confidence is then set to $1 - (\text{smallest distance} / \text{length of command word with the smallest distance})$. If a command is found with any amount of data provided that suggested command is used, otherwise the distance based suggested command is only used if it has a confidence greater than 0.5. The threshold for the distance based suggestion is especially important for the shorter command words.

8.1.4. Additional Speech Recognition Engines Researched

Before committing to the use of Google Cloud's Recognition Software due to its billing aspect, we decided to research other options that could meet our requirements for speech recognition whilst removing the need to use a premium service. The most promising solution available was a research inclined open-source software named Kaldi ASR [5]. Initial reading of Kaldi's website made obvious that the software is highly supported for research purposes and lacks a beginner's introduction, instead it opts to assume the user has a high-level understanding on the foundation of speech recognition algorithms and how the Kaldi model is designed. This led to extensive research on topics surrounding speech recognition and how different machine learning models have catapulted our understanding and accuracy in this scientific field. With a deeper understanding the software was installed on a team members system along with an example found on Kaldi's website [6]. Unfortunately, due to the steep learning curve of the software it was extremely difficult to set it up. As well as this, the issue of providing our own dataset for training would prove to be a huge task as most examples provided had on the low end 100 hours of training content. This would give us a decently accurate model but google cloud is still at the cutting edge of speech recognition software, so it was decided in a team meeting to forgo the alternative option in favour of our original choice.

8.2. Voice Chat

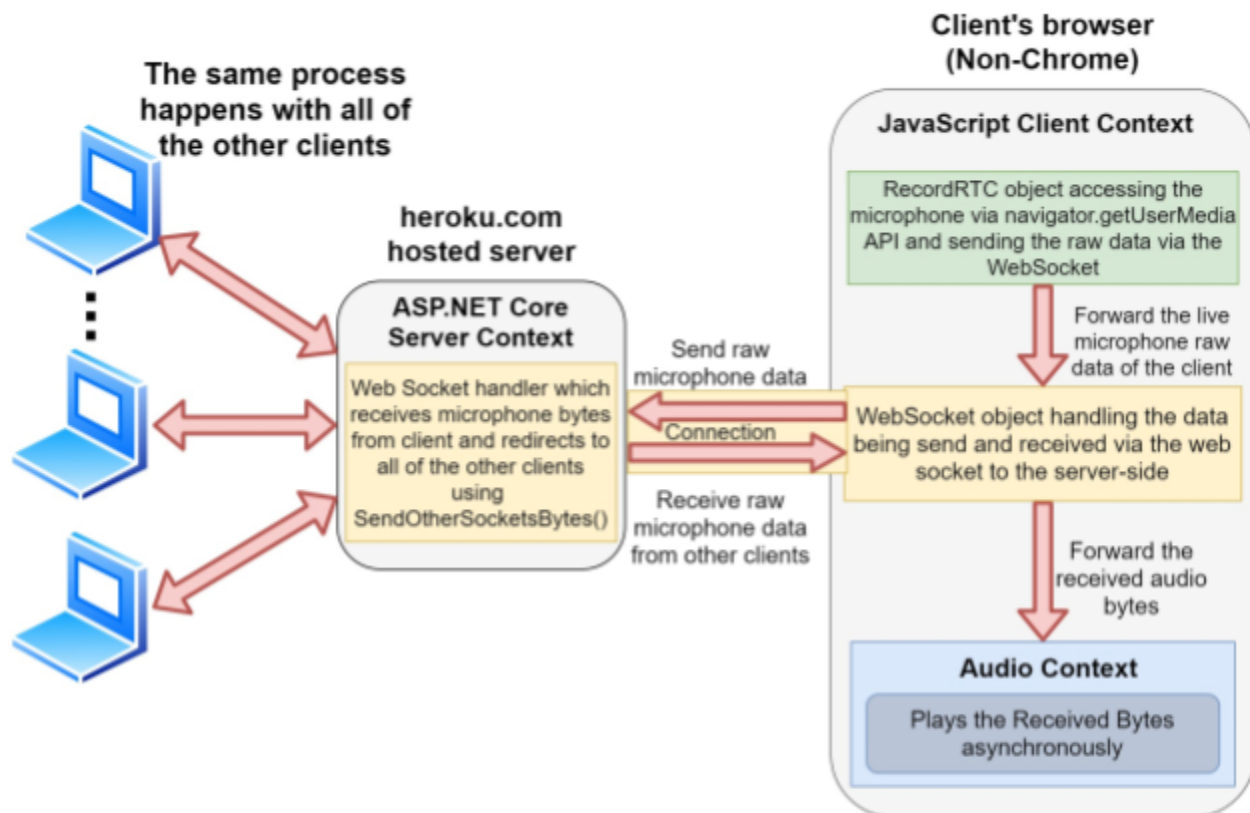
8.2.1. Voice Chat Using RecordRTC and WebSockets (Initial version)

The initial version of the Voice Chat was working very primitively and because of that it was decided that its quality was not sufficient for our purposes and therefore the method was eventually scrapped in favour of the modern WebRTC technology. The source code is still inside of the server-side project in VoiceChat.js (client-side) and Startup.cs (server-side).

On the client-side, at first a WebSocket is opened to the necessary URL pointing to our server. It is used to send the raw microphone data from the client to all of the other clients connected to the server and waiting for audio to be received. A handler for received bytes from the server is also set, which directly plays the audio in these bytes. It does this by first decoding them and saving them to a buffer and then creating an Audio Buffer Source using the `createBufferSource()` method of the AudioContext API [7]. Then the source is connected to the output node and played immediately. From the client's perspective all of the received bytes will be played 'asynchronously' as the server receives bytes from all of the other clients in the chat at the same time and these bytes are not combined in any way. Whatever is received, irrespective of the client from which these bytes come, the audio will be played straight away. The fact that they are played simultaneously will make it sound like a single audio channel. However, in reality there will be many audio fragments which are all played simultaneously and 'chaotically', which is the reason why this method did not provide very clear audio and consequently, a better alternative had to be found.

Regarding the way audio is encoded before being sent via the WebSocket, the same method is used as in the Non-Chrome Speech Recognition (8.1.2). The navigator.getUserMedia API is used to obtain the media stream from the user's microphone and then the audio is encoded using the RecordRTC library [3]. Finally, the captured and encoded microphone bytes are sent straight to the server via the WebSocket.

On the server-side, a WebSocket connection is opened to all the clients that are in the voice chat. Each of these clients is added to a list of WebSockets. Every time one of these clients sends its microphone bytes, these bytes are directly sent to all of the other clients via the SendOtherSocketsBytes() method. The method uses the list of sockets and provides each of its members (except the one that sent the bytes initially) with the received bytes. Therefore, everything that is received by one user is routed to all of the other users who then receive the microphone data and play it directly.



8.3. Process Flow of RecordRTC with WebSockets Voice Chat

8.2.2. Voice Chat Using WebRTC and SignalR (Production version)

The production version of our in-game voice chat has very high audio quality (as good as it actually gets), because WebRTC is a very modern peer-to-peer real-time communication solution and it has proven to be the standard for the purpose. The technologies behind WebRTC are implemented as an open web standard and are available as regular JavaScript APIs in all

major browsers [8]. The way we implemented it actually supports a video chat as well and everything that needs to be changed in order to transform the voice chat into a video chat is just to change a single constant in the javascript file called USE_VIDEO.

In order to use the WebRTC technology a signalling server had to be implemented, which is to serve as an intermediary to let two peers find and establish a connection while minimizing exposure of potentially private information as much as possible. In ASP.NET Core the technology responsible for this purpose is SignalR [9], which is like an interface, and can be used to invoke methods on the server-side from the client and the other way around and pass information via the methods' parameters.

On the client-side (VoiceChat2.js), initially the URL for the signalling server is determined and then a connection is initialised using the *signalR.HubConnectionBuilder()* API. However, only after the DOM tree has been fully loaded (jQuery.ready function handler) the connection is actually opened. To join the voice chat, the *joinVoiceChat(chatName)* method needs to be called from the Unity Context with the Lobby Room Id used as the chatName. This Lobby ID is a random GUID and therefore, it was the perfect candidate for using as the chat room ID. Then the function responsible for setting the local media is invoked, which uses the *navigator.getUserMedia* API to create the stream object of the current user's microphone capture and the callback invoking the *JoinChat(string channelName)* method on the server-side is executed. This method checks whether there is already a channel with this name and if not creates it. Then it makes all of the other clients in the channel add the new user to the call and also makes the new client add all of the others in its own side of the call. This is because in order to start a WebRTC connection between 2 users they both need to 'work' on creating the connection. All users create a new *RTCPeerConnection* object and they both share their ICE (Interactive Connectivity Establishment) Candidate objects used for detailing the available methods the peer is able to communicate to the other peer. The newly joining user is the one that needs to send the WebRTC connection offer, which is simply sending the local description [10] (used for specifying the properties of the local end of the connection, including the media format) to the other user.

A new video or audio node is created when a client receives the stream object of a peer they are connected to (via the onaddstream callback). It is then appended to the DOM tree and the peer stream is added as a source object to the media node. In this way the user now has the stream actually being played on its browser. After all peers have established a bi-directional connection with each other and they all play the streams from the others and share their own to the rest the voice chat connection is completed.

In order to leave the voice chat, the VoiceChatController script in the Unity Context calls the leaveVoiceChat function in the javascript context using *Application.ExternalCall()*. This function then makes a call to the LeaveChat function on the server using the SignalR connection. Then all of the other clients in the call remove the leaving peer from their connections list and the leaver removes all of the other clients on its side. If the user to leave is the last participant in the specific voice chat, then the voice chat is removed from the list of chats on the server-side.

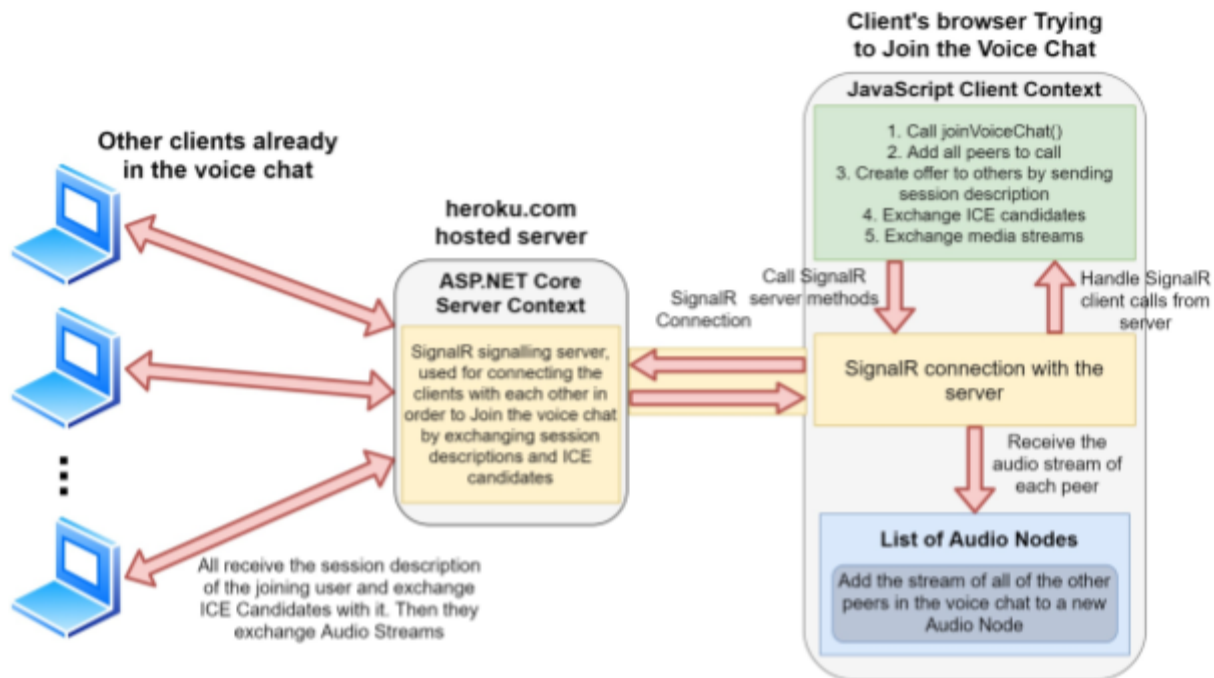


Figure 8.4. Process flow of WebRTC and SignalR Voice Chat

8.3. Multiplayer Networking

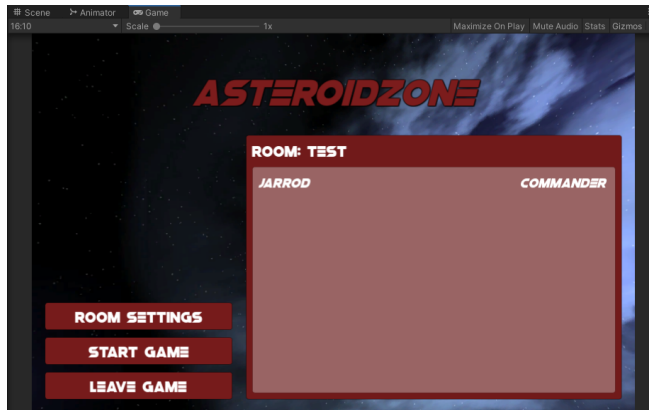
8.3.1. Photon Unity Networking (PUN)

After researching the multiplayer networking options available for unity we settled on using PUN as we felt it was most suited to how our game would work. If we chose to host our own dedicated server to which players connect it would be both time consuming and expensive. Using the PUN API means this is no longer an issue as games are instead hosted by one of the clients (in our case the station commander).

There are however downsides to using this networking model. For one it makes our game more susceptible to host hacking, where the host player uses the fact that they have access to more host controlled behaviors to their advantage. As a team we decided that this wouldn't be an issue for our game, as it's focus is more on cooperation than being competitive. Another potential problem is the need for host migration in the case that the host player leaves the game or loses connection. Fortunately Photon manages this and makes the transition as seamless as possible. The last and biggest downside is that the device of the host player can negatively affect the game as a whole, if the device is outdated or the internet connection is poor which can result in more inconsistencies in gameplay. To help mitigate this issue we worked to minimise the amount of data transferred over the network, only doing so when absolutely necessary.

In order to synchronize the gameplay across the network we used remote procedure calls (RPCs) on networked objects. Each networked object has a unique ID called a PhotonView, which allows Photon to differentiate between game objects belonging to different players. We aimed to limit the amount of game objects that would require a PhotonView to benefit game performance. One example of this was using the position and rotation of a networked player ship to determine the trajectory of a laser shot by a player, rather than making the laser a networked object in itself, as this would put more strain on the network.

8.3.2. Lobby System



The lobby system is a key component to the game as it allows players to select and join specific lobbies in a pre-game state, this means users don't have to rely on random joining to connect to their desired game.

To implement this feature the MainMenu scene was extended to include 2 additional control panels that would relate to the lobby listing room and the lobby itself. As a player traverses these menus the client is updating the player's network state, this allows the

client to retrieve a list of all available rooms that the player can join. This list is traversed by the client and each room instantiates a RoomListing prefab that will show the player the name of the room and appear on the roomListingPanel. The player can click on the listing to invoke a join method where the player will see the lobby the host has created.

In this lobby the players are listed using a similar UI design to the lobby listings, this gives a clean fluid feel to the whole system. The player can see who the host is as they are labelled as the station commander. Whenever a lobby is created or no longer available the method OnRoomListUpdate is called with the changed lobby, whether it be no longer available or a newly created lobby. The client then iterates through each currently available room to remove any matching room names. If a matching room name is found it means the room is no longer available, otherwise a new listing is created.

8.4. Tactical Camera

8.4.1. Determining Functionality

The station commander's tactical camera is the only part of the gameplay which requires the use of the mouse rather than being voice controlled. We made this decision because we felt that a voice controlled camera would be too slow and clunky to use leading to player frustration. In order to determine what functionality the camera would have we looked at a variety of space strategy games with similar cameras and came to the conclusion our camera would require 4 basic actions: panning, pivoting/rotating, zooming, and focusing on an object.

8.4.2. Camera Rotation

In order to aid camera functionality an empty GameObject was attached as a child to serve as the focus point for the camera. This focus point allows easy camera pivoting with the use of Unity's `RotateAround` function.

8.4.3. Tracking Objects

The camera controller also holds a reference to the currently tracked object.

The first step in tracking an object is determining which object to track. A ray is generated using Unity's `Camera.ScreenPointToRay()` based on the mouse position. The ray is then fed to Unity's physics raycast system which attempts to find an object along that ray. If an object is found, `trackingObject` is set to the GameObject and the camera zooms to its minimum distance.

Each frame while an object is being tracked, the camera controller checks the positions of the target and the camera's focus point. If there is a positional difference then the camera and focus point's positions are adjusted by the positional difference.

8.4.4. Camera Panning

The camera controller first `null`'s the `trackingObject`. To calculate the position change of the camera it's `forward` and `right` transform vectors are multiplied by the mouse movement. The new focus point position is then bounds checked to ensure that it stays within the gamespace grid so that the station commander doesn't lose sight of the play area. If it is within bounds then the camera and focus point positions are updated accordingly

8.4.5. Camera Zooming

When zooming a new camera position is calculated by multiplying the camera forward direction by the mouse scroll delta. The distance of this new camera position from the focus point is then calculated and if it is within the min/max distance set in the inspector then the camera position is updated.

8.5. Fog of War

8.5.1. Initial (Tile-based) Fog of War

When we first implemented the fog of war system we used a tile based system where each square on the grid was a tile. We used four layers of masks on each tile, one for each player. Then the Unity Context calculates which tiles should be visible to each player every frame and subsequently, decides whether or not to render each `FogOfWarObject` (asteroids, pirates and players) based on the visibility of the tile it is on. This method was used instead of Unity's culling mask due to its lack of flexibility for multiple clients.

8.5.2. Final (Radius-based) Fog of War

The final fog of war system that we used was radius based rather than tile based. After some discussion within the team we decided this would make more sense. The only issue we had with this was working out how to indicate the visible area to the player. This was less of a problem with the tile based system because we used black planes to cover each grid square that was out of range. In the end we decided to use a ring around the player's ship model and a black plane with a transparent circle to hide the rest of the minimap. `FogOfWarObjects` are then hidden using Unity's layer system by being assigned to a layer and the camera only rendering only selected layers, which are supposed to be visible to the particular player.



8.6. Pirate AI

The pirate AI is a decision tree based on multiple variables that hold information about the current situation. For example whether or not the pirate currently knows the location of the space station, or the distance to the nearest player. Pirates have a 'personality' variable which controls whether they prioritise targeting the space station or nearby players. We introduced this variable because previously players could shoot at the pirates attacking the station with no danger of being shot back, now some of the pirates will continue attacking the space station and some will shoot the players.

If a pirate finds the space station it will alert all other pirates to tell them its location. It does this using the `PirateSpawner` class, we set the pirate parent transform to be the spawner so we could use the `GetComponentInChildren<PirateController>()` method. This is better for the game's performance than using the `GameObject.FindGameObjectsWithTag()` method that we were using previously.

We implemented the code to handle the pirates losing the space station's position if it moved, however we didn't add the commands to move the station in the end. Scout pirates would

continue searching as they did before they found the station, but any elite pirates would either attack any players they saw or try to leave the area. They would use an algorithm we implemented to calculate the nearest point on the edge of the grid from a given position, which would then be set as their destination. If they got within a small distance of it they would despawn.

9. References

- [1] Web Speech API. MDN Contributors. Last modified: 15/02/21. Accessed: 04/05/21. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API
- [2] UAParser.js. Faisal Salman. Accessed: 07/05/21. Documentation available at: <https://faisalman.github.io/ua-parser-js/>
- [3] RecordRTC Library. Muaz Khan. WebRTC JavaScript Library for Audio+Video+Screen+Canvas (2D+3D animation) Recording. Accessed: 11/05/21. Available at: <https://recordrtc.org/>
- [4] Google.Cloud.Speech.V1 Nuget Package. Google Inc. Accessed: 10/05/21. Available at: <https://www.nuget.org/packages/Google.Cloud.Speech.V1/>
- [5] Kaldi Home Page : <https://kaldi-asr.org/doc/index.html>
- [6] Kaldi Example Listing : <https://kaldi-asr.org/doc/examples.html>
- [7] AudioContext. MDN Contributors. Last modified: 19/02/21. Accessed: 13/05/21. Available at: <https://developer.mozilla.org/en-US/docs/Web/API/AudioContext>
- [8] WebRTC. Google WebRTC team. Accessed: 14/05/21. Available at: <https://webrtc.org/>
- [9] Real-time ASP.NET with SignalR. Microsoft Developers. Accessed: 14/05/21. Available at: <https://dotnet.microsoft.com/apps/aspnet/signalr>
- [10] WebRTC Local Description. MDN Contributors. Accessed: 14/05/21. Available at: <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/setLocalDescription>