TEAM H - BAG OF SNAILS

KITCHEN FEUD

LINDA LOMENCIKOVA *TEAM MANAGER*
PRAGYA GURUNG *LEAD DESIGNER*
LOKHEI WONG *LEAD PROGRAMMER*
HAMZA SHAHID
JACKSON LAWRENCE
KAROLINA KADZIELAWA

University of BRISTOL

TABLE OF CONTENTS

Table of Contents 1

1. Signed Contributions 3

2. Top Five Contributions 3

3. Nine Aspects Pages 4

3.1 Team Process 4
3.2 Technical Understanding 4
3.3 Flagship Technologies 4
3.4 Implementation & Software 4
3.5 Tools, Development & Testing 4
3.6 Game Playability 5
3.7 Look & Feel 5
3.8 Novelty & Uniqueness 5
3.9 Report & Documentation 5

4. Abstract 5

4.1 Story 5
4.2 Gameplay 6
4.3 Point system and minigames 6
4.4 Sabotaging 6
4.5 Controls 6

5. The Team Process and Project Planning 7

5.1 Scrum Organisation 7
5.2 Task Management 7
5.2.1 Trello 7
5.2.2 Individual Contribution spreadsheet and Points System 7
5.3 Communication and Organisation 8
5.3.1 Communication on Discord 8
5.3.2 Meetings in Teams 8
5.3.3 Planning in OneNote 8
5.4 GitHub 9
5.4.1 branching strategy and Git workflow 9
5.5 Reflection 9
5.5.1 What did not work 9
5.5.2 Possible Improvements 9

6. Individual Contributions 10

6.1 Linda Lomencikova 10
6.2 Pragya Gurung 11
6.3 Lokhei Wong 12
6.4 Hamza Shahid 13
6.5 Jackson Lawrence 14
6.6 Karolina Kadzielawa 15

7. Software, Tools, and Development 16

7.1 Development and Testing 16
7.1.1 Git and our branching Strategy 16
7.1.2 GameCI and Testing 16
7.1.3 Adding new logic 17
7.1.4 Other software tools 17
7.2 Software Developed 17
7.3 Design Software and Tools 18

8. Technical Content 18

8.1 Photon 18
8.1.1 Synchronising Objects 18
8.1.2 RPC Calls 18
8.1.3 Servers and Lobbies 19
8.1.4 Custom Properties 19
8.1.5 Re-joining the game 19
8.2 Voice chat 19
8.2.1 Speed Test, Security and Privacy 20
8.2.2 Technical Difficulties 20
8.3. Controls, cameras and movement 21
8.3.1 Cameras 21
8.3.2 RigidBody Movement 21
8.3.3 Online Movement 21
8.3.4 Optimisations 21
8.3.4 Control-driven Animations 22
8.4 Game Setup 22
8.4.1 Navigation 22
8.4.2 Interactable system 22
8.4.3 Minigame management 23
8.4.4 Order and serving system 23
8.5 Sabotaging and Uniqueness 24
8.5.1 Sabotaging, opponent team 24
8.5.2 2D Mini Games 24
8.5.3 Less restrictive gameplay - cooking or sabotaging 24
8.5.4 Voice-Chat Interaction 24
8.5.5 Replayability 24

8.6 AI 24
8.6.1 Waiters 25
8.6.2 Owners 25
8.7 Complex particle systems 26
8.7.1 2D Particle Systems 26
8.7.2 3D Particle Systems - Cooking Systems and Random particle event system 26
8.7.3 3D Portal Doors 27
8.8 Shaders 27
8.8.1 Toon Shader 27
8.8.2 Outline Shader 28
8.8.3 Post Processing 28
8.9 Dynamic Audio - SFX and Music 28
8.9.1 Dynamic Music 28
8.9.2 Dynamic SFX 29

1. SIGNED CONTRIBUTIONS

Linda Lomenčíková:	1.0
Pragya Gurung:	1.01
Lokhei Wong:	1.03
Hamza Shahid:	1.04
Jackson Lawrence:	1.0
Karolina Kadzielawa:	0.92

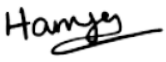
I, below-signed team member of Bag of Snails, agree that the above-mentioned contribution weights accurately and fairly represent the relative contributions by each team member. I can confirm I have spent at least 400 hours on the project and actively engaged with the team and contributed to all deliverables. I have read the report and agree it is a fair and accurate representation of the team's efforts.

Signed on 5.5.2022 by:

Linda Lomenčíková 

Pragya Gurung 

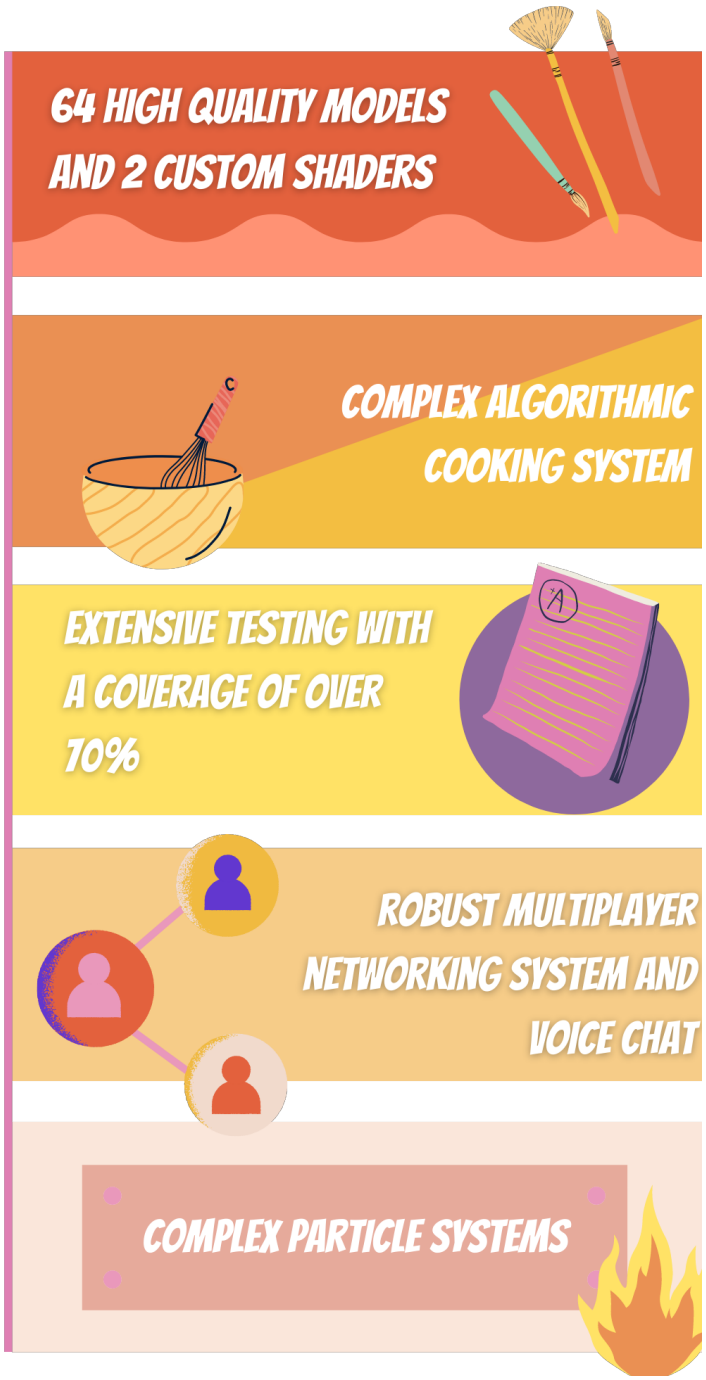
Lokhei Wong 

Hamza Shahid 

Jackson Lawrence 

Karolina Kadzielawa 

2. TOP FIVE CONTRIBUTIONS



KITCHEN FEUD TEAM VIDEO

3. NINE ASPECTS PAGES

3.1 TEAM PROCESS

Working on a project of this scale required adhering to a methodological and systematic development process. We strived to stay efficient and productive to deliver high quality results.

We followed the AGILE development process and scrum framework for project management [5.1].

In order to manage our tasks and keep track of the deadlines and milestones we used an agile board [5.2.1]. It served not only as a planning tool, but also an informative record of our efforts.

On top of consistent online communication and subgroup meetings, we would meet in person or online twice a week [5.3].

We tracked and managed our work distribution among team members via a points system, where we would grade our tasks based on complexity and time invested [5.2.2].

We also employed a clear and well defined strategy for managing our GitHub workflow [5.4.1].

3.2 TECHNICAL UNDERSTANDING

Tackling some of the technical challenges required thorough research and deep technical understanding.

The issue of items clipping through walls was resolved through camera layering [8.3.1]; we added a sub camera and a culling mask. This also allowed for the items held by the players to be in their field of vision at all times.

Creating realistic and dynamic particle effects was also made possible by researching how different phenomena like boiling or smoke behave in real life [8.1.2].

In order to cater to players with poor Internet connection, we scale the voice chat audio quality to their bandwidth [8.2.1]. We carried out multiple tests to get the appropriate values for the bitrate, bit depth, sample size and sample rate of the audio received on the user's end to find the right bandwidth/audio quality tradeoff.

Deciding on the right pathfinding navigation method for the AI servers is another example of how research informed our development process [8.6]. Having considered options from reinforcement learning, to algorithms such as A*, we settled on Unity's built in NavMesh mechanism.

3.3 FLAGSHIP TECHNOLOGIES

- Complex Particle Systems [8.7]
- Voice Chat [8.2]
- Artificial Intelligence [8.6]

3.4 IMPLEMENTATION & SOFTWARE

We adhered to the 4 principles of Object Oriented Programming [7.1.3].

Our complex algorithmic ordering and serving system, which is one of the central mechanisms of the game, is heavily interlinked with the dish and ingredient databases, tray assignment, points system, minigames, ticket generation and more [8.4.4].

The majority of game logic had to be synced across the network using Photon [8.1], this involved advanced solutions to issues such as maintaining smooth movement or syncing the timer [8.1.4].

Additionally, we manipulated javascript files from the Agora Unity WebGL Plugin [8.2] to edit certain functionality.

3.5 TOOLS, DEVELOPMENT & TESTING

We followed the GitFlow branching strategy and enforced branch protection rules for our development branch [7.1.1].

Thanks to our discord bot set up using GitHub Actions [7.1.1], in conjunction with our continuous integration, continuous delivery, and continuous deployment (CI/CD) system set up using GameCI [7.2], we were notified about actions such as the outcome of our automated tests or deployment progress.

Our testing consisted of a vast suite of both Edit Mode and Play Mode tests, as well as rigorous and extensive manual testing and user testing sessions. We used code coverage metrics to evaluate our code [7.1.3].

We also used a number of professional design tools, such as Maya, Blender and Adobe's Mixamo to create our models and other assets, which aided in creating the signature look of the game [7.3].

3.6 GAME PLAYABILITY

The adequacy and intuitiveness of our control system [8.3] makes the game engaging and immersive. This is largely thanks to the feedback from our user testing sessions, which allowed us to adjust them to the players' needs. The user feedback informed the choice of movement sensitivity as well as music and sound effects volume. It also provided us with ideas for new features, for example the inclusion of alternative controls for people who are left-handed or first person camera view to increase intractability and immersion. Even though we adjusted the controls to suit the users' preferences, the players can still change the movement and rotation sensitivity, as well as the music volume in game through a settings menu.

We also make sure to include a tutorial animation at the beginning of the game [8.4.1], as well as a bar at the bottom of the screen with instructions on how to play throughout the duration of the entire game [8.4.1]. What is more, players are instructed on how to cook dishes through the use of a glowing effect [8.8.2].

Rejoining the game is a feature that also enhances the robustness of the game and caters to players with an inconsistent Internet connection [8.1.5].

The inclusion of 2D minigames allow for frequent changes in the pace of the gameplay and makes it more engaging and diverse [8.5.2].

3.7 LOOK & FEEL

Early in the project, we established the overall look of the game. This included designs for the UI, kitchens and characters as well as the main colour palette. Doing so ensured the aesthetics of our game stayed consistent in all aspects.

Based on those concept designs, all 64 core models for the game were made in house using Maya and Blender [7.3]. These models were kept low-poly with only the character models exceeding 1k polys.

We also constructed custom shaders for both design and gameplay purposes [8.8], as well as complex 3D particle effects such as heat distortion and fire extinguisher particles interacting with flame particles [8.6].

3.8 NOVELTY & UNIQUENESS

The conflict between two kitchens is not only one of the central themes of the game, but also one of its unique features. The fact that the users are split into two competing teams, as well as the various sabotaging mechanisms are what sets the project apart from similar games [8.5.1].

The cooking mechanism itself is unique too, as it involves completing 2D minigames [8.5.2].

Finally, at any time, the players can perform any of the available actions and are not limited in their choice of strategy. The unrestricted and free nature of the gameplay is very uncommon in cooking games [8.5.3]

3.9 REPORT & DOCUMENTATION

We wrote our report and produced the video with a high degree of attention to detail and aesthetic quality. The video features a range of gameplay, development and interview footage, as well as custom animations that depict the game in a clear and easy to understand manner.

The report is divided into clearly separable sections making it easy to navigate. It also utilises diagrams, graphs and tables in order to better convey its contents.

4. ABSTRACT

4.1 STORY

The owner of Panda-King had one ambition: to build the best restaurant in town. Nothing could stand in the way of reaching that goal, not even Meow Express - the new restaurant being built at the same time right next to his establishment. The competitor's confidence was admirable, as he made multiple remarks about his restaurants' superiority.

The Meow-Express owner knew his worth and was not afraid to let everyone know that it is him who will be running the best restaurant in town. What was the

Panda-King owner thinking? That his restaurant could compete with the great Meow Express?

The pandas already knew how they could teach the cats a lesson. The plan was simple: win the hearts of customers and critics by cooking the highest quality dishes, thus pushing the competition out of business. However, the pandas wouldn't stop there. The cats would suddenly find their ingredients missing, smoke suddenly appearing in their kitchen and appliances catching fire. The pandas were attacking and the cats did not hesitate to retaliate. This is war and there could only be one best restaurant in town.

4.2 GAMEPLAY

The main objective is to efficiently cooperate on preparing and serving orders made to the kitchen the player is in. An order can consist of multiple dishes and preparing a dish comprises multiple steps. The player needs to first collect the necessary ingredients from the fridge, pantry and freezer and put them onto the correct cooking appliance. Only then can they cook the dish by playing an either single or multiplayer 2D minigame. One finished, the food can be placed onto the appropriate serving tray and served, earning the team points. Cooking is a team effort, therefore the players can communicate using voice chat. However, fulfilling orders is not the only way to affect the outcome of the game. Players have the ability to enter the enemy's kitchen and sabotage the opposite team's cooking efforts through various methods [8.5.1]. Which strategy they choose is entirely up to them.

4.3 POINT SYSTEM AND MINIGAMES

The total points acquired by each team determines the winner of the game. Points are gathered by serving orders. The amount of points awarded for a given dish is determined by the score achieved in the minigame and the dish's complexity - dishes with more ingredients, as well as those that require multiple players to prepare are worth more points. Different dishes require playing different minigames, all of which share some key characteristics: it is possible to complete them in a short amount of time (around 20 to 30 seconds), they are 2D, and require speed and agility. They can be completed quicker, but at the cost of accuracy - players have to consider the trade-off between cooking fast, or cooking well. One of the minigames requires 2 players to complete and allows for gaining more points than others, while the rest allow only one player at a time to use the appliance the minigame is played on.

4.4 SABOTAGING

Sabotaging the opposite team's work is a rewarding and an encouraged strategy in our game. When in the enemy kitchen, players can make use of, but are not limited to, a number of sabotaging mechanisms. Each player carries one smoke bomb, which becomes available the moment they enter the enemy kitchen. When set off, it obstructs the view of players in that kitchen. The intruding players can perform all the actions they would in their own kitchen, which allows for example, setting off fires by leaving kitchen appliances on, blocking sub-rooms with overflowing ingredients, or even stealing cooked dishes. What is more, if a player manages to cook a dish in the competing kitchen and bring it back to their own, it is worth double the normal amount of points. Intruders can be removed from the kitchen by making use of the kicking mechanism: the players have to "hit" the intruder until they run out of health, as indicated by the bar which appears above their head when they enter the kitchen.

4.5 CONTROLS

The players move in four directions using the WASD keys. Using arrows is also available in order to cater to left-handed players. Holding the right mouse button locks the cursor and allows the mouse movement to be read as camera movement. Alternatively, players can turn using Q and E keys, should they be using a touchpad instead of a mouse. The view is in first person. Left mouse button is reserved for interacting with elements of the game. Left clicking on objects translates to picking them up and dropping them, entering a minigame, or serving, depending on the type of the object. In addition, user interface (UI) elements can also be interacted with via the mouse. The cursor changes when hovered over clickable items, making it easier to recognise objects that can be interacted with.

5. THE TEAM PROCESS AND PROJECT PLANNING

5.1 SCRUM ORGANISATION

[1]From the beginning, we acted in accordance with an agile development process. The project started with the creation of the product backlog, which contained a breakdown and specification of all work that needed to be done, in order of importance. As the project continued, this was updated at three different backlog refinement sessions.

Each week consisted of one sprint cycle. Each sprint involved at least two meetings, one for the start of the sprint and the other as a mid-point session. At the start of the sprint, the team manager would lead the team through the process of sprint planning, defining sprint goals and taking priorities out of the backlog. At this point we would allocate tasks. We encouraged self organisation, with each member pulling tasks from the backlog according to their skill and capacity.

During the midway meeting, we discussed the progress of each developer's tasks. As well as, to hold longer sessions for debugging, testing, planning or any other situation where the team needed to work together.

At the end of each sprint we held sprint reviews, giving and receiving feedback plus suggestions about the completed tasks. When needed we conducted a sprint retrospective. This is when we reflected upon the sprint, each individual team member and their task, the team's interactions and communication and the team process.

5.2 TASK MANAGEMENT

5.2.1 TRELLO

At the beginning of every sprint during task allocation. tasks were pulled from the top of the product backlog, containing all necessary items and artefacts to solve for the next deadline. The backlog containing all agreed goals was hosted on a Trello board, our main source of task management.

The Trello project board consisted of several lists: a list for each week (sprint), a list of all deadlines, as well as the project backlog list. When a new sprint started, the team would drag the relevant tasks from the backlog to the corresponding list, and assign their name and the tag. The tags used were: “done”, “in progress”, “resolved”, “high priority”, “low priority”. The items in the backlog were sorted by priority. This allowed for easy access to high priority items at the start of each sprint. We had three big revisions of the backlog where we went through all the tasks and reassessed their priority ranking.

This organisation was helpful for the overall smoothness of task allocation and reallocation. The board served as a nice visualisation of the amount of work to be finished at every point of the development process. The tags associated with each task allowed us to assess whether progress during the sprint was on track.

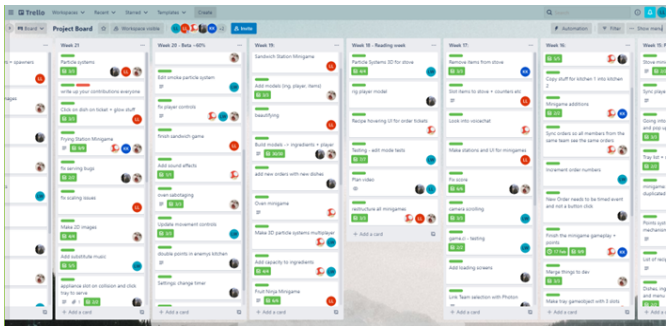


Figure 1: A screenshot of the team’s Trello board

5.2.2 INDIVIDUAL CONTRIBUTION SPREADSHEET AND POINTS SYSTEM

There were several long term tasks, where the responsible developer would start the sprint by dragging the task into a new list. As we needed to keep track of what were the exact tasks fulfilled by every team member, we made a simple spreadsheet shared in the groups Teams channel. We organised the spreadsheet by weeks and at the end of which, it was every member's responsibility to fill out.

In addition to simply writing the task, we implemented a points system. Each task could be assigned a number of points on the scale from 1 to 5, depending on its difficulty to execute and time needed to complete. These numbers were used to adjust the task allocation. After we implemented this system and noticed that the work was unequally distributed, we made sure during the next sprint, the respective members would complete work worth less or more points. Eventually the points normalised around 10-15 points of tasks per member per sprint.

[1] https://scrumguides.org/scrum-guide.html

The point system proved very useful for fairly judging individual contributions.

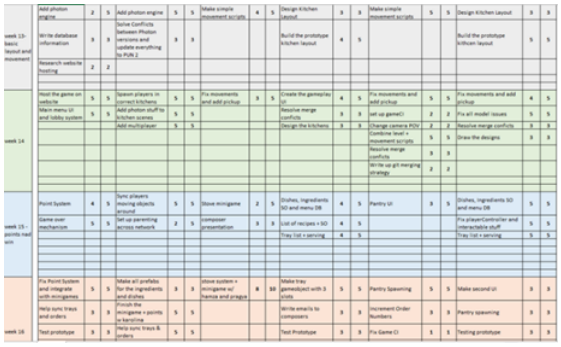


Figure 2: A screenshot of the team's point system spreadsheet

5.3 COMMUNICATION AND ORGANISATION

During the development process, the team made sure to keep in constant contact, using different communication methods such as Discord and Microsoft Teams.

5.3.1 COMMUNICATION ON DISCORD

We created a discord server For the team to use as the primary method of chatting. The server was primarily used as a more casual and accessible method to organise meetings and ask questions. In addition, three main text channels were utilised: general to chat, notes to share detailed progress explanations and resources as an easy way to store all interesting and important links to assets and tutorials.

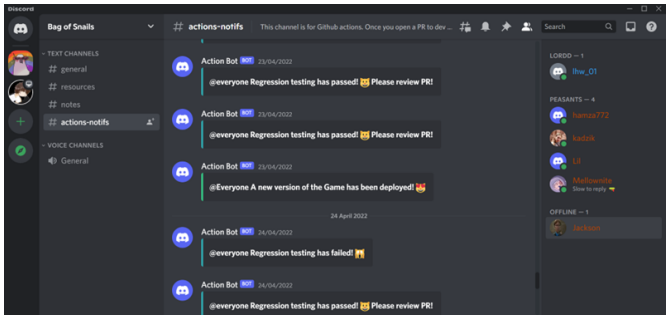


Figure 3: A screenshot of the team's Discord server

Out of all the different messaging applications we deliberately chose discord, due to the discord actions notification feature. Having discord on our phones and laptops, we were in constant contact to receive updates about the state of the deployment of the game. If a test failed or a new game was deployed, we were already in the space where we could start a conversation to decide the next steps.

5.3.2 MEETINGS IN TEAMS

We held our frequent meetings in person and online, depending on the members' availability. For the online space, we decided to use Teams. Not only is it a platform the members were already familiar with, it provided a space that accommodated all our needs. Firstly, we could hold several meetings at once, in the general and sub groups channels. Teams also has a mechanism to schedule the meetings. We utilised this very systematically, as whenever a meeting was agreed on the team manager would schedule it, preventing anybody from forgetting and missing out. Second of all, as opposed to discord which has a limit of sharing files only up to 8MB, we could share documents of all shapes and sizes and nicely organise them in the files section. In addition, we added our own personal sections. Trello can be integrated into teams, meaning that during calls, it was always at hand when needed. Another useful feature was the iterated OneNote class notebook.

5.3.3 PLANNING IN ONENOTE

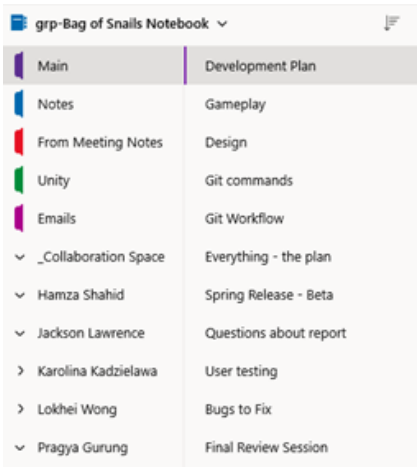


Figure 4: A screenshot of the team's Onenote notebook

The class notebook was a planning and organisation tool that was very important to create documentation and archive all our work related to the project. No matter the meeting circumstances, a team member was always tasked to take notes. If it was a planning or brainstorming meeting, it would hold all our ideas for

future referencing. If we were in a meeting receiving feedback, to not get lost in the overwhelming amount of useful information, two people would make notes. Whatever happened at any point during the development weeks, we made sure that there is a page in the notebook with detailed notes.

5.4 GITHUB

5.4.1 BRANCHING STRATEGY AND GIT WORKFLOW

The lead programmer had set up a git workflow and branching strategy. In the beginning stage of learning all the steps of our workflow we had it formalised in the shared notebook. Section 7.1.1 explains the workflow and branching strategy in detail. In short, we had one repository with a main branch and a dev branch. The main branch was reserved for deploying the game and only a tested version of the game was merged into the dev branch. We maintained the strategy of regular merges which meant we kept the number of merge conflicts down.

5.5 REFLECTION

Overall, the development process was smooth without any major problems. This is mainly due to how consistent we were with the structure of every sprint, from the time of the meeting to the person always responsible for updating the Trello board. Standouts in our team process include the notebook, which was useful in situations when we needed to refer back to past feedback or ideas. In the technical aspect of our organisation, having one person be mainly responsible for approving pull requests reduces mistakes and prevents merging in dirty code, as everybody had different standards for clean code, for example. After a few weeks, this difference disappeared, thanks to the uniform review of our programming lead.

5.5.1 WHAT DID NOT WORK

As mentioned in Section 5.4 we formalised a branching strategy from the beginning of our process. However not everybody was completely familiar with Git, leading to confusions and merging issues. As a result, the current programming lead wrote up git commands to teach others how to merge, create and use pull requests. Additionally, a games project requires certain uniqueness in the way git is utilised. This meant we had to revise our git approach when certain issues arose. For example, when working in a feature branch, one is supposed to make a copy of the scene and work there. Then, when they are ready to merge into dev, there won't be any scene conflicts and they can simply copy their work into the correct scene. Introducing and enforcing a uniform git strategy greatly reduced the time spent on merging work.

The already mentioned point system was also something we added in later in the development process. When it came to evaluating the individual contribution of members, we struggled to find a way to fairly assess the varying level of time and complexity of each task. As a solution, we ranked each task to gauge the significance of the contribution of an individual member. This spreadsheet was also a simpler way to visualise the amount of work being done. When a member's contribution for a sprint exceeded twenty, this was a signal to adjust the number of tasks they were taking on in the following sprint.

In week 3, we made the decision to change the programming lead. We assigned the team positions before we had started working as a team and were therefore unaware of the technical abilities of individual members. As none of the team members had previous experience with Unity, the first two weeks were dedicated to this. As such, the role of the lead programmer was not as prevalent. The programming lead was a very specialised position, in the sense that it involved the technical understanding of the tools, software and strategy at hand. After spending the following weeks learning to work as a team on the game, it emerged that Lokhei Wong was better suited for the role and the required work of a programming lead came to her naturally. Our original lead, Karolina Kadzielawa, acknowledged this and after their mutual agreement, the position was transferred.

5.5.2 POSSIBLE IMPROVEMENTS

We rarely branched into pair programming. But when we did, we noticed how helpful the process was. If a task is assigned to two people, there will always be a person available to help you. Instead, if every developer has their own task, they may not be able to find the time as they are already preoccupied.

We specialised very early on, and somehow a member ended up being in charge of the more difficult parts of the game. We didn't know the difficulty of a feature or

how important it would be to our game, so we couldn't foresee this happening. With hindsight, we could have put more thought into researching these details or get advice from more experienced teaching staff, to make more informed decisions on assigning tasks.

We held very long testing and debugging sessions where all team members stayed in a call and tested the game. Usually in these situations only one or two people were working on solving a bug. This meant the others were wasting time they could have used to work on their respective tasks. But as we needed more than four people to test the game, we could not see a different way to go about doing this.

6. INDIVIDUAL CONTRIBUTIONS

6.1 LINDA LOMENCIKOVA

TEAM MANAGER TASKS

- Kept team harmony, kept in contact with the composers and in other official communications.
- Kept the team organised by assigning tasks (together with lead programmer and design lead).
- Worked in close contact with the lead programmer and design lead to be always aware of the state of the team to make sure the team was on track.
- Was in charge of scheduling team meetings and speaking on behalf of the team.
- Coordinated and organised 2 iterations of user testing.

DESIGN TASKS

- Put together the visual concept for the kitchens.
- Designed kitchens layout, built the prototype kitchen layout.
- Created custom UI in canva, including user UI buttons and elements, main menu buttons, and minigame buttons.

CODING TASKS

- Created <BaseFood>, <Dishes> and <Ingredients> scriptable objects along with the databases + tickets logic (with Pragya Gurung).
- Implemented trays and serving logic so that order tickets would be associated with specific trays. The items on the trays would disappear along with the ticket when served. (With Pragya Gurung).

USER INTERFACE

- Was in charge of creating the user interface: Created the gameplay UI, UI camera, re-did all existing mini game canvases to be rendered by a UI camera.
- Continuously made aesthetic changes by adding in: skyboxes, trees, changing UI elements to fit with the game visuals.

MINIGAMES

- Created 2 minigames (Cutting board game, Sandwich game) which included making the design, UI, making the logic behind the game.and the logic behind the games).
- Created 2D particle systems for the cutting game.

PLAYABILITY

- Created the glow instructions for recipes, so on user clicking a specific dish, the appropriate appliance will glow white and the sub-room door will change to white. This included adding in Post-processing and making a custom glow shader.
- Made the cursors reactive: change when hovered oven clickable objects (both UI elements and in-game objects).

3D PARTICLE SYSTEMS

- Create the majority of particle systems in the game, namely smoke, heat haze, bubbles, smoke for smoke bomb, fire extinguisher, sprinklers.
- Implemented collision system to the appropriate particles.
- Created a random particle effect events system, which instantiates particle systems.

6.2 PRAGYA GURUNG

LEAD DESIGNER TASKS

- Created all the concept artworks for the game. This included the designs for the players, appliances, kitchens, ingredients, dishes and UI.
- Drew the final game logo.
- Modelled most of the custom assets using Maya and Blender. Ensured the models were kept low-poly but still stylized in accordance to aesthetics of the game.
- Produced all of the custom textures for the assets using Blenders 3D paint tool and Photoshop.
- Built the entire layout of the set using all the models.
- Created custom UI elements using procreate and canva. Once again this was designed in accordance with the game's colour scheme and aesthetics.
- Drew all of the ingredients, recipe cards and minigame sprites.
- Storyboarded and animated a 2D cutscene that plays before the start of the game. This cutscene tells the players about the backstory of the game and why the two kitchens are feuding.
- Created a custom toon shader with cell shading, rim-lighting and specular highlights.
- Made particle systems for the portal doors and fire. Most of the textures used for the particles were hand-drawn in photoshop.

PLAYABILITY

- Implemented the core logic in how players interact with interactable objects.
- Wrote the logic for how players pick up, drop, hold and slot items.
- Made a tutorial that shows players how the movement controls work.
- Created the popup instructions that the player sees when they load into the game. (with Jackson Lawrence) Then created hints and tips that appear throughout
- Added oven sabotaging logic. Players are able to add 10 fake seconds to their opponents oven timer.

GAME LOGIC

- Reconstructed the minigame and appliance classes to make use of OOP concepts. A child class of I*nteractable* called A*ppliance* would contain all the generic logic that all appliances would have and minigames would need. (with Hamza Shahid)
- Created *BaseFood*, *Dishes* and *Ingredients* scriptable objects along with the databases to contain all of the objects. (With Linda Lomenčíková)
- Implemented the ticket generation and tray serving logic so that order tickets would be associated with specific trays. The items on the trays would disappear along with the ticket when served. (With Linda Lomenčíková).
- Re-worked serving score calculations so that points are added/deducted depending how well the order was fulfilled. (with Jackson Lawrence)

OTHER

- Extensive debugging for the game.

6.3 LOKHEI WONG

LEAD PROGRAMMER TASKS

- Kept track of team progress and assigned tasks to team members (together with Team Manager and Design Lead).
- Set up and formalised our Git branching strategy, wrote useful git commands to help other members, and helped with merge conflicts at the start of the project.
- Led on continuous integration by setting up GameCI for automated testing and deployment, with a discord bot for automated notifications.
- Fixed any deployment issues.
- Encouraged good code design (OOP concepts), clean code and smooth workflow by reviewing pull requests, as well as testing new features and running code coverage reports to see what areas of code can be improved.

TESTING

- Acted as the core tester by writing unit and integration tests after features are implemented, to be run during GitHub actions workflow.
- Helped fix subsequent failed tests.
- Added regression tests for fixed bugs.

PLAYER CONTROLS AND CAMERA

- Executed multiple iterations of player controls in response to a combination of internal and external feedback.
- Added settings to control sensitivity of player movement and rotation.
- Changed player camera view multiple times, including adding a sub-camera with a culling mask to fix the problem of objects phasing through walls and changing layering of items when picked up and dropped.

PARTICLE SYSTEM

- Created a smoke particle system for the stove which plays when players start cooking on the relevant stove.

SPAWNING INGREDIENTS

- Implemented logic for spawning raw ingredients in the pantry/fridge/freezer.
- Created the prefabs for the ingredients.
- Added a limit to the number of each ingredient in both kitchens with replenishment at half time.
- Added a trash functionality to destroy items as a mechanism for sabotaging.

MUSIC AND AUDIO

- Communicated with music developers in order to implement dynamic music.
- Implemented music switching and cross-fading between scenes (kitchens, hallway, minigames, and menu canvas), switching to faster track near end of game, pseudo-random looping of arrays of tracks in both kitchens, and pitching in response to certain events in game.
- Added settings to control volume.
- Attempted to add distortion sound effects to voice chat (with Hamza Sahid)

DESIGN TASKS

- Fixed walls of kitchen scene and UI scaling.
- Came up with the overall design for visual instructions of the game.
- As a team, agreed on the design for the UI, kitchens and characters.

6.4 HAMZA SHAHID

NETWORKING

- Added Photon Unity Networking 2 (PUN2) and configured networking for multiplayer use of the game. This includes synchronising object/player positioning/movements as well as things like dish scores, animations, particle systems, as they all need to be synchronised over the network so are visible to all clients playing the game.
- Reworked the frying game made (by Karolina Kadzielawa), which involved flipping an item and catching it with a plate, to make it multiplayer. This involved enabling two users to join the game, and synchronising positions of the plate, flipped item, and frying pan. As well as allowing the flipped items to be stacked.
- Added name tags that appear above users that allow others to identify them, and synchronised these name tags over the network.
- Helped get rejoining game to work (with Jackson Lawrence)
- Helped to make the Play Again feature to work (with Jackson Lawrence).

VOICE CHAT

- Added WebRTC voice chat, with multiple "channels" that splits the game up into different sections, I.e. Kitchen One, Kitchen Two, Outside area, allowing members in a certain area to hear each other.
- Added speed test to identify what "band" should be assigned to the user for the voice chat. Each band has a different audio sample rate, bit rate and bit depth to suit the user's bandwidth.
- Attempted to add distorting voice effects to the voice chat (with Lokhei Wong).
- Created Oven Minigame.
- Added tooltip for hovering over dishes.

GAMEPLAY

- Added animations to the doors.
- Reworked player movements without use of the "character controller". Reworked the player movement so the player's rigid body's velocity is set.
- Added sound effect for players entering kitchens.
- Added a health bar that appears above users when being kicked out.
- Helped start PlayMode testing (with Lokhei Wong)
- Extensive debugging for the game.
- Created the initial and final kicking mechanism. The initial mechanism involved pressing a "kick" button simultaneously to kick a player. I later reworked this so that to kick a player you need to physically push them out of the kitchen.

AI

- Added in AI waiters and owners:
  - Waiters: Collect orders from specific trays and serve to relevant customers seated in the kitchen.
  - Owner: Visits the kitchen at a specific time, analyses team and individual player performance, and acts accordingly. The Owner is also able to help out users in certain tasks when necessary.
  - Both Waiters and the Owners navigate using Unity's NavMesh pathfinding and navigation mechanism.

GAME LOGIC

- Reconstructed the minigame and appliance classes to make use of OOP concepts. A child class of I*nteractable* called A*ppliance* would contain all the generic logic that all appliances would have and minigames would need. (withPragya Gurung)

6.5 JACKSON LAWRENCE

GAMEPLAY

- Added scripts and colliders to each appliance to allow players to slot ingredients/dishes into the appropriate appliances by walking into them.
- Created a global timer and team's point system which utilises Photon's Custom Server properties to synchronise their values across the network.
- Created the popup instructions that the player sees when they load into the game. (with Pragya Gurung).
- Created a custom game over scene with animations showing the winning team and displaying player statistics gathered from the game.
- Implemented a Play Again button at game over which resets all relevant game assets so that the game can be played again smoothly.
- Made throwable smoke bombs detonating after set time used for sabotaging and irrating the enemy team, whilst only being useable inside an enemy kitchen.

MINIGAMES

- Created the stove minigame where the player must move a pot left and right whilst catching the correct ingredients and avoiding incorrect ones.
- Doubled the points of all minigames when the user tries to cook in an enemy kitchen to incentivise cooking in their kitchen.

AESTHETICS

- Hooked up all player animations with Mixamo. These were then synchronised across the network using Photon. Also animated custom canvas transition animations using Unity's animator.
- Added loading screens in all relevant places with an appropriate loading bar which shows actual progress of loading.
- Helped create some of the assets used in the game using Maya (with Pragya Gurung). This includes some ingredients, utensils and the fire extinguisher.

PHOTON

- Hooked up all the lobbies and rooms within the network with Photon where any player can find public lobbies from a list on the server. Also added settings to the lobby menu where the master-client can edit the timer of the game before starting.
- Added ability to change teams within a lobby, changing their custom property to be used throughout the game. This also included auto-balancing teams so it was not unfairly matched.
- Helped create and test RPCs within the game for syncing items/values across the network, such as the tray, points and items (with Hamza Shahid).
- Added a rejoining feature where a player can rejoin a game after previously leaving/disconnecting from the game. This enables them to rejoin in the middle of a game whilst being seamless and smooth (with Hamza Shahid).
- Tracked player statistics like amount of cooked dishes so they can be displayed at game over by using Photon's custom properties.

OTHER TASKS

- Researched and proposed design on database for game.
- Manually deployed our game to GitHub pages for prototypes.
- Edited the video and created timelapses of building assets in Maya.

6.6 KAROLINA KADZIELAWA

Contributed to the initial movement scripts and the mechanism for the players to pick up items:

- Using keys to move with the player and camera turning in the direction of movement
- Holding the items in a slot front of the player and dropping them on mouse click (with Lokhei Wong)

CO-CREATED THE INITIAL COOKING LOGIC

- putting and removing ingredients from the stove – the stove logic recognizes the ingredients in order to prepare the right dish
- cooked dishes spawning – the right dish needs to be instantiated when the stove minigame finishes
- updating the points of the dish – the newly instantiated dish has a points attribute that gets updated with the minigame score when it's finished.
- Introduced the event system for assigning dish points – the event system handles the point logic for all the minigames on all the appliances. This reduces the dependencies among classes necessary to perform this operation.
- The cooking logic was later heavily modified to allow for different cooking mechanisms and the use of different kinds of appliances.

FRYING GAMES

Co-created the frying minigame. The minigame involves complex controls and is the only multiplayer minigame in the game (with Hamza Shahid):

- One object is controlled by mouse movement and performs actions depending on the average mouse movement speed at certain intervals.
- The food sprites in the minigame are selected based on the dish being cooked
- The throwing and flight of the items is made realistic using 2D physics
- Depending on the order the players enter the minigame in, they get assigned different objects to play with
- The minigame score is being updated by both players playing the minigame.

ADDED SOUND EFFECTS TO THE GAME

- The sounds can be heard when certain events occur, such as dropping items, the oven catching fire, or doors opening. Other sounds are persistent in the background, like the talking noise coming from the restaurants.
- The sounds are 3D and depending on the context can be heard by either everyone in a certain radius thanks to Photon networking, or by one player only for the sounds inside minigames.
- Some of the sounds are randomised: when an action is performed repeatedly, the player will hear noises of different pitch and volume, as well as different clips.

PARTICLE EFFECTS

- Developed a mist effect for the fridge and freezer. (Not present in the current version of the game)

CO-CREATED THE VIDEO (with Jackson Lawrence)

- Wrote the script.

7. SOFTWARE, TOOLS, AND DEVELOPMENT

7.1 DEVELOPMENT AND TESTING

7.1.1 GIT AND OUR BRANCHING STRATEGY

Throughout the course of the project, we used a wide range of tools to aid our development process. First and foremost, we used GitHub to host our code for easy version control and collaboration. It was therefore important that we have a branching strategy that would enhance our productivity. After thorough research into the different strategies employed in industry, we all agreed to work with the Git Flow branching strategy. As illustrated in Figure 5, Git Flow is based on 2 major branches: the development branch and the master branch. The development branch contains our pre-production code and as soon as a new feature is implemented or bugs fixed, it is merged into the development branch via a pull request (PR). This branch would then be merged into a testing branch for testing. When our code in the development branch reaches a stable point with no detected bugs, it is released into the master branch, which is automatically deployed to our production environment. We use a variety of supporting branches:

- Release branch for clean up and versioning when a release is due
- Feature branches for developing new features/tests
- Hotfix branches for immediate reactions on critical issues/bugs in the master branch
- Additional bugfix branches for handling other bug fixes

Figure 5: Diagram of the Git Flow branching strategy

To complement our workflow, we created branch protection rules for our development branch. This includes requiring a PR and approval before a branch can be merged in. Pull requests provide an opportunity for code review, providing a place for team members to make suggestions on features and ensuring that they agree with the proposed features. The code reviewer will also be able to check that the logic is sound and that the code is sufficiently clean. PRs help with code stability, as all the automated tests are executed against the codebase, to ensure new code doesn't break old features and that old bugs don't reappear.

In order to make code review more efficient and to streamline our workflow, we set up a discord bot using GitHub Actions, in conjunction with our CI/CD system [7.2]. Consequently, team members are instantly notified about the test results, allowing for more efficient integration as we know at the earliest instance when the PR is ready for review, or when amendments are required, preventing a backlog of PRs waiting to be reviewed.

7.1.2 GAMECI AND TESTING

In order to deliver code frequently and reliably, we decided very early on to set up a CI/CD pipeline. Together with our agile methodologies [5.1], this greatly streamlined our software development cycle, resulting in faster delivery. A large part of our CI/CD pipeline relies on GameCI, a system specifically targeting game projects. We used GameCI along with GitHub Actions to automate both our testing and deployment.

As discussed in Section 7.1, when a pull request is made to the development branch, all tests will be automatically run. These tests consist of both unit tests and integration tests, using edit mode and play mode tests from the Unity Test Framework package respectively. These are created during our sprints [5.1], where our tester writes tests for new functionality added. This ensures that our logic is sound and we are able to catch bugs that would perhaps not be obvious during manual testing, as well as ensuring future features will not break older code.

However, due to the extensive codespace of our project and the limited time for the project, it would not be feasible to write tests to cover every bug. In any case, we made sure to write robust tests for the logic core to our game, such as the ordering and serving mechanism [7.3], ensuring that any bug would not be detrimental to our game. In terms of statistics, our code coverage

exceeds 70%, where some researchers have claimed that the correlation between code coverage and software quality starts weakening after reaching 70%[2]. Additionally, we held regular testing sessions to find bugs that were not covered in our testing suite. By conducting this testing process very early on in our project, we were able to prevent the build up of bugs, which made it easier to detect the source of the bug. These would be fixed and subsequently, regression tests would be written to prevent the same bug reappearing.

Additionally, we utilise GameCI to automate the deployment of our system. This proved to be time efficient as a member of our team did not need to sit and wait for our system to finish manually building. It also contributed in reducing the risk of broken code being pushed to production, as the build would need to successfully complete before it is automatically pushed to our secondary repository. The combination of all of the above greatly decreases the chances of major bugs seeping into our deployed system.

Although our combination of automated and manual testing greatly increases the robustness of our system, it is not enough. Some elements simply cannot be tested through these means, such as the playability of our game, and the look and feel. As such, in addition to 3 panel sessions where university staff tested our game, we also received constructive feedback by holding 2 user testing sessions. In these sessions, we invited other computer science and music students to test our game, where we would receive feedback in an interview with the individual testers. This was invaluable to our game, as we were able to cater to both experienced and inexperienced gamers.

7.1.3 Adding New Logic

With an extensive codebase, very naturally, members of the team would not be able to be familiar with all the logic. Resultantly, it may be hard to extend functionality. To prevent this, we aimed to write well documented code throughout the project. This involved naming classes, methods and variables intuitively. In the case that these names were unable to succinctly describe the intent, or in instances of ambiguity or complex code, we would supplement this with comments to aid team members in quickly familiarising themselves with the logic. We also made sure that all our scripts were well-organised into folders, making it easy for our team to find the relevant code.

In order to write high-quality code, we adhered to Object-Oriented Programming (OOP) concepts - most notably the 4 fundamental concepts: inheritance, encapsulation, polymorphism, and data abstraction. Taking this one step further, we used code coverage metrics to evaluate our code periodically, which provided metrics for both the quality and the complexity of different methods. This improved our code quality as we sought to improve the score, and at the same time, made it easier for other team members to understand our code, as we simplified and reduced the complexity of convoluted code.

With these steps added, the ease of adding code ultimately lies on the feature being added. Features which are interlinked with many different parts of the codebase would require a wider understanding of the implementation of our software. In these instances, we would pair program so that the pair of programmers would be able to quickly fill gaps in their knowledge regarding the way the code works.

7.1.4 Other Software Tools

As well as the aforementioned tools to aid us in our development process, there were also a number of other libraries we utilised to help with the implementation of certain features. Some tools which were especially helpful were Photon [8.1], which managed state across two clients, and the Agora Unity WebGL Plugin [8.1], which saved us from implementing our own signalling server. In order to increase the immersive nature of our game, we also made use of the in-built unity particle system to simulate certain kitchen particle effects [8.7].

7.2 Software Developed

Arguably, the most notable software developed was our ordering and serving system, which was relatively complex. This mechanism was core to our game, as it is the way in which players know what to cook and how they receive points. The complexity of this system can largely be attributed to the large number of aspects it

[2] Mechelle Gittens, Keri Romanufa, David Godwin, and Jason Racicot. 2006. All code coverage is not created equal: a case study in prioritized code coverage. https://doi.org/10.1145/1188966.1188981

interleaves with, including the dish and ingredient databases, tray assignment, points system, minigames, ticket generation, serving etc. [8.4.4]. On top of all this, a lot of this logic had to be synced over the network in order for players to cooperate well within their team. This was an area which we struggled with, as there were a lot of desyncing issues to be overcome, some of which could only be spotted with rigorous manual testing.

Additionally, we manipulated javascript files from the Agora Unity WebGL Plugin [8.2] to edit certain functionality. This included altering different audio properties such as the bitrate and disabling the camera popup [8.2.1]. This made our game more accessible to players with poor connection and solves a major privacy issue for our game, as our game does not utilise any kind of camera functionality.

7.3 DESIGN SOFTWARE AND TOOLS

Modelling, texturing, rigging and animation assets for the project could have easily consumed a lot of time and energy. Especially since we created so many. Therefore, we aimed to streamline the process by using our tools in a way to increase efficiency. For the 3D assets, Maya and Blender were used for various reasons. Maya was a program our lead designer was the most familiar with for modelling. Whilst Blender offered a more efficient way to UV unwrap, bake and create custom textures. In addition, there were many more tutorials and guides online for Blender in comparison to Maya, as the program is free.

All of the texture images were hand drawn using Blender's built-in 3D paint tool. Some of those images, such as the ones for the character models, were then brought over to photoshop for cleaning and adding further details. All of the models were stylised in a way to adhere to the cartoony look of the game. The colours are rich in saturation and shading was kept to a minimum. In turn, this style added more interest to the models which greatly helped as they were so low-poly.

Adobe's Mixamo was used to rig and create animations for the character models. This tool offered a large library of humanoid animations free for use. Using this tool greatly reduced the time it would have taken to rig and animate in-house.

8. TECHNICAL CONTENT

8.1 PHOTON

Adding networking in our game complicated things which would otherwise be remarkably simple to implement had the game been single player. To offer a smooth multiplayer experience, we had to ensure we chose a suitable solution for our game. We decided to use the Photon Unity Networking Engine (PUN) package which allowed us to focus on networking at a high level rather than needing to deal with data packets.

8.1.1 SYNCHRONISING OBJECTS

We synchronised properties of the game objects, such as their position and rotation using Photon Transform Views. We made use of Remote Procedure Calls (RPC) calls and "Room Properties" to synchronise the state of players, objects and various other properties that were a part of our game, such as appliances, dishes, points, timers etc. Furthermore, animation sequences were synchronised where each animation was played through a Boolean value. These values were then synced using Photon's Animator View which allowed each animation to be played at the same time for everyone when appropriate.

8.1.2 RPC CALLS

When a function needs to be called, or a value needs to be synced across the network we had to utilise RPCs, through Photon. This required a Photon View (which is a unique identifier for an object), an action to be broadcasted and a target (e.g. everyone, or everyone but yourself). Then, you could optionally pass a parameter and with a specialised tag above the function [PunRPC], it would be recognized as an RPC function by Photon. Since RPCs are sent over the network, and might not be as consistent for users with a weaker connection, we made sure to keep RPCs to a minimum. Instead of using multiple RPCs for multiple things at a time, it was more optimal to group all the functions into one RPC. Also, we are only able to pass primitive data types through RPCs, so things like GameObjects in Unity could not be sent - a workaround was to pass a ViewID of the relevant GameObjects and the client could then receive the object from the specified ViewID.

8.1.3 SERVERS AND LOBBIES

Pun also allows for the use of creating, finding, and joining private or public rooms to encapsulate gameplay for a certain number of players. This meant we could run multiple different game instances in tandem without them affecting each other. The "Master Client," which is the creator of the lobby, owns most of the networked objects. This way we could set a lot of global values on the server through the master client, such as the decrementing timer value or the order tickets for each kitchen.

8.1.4 CUSTOM PROPERTIES

We utilised Photon's Server Custom Properties and Player Custom Properties to abstractly sync hidden values across the network. For example, the server's timer (set by the master client) was set as a Custom Property for the server which could then be fetched by a client when the game starts, or on rejoining. They also helped store the players' teams, which was incredibly useful as this needed to be checked frequently. Player Properties were also used to store and showcase player achievements in the gameover scene. These achievements were statistics, such as the "Player who cooked the most dishes".

8.1.5 RE-JOINING THE GAME

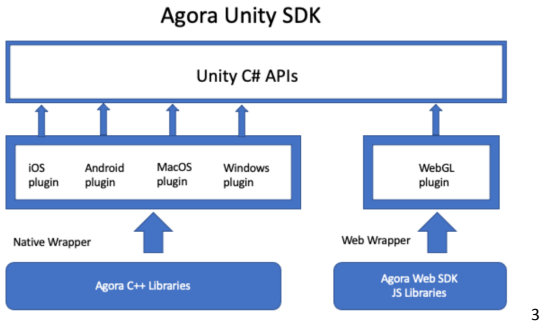
We aimed to make our game robust and usable for players with weaker connections. This was done by allowing disconnected players to rejoin half-way through a game. Implementing this was a difficult task as we needed to ensure that reloading the game felt seamless.

First, we needed to detect whether the player was in a disconnected or "closed" game state. This was done by utilising Unity's PlayerPrefs which allowed us to store data locally, similar to how cookies work on a browser. Before asking a player whether they wanted to rejoin, we needed to check whether the game still existed. Photon has no built-in way to detect this. Therefore, our best option to detect this was by attempting to create a lobby with the same name as the one the player just left. If this new lobby cannot be made, then the previous game still exists. We then needed to change all the relevant RPCs to be PUN's buffered RPCs. These buffered RPCs are stored on a server and called on the user rejoining, which enables us to store relevant game data while offline. However, these buffered RPCs had to be used sparingly as there was a limit to how many you could call, especially after being disconnected for a long time. Lastly, things that were set on the server's Custom Properties could be fetched and updated accordingly without any extraneous steps.

Though we successfully created a rejoining system using buffered RPCs, this solution was by no means the most efficient approach. In hindsight, a better solution would have been to have a single GameObject that holds all the key information about the current game's state. This information would then be copied over to anyone who needs it. Having all this information in one place would have been far easier to maintain and expand upon. In comparison, implementing a system where all the components sync themselves is significantly harder to test completely.

8.2 VOICE CHAT

Communication and coordination are amongst the core principles of our game. Hence, we needed to ensure we were able to provide players with a reliable way to communicate within the game. Our initial plan was to use Photon Voice, which contains a multitude of features that would provide us with exactly what we needed when implementing voice chat in our game. However, requiring the game to be deployed on WebGL created quite a few complications that otherwise would not have been present. There are many packages that offer the ability to incorporate voice chat in a Unity game, however, the vast majority of the free options available do not officially support WebGL compatibility, including Photon Voice.



[3]

Figure 6: How the the WebGL plugin is used for Agora's Web SDK and Unity SDK to communicate

[3] shorturl.at/julX4

Another package that offers the ability to implement voice chat in Unity is Agora. Similar to Photon Voice, Agora by default does not offer a WebGL SDK that would allow us to implement voice chat in our WebGL game. Agora does, however, have a "WebSDK", that is a JavaScript Library which uses APIs in the web browser to establish connections and control communication and live broadcast services. A solution to allow Agora's WebSDK to work on WebGL would be to create a WebGL plugin. This plugin would be a wrapper library that uses Agora's WebSDK, and it would enable us to use WebRTC features with Agora's SDK. After some research, we were able to find an open source project that did exactly this, and were able to use and refactor this plugin to fit in with our game. The structure for how the plugin works with Unity and Agora is illustrated in Figure 6. As can be seen in the diagram, the WebGL plugin acts as a bridge between the Agora WebSDK and Unity.

8.2.1 SPEED TEST, SECURITY AND PRIVACY

As vital as voice chat is to our game, we also needed to ensure that this would not hinder the performance or negatively impact the player's experience. We therefore consider the player's bandwidth. At the beginning of the game, we send a HTTP request to the server hosting the game, and calculate the internet speed by dividing the length of the data downloaded in bytes with the time taken. This was then converted into Megabits per second. We added multiple "bands" for users, which allocates the bitrate, bit depth, sample size and sample rate for the audio received on the user's end. This would be done based on their internet speed. Essentially, players with a low bandwidth, will receive lower quality audio which saves bandwidth usage. We carried out multiple tests to get the appropriate values for the audio properties mentioned - to find the right bandwidth/audio quality tradeoff.

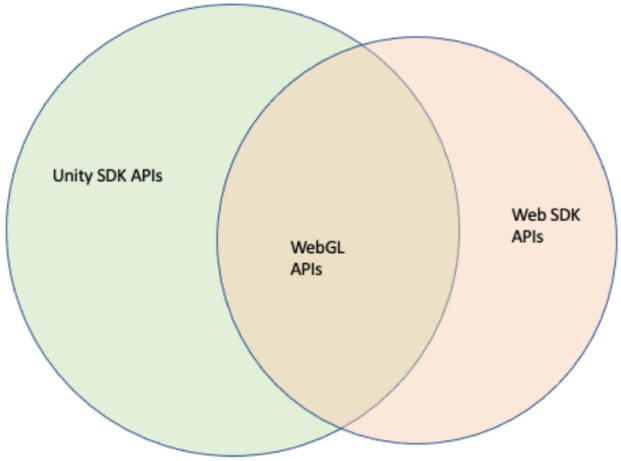
In terms of security, we wanted to ensure appropriate measures were in place to give users a secure experience. Since the WebGL plugin is a wrapper for Agora's WebSDK, we are assured that Agora's security measures are in place. Agora has incorporated "Channel Separation", which means that there is an independent and isolated channel for each audio, video, or messaging data transmission. Additionally, it means that all channels are logically separated and only users authenticated from the same App ID can join the channel. We also name our channels by creating a one-time random string. Doing so adds an additional layer of security.

Finally, the WebGL plugin originally asked for both audio and camera permission from the user. However, since we are not using any user video in our game, we disabled this prompt to ensure users were strictly allowing microphone permission only.

8.2.2 TECHNICAL DIFFICULTIES

As communication between JavaScript and Unity was required for our WebRTC voice chat to work, we faced many challenges. Specifically, when it came to manipulating the plugin to add or amend features to fit our game.

There were a lot of limitations to the features that we could add to our voice chat, since we were restricted by the features available in the Agora WebSDK. In addition, not all of the features available in the Agora WebSDK could be mapped to the WebGL plugin. This is due to the WebGL plugin also being limited by Agora's UnitySDK. The diagram below illustrates the overlap between the Agora WebSDK and the UnitySDK (features that can be implemented in the WebGL plugin).



[4]

Figure 7: Overlap between Unity SDK and Agora Web SDK

[4] shorturl.at/julX4

8.3. CONTROLS, CAMERAS AND MOVEMENT

8.3.1 CAMERAS

We had two main types of cameras: one which controlled the actual world camera, with player movement, and the other which was used for the minigames and 2D particles. This was a suitable solution because it was less cumbersome changing the camera between two different areas, compared to any other options available. The UI camera also scaled up all the canvas elements to aid with scaling to relevant resolutions.

One issue we encountered was that items in the player's hand would phase through the wall. After thorough research into the possible different solutions, we considered two final options. The first was to create a custom shader. Objects with this shader would be the last item rendered by the camera, and players would therefore be unable to see objects clipping through walls. However, this caused problems with items being seen through walls and counters, so the material of the object would need to be switched back when dropping the item. The second option we considered was to add a sub-camera to the player prefab. As can be seen in Figure 8, this sub-camera would have a culling mask set to pickable, and so would only be able to see items with the pickable layer. Conversely, we set the main camera to see everything but the pickable layer. Items being picked up would then be switched to the pickable layer, and switched back to default when dropping the item, or when another player takes the object off your hands.

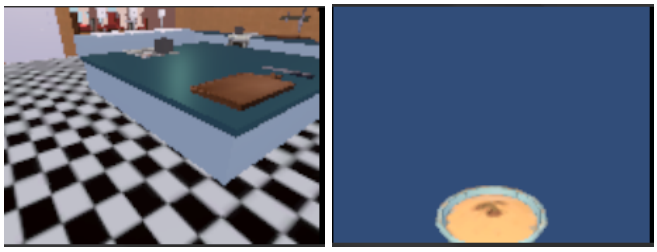


Figure 8: main camera versus sub-camera view of held item

After considering both options, we decided that the second option would be better. At this stage, we wanted to incorporate a toon shader [8.8.1] into our models, and would therefore have to combine the shaders if we had picked the first option. The second option on the other hand, had an additional advantage, whereby the held item would stay in the same position on the player's screen. This was a suitable way to implement the panel feedback [7.1.2] we received, where the players wanted a way to see the held item at all times.

8.3.2 RIGIDBODY MOVEMENT

Movement was created with the help of Unity's RigidBody which were controlled by the WASD or Arrow keys. Using RigidBody, we had 3 options to implement movement: velocity, movePosition or addForce. We used velocity primarily because it detects collisions accurately, and from early testing, players were phasing through walls as collisions weren't being detected. It also keeps the player speed constant. In comparison, velocity calculates the speed from the physics denoted by the RigidBody, so it does not need to do any calculation internally. We did not use addForce for the movement as it is more suited for acceleration, or jolting type movement. As such, addForce instead was used when throwing a smoke bomb. We also chose not to use movePosition because our player was non-kinematic. If we did, the RigidBody would be calculating position driven by the physics simulation, as well as calculating its position internally, causing jittery and erratic movement. By accessing the vertical and horizontal axis of the player, you could check where the player wanted to move, as well as considering the camera rotation. Movement would then be driven in the direction of the camera.

8.3.3 ONLINE MOVEMENT

We use Photon to synchronise player's movements on the network by adding a Photon Transform View to the player when spawned in. This automatically syncs the transform values when applied to an object so that each player can see it being synced over the network.

8.3.4 OPTIMISATIONS

Our control scheme went through multiple iterations due to a combination of both internal and external testing. Testing solely with the team was not enough because we might have become accustomed to the controls, and not know what a new player would feel like using them. Our player controls started off with 'a' and 'd' controls to solely rotate the camera (and 'w' and 's' to move the player). In a subsequent iteration, you had access to vertical movement by holding down the right click to look up and down.

We understood the controls were getting in the way of the frantic nature of the game, as players struggled with the movement, and so we needed to find a control scheme which was a good balance for both people used to games, and those who aren't. Combining all our feedback, we eventually settled with a control scheme, where players hold down right click to look around everywhere, with 'a' and 'd' to strafe. Users loved this style because it aided players in completing their tasks as fast as they could, no longer hindering them from trying to complete a task because of the awkward controls. You can also edit your local rotation speed (looking around with right click) and movement speed to allow players more accessibility. We also considered using mouse movement without any clicking to control the camera view, as in many popular first-person shooter games, however, this was not suitable for our game due to our UI system and the required frequent use of our game.

8.3.4 CONTROL-DRIVEN ANIMATIONS

All player movement animations are based on your controls, so will detect your movement vectors and set a Boolean value to True in the Animator to tell Unity to start playing the appropriate animation. This includes, forward, backward, left, and right strafe animations (for movement). When stopping movement these values are then set back to False which signals the animator to stop playing the animation. The Boolean values handled by the Animator can then be synced on the Network using Photon's Animator view which will (when set to discrete) be synchronised across the network.

8.4 GAME SETUP

8.4.1 NAVIGATION

As our game contained a lot of features that may not be obvious for a new player, we needed to let the players know of the different functionality and the method in which to use them. During our user testing phase, some players stated that learning the controls was confusing without guidance and that they weren't aware of how to cook and sabotage.

To resolve the first issue, a tutorial animation is played before the game loads that clearly shows how the movement controls work. This is ideal as we did not want players to be learning how to move after the game and the timer has already started.

As for the second issue, an instruction panel was added to the bottom of the UI. Once the game starts, instructions are shown to help lead the players through the process of making and serving their first order. Yet again, this is ideal because even though players need to read instructions to know what to do, they are still actively playing the game and earning points. After the initial tutorial, the instruction panel becomes a place to provide hints and tips for the player. In addition, we added glow instructions using an outline shader [8.8.2]. When the player clicks on a dish on the order ticket, its recipe card will appear on the top right of the screen. Along with that, the doors to the rooms that hold the ingredients and the appliances that are needed to cook the dish, start to glow.

Figure 9: Recipe card, instruction panel and glowing

8.4.2 INTERACTABLE SYSTEM

As stated before, our game will have a lot of features players can interact with. To ensure that interaction logic is kept uniform and easy to expand upon, we created a parent class called *Interactable*. This class along with the *PlayerHolding* and *PlayerController* contained the logic to allow players to focus on objects. This parent class contains a function called *Interact()*, which is meant to be overwritten by its children. Doing so makes use of one the core concepts of OOP, polymorphism. One example of this is the *Appliance* class. On Interact(), if the player is holding something, it will slot it to the appliance slots, otherwise, it will attempt to cook a dish. Another example is the *PickableItem* class, which on *Interact()*, gets picked up or dropped by the player. These classes make it very easy to integrate further objects that the player can interact with into the game. For instance, an object that was added late into the game were the fire extinguishers, however, the process of implementing it was very simple as the majority of the logic was already there.

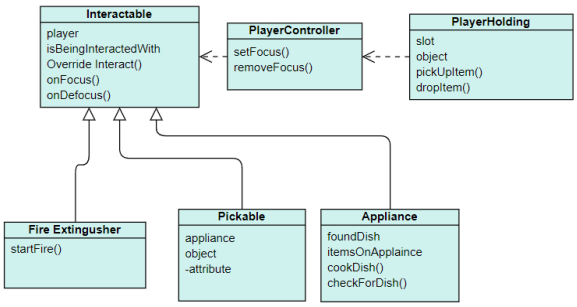


Figure 10: UML diagram; interactable system

8.4.3 MINIGAME MANAGEMENT

To cook in our game, the players will need to place the correct ingredients on the appliance and play a short minigame. As each appliance type has its own 2D minigame, the logic behind how to start cooking a dish is essentially the same for each one. Quickly we realised that code would be repeated, so we made use of OOP concepts to reconstruct the classes. As mentioned before, we created a class *Appliance* which inherits from the *Interactable* class. It was then turned into a general class that holds all the information every minigame would need - information such as the dish recipe, ingredients and player.

In addition, a general minigame class could have been created; however, as the logic behind each minigame was different, this would not have been suitable in this instance. This was especially true for our frying minigame, which required two players instead of one. As we only have one multiplayer minigame, we did not see the need to create a separate subclass to distinguish the two types. However, had we created more multiplayer minigames, this distinction would have greatly helped to keep the code clean and easy to manage.

The way the general class *Appliance* works is as follows. If the player clicks on the appliance empty handed, the *CookDish()* function is called. In there, it will attempt to find a dish that is made from the items slotted onto the appliance. Once a dish is found, the attributes of that dish are checked to make sure the appliance is of the correct type. Each dishSO has a “toCook” variable which correlates to an appliance tag. After everything is checked, the minigame canvas, which is passed in through the inspector to the appliance script, is set active. The cutting board has the cutting minigame canvas, the sandwich station has the sandwich minigame canvas etc.

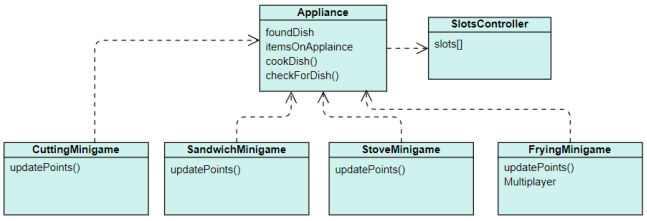


Figure 11: UML diagram; How appliances connect to the minigames

Another generalised class that works alongside *Appliance* is *SlotsController* which controls the slotting and unslotting of items to a parent object. Once again, this generalisation ensures that if we were to add more minigames or appliances in the future, most of the logic would already be there to easily integrate the new functionality into the game.

8.4.4 ORDER AND SERVING SYSTEM

Scriptable objects (SO) were used heavily for the creation of dishes, ingredients and trays. A BaseFood SO was the parent SO of the IngredientSO and DishSO, which would inherit from it. Dishes cannot be slotted onto appliances, they have their own dish IDs which is a combination of the ingredient IDs, and they contain information about the recipe. In comparison, ingredients contain their own ingredient IDs and the location of where they are stored (pantry, fridge or freezer). However, both SO’s need a sprite image and should be able to be served on a tray (for both serving and sabotaging). This means that trays should allow *BaseFoods* to slot onto it, and not just IngredientSO’s.

Orders are also a scriptable object made up of an order ID, order number and a list of *BaseFood’s*. It needs to be a list of BaseFood instead of dishes so that the list can later be compared to the list on the trays when it is served.

When a new order ticket is generated by the master client, an RPC function is used to synchronise its values to everyone else. The ticket is also linked to a specific tray and its order stand number. Implementing it this way ensured that desyncing issues were kept to a minimum.

8.5 SABOTAGING AND UNIQUENESS

Our game makes use of several common cooking game tropes. We do however implement features that set it apart from other positions in the genre.

8.5.1 SABOTAGING, OPPONENT TEAM

The rivalry between two kitchens is at the very core of our story and gameplay. The ability to engage in sabotaging action makes this aspect of the game unique and innovative. As described in [4.4], players can sabotage and hinder the opposing teams in a variety of ways. For instance, throwing a smoke bomb to obscure vision, stealing their ingredients and dishes, or even changing the timer on their ovens (which could sset off fires). However, players can defend their kitchen by removing intruders. They are notified when a member of the competing team enters their kitchen through sound effects and by repeatedly clicking on the intruder, they can attempt to kick them out. The sabotaging aspect helps our game feel more competitive and ensures a need to interact with the opposing team. Furthermore, you can attempt to cook in the opposing kitchen and earn double points. This gives further incentive to go to the other kitchen.

8.5.2 2D MINI GAMES

There are some cooking games where players need to navigate a chaotic kitchen to collect ingredients and cook, whilst others only focus on their 2D minigames. In our game, we combine both of these elements to create the best of both worlds. To cook a dish, a 2D minigame must be completed. There are four 2D minigames and some have features such as 2D particle effects and randomised sound effects. Each minigame has their own mechanism for calculating the score, but we made sure to normalise the maximum score for all. The score earned then gets assigned to the cooked dish. The number of points gained heavily depends on how well the minigames are completed. Players will need to balance prioritising speed or quality depending on the game state. In addition, the frying minigame can only be completed when played by 2 players, forcing teammates to communicate. However, dishes made using this minigame are worth more points. Once again this adds to the possible strategic moves players could make to defeat their opponents.

8.5.3 LESS RESTRICTIVE GAMEPLAY - COOKING OR SABOTAGING

Unlike other cooking games, the players have the freedom to organise their gameplay as they desire. At no point in the game are they forced to perform a certain sequence of actions. Whether they decide to cook their own dishes, help their teammates, or visit the enemy kitchen and sabotage is entirely up to them. Players are not restricted to one form of gameplay. Moreover, allowing different styles of gameplay help our game feel more chaotic and add to the social atmosphere. While new players are provided with guidance in the form of a tutorial as well as cooking instructions, more experienced players can choose to forgo those instructions and experiment. A fast-paced, highly competitive gameplay is encouraged, but using the game as just an environment to virtually hang out with friends is just as valid. Even when the players opt for the former option, they still can employ whatever strategy they see fit - they can cook dishes in whatever order they feel like, they can spend the whole game sabotaging, or not sabotage at all. This freedom is often missing from similar games on the market, where players are restricted to performing actions in a set order.

8.5.4 VOICE-CHAT INTERACTION

When leaving your kitchen you can no longer speak with your teammates, therefore it adds another layer to communication in the game. The freedom to chat with anyone in the game instead of just your team, adds to the competitive atmosphere. Some may choose this opportunity to trash-talk, whilst others would quietly spy in on the opponents' conversation.

8.5.5 REPLAYABILITY

After playing the game for months, we did not tire of the fundamental gameplay. At its core, we feel our game is very good for replayability. This point is strengthened by the variety of different actions/events that could happen in a game, thus making no two games similar. Even after our panel reviews and two user testing sessions, the core gameplay of cooking dishes and sabotaging did not change, and there were not many suggestions to add to it. There is no narrative focus other than a basic backstory to set the scene and it is not level based where it progressively gets more difficult; it is purely a competitive game that can be played in many different ways.

8.6 AI

To enhance user experience and make the game livelier, we added AI (Artificial Intelligence) waiters and owners. These waiters and owners are there to simulate a more realistic restaurant environment, so users feel more connected to the game as opposed to creating dishes that disappear when you serve them.

For our AI to work, we took some time to figure out the best pathfinding and navigation method for our agents. We considered using a couple of different approaches, including creating our own navigation system from scratch. Initially, we deliberated over taking a reinforcement learning approach, where our AI would be rewarded positively for certain actions and negatively for others. However, we found that this would not be a suitable approach as the purpose of AI in our game was not to carry out extremely complex tasks but to contribute to the atmosphere of the game. This is not favourable when using reinforcement learning as there is a lot of cost and time involved when training the agents to carry out desired tasks. We had to evaluate whether this cost and time was worth sacrificing for the actions we required from our AI. Using reinforcement learning to educate our agents about navigating through our game would be quite inefficient and would not be the best approach, due to the brute-force-like nature of training in reinforcement learning.

Alternatively, we could use a pathfinding algorithm, such as Dijkstra's algorithm or the better variant, A*. This would be a much more suitable approach, as it caters more to the type of navigation that would be required by our AI agents. However, creating our own pathfinding mechanism from scratch would be very time consuming. We decided to add AI into our game late into development, as part of additional functionality in order to improve the feel of the game. As a result, we had to consider the necessity of implementing the pathfinding algorithm, over other refinements.

Eventually, we decided to use Unity's built in NavMesh system which allowed us to handle agent navigation. There are many features of Unity's NavMesh that made it an attractive choice for us, such as:

- Being able to easily "bake" a NavMesh surface (including during run-time), either by using Physics colliders or Mesh Renderers present in our game, which defines the area that can be travelled by the agents.
- Bake NavMesh based on sorting layers; this is particularly useful if we are designing our NavMesh to be baked around mesh renderers but want to exclude certain objects, such as doors.
- Carve obstacles at runtime, which allows us to define what should and should not be avoided by the agent.

Using these features alongside the models for our agents, we were able to define the precise region we wanted our agents to be able to travel through. Each Agent used two scripts as well as a Unity NavMesh agent to handle the AI aspect of our agents:

- Agent Script: Specific to each type of agent (Waiter or Owner). This script handles the logic for all the tasks that are to be carried out by the agent.
- Agent Movement Script: Handles movement animations of the agent, to ensure the agent is playing the correct animation at the correct time when navigating through our game.

8.6.1 WAITERS

The core purpose of waiters in our game is strongly based off a real-life scenario. Waiters are instantiated in the restaurant part of our building. They collect orders that are served by players and serve customers that are currently awaiting an order. We created a simple algorithm that assigns an agent to the tray that has been served, as well as assigning a destination table to the agent that is collecting an order. To add robustness, we had to consider cases where multiple orders are being served at the same time, which is certainly a possibility due to the frantic nature of our game. We had to refactor our algorithm, so it took these cases into account and was suitable to use with the fast pace of our game.

8.6.2 OWNERS

The owners hold a more complex role in our game. As our game concept is based on a feud between the two kitchen owners, we wanted this to be apparent in the game. The owners are instantiated outside of the kitchen and enter the kitchen when a certain amount of time has passed. As we record various properties of the state of our game throughout, the owners can analyse game performance on an individual level within members of their team, such as points gained by

players, dishes cooked by players and the number of players kicked out by a player.

After analysing game performance, the owners can carry out various tasks. The owner decides what task it will take depending on the current state of the game. For instance, if the team is losing by a small margin and there isn't much time left before the game ends, the owner may decide it would be more reasonable for a player not actively contributing to go over to the other kitchen and sabotage. Having the owner as a part of the game not only makes the story a part of our game but also helps players explore more areas of the game and hence get a more complete experience.

8.7 COMPLEX PARTICLE SYSTEMS

Unity allows for the creation of particle systems, usually used to create smoke, fire, mist or clouds.

One of the aims of our game is to create a chaotic environment in a realistic kitchen. Including particle systems of fire, bubbles, smoke and others would achieve this. Due to the particle systems being highly customisable, we were able to create realistic looking effects that compliment the overall aesthetic of our game.

	Shuriken	**VFX**
Particle rendering, logi, simulation	CPU	GPU
Amount of particles	Thousands (<2000)	Millions
Pipeline compatibility	URP	URP in preview
Behaviour customization	Less	More
Physics	Access to main physics system	On limited particle count

Table I: Comparison of Shuriken and VFX particle systems across five different categories[5]

[5]https://docs.unity3d.com/Manual/ChoosingYourParticleSystem.html
https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@12.0/manual/index.html

It is essential to distinguish between the two types of particle systems available in Unity - Shuriken particle systems and Visual Effect Graph (VFX) .

The main driving force for us to exclusively use shuriken particle systems is the aim to have multiple smaller particles in many different parts of the game. The way we intended to utilise the systems would render most additions and unique VFX attributes useless, leaving us with many powerful effects that could potentially lag the game. We wanted to make our particles reactive and responsive to their environment by implementing collisions. Shuriken particle systems provided us with all the means to achieve these proposed goals.

8.7.1 2D PARTICLE SYSTEMS

In order to achieve a more visually interesting look for the minigames, we added particle systems. This proved to be more complicated as we had already made all the canvases and 2D elements for the minigames. Each minigame consists of a UI canvas where the minigame is rendered. However particle systems are not UI elements – meaning they don't render on top of UI elements. Consequently, we needed to add in a UI camera and use it to render the canvases, reworking the majority of the spawning code for all the minigames. For example, the cutting game uses a burst of droplets that appears when an ingredient is cut.

The main menu is the first element of our game the user sees. As such, it should be a good representation of the look and the nature of the game. To best showcase the feel of our game, in the background of the main menu we added particle systems of falling items commonly used in kitchens.

8.7.2 3D PARTICLE SYSTEMS - COOKING SYSTEMS AND RANDOM PARTICLE EVENT SYSTEM

There are a variety of particle systems implemented with specific appliances in order to create a realistic feel of cooking in the kitchen.

When a stove is in use, three different particle systems are triggered. Realistic smoke that pools on the ceiling and clouds the room bursts from the pot. Inside the pot, bubbles can be seen forming on the bottom, floating to the surface where they burst. This system also implements the concentration of bubbles in the middle

to simulate realistic boiling. For boiling to happen there needs to be enough heat. To show this, a heat haze particle system was added that forms above the pot, distorting the smoke and anything else surrounding it from the point of view of the player.

If one leaves an item unattended in an oven, it might start a fire. We modelled such situations in our game. A big fire with flames, smoke, sparks and glow elements starts emitting if a dish is overcooked by more than 5 seconds. As detailed in Section [8.5] one of the methods for sabotaging is to change the timer on an opponent's stove timer. When this happens, without the player's knowledge they have no way of knowing they need to take the dish out sooner, so it overcooks, burns and starts a fire. The only way to stop this fire and to be able to use the oven again is to use the fire extinguisher to put it out.

After thorough research, it was found that in environments where oil fires occur, such as commercial kitchens, the ideal fire extinguisher to use is wet chemical based. However, these types of fire extinguishers look different than the ones most people are used to. It was decided to prioritise recognisability over realism, so a traditional red fire extinguisher model was made in-house. For the particles, we made them look like a mixture of foam and water with a slight glow. The logic behind putting the fire out is based on particle collisions. The water particles are tagged with “water” and on every particle collision with the fire element, the fire emitter rate is decreased by a constant rate. Once it hits zero and the fire is out, all the other particles producing smoke and sparks die out as well.

Another particle system triggered by collisions are the sprinklers. There are three in each kitchen, always going off in unison if there is too much smoke in the kitchen. The water particles are also tagged, so when they come into contact with fire, they have the ability to put it out. The main reason for their creation was to create even more chaos in an already chaotic environment.

Smoke that does not trigger the sprinklers is produced by the smoke bomb. Yet another sabotaging tactic, the smoke bomb particle system is designed to create chaos in the kitchen by filling the room with dark puffs of smoke, clounding the players vision.

We aimed to put the chaotic experience to the forefront, thus a random particle event system was created. Three particle events happen in each kitchen during each game. One is always the sprinklers going off and the other two are small fires that start on a random stove. These fires are different from the one in the oven, because if they are not put out, they will die down on their own after some time. They are still extinguishable by water, so if the sprinklers start at the same time, they would kill the fire. We limited these events to the second half of the game, to let the game build up on its own. As it gets closer to the end and the team is rushing to either cook more dishes or sabotage the other team in order to win, the fire and sprinkler event will add to the tense atmosphere.

8.1.3 3D PORTAL DOORS

Early feedback we received was to better represent which sub-room an ingredient is located. In response, we thought to colour coordinate the doors of the sub-rooms with the background colours on the recipe cards.

We did not want to have actual doors as it would slow down the gameplay - players should be able to swiftly enter and leave the sub-rooms. Therefore, we opted to create portal doors. Doing so not only added visual interest to the games setting, but also quickly drew the users attention when first entering the game. This is ideal when trying to explain to the player how to fetch ingredients.

Figure 12: Portal-like doors for the subrooms

Different particle systems were layered on top of each other to create the portal doors. All textures used were made in-house in photoshop. The parent particle system used a shockwave texture and gravitated its particles towards the middle to bring attention to the contents inside the rooms. To not obstruct the player's view of the ingredients inside, the centre of the portals were

kept relatively transparent with only slight embers and distortion.

8.8 SHADERS

8.8.1 TOON SHADER

Choosing to add a Toon Shader was a conscious choice as it connects back to our initial concept designs. We wanted our game to have a cartoony style with bright saturated colours. To create the Toon shader, a custom node with a custom lighting script was created to access the directional lighting in the scene. The amount of light that a surface receives was calculated using the dot product between the normal of the surface and the direction of the light. Then, to create a cell shading effect, a smooth step node was applied to divide the lighting into two bands. Ambient lighting was also added to create a slight bloom effect, reminiscent of games like Breath of The Wild and Windwaker. This effect also greatly brightened the game overall and made it feel more colourful. Finally, rim lighting was incorporated to add further dimension to the low poly models.

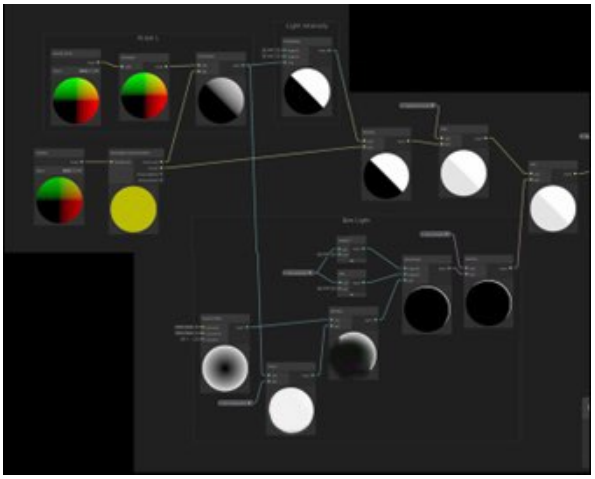


Figure 13: Toon shader graph

8.8.2 OUTLINE SHADER

The outline shader named “Contour” was implemented to create glowing instructions, in response to feedback that it was not very clear how players could cook dishes. To make the appliances glow, we first needed to create an outline. A prefab of the appliance was instantiated and the material, that used the “Contour” shader, was applied to it. After adjusting the thickness attribute, a white outline surrounding the appliance was achieved. To make these outlines glow, the bloom post processing effect was utilised.

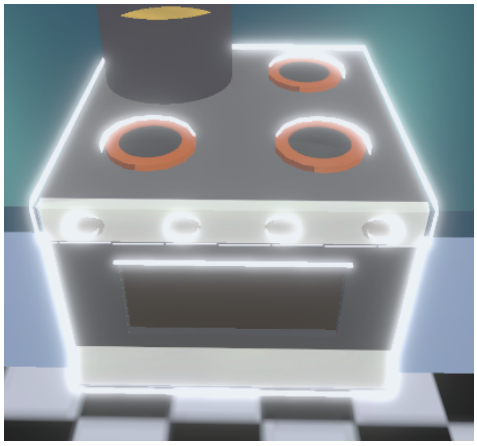


Figure 14: Outline shader

8.8.3 POST PROCESSING

Further post processing was added to the game. Mostly for aesthetic reasons; for example, colours were adjusted globally to liven the atmosphere of the game and depth of field was added to blur far away objects in the camera's view. However, similar to the glowing instructions, there were also other effects which helped provide a more intuitive game play, such as a red vignette that would flash on the screen whenever a player is hit.

8.9 DYNAMIC AUDIO - SFX AND MUSIC

Audio plays an important role in games, and if used correctly can help build an immersive atmosphere and make the game more engaging. We aimed to do this by targeting both the music and sound effects.

8.9.1 DYNAMIC MUSIC

Dynamic music involves a change in volume, rhythm or tune in response to specific events in game. There are many different methods to do this in the games industry, with creative freedom to devise how to fit all the different aspects into our game.

All of our logic is implemented using a *MusicManager* singleton. This simplifies our logic, as the relevant scripts only need to call the correct method to switch tracks, while the *MusicManager* takes care of the remaining logic.

As a first step, we use horizontal sequencing. This involved crossfading between tracks depending on the location of the player. This is done by using 2 different audiosources and a coroutine to interpolate between volumes. Switching tracks makes our game more life-like as players feel the change in mood when exiting a kitchen to the outside world, or otherwise entering another kitchen. Additionally, we switch to a different soundtrack when entering a minigame. In conjunction with the shift of view to 2D, the sudden change in music, brought about by not crossfading the tracks, helps create a different feel and shows the player that it is a different segment of the game.

A more elaborated technique we considered for horizontal sequencing was phrase branching, in which the next segment would only start when the current one has ended. However, this would not be appropriate in our case as players may not be in the same location long enough to hear the switch in music. Instead, we split our music into sections and randomly assign sections of the correct track to the audiosource. We implement this using a pseudo-random algorithm to avoid repetition. As an added benefit, the music sounds less monotonous, as each time they play the game, players would hear a different sequence of the music.

Another dynamic element we added to our music was switching the music to a faster-paced track towards the end of the game. This adds to the urgency of the game as players feel the shift in mood in the last bid to outscore their opposing team, at the same time acting as an audible reminder of the remaining time.

As music has a huge effect on the mood of the game, we also sought to make our music respond to certain events of the game. This was done by pitching the audiosources, either increasing or decreasing the pitch and speed of the track, which made for a more frantic or relaxed atmosphere respectively. The events the *MusicManager* responded to include the oven fire, entering your opponent's kitchen or when there is a noticeable difference between team scores.

8.9.2 DYNAMIC SFX

In order to add more realism into our game, we incorporated dynamic sound effects. This was achieved by randomly playing different sound effects from a pool of similar sounding audio clips whenever certain events occur. These events could be the player dropping items, hitting other players, or using the cutting board.

Where these dynamic sound effects were chosen to be applied was a conscious choice. Some sounds such as cutting/slashing are heard repetitively in short time intervals; therefore to add variety, different audio clips are used. For cases where we could not find suitable high quality audio clips, we would instead randomly pitch our original audio clip. This makes our audio more realistic, as cutting for example, does not have a homogenous sound.

Most sound effects in the game are 3D, meaning they fade in and out with distance from the source, and are also directional. Using Photon's RPC calls, we synchronise the relevant sound effects for other players. As for sound effects inside minigames, they can only be heard by the player who is playing them. Other atmospheric sounds are persistent in the background, such as talking noise from the restaurants. This aids in creating an immersive environment as it makes the restaurant appear more crowded, which also contributes to the chaotic feel of the game.