University of BRISTOL

Feathered Fugitives

Oh Deer

Team Manager - Sonny Cooper
Technical Lead - Dylan Quinton
Lead Designer - Alexander Horsman
Team Members - Charlie Nasiadka, Jack Wayt, Xin Yan Lim

Contents

1 Signed Declaration

We declare that the work in this report was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that except where indicated by specific reference in the text, this work is our own work.

Alexander Horsman:	Alexander Horsman	Date:	04/27/2025
Charlie Nasiadka:	Charlie N.	Date:	04/27/2025
Dylan Quinton:	[illegible]	Date:	04/27/2025
Jack Wayt:	[illegible]	Date:	04/27/2025
Sonny Cooper:	scooper	Date:	04/27/2025
Xin Yan Lim:	[illegible]	Date:	04/27/2025

2 Top Five Contributions

1. We developed three distinct AI agents driven by behaviour trees, coordinating via a global state and choke point detection for advanced behaviours. They interact with the world by influencing music, visuals and using a complex detection system alongside OpenAI for creative chicken-related puns.

2. We made use of the graphics engine and computational resources to produce exceptional graphics at a reasonable performance.

3. Continuously playtested with new players and iteratively adapted development based on feedback, with a strong focus on user experience. This led to impactful improvements, most notably, the creation of a custom tutorial that clearly introduced core gameplay concepts and dynamically adjusted instructions based on input method (controller vs. keyboard). This allowed us to produce a game that's polished to a near-publishable level.

4. We used professional development techniques and patterns to manage our codebase and produce high quality modular code contained within appropriate assemblies.

5. We used an agile development cycle along side GitHub to organise our work, completing 30 team meetings, over 800 commits, 86 pull requests and over 150 closed issues in 12 weeks.

Video link: https://youtu.be/x8is0vM1Ce0

3 Nine Aspects

3.1 Team Process

- Two weekly in-person meetings, one at the start of the week to define that weeks tasks and one near the end to discuss issues and follow up on that weeks progress. (See section 5.1)
- Utilised Agile sprints to structure our development, ensuring continuous progress and improvement
- Organised brainstorming sessions when faced with issues and used Milanote to map our ideas out.
- Pair programming was used on larger coding tasks.
- Kanban board used to visualise the workflow, allowing team members to monitor progress. (See section 5.7)
- Regularly updated Gantt chart, providing a clear roadmap to stay on track with deliverables and to help identify task dependencies. (See section 5.7)
- Documented the primary weekly meetings along with project supervisors inputs into weekly minutes, allowing team members to refer back to for advice/key details.
- Iterative development driven by feedback to reduce risk of wasted development time and guide further development.

3.2 Technical Understanding

- Read book on Unity fundamentals [1] to help understand movement, animation, graphics and basic AI (see section 6).
- Read the Unity Documentation at https://docs.unity.com/.
- Read the Blender Documentation at https://docs.blender.org/.
- Researched algorithms and decided on the Hungarian algorithm for optimal assignments of agents to offsets [2]. (See section 8.1.4)
- Researched the OpenAI API to be able to integrate it with the AI agents for realistic text. (See section 8.2)
- Researched optimisation to add general optimisations to the game to improve performance.
- Researched profiling to understand where the bottlenecks in performance are.
- Researched lighting and graphics to ensure that the game was beautiful.

3.3 Flagship Technology Delivered

- Implemented 3 distinct types of agents which are controlled by Behaviour Trees and share a global state contained in a Manager to allow for advanced coordination. (See section 8)
- Allow for each agent to update the global state to allow for inter-agent communication.
- Implemented advanced detection scripts that allow for the player to be detected using sight, sound and proximity.
- Visualised the AI's "thoughts" using a combination of sound effects, visual elements and music.
- Agents are integrated with the OpenAI API to allow for them to generate realistic comments in context of what is happening. (See section 8.2)

3.4 Implementation & Software

- Utilised the Observer design pattern to ensure modularity and remove dependencies from the package.
- Utilised events to further modularise and remove dependencies.
- Created a choke point detection function that uses voxels and clustering to determine the location of choke points. (See section 8.3)
- Created an editor script to pre-process the choke point detection, removing the need for it to be calculated during gameplay, improving game performance.
- Utilised the Hungarian algorithm to assist in the coordination of the agents during the surround behaviour. (See section 8.1.4)
- Employed Behaviour Trees to control the individual agents, in which the AI Manager can give them "Orders" to override their behaviours. (See section 8.1.5)
- Implemented a layered soundtrack that plays more layers in the music as the AI gets closer to the player.
- Implemented a day and night cycle used to create tension and visualise the agent's vision cones.
- Implemented the ability for agents to "speak" using the OpenAI API. (See section 8.2)
- Made use of industry-standard tools including Blender, GitHub, Unity, and GIMP to support development and collaboration.

3.5 Tools, Development & Testing

- Used an agile git-flow development style [3] with weekly meetings and a total of 5 sprints. (See section 5.4)
- Pull requests we reviewed and tested by others before merging.
- Frequent user testing, utilising methods such as think out loud testing and questionnaires which used the industry standard PXI questions. (see User Testing, section 7.3.1)
- Creation of tools such as a component finder to assist in bug fixing.

3.6 Game Playability

- Movement is generally seen as smooth and simple. (see User Feedback, section 7.3.2)
- Key system allows for people of all skill levels to play and enjoy the game.
- Key system and randomness in AI allow for replayability as every run is different.
- Controller and Keyboard and mouse input devices are supported, allowing people to use the device they are comfortable with.
- The enemy agents are distinctive colours which will assist colour blind players in identifying threats.
- Re-spawn mechanic is forgiving and allows players to make mistakes while continuing to advance through the game. (See section 8.6.3)

3.7 Look & Feel

- Custom chicken, Farmer and Dog assets along with their respective animations allow for a distinctive look to the game. (See section 8.7.1)
- Use of post processing such as the tone mapper, bloom and vignette improve the visual qualities of the game.
- Baking the lighting and reflections improve the visual aspects of the game. (See section 8.7.2)
- The day and night cycle allow for the game to show off different styles of lighting and emphasise different elements of the scene.
- The use of the physics engine for movement leads the player movement and interaction with the environment to feel natural.
- Implemented a visual audio system displaying on-screen cues to indicate the direction of nearby AI enemies.

3.8 Uniqueness & Innovation

- Emphasis on the visualisation of the agents ”thoughts” to show the player what the AI is up to.
- Use of OpenAI API allows for the agents to make in context comments about their situation.
- Use of choke point detection allows for highly complex behaviours in the AI. (See section 8.3)
- The use of several types of AI agent allow for complimentary interactions that bring unique challenges to the player. (See section 4.3)
- The custom assets, animations, lighting and post processing lead to unique visuals. (See section 8.7.1)
- Created a team wiki, documenting key systems along with UML Diagrams, enabling members unfamiliar with certain sections to quickly understand and contribute effectively.

3.9 Report & Documentation

- Created a comprehensive report explaining the technical aspects of the game, including coherent diagrams to clarify system details and visually represent gameplay elements.
- Created diagrams to model the agent behaviours.
- Created a Wiki consisting of brief descriptions and UML diagrams for the key technical areas [4] to allow developers to easily change between sections in the game.
- Utilised markdown files in the repository to keep track of weekly minutes, test day takeaways, and bugs.

4 Abstract

4.1 Overview

Feathered Fugitive is a third-person 3D game played on PC with either a keyboard and mouse or controller. In this game, you are a chicken on a farm plotting your escape. You will need to gather a number of keys through a series of challenges and exploration, unlock the electrical box to turn off the power to the gate and make your grand escape, all the while farmers and their dogs try to stop you.

4.2 Gameplay Loop

The game begins in a chicken coop tutorial area where you will find an escape plan pinned to the back wall briefly detailing the dangers, the plan and the map. This is accompanied by a skippable tutorial which teaches you the base controls for the game (Figure 1). This tutorial will change instructions in real time depending on whether you use keyboard and mouse or controller. Once the tutorial is completed or skipped the coop doors open and you can begin the game.

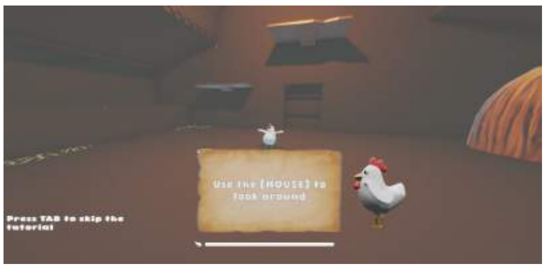

Use the [MOUSE] to look around
Press TAB to skip the tutorial

Figure 1: Tutorial

Once through the doors the timer starts, and the game begins. You will find yourself on the farm, surrounded by dangerous AIs who want to foil your escape plan and glowing keys that take you one step closer to freedom. For this game you will be given 10 minutes to escape, in the beginning it is sunny with more relaxed music and a more colourful scenery, however, the sun will set halfway through the game, with the darker pallet signalling to you, the player, that time is running out (see figure 2). Creating a sense of urgency. Intensity is further increased through a change in background music after you turn off the power.

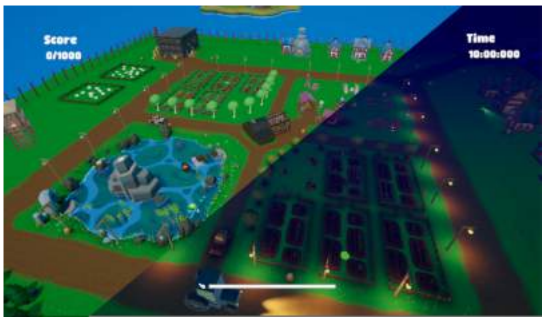


Figure 2: Day/Night

4.3 AI Enemies

Within the game 3 types of enemies will be looking for you and will try to catch you from the moment you leave the coop until your grand escape. All of these enemies are different with different strengths and weaknesses. They will work together, communicating and utilising their strengths in order to catch you.

The first type of AI is the tractor. It has a very large vision cone with its sight lines up to 150 meters long, but balanced with a narrow field of view. It's horn means it can alert many others of your whereabouts across long distances. However, this tractor is restricted to the wide square-shaped farm road, because of this it is relatively easy to escape if spotted.

The farmers on the contrary have much shorter sight lines but a much larger field of view allowing them to effectively spot the player at short to medium ranges. They have a much smaller alert area, meaning less enemies will join in the pursuit if spotted. Unlike tractors, the Farmer is not constrained by paths and will chase you throughout the map. Every farmer carries a shotgun capable to knocking the player out of any unreachable areas.

The final type is the dog. It has a short-range vision and a very wide field of view. It is the fastest entity in the game, meaning that players will have to find higher ground to escape it. It also has proximity detection, as it can smell the chicken, making the dogs extremely powerful in close-range encounters. This is balanced by preventing them from communicating accurate locations of the player to the other AIs and their inability to follow the player to higher ground.

We have made an attempt to showcase the thinking and communication of the AIs in a way that feels natural to the player. We have utilised sound effects to showcase events such as being spotted, and we have a layered soundtrack that adds layers to the music as the agents get closer to the chicken. Along with this, visuals have also been added to the game to further showcase the communication, with alerted AIs being made more obvious with pop-ups above their heads and visual audio shown around the player to further display the direction in which the enemies are coming from. Finally, the map changes to night as time begins to run out. The uses of torches and headlights further showcase the views of these AIs in the second half of the game, again enforcing the concept of AI thinking.

4.4 Key System

To ensure the game is fun and playable for both casual gamers and die-hard PC fans, we have implemented a key system. Throughout the map there are many keys, some are gold keys; showcasing the harder more challenging tasks such as parkour in the windmill, some are bronze keys; showcasing the easier basic tasks such as climbing into the boat, and finally in the middle there are silver keys, which provide a bit of a challenge such as climbing onto the window of a house. To unlock the electrical box you will need 1000 points, a bronze key is 100 points, a silver 250 and a gold 500. This system means the more casual players are not forced to do tasks they find extremely difficult whilst the more competitive players can go above and beyond, getting more than enough keys to escape in order to improve their overall score.

4.5 Score

The score functions similar to the old Super Mario Bros game [5] concept where a clock will count down from 10 minutes and the time you have left once completing the game will be added to the overall score. This lends itself to different play styles as some players may go for overall speed to get a good score whilst others may attempt to collect as many keys as possible within the 10 minutes to achieve their score. It was important to us that different strategies could be utilised in our game to ensure it didn't become a one play game, and so we adopted and tuned this score system over of our original timer system.

5 The Team Process and Project Planning

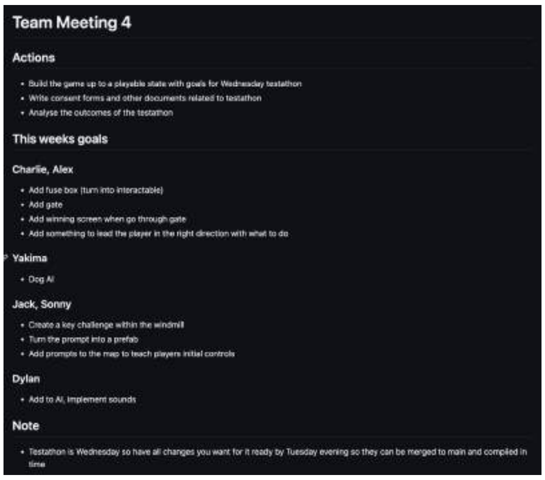
Coming into this project only one member was proficient in Unity. To combat the potential learning curve, the project manager read a book on Unity fundamentals [1], with this we organised ourselves to ensure learning had a minimal impact on the progress of our game.

To start, our technical manager, who is proficient, set up the base structure of our game and worked on the more complex issues. The others split into two groups, a group of 3 which began on the more simple and beginner issues to get an idea of the language, and a group of 2 where the project manager utilised pair programming to teach the other member more complex concepts. After the first two weeks the technical manager then reviewed the codebase, refactored, and gave points to improve on. This ensured an overall fast start to the project. Occasionally throughout the project and during the meetings the technical manager would point out where the code could be improved to help the others develop their skills.

5.1 Weekly Meetings

Throughout the project we had weekly meetings on both Mondays and Thursdays. This ensured the team stayed together as one and people didn't go away and complete things others did not know about. It also ensured people always had an issue to work on.

The meetings were set so Mondays would be the primary meeting where we would discuss ideas and new issues to complete that week; if people wanted to change pace from something they had been working on for a while, this is when it occurred. As this was the more important meeting all things that were discussed were written down and sent to the repository so that they may be referenced if there was any confusion (Figure 3). Along with this a weekly contributions sheet was also added to for each team member.



Team Meeting 4

Actions

- Build the game up to a playable state with goals for Wednesday testathon
- Write consent forms and other documents related to testathon
- Analyse the outcomes of the testathon

This weeks goals

Charlie, Alex

- Add fuse box (turn into interactable)
- Add gate
- Add winning screen when go through gate
- Add something to lead the player in the right direction with what to do

Yakima

- Dog AI

Jack, Sonny

- Create a key challenge within the windmill
- Turn the prompt into a prefab
- Add prompts to the map to teach players initial controls

Dylan

- Add to AI, implement sounds

Note

- Testathon is Wednesday so have all changes you want for it ready by Tuesday evening so they can be merged to main and compiled in time

Figure 3: Weekly meeting notes

Along with the primary meeting the Thursday meeting was utilised as a way to review progress and resolve issues. In this we would begin by creating a PowerPoint on the changes that had occurred since last Thursday and then discussed any difficulties or questions that may have arose from this weeks goals. The rest of this meeting then consisted of pair programming upon these challenging areas, this minimised the chance of people getting stuck on an issue throughout the weekend.

5.2 Brainstorming Sessions

Throughout the project, whenever we encountered issues or hit a wall, we organised brainstorming sessions to encourage creativeness and problem-solving. To visually map out our ideas we used Milanote (Figure 4), which helped us in shaping a cohesive design direction for the game.

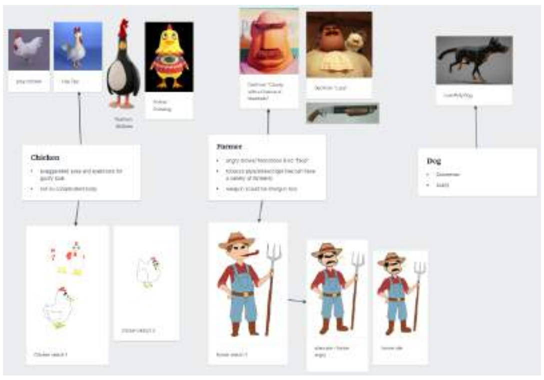


Figure 4: Example of one brainstorming session on Milanote

5.3 Conflict Resolution

From the start of the project, our team leader made a conscious effort to involve everyone in the decision-making process. During meetings, he regularly asked for each team member's input and ensured that all voices were heard before any decisions were made. This inclusive approach ensured that no one felt left out in the team, and at the same time minimised the likelihood of conflicts escalating.

Overall, due to the frequent meet-ups there were minimal conflicts. Where there were conflicts, both sides would typically voice their reasoning on the issue before coming to a compromise. The primary example of this comes from the egg shooting dispute... Half the team wanted the eggs to shoot forwards so it would be easier to aim, the other half of the team thought it would make no sense for eggs to come out of the front of the chicken due to their anatomy. The compromise for this came from having an aiming feature be combined with the reverse camera, giving the ease of aiming the egg whilst maintaining the basic logical anatomy of the chicken.

5.4 Agile Sprints

At the beginning we split the game into three iterations, an MVP, a beta release and a final release. This was further split into 5 total sprints, each leading up to an important date such as a panel meeting or a test day. The frequent sprints ensured that frequent integration into the dev branch occurred, ensuring a relatively up to date version of the game was always present.

Using the initial requirements defined for each iteration of the game from week 1, we further defined the feature requirements for each sprint at the end of the last sprint. This occurred via a group review on the Thursday meeting, this was generally productive as each sprint ended in a form of testing event, and so issues typically arose that could be placed straight into the next sprint.

A key part of our sprint planning process was allowing each team member to choose the tasks they were most interested in. Each Monday, we would summarise what needed to be done for the week and break it down into a set of tasks. Team members then selected the ones they felt most motivated to work on. This approach ensured that tasks were picked up efficiently, while also keeping everyone engaged and invested in their contributions. As a result, tasks were completed more effectively, and overall team morale remained high throughout the development process.

5.5 Planning

In the early planning phase, we held a series of meetings to brainstorm gameplay mechanics and map the structure of the game. The initial idea was simple: a farm setting where the player, playing as a chicken, must escape while being chased by AI-controlled entities — a farmer, a dog and a tractor. The objective was to collect two keys through puzzles or parkour challenges in order to unlock the electric box and escape.

Once the core concept was established, we identified major components for the MVP, including basic movement, AI behaviour, key mechanics, and win conditions to ensure the game was fully playable and testable by the panel, allowing us to gather early feedback on gameplay and core interactions. These tasks were distributed across the team and aligned to our sprint schedule.

Following our first round of playtesting, we found that some players, especially those less familiar with games struggled with the original setup of finding two fixed keys. To make the experience more enjoyable for a wider audience, we redesigned the key mechanic by introducing a gold/silver/bronze key system where each key awarded a different number of points. This allowed players to choose how much to engage with the map and reduced the difficulty barrier for progression.

Then, we started focusing on increasing replayability and refining the scoring system. A timer was added, encouraging players to escape as quickly as possible to maximise their score. We also drew inspiration from classic arcade-style games Super Mario by making the remaining time contribute positively to the final score, while player deaths would subtract from it. To support this, we implemented a respawn system which when the player dies, they instantly respawn with the timer still counting down and their collected points unchanged. This made gameplay faster-paced and allowed for competitive, repeatable runs where mistakes were penalised, but players could continue to immerse in the game experience.

To avoid last-minute issues, we made a conscious decision during the final sprint to stop implementing any large new features. Instead, we shifted our focus toward polishing existing systems and refining gameplay based on user testing feedback. This involved reprioritising tasks, postponing less critical ideas, and concentrating on improving player experience through bug fixes, UI tweaks, and minor gameplay adjustments. The final week was reserved for testing, finalising ongoing work, and compiling the release build. We also allocated time for creating the gameplay video and presentation materials. To mitigate the risk of last-minute bugs, we compiled a stable version of the game a few days before Games Day. This ensured that even if a late addition introduced critical issues, we had a reliable build ready to present. This approach helped us avoid common problems such as game-breaking bugs or rushed content and ensured the game was stable, well-tested, and ready for Games Day.

5.6 Integration

For integration, we defined early on a general workflow which consisted of taking an issue from the Kanban board, creating a branch off of dev for which this issue would be resolved and then creating a pull request back into dev. Each PR was linked to its corresponding issue on the Kanban board to maintain clear traceability between tasks and code changes. Inside the PR, we clearly described what had been done, which made it easier for reviewers to understand the purpose and scope of the changes. Whenever relevant, we also included screenshots or recordings to help visualise visual/UI updates or confirm that bugs had been resolved. This request would be reviewed by another member, tested, and then merged. This flow ensured frequent integration into the dev branch as all personal branches would only consist of one issue on the Kanban and so would be relatively small and simple to integrate in.

Coming up to the end of sprints the rule was any feature you wanted in the compiled version must have a pull request by Monday evening. Tuesday was then spent integrating all components in the morning before compiling in the afternoon so it would be ready for the testing activity that would occur on Wednesday. The code for this fully integrated version would then be pushed to main to ensure a complete version was always present.

5.7 Project Management

To ensure everyone had a general idea of the timeline and when issues needed to be finished we utilised a Gantt chart (Figure 5), this showcased both the expected completion dates for each iteration along with the dates for each sprint cycle. This combined with the details within the weekly notes ensured team members both understood upcoming deadlines along with the requirements for those deadlines.

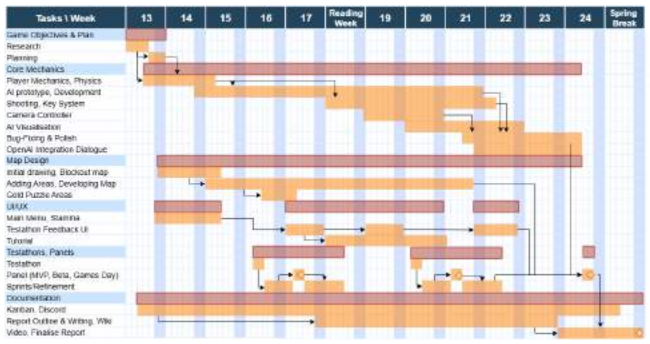


Figure 5: Gantt Chart

For task allocation we used GitHub's Kanban board (Figure 6), we would discuss all issues for that week in our Monday meeting then add those tasks to the board along with the allocation of who was going to do what. We refined our Kanban workflow over time by splitting tasks into the columns: 'To Do', 'To Do High Priority', 'In Progress', 'In Review', and 'Done'. Within each issue card on GitHub, we made full use of the available features to clearly document the task. We tagged whether it was a bug, a feature, or a general task (if it didn't fall cleanly into either category). For bugs, we provided detailed steps on how to reproduce the issue, expected behaviour, and any relevant screenshots. For features, we included a breakdown of the functionality and intended behaviour. This level of detail helped reduce misunderstandings and made it easier for other team members to pick up and complete tasks effectively. This meant team members could both understand what their weekly tasks were in a simple and visual way, whilst also ensuring others could better view the progress on tasks that may overlap with their own.

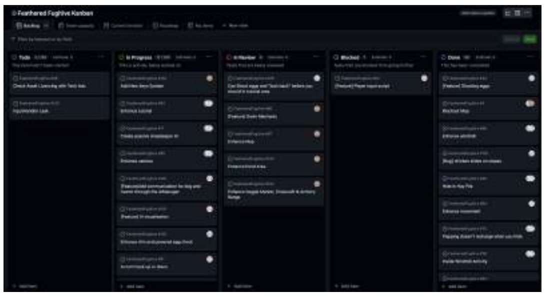
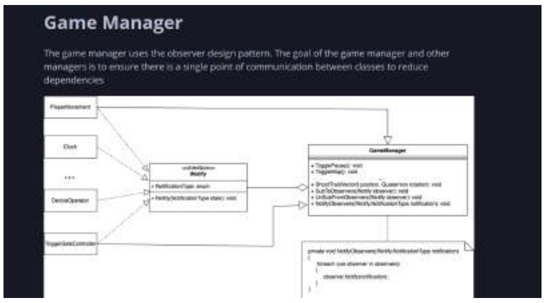


Figure 6: Github's Kanban board

To allow team members to move between aspects of the game with minimal resistance we utilised a form of Wiki to create documentation [4] (Figure 7). With this each key aspect of the game was broken down into architecture diagrams and descriptions. The main justification for this stemmed from the concern that members could remain fixed on one section of the project due to a lack of understanding of the other aspects, possibly leading to a loss of motivation and burnout.



Game Manager

The game manager uses the observer design pattern. The goal of the game manager and other managers is to ensure there is a single point of communication between classes to reduce dependencies

Figure 7: Technical Documentation

5.8 Skill Allocation

Due to the nature of games involving skills other than coding, and the weakness of our team being the fact that none of us took the CGI unit, skill allocation was an important issue. For this we decided that one person should become familiar with blender, one familiar with GIMP and one familiar with Unity animations. This along with the technical lead already being proficient in all three, led to a smoother overall process when it came to graphics.

5.9 Reflection

Over the course of 12 weeks, we gradually improved our workflow and learned from our mistakes. Most of these improvements came from the test days, panel meetings, and their subsequent reviews post sprint. Presenting our game, we learned to bring chargers and compile early. From sprint retrospectives, we learned to define goals more rigidly and communicate in more detail task allocation so members are less confused about what they need to complete that sprint.

One mistake we made throughout the early days was not prioritising key tasks, areas such as asset creation and animation were given more attention than the AI. We resolved this management problem through organising our issues into a priority list in the Monday meeting, this ensured that key components of the game had team members allocated, meaning if progress slowed one week the less important issues would be the ones impacted.

Another challenge we encountered was dealing with conflicting feedback from a diverse range of playtesters. A key example was the minimap. Some players wanted a minimap that showed the layout and real-time positions of AI to help them plan their movements. However, others felt that this would reduce tension and immersion, as players might end up focusing on the minimap instead of the actual game world. Another point of contention was whether the game should pause when the map was open. While some players found it frustrating to manage the map under pressure, others enjoyed the added challenge of needing to find a safe spot to open it, which aligned more with the survival aspect of the game.

These varied opinions made it difficult to satisfy all players. In the end, we had to make trade-offs and prioritise what aligned best with the tone and gameplay experience we wanted to deliver.

Overall, our key strength were these frequent meetings and their positions within the week relative to test days and panel meetings. They meant we could quickly resolve conflicting advice, leading to fewer disputes between teammates as people did not have the time to gain strong opinions about certain advice before the entire team discussed it. Alongside this new high-priority tasks highlighted within the panels could begin work faster.

Through this experience, we learned the value of consistent communication, clear documentation, and proactive collaboration, which helped us maintain momentum and deliver a polished final product.

6 Individual Contributions

6.1 Sonny Cooper

Prep Work - 2 Weeks
Read a book on Unity and learned the basics so that the start of the project could run smoothly and I could pair program to help bring others up to speed.

Initial Planning - 1 Week
As a team we spent the first week discussing the ideas for our game, deciding how we were going to manage the workload and when we would meet.

Third Person Camera - 3 Days
Created an initial script to move the main camera around the player depending on their direction of movement and mouse use. This was not powerful enough so transitioned to a Cinemachine camera for the third person view, altering the script so the player's forward movement will be the direction the camera is pointing.

Interactables - 2 Weeks
Created a script to allow the player to interact with some obstacles in certain ways, an example being mud which slowed the player's speed. Hay was created, so when you pressed 'E' you would be hidden inside the hay.

Interactables Text Prompt - 3 Days
Created a script which converts world coordinates to 2D coordinates, allowing text prompts to be displayed above game objects via the HUD.

Clucking - 1 Day
Created a script to allow the player to cluck, initially this came with a charge-up time of 10 seconds, which was displayed within the HUD and communicated everything through the GameManager.

Coop Asset - 3 Days
Worked alongside Jack in Blender to create the first iteration of the coop asset.

Reverse Camera - 1 Week
Implemented a reverse camera that looked directly behind the player when a button was held and went back to front facing once let go. This required setting up a camera swapping script to be used as a consistent place for all view alterations.

Initial Bug Fixes - 3 Days
Separate flap and jump functionality, allow flap to charge in the hay, improve camera issue within the hay.

Windmill - 2 Days
Wrote a script to raise a platform when an egg was shot into the bucket. Wrote an initial script to change the camera view inside the windmill, made controls feel weird so scrapped it.

Admin - 3 Weeks
Wrote all weekly minutes, set up all test day consent forms, player participation sheets and questionnaires for both test days. Was in charge of test day observations, observing bugs, players interactions, and personal opinions; I was also responsible for filtering all this information into a markdown document to be discussed on the Thursday meeting. Finally, I wrote most of the Wiki to allow team members to better transition between the complex areas of the project, this consisted of brief descriptions accompanied by UML diagrams [4].

Tutorial Area Sprites - 1 Week
Worked within GIMP to create the maps and other sprites within the coop to showcase the general ideas of the game. This took a while as we wanted a 'chicken has drawn this' vibe and so this consisted of drawing everything with a trackpad to make it intentionally scruffy.

Egg Shooting - 5 Days
Created a crosshair which activated when the player aimed, and charged as the fire button was held. The audio increased with the power level and triggered a cluck when fired.

Report Template - 2 Weeks
Wrote the first 5,000 words of the report, building the template of what sections should go where, sorting the bibliography, page formats and links before filling in all possible sections.

Timer - 1 Day
Helped Charlie with the initial timer, adding it to the HUD and creating a script to update each second.

Cut Scenes - 2 Day
Created a death cutscene that was later removed, and a gate cutscene which was triggered when the power was turned off, this scene traversed the map in the direction of the gate before showing to the player the gate opening.

Play-testing And Bug Fixing - 3 Weeks
Spent time playtesting and resolving issues found. Checked every collider on the map and changed all viable mesh colliders to box colliders, fixed colliders, added mud to the map, stopped player from being able to jump over the gate and other sections, added gravity to square hay, updated sprites and created a rope for the pulley system. Stopped the lead camera from going through walls.

Video - 3 Days
Jack created the video and told me what clips he wanted and where in the game he wanted them. I captured and sent them to him.

Games Day - 1 Days
Helped plan the layout for the day, tried dealing with the controller issues alongside the Rook Sacrifice team. Helped set up all lab machines with the game and did the majority of post game day disassembly.

6.2 Dylan Quinton

Project setup and Core Functionality - 1 Week
Created the Unity Project and uploaded it to GitHub for collaboration and version control. Created frameworks for core functionality including: Main menu which had a play and quit button; Game Manager, a singleton which managed the game state, stores globally accessible information as well as an event system (see Game Manager and INotify); Basic Player movement which allowed for walking, sprinting and jumping; Gameplay UI for pausing (via the Game Manager), restarting the level, and returning to the main menu; Input handler which manages the input for keyboard and mouse and gamepad.

Enemy AI - 4 Weeks
Created/Modified the behaviour trees for the AI types (Farmer, Dog and Tractor). Wrote the Detector script, which handles their ability to detect the player through sight, sound and proximity. The AI Manager communicates with the Detectors through the IDetector Interface, allowing them to coordinate and communicate.

Choke Point Detection - 1 Week
Created an editor script which is capable of detecting choke points in the map. It runs in the editor and then saves the variables into the AI Manager so that the locations can be accessed during gameplay without having to execute the detection script again, improving performance.

AI Manager - 2 Weeks
Created the AI Manager singleton which coordinates the AI Agents and stores a global state. It includes conditions to start group behaviours such as Surround, Choke, and Swarm. It also handles the Ghost Chicken visualisation allowing players to see their last known location.

Models, Animations and Particles - 1 Week
Created the models and animations for the chickens and farmers in Blender. The chicken has idle, walk, run, flap and concussed animations. The Farmer has run, walk, idle, tired, shoot, dive and call animations. Created particle effects for the chicken's death and shooting.

Sound Detection - 1 Week
Added the ability for the Detectors to "hear" nearby sounds such as the chicken clucking and objects falling, prompting them to investigate those locations. The sound notifications can also propagate through the AIs as they can alert each other to sounds they have heard. The communication is done through the AI Manager.

Detectables - 2 Days
Added detectable objects using the Detectable class. These objects when spotted by the AI will invoke different behaviours in the AI, such as investigation and calling for backup based on the priority value of the detectable object. (Removed as it was under utilised and added unnecessary complexity.)

Lighting and Reflections - 3 Days
Improved and baked the lighting as well as reflection probes. Added post-processing to the scenes to improve visual quality, including an ACES Tone-mapper, Vignette, and Bloom.

Prefabs - 1 Day
Created the key prefab, added a spotlight and animated them. Created a walking sheep and cow and chicken prefabs. Created a bullet tracer prefab. Created a Ghost Chicken that can have the current pose of the player applied to it by the AI Manager.

Gameplay UI - 2 Days
Added UI buttons to restart the level and return to the main menu and UI panels for: Game Win which shows your score and time and the buttons; Game Lost which shows the buttons and a title; Pause which shows the buttons and an additional unpause button. The UI Panels are displayed to the user using notifications from the Game Manager.

Main Menu - 1 Day
Created the main menu including placing assets, UI, lighting, reflections, music, and wrote a script to have a wandering chicken in the scene.

Sixth Sense - 1 Days
Added sixth sense into the game so that the player knows when they have been spotted, this is in the form of a UI element that displays an ! above the player and SFX which plays an alert sound.

AI Visualisation - 2 Days To help visualise the AI's intelligence I added a canvas above each enemy that displays an ! if it can see the player and a ? if it is investigating an area. I also added sound effects to the farmer and dog to indicate that specific behaviours are occurring, as well as a wave animation for the Farmer to indicate they are calling for backup.

Concussions - 1 Day
Added the ability for enemies to concuss the chicken, which disables movement, plays a particle effect and an animation.

Code and Tech Management - 5 Days
Technical Lead, therefore responsible for managing the codebase and implementation of features. Re-wrote and modularised code on occasion to reduce dependencies between classes. Added assembly definitions to manage and reduce compilation time, as well as to somewhat enforce modularity. Handled the compilation of several versions of the game.

Optimisation and Profiling - 2 Days Did deep profiling to identify any major bottlenecks and basic optimisation to improve the performance of the game.

6.3 Alexander Horsman

Initial Planning - 1 Week
As a team we specified what it was that we wanted to create, assigned roles and created a schedule. I collaborated with Charlie to brainstorm our initial gameplay design and aesthetic. Researched previous game designs and spent time learning Unity.

Key & Gate System – 1 Week
Created the initial gameplay loop of grabbing two keys, turning off electric and opening the gate. The key count and logic I later converted to be handled by the Game Manager. Added key sound effect, a UI key counter and an animation for the gate opening to make gameplay reactive to player actions.

Gameplay Design & Scope – 2 Days
Worked with Charlie to develop a gameplay loop focused on showcasing our key tech and map. Defined the initial scope of our game design and created a list of features such as landmarks, puzzles and AI types.

Unity Terraining – 3 Days
Imported an initial asset package and researched how to make realistic terrain. Added terrain, textures and flora to create a forest path. This forest did not fit the final game design but was useful for learning terrain tools and fixing render bugs.

Tractor AI – 1 Week
Made a tractor behaviour tree from the farmer behaviour tree and detector created by Dylan. Made the tractor only catch the player when they are detected and colliding with the tractor mesh. Added the tractor asset, engine noise, and horn noise when spotted. Restricted the tractor to roads by creating a new nav mesh and agent type. This initial tractor AI was later changed with the AI manager.

Objectives Panel – 2.5 Weeks
Based on testathon feedback, I changed the compass design to an objectives panel. The panel appears when a user presses ‘m’ or up on the d-pad and includes a live map, objectives and icons. Implemented an orthographic camera to create a live 2D map, I used layers to specify what each camera should render. Streamlined the code for the key system and added updating objectives and key count to the objectives panel. Used GIMP to modify and import map icons for the gate, electric box, and keys which appear or disappear when necessary. Also created a tractor icon based on the tractor asset. As this feature was more ingrained with our environment than originally planned, it caused too many merge errors. I therefore made it a second time with a newer build of our game.

Fixed Unity Library – 1 Day
Fixed issue with Unity that would change the ID of some prefabs and incorrectly import blender files, leading to invisible models and compiler errors. Fixing this allowed work to be done on the lab machines.

Day-Night Cycle – 2 Weeks
Created a day-night cycle to showcase the AI vision and allow the player to get a natural intuition for the timer. This included the creation of a time controller which allowed us to select a start hour, sunrise / sunset hour and a time multiplier. Initially, this changed the angle and intensity of the sun and moon directional lights and the environment’s ambient colour. I then made sun and moon sprites and created a bloom material to fit our environment. Sourced and added skyboxes and a skybox blend shader, which is adjusted in the time controller script. The environment fog and skybox fog were also added to change dynamically with time. Added a script to disable the map camera rendering fog and also changed the sea shader to dynamically change its colour tint as it could not process environment fog.

Adding & Blending Lighting – 1 Week
Added lighting to our map via street lamps and a script that turns lights on at a specified time. Added tractor headlights and farmer torches to match their respective vision cones. Spent considerable time playtesting and adjusting the post processing, environment and day-night cycle variables until the lighting blended and fit our game’s aesthetic. This included the addition of varying the ambient light intensity with time. Adjusted shadows and light effects to improve performance without affecting the overall design.

Respawning – 2 Weeks
Implemented a respawn mechanic based on user feedback which allowed players to experience our full gameplay loop. This required resetting any features that adjusted the player state such as hay, mud and flapping. Fixed respawn bugs caused by the AI grabbing multiple times, last spotted or concussing the chicken. Added a death count in the game manager which I could then use to visualise in the UI. To keep within our game design I added an insta-death mode which triggered when the timer ran out or when the player turned off the electric box. Added a new death sound and reworked the respawn transition to get an ’arcade-like’ design.

Game Won Panel & UI Changes – 5 Days
Reworked the design of the Game Won Panel. Modified the final score to remove points for each death and changed the score animation to reflect this. Added new animations for time left and final death count. Went through and polished the look of other panels and modified the timer UI. Fixed issues with the map in the objectives panel and remade the key icons.

Games day – 1 Day
Helped plan and set up the room layout for games day. Created a detective-style pinboard as decoration. Advertised our game with Xin by pinning posters at University hotspots.

6.4 Charlie Nasiadka

Initial Team Planning – 1 Week
Worked with the team to define the core gameplay concept, set up meeting schedules and organised the initial development plan. Created a project to-do list with Dylan and collaborated with Alex on gameplay objectives and early asset planning. Learned to use Git for team-based development, Unity, and Kanban for task management.

Flight Mechanic & Asset Planning – 1 Week
Created the chicken's flight mechanic. This mechanic was supported by a Stamina UI, for which I developed both the visual UI and the logic. I also began compiling a detailed asset list with the help of Xin.

Map Concept – 2 Days
Sketched a rough map layout, defining key areas and their purpose in gameplay. Painted terrain onto the map to establish the initial environment.

Design Board – 1 Day
Created a Milanote board with reference materials and sketched side/profile chicken drawings for Dylan to use in blender.

Environment building & Object Physics – 2.5 Weeks
Designed and built the game map, focusing on:

- Selecting assets to match the intended visual style and atmosphere.
- Implementing various physics interactions, including custom player and object physics.
- Developing a custom beach ball physics script.
- Developing majority of the map such as detailed Crops, Forest, Lake, Market and surroundings.
- Added optimisations by labelling appropriate objects as occluder and occludee to improve rendering performance and adding custom box colliders where necessary.

Gantt Chart – 1 Day
Created a Gantt chart to ensure the team had a clear development timeline and knew when issues needed to be finished.

Playtesting – 3 Days
Played the game to ensure that map design, such as asset placement and sizing was optimal, fixing small bugs and adjusting layouts during testing.

Barn Task & Swimming Mechanic – 1 Week
Designed and implemented the Barn Parkour puzzle. Extended the map with the new pond area, helping expand gameplay variety and navigation. Added a custom button script which detects if player or box is on it to open a door. Implemented a Swimming Mechanic for the lake, expanding player movement options.

Testathon Feedback & Player Guidance – 0.5 Weeks
Helped document feedback from testathon and worked on necessary improvements such as implementing an on-screen objective popup UI to improve player guidance and fixing small issues.

New Key System & Score Display – 1 Week
Added Silver and Bronze keys with unique visuals, integrated Gold key particles, and distributed keys throughout the map. Built Time and Score UI and added it to GameWon panel with animated feedback.

External Coordinator – 2 Days
Throughout the project I coordinated with composers and panel members, ensuring we met certain deadlines and regularly communicated with the team by sharing updates, supporting the team manager. Created a presentation detailing audio requirements, deadlines, and reference tracks for composers.

Visual Awareness – 1 Week
Developed a visual audio system UI in response to feedback, which dynamically reacts based on the chicken's camera position. Calculated enemy directions using vector normalisation and converting from polar to Cartesian coordinates.

Games Day Logistics – 2 Days
Organised the Games Day equipment, props and printed materials. Contributed to poster and sticker designs. Designed T-shirts to boost team identity.

Final Video – 1 Day
Provided voiced narration for the project video, clearly communicating the game's features.

OpenAI Feature Development – 2.5 Weeks
Researched and implemented a new AI dialogue feature which used the OpenAI API. Integrated the system with Game Manager and AI Manager for unique responses based on context of live in-game events and player actions. Iteratively refined it and optimised the API calls to minimise token usage while maintaining response quality. This feature made farmer interactions feel more lifelike and coordinated.

Resolving Conflicts & Play Testing – 2 Days
Merged changes with the Dev branch, resolving game-breaking conflicts and conducted hours of rigorous testing to ensure a stable Games Day build.

Games Day Delivery – 1 Day
Worked alongside the Tech and IT departments to meet technical requirements for Games Day. Organised the station setup, delegated last-minute tasks, arranged the station layout and actively engaged with panel members and players throughout the event.

6.5 Jack Wayt

Initial Planning and Preparation - 1 Week
In the first week, our group met to brainstorm and refine our ideas, narrowing them to two clear game concepts to pitch. We assigned roles and set a regular meeting schedule to stay organised and manage the workload. We also discussed the technical skills needed for development. To contribute more effectively, I researched and watched tutorials on Unity and Blender, focusing on improving my skills in Blender.

Camera System – 2 Weeks
Using Cinemachine, I created a third-person camera system where player movement aligns with the camera's facing direction. Later, I enhanced the system to improve gameplay. For tasks requiring upward visibility, I scripted the camera to offset the player downward on the screen when looking up, maintaining visibility. I also added a rear-facing camera to allow players to look behind, essential for specific challenges. Finally, I implemented camera collision detection, preventing it from clipping through walls and ensuring the player remains visible, even in enclosed spaces.

Interactables - 2 Weeks
I created a script enabling interaction with specific objects. Players could hide from AI by interacting and hiding within hay bales, while walking through mud triggered a decreased movement effect.

Interactable Prompts - 1 Week
I developed a script that displays a prompt (image or text) above interactable objects when the player is within a certain range, showing what the object does and which button to press to interact with it. It uses world-to-screen coordinate conversion to keep the prompt correctly positioned above the object. Later, I added linecasting and collision detection so prompts only appear when the player is facing the object and it's not obstructed by something like a wall.

Input Device Detector - 3 Days
I developed a script to track the last input device used, primarily for use with the interactables and tutorial. This allowed players to switch controls mid-game, with all prompts updating to match the current device. The system also distinguishes between controller types (e.g. Xbox, PlayStation) to ensure accurate input bindings are displayed.

Coop Area – 3 Days / 2 Weeks
I initially worked with Sonny on the first iteration of the coop, focusing on the basic layout and concept. Later in the project, with improved Blender skills, I revisited the coop to add more detail inside and out. Since the tutorial took place inside, I focused heavily on that area, creating additional assets like chicken beds, straw, shelves and adding extra chickens to the area.

Windmill Task – 1 Week
I modified a pre-existing windmill asset by hollowing out the interior and designing a spiral parkour path with platforms, ladders, and added details. I also designed and created the bucket asset for the pulley system. Additionally, I helped Sonny create the detection script that activates the pulley when an egg lands in the bucket.

Tutorial - 1.5 Weeks
I created a script that runs a mandatory tutorial at the start of the game, guiding players through all core controls before allowing gameplay to begin. Each instruction appears on-screen with the relevant keybind, dynamically updated based on the player's input device using the device detection script. Once completed, the coop door lowers to signal the start of the game.

Archery Task & Shooting Eggs – 1 Week
I designed the archery area and scripted a detection system that unlocks a chest revealing a key when an egg hits the target. I expanded on the existing egg shooting by adding a "look back" mechanic and a chargeable shot, indicated by a charging crosshair and clucking sound. This also introduced vertical aiming. After testing different control setups, we settled on holding left-click to look back and right-click to charge the shot.

Electric Area - 5 Days I created the electric area asset in Blender, then added details like warning posters and electrical particle effects along the fence. I created a script that detects if the player tries to jump over the fence, triggering an electrocution effect that plays an electrocuted sound as well as shaking and launching them back. I also implemented the ability to interact with the door once you've gained enough points, triggering a HUD loading bar. Then finally interacting with the lever flips it, plays the cutscene, and unlocks the gate.

Play Testing and Bug Fixing - 3 Weeks
Throughout the project, I handled bug fixes, but in the final three weeks, I focused entirely on playtesting and refining the game. I identified and resolved numerous issues, with a major focus on HUD bugs, such as text appearing in incorrect locations across different panels and features breaking after certain combinations of map or pause menu use. I also tackled gameplay bugs, fine-tuning elements like object sizes, look, player speed, and overall game feel for a smoother experience.

Video - 3 Days
I developed a structured timeline for the trailer, focusing on building interest while effectively showcasing the game. I wrote the script for the technical segment, highlighting key technologies and the effort behind development. For the footage, I directed Sonny with specific scenes to capture, which involved implementing custom camera angles, movement scripts, and varied character perspectives. I then edited and compiled the clips, adding sound effects, music, and voice-over to produce an engaging and informative final video.

6.6 Xin Yan Lim

Initial Planning – 1 week
During the first week, our team focused on discussing initial game ideas, exploring mechanics we wanted to implement, and deciding on a general direction for the project. We also established a plan for dividing tasks, using version control, and set up recurring team meetings to ensure regular collaboration throughout development.

Asset Research and Selection – 3 days
Searched and compiled a list of potential assets (low poly farm, textures) aligned with the game's visual style. Collaborated with the team to shortlist final assets based on aesthetic consistency and Unity compatibility.

Character Sketches – 1 day
Created early concept sketches of the chicken and farmer based on our visual references and game concept. These were added to the team moodboard to help establish a consistent visual tone.

Tutorial Restrictions – 1 day
Implemented checks to prevent the chicken from shooting eggs or using the reverse camera before these features are introduced in the tutorial. These functionalities are now unlocked only after the corresponding tutorial segments are triggered, ensuring structured and guided progression.

Egg Shooting Mechanic – 1 week
Developed the chicken's egg shooting feature using Unity's Input System, supporting both gamepad and keyboard/mouse input. Instantiated and launched projectiles with physics-based force, triggered explosion effects on collision with specific tags, and managed cooldowns for shooting.

Dog AI and Behaviour System – 1 week
Implemented the dog's AI using Unity's Behaviour Tree tools and NavMeshAgent. The dog patrols between waypoints and chases the player upon detection. If the player escapes, it resumes patrolling. I initially implemented event-based alerts to trigger the farmer AI when the dog detects the player; this was later replaced with a centralised AI Manager for better scalability.

Research Task: Learning Blender and Unity Behavior Trees – 1 week
Spent time learning how to use Blender for 3D modeling and animation by following online tutorials covering the basics of mesh editing, rigging, keyframe animation, and rendering. I also spent time learning how to use Unity's new behaviour graphs and trees.

Dog Model and Animation – 2 weeks
Designed, modeled, rigged, and animated the dog character in Blender. Final assets included idle, walk, and run animations prepared for Unity integration.

Playtesting – 1 week
Playtested the game regularly throughout development to identify bugs. Fixed a movement bug by reducing air control to stop acceleration upon jumping. Fixed a bug where interactable prompts appeared in the sky by switching them to the correct world-space coordinates. Also added friction materials to certain objects to prevent chicken from sliding.

Timer System Fix – 2 days
Fixed incorrect time counting logic in the game. Updated the system to start only after the player exits the coop and to pause when the game is paused.

Main Menu Input Handling – 1 day
Resolved a UI bug where the first menu button was always selected by default, which was needed for gamepad support but visually unappealing for keyboard/mouse users. Adjusted button selection accordingly using the input device detector.

AI Swarm and Aggression Triggers – 2 weeks
Made the AI swarm the player when the timer runs out to increase challenge, and implemented a more aggressive AI state after the cut scene stops.

Chase Audio Layers – 1 day
Added six dynamic chase audio layers based on AI distance to the player. The closer the enemy, the more intense the audio, providing players with escalating auditory tension.

Input Handler Initialisation Bug – 2 days
Resolved a persistent error caused by 'InputHandler.Player.Disable()' when restarting or exiting the game. The issue stemmed from redundant initialisation in both 'Start()' and 'Awake()' methods. Traced through multiple scripts that referenced the shared input handler to locate and fix the bug.

Background Music Implementation – 1 day
Added a background music track for the tutorial and a separate track that plays after the post-electric-box cutscene. Fine-tuned volume levels to prevent the music from overpowering other in-game sounds.

Promotional Materials – 1.5 week
Created the Games Day promotional poster using Canva. Rendered in-game assets in Blender, adjusting lighting and camera angles for clean visuals to get a png image file of them. After team feedback, I simplified the design and repositioned key text elements to make venue and timing more readable. Also designed the game logo, which was featured on T-shirts and stickers distributed during the event.

Games Day – 1 day
Helped plan and set up our team's station and room layout for Games Day. I recreated the graphics outlining the dangers, a map, the objectives, and controls to help visitors understand the game. Worked with Alex to pin posters at University hotspots.

7 Software, Tools, and Development

7.1 Development Software & Tools

7.1.1 Unity

Unity was our chosen game engine. It is a well established game engine with a significant amount of well written documentation as well as learning materials which allowed for the inexperienced members of the team to easily find solutions to their problems. The Unity Asset store was particularly useful for finding models and textures for the game, and packages such as the Behaviours package and AI-Navigation allowed us to focus our efforts on creating novel systems rather then re-inventing the wheel. Godot was considered but due to its relative infancy there aren't as many learning materials available online, which could potentially hinder learning. Unreal engine was also considered but deemed too computationally demanding and unnecessarily complex for our needs, alongside the steep learning curve.

7.1.2 Blender

Blender was used to create all our custom 3D models, such as the farmers, the dogs, the chicken, the chicken coop and the electrical box. It was also used to alter assets we acquired in the Unity store, such as updating the windmill to have parkour inside. It was chosen as it is a free open-source computer graphics tool set that allows us to easily create and edit models.

7.1.3 GIMP

GIMP was primarily used for the sprites found within the coop tutorial area. We created maps to help more confused players with the objectives and dangers within the game. For this we wanted to give the maps a "chicken has drawn this" vibe, and so the workflow primarily consisted of tracing over game images with the pen tool. GIMP was chosen for similar reasons to Blender.

7.1.4 GitHub

To keep track of ongoing issues and the tasks everyone was working on, we used GitHub's built-in project tools. From the start, we created a Kanban board with four key status columns, which gave us a clear overview of all tasks and who was handling what. Assigning issues and using labels helped ensure everyone knew which part of the game they were responsible for, reducing the risk of multiple people accidentally working on the same task. Later in the project, during one of our larger sprints, we added a new "Urgent" column. This made it easy to highlight high-priority bugs and features that needed immediate attention. We also made use of milestones, setting three main ones for the MVP, Beta, and Final release. At the beginning of each milestone, we outlined what needed to be completed, which helped us gauge how much work was required each week and whether a sprint was necessary to stay on track for release deadlines.

7.2 Development Process & Software Maintenance

We developed the game using the agile approach. We had a master and dev branch. The master branch would hold the latest working version of the game, and the dev branch would contain all the new features from the latest sprint. Team members would tackle issues by assigning themselves to the issue and creating a new branch where they would complete the issue. Once completed, they would create a pull request which would be reviewed and tested by at least one other team member. If there were any issues a comment would be left the team member who created the pull request would fix it.

To ensure software was maintained and bugs were dealt with whilst development of new features continued, after each Testathon the bugs found would be noted and two team members would spend the next week squashing them whilst the rest of the team worked on the new features. As we entered the final 3 weeks of the project, bug squashing became more of a priority, and so the two team members were delegated purely to playtesting and bug squashing. This consisted of changing many mesh colliders to box colliders to improve performance, improving graphics such as adding a rope to the windmill pulley system, fixed issues such as seeing prompts through walls, and tweaking mechanics such as map sizing, swim speed and the time left overall weighting on the final score.

7.3 Testing

7.3.1 User Testing

All testing days consisted of think out loud testing where a group member noted the issues found, this was followed by a questionnaire which used the industry standard PXI questions. After this, the findings were written up and relevant issues were placed on the Kanban board to make the changes. Panels and discussions with musicians were also used as a more informal form of testing to further enhance the game.

- First test day - 5th February (week before the first panel) - 14 people tested
  - AI can't see the chicken when on higher ground (can't look up or down)
  - Prompts need to be a lighter colour as hard to see in darker areas
  - Camera goes through walls and is hard to look up with
  - Map needs more direction in showing the player what to do
- First panel - 12th February - 7 people tested
  - Needs more direction in the base concept; is it a stealth game or an action game.

  - Needs to showcase the AI more so the user understands that the enemy is in fact 'smart'.
  - Needs a tutorial to help with the game understanding so we do not need to explain it.
- Musician meeting - 28^{th} February - 2 people tested
  - Utilise music to showcase the AI getting closer
  - Shift midway through from day to night, reflect in music to give more a sense of time pressure
- Second test day - 5^{th} March - 37 people tested
  - Better showcase progression
  - Outline objectives clearer
  - Further showcase the AI thinking
- Second panel - 12^{th} March - 7 people tested
  - Look into Pac Man AI [6]
  - Further showcase AI thinking

7.3.2 User Feedback

The overall census was that people enjoyed playing the game and liked it more as it developed. This was reflected in the forms where throughout all stages of testing, "I had a good time playing the game" consistently scored above 4.5/5 with some written feedback even stating "I loved it, I could play all day".

The enthusiasm was also shown by the fact that the number of people to formally test our game doubled between the first Testathon and the second, with many people not being able to play due to the queue. This was further reinforced by game day, where we had 18 lab machines running for over 3 hours and still had people waiting to play.

The standout feedback was that the controls felt nice to use; however, some found them slightly glidey. The map looked good and "like a real game" and, most importantly, the game was fun. People generally enjoyed running from AI, the goofiness of clucking and laying eggs, and the high-stakes final dash for the end. One player also really enjoyed hiding in the hay, spending half an hour jumping in and out before being told to give others a turn.

The biggest piece of feedback we received from the testathons was about the game's objective. During the first testathon, it became clear that while players enjoyed running around the map and avoiding getting caught, they were doing so without a clear sense of purpose and were unsure to what they were ultimately meant to be achieving. To help this, we implemented a tutorial to guide players on how to play and to direct their attention to clearly outlined objectives on the coop's walls.

By the second testathon, we noticed a significant improvement in players' understanding of the game and its core objectives. Some were even able to complete the entire game without any assistance from our team. However, a new issue was uncovered, while players were collecting keys and earning points as intended, they weren't aware of when they had collected enough to turn off the power and escape. As a result, we had to interrupt their gameplay to inform them.

To address this, we added an on-screen prompt to notify players when they had enough points. This change had a clear impact on the final game day, as none of our team members needed to help players with any part of the game.

Our final area of user feedback came from the panel days. For the first panel day, the idea of game direction was further enforced, leading to our focus on user experience and accessibility for all levels of players. This was proven to us as we spent the majority of this panel explaining how to play the game, something that should be self-explanatory in complete games. This led us to create the tutorial area and the map to better showcase the goals of the game. The second and more focused feedback came from the key technology, where the point was raised that for AI to perform well as a key technology it not only has to think well, but also be shown to the player that it is thinking well. From this feedback, we created farmer emotes, different AI noises and visual queues to show when you are spotted.

The second panel day was positioned towards the end of the overall project, so the majority of feedback was given with the intention of polishing, leading to a smaller impact on the overall game. From this we found that our AI needed to be even smarter and further showcased. This led to choke points being added to the game along with visual audio to display in what direction you are being chased from. OpenAI was also used for this final version to generate witty jokes using the real-time game context, further showcasing the AIs thinking.

8 Technical Content

8.1 AI

8.1.1 AI Overview

Having intelligent AI is very important for the success of the game as it provides a challenge to the player throughout gameplay. The AI will attempt to catch the player as they make their escape from the farm. The AI is several agents, each capable of making their own decisions as well as communicating with the AI Manager. There are 3 main types of agent: farmers, dogs and tractors; each will be explained in more detail in their respective sections. To make the AI uniquely threatening we wanted to find ways of displaying its intelligence and thoughts to the player providing a fear of being hunted by intelligent beings.
From the beginning of development we were aware that balancing the AI to prevent it from being too good or bad is critical as if it leans too far either way the game may become frustrating for the players. We sought regular feedback throughout development to help guide our decisions allowing us to fine tune the difficulty to be appropriate for all users.
The AI is split into 3 main components. The AI Manager,

the detectors and the behaviour trees (of which there are 3 types one; for each agent type listed above). There are a few important supporting classes such as the ChokePointDetector and Order.

8.1.2 Orders

The agents needed a form of short term memory for many of the behaviours that we wanted them to exhibit such as investigate (Search a location for the chicken) and follow (Go to a location the chicken is predicted to be in based off of variables from when it was last seen). This functionality was fulfilled by the Order class which inherits from ScriptableObject. Orders are used to provide and control what information each Agent has access to and at what time. Each order has a type, priority, location and many other variables which define what behaviour should be executed and how, as well as if one behaviour should be cancelled in favour of another. Agents receive orders from both the Detector class and AI Manager based on the conditions of the scene. For example on spotting the chicken, information is passed from the detector which saw the chicken to the AI Manager which then makes a decision on what actions should be taken and by which agent. The relevant orders are then instantiated and passed to the agents's, starting their behaviours. The detector issues a follow order if the chicken is lost.

8.1.3 Detector class

The Detector class is used to allow the agents to detect the player and other game objects in the scene. It can use sight, proximity and sound to detect game objects in the scene. We decided to give the agents multiple methods to detect the player as it helps the agents display a higher level of sophistication and ability to interact with the world. This is visualised with cones and spheres in Figure 8.

The Detectors contain an order field which stores a scriptable object of type Order which can be accessed by the Behaviour tree and written to by the AI Manager and Detector class itself, allowing both the AI Manager and the Detector to control the agents. This provides an advantage as the AI Manager does not have to control every behaviour, only the group behaviours, and when one isn't occurring each agent makes decisions for itself. This also adds to the realistic feel as the agents will all act autonomously unless required otherwise.

The Detector class inherits from MonoBehaviour and implements the IDetector interface which defines functions that allow for orders to be set in the detector, as well a notify function used in the observer pattern between the Detectors and the AI Manger. The Detectors subscribe to the AI Manager for event notifications so that the AI Manager can more easily coordinate actions when an event occurs, such as when the game ends, the AI Manager will notify the detectors to disable the behaviour trees and to stop scanning for the player. Using this method prevents the detectors from being dependent on the AI Manager, helping modularise the code.

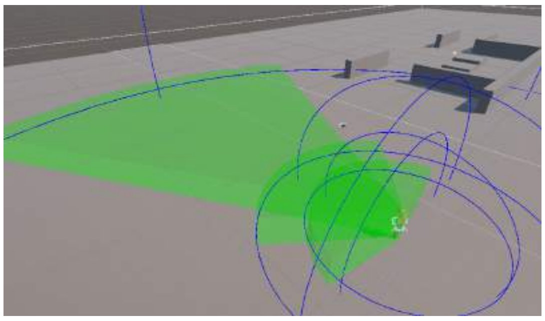


Figure 8: Figure shows a Farmers vision cones with the overlap spheres in blue.

The Detection works as follows: Every 1/10th of a second the sight detection and proximity detection functions are run which add detected objects to a list, followed by a function that checks that list for the chicken or other detectable objects. This was chosen to run every 1/10th of a second to improve performance as it hugely reduces the number of times the code needs to be run, without noticeably affecting the agents ability to detect the player (See more Profiling, Performance and Optimisation). Sound detection is handled by events that are routed through the AI Manager.

Sight Detection works as follows, for defined each vision cone an overlap sphere is cast with its collision layers set to player and detectable. For each object within the sphere it checks if it is within the height limit, angle limit and if it has line of sight to the object. If these tests all pass then the object is visible to the detector. The object is then added to the list of detected colliders. The detector is capable of running multiple vision cones at once to allow for more complex sight detection to occur. Each vision cone's settings are defined by a VisionCone ScriptableObejct.

Proximity Detection works very similarly to sight detection, an overlap sphere is cast over selected layers and for each object within the sphere it checks for line of sight. If true then these objects are added to the list of detected colliders.

Once the list of detected colliders is built, iterate over it and check to see if the chicken is one of them. If it is then the target field is set to the chicken. This field is accessible by all other components in the agent.

Sound Detection is handled by the AI Manager by calling the Sound function. Whenever a sound that the agents are supposed to react to is played, the sound function of the AI Manager is called with the location and alert radius. For each agent within this area it is alerted using an Sound Primary order. Upon receiving a Sound primary notification, the agent will propagate this and any other agents near that one will receive a Default order telling them to investigate the area as well.

Previously each detectable object has a Detectable component. This defined what behaviour should be taken once spotted. Each object had a priority to allow the agent to focus on the objects that were most likely to lead to the chicken or the chicken itself if spotted. Ultimately this feature was removed as it added unnecessary complexity causing bugs in the detection code. Paired with the sparse utilisation of this feature and it was deemed too costly to keep in the final version. It is likely given more time that it would have returned with a more prominent role in driving enemy behaviours.

8.1.4 AI Manager and Group Behaviours

The AI Manager is a singleton class (there can only be one instance of it at a time). It is used to coordinate group behaviours for the agents as well as a few solo behaviours. Only farmers can participate in the group behaviours for continuity and balancing reasons. The current group behaviours are Swarm, Surround and Choke. The solo behaviours it can control are Chase and Shoot. It can be notified if the player has been spotted by the agents using the Detector class. It stores a reference to the player so that it does not need to have the player passed to it every time the player is spotted. Upon being notified that the player has been spotted it uses available information such as the players location, velocity, the agent that spotted the player, the distance between the player and all other agents and nearby choke points (see more in the Choke Point Detection section) to decide which behaviour it will get the agent's to execute. The behaviours are chosen like this to easily allow for coordinated behaviours between the agents, helping them appear to communicate and display intelligence to the player. The AI Manager also handles the layered music, adding more layers as the agents gets closer to the player in order to increase the tension and serve as a warning system to the player.
The Group behaviours are an essential part of making the AI feel intelligent. The first is the Swarm behaviour. This is a simple behaviour that is called when the final stand is started, it gives the agents a speed boost and tells them the location of the player so that they can close in quickly across large distances and apply pressure. This behaviour occurs during events such as when the player unlocks the gate to begin the final sequence of the escape.
The Second is the Surround behaviour, as depicted in Figure 9. This behaviour occurs if the player is not facing a choke point that is within 30 meters of the character. This behaviour creates an offset for each agent within 30 meters of the player. The agents will need to add this offset to the players position and then move to that new location. The offsets are then assigned to each agent using the Hungarian Algorithm to reduce the overall distance the agent's have to travel to get into position, speeding up the set up time of this behaviour. Overtime the magnitude of the offsets will shrink resulting in the agents closing in and once within 3 meters of the player they will all dive at the player. This behaviour has been exceptionally effective at catching players as the only way to reliably dodge is to fly and due to concussions and limited stamina this isn't always possible. Despite the success in catching the player this behaviour is quite messy once it finishes as all of the farmers will end up very close to each other, which does provide the player a chance to escape but it also can feel very chaotic and break the illusion of intelligence.

The Hungarian algorithm (also known as Kuhn–Munkres algorithm or Munkres assignment algorithm) was chosen to pair the agents to the offsets because some agents would get to their position much sooner than others. By using it to pair them and minimise the total distance the agents travel this will speed up how quickly the agents are able to get into position. Whilst the Hungarian algorithm has a poor time complexity of O(n^3) [7] since it is limited to 4 agents, this will have a negligible performance impact.

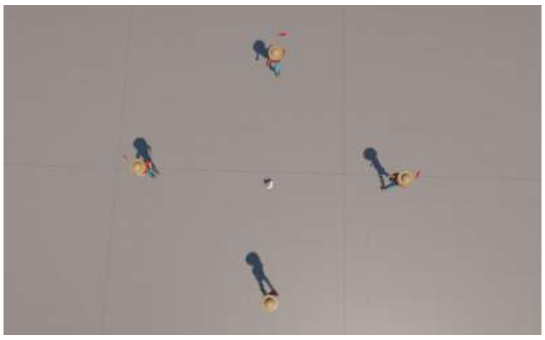


Figure 9: Figure shows 4 Farmers surrounding the player.

The next behaviour is the Choke behaviour, as depicted below in Figure 10. This behaviour occurs if the player is facing a choke point within 30 meters of their location. This behaviour will send one agent to wait at the choke point known as the blocker. The next agent will act as the net and will hold an offset of the players location. The offset for the net is calculated using the vector from the choke point to the player and the forward vector in the equation below:

$$\frac{2 \cdot \vec{v}}{\|\vec{v}\|} + \frac{\vec{f}}{\|\vec{f}\|} \tag{1}$$

Where $\vec{v}$ represents the vector "Chokepoint To Player" and $\vec{f}$ represents the "Player Forward" vector. The norm symbol $\| \cdot \|$ denotes the magnitude of a vector.
This created an effect where the 'net' agent guides the player to the blocker's location by attempting to steer them towards it. If a third and/or fourth agent is available then it will sprint straight at the player to panic them and cause them to make an error. This behaviour relies heavily on the quality of the choke points that have been found. It also struggles to deal with the players ability to fly. Despite this it shows a very high level of complex coordination and has resulted in a fair number of caught chickens.

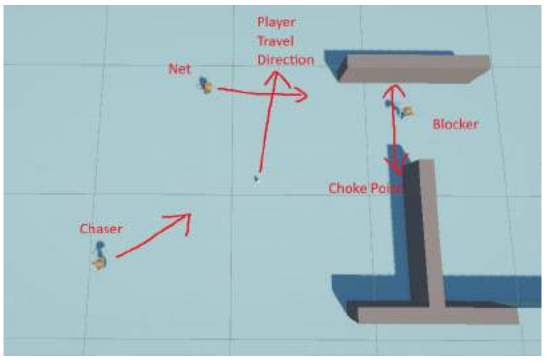


Figure 10: Figure shows Farmers performing a choke, with one blocker, one net and one chaser.

The next behaviour is the chase behaviour. The AI Manager will send the agent a chase order if there is only one agent within 30 meters of the player when spotted or for some of the group behaviours one agent will be designated as the chaser. The chase behaviour is very simple. The agent that has this behaviour will sprint straight towards the player. Farmers in this state will dive at the player when within 2 meters. During the dive animation an event is triggered and if the player is within 1 meter of the farmer when this event occurs the game manager is alerted and the chicken is considered caught. Otherwise the Farmer will get up and enter the tired state. Dogs in the chase state will run at the player. If they touch the player then the player will be concussed, disabling input for 3 seconds. Both the farmers and dogs get tired during the chase state, after 5 seconds of chasing they give up and stand still for a period to catch their breath. This helps with balancing.

The final behaviour is the shoot behaviour (Figure 11) which only the farmers can participate in. This behaviour occurs when the player is not reachable. The player is deemed unreachable if they are above 2 meters or if they are not standing on top of a navmesh. If this occurs the farmers will pull out a shotgun and shoot at the chicken using raycasts. A bullet tracer, lights and sound effects are used to make it seem like an actual projectile has been fired. If the player is not moving then they are perfectly accurate and will hit the chicken every time. If the player is moving at more than 0.3 meters per second then a small random rotation is applied to the vector between the farmer and the player, giving it a chance to miss. On a hit a concussion is applied to the player, with a large force causing a lot of knock back, hopefully moving the player to a position where they can be caught and preventing players from camping.

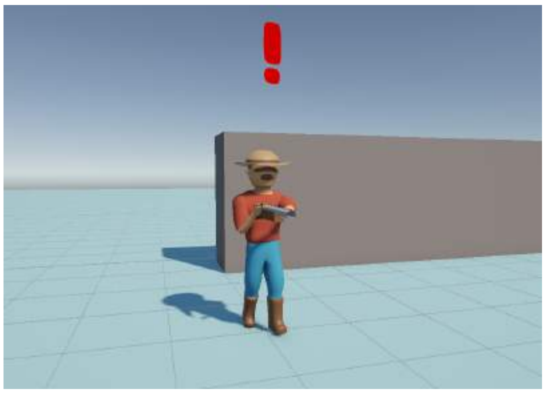


Figure 11: Figure shows a Farmer about to shoot the chicken.

The layered music is also controlled by the player. Each layer is on a different audio source and they all play at the start of the game. Layers volumes are adjusted based on how close or far away this nearest enemy agent is. This adds an auditory component to the 'visualisation' aspect of the AI and helps to add suspense and tension to the game.

8.1.5 Behaviour Trees

Each agent is controlled by a behaviour tree. A behaviour tree is a tree in which each node represents a decision or action that the agent needs to make. Each agent type has its own behaviour tree which controls its actions. The behaviour trees also control the animations of the agents. The Behaviour trees for the farmer and dog follow the same structure: If you have no target or order, ask the AI Manager for a point to wander to (The AI Manager selects a random point within 50 meters of the player so that they never wander too far away); If you have a target and no order, notify the AI Manager and wait for instructions and if you have an order follow its respective branch. Orders can call other orders. The Behaviour trees can access the Detector class to collect information. The tractor does not seek out the player and thus all its tree does is handle navigation between waypoints and also notify the AI Manager if the player has been spotted. The behaviour trees were set up with the order system to allow then to act autonomously or as a group which helps improve the perception the intelligence as the farmers can act individually or as group.
The Behaviour trees were constructed using Unity's Behaviours package. This package contains a basic set of nodes and flow control available to be edited in a graph format. We added our own custom nodes, such as: wander, dive, call investigation etc (Figure 12). These custom behaviours allow our behaviour trees to exhibit custom behaviours.

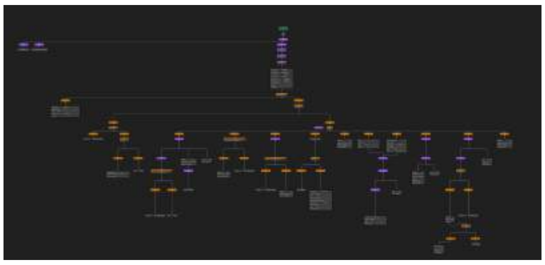


Figure 12: Figure shows the Farmer's behaviour tree.

8.1.6 Balancing

Balancing the AI was essential for the enjoyment of the game. If the AI is too difficult then it will become frustrating for players who are trying to complete the game. If the AI is too easy then there is no threat to the players attempting to complete the game, removing all tension from the session. To help with this problem each AI was given strengths and weaknesses which are designed to be able to complement each other to make the AI a challenge to deal with but not impossible to overcome.
The Dogs are extremely strong at close range. They are faster than the player which means the only way to escape this it to fly or find a barrier. The also have a very wide field of vision and proximity detection so that they

can smell you if you get too close. To help balance this, they have a very short range on their vision and cannot look up, meaning the player only has to fly over them to defeat them.
The Tractor is capable of detecting the player from extremely long distances and can alert the farmers over vast distances. This was balanced by making them patrol a set route at regular intervals allowing players to anticipate their movements. They also have a very narrow vision cone to reduce the power of their sight.
The Farmers are possibly the most difficult to defeat. They are just as fast as the player, have a decent sized vision cone, can call for backup, and have a shotgun to prevent the player from flying out of reach. Pair this with the coordination and multiple farmers can become particularly difficult for the player to deal with. There are a few ways that they were balanced. Firstly they get tired when running so they can't chase you indefinitely. Secondly there is a short amount of time between them diving and catching the player, giving the player a small window to evade the attack. Finally the shotguns are inaccurate if the player is moving, meaning there is a reasonably high chance that they will miss the player and this chance goes up as the distance increases.

8.1.7 Visualisation

An important aspect of the AI was to use visualisations to convey the feeling of intelligence to the player. We used several different methods to showcase the AI and agents thoughts to the player.
We put alert text above each individual agent. This text will display an ! if the agent has spotted the player and a ? if it is searching for the player. It is also integrated with OpenAI allowing the farmers to make in context comments about what is happening. As described in the section 8.2.

We introduced the day-night cycle to also help AI visualisation. During the night, the farmers and tractors turn on their lights. This allows the player to easily visualise their vision cones, helping showcase the detection systems in the game.

The day night cycle uses a time controller script which allows us to specify a start hour and sunrise / sunset hour. Each update() call the script completes adds a second multiplied by a time multiplier, configured to have sunset 3.5 minutes into the gameplay.

Every update() call also updates the angle of our sun and moon directional lights. We first calculate the percentage of day completed by using our sunrise and sunset hours. We then interpolate this percentage to between 0 and 180 to get our angle for the sun's directional light, similarly between 180 and 360 during the night. Our moon's directional light is the same angle plus 180 degrees.

The most important aspect of the cycle was having a smooth transition from day to night. To achieve this, we take the dot product of the forward vector of the sun's directional light and the down vector. The dot product result is between 1, if the light is pointing down, or -1, if pointing up. To have a transition focused during sunset and not linear between midday and midnight, we map the result to a light change curve (Figure 13).

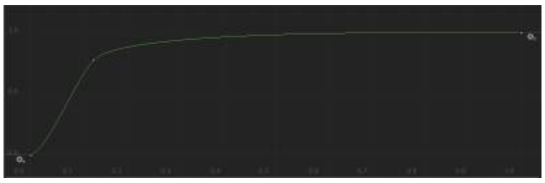


Figure 13: Figure showing the light change curve.

The steep incline in the curve creates a focused transition at sunset. We then interpolate the mapping from the curve to any variables that change between day and night. These include directional light settings, ambient light settings, fog settings, skybox blending and colour tinting.

The introduction of environmental fog meant our sea shader would not blend during night. To solve this we interpolated its colour tint between its daytime colour to the night time fog colour (Figure 14). We also created a script that tells unity our map camera should not render fog, as otherwise the objective map becomes unreadable.

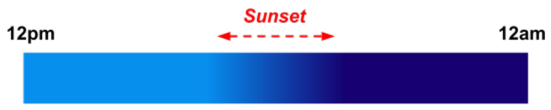


Figure 14: Figure showing the transition of the sea colour tint with time.

Additionally, the time controller sends the game manager a signal for when to turn on the lights. Farmer torches, tractor headlights and street lamps all have a script that listens to the game manager and turn on at the specified time.

Sound effects were also used to help showcase intelligence. The farmers will make different noises depending on what they're thinking. For example after hearing a suspicious noise they make a "huh" sound, and if you are close to a dog and it can smell you it will begin to growl. These audio cues add another sensory element to the AI's intelligence. These sound effects are attached to behaviours and conditions in the behaviour trees, and are activated when they are met.

8.1.8 Visual Audio

One of the areas for improvement after the second panel was to show the player the direction from which they were spotted by the AI. Visual audio was the solution to this, as it provides visual indicators of nearby enemy AIs when the chicken is spotted. Not only did this help the player but it also highlighted the AI's decision-making.

When the chicken is spotted, a transparent UI ring pops up and icons show up around the ring representing the direction and type (farmer, dog or tractor) of nearby enemies relative to the player (Figure 15) and camera forward vector $\mathbf{c}$. The enemy direction $\mathbf{d}$ is calculated by subtracting the player's position $\mathbf{p}_{\text{player}}$ from the enemy's position $\mathbf{p}_{\text{enemy}}$, followed by normalisation to get a unit vector. The angle between the player's camera direction and enemy is then calculated. This angle is then converted into a 2D direction using polar to cartesian conversion (sine and cosine of the angle). For ease of interpretability, if there are multiple enemies coming from the same direction, the position of the icons is moved apart slightly. The enemy direction vector **d**, angle θ and 2D UI position **u** are computed as shown in Equations 2–4:

$$\mathbf{d} = \frac{\mathbf{p}_{\text{enemy}} - \mathbf{p}_{\text{player}}}{|\mathbf{p}_{\text{enemy}} - \mathbf{p}_{\text{player}}|} \tag{2}$$

$$\theta = \text{signed\_angle}(\mathbf{c}, \mathbf{d}, \hat{y}) \tag{3}$$

$$\mathbf{u} = (\sin\theta, \cos\theta) \tag{4}$$

Lastly, each icon is positioned using its direction at a fixed distance from the centre of the ring sprite.

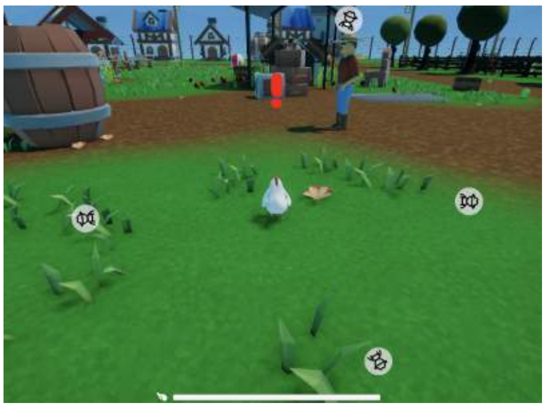


Figure 15: Figure shows the visual audio around the player.

8.2 Open AI Dialogue

8.2.1 System Overview

To enhance the interactivity of the AI, we integrated OpenAI's ChatGPT-4o mini model to generate dialogue for the farmers (Figure 16). This system allows each farmer AI agent to respond to the player's actions and game events, making their decisions feel more spontaneous and lifelike. Their humorous dialogue is on theme with our game's style and our LLM-driven system offers a fairly novel solution to NPC scripting by replacing the traditional list of dialogues written manually with unique and contextual responses.

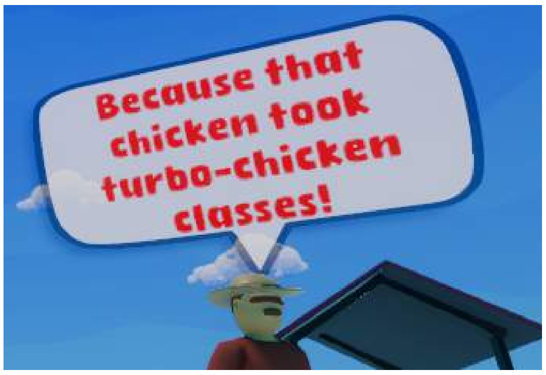


Figure 16: Figure shows a farmer AI responding to another farmer via OpenAI integration.

8.2.2 API Interaction

The OpenAI Manager is responsible for the entire interaction with OpenAI's API, overseeing the process of requesting and processing responses. When dialogue is needed, the script gathers specific context on the game's current situation from the Game Manager and AI Manager, such as whether the chicken has been spotted, if a player has collected a key and calculates additional details such as how long has the farmer been chasing for. To ensure the context remained relevant, a timer was implemented to keep certain context active for only a few seconds, ensuring that context remained relevant to in-game events.

Once context is prepared, a prompt is randomly selected and they are sent to OpenAI's servers via Unity's UnityWebRequest class. The process begins with the creation of an OpenAIRequest object, which is serialised into JSON and sent as part of the HTTP POST request to the API. The system then awaits a response and parses it - removing any unnecessary characters and ensuring the dialogue fits within a required length (5 - 67 characters). If the response doesn't meet these conditions, the request is tried again, up to a maximum of two attempts. If both retries fail, a fallback message is used. The dialogue generation also includes a 50% chance that the farmer responds to previous dialogue, allowing for continuity and making the game feel more immersive.

8.2.3 API Key Security

In order to prevent accidental exposure of the API key, the key is stored in a separate script which is excluded from the github repo via the .gitignore file. This basic security measure reduces the risk of the API key being publicly accessible, keeping credits safe.

8.2.4 Performance Optimisation

To optimise performance, we limited the number of API calls by ensuring only the closest two farmers request dialogue, with a timer in place to ensure a farmer's dialogue continued to be active despite any abrupt changes of farmers swapping closeness. This reduced the number of overall calls being made and the amount of visual clutter on screen. Dialogue was also limited to a manageable length by setting maximum tokens to 18, ensuring the model didn't exceed the desired length.

8.2.5 Future considerations

Whilst developing this we considered using a smaller LLM, such as Meta's LLama, which could be stored locally and wouldn't require API credits. This option would have reduced the reliance on external services and potentially lowered latency, which would which would be useful for a future version of our game if we wanted to publish it with this feature. However, we decided against it for Games Day, due to the limitations of smaller models, trained on less parameters,as they are less capable of generating the high quality dialogue that gpt-4o mini provides. Additionally, local models would be large in size and would have significantly increased our game's download size.

We also explored the possibility of integrating the Model Context Protocol (MCP) to enhance the game's AI interaction. MCP could allow for seamless integration of LLMs with the game environments by enabling AI agents to access full game data (e.g. player stats or map layout) to help them make more informed decisions with more context. While we ran out of time to implement this into the game, it remains a promising area for future development, offering potential for greater AI adaptability and more interactive gameplay.

8.3 Choke Point Detection

Being able to detect choke points was essential for the choke group behaviour to be able to exist. Choke points across the map were detected as follows. Firstly the world is split into 0.5 by 0.5 meter voxels across the ground. For each voxel, 16 pairs of raycasts are projected out in a circle around the centre of the voxel in opposite directions, each one going 4 meters. A pair is considered blocked if both raycasts intersect an object on the ground or default layer. If more than 4 pairs are blocked then the voxel is considered to be choke point and is added to a list of raw chokepoints (Figure 17).

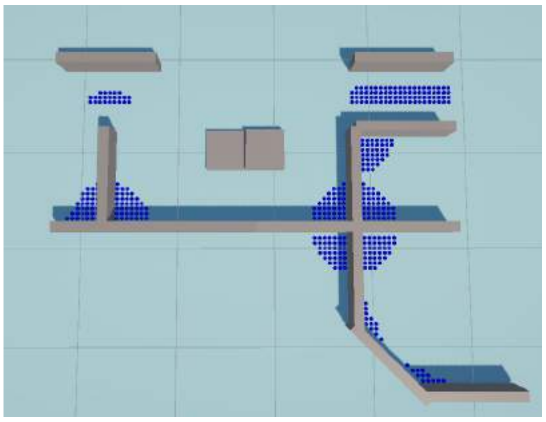


Figure 17: Figure shows the raw choke points as blue spheres calculated on the test map.

The raw choke points are then clustered by taking any raw choke point and if it is within 5 meters of a cluster and has line of sight to it then it is added to that cluster, if not it creates a new cluster and adds itself to the cluster (Figure 18). Once all of the raw choke points have been considered then the mean locations of each cluster is considered to be the final set of choke points.

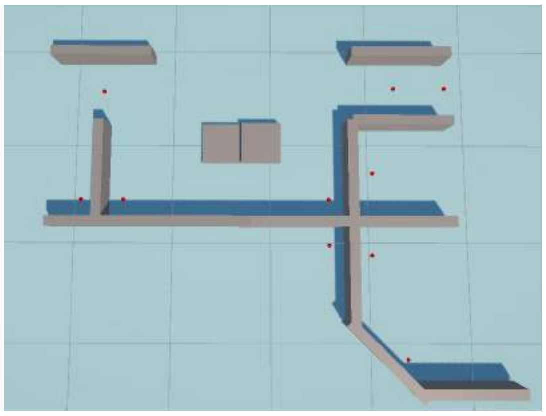


Figure 18: Figure shows the clustered choke points as red spheres calculated on the test map.

It took multiple attempts to create this system. The first attempt used the navmesh data and would place points along its edges, based on the edges length. Due to the way that navmeshes are created this lead to choke points existing in open spaces so this method was abandoned.
Another failed approach was to randomly sample the walkable area in the scene, however due to the indeterminate nature of this it was difficult to test and adjust the constants to work reliably. The only advantage this method has over the chosen one is it is much less computationally expensive, however since this is not calculated at runtime and the result is written into the game files, computational resources were not a concern. This method ultimately lead to the chosen method with voxels.
Clustering was used because it would not be uncommon is tight spaces or corners to have +10 clusters in extremely close proximity. This lead to odd looking behaviour as the blocker agent (see more in AI Manager and Group Behaviours) would always pick a choke point on the edge leading them to jam themselves tightly against surfaces. This looked odd and lead to bizarre pathfinding so instead the points were clustered. This prevented that behaviour.

8.4 General Game Architecture

8.4.1 Assemblies

Assembly definitions were used to separate the code into assemblies, creating barriers between unrelated or loosely related code, as well as speeding up compilation times as only scripts within the same or dependent assembly need to be recompiled on changes. The different assemblies are: Core Gameplay, Enemy Behaviours, Gameplay UI, Main Menu UI, Player, Minor Gameplay and Utilities. These assemblies were chosen as they break the code into logical sections which should not be dependent upon each other (Figure 19). This improved development as it prevented "spaghetti" code, making it much easier for team mates to read each others code and fix bugs.

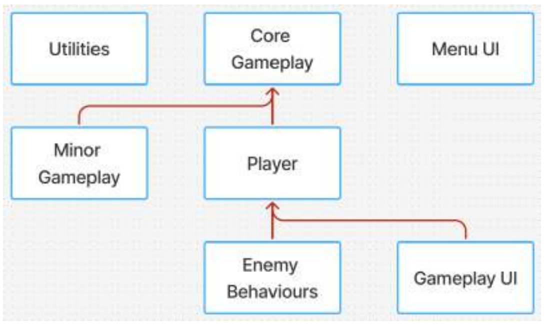


Figure 19: Figure shows the dependencies between the different assemblies. The red arrows point to the assembly that the source is dependent on.

8.4.2 Game Manager and INotify

The Game Manager is a crucial component of the game, which is responsible for managing the game state and coordinating each component of the game. The Game Manager is in the Core Gameplay Assembly. The observer pattern was used to allow the game manager to send events to the components of the game. These events include but aren't limited to: Game Over, Start Cutscene, Show Map, Update UI, Concuss Chicken and so forth. Any components that need to subscribe to the Game Manager and listen for events implements the INotify interface. This allowed the code base to remain modular as it reduced the need for dependencies and massively simplified the coordination of components in the game. This also improved the performance as components no longer need to use polling to collect information and instead listen for events.

8.4.3 Player

The Architecture of the player is fairly simple. First is a player game object which has the following children: the main camera, player character and the cinemachine free look components (Cinemachine is a Unity package which provides advanced camera controllers). The player character is a gameobject that contains the player movement, shoot, cluck and other player scripts. The player character has children which include the mesh and bones of the chicken, its map icon, and the location for the chicken to shoot eggs from. The Player was created like this so that it can be turned into a prefab and drag and dropped into any scene, with all the components it needs already on it. This improved the testing and development workflows.

8.4.4 Gameplay UI

The gameplay UI is a canvas that contains the UI elements of the game during gameplay. It has 7 panels which are: HUD, Pause, Map, Won, Lost, Tutorial and Objectives. The GameplayUI has a script that manages all the panels and has functions that can be used as callbacks when a button is pressed. Only one panel can be active at a time. This is also a prefab like the player and was also created in such a way to allow it to be drag and dropped into any scene which also improved the testing and development workflows.

8.4.5 AI

The AI Manager subscribes to the Game Manager for events. Each agent subscribes to the AI Manager for events. If needed events will propagate from the Game Manager through the AI Manager and to the agents, this was done so that the agents did not need to implement multiple interfaces and subscribe to multiple components to be able to receive all the notifications, simplifying the code base. This system means that as long as an AI Manager and agents are in the scene they are capable of running independently of every other element of the game, which also improved the testing and development.

8.5 Performance, Profiling and Optimisation

8.5.1 Performance

The performance of a game significantly effects a players enjoyment of the game. In extreme cases it can effect the playability and the way players play, potentially putting them at an advantage or disadvantage [8].
The concept of frame times will be important for this section. A brief overview is that a frame time is the time it takes for a single frame of the game to be executed. This includes all the code that needs execution and all rendering on both the CPU and GPU. For reference 16.6ms is 60 frames per second and is an industry standard target for smooth gaming. 33.3ms is 30 frames per second and is considered to be the lowest you should go [9] as the performance is not consistent and can spike for various reasons which will lead to major stuttering, breaking immersion. Higher frame rates are good as they improve the visual quality of the game. Movements appear smoother during gameplay and it also reduces the effect of screen tearing. Screen tearing occurs when the frame rate of the game and frequency of the display are not aligned and the display is halfway through drawing a frame to the screen when a new frame is drawn to the buffer it is reading from by the GPU. [8]
Throughout development, performance was never a large issue so we did not spend a lot of time optimising the game. If we needed to optimise, it is likely that to optimise the CPU, we would have utilised the jobs system for parallelisation and the burst compiler to remove the majority of C#'s overhead. This would be at the cost of memory safety and ease of implementing new features but has the capability to improve performance by several orders of magnitude. For the GPU implementing a Level Of Detail system to reduce the number of vertices would provide massive benefit, as well as combining meshes to reduce the number of batches and draw calls, which would also help reduce overhead. To reduce VRAM usage, compressing textures would provide large benefit as currently they are using 1.52 GB of VRAM on the GPU which is significantly more than it needs to maintain visual fidelity. To optimise RAM usage, music audio files

can be loaded from storage as needed and removed after use instead of all in one go on start. This adds a little CPU overhead and potential lag so it is not recommended to do this for sound effects. Reducing triangle count and compressing images and would also help.
The average specs of a player's machine matter as well. The Steam Hardware & Software Survey [10] has information the percentage of players that meet a specific specification for example processor count, processor speed and memory. This information in conjunction with profiling can be used to determine where the most impactful optimisations can be made.

8.5.2 Profiling

Profiling was done on a Windows 11 PC that has a Ryzen 7800x3D CPU, RTX 4070 Super GPU and 32GB of RAM. Analysing the performance through the profiler shows for each frame, 20.601ms were spent processing on the CPU and only 13.254ms were spent on the GPU during execution. This means that to improve the frame rate, the CPU time has to be reduced through optimisations. Further analysis revealed that on the CPU the main thread is the bottle neck and that the rendering thread and jobs thread are idle the vast majority of the time. This means that the majority of the performance gains can be achieved through further optimising execution on the main thread. There is also an immediate opportunity for optimisation by shifting anything that doesn't need to be executed on the main thread to the jobs thread, significantly reducing the CPU time.
Of the 20.601ms, 0.6ms were spent on functions in fixed update, 2.14ms on Update, 0.53ms on late update, 14.31ms on the render loop and the majority of the rest (2.23 ms) was spent on the editor loop. Some important notes about these results: The editor loop only exists when running in the editor so there will be an immediate small performance boost when the game is built; Profiling adds a reasonable amount of CPU overhead. When the profiler is deactivated the CPU frame time is closer to 10ms and when activated it jumped to 20ms. Between these it is plausible that in a built version of the game the GPU will be the bottle neck and not the CPU.

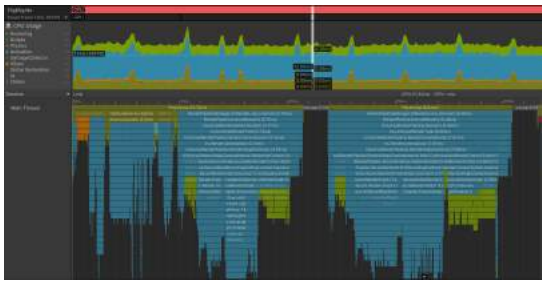


Figure 20: Figure shows a screenshot of the functions the CPU ran on the main thread during a single frame and the time it took to execute each one.

The game allocates 5.64GB of memory and used 3.79GB of memory. Allocated memory is not constrained to the exact amount of RAM a system has due to paging and other memory management techniques. Regardless both of these values are significantly below the median RAM of 16GB so this will not be a bottleneck. [10]
At the moment there are 15.7 million vertices in the scene. Reducing this count would greatly improve the GPU performance. To optimise the VRAM usage, using smaller textures will provide the largest benefit as currently textures are taking up 1.52GB of VRAM compared to the 400MB of meshes and 3.5MB of materials. The VRAM usage is still considerably lower than the median of 6GB. [10]

8.5.3 CPU Optimisation

Each agent has a detector class and each detector class runs the proximity and visual scan functions. Originally this would run every frame, which was quite computationally expensive. We changed this so that instead it would run every 1/10th of a second, which still gave the appearance of instant detection of the player, whilst cutting down the number of time it executes dramatically. The exact performance saving depends on the machine used as computers with higher single core performance will be able to achieve higher frame rates and thus more benefit from this. However as an example if a machine is running the game at 60 frames per second, this would reduce the number of times the scan is executed from 60 to 6 times a second. When executed on the same machine used for profiling, this reduced the time that the detector scripts spent executing on the main thread from 0.98ms down to 0.01ms, representing a 99% decrease.
Another major optimisation comes from the use of the Observer pattern and events. This reduced or eliminated the need for polling which considerably improves the performance of the game.

8.5.4 GPU Optimisation

To improve the performance of rendering we made use of a few easy to implement features of unity. Firstly is occlusion culling, in which every object that is not contained inside the frustum of the active camera is not send to be rendered as it cannot be seen. This reduces the amount of data that needs to be passed to the GPU from the CPU, saving a lot of time.
Static objects were marked as static. This allows unity to optimise by using batching. Batching is when several meshes are merged together to reduce the number of draw calls, reducing the amount of overhead when rendering.

8.6 User Experience

8.6.1 Tutorial area

After some user testing it was suggested that we make a tutorial to help the players get used to the controls of the game. We decided to put the player in a locked chicken coop and walk them through the controls at the start of the game. Each control is displayed on the screen until the player performs the action. Once complete it moves onto the next action. This goes on until the end of the tutorial where the player is prompted to look at the back of the coop. On the wall there are images with information and instructions about the farm, such as a map, enemy types, weaknesses, and information on the keys. The tutorial can be skipped with tab for the more experienced players. The inclusion of a tutorial massively improved the onboarding experience and allowed for players to quickly pick up the controls improving their skill level and the enjoyment of the game as a whole.

8.6.2 Scoring System

Originally the scoring system was purely based on time. It encouraged the players to speed run the game to get the best score possible. However we determined that this system was not as fun as it could be since it didn't allow players the option to explore and find as many keys as possible. It also heavily favoured those who are already familiar with playing games. Finally is reduced the replayability of the game as once complete it provided little incentive for the player to try something new to improve their score. To alleviate these issues, the system was altered so that the players would be scored on how many keys they collected as well as their time and number of deaths (see more on respawning in Multiple Lives). This allowed players to play in multiple ways. They could speed run and collect points for achieving a fast time or they can explore and get points for finding lots of the keys. This allows for player of all skills to play and enjoy the game. It also improves the replayability as there are now multiple ways to play.

8.6.3 Multiple lives

We found that the original instant death system was too harsh. It was annoying for both experienced and inexperienced players as it would take a single mistake to end a run or if you weren't particularly good it became almost impossible to complete the game. To remedy this we added respawning. This was integrated with the new score system by taking away 100 points per extra life you needed. Respawns are disabled in the final sequence of the game, after the electric box is disabled, to add an extra element of danger and risk-reward for those tempted to continue exploring. This massively improves the gameplay experience as now getting unlucky will not end a run, reducing the frustration caused by that. It also assists new players in being able to beat the game as they no longer have to collect all the keys and escape in one life.

8.6.4 Objective Panel

Through user testing, we found that it is important that the player is aware of their relative location. To provide this, we built a panel that would show the current objective and a live map. The live map included the location of the keys, tractor locations and the player's direction. The location of the electric box and gate appear when their respective objective is active. This was achieved by implementing an orthogonal camera overlooking the map (Figure 21). Icons are displayed in the sky in the form of sprites and using layers would only be rendered by the map camera. The camera view was then output to a render texture in the objective panel.

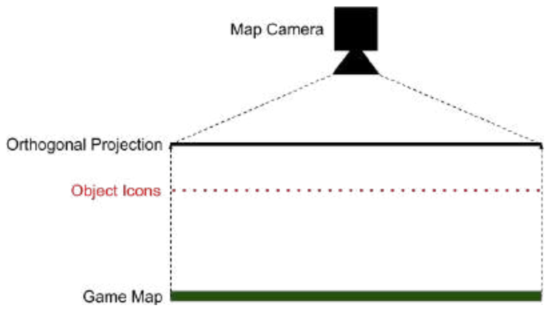


Figure 21: 2D Diagram of the projection of the game map onto the map camera.

8.6.5 Objectives UI Popup

Following feedback from the second testathon, it was clear to that players needed more guidance on when they had completed specific objectives. To address this we implemented a UI screen that pops up for 5 seconds when a player reaches a milestone, such as being able to open the electric box after collecting enough keys or once the cutscene ends and they can escape the farm. This popup helps the player understand their progress in-game, without having to interrupt gameplay to ask us what to do next.

8.7 Animations, Graphics and Sound

8.7.1 Models and Assets

We wanted our game to have a fun unique feel. We decided on using low poly cartoon style graphics as it fit the theme nicely and it is relatively easy to achieve. We wanted some assets to be custom made but also acknowledged due to the time constraints that it would not be possible for us to make custom assets for every part of the game. We focused on creating assets for the most pivotal aspects of the game. These were: The chicken, Farmer, Dog, Coop and Electric box (Figure 22). Every other model was acquired off the Unity asset store through various asset packs.

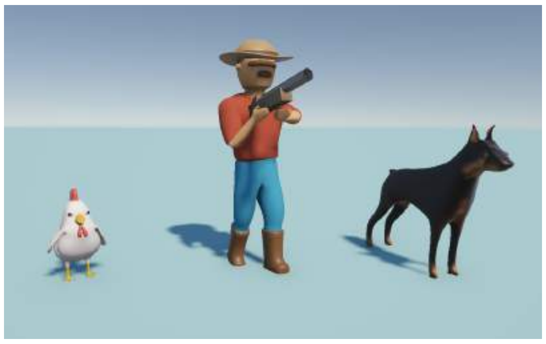


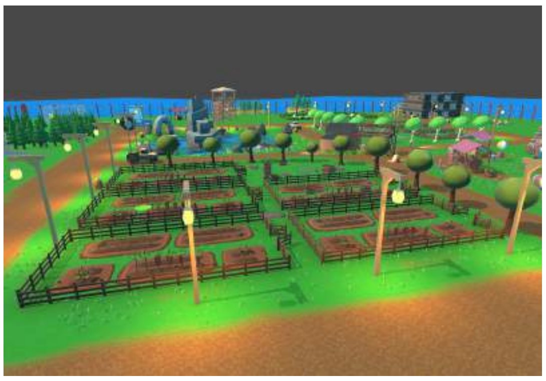
Figure 22: Figure shows the chicken (left), the farmer (middle) and the dog (right) models.

8.7.2 Lighting, Reflections and Shadows

Lighting is essential for altering the look and theme of a game. It can also have a major impact on the performance of the game. The colour and intensity of the lighting changes throughout the day and night cycle. We decided to use a mix of real-time and baked lighting to uphold visual fidelity whilst balancing the performance. All static objects in the scene have baked lighting, meaning that their shadows are stored in a shadow map. The shadow map had its maximum size set to 2048 x 2048 to limit the amount of memory that it would take to store it in VRAM. This baked lighting is combined with real time lighting from the day and night cycle to create the lighting for the scene.
The lighting settings were altered to disable real-time global illumination, which adds more realistic lighting at the cost of performance, and enable ambient occlusion which adds shadows to surfaces which are near another object, adding to the cartoon style look. The lightmaps were limited to 1024 x 1024 for similar reasons to the shadowmaps.
In interior spaces reflection probes where utilised to improve the accuracy of the lighting. By default it uses the sky box for approximate reflections which was undesirable as the reflections were extremely bright compared to the darker interiors. The reflection probes were baked to improve performance at the cost of VRAM.

8.7.3 Post Processing

Post Processing was used to further improve the visual quality of the game. Firstly an ACES Tonemapper was used to adjust the brightness and contrast of the image to preserve details in the image, which especially helps in the transition between interior and exterior spaces. Next Bloom was added. This adds blur to exceptionally bright areas of the image and can make objects like the keys "glow" without using emissive materials. Finally a subtle vignette was added to direct the players attention away from the edges of the screen. The effect of these effects combined is subtle but contributes to a more natural and visually pleasing gaming experience (Figure 23).



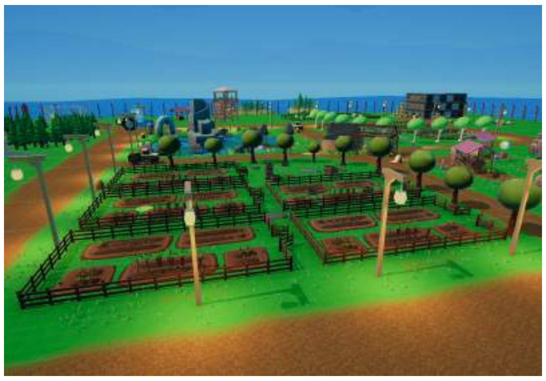


Figure 23: Figure shows the scene with post processing disabled (Top) and then enabled (Bottom).

8.8 Music and Sound Effects

The music for the project was created by composers from the Music Department. We worked closely with them over the course of the project to ensure that the produced music fit the theme and worked well with the game mechanics. One such example is having the music play in layers, with different layers active depending on how far from the player the AI agents were. In the end we received music for the Main Menu, gameplay, tutorial and final chase sequence.
For the sound effects, these were downloaded from Pixabay's free sound effects library [11]. Similar to the models we would have loved to make our own but time constraints forced us to prioritise on the rest of the content in the game.

References

[1] J. S. J. Hocking, *Unity in Action: multiplatform game development in C#*, third edition ed. Manning Publications Co, 2022.

[2] T. Autonomous. (2023, February) Exactly how the hungarian algorithm works. Accessed: 2025-04-25. [Online]. Available: https://www.thinkautonomous.ai/blog/hungarian-algorithm/

[3] B. Driessen, “A successful git branching model.” [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/

[4] F. Fugative, “Feathered fugative documentation.” [Online]. Available: https://ed22699.github.io/FeatheredFugitivesDocs/

[5] Nintendo, “Super mario bros.” Video Game, Japan, 1985, released for the Nintendo Entertainment System (NES).

[6] T. Tovmasyan, “Ai and pac-man: A story of ghosts’ intelligence,” https://tateviktome-tovmasyan.medium.com/ai-and-pacman-a-story-of-ghosts-intelligence-d2f296c31675, 2021, accessed: March 20, 2025.

[7] R. M. K. Jack Edmonds, “Theoretical improvements in algorithmic efficiency for network flow problems,” https://dl.acm.org/doi/10.1145/321694.321699, 1972, accessed: April 12, 2025.

[8] T. Tamasi, “Why does high fps matter for esports?” https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/, 2019, accessed: April 11, 2025.

[9] I. A. F. Rate, “Information about frame rate,” https://www.logicalincrements.com/articles/framerate, 2023, accessed: April 11, 2025.

[10] S. H. . S. S. M. 2025, “Steam hardware software survey: March 2025,” https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam, 2023, accessed: April 11, 2025.

[11] Pixabay, “Royalty-free sound effects for download,” https://pixabay.com/sound-effects/.